

操作系统上机实验

Lab 3：用户环境

实验报告

December 2, 2019

曹元议 1711425
黄艺璇 1711429
王雨奇 1711299
张瑞宁 1711302
阔坤柔 1611381
冯晓妍 1513408

Abstract

完成了 Lab 1 和 Lab 2 的实验以后。我们将在前两个实验环境的基础上进一步探究操作系统的用户环境。这里的“环境”可以简单理解为“进程”。本次实验主要分为两个部分：第一部分（Part A）是用户环境与异常处理，在这一部分将会实现环境数组的分配、创建并运行环境，并使 JOS 内核具有最基本的异常处理能力。第二部分（Part B）是缺页中断、断点异常、系统调用，这一部分是对第一部分的异常处理功能的细化。

Keywords: 操作系统；QEMU；JOS；用户环境；进程；异常；中断；系统调用

Contents

1 小组分工情况	1
2 用户环境与异常处理	1
2.1 作业 1	4
2.2 作业 2	7
2.3 练习 1	29
2.4 控制权转移	34
2.5 中断和异常的嵌套	36
2.6 作业 3	36
2.7 Challenge!	46
2.8 问题 1	51
3 缺页中断、断点异常、系统调用	53
3.1 背景知识	53
3.2 作业 4	56
3.3 作业 5	58
3.4 问题 2	59
3.5 作业 6	60
3.6 作业 7	64
3.7 作业 8	67
3.8 作业 9	76
4 总结	77

1 小组分工情况

	问题解决	报告整理	报告检查	小组讨论	Lab 3 任务
曹元议	√	√		√	作业 1、作业 2、Challenge
黄艺璇	√		√	√	作业 5、问题 2
王雨奇	√		√	√	作业 3、问题 1
张瑞宁	√		√	√	练习 1、作业 4
阔坤柔	√		√	√	作业 8、作业 9
冯晓妍	√		√	√	作业 6、作业 7

Table 1: 小组分工

曹元议	黄艺璇	王雨奇	张瑞宁	阔坤柔	冯晓妍
1/6	1/6	1/6	1/6	1/6	1/6

Table 2: 贡献比

2 用户环境与异常处理

由于这里的“环境”的意思等同于“进程”，因此下文的任何地方都可能会混用这两个词。Lab 2 已涉及到的宏定义不再过多解释。

在本次实验中，我们仍使用 Lab 1 和 Lab 2 配置好的实验环境。根据实验指导取得 Lab 3 的代码，并把分支切换到 Lab 3 的分支。

我们取得了这些新的的实验文件：

目录	文件	说明
inc/	env.h	用户模式环境的公有定义
	trap.h	trap 处理的公有定义
	syscall.h	用户环境向内核发起系统调用的公有定义
	lib.h	用户模式支持库的公有定义
kern/	env.h	用户模式环境的内核私有定义
	env.c	用户模式环境的内核代码实现
	trap.h	trap 处理的内核私有定义
	trapentry.S	trap 处理函数入口点的汇编代码
	syscall.h	系统调用处理的内核私有定义
	syscall.c	与系统调用实现有关的代码
	Makefrag	构建用户模式调用库的 Makefile fragment, obj/lib/libuser.a
lib/	entry.S	用户进程入口点的汇编代码
	libmain.c	从 entry.S 进入用户模式的库调用
	syscall.c	用户模式下的系统调用桩(占位) 函数
	console.c	用户模式下 putchar() 和 getchar() 的实现, 提供控制台输入输出
	exit.c	用户模式下 exit() 的实现
	panic.c	用户模式下 panic() 的实现
user/	*	检查 Lab 3 内核代码的各种测试程序

Table 3: Lab 3 的新文件

上面出现了很多同名却不同路径的文件, 它们的区别在于“公有”和“私有”。这里所说的公有指的是内核和用户都可见的, 私有是只有内核可见的。

其中, 有一个比较重要的文件是kern/env.c, 有3个结构体Env指针类型的全局变量:

```
struct Env *envs = NULL;           // All environments
struct Env *curenv = NULL;          // The current env
static struct Env *env_free_list;   // Free environment list
                                    // (linked by Env->env_link)
```

Figure 1: kern/env.c

- **envs:** 指向所有环境的数组, 长度为NENV。
- **curenv:** 指向当前正在运行的环境。
- **env_free_list:** 空闲环境链表, 它是指向链表中首元素的指针。

另一个重要的文件是`inc/env.h`。上面提到的结构体`Env`是在这里定义的，它用来保存每个用户环境（进程）的关键数据。定义如下：

```
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_pgd;                  // Kernel virtual address of page dir
};
```

Figure 2: inc/env.h

- `env_tf`: 类型为结构体`Trapframe`，在`inc/trap.h`中定义。当内核或者其他用户环境在运行时，它存储当环境没有在运行时被保存下来的寄存器的值，以便环境可以从它被暂停的时间点恢复过来。
- `env_link`: 指向下一个位于`env_free_list`中的`Env`结构体的指针。
- `env_id`: 当前使用该`Env`结构体变量的用户环境标识符。每个不同的用户环境都会有一个不同的`env_id`，即使新的用户环境和旧的使用的是同一个`Env`结构体变量。
- `env_parent_id`: 创建这个环境的环境（即父进程）的`env_id`。
- `env_type`: 环境的类型，通常是`ENV_TYPE_USER`（用户进程）。
- `env_status`: 环境所处的状态，在Lab 3中会用到这些值：
`ENV_FREE`: 这个`Env`结构未被利用，因此在`env_free_list`中。
`ENV_RUNNABLE`: 表示该环境处在就绪态，等待被CPU调度。
`ENV_RUNNING`: 表示目前正在运行的环境。
`ENV_NOT_RUNNABLE`: 表示目前已经激活的环境，但是它还没有准备好运行，处在阻塞态。
- `env_runs`: 环境被执行的次数。
- `env_pgd`: 保存这个环境页目录的虚拟地址。

2.1 作业 1

2.1.1 题目原文

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.

2.1.2 题目大意

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

修改好后，`check_kern_pgdir()` 应该能够成功执行。

2.1.3 回答

此题较简单，只需使用 Lab 2 中的函数即可。只需修改两个地方：

2.1.4 第一处修改

先要为上述提到的 `env` 数组分配空间并对其初始化。修改的思路同 Lab 2：

```
//////////  
// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.  
// The kernel uses this array to keep track of physical pages: for  
// each physical page, there is a corresponding struct PageInfo in this  
// array. 'npages' is the number of physical pages in memory. Use memset  
// to initialize all fields of each struct PageInfo to 0.  
// Your code goes here:  
pages = (struct PageInfo*)boot_alloc(npages * sizeof(struct PageInfo));  
memset(pages, 0, npages * sizeof(struct PageInfo));
```

Figure 3: Lab 2 中 `mem_init()` 的为 `pages` 数组分配并初始化空间

其中，`boot_alloc()` 函数是 Lab 2 中修改的，用于数组空间的分配；`memset()` 函数是 C 库函数，用于被分配空间的初始化。`envs` 数组所占空间的大小为 `NENV * sizeof(struct Env)`。可仿照修改成如下代码：

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

Figure 4: Lab 3 中 mem_init() 的第一处修改

2.1.5 第二处修改

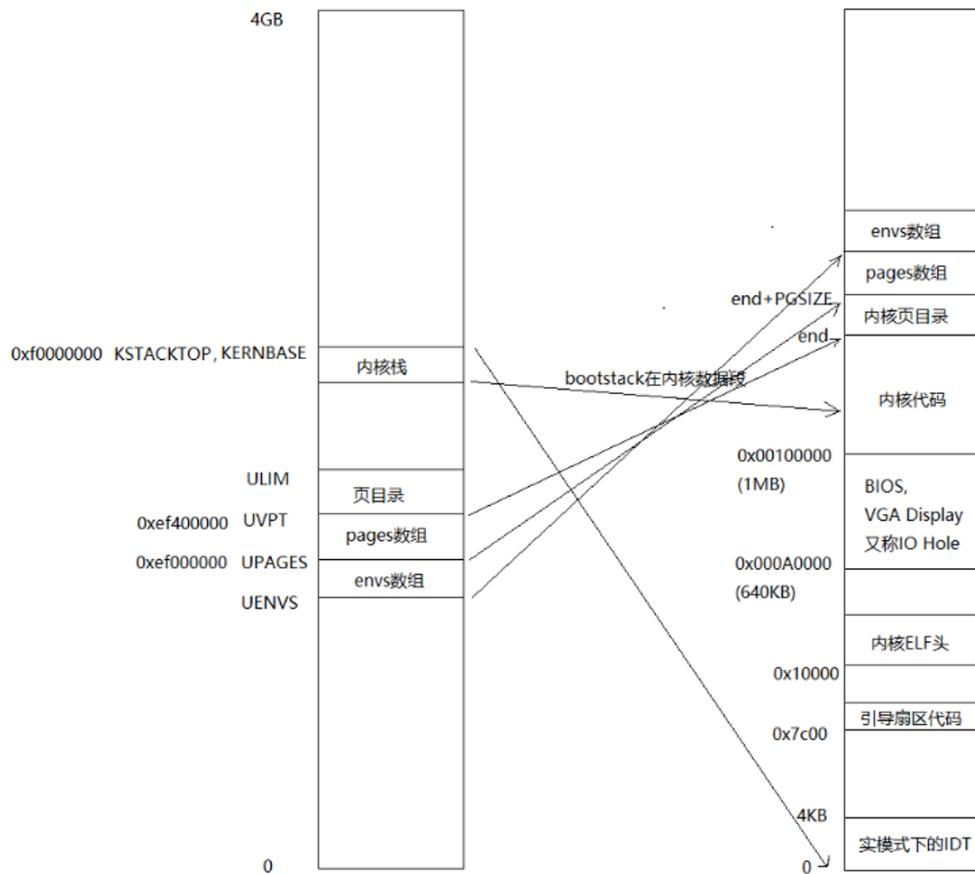
需要将虚拟内存的UENVS段映射到envs数组首元的物理地址上，并将映射关系加入到内核页目录中。由inc/memlayout.h可知，UENVS对用户和内核的权限都是只读，因此需要置权限位 U（用户可使用）和 P（有效页面）有效，W（页面可写）无效。Lab 2 的函数boot_map_region()可以实现虚拟地址空间[va, va+size)到物理地址空间[pa, pa+size)的映射，并将这个映射关系加入到页目录中。

```
//////////  
// 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, (uintptr_t) UENVS, ROUNDUP(NENV * sizeof(struct Env), PGSIZE), PADDR(envs), PTE_U | PTE_P);
```

Figure 5: Lab 3 中 mem_init() 的第二处修改

其中，第一个参数kern_pgdir表示页目录（kern/pmap.c的全局变量，内核进程页目录），第二个参数UENVS表示要映射到的虚拟地址空间的段，第三个参数表示要映射的地址空间的大小（由于这里的映射是以页为单位的，因此需要将数组实际占用的空间 12 位对齐），第四个参数是envs数组所对应的物理地址，最后一个参数PTE_U | PTE_P是要设置的虚拟地址空间的权限，表示权限位 U 和 P 都有效。

这样，执行完mem_init()函数的效果应如下图所示：

**Figure 6:** `mem_init()` 函数的执行效果

经过上述的两个修改后，`check_kern_pgdir()`的成功报告如下：

```
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
map region size = 4194304, 1024 pages
map region size = 98304, 24 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
kernel panic at kern/env.c:460: env_run not yet implemented
```

Figure 7: `check_kern_pgdir()` 的成功报告

2.2 作业 2

2.2.1 题目原文

```
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"
```

2.2.2 题目大意

在 kern/env.c 中，完成接下来的这些函数：

- **env_init()**: 初始化全部 envs 数组中的 Env 结构体，并将它们加入到 env_free_list 中。还要调用 env_init_percpu，这个函数会通过配置段式内存管理系统，让它所管理的段，可能具有两种访问优先级其中的一种，一个是内核运行时的 0 优先级，以及用户运行时的 3 优先级。
- **env_setup_vm()**: 为新的用户进程分配一个页目录，并初始化新用户进程的地址空间对应的内核部分。
- **region_alloc()**: 为进程分配和映射物理内存。
- **load_icode()**: 处理 ELF 二进制映像，类似引导加载程序 (boot loader) 做好的那样，并将映像内容读入新进程的用户地址空间。

- `env_create()`: 通过调用 `env_alloc` 分配一个新进程，并调用 `load_icode` 读入 ELF 二进制映像。

- `env_run()`: 在用户模式下，启动给定的在用户模式运行的进程。

`cprintf` 的格式控制符`%e`可以打印出与错误代码相对应的描述，例如：

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

会 `panic` 并打印出 `env_alloc: out of memory`。

2.2.3 回答

需要修改的函数均在 `kern/env.c` 中。

2.2.4 `env_init()`

首先是 `env_init()` 函数，下面是这个函数的注释：

```
// 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]).
// ...
```

Figure 8: `env_init()` 函数的注释

由上面的注释可知，在初始化 `envs` 数组时，需要先将数组中所有元素的 `env_id` 都设为 0，状态 (`env_status`) 都标为空闲 (`ENV_FREE`)，并将数组中所有元素都插入到链表 `env_free_list` 中。必须要保证 `env_free_list` 中的元素排列顺序和它们在数组中的顺序一致，这样就可以确保最后 `env_free_list` 每次指向的是数组下标最小的元素。

因此需要遍历整个 `envs` 数组，在遍历数组的同时将该元素插入 `env_free_list` 中。为了保证链表中和数组中的元素顺序一致，从理论上来说，可以采用下面的两种方法：

1. 正向（按下标从小到大）遍历数组，在链表尾部插入新元素。
2. 反向（按下标从大到小）遍历数组，在链表头部插入新元素。

由于 `env_free_list` 是指向链表头部元素的指针，假设数组大小为 n 。如果在尾部插入，需要在遍历数组的时候遍历整个链表 n 次，时间复杂度为 $O(n^2)$ ；如果在头部插入，则在遍历数组的时候不需要遍历链表，链表的插入只需要常数时间，时间复杂度为 $O(n)$ 。由于在操作系统中需要保证每个操作的时间复杂度尽可能低，因此采用第 2 种方法。

因此应补齐的代码为：

```

1 void
2 env_init(void)
3 {
4     // Set up envs array
5     // LAB 3: Your code here.
6     int i;
7     env_free_list = NULL;
8     for(i = NENV - 1; i >= 0; i--)
9     {
10         envs[i].env_id = 0;
11         envs[i].env_parent_id = 0;
12         envs[i].env_type = ENV_TYPE_USER; //env_type 通常为 ENV_TYPE_USER
13         envs[i].env_status = ENV_FREE;
14         envs[i].env_runs = 0;
15         envs[i].env_link = env_free_list;
16         env_free_list = &envs[i];
17     }
18     // Per-CPU part of the initialization
19     env_init_percpu();
20 }
```

2.2.5 env_setup_vm()

接下来是函数`env_setup_vm()`, 下面是这个函数的注释:

```

//
// 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.
//
```

Figure 9: `env_setup_vm()` 函数的注释

由注释可知, 该函数的作用就是初始化参数`e`指定进程的内核地址空间, 具体的做法是: 先分配并设置`e`的页目录 (其实相当于这个进程的地址空间), 然后初始化`e`的内核地址空间。不需要映射`e`的用户地址空间。

在函数体中还给了一个关于设置进程页目录的提示:

```

// 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.

```

Figure 10: env_setup_vm() 函数内部的提示

提示的意思是说，根据`inc/memlayout.h`，UTOP之上属于内核地址空间，之下是用户地址空间。事实上UTOP之上的地址映射（即内核地址空间）该设置的都已经设置好了，在这里除了UVPT这一块的地址映射需要单独设置。可以直接将`kern_pgdir`指向的页目录的拷贝给`e->env_pgdir`。此时用户地址空间为空（即目前不需要作任何映射）。在这里也不需要调用函数`page_alloc()`。在通常情况下，UTOP以上的虚拟地址空间的映射是不维护`pp_ref`的值的，但是`env_pgdir`是个特例，在这里必须将`pp_ref`的值加1才能使`env_free()`函数正常工作。

观察`kern/pmap.c`和`inc/memlayout.h`可知，是`kern/pmap.c`中的`mem_init()`函数作了内核地址空间的映射。这个映射工作主要是在 Lab 2 的作业 5 和 Lab 3 的作业 1 中使用函数`boot_map_region()`进行的（还有单独对UVPT的映射），而不是通过函数`page_insert()`，这就是静态映射。这两个函数的区别就是，前者是直接用物理地址去映射，在调用前这个物理地址对应的物理页（即在空闲物理页链表`page_free_list`中）可能还没有被分配；后者是直接用已经实际分配了的物理页（即用这个物理页对应的 `PageInfo`，且这个 `PageInfo`并不在空闲物理页链表`page_free_list`中，为什么不是空闲物理页的原因稍后作解释）去映射。由`inc/memlayout.h`可知，[UTOP, 2^32)的虚拟地址空间都已经被映射（图略）。

特别地，除了使用`boot_map_region()`的映射以外，`mem_init()`单独对UVPT段的映射如下图所示：

```

///////////
// Recursively insert PD in itself as a page table, to form
// a virtual page table at virtual address UVPT.
// (For now, you don't have understand the greater purpose of the
// following line.)

// Permissions: kernel R, user R
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;

```

Figure 11: mem_init() 单独对 UVPT 段的映射

也就是将内核页目录的起始地址单独映射给UVPT段，因此可知这个段对应的是内

核进程的页目录。提示还说在`env_setup_vm()`的底部单独设置好了UVPT段的映射，如下图：

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

Figure 12: `env_setup_vm()` 单独对 UVPT 段的映射

也就是将`e`的页目录映射给UVPT段。因此可知，UVPT段对应的是一个进程的页目录：如果是内核进程，那么映射的就是内核页目录；如果是用户进程，那么映射的就是用户页目录。

由上述可知，对于 JOS 中所有的进程来说，内核地址空间的映射（除了UVPT段）是一模一样的。

还有，`mem_init()`中，`kern_pgd`空间的分配和初始化如下：

```
///////////////////////////////
// create initial page directory.
kern_pgd = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgd, 0, PGSIZE);
```

Figure 13: `mem_init()` 中 `kern_pgd` 的空间分配和初始化

从这里也可以知道，由于在`mem_init()`中被映射了的虚拟地址空间都会加入到`kern_pgd`中，而这个页目录的空间被初始化为 0。因此没被映射的虚拟地址空间，对应的`kern_pgd`的项为 0，即为空。

因此就可以按照上述的提示，直接将`kern_pgd`的页目录拷贝给`e->env_pgd`。

最终，函数`env_setup_vm()`的修改如下：

```
1 static int
2 env_setup_vm(struct Env *e)
3 {
4     int i;
5     struct PageInfo *p = NULL;
6
7     // Allocate a page for the page directory
8     if (!(p = page_alloc(ALLOC_ZERO)))
9         return -E_NO_MEM;
10
11    // Now, set e->env_pgd and initialize the page directory.
12    //
13    // Hint:
14    //   - The VA space of all envs is identical above UTOP
15    //   (except at UVPT, which we've set below).
```

```
16 // See inc/memlayout.h for permissions and layout.
17 // Can you use kern_pgdir as a template? Hint: Yes.
18 // (Make sure you got the permissions right in Lab 2.)
19 //   - The initial VA below UTOP is empty.
20 //   - You do not need to make any more calls to page_alloc.
21 //   - Note: In general, pp_ref is not maintained for
22 // physical pages mapped only above UTOP, but env_pgdir
23 // is an exception -- you need to increment env_pgdir's
24 // pp_ref for env_free to work correctly.
25 //   - The functions in kern/pmap.h are handy.
26
27 // LAB 3: Your code here.
28 e->env_pgdir = (pde_t*)page2kva(p); //上面新分配的物理页作为e的页目录
29 p->pp_ref++; //由于该物理页将被映射，故引用数加1
30 memmove(e->env_pgdir, kern_pgdir, PGSIZE); //将kern_pgdir拷贝给e->
31     env_pgdir
32
33 // UVPT maps the env's own page table read-only.
34 // Permissions: kernel R, user R
35 e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
36
37 }
```

这里`memmove()`函数是C的库函数,它的作用是从地址`kern_pgdir`开始拷贝`PGSIZE`个Byte 的内容给`e->env_pgdir`, 即把内核页目录拷贝给`e->env_pgdir`, 而一个页目录也是页, 大小为4KB。注意这里不需要设置对用户地址空间的映射。

2.2.6 region_alloc()

接下来是函数`region_alloc()`。下面是该函数的注释:

```

// 
// 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!)
}

```

Figure 14: region_alloc() 的注释

由注释可知，该函数的作用是分配由参数len指定字节的物理内存空间给进程e，然后将这个物理空间映射到给定的e的虚拟地址上。不要对这个物理空间作任何的初始化。这个页的权限是对用户和内核均可写，因此权限位应为W、P、U均有效。当中如果有任何操作出错，就panic并返回。

根据函数内部的提示也可知，为了页对齐，需要对虚拟地址va向下取整到PGSIZE的倍数，对va+len向上取整到PGSIZE的倍数。这样就可以保证分配的空间至少为一个页的大小。

因此函数修改如下：

```

1 static void
2 region_alloc(struct Env *e, void *va, size_t len)
3 {
4     // LAB 3: Your code here.
5     // (But only if you need it for load_icode.)
6     //
7     // Hint: It is easier to use region_alloc if the caller can pass
8     //       'va' and 'len' values that are not page-aligned.
9     //       You should round va down, and round (va + len) up.
10    //       (Watch out for corner-cases!)
11    void* start = (void *)ROUNDDOWN((uint32_t)va, PGSIZE); // 对va向下取整
12    void* end = (void *)ROUNDUP((uint32_t)va+len, PGSIZE); // 对va+len向上取整
13    struct PageInfo *p = NULL;
14    void* i;
15    int ret;
16    for(i=start; i<end; i+=PGSIZE) {
17        p = page_alloc(0); // 分配一个新的物理页，参数为0表示不要初始化此物理页
18        if(p == NULL)
19            panic("region_alloc error: physical page allocation failed!\n");
20        ret = page_insert(e->env_pgdir, p, i, PTE_W | PTE_U); // 映射这个物理页
}

```

```

21     if(ret != 0) {
22         panic("region_alloc error: page mapping failed! %e\n", ret);
23     }
24 }
25 }
```

ROUNDDOWN和ROUNDUP的宏定义在inc/types.h中。page_alloc()的参数为0,是alloc_flags。这样与ALLOC_ZERO(值为1)相与的结果为0,不进行初始化。注意在page_insert()使用之前必须先分配一个物理页,这个物理页不能是一个空闲页。并且page_insert()会将这个物理页的引用数pp_ref加1,因此在这里就不用直接加1了。如下图所示,page_free_list是static全局变量,具有文件级作用域,不能被其他文件,例如kern/env.c所引用。而数组pages虽然不是static变量,由于page_free_list不能被引用,并且 PageInfo只有一个指向下一个链表元素的指针,因此从这里也找不到pages中的哪个元素对应这个链表的头。因此在正确的前提下,要使用的物理页一定是一个被分配了的物理页。

```

// These variables are set in mem_init()
pde_t *kern_pgdir;           // Kernel's initial page directory
struct PageInfo *pages;       // Physical page state array
static struct PageInfo *page_free_list; // Free list of physical pages
```

Figure 15: kern/pmap.c 的部分全局变量

```

int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    // Fill this function in
    pte_t *pgtab = pgdir_walk(pgdir, va, 1);
    if (!ptab) {
        return -E_NO_MEM;
    }
    pp->pp_ref++; !!!
    if (*ptab & PTE_P) {
        tlb_invalidate(pgdir, va);
        page_remove(pgdir, va);
    }
    *ptab = page2pa(pp) | perm | PTE_P;
    pgdir[PDX(va)] |= perm;
    return 0;
}
```

Figure 16: kern/pmap.c 的 page_insert()

2.2.7 load_icode()

接下来是函数load_icode()。下面是该函数的注释:

```
//  
// 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?  
//
```

Figure 17: 函数 `load_icode()` 的注释

由注释可知，该函数要实现的是为每一个用户进程设置它的初始代码区，堆栈以及处理器标志位。它将 ELF 可执行文件中加载所有可加载的段到该进程的用户地址空间，并让进程从 ELF 文件头所指定的虚拟地址开始执行。同时，它还会将被映射但是没出现在 ELF 文件中的段自动初始化为 0，例如**.bss**段（可能称为节会更好一些）。加载 ELF 文件的过程类似于 boot loader 做的事，思路与**boot/main.c**中的代码也很类似。唯一的不同是 boot loader 需要从磁盘中读取代码。最后，该函数需要为该进程的初始用户栈映射一个物理页。如果函数在执行过程中出现了问题，需要**panic**。该函数只在内核初始化时、运行第一个用户进程前被调用。

函数`load_icode()`的第二个参数**binary**是字节型数据的指针，指向这个 ELF 文件的二进制码，我们首先需要解析它。由于需要解析 ELF 文件，先看 ELF 的文件结构和**inc/elf.h**中 ELF 文件头和程序头的结构体定义：

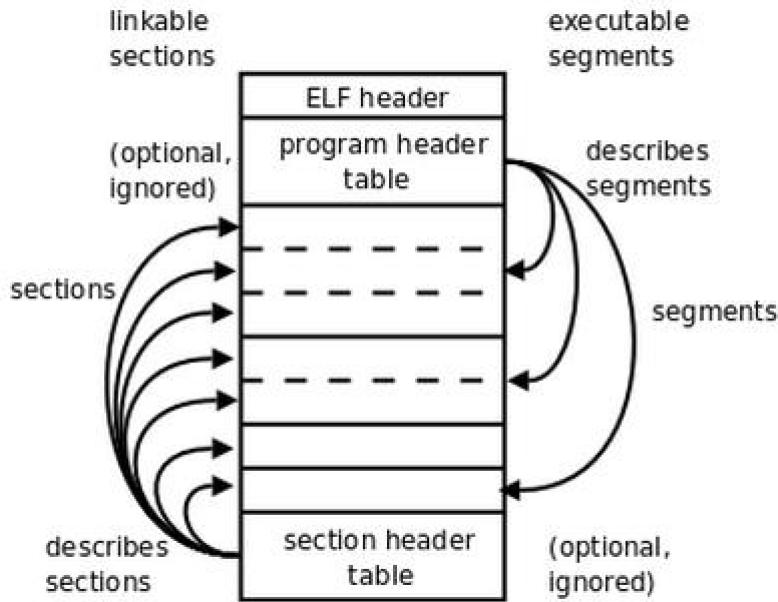


Figure 18: ELF 文件结构

```

#define ELF_MAGIC 0x464C457FU /* "\x7FELF" in little endian */

struct Elf {
    uint32_t e_magic;           // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint32_t e_entry;
    uint32_t e_phoff;
    uint32_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
};

struct Proghdr {
    uint32_t p_type;
    uint32_t p_offset;
    uint32_t p_va;
    uint32_t p_pa;
    uint32_t p_filesz;
    uint32_t p_memsz;
    uint32_t p_flags;
    uint32_t p_align;
};

```

Figure 19: boot/elf.h 的结构体定义

结构体**Elf**是用来解析 ELF 文件头（对应上述 ELF 文件结构的 ELF header）的。以下是结构体**Elf**各个成员变量的说明：

- **e_magic**: ELF 文件类型的标识，它必须要等于一个魔数**ELF_MAGIC**（等于小端字节序的`0x464c457f`）。事实上，绝大多数二进制文件的最开始都有一个独特的标识，用来区分二进制文件的不同格式，ELF 文件也是如此。
- **e_elf**: 存放 ELF 文件的类别、数据、版本等信息的数组。
- **e_type**: ELF 文件的类型，可以是可重定位文件（`.o`），可执行文件，共享目标文件（`.so`）的其中一种。
- **e_machine**: 该 ELF 文件支持的体系结构。
- **e_version**: 该 ELF 文件的版本。
- **e_entry**: ELF 文件的入口点。
- **e_phoff**: ELF 程序头的起始偏移。
- **e_shoff**: ELF 段头的起始偏移。
- **e_flags**: 与 ELF 文件相关的，特定于处理器的标志。（x86 都是 0）
- **e_ehsize**: ELF 文件头的大小。
- **e_phentsize**: ELF 程序头的大小。
- **e_phnum**: ELF 程序头的数目。
- **e_shentsize**: ELF 段头的大小。
- **e_shsize**: ELF 段头的数目。
- **e_shstrndx**: ELF 文件的字符串表段索引。

结构体**Proghdr**是用来解析程序头的。程序头（对应上述 ELF 文件结构的 program header table）是描述一个段或者系统准备程序执行所必需的信息的，它用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。程序头仅对于可执行文件和共享目标文件有意义。以下是结构体**Proghdr**各个成员变量的说明：

- **p_type**: 声明一个段的作用类型，例如从这里可以知道这个段是否可加载。
- **p_offset**: 段相对于文件的起始偏移。
- **p_vaddr**: 段在内存中的起始虚拟地址。

- **p_pa:** 段的起始物理地址。
- **p_filesz:** 段在文件映像中所占的字节数。
- **p_memsz:** 段在内存中所占的字节数。
- **p_flags:** 与段相关的标志，例如 read、write、exec 等。
- **p_align:** 段在文件和内存中的字节对齐方式。

此外还有节头**Secthdr**的结构体定义（对应上述 ELF 文件结构的 section header table），由于这里用不到就不讲了。

然后看boot/main.c中涉及到加载 ELF 文件的函数bootmain():

```

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 */;
}

```

Figure 20: boot/main.c 的 bootmain() 函数

由于要加载的文件由参数**binary**给出，并且目前 JOS 并没有建立文件系统，故不需要从磁盘中加载文件，也就是不需要调用**readseg()**函数。除此之外，**bootmain()**做的第一件事是先判断这个文件是不是一个 ELF 文件（上图红色框部分）。

意思就是先判断该文件最开始的地方是否等于**ELF_MAGIC**，如果不等于**ELF_MAGIC**，则它不是 ELF 文件，需要报错，这是需要**panic**的地方。

在函数体内部还给出了提示：

```

// Hints:
// Load each program segment into virtual memory
// at the address specified in the ELF section 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.

```

Figure 21: 函数 load_icode() 的提示

提示中说我们需要将每个程序段加载到该进程中虚拟内存中的指定位置。我们只需要加载可加载的段（由结构体**Proghdr**的**p_type**指定，值为**ELF_PROG_LOAD**为该段可加载）即可。**binary+ph->p_offset**指向 ELF 文件中的一个程序头的起始位置，我们需要将从这里开始的**ph->p_filesz**个字节的内容拷贝到虚拟地址（VMA）**ph->p_va**（对应一个加载地址（LMA）**ph->p_pa**），剩余的字节应清 0。需要注意一个段在文件中的大小**ph->p_filesz**应不超过其在内存中的大小**ph->p_memsz**（如果不是应**panic**），因此“剩余的字节”即为**ph->p_filesz**到**ph->p_memsz**的部分。**ph->p_filesz**与**ph->p_memsz**不一致的原因是**.bss**段中存在一些未被初始化的静态变量，它们不占用文件存储空间，但是在实际载入之后会占用内存空间（这就是未显式初始化的静态变量会被自动初始化为 0 的原因）。我们需要映射这些内容到物理页上，这些页的权限为用户可读可写，即权限位 U、W、P 均有效。因此我们可以模仿**bootmain()**函数的遍历各个程序头的方式去加载各个段到内存中（紫色框部分，只是不是从磁盘中加载而已）。

此外，还应确保该进程可以从入口点开始执行，观察函数**env_run()**可知，我们需要将地址空间从内核空间切换到该用户进程的空间。

```

// 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.

```

Figure 22: 函数 env_run() 的提示

切换地址空间的方式是使指向页目录首地址的寄存器 cr3 指向该用户进程地址空间的页目录 `e->env_pgdir`, 可以使用函数 `lcr3()`。其定义在 `inc/x86.h`, 是一个内嵌汇编 `movl` 指令, 实现了将页目录首地址赋给寄存器 cr3。注意, 页目录的地址、页表的地址、页表项中的地址都是物理地址, 因此传入的参数应是 `e->env_pgdir` 的物理地址。

```

static __inline void
lcr3(uint32_t val)
{
    __asm __volatile("movl %0,%cr3" : : "r" (val));
}

```

Figure 23: inc/x86.h 的 lcr3() 函数

最后, 需要将用户栈所对应的物理页映射到虚拟地址的 `USTACKTOP-PGSIZE` 处。
因此函数 `load_icode()` 修改如下:

```

1 static void
2 load_icode(struct Env *e, uint8_t *binary)
3 {
4     // Hints:
5     // Load each program segment into virtual memory
6     // at the address specified in the ELF section header.
7     // You should only load segments with ph->p_type == ELF_PROG_LOAD.
8     // Each segment's virtual address can be found in ph->p_va
9     // and its size in memory can be found in ph->p_memsz.
10    // The ph->p_filesz bytes from the ELF binary, starting at
11    // 'binary + ph->p_offset', should be copied to virtual address
12    // ph->p_va. Any remaining memory bytes should be cleared to zero.
13    // (The ELF header should have ph->p_filesz <= ph->p_memsz.)
14    // Use functions from the previous lab to allocate and map pages.
15    //
16    // All page protection bits should be user read/write for now.
17    // ELF segments are not necessarily page-aligned, but you can
18    // assume for this function that no two segments will touch
19    // the same virtual page.

```

```

20 // 
21 // You may find a function like region_alloc useful.
22 //
23 // Loading the segments is much simpler if you can move data
24 // directly into the virtual addresses stored in the ELF binary.
25 // So which page directory should be in force during
26 // this function?
27 //
28 // You must also do something with the program's entry point,
29 // to make sure that the environment starts executing there.
30 // What? (See env_run() and env_pop_tf() below.)
31
32 // LAB 3: Your code here.
33 //获取ELF文件头
34 struct Elf* header = (struct Elf*)binary; //让Elf结构体指针指向binary，以
解析binary的内容
35
36 if (header->e_magic != ELF_MAGIC) //判断是不是ELF文件
37     panic("load_icode error: The binary we load is not elf!\n");
38     if (header->e_entry == 0) //程序入口点是否为空
39         panic("load_icode error: The elf file can't be executed!\n");
40
41 e->env_tf.tf_eip = header->e_entry; //让该进程的eip寄存器指向程序入口点，
以使env_run()可以运行此进程
42 lcr3(PADDR(e->env_pgdir)); //切换地址空间到该用户进程的地址空间
43
44 struct Proghdr *ph, *eph;
45 ph = (struct Proghdr*)((uint8_t *)header + header->e_phoff); //ph为指向
ELF的第一个程序头
46 eph = ph + header->e_phnum; //eph为指向该ELF文件最后一个程序头最末尾（一
共有header->e_phnum个程序头）
47 for(; ph < eph; ph++) {
48     if(ph->p_type == ELF_PROG_LOAD) { //只加载可加载的段
49         if(ph->p_memsz - ph->p_filesz < 0)
50             panic("load_icode failed : p_memsz < p_filesz !\n");
51         region_alloc(e, (void *)ph->p_va, ph->p_memsz); //将从ph->p_va开
始大小为ph->p_memsz的虚拟地址空间映射到物理页
52         memmove((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
53         //将从binary+ph->p_offset开始的ph->p_filesz个字节拷贝给ph->p_va
54         memset((void*)(ph->p_va + ph->p_filesz), 0, ph->p_memsz - ph->
p_filesz); //将该段的虚拟地址空间剩余部分清0
55     }

```

```

55 }
56
57
58 // Now map one page for the program's initial stack
59 // at virtual address USTACKTOP - PGSIZE.
60
61 // LAB 3: Your code here.
62 region_alloc(e,(void *)(USTACKTOP-PGSIZE), PGSIZE); // 映射用户栈
63 lcr3(PADDR(kern_pgdir)); // 注意此时要把地址空间切换回内核空间，因为现在还
64     没有进入用户态
65 }
```

注意，由于这里要映射的地址空间都是用户进程地址空间，用户可读可写，可以直接使用前面修改的函数`region_alloc()`来映射，它直接将权限位设置为用户可读可写(U 和 W 均有效)。

2.2.8 env_create()

接下来是`env_create()`函数，它的注释如下：

```

// 
// 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.
}
```

Figure 24: 函数`env_create()`的注释

此函数较简单，直接使用上面修改的函数`env_alloc()`来分配一个新的用户进程，并使用函数`load_icode()`将一个 ELF 文件(`binary`参数传入)加载到这个新的用户进程中，并指定这个用户进程的类型(由参数`type`指定)。只是要注意的是，如果新的进程没有分配成功需要`panic`。同样，该函数只在内核初始化时、运行第一个用户进程前被调用。

因此，函数`env_create()`修改如下：

```

1 void
2 env_create(uint8_t *binary, enum EnvType type)
3 {
4     // LAB 3: Your code here.
5     struct Env *e;
```

```

6 int ret;
7 if ((ret = env_alloc(&e, 0)) != 0) //新建一个用户进程
8     panic("env_create failed: env_alloc failed!\n");
9
10 load_icode(e, binary); //将ELF文件加载到这个进程中
11 e->env_type = type; //设置该进程的类型，通常为ENV_TYPE_USER（用户进程）
12 }

```

2.2.9 env_run()

最后一个函数env_run()的注释如下，它是真正运行一个进程的函数：

```

// 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.

    panic("env_run not yet implemented");
}

```

Figure 25: 函数 env_run() 的注释

由注释可知，如果当前有一个进程在运行，需要将那个正在运行的进程的状态切换到就绪态（ENV_RUNNABLE）。然后重新将当前的进程设为要运行的进程（参数e传入），将该进程的状态设为运行态ENV_RUNNING，该进程的运行次数+1，当前的地址空间切换为该进程的用户地址空间。然后需要调用env_pop_tf()函数实现进程的上下文切换（该函数也是通过内嵌汇编指令实现一些功能），来恢复该进程的寄存器值（通过pop指令），并将内核态切换到用户态（iret），运行该进程，如果该函数执行成功，将不返回。

```

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

```

Figure 26: 函数 env_pop_tf() 的注释

因此，函数env_run()修改如下：

```

1 void
2 env_run(struct Env *e)
3 {
4     // Step 1: If this is a context switch (a new environment is running):
5     //         1. Set the current environment (if any) back to
6     //             ENV_RUNNABLE if it is ENV_RUNNING (think about
7     //             what other states it can be in),
8     //         2. Set 'curenv' to the new environment,
9     //         3. Set its status to ENV_RUNNING,
10    //         4. Update its 'env_runs' counter,
11    //         5. Use lcr3() to switch to its address space.
12    // Step 2: Use env_pop_tf() to restore the environment's
13    // registers and drop into user mode in the
14    // environment.
15
16    // Hint: This function loads the new environment's state from
17    // e->env_tf. Go back through the code you wrote above
18    // and make sure you have set the relevant parts of
19    // e->env_tf to sensible values.
20
21    // LAB 3: Your code here.
22    //如果当前有一个进程，且这个进程处在运行态
23    if(curenv != NULL && curenv->env_status == ENV_RUNNING)
24        curenv->env_status = ENV_RUNNABLE;//则将这个进程切换到就绪态
25
26    curenv = e;//将当前进程设置为参数e传入的进程
27    curenv->env_status = ENV_RUNNING;//将当前进程的状态置为运行态
28    curenv->env_runs++; //该进程被运行的次数加1

```

```

29 lcr3(PADDR(curenv->env_pgdir)); // 将地址空间切换到当前进程的地址空间
30
31 env_pop_tf(&curenv->env_tf); // 从该进程上次被暂停的时间点恢复该进程的寄存器的值，并从这里开始运行这个进程
32
33 panic("env_run not yet implemented"); // 如果该进程执行成功，env_pop_tf()就不会返回，也就不会执行到这里
34 }

```

前面的函数就是进程分配、创建、运行的一系列流程，函数调用的关系概括如下：

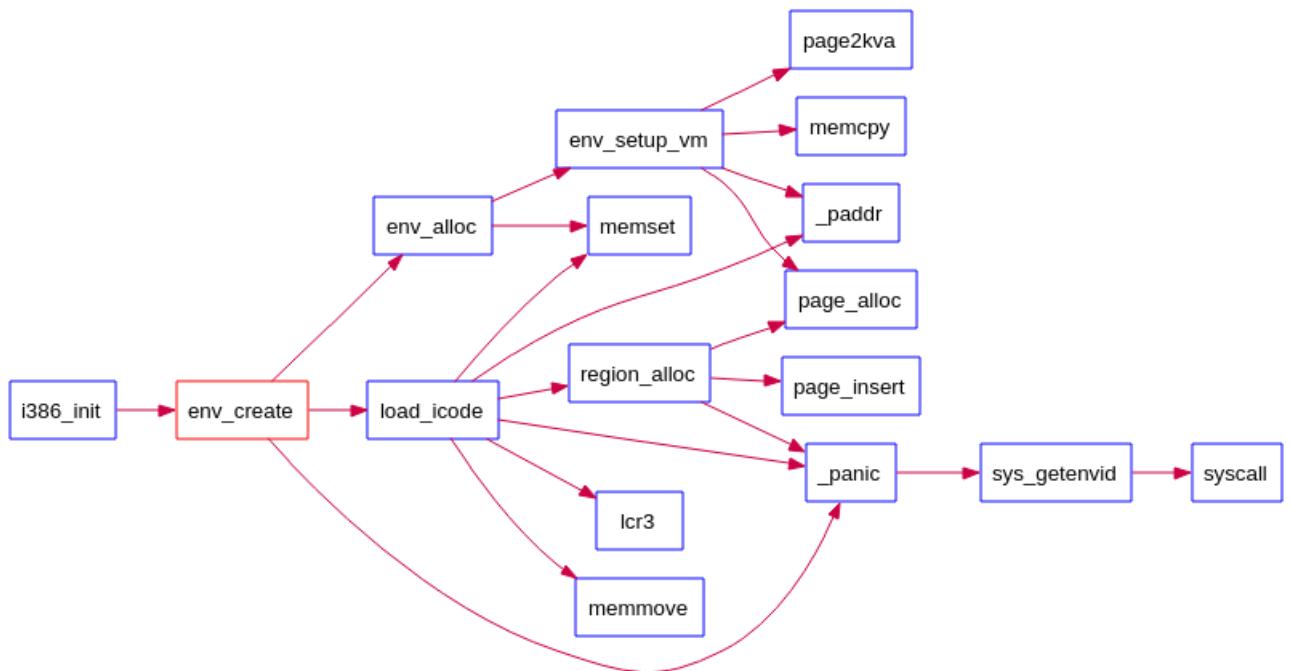


Figure 27: 函数调用关系

现在打开终端，输入`make qemu`来编译内核并查看结果。由于后面的作业还没有完成（还没有异常处理机制），此时 JOS 还没有创建硬件以进行从用户空间到内核空间的转换，在

`[00000000] new env 00001000`

处停下并出现了实验说明中的 register dump 和 triple fault。

```

6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
map region size = 4194304, 1024 pages
map region size = 98304, 24 pages

```

```
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
EAX=00000000 EBX=00000000 ECX=0000000d EDX=eebfde88
ESI=00000000 EDI=00000000 EBP=eebfde60 ESP=eebfde54
EIP=00800b1c EFL=00000092 [--S-A--] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
SS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
DS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS [-WA]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0028 f017d7a0 00000067 00408900 DPL=0 TSS32-avl
GDT= f011b320 0000002f
IDT= f017cfa0 000007ff
CR0=80050033 CR2=00000000 CR3=003bc000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=fffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault. Halting for inspection via QEMU monitor.
```

打开两个终端，一个输入`make qemu-gdb`，另一个进入 GDB，进入调试模式。在 GDB 的终端下，输入命令`b env_pop_tf`在函数`env_pop_tf()`处设置断点，并输入命令`c`运行到此断点处，再多次输入命令`si`单步执行到`iret`以及之后的一条`cmp`指令。由于看不到下一条指令是什么了，因此可以认为下一条指令进入用户态。

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
The target architecture is assumed to be i8086
[f000:ffff] 0xfffff0: ljmp $0xf000,$0xe05b
0x0000ffff in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b env_pop_tf
Breakpoint 1 at 0xf01036a2: file kern/env.c, line 481.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf01036a2 <env_pop_tf>: push %ebp

Breakpoint 1, env_pop_tf (tf=0xf01a0000) at kern/env.c:481
481   {
(gdb) si
=> 0xf01036a3 <env_pop_tf+1>: mov %esp,%ebp
0xf01036a3    481   {
(gdb)
=> 0xf01036a5 <env_pop_tf+3>: sub $0x18,%esp
0xf01036a5    481   {
(gdb)
=> 0xf01036a8 <env_pop_tf+6>: mov 0x8(%ebp),%esp
482           __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0xf01036ab <env_pop_tf+9>: popa
0xf01036ab    482           __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0xf01036ac <env_pop_tf+10>: pop %es
0xf01036ac in env_pop_tf (
    tf=<error reading variable: Unknown argument list address for 'tf'.>
    at kern/env.c:482
        __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0xf01036ad <env_pop_tf+11>: pop %ds
0xf01036ad    482           __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0xf01036ae <env_pop_tf+12>: add $0x8,%esp
0xf01036ae    482           __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0xf01036b1 <env_pop_tf+15>: iret
0xf01036b1    482           __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0x800020: cmp $0xeebfe000,%esp
0x000000020 in ?? ()
(gdb) █

```

Figure 28: GDB 终端

打开文件obj/user/hello.asm，找到指令int \$0x30所在的地址为0x800b1c，输入命令b *0x800b1c在这条指令处设置断点。输入命令c执行到此断点处，再输入命令si多次，发现不能再往下执行了，因此就是这里引发的系统崩溃。

```

user@pc: ~/oslab/lab1/src/lab3/obj/user
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
void
sys_cputs(const char *s, size_t len)
{
    800afc: 55          push %ebp
    800afd: 89 e5        mov %esp,%ebp
    800aff: 83 ec 0c     sub $0xc,%esp
    800b02: 89 5d f4     mov %ebx,-0xc(%ebp)
    800b05: 89 75 f8     mov %esi,-0x8(%ebp)
    800b08: 89 7d fc     mov %edi,-0x4(%ebp)
    800b0b: b8 00 00 00 00  mov $0x0,%eax
    800b10: 8b 4d 0c     mov 0xc(%ebp),%ecx
    800b13: 8b 55 08     mov 0x8(%ebp),%edx
    800b16: 89 c3        mov %eax,%ebx
    800b18: 89 c7        mov %eax,%edi
    800b1a: 89 c6        mov %eax,%esi
    800b1c: cd 30        int $0x30
    syscall(SYS_cputs, 0, (uint32_t)s, len, 0, 0, 0);
}
    800b1e: 8b 5d f4     mov -0xc(%ebp),%ebx
    800b21: 8b 75 f8     mov -0x8(%ebp),%esi
    800b24: 8b 7d fc     mov -0x4(%ebp),%edi
    800b27: 89 ec        mov %ebp,%esp
    800b29: 5d            pop %ebp
    800b2a: c3            ret

```

1601,1 79%

Figure 29: obj/user/hello.asm

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
=> 0xf01036b1 <env_pop_tf+15>: iret
0xf01036b1      482      __asm __volatile("movl %0,%esp\n"
(gdb)
=> 0x800020: cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) b *0x800b1c
Breakpoint 2 at 0x800b1c
(gdb) c
Continuing.
=> 0x800b1c: int    $0x30

Breakpoint 2, 0x00800b1c in ?? ()
(gdb) si
=> 0x800b1c: int    $0x30

Breakpoint 2, 0x00800b1c in ?? ()
(gdb) si
=> 0x800b1c: int    $0x30

Breakpoint 2, 0x00800b1c in ?? ()
(gdb) si
=> 0x800b1c: int    $0x30

```

Figure 30: GDB 终端

在上面的控制台输出中的 register dump 中可以看到 EIP=00800b1c 即这条指令 int \$0x30 所在的地址，它也指示了系统崩溃的具体位置。

当系统启动完成，会加载链接到内部的程序，然后执行，但是因为这个程序内部会调用 int 指令，所以会产生中断，但是还没有建立任何允许从用户空间到内核空间的方

式，所以会产生一次保护异常，然后发现保护异常也处理不了，然后又发生了一次异常，直到发生了三次保护异常，CPU 就会重置。

2.3 练习 1

Read Chapter 9
Exceptions and Interrupts (<http://pdos.csail.mit.edu/6.828/2011/readings/i386/c09.htm>) in the 80386 Programmer's Manual (<http://pdos.csail.mit.edu/6.828/2011/readings/i386/toc.htm>) (or Chapter 5 of the IA-32 Developer's Manual <http://pdos.csail.mit.edu/6.828/2011/readings/ia32/IA32-3A.pdf>), if you haven't already.

2.3.1 题目大意

熟悉 x86 的异常和中断机制。

2.3.2 回答

2.3.3 中断与异常的分类

中断与异常都属于控制流转移机制的一种，两者之间的区别在于：中断是一种来源于处理器外部的事件，处理器用异步方式予以处理。异常则由处理器本身负责检测，属于同步处理。按来源，可以对中断与异常进行以下分类。

中断的分类：

- Maskable Interrupts: 由 INTR 针脚发送
- Nonmaskable Interrupts: 由 NMI 针脚发送

异常的分类：

- 由处理器检测：进一步分类为 Faults, Traps, Aborts 三类
- 通过软件编程（也称为软中断）

三种异常：

- Faults: Faults 异常逻辑上在产生异常的指令之前触发，它可以在指令执行过程中被发现，发现后需要把机器状态回溯到指令执行之前。CS 寄存器和 EIP 寄存器指向产生异常的指令。

- **Traps:** Trap 异常紧接着在产生异常的指令结束后马上触发。CS 寄存器和 EIP 寄存器动态的指向产生异常的指令的下一条指令，如果产生异常的指令是控制流转移指令，例如 JMP 指令产生了 Trap 异常，则转移后的 CS 和 EIP 寄存器值应该被压入栈。
- **Aborts:** Abort 异常既不提供异常产生的准确位置，也不重置机器状态，Abort 异常表示严重的系统错误。

2.3.4 与中断相关的概念

- **中断号:** 每一种中断/异常的都对应一个唯一的中断号，NMI 中断和异常的中断号由 i386 预定义，占据 0 ~ 31。Maskable 中断的中断号由中断控制器（例如典型的 Intel8259A Programmable Interrupt Controller）赋值，并且其赋值逻辑是可以通过软件编程的，中断号空间占据 32 ~ 255.
- **中断屏蔽:** 中断处理的正常时机是在一条指令结束之后，下一条指令开始之前进行。特别的，对于带有 Prefix 前缀的 String 指令在执行时，每完成一次 Repeat，都会进行中断响应。在处理器运行的某些特殊时段，需要对中断处理进行屏蔽，主要包括以下三种情况。
 - 在处理 NMI 中断的时候：在处理一个 NMI 中断的时候，处理器会忽略期间通过 NMI 针脚发送的中断，直到遇到第一个 IRET 指令为止。
 - 通过 IF 标志位屏蔽 Maskable 中断：当 IF=0 时，处理器会延迟通过 INTR 针脚发送的 Maskable 中断的处理，直到 IF=1。CLI 和 STI 指令分别可以用来 clear 和 set IF 标志位，优先权要求：CPL 小于等于 IOPL。
 - 在堆栈切换时：在进行堆栈切换时通常需要类似下面的指令，分别用来更新堆栈段和栈顶：MOV SS, AX、MOV ESP, StackTop 若在这两条指令之间发生中断，将导致堆栈管理数据不一致。为避免这一情况，i386 在通过 MOV 或 POP 指令修改 SS 寄存器之后会延迟处理以下中断异常：NMI 中断、INTR 中断、debug 异常、单步 Traps。其他异常将会获得响应不受影响。
 - 多个中断/异常发生时的优先级：当某个指令间隙有多个中断/异常等待处理时，按照下表的顺序先处理最高有限级中断/异常，低优先级中断在下一个指令间隙执行，低优先级异常在当前处理程序结束后再处理。

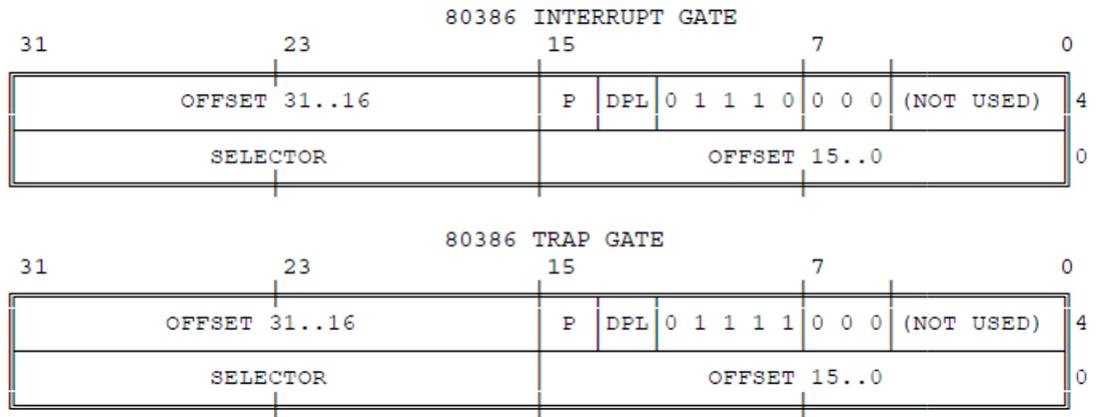
Priority Among Simultaneous Interrupts and Exceptions	
Priority	Class of Interrupt or Exception
HIGHEST	Faults except debug faults Trap instructions INTO, INT n, INT 3 Debug traps for this instruction Debug faults for next instruction NMI interrupt
LOWEST	INTR interrupt

Figure 31: 多个中断/异常发生时的优先级

2.3.5 中断处理

2.3.5.1 中断描述符表 (IDT) 中断描述符表维护了一个中断号到中断处理指令描述符的映射关系，其结构与 GDT、LDT 类似，中断号作为其索引。因为最多只有 256 个中断/异常，所以 IDT 的长度不需要超过 256×8 ，可以小于 256×8 ，因为可能不是所有中断/异常都会发生且有处理程序。处理器通过 IDTR 寄存器找到 IDT，通过 LIDT 和 SIDT 指令分别进行 IDTR 寄存器的 set 和 get 操作。LIDT 指令要求 CPL=0，SIDT 没有优先权要求，可以在任何 CPL 执行。

2.3.5.2 中断描述符 IDT 中可以包含的描述符分为三种：任务门、中断门和 Trap 门。任务门的作用详见 i386 多任务支持的内容，中断门和 Trap 门的格式如下所示。

**Figure 32:** 中断门和 Trap 门的格式

通过任务门、中断门和 Trap 门可以使用以下两种方式来处理中断：使用 Interrupt Procedure 处理中断、使用 Interrupt Task 处理中断。

2.3.5.3 ErrorCode 某些异常会给处理程序/任务传递 ErrorCode, ErrorCode 的结构如下图所示:

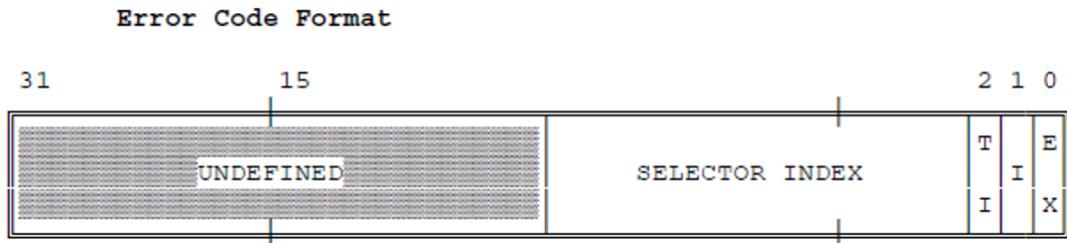


Figure 33: ErrorCode 的结构

ErrorCode 的结构和选择符类似低位的三个标志位的含义为:

- EX: 当中断/异常来自产生异常的程序外部时设置此位
- I: 当 SELECTOR INDEX 指向 IDT 的一个门描述符时设置此位
- TI: TI=0 时, SELECTOR INDEX 选中 GDT 中的描述符; TI=1 时, SELECTOR INDEX 选中 LDT 中的描述符

2.3.5.4 使用 Interrupt Procedure 处理中断 当 IDT 相应位置保存了 Trap 门或者中断门的时候, 将采用 Interrupt Procedure 的方式处理中断, 如下图所示:

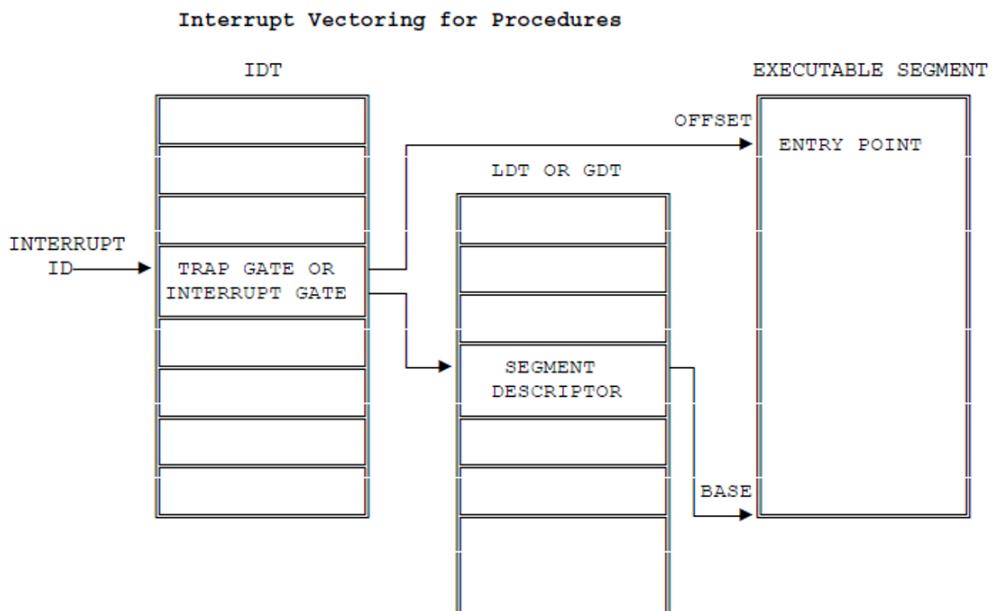


Figure 34: Interrupt Procedure 的方式处理中断

使用 Interrupt Procedure 处理中断时的堆栈操作分情况如下图所示：

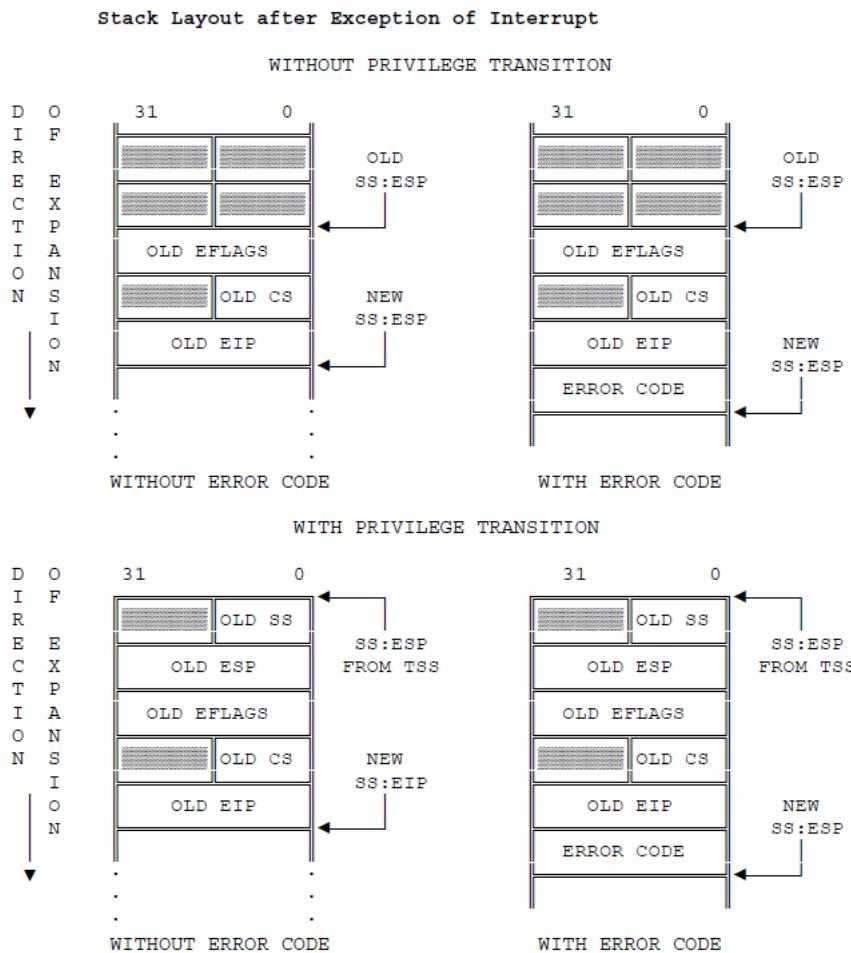


Figure 35: Interrupt Procedure 处理中断时的堆栈操作

使用中断门和 Trap 门的区别在于通过中断门会设 IF 为 0，而通过 Trap 门不会。

2.3.5.5 使用 Interrupt Task 处理中断 当 IDT 相应位置保存了任务门的时候，将采用 Interrupt Task 的方式处理中断，如下图所示：

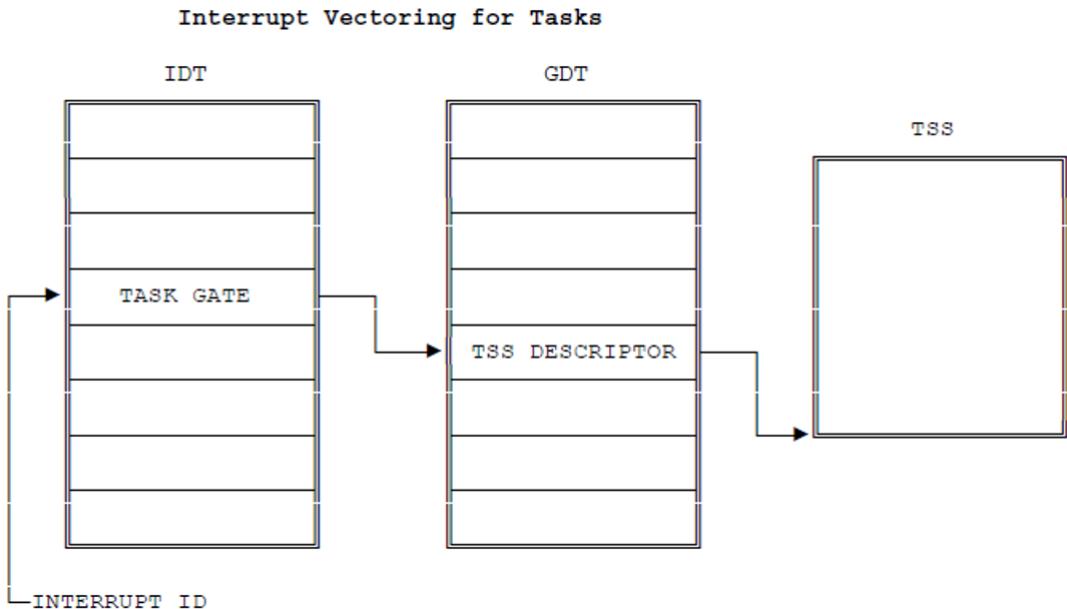


Figure 36: 使用 Interrupt Task 处理中断

Interrupt Task 将会切换到一个独立的上下文来进行中断处理。如果任务切换是由带有 ErrorCode 的异常引起的，那么处理器会自动的把 ErrorCode 推到新任务正确的堆栈上。

2.4 控制权转移

异常和中断都是“控制权转移”，这些都会导致处理器从用户态转移到内核态 (CPL = 0)，用户态在这一过程中没有任何机会来干预内核或者其他进程。在 Intel 的术语中，中断通常是在处理器外部发生的异步的控制权转移，例如外接设备的 I/O 活动通知。相反，异常一般是是一个由当前正在运行的代码造成的同步的控制权转移，例如除零或者非法内存访问。

在 x86 架构中，有两种机制协同工作来提供控制权转移的保护：

1. 中断描述符表 (IDT): x86 允许最多 256 个不同的进入内核的中断或者异常入口点，每个中断都有一个中断号。当中断发生时，CPU 用这个中断号来索引中断描述符表，而这个中断描述符表是在内核的私有内存中建立的，就像 GDT 一样。在这个表的某一项中，处理器会读取：

- EIP 寄存器的值，它指向用于处理这一类型异常的内核代码。
- CS 段寄存器的值，其中包含一些位来表示异常处理代码应该运行在哪一个特权等级（在 JOS 中，所有的异常都在内核模式处理，特权等级为 0）。

例如，x86 处理器可产生的全部同步异常在内部使用 0 ~ 31 作为中断号，因此被映射为中断描述符表项的 0 ~ 31。例如，一个缺页异常的中断号为 14。大于 31 的中断向量只能被用于由 `int` 指令产生的软件中断或者异步硬件中断。

- 任务状态段 (TSS): 在中断或异常发生前，处理器需要一个保存旧的处理器状态的地方。例如，可以保存在处理器调用异常处理函数前的 EIP 和 CS 的值，使得随后异常处理函数可以恢复旧的状态并从中断的地方继续执行。但是这个地方必须避免被没有被授权的代码访问到，否则有错误的或恶意的用户模式代码可能危及到内核。因此，当 x86 处理器遇到使得特权等级从用户模式切换到内核模式的中断或陷阱时，它也会将栈切换到内核栈。TSS 指定了这个栈的段选择子和地址。处理器将 SS, ESP, EFLAGS, CS, EIP 和可能存在错误的代码压入新栈，接着它从中断向量表中读取 CS 和 EIP，并使 ESP 和 SS 指向新栈。

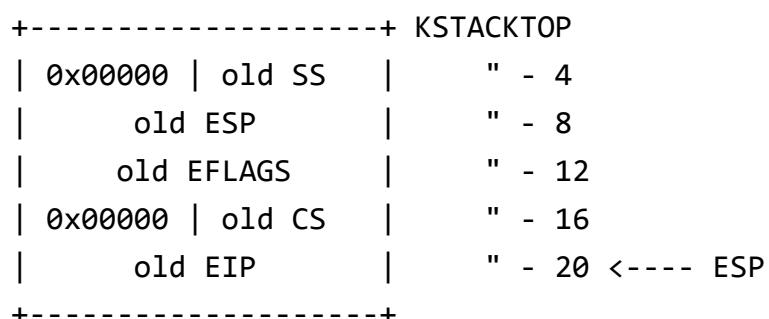
2.4.1 一个例子

例如发生了除 0 错误，那么将会发生如下的事情：

- 处理器转换到由 TSS 中的 SS0 和 ESP0 定义的堆栈，在 JOS 中，这两个区域分别保存着 `GD_KD` 和 `KSTACKTOP`；
- 处理器将异常参数压入到内核栈，起始地址为 `KSTACKTOP`；
- 由于正在处理除零错误，它在 x86 中的中断号是 0，因此处理器读取 IDT 表项 0，并将 CS:EIP 置为这一表项对应的处理程序。
- 处理程序接管 CPU 并处理异常，例如，终止用户进程。

部分中断或异常还需要向内核栈压入错误码，例如缺页异常（中断号 14）。

以下是发生除 0 异常的示意：



以下是发生缺页异常的示意：

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

2.5 中断和异常的嵌套

处理器在内核态或用户态都可能发生中断和异常，但是，只有在从用户态进入内核模式时，x86 处理器才会在将旧寄存器状态压栈、通过 IDT 找到合适的异常处理函数并在调用前自动切换栈。如果中断或异常发生时处理器已经处于内核态（CS 寄存器的低 2 位已经是 00 了），CPU 就只会在同一个内核栈再压入更多的值。通过这种方式，内核可以优雅地处理在内核中发生的嵌套异常。这种特点在实现保护机制时是十分有效的。

如果处理器已经处于内核态并发生了嵌套中断或异常，因为它不需要切换栈，因此它也不需要保存旧的 SS 或者 ESP 寄存器。内核栈在即将调用异常处理程序时状态如下（上为不需压入错误码的异常，下为需要压入错误码的异常）：

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

<---- old ESP		
old EFLAGS " - 4		
0x00000 old CS " - 8		
old EIP " - 12		
error code " - 16 <---- ESP		

2.6 作业 3

2.6.1 题目原文

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.

2.6.2 题目大意

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

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

考虑使用 `pushal` 这条指令。它在形成 `struct Trapframe` 的层次结构时非常合适。

用一些 `user` 目录下会造成异常的程序测试一下你的陷阱处理代码，比如 `user/divzero`。现在，你应该能在 `make grade` 中通过 `divzero`, `softint` 和 `badsegment` 了。

2.6.3 回答

首先，根据作业要求，可以总结出 JOS 对于中断或异常的处理全过程如下所示：

1. 在操作系统启动时，调用函数 `trap_init()` 初始化中断描述符表 IDT，将每一个中断处理程序的起始地址填入 IDT。
2. 当中断发生时，根据任务状态段 TSS 的指定切换到新的内核栈，将被中断的程序的信息（SS，ESP，EFLAGS，CS，EIP，以及可选择是否压栈的错误码）存储到内核栈中。
3. 根据中断向量号索引到存储在中断描述符表 IDT 中的中断处理程序的起始地址，加载其 CS 和 EIP，开始执行中断处理程序。

2.6.4 trapentry.S

首先，每一个中断处理程序都会在内核栈中创建出一个类似于 `struct Trapframe` 结构。然后将该结构传递给 `trap()` 函数做下一步的处理。因此，根据作业要求，首先在 `kern/trapentry.S` 中为每一个中断处理程序定义一个入口点。需要用到下图中定义在 `trapentry.S` 中的以下两个宏定义。

```

/* 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

```

Figure 37: trapentry.S 中的两个宏定义

根据注释可以了解到，两个全局函数都用来定义中断处理函数，且均会传入两个参数，分别为要创建的中断处理函数的函数名和中断向量号。不同的是，由于在进程中断要发生切换时，错误码可以选择压入或不压入栈，因此两个函数分别处理了错误码压栈和不压栈的情况，在第二个宏定义函数TRAPHANDLER_NOEC中处理了不压栈的情况，它比第一个函数TRAPHANDLER多了一句pushl \$0;，说明该函数向栈中压入了一个0，以使trapframe对于两种情况保持相同。此外，注释中还提到，在kern/trapentry.S中定义的符号不能直接.c文件中引用，需要以void NAME()的形式对中断处理函数作出声明。在inc/trap.h中可以了解到不同的中断类型和中断号，如下图所示。

```

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

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

```

Figure 38: inc/trap.h

此外，经过查询可以得知，不同的中断类型是否需要压入错误码、是异常还是中断如下图所示。

Vector	Description	Type	Error code	Exception types
0	Divide error	Fault	No	Fault Faulting instruction not executed CS:EIP is the faulting instruction
1	Reserved			
2	Non-maskable interrupt	Interrupt	No	Trap Trapping instruction executed CS:EIP is the next instruction
3	Breakpoint	Trap	No	
4	Overflow	Trap	No	Abort Location is imprecise; cannot safely resume execution
5	BOUND range exceeded	Fault	No	
6	Invalid/undefined opcode	Fault	No	
7	No math coprocessor	Fault	No	
8	Double fault	Abort	Zero	
9	Reserved			
10	Invalid TSS	Fault	Yes	
11	Segment not present	Fault	Yes	TI 0=GDT, 1=LDT
12	Stack-segment fault	Fault	Yes	IDT 0=GDT/LDT, 1=IDT
13	General protection	Fault	Yes	EXT External event
14	Page fault	Fault	Yes	
15	Reserved		No	
16	x87 FPU error	Fault	No	
17	Alignment check	Fault	Zero	
18	Machine check	Abort	No	P 0=Non-present page 1=Protection-violation
19	SIMD FP exception	Fault	No	W/R Cause (0=Read, 1=Write)
20-31	Reserved			U/S Mode (0=Supervisor, 1=User)
32-255	User defined interrupts	Interrupt	No	

Figure 39: 不同中断的信息

下面完成`_alltraps`的代码部分。在`TRAPHANDLER`和`TRAPHANDLER_NOEC`函数中，其最后都执行了语句`jmp _alltraps`，根据作业要求可知，`_alltraps`函数完成了如下功能：

1. 将被中断的进程的信息压栈，使其形成类似`Trapframe`的结构；
2. 将`GD_KD`，即程序代码段的地址加载到寄存器`%ds`和`%es`中；
3. 将栈顶寄存器`esp`的值作为指向该`Trapframe`的指针压栈，作为即将调用的函数`trap()`的参数；
4. 调用函数`trap()`。

首先了解`Trapframe`的具体结构，`struct Trapframe`定义在文件`inc/trap.h`中，其定义如下所示。

```
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));
```



```
struct Trapframe {
    struct PushRegs tf_regs; //PushReg 定义见上，其中定义了8个通用寄存器
    uint16_t tf_es;
    uint16_t tf_padding1; //将16位寄存器填充为32位
    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));

```

因此，根据代码和注释内容可以画出如下 Trapframe 结构。



Figure 40: 内核栈中 Trapframe 的结构

根据以上信息，可以写出_alltraps的代码。kern/trapentry.S的完整修改如下：

```

1 .text
2
3 /*
4 * Lab 3: Your code here for generating entry points for the different
      traps.

```

```
5  */
6 TRAPHANDLER_NOEC(divide_handler, T_DIVIDE);
7 TRAPHANDLER_NOEC(debug_handler, T_DEBUG);
8 TRAPHANDLER_NOEC(nmi_handler, T_NMI);
9 TRAPHANDLER_NOEC(brkpt_handler, T_BRKPT);
10 TRAPHANDLER_NOEC(oflow_handler, T_OFLOW);
11 TRAPHANDLER_NOEC(bound_handler, T_BOUND);
12 TRAPHANDLER_NOEC(illop_handler, T_ILLOP);
13 TRAPHANDLER_NOEC(device_handler, T_DEVICE);
14 TRAPHANDLER(dblflt_handler, T_DBLFLT);
15 // 9 deprecated since 386
16 TRAPHANDLER(tss_handler, T_TSS);
17 TRAPHANDLER(segnp_handler, T_SEGNP);
18 TRAPHANDLER(stack_handler, T_STACK);
19 TRAPHANDLER(gpflt_handler, T_GPFLT);
20 TRAPHANDLER(pgflt_handler, T_PGFLT);
21 // 15 reserved by intel
22 TRAPHANDLER_NOEC(fperr_handler, T_FPERR);
23 TRAPHANDLER(align_handler, T_ALIGN);
24 TRAPHANDLER_NOEC(mchk_handler, T_MCHK);
25 TRAPHANDLER_NOEC(simderr_handler, T_SIMDERR);
26 // system call (interrupt)
27 TRAPHANDLER_NOEC(syscall_handler, T_SYSCALL);
28
29 /*
30 * Lab 3: Your code here for _alltraps
31 */
32 _alltraps:
33     //第一步: 将被中断的程序的%ds, %es和通用寄存器内容压栈保存
34     pushl %ds
35     pushl %es
36     pushal //压入8个通用寄存器
37
38     //第二步: 将GD_KD的值存入%ds和%es
39     //注意段寄存器不能直接传入立即数, 因此需要%eax中介
40     movl $GD_KD, %eax
41     movl %eax, %ds
42     movl %eax, %es
43
44     //第三步: %esp入栈
45     pushl %esp
46     //第四步: 调用trap()函数
```

47 call trap

2.6.5 trap_init()

下一部分将编写文件kern/trap.c中的trap_init()函数，以便在操作系统启动时使用该函数对中断符号表进行初始化。首先，根据trapentry.S中的注释可知，在创建IDT时需要对中断处理函数作出声明。其次在这一步中，使用到了在文件inc/mmu.h中的SETGATE宏定义：SETGATE(gate, istrap, sel, off, dpl)，其完整定义如下图所示。该函数用于初始化IDT，其中各个参数的意义为：

- **gate**——中断在IDT中的索引
- **istrap**——表示异常(1)或中断(0)
- **sel**——代码段选择器，选择对应的中断或异常的处理程序，本题中进入内核因此是GD_KT
- **off**——段偏移量
- **dpl**——权限描述符，0为内核态，3为用户态。

```
// 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;
}
```

Figure 41: SETGATE 宏定义

根据上述信息完成trap_init()函数的编写，完整代码为：

```
1 void
2 trap_init(void)
3 {
4     extern struct Segdesc gdt[];
5
6     // LAB 3: Your code here.
7     // 声明后才能使用
8     void divide_handler();
9     void debug_handler();
10    void nmi_handler();
11    void brkpt_handler();
12    void oflow_handler();
13    void bound_handler();
14    void illop_handler();
15    void device_handler();
16    void dblflt_handler();
17    void tss_handler();
18    void segnp_handler();
19    void stack_handler();
20    void gpflt_handler();
21    void pgflt_handler();
22    void fperr_handler();
23    void align_handler();
24    void mchk_handler();
25    void simderr_handler();
26    void syscall_handler();
27
28    //Exception
29    SETGATE(idt[T_DIVIDE], 1, GD_KT, divide_handler, 0);
30    SETGATE(idt[T_DEBUG], 1, GD_KT, debug_handler, 0);
31    SETGATE(idt[T_NMI], 0, GD_KT, nmi_handler, 0); //non-maskable interrupt
32    SETGATE(idt[T_BRKPT], 1, GD_KT, brkpt_handler, 3); //需要调用monitor, 所以权限为3
33    SETGATE(idt[T_OFLOW], 1, GD_KT, oflow_handler, 0);
34    SETGATE(idt[T_BOUND], 1, GD_KT, bound_handler, 0);
35    SETGATE(idt[T_ILOP], 1, GD_KT, illop_handler, 0);
36    SETGATE(idt[T_DEVICE], 1, GD_KT, device_handler, 0);
37    SETGATE(idt[T_DBLFLT], 1, GD_KT, dblflt_handler, 0);
38    SETGATE(idt[T_TSS], 1, GD_KT, tss_handler, 0);
39    SETGATE(idt[T_SEGNP], 1, GD_KT, segnp_handler, 0);
40    SETGATE(idt[T_STACK], 1, GD_KT, stack_handler, 0);
41    SETGATE(idt[T_GPFLT], 1, GD_KT, gpflt_handler, 0);
```

```

42 SETGATE(idt[T_PGFLT], 1, GD_KT, pgflt_handler, 0);
43 SETGATE(idt[T_FPERR], 1, GD_KT, fperr_handler, 0);
44 SETGATE(idt[T_ALIGN], 1, GD_KT, align_handler, 0);
45 SETGATE(idt[T_MCHK], 1, GD_KT, mchk_handler, 0);
46 SETGATE(idt[T SIMDERR], 1, GD_KT, simderr_handler, 0);
47
48 //Interrupt
49 SETGATE(idt[T_SYSCALL], 0, GD_KT, syscall_handler, 3); //由于是用户程序请求系统调用，因此权限也为3
50
51
52 // Per-CPU setup
53 trap_init_percpu();
54 }
```

以下是完成作业 3 的运行结果：

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xfffffb0c
TRAP frame at 0xf01a1000
    edi 0x00000000
    esi 0x00000000
    ebp 0xeebfde60
    oesp 0xfffffdcc
    ebx 0x00000000
    edx 0xeebfde88
    ecx 0x0000000d
    eax 0x00000000
    es 0x----0023
    ds 0x----0023
    trap 0x00000030 System call
    err 0x00000000
    eip 0x00800b1e
    cs 0x----001b
    flag 0x00000092
    esp 0xeebfde54
    ss 0x----0023
[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>
```

Figure 42: 运行结果

2.7 Challenge!

注：这个 Challenge 不是实验文档上的，但是我觉得这个题比较有指导意义，有必要做一下。

2.7.1 题目原文

You probably have a lot of very similar code right now, between the lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean this up. Change the macros in trapentry.S to automatically generate a table for trap.c to use. Note that you can switch between laying down code and data in the assembler by using the directives .text and .data.

2.7.2 题目大意

在作业 3 中，出现了很多行非常相似的代码，事实上这些相似的代码没有必要重复写这么多行。调整 `trapentry.S` 中的宏定义，让它自动生成一个给 `trap.c` 使用的表。可以在汇编中通过 `.text` 和 `.data` 来随时在代码段和数据段切换。

2.7.3 第一处修改

通过对 `kern/trapentry.S` 中的两个宏定义可知，我们可以将这两个宏定义合并成一个。因为这两个宏定义根本区别就是这个异常/中断是否需要向栈中压入错误码，而宏定义 `TRAPHANDLER_NOEC` 只比 `TRAPHANDLER` 多了一行 `pushl $0;`。这个完全可以通过条件判断来解决，而不必将相似的宏定义写两遍。

首先要合并这两个宏定义，合并之后的宏定义如下：

```
#define TRAPHANDLER(name, tori, num, errc, dpl) \
    .text; \
        .globl name; /* define global symbol for 'name' */ \
        .type name, @function; /* symbol type is function */ \
        .align 2; /* align function definition */ \
        name: /* function starts here */ \
            .if errc==0; \
                pushl $0; \
            .endif; \
            pushl $(num); \
            jmp _alltraps; \
    .data; \
        .long name, tori, num, dpl
```

Figure 43: 修改后的宏定义

这里将原来宏定义的三个变量增加到了 5 个变量，增加的变量含义如下：

- `tori`: 是中断还是异常。值为 0 是中断，为 1 是异常。
- `errc`: 是否需要错误码，如果不需要则为 0。
- `dpl`: 触发该异常或中断的用户权限。

最后将变量name、tori、num、dpl写进了.data段的定义，意在是要声明一个二维数组。

下面是这个二维数组的声明和调用这个被修改的宏定义：

```
.data
    .globl handlers
handlers:
.text

/*
 * Lab 3: Your code here for generating entry points for the different traps.
*/
TRAPHANDLER(divide_handler, 1, T_DIVIDE, 0, 0);
TRAPHANDLER(debug_handler, 1, T_DEBUG, 0, 0);
TRAPHANDLER(nmi_handler, 1, T_NMI, 0, 0);
TRAPHANDLER(brkpt_handler, 1, T_BRKPT, 0, 3);
TRAPHANDLER(oflow_handler, 1, T_OFLOW, 0, 0);
TRAPHANDLER(bound_handler, 1, T_BOUND, 0, 0);
TRAPHANDLER(illop_handler, 1, T_ILLOP, 0, 0);
TRAPHANDLER(device_handler, 1, T_DEVICE, 0, 0);
TRAPHANDLER(dbldlt_handler, 1, T_DBFLT, 1, 0);
TRAPHANDLER(tss_handler, 1, T_TSS, 1, 0);
TRAPHANDLER(segnp_handler, 1, T_SEGNP, 1, 0);
TRAPHANDLER(stack_handler, 1, T_STACK, 1, 0);
TRAPHANDLER(gpflt_handler, 1, T_GPFLT, 1, 0);
TRAPHANDLER(pgflt_handler, 1, T_PGFILT, 1, 0);
TRAPHANDLER(fperr_handler, 1, T_FPERR, 0, 0);
TRAPHANDLER(align_handler, 1, T_ALIGN, 1, 0);
TRAPHANDLER(mchk_handler, 1, T_MCHK, 0, 0);
TRAPHANDLER(simderr_handler, 1, T_SIMDERR, 0, 0);
TRAPHANDLER(syscall_handler, 0, T_SYSCALL, 0, 3);
.data
    .long 0, 0, 0, 0
```

Figure 44: 声明二维数组、调用宏定义

由于这里要在idt中插入19个表项，这些对宏定义的调用是数组的前19行，最后一行的.long 0,0,0,0指明数组的定义结束了（这是数组的最后一行）。因此数组的大小为20*4。其中，宏定义的最后一行.long name,tori,num,dpl指明在数组的每一行中，name是第0个元素、tori是第1个元素、num是第2个元素、dpl是第3个元素。这样，对宏定义的调用就是对数组的填充，这个数组的首地址将会被汇编器赋给符号handlers。

当然，这里对宏定义的调用虽然也重复了很多行，但是这个重复是不可控的，因为每个中断/异常的信息都不一样。

因此对kern/trapentry.S的修改如下：

```
1 /* 这里的反斜杠代表这个宏定义是跨行的，还要注意宏定义未结束的语句末尾的分号
   */
2 #define TRAPHANDLER(name, tori, num, errc, dpl)      \
3 .text;          \
4     .globl name;    /* define global symbol for 'name' */ \
5     .type name, @function; /* symbol type is function */ \
```

```
6 .align 2; /* align function definition */ \
7 name: /* function starts here */ \
8 .if errc==0; \
9     pushl $0; \
10 .endif; \
11     pushl $(num); \
12     jmp _alltraps; \
13 .data; \
14     .long name, tori, num, dpl
15
16 .data
17     .globl handlers
18     handlers:
19 .text
20
21 /*
22 * Lab 3: Your code here for generating entry points for the different
23 traps.
24 */
25 TRAPHANDLER(divide_handler, 1, T_DIVIDE, 0, 0);
26 TRAPHANDLER(debug_handler, 1, T_DEBUG, 0, 0);
27 TRAPHANDLER(nmi_handler, 0, T_NMI, 0, 0);
28 TRAPHANDLER(brkpt_handler, 1, T_BRKPT, 0, 3);
29 TRAPHANDLER(oflow_handler, 1, T_OFLOW, 0, 0);
30 TRAPHANDLER(bound_handler, 1, T_BOUND, 0, 0);
31 TRAPHANDLER(illop_handler, 1, T_ILLOP, 0, 0);
32 TRAPHANDLER(device_handler, 1, T_DEVICE, 0, 0);
33 TRAPHANDLER(dblflt_handler, 1, T_DBLFLT, 1, 0);
34 // 9 deprecated since 386
35 TRAPHANDLER(tss_handler, 1, T_TSS, 1, 0);
36 TRAPHANDLER(segnp_handler, 1, T_SEGNP, 1, 0);
37 TRAPHANDLER(stack_handler, 1, T_STACK, 1, 0);
38 TRAPHANDLER(gpflt_handler, 1, T_GPFLT, 1, 0);
39 TRAPHANDLER(pgflt_handler, 1, T_PGFLT, 1, 0);
40 // 15 reserved by intel
41 TRAPHANDLER(fperr_handler, 1, T_FPERR, 0, 0);
42 TRAPHANDLER(align_handler, 1, T_ALIGN, 1, 0);
43 TRAPHANDLER(mchk_handler, 1, T_MCHK, 0, 0);
44 TRAPHANDLER(simderr_handler, 1, T_SIMDERR, 0, 0);
45 TRAPHANDLER(syscall_handler, 0, T_SYSCALL, 0, 3);
46 .data
```

```

47 .long 0, 0, 0, 0
48
49
50 .text
51 /*
52 * Lab 3: Your code here for _alltraps
53 */
54 _alltraps:
55     //第一步: 将被中断的程序的%ds, %es和通用寄存器内容压栈保存
56     pushl %ds
57     pushl %es
58     pushal //压入8个通用寄存器
59
60
61     //第二步: 将GD_KD的值存入%ds和%es
62     //注意段寄存器不能直接传入立即数, 因此需要%eax中介
63     movl $GD_KD, %eax
64     movl %eax, %ds
65     movl %eax, %es
66
67     //第三步: %esp入栈
68     pushl %esp
69     //第四步: 调用trap()函数
70     call trap

```

2.7.4 第二处修改

第二处修改，是对函数`trap_init()`的修改。由于原来这里对`SETGATE`宏的调用也重复了很多次，因此可以利用上面定义的数组把对`SETGATE`宏的一系列调用改写成循环形式。具体的修改如下：

```

1 void
2 trap_init(void)
3 {
4     extern struct Segdesc gdt[];
5
6     // LAB 3: Your code here.
7     extern long handlers[][][4];
8     int i;
9
10    for (i = 0; handlers[i][0] != 0; i++) //循环结束的标志是对应trapentry.S
        中数组的name值为0, 即已经遍历到数组的最后一行, 退出循环

```

```

11     SETGATE(idt[handlers[i][2]], handlers[i][1], GD_KT, handlers[i][0],
12             handlers[i][3]);
13
14 // Per-CPU setup
15 trap_init_percpu();
16 }
```

这里不直接声明数组大小的而用全 0 的一行作为数组的结束标志的原因是方便对kern/trapentry.s中定义的表项进行扩充，这样就不需要修改trap_init()了。

2.8 问题 1

2.8.1 题目原文

- 1 . What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)
- 2 . Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int \$14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int \$14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

2.8.2 题目大意

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

2.8.3 回答

1. 对于每一个不同的中断/异常类型，x86 架构都会根据不同的中断号到中断符号表中寻找对应的中断处理程序，以便对不同的中断/异常做出不同的响应。例如，有些中断的产生原因是程序要执行 I/O 操作，在内核与硬件完成交互后，还要返回继续执行当初被中断的程序，这也是在切换栈帧之前，要储存被中断的进程的寄存器值的原因。而有的中断/异常则表示程序的执行出现了问题，这时在处理过错误后，就不会继续返回执行该程序，因此区分不同的中断处理程序是有必要的。尽管不同的中断/异常在处理时会有部分相似的内容，但如果为所有的中断/异常设定统一的处理函数，必定还要在处理完相同的步骤后在函数内部区分不同的中断/异常类型已完成其特殊化的功能，那么这与为每一个中断/异常创建不同的处理程序本质上并无区别。
2. 通过前述定义在 `inc/trap.h` 中的中断向量号可以了解到，中断号 13 是 general protection fault，中断号 14 是 page fault。而 `user/softint` 程序运行在用户态，特权级为 3，且根据所学知识，`int $14` 即 page fault 应由内核态处理，特权级为 0。因此 `softint` 无权处理中断号为 14 的错误，因此导致了 general protection fault 的产生。

如果内核允许 `softint` 产生中断号为 14 的中断，进而调用内核的缺页处理函数，就会导致用户态进程能够直接调用内核函数，使内核暴露在用户态进程中，极易造成系统的执行紊乱和错误，同时也使得程序易于被篡改，影响系统的正常、安全运行。

至此，Lab 3 的 Part A 已结束，以下是在终端输入命令 `make grade` 得到的打分情况：

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
+ mk obj/kern/kernel.img
make[1]: 离开目录“/home/user/oslab/lab1/src/lab3”
divzero: OK (3.4s)
softint: OK (0.9s)
badsegment: OK (1.6s)
Part A score: 30/30

faultread: FAIL (1.9s)
...
    check_page_installed_pgdir() succeeded!
    [00000000] new env 00001000
GOOD Incoming TRAP frame at 0xeffffffbc
GOOD TRAP frame at 0xf01a0000
    edi 0x00000000
    esi 0x00000000
...
    es 0x----0023
    ds 0x----0023
GOOD trap 0x0000000e Page Fault
    cr2 0x00000000
GOOD err 0x00000004 [user, read, not-present]
    eip 0x00800039
    cs 0x----001b
...

```

Figure 45: Part A 打分情况

3 缺页中断、断点异常、系统调用

经过 Part A 的修改，JOS 内核已经有了一些基本的异常处理能力，但是还需要细化它来提供基于异常处理的重要的操作系统机制。下面是 Part B 涉及的 3 个重要的中断：

3.1 背景知识

3.1.1 缺页中断

缺页中断的中断号为 14 (`T_PGFLT`)。当处理器发生缺页时，它将造成缺页的虚拟地址存储在一个特别的处理器控制寄存器 CR2 中。在 `kern/trap.c` 的 `page_fault_handler()` 是处理缺页中断的函数。

3.1.2 断点异常

断点异常的中断号为 3 (`T_BKPT`)，它可以看作是我们平时在调试时设置的断点，设置断点的机理就是调试器临时将想要设置断点处指令替换为 1 字节 `int 3` 软中断指令。在 JOS 中，它的使用范围被扩大到任何用户进程都可以唤起 JOS 内核监视器的伪

系统调用。在 `lib/panic.c` 中定义的用户模式下的 `panic()` 方法，就是打印出 `panic message` 之后调用一个 `int 3`。如下图：

```

/*
 * Panic is called on unresolvable fatal errors.
 * It prints "panic: <message>", then causes a breakpoint exception,
 * which causes JOS to enter the JOS kernel monitor.
 */
void
_panic(const char *file, int line, const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);

    // Print the panic message
    cprintf("[%08x] user panic in %s at %s:%d: ",
            sys_getenvid(), binaryname, file, line);
    vprintf(fmt, ap);
    cprintf("\n");

    // Cause a breakpoint exception
    while (1)
        asm volatile("int3");
}

```

Figure 46: lib/panic.c

这是 Lab 3 的新文件，这下总算知道了 `panic()` 到底是干什么的了，在 Lab 2 的只有一个宏定义和函数声明，还真的没看出来 `panic` 是干什么的。

3.1.3 系统调用

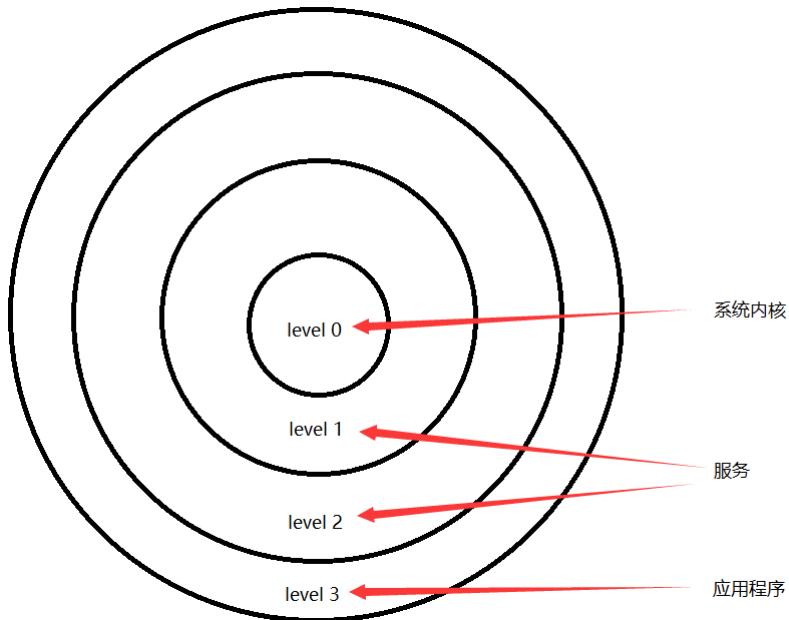
用户程序通过系统调用来请求内核为其做些事情。当用户进程进行系统调用时，处理器进入内核态，处理器和内核协作来保存用户进程的当前状态，内核执行对应的代码来处理系统调用，并恢复用户进程。用户进程发出系统调用请求的方式因系统而异。

在 JOS 内核中，系统调用是使用会造成处理器中断的 `int $0x30` 指令。特别地，`int $0x30` 被作为系统调用中断，中断号为 48 (`T_SYSCALL`)。注意，硬件是不会产生这个中断的，所以允许用户代码生成它也不会有什么歧义。

应用会将系统调用号和系统调用参数放入寄存器。这样的话，内核也不用去用户进程的栈或者指令流中到处找了。系统调用号会存在 `%eax` 中，最多 5 个参数会相应地存在 `%edx`, `%ecx`, `%ebx`, `%edi` 和 `%esi` 中。内核将返回值放在 `%eax` 中。`lib/syscall.c` 的 `syscall()` 是发起系统调用的函数。

3.1.4 特权级

在分段机制中，特权级总共有 4 个特权级别，从高到低分别是 0、1、2、3，数字越小表示的特权级别越大。如下图所示：

**Figure 47:** 特权级示意图

CPL、DPL 和 RPL 是三个重要的特权级，解释如下：

- CPL 是当前执行的程序或任务的特权级。它被存储在 CS 和 SS 的第 0 位和第 1 位上。通常情况下，CPL 代表代码所在的段的特权级。当程序转移到不同特权级的代码段时，处理器将改变 CPL。只有 0 和 3 两个值，分别表示用户态和内核态。
- DPL 表示段或门的特权级。它被存储在段描述符或者门描述符的 DPL 字段中，当当前代码段试图访问一个段或者门，DPL 将会和 CPL 以及段或者门选择子的 RPL 相比较，根据段或者门类型的不同，DPL 将会区别对待。
- RPL 是通过段选择子的第 0 和第 1 位表现出来的。RPL 是代码中根据不同段跳转而确定，以动态刷新 CS 里的 CPL，在代码段选择符中。而且 RPL 对每个段来说不是固定的，两次访问同一段时的 RPL 可以不同。操作系统往往用 RPL 来避免低特权级应用程序访问高特权级段内的数据，即便提出访问请求的段有足够的特权级，如果 RPL 不够也是不行的，当 RPL 的值比 CPL 大的时候，RPL 将起决定性作用。也就是说，RPL 相当于附加的一个权限控制，只有当 $RPL > DPL$ 的时候，才起到实际的限制作用。

通过调用门进行程序的转移控制时，CPU 会检查以下这几个字段：

1. 当前代码段的 CPL；
2. 调用门描述符中的 DPL；

3. 调用门描述符中的 RPL;
4. 目的代码描述符的 DPL;
5. 目标代码段描述符中的一致性标志 C。

如下图所示：

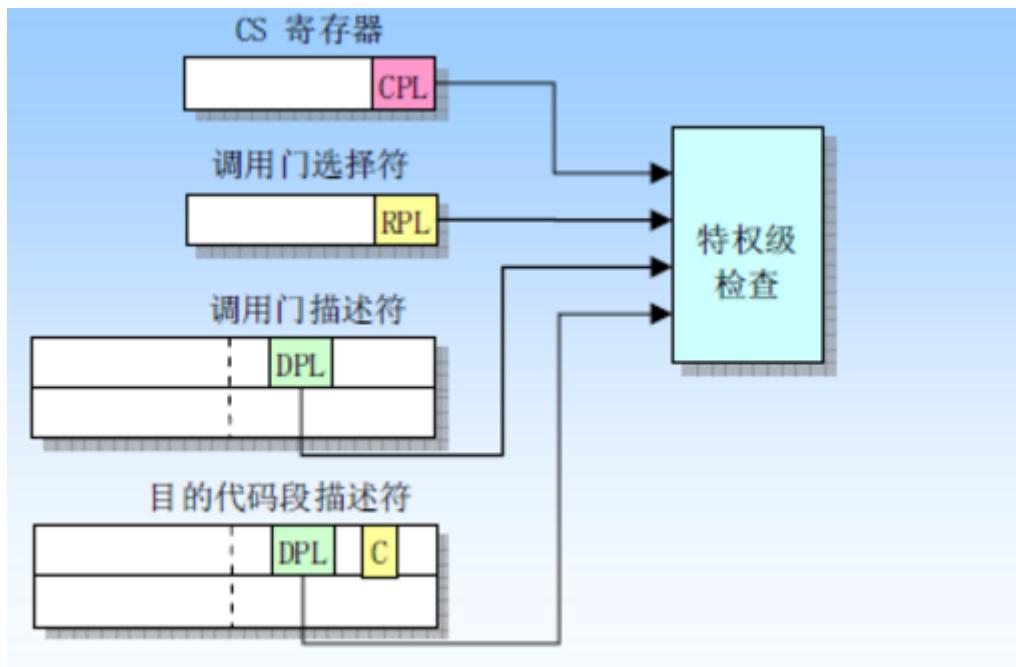


Figure 48: 特权级检查

3.2 作业 4

3.2.1 题目原文

```
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.
```

3.2.2 题目大意

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

3.2.3 回答

在 `trapentry.S` 中，这两个宏定义 `TRAPHANDLER` 和 `TRAPHANDLER_NOEC` 定义了当系统检测到一个中断 / 异常时，需要首先完成的一部分操作。我们要根据这个中断是否有中断错误码，来选择调用 `TRAPHANDLER` 还是 `TRAPHANDLER_NOEC`。前者是需要中断错误码的，后者是不需要的。然后再统一调用 `_alltraps` 来执行中断处理程序。

根据 `inc/trap.h` 文件中的 `Trapframe` 结构体（详见作业 3）可以知道，`Trapframe` 中的 `tf_trapno` 成员代表这个中断的中断码。所以在 `trap_dispatch()` 函数中我们需要根据输入的 `Trapframe` 指针 `tf` 中的 `tf_trapno` 来选择中断。在这里我们需要判断是否是缺页中断，如果是则执行缺页中断的处理函数 `page_fault_handler()`。

因此 `trap_dispatch()` 可修改如下：

```

1 static void
2 trap_dispatch(struct Trapframe *tf)
3 {
4     // Handle processor exceptions.
5     // LAB 3: Your code here.
6     switch(tf->tf_trapno) {
7         case T_PGFLT:
8             page_fault_handler(tf);
9             return;
10    }
11
12    // Unexpected trap: The user process or the kernel has a bug.
13    print_trapframe(tf);
14    if (tf->tf_cs == GD_KT)
15        panic("unhandled trap in kernel");
16    else {
17        env_destroy(curenv);
18        return;
19    }
20 }
```

3.3 作业 5

3.3.1 题目原文

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.

3.3.2 题目大意

修改trap_dispatch()使断点异常发生时，能够触发 kernel monitor。修改完成后运行 make grade，运行结果应该是你修改后的 JOS 能够正确运行 breakpoint 测试程序。

3.3.3 回答

作业 5 其实是在作业 4 的trap_dispatch()函数上修改的，我们只需加上处理断点中断 (T_BRKPT) 的情况就好了。

通过调用定义在 kern/monitor.c 文件中的 monitor() 函数（如下图），这是断点异常的处理函数，来完成作业 5 代码的修改。

```
void
monitor(struct Trapframe *tf)
{
    char *buf;

    cprintf("Welcome to the JOS kernel monitor!\n");
    cprintf("Type 'help' for a list of commands.\n");

    if (tf != NULL)
        print_trapframe(tf);

    while (1) {
        buf = readline("K> ");
        if (buf != NULL)
            if (runcmd(buf, tf) < 0)
                break;
    }
}
```

Figure 49: kern/monitor.c 的 monitor() 函数

因此，trap_dispatch()函数修改如下：

```
1 static void
2 trap_dispatch(struct Trapframe *tf)
3 {
4     // Handle processor exceptions.
```

```
5 // LAB 3: Your code here.
6 switch(tf->tf_trapno) {
7     case T_PGFLT:
8         page_fault_handler(tf);
9         return;
10    case T_BRKPT:
11        monitor(tf);
12        return;
13    }
14
15 // Unexpected trap: The user process or the kernel has a bug.
16 print_trapframe(tf);
17 if (tf->tf_cs == GD_KT)
18     panic("unhandled trap in kernel");
19 else {
20     env_destroy(curenv);
21     return;
22 }
23 }
```

3.4 问题 2

3.4.1 题目原文

1、The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

2、What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

3.4.2 题目大意

3.4.3 回答

- 在kern/trap.c 里面的trapname()函数中我们可以对应的看到各种trap的名字。在kern/trap.c 里的trap_init()函数中我们调用了很多次SETGATE宏定义来实

现对 IDT 的初始化，详见作业 3。如果我们想要当前执行的程序能够跳转到这个描述符所指向的程序哪里继续执行的话，有个要求，就是要求当前运行程序的 CPL, RPL 的最大值需要小于等于 DPL，否则就会出现优先级低的代码试图去访问优先级高的代码的情况，就会触发 general protection exception。我们的测试程序首先运行于用户态，它的 CPL 为 3，当异常发生时，它希望去执行 int 3 指令，这是一个系统级别的指令，用户态命令的 CPL 一定大于 int 3 的 DPL，所以就会触发 general protection exception。但是如果我们把 SETGATE 中的参数 dpl 设置为 3 时，就不会出现这样的现象了，我们就能让断点异常正常工作了。这个已经在作业 3 中完成了。

2. 在 inc/mmu.h 中可以找到以下门描述符定义：

```
// 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
};
```

Figure 50: inc/mmu.h 的 Gatedesc 结构体

这是一种保护机制。如果特权级被 SETGATE 设为 0（内核），用户态调用会触发保护。如果用户态可调用，则恶意程序一直调用该中断，造成资源浪费。优先级高低由 gd_dpl 来决定，数字越小优先级越高。

3.5 作业 6

3.5.1 题目原文

在内核中为中断 T_SYSCALL 添加一个处理程序。
完成在 kern/trapentry.S 和 kern/trap.c 文件中的 trap_init()
修改 trap_dispatch() 来处理系统调用中断，通过调用 syscall() 在文件 kern/
syscall.c 中），使用合适的参数并将返回值写回 %eax
完成 kern/syscall.c 文件中的 syscall() 函数。如果系统调用号是无效的，syscall
() 函数返回 - E_INVAL。阅读并理解 lib/syscall.c 文件可以让你更加理解系统
调用。
在你的内核运行 user/hello 程序（make run-hello）。它将在控制台输出“hello
world”然后会引起一个用户模式下的缺页中断。

3.5.2 回答

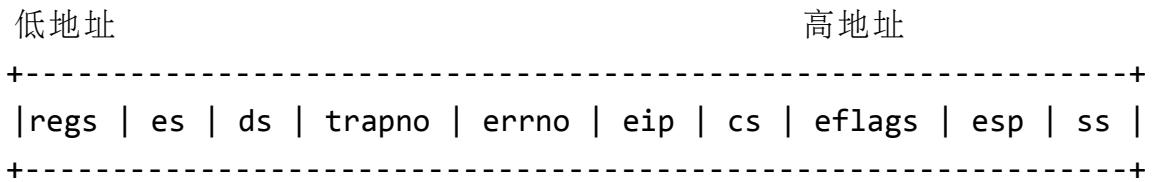
首先需要在kern/trapentry.S中添加如下的代码:

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

trapentry.S中的TRAPHANDLER和TRAPHANDLER_NOEC用来初始化中断处理函数。它们定义了一个全局可见的函数来处理陷阱。它将陷阱号推入堆栈,然后跳转到_alltraps。TRAPHADLER_NOEC用来处理CPU没有指定错误代码的trap。它用0代替错误代码,因此它的trap frame和TRAPHADLER具有同样的格式。

由用户模式发生中断进入内核时, CPU会切换到内核栈,并压入旧的SS,ESP,EFLAGS,CS,EIP寄存器的值。接着,执行中断处理程序。这里,会先通过TRAPHANDLER压入中断向量以及错误码(如果有),然后在_alltraps中压入旧的DS,ES寄存器以及通用寄存器的值,接着将DS,ES寄存器设置为GD_KD,并将此时ESP寄存器的值压入到内核栈中作为trap函数的参数,然后才调用trap(tf)函数。

做这些处理的作用是在内核栈中构造Trapframe的结构,这样_alltraps之后,trap(Trapframe tf)中参数tf指向的内核栈,而栈中内容正好是一个完整的Trapframe结构。



tf是传递给函数trap_dispatch(struct Trapframe *tf)的Trapframe结构体实例。

然后需要在kern/trap.c的trap_init()函数中添加如下代码:

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

它定义了一条编号为T_SYSCALL(48),名称为th_syscall的trap。SETGATE定义在inc/mmu.h中,用来设置一个一般的interrupt/trap门描述符,各个参数的意义见作业3。

idt []定义在kern/trap.h中:

`extern struct Gatedesc idt[];`, Gatedesc是定义在inc/mmu.h中的门描述符(图见问题2)。因此idt[]是全局的门描述符表。

上述工作已在作业3中完成。

3.5.3 第一处修改

在kern/trap.c的trap_dispatch()的修改如下:

```
1 static void
2 trap_dispatch(struct Trapframe *tf)
3 {
4     // Handle processor exceptions.
5     // LAB 3: Your code here.
6     switch(tf->tf_trapno) {
7         case T_PGFLT:
8             page_fault_handler(tf);
9             return;
10        case T_BRKPT:
11            monitor(tf);
12            return;
13        case T_SYSCALL:
14            tf->tf_regs.reg_eax = syscall(
15                tf->tf_regs.reg_eax,
16                tf->tf_regs.reg_edx,
17                tf->tf_regs.reg_ecx,
18                tf->tf_regs.reg_ebx,
19                tf->tf_regs.reg_edi,
20                tf->tf_regs.reg_esi);
21            return;
22    }
23
24    // Unexpected trap: The user process or the kernel has a bug.
25    print_trapframe(tf);
26    if (tf->tf_cs == GD_KT)
27        panic("unhandled trap in kernel");
28    else {
29        env_destroy(curenv);
30        return;
31    }
32}
```

函数trap_dispatch (struct Trapframe *tf)用来处理程序异常。tf是一个Trapframe结构型变量，struct Trapframe定义在inc/trap.h中，它存储了当前进程的寄存器的值。Trapframe中的tf_regs是一个PushRegs结构型变量,struct PushRegs同样定义在inc/trap.h中。函数syscall()声明在kern/syscall.h中，它的原型是syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)，作用是调用与syscallno参数对应的函数。

3.5.4 第二处修改

对kern/syscall.c中的syscall()函数的修改如下:

```

1 // Dispatches to the correct kernel function, passing the arguments.
2 int32_t
3 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
4         a4, uint32_t a5)
5 {
6     // Call the function corresponding to the 'syscallno' parameter.
7     // Return any appropriate return value.
8     // LAB 3: Your code here.
9
10    //panic("syscall not implemented");
11    switch (syscallno) {
12        case SYS_cputs:
13            sys_cputs((const char *)a1, a2);
14            return 0;
15        case SYS_cgetc:
16            return sys_cgetc();
17        case SYS_getenvid:
18            return sys_getenvid();
19        case SYS_env_destroy:
20            return sys_env_destroy(a1);
21        default:
22            return -E_INVALID;
23    }
24 }
```

其中，switch语句的变量syscallno，它代表系统调用的序号。是以下枚举值的其中一个，定义在inc/syscall.h中，如下图所示：

```

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

Figure 51: inc/syscall.h 的枚举常量

这些枚举常量（除了NSYSCALLS以外）分别选择inc/syscall.c中定义的一系列系统调用函数：sys_cputs(), sys_cgetc(), sys_getenvid(), sys_env_destroy()。它

们实现的功能为将字符串打印到系统控制台，或者当发生内存错误时结束进程。

至此，作业 6 完成了。但此时会在执行`user/hello.c`的第二句`cprintf()`报 page fault，因为还没有设置它用到的`thisenv`的值，如下图所示。我们必须要在作业 7 中，在设置`lib/libmain.c`的`libmain()`设置`thisenv`的值才能成功。

```
// hello, world
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    cprintf("hello, world\n");
    cprintf("i am environment %08x\n", thisenv->env_id);
}
```

Figure 52: user/hello.c

3.6 作业 7

3.6.1 题目原文

添加所需代码到用户字典中，然后启动你的内核。让它可以使 `user/hello` 打印出“`hello, world`”然后打印“`i am environment 00001000`”。然后 `user/hello` 会尝试调用 `sys_env_destroy()` 退出（详见 `lib/libmain.c` 和 `lib/exit.c`）。由于内核当前只支持一个程序，所以当前程序退出后，内核就会显示当前唯一的程序已经退出并且陷入内核监视器。此时，`make grade` 就可以通过 `hello` 测试了。

3.6.2 题目大意

3.6.3 回答

首先回顾一下系统调用的完整流程。

以`user/hello.c`输出字符串为例，其中调用了`cprintf()`，注意这是`lib/print.c`中的`cprintf`，该`cprintf()`最终会调用`lib/syscall.c`中的`sys_cputs()`，`sys_cputs()`又会调用`lib/syscall.c`中的`syscall()`，该函数将系统调用号放入`%eax`寄存器，五个参数依次放入`DX, CX, BX, DI, SI`寄存器中，然后执行指令`int 0x30`（系统调用），发生中断后，去 IDT 中查找中断处理函数，最终会走到`kern/trap.c`的`trap_dispatch()`中，我们根据中断号`0x30`，又会调用`kern/syscall.c`中的`syscall()`函数（注意这时候我们已经进入了内核模式`CPL=0`），在该函数中根据系统调用号调用`kern/print.c`中的`cprintf()`函数，该函数最终调用`kern/console.c`中的`cputchar()`将字符串打印到

控制台。当`trap_dispatch()`返回后，`trap()`会调用`env_run()`，该函数会将`currenv->env_tf`结构中保存的寄存器快照重新恢复到寄存器中，这样又会回到用户程序系统调用之后的那条指令运行，只是这时候已经执行了系统调用并且寄存器`eax`中保存着系统调用的返回值。任务完成重新回到用户模式`CPL=3`。

当从内核态切换到用户态时，是先调用`lib/libmain.c`文件中的`libmain()`函数（需要在这里给`thisenv`赋值），再调用这个程序的`umain()`（从`thisenv`取得它的`env_id`）。

我们需要修改该函数初始化其中的`const volatile struct Env *thisenv`变量。`libmain()`函数修改如下：

```
1 void
2 libmain(int argc, char **argv)
3 {
4     // set thisenv to point at our Env structure in envs[].
5     // LAB 3: Your code here.
6     //thisenv = 0;
7     thisenv = &envs[ENVX(sys_getenvid())]; //取当前环境currenv所对应的envs数组元素的地址，赋给thisenv。
8
9     // save the name of the program so that panic() can use it
10    if (argc > 0)
11        binaryname = argv[0];
12
13    // call user main routine
14    umain(argc, argv);
15
16    // exit gracefully
17    exit();
18 }
```

其中，`ENVX`的宏定义在`inc/env.h`中，如下图：

```

// An environment ID 'envid_t' has three parts:
//
// +-----+-----+-----+
// |0| Uniqueifier | Environment |
// | |           |       Index   |
// +-----+-----+-----+
//                               \--- ENVX(eid) ---
//
// The environment index ENVX(eid) equals the environment's offset 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).
// envy_ts less than 0 signify errors. The envy_t == 0 is special, and
// stands for the current environment.

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

```

Figure 53: ENVX 宏定义

由注释可知，一个envid的结构如下：

1. 第 31 位恒为 0。
2. 第 10 位到第 21 位是以环境的不同的创建时间为标识的。前面说过，不同时间创建的环境可能共用一个envs数组中的元素。
3. 第 0 位到第 9 位才是真正ENVX宏定义的含义，它表示该环境所对应的envs数组元素相对于数组首元素的偏移（位置）。因此，((envid) & (NENV - 1))的含义就好理解了，即只取envid最后的 10 位。

函数sys_getenvid()的定义在kern/syscall.c，它返回当前环境的env_id。

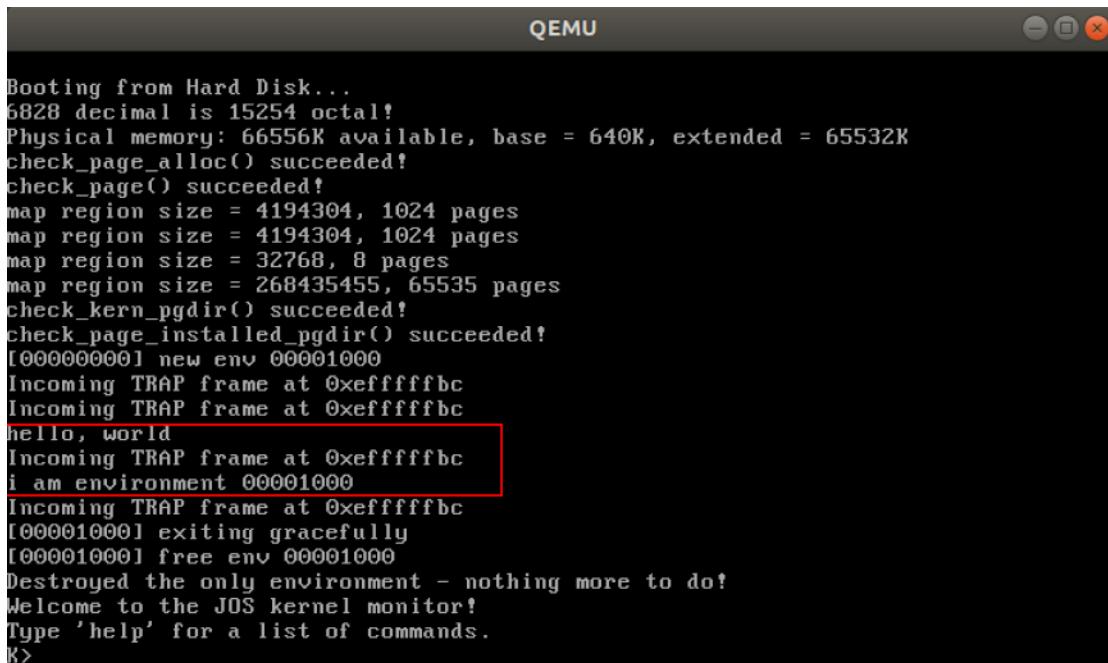
```

// Returns the current environment's envyd.
static envyd_t
sys_getenvid(void)
{
    return curenv->env_id;
}

```

Figure 54: kern/syscall.c 的 sys_getenvid() 函数

此时启动内核就可以使 user/hello 打印出hello, world然后打印
i am environment 00001000。



```

Booting from Hard Disk...
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
map region size = 4194304, 1024 pages
map region size = 4194304, 1024 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 000001000
Incoming TRAP frame at 0xfffffb0
Incoming TRAP frame at 0xfffffb0
hello, world
Incoming TRAP frame at 0xfffffb0
i am environment 000001000
Incoming TRAP frame at 0xfffffb0
[000001000] exiting gracefully
[000001000] free env 000001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

Figure 55: 运行结果

3.7 作业 8

3.7.1 题目原文

完成以下内容：

修改文件 kern/trap.c，使得在内核中出现缺页错误时，就终止(使用 panic)内核。

要检验缺页中断是发生在用户模式还是内核模式，可以检查 tf_cs 的最低几位。

阅读文件 kern/pmap.c 中的 user_mem_assert()并实现同文件中的 user_mem_check()。

修改文件 kern/syscall.c 来检查传递给内核的参数

启动内核，运行 user/buggyhello。用户进程会被销毁而内核将会中止(panic)。将会出现如下输出：

```
[000001000] user_mem_check assertion failure for va 000000001
[000001000] free env 000001000
Destroyed the only environment - nothing more to do!
```

最后，修改 kern/kdebug.c 中的 debuginfo_eip，在usd、stabs、stabstr 调用 user_mem_check。如果你运行 user/breakpoint，你就应该从内核监视器中可以运行 backtrace，并且可以在内核发生缺页错误之前看见 backtrace 贯穿 lib/libmain.c。这个缺页错误你不需要修改，但你需要知道为什么会发生这个错误。

3.7.2 第一处修改

在kern/trap.c中: (page_fault_handler()函数中)。

由2.5节可知,可以根据CS寄存器的最低2位判断当前运行的程序是处于内核模式还是用户模式。`tf->tf_cs`表示CPU的当前权限级别(CPL),等于0表示内核模式,等于3表示用户模式。当缺页中断发生在内核模式时,使用`panic`终止内核。

```

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.
    if((tf->tf_cs & 3) == 0) {
        panic("page_fault in kernel mode, fault address 0x%08x\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.
    cprintf("[%08x] user fault va %08x ip %08x\n",
            curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}

```

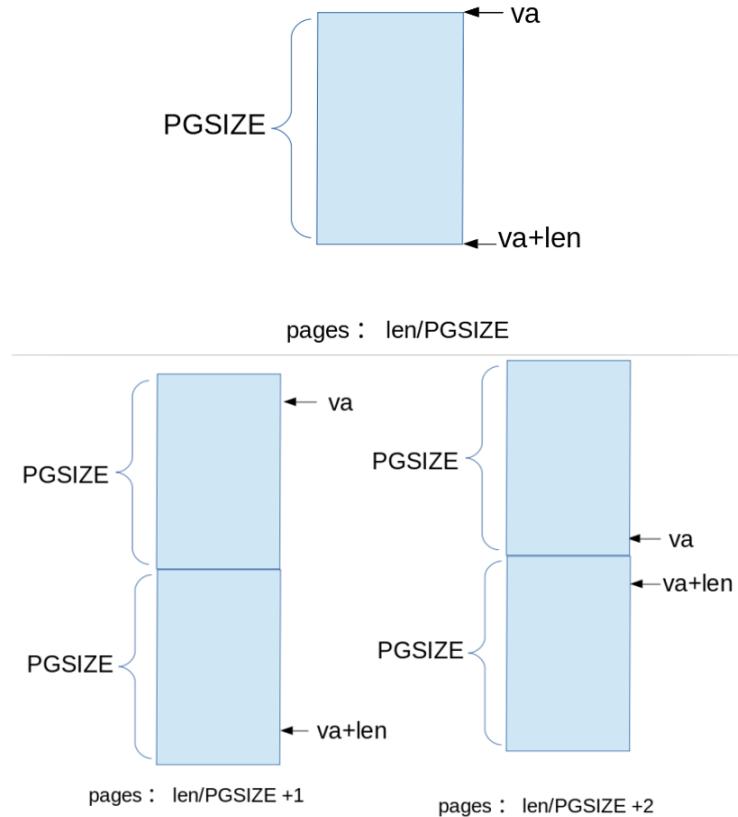
Figure 56: kern/trap.c

3.7.3 第二处修改

在kern/pmap.c中:

`user_mem_assert()`函数调用了`user_mem_check()`函数。`user_mem_check()`函数检查当前用户模式程序是否具有`perm | PTE_P`权限访问对虚拟地址空间`[va, va+len]`。在当前用户模式程序的页表中,找到与该地址范围相对应的页表条目,然后查看页表条目中有关访问权限的字段,检查是否包含`perm | PTE_P`。若不包含,则当前程序没有对该地址范围的`perm | PTE_P`权限。

`va`向下取整, `va+len`向上取整, 检查每一页。根据`va`和`len`的位置关系,需要检查的页面的个数可能为: `len/PGSIZE`、`len/PGSIZE+1`或`len/PGSIZE+2`。

**Figure 57:** va 和 len 的位置关系

如果虚拟地址低于ULIM，用户程序可以访问该地址，且给予权限。如果有错误，将user_mem_check_addr设置为第一个错误的虚拟地址。如果用户程序可以访问该地址范围，则返回0，否则返回-EFAULT。因此user_mem_check()修改如下：

```

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

```

Figure 58: kern/pmap.c 的 user_mem_check() 函数

3.7.4 第三处修改

在kern/syscall.c中：

检查用户程序是否有权访问虚拟地址空间[s, s+len]，使用user_mem_assert()函数实现。

```
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);
}
```

Figure 59: kern/syscall.c 的 sys_cputs() 函数

打开终端，输入命令make run-buggyhello运行结果如下（用户进程销毁，内核中止）：

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

Figure 60: make run-buggyhello

3.7.5 第四处修改

在kern/kdebug.c中：

调用user_mem_check()函数，检查usd、stabs、stabstr是否有效。若无效，返回-1值。

```
// 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;
}
```

Figure 61: kern/kdebug.c 的 debuginfo_eip() 函数

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

Figure 62: kern/kdebug.c 的 debuginfo_eip() 函数

此时，打开终端，输入命令`make run-breakpoint`，并在内核监视器中输入命令`backtrace`，得到以下输出：

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Incoming TRAP frame at 0xfffffb0
Incoming TRAP frame at 0xfffffb0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01a1000
    edi 0x00000000
    esi 0x00000000
    ebp 0xebfd0
    oesp 0xfffffdc
    ebx 0x00000000
    edx 0x00000000
    ecx 0x00000000
    eax 0xec000000
    es 0x----0023
    ds 0x----0023
    trap 0x00000003 Breakpoint
    err 0x00000000
    eip 0x08000037
    cs 0x----001b
    flag 0x00000046
    esp 0xebfd0
    ss 0x----0023
K> backtrace
Stack backtrace:
    ebp efffff10 eip f01008f8 args 00000001 efffff28 f01a1000 00000000 f017e960
        kern/monitor.c:147: monitor+268
    ebp efffff80 eip f0103c8a args f01a1000 efffffb0 00000000 00000000 00000000
        kern/trap.c:200: trap+192
    ebp efffffb0 eip f0103d8 args efffffb0 00000000 00000000 eebfd0 efffffdc
        kern/trapentry.S:97: <unknown>+0
    ebp eebfd0 eip 00800007c args 00000000 00000000 00000000 00000000 00000000
        lib/libmain.c:27: libmain+67
Incoming TRAP frame at 0xfffffe7c
kernel panic at kern/trap.c:275: page_fault in kernel mode, fault address 0xeebf e008
Welcome to the JOS kernel monitor!

```

Figure 63: breakpoint 程序的 backtrace

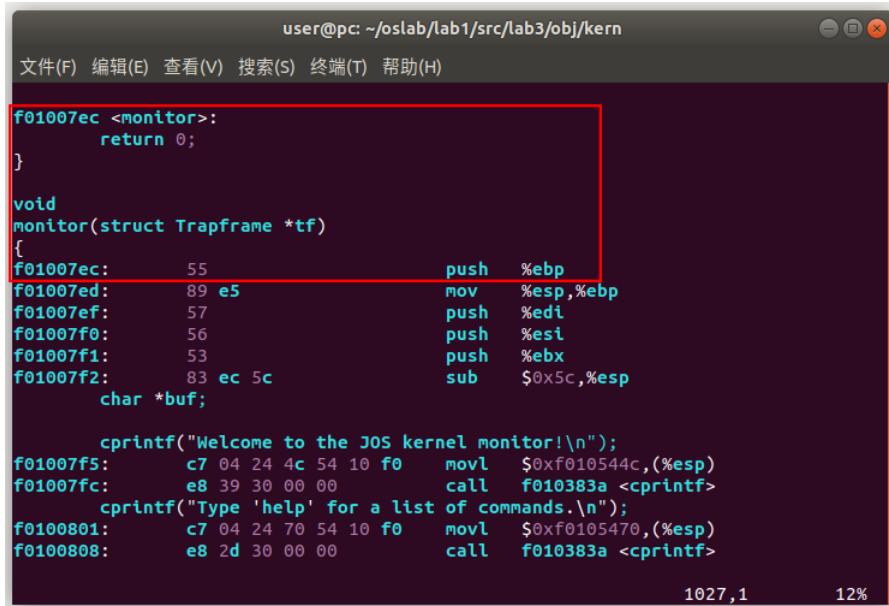
在地址`0xeebf e008`处发现了缺页错误。

3.7.6 为什么会发生缺页错误？

由`inc/memlayout.h`可看出，用户栈顶`USTACKTOP`的虚拟地址`0xeebf e000`。在上图的`ebp`中（红色框部分），只有`0xebfd0`在用户栈空间中（值小于`USTACKTOP`），其他都在内核栈空间中。

打开两个终端，一个输入`make run-breakpoint-gdb`，另一个启动 GDB，进入调试模式。

从`obj/kern/kernel.asm`中找到断点异常的处理函数`monitor()`的首地址为`0xf01007ec`，在文件`kern/monitor.c`中。



```

user@pc: ~/oslab/lab1/src/lab3/obj/kern

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

f01007ec <monitor>:
    return 0;
}

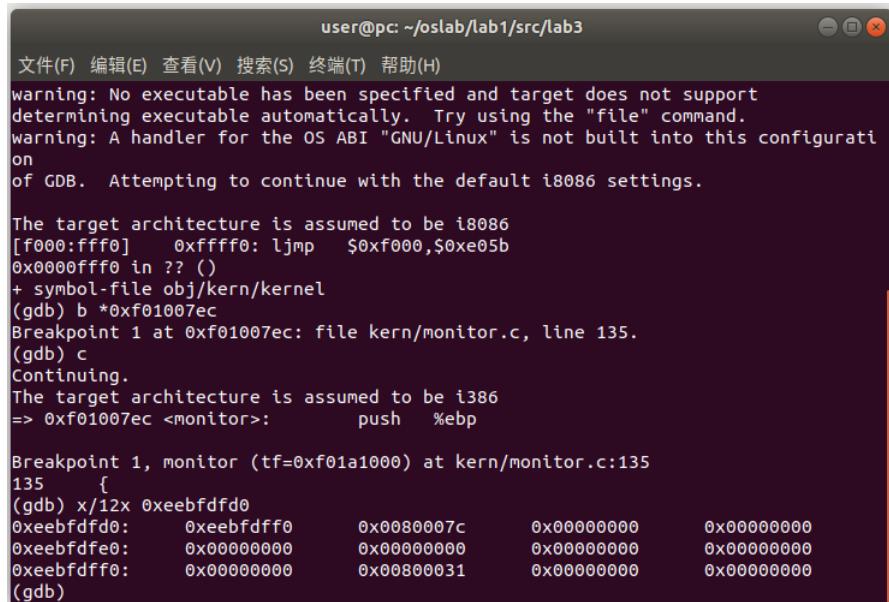
void
monitor(struct Trapframe *tf)
{
f01007ec:    55          push   %ebp
f01007ed:    89 e5       mov    %esp,%ebp
f01007ef:    57          push   %edi
f01007f0:    56          push   %esi
f01007f1:    53          push   %ebx
f01007f2:    83 ec 5c   sub    $0x5c,%esp
    char *buf;

    cprintf("Welcome to the JOS kernel monitor!\n");
f01007f5:    c7 04 24 4c 54 10  f0    movl   $0xf010544c,(%esp)
f01007fc:    e8 39 30 00 00       call   f010383a <cprintf>
    cprintf("Type 'help' for a list of commands.\n");
f0100801:    c7 04 24 70 54 10  f0    movl   $0xf0105470,(%esp)
f0100808:    e8 2d 30 00 00       call   f010383a <cprintf>

```

Figure 64: obj/kern/kernel.asm

因此在 GDB 中输入命令 `b *0xf01007ec` 在 `monitor()` 函数设置断点，并输入 `c` 执行到断点处，再输入命令 `x/12x 0xeeebfdfd0` 查看 `0xeeebfdfd0` 周围的数据。



```

user@pc: ~/oslab/lab1/src/lab3

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:ffff] 0xfffff0: ljmp $0xf000,$0xe05b
0x00000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0xf01007ec
Breakpoint 1 at 0xf01007ec: file kern/monitor.c, line 135.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf01007ec <monitor>: push %ebp

Breakpoint 1, monitor (tf=0xf01a1000) at kern/monitor.c:135
135  {
(gdb) x/12x 0xeeebfdfd0
0xeeebfdfd0: 0xeeebfdfd0 0x0080007c 0x00000000 0x00000000
0xeeebfdfe0: 0x00000000 0x00000000 0x00000000 0x00000000
0xeeebfdff0: 0x00000000 0x00800031 0x00000000 0x00000000
(gdb)

```

Figure 65: GDB

下图为原来 Lab 1 修改的 `mon_backtrace()` 函数：

```

int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // Read ebp of mon_backtrace()
    unsigned int *ebp = (unsigned int *) read_ebp();
    // The first five args of the current function
    unsigned int args[5];
    cprintf("Stack backtrace:\n");
    while(ebp) {
        unsigned int eip = (unsigned int)*(ebp + 1);
        args[0] = (unsigned int)*(ebp + 2);
        args[1] = (unsigned int)*(ebp + 3);
        args[2] = (unsigned int)*(ebp + 4);
        args[3] = (unsigned int)*(ebp + 5);
        args[4] = (unsigned int)*(ebp + 6);
        cprintf("    ebp %08x    eip %08x    args %08x %08x %08x %08x\n",
               ebp, eip,
               args[0], args[1], args[2], args[3], args[4]);
        struct Eipdebuginfo info;
        debuginfo_eip((uintptr_t)eip, &info);
        cprintf("    %s:%d: %.%s+%d\n",
               info.eip_file, info.eip_line,
               info.eip_fn_namelen, info.eip_fn_name, eip - info.eip_fn_addr);
        ebp = (unsigned int *)*ebp;
    }
    return 0;
}

```

Figure 66: mon_backtrace()

GDB 输出的最后一行数据分析如下：

```

Breakpoint 1, monitor (tf=0xf01a1000) at kern/monitor.c:135
135  {
(gdb) x/12x 0xeeebfdfd0
0xeeebfdfd0: 0xeeebfdff0      0x0080007c      0x00000000      0x00000000
0xeeebfdfe0: 0x00000000      0x00000000      0x00000000      0x00000000
0xeeebfdff0: 0x00000000      0x00800031      0x00000000      0x00000000
(gdb)                                     eip          args[0]      args[1]
                                         ↓下一个ebp为0

```

Figure 67: 分析

由ebp为0xeeebfdfd0中取得的调用者ebp为0xeeebfdff0，从调用者ebp取得的下一个调用者的ebp为0（即下一步就是回溯的终点）。也就是说mon_backtrace()要输出的下一行的eip为0x00800031，前两个参数args[0]和args[1]均为0。而args[1]的值所在的地址刚好就是0xeeebfe000。第5个参数args[4]的地址0xeeebfe008刚好就是panic的fault address。因此期望输出的args[2]以及后面的参数的地址均已越过（大于）用户栈顶。

如果将mon_backtrace()修改成只输出前2个参数(args[0]和args[1])，而不是前5个参数，那么就没有发生缺页错误：

The screenshot shows a terminal window with the following output:

```

user@pc: ~/oslab/lab1/src/lab3
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
ecx 0x00000000
eax 0xeeec0000
es 0x----0023
ds 0x----0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x00800037
cs 0x----001b
flag 0x00000046
esp 0xebfd0
ss 0x----0023
K> backtrace
Stack backtrace:
    ebp efffff10 eip f01008e3 args 00000001 efffff28
        kern/monitor.c:149: monitor+268
    ebp efffff80 eip f0103c7a args f01a1000 efffffb0
        kern/trap.c:200: trap+192
    ebp efffffb0 eip f0103d98 args efffffb0 00000000
        kern/trapentry.S:97: <unknown>+0
    ebp eebfdfd0 eip 0080007c args 00000000 00000000
        lib/libmain.c:27: libmain+67
    ebp eebfdfff0 eip 00800031 args 00000000 00000000
        lib/entry.S:34: <unknown>+0
K>

```

Figure 68: 修改 mon_backtrace() 之后的输出

```

// Entry point - this is where the kernel (or our parent environment)
// starts us running when we are initially loaded into a new environment.
.text
.globl _start
_start:
    // See if we were started with arguments on the stack
    cmpl $USTACKTOP, %esp
    jne args_exist

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

args_exist:
    call libmain
1:     jmp 1b

```

Figure 69: lib/entry.S

由此可以看出,回溯的终点在lib/entry.S中。lib/entry.S决定内核是否根据esp的位置加载用户环境。用户环境只有一个,因此最初用户堆栈为空,esp指向USTACKTOP。因此会跳过跳转语句执行到下面,程序压入两个伪参数(pushl \$0),然后调用libmain()。当mon_backtrace()函数打印完前两个参数后想要打印第三个参数时,由于从当前栈帧中参数个数小于5,mon_backtrace()又是从低地址向高地址遍历的,已超出界限,到了libmain()的栈帧中。由lib/entry.S的注释可知,在调用libmain()前仍处于内核态,调用之后才真正进入用户态,导致内核态的mon_backtrace()函数就进入到了用户栈空间取数据(由作业2可知,内核地址空间只映射了UTOP及以上的部分,而USTACKTOP低于UTOP,USTACKTOP并没有映射到相应的物理页中),这就引发了缺页

错误。

因此，解决的方法就是让`mon_backtrace()`在调用`libmain()`之前就打印完 5 个参数，所以需要继续压入 3 个伪参数：

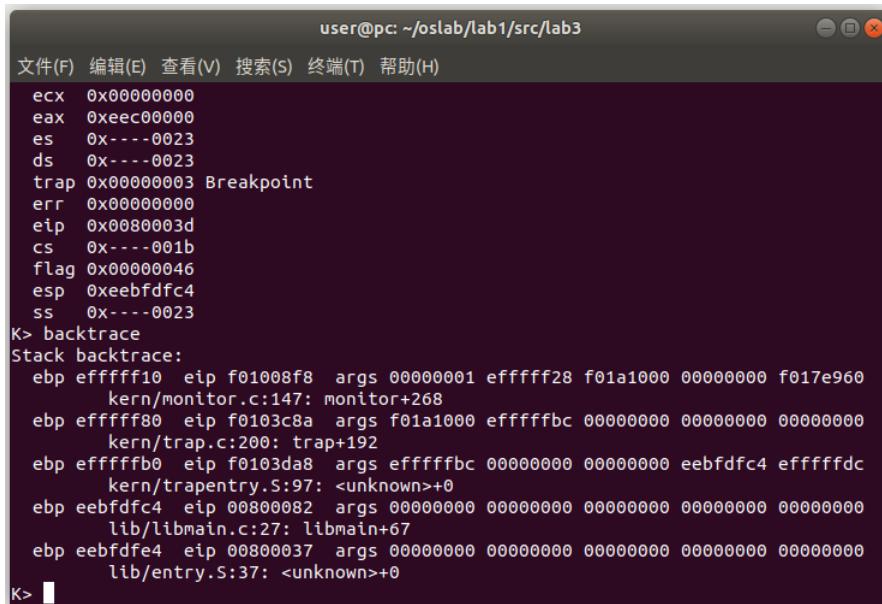
```
// Entry point - this is where the kernel (or our parent environment)
// starts us running when we are initially loaded into a new environment.
.text
.globl _start
_start:
    // See if we were started with arguments on the stack
    cmpl $USTACKTOP, %esp
    jne args_exist

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

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

Figure 70: 修改后的 lib/entry.S

这样，将`mon_backtrace()`改回去之后就不会再出现缺页错误了。



The screenshot shows a terminal window titled "user@pc: ~/oslab/lab1/src/lab3". The window contains the following text:

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

ecx 0x00000000
eax 0xeec00000
es 0x----0023
ds 0x----0023
trap 0x00000003 Breakpoint
err 0x00000000
eip 0x0080003d
cs 0x----001b
flag 0x00000046
esp 0xeebfdfc4
ss 0x----0023

K> backtrace
Stack backtrace:
    ebp efffff10 eip f01008f8 args 00000001 efffff28 f01a1000 00000000 f017e960
        kern/monitor.c:147: monitor+268
    ebp efffff80 eip f0103c8a args f01a1000 efffffb0 00000000 00000000 00000000
        kern/trap.c:200: trap+192
    ebp efffffb0 eip f0103da8 args efffffb0 00000000 00000000 eebfdfc4 efffffdc
        kern/trapentry.S:97: <unknown>+0
    ebp eebfdfc4 eip 00800082 args 00000000 00000000 00000000 00000000 00000000
        lib/libmain.c:27: libmain+67
    ebp eebfdfc4 eip 00800037 args 00000000 00000000 00000000 00000000 00000000
        lib/entry.S:37: <unknown>+0

K>
```

Figure 71: 修改后的内核输出

3.8 作业 9

3.8.1 题目原文

启动你的内核，运行 user/evilhello。你的环境将会崩溃，内核将会 panic 停止，你将看见以下信息：

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

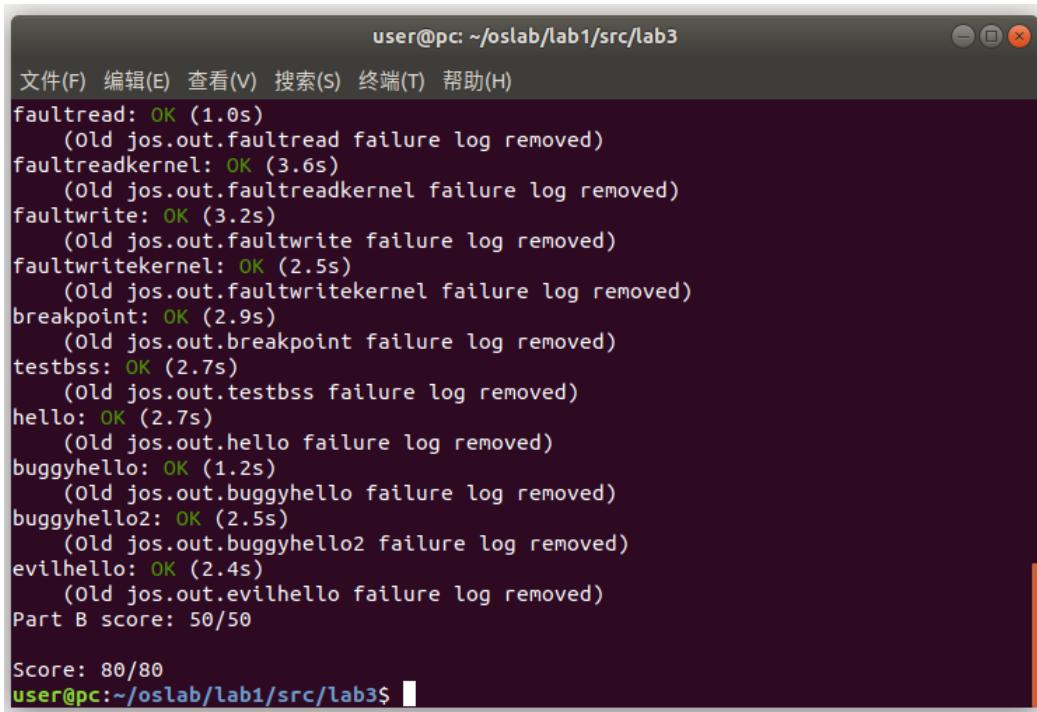
3.8.2 回答

相关代码在作业 8 中修改完毕，在终端中运行 make run-evilhello 命令。运行结果如下（环境崩溃，内核停止）：

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffb0
Incoming TRAP frame at 0xefbfffb0
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
```

Figure 72: make run-evilhello

至此，Lab 3 的 Part B 已完成。以下是在终端输入命令 make grade 之后得到的打分情况：



The screenshot shows a terminal window with the title "user@pc: ~/oslab/lab1/src/lab3". The window contains a list of test cases and their results:

```
faultread: OK (1.0s)
  (Old jos.out.faultread failure log removed)
faultreadkernel: OK (3.6s)
  (Old jos.out.faultreadkernel failure log removed)
faultwrite: OK (3.2s)
  (Old jos.out.faultwrite failure log removed)
faultwritekernel: OK (2.5s)
  (Old jos.out.faultwritekernel failure log removed)
breakpoint: OK (2.9s)
  (Old jos.out.breakpoint failure log removed)
testbss: OK (2.7s)
  (Old jos.out.testbss failure log removed)
hello: OK (2.7s)
  (Old jos.out.hello failure log removed)
buggyhello: OK (1.2s)
  (Old jos.out.buggyhello failure log removed)
buggyhello2: OK (2.5s)
  (Old jos.out.buggyhello2 failure log removed)
evilhello: OK (2.4s)
  (Old jos.out.evilhello failure log removed)
Part B score: 50/50

Score: 80/80
user@pc:~/oslab/lab1/src/lab3$
```

Figure 73: Part B 打分情况

4 总结

感觉 Lab 3 的作业量明显变大了。做完 Lab 3 以后，不仅深入理解了操作系统进程的创建和异常处理的机制，还巩固了 Lab 2 内存管理的内容。但是这里的与进程相关的知识也只是涉及到了一小部分，其他我一直搞不懂的信号量等等在这里没有涉及，因此还是要抽时间多看看书，多看看课件。还剩下最后一个 Lab 4，坚持就是胜利！