

操作系统实验报告



名称：操作系统 Lab3

学 院：计算机学院 网络空间安全学院

专 业：计算机科学与技术 信息安全

成 员：徐晖宇 付宇 顾知晨 姜奕兵 孙铭 肖阳

课程教师：蒲凌君

时 间：2019 年 11 月

小组成员分工

徐晖宇（1713666）：作业 2（后三个函数）+问题 2

付 宇（1612120）：作业 1+作业 2（前三个函数）+问题 1

顾知晨（1711323）：作业 3+作业 4

姜奕兵（1710218）：作业 8+作业 9

孙 铭（1711377）：作业 7+作业 8

肖 阳（1610292）：作业 5+作业 6

Make grade 检查

```

fy740@ubuntu: ~/OS/src/lab3
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/fy740/OS/src/lab3'
divzero: OK (1.3s)
softint: OK (0.8s)
badsegment: OK (0.9s)
Part A score: 30/30

faultread: OK (0.9s)
faultreadkernel: OK (1.8s)
faultwrite: OK (1.3s)
faultwritekernel: OK (1.7s)
breakpoint: OK (1.2s)
testbss: OK (1.7s)
hello: OK (2.3s)
buggyhello: OK (1.8s)
buggyhello2: OK (1.3s)
evilhello: OK (1.7s)
Part B score: 50/50

Score: 80/80
fy740@ubuntu:~/OS/src/lab3$

```

一、作业一

作业 1

Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENV`s (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.

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 `UENV`s 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。

首先，有必要了解一下 `pages` 和用户环境的数据结构（`inc/env.h`）：

```

struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;      // Next free Env
    envid_t env_id;            // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    enum EnvType env_type;      // Indicates special system environments
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;          // Number of times environment has run

    // Address space
    pde_t *env_pgdir;          // Kernel virtual address of page dir
};

```

结构体 `Env` 中，维护了当前环境的各种属性，实验书中已经详细介绍，在此补充了解以下内容：

`env_tf`: `TrapFrame` 定义在(`inc/trap.h`)中

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

该结构可以保存 `es`, `ds` 段寄存器的值以及中断号，值得注意的是在该结构中，既包含了 `x86` 硬件的信息，也保存了用户态的信息，以完成用户态向内核态的转变；`trapno`: 有关 `trap` 的标号，`env_link`: 注意是指向空闲列表第一片空闲区域的内容；`env_pgdir`: 注意保存的是该环境的虚拟地址。

题目中指出用户环境数组的内存分配和 `pages` 数组的内存分配方式类似，因此在此在 `mem_init()` 中有如下代码：

```
// 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));
```

第一句代码使用 `boot_alloc()` 为 `envs` 分配 `NENV` 个 `env` 结构的内存，并把首地址返回给 `envs` 变量。第二句初始化这 `NENV` 个 `env` 结构，将其数组表项都置为 0。

完成分配内存和初始化后，将逻辑地址 `UENVS` 映射到 `envs` 的物理地址：

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

该句代码同样参考 `pages` 的映射过程，第一个参数是页目录，第二个参数是映射的虚拟地址，第三个参数是映射地址的长度，第四个参数是 `envs` 对应的物

理地址，第五个参数 PTE_U 用来规定访问权限，为 1 则表示用户可读，为 0 则表示需要更高的权限。

二、作业二

作业 2

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

`env_init()`: Initialize all of the Env structures in the `envs` array and add them to the `env_free_list`. Also calls `env_init_percpu`, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

`env_setup_vm()`: Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

`region_alloc()`: Allocates and maps physical memory for an environment

`load_icode()`: You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.

`env_create()`: Allocate an environment with `env_alloc` and call `load_icode` 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,

(1) `env_init(void)` 初始化 `envs` 数组中的全部 Env 结构体，并将它们加入 `env_free_list` 中。还要调用 `env_init_percpu()`，这个函数会通过配置段硬件，将其分隔为特权等级 0 (内核) 和特权等级 3 (用户) 两个不同的段。

```
// Mark all environments in 'envs' as free, set their env_ids to 0,
// and insert them into the env_free_list.
// Make sure the environments are in the free list in the same order
// they are in the envs array (i.e., so that the first call to
// env_alloc() returns envs[0]).
//
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int i;
    env_free_list = NULL;
    for(i=NENV-1; i>=0; i--){
        envs[i].env_id = 0;
        envs[i].env_link = env_free_list;
        env_free_list = &envs[i];
    }
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

`env_free_list` 应该指向 `envs[0]`，也就是 `envs` 数组的首值，因此，初始化循环从后往前开始，通过 `env_link` 就能将链表串接起来。`env_init_percpu()` 加载全局描述符表并且初始化段寄存器 `gs`, `fs`, `es`, `ds`, `ss`。GDT 定义在 `kern/env.c` 中。

(2) `env_setup_vm()` 为新的进程分配一个页目录，并初始化新进程的地址空间

对应的内核部分。

```
// LAB 3: Your code here.
p->pp_ref++;
e->env_pgdir = (pde_t *) page2kva(p);           //刚分配的物理页作为页目录使用
memcpy(e->env_pgdir, kern_pgdir, PGSIZE);       //继承内核页目录
```

static int env_setup_vm(struct Env *e), 参数为 Env 结构指针。返回值：0 表示成功，-E_NO_MEM 表示失败，没有足够物理地址。pp_ref++ 确保之后的 env_free 正确工作。上述作业提示：//Can you use kern_pgdir as a template? Hint: Yes。这里做的是地址的静态映射，即并没有调用 page_insert() 分配物理空间，而是只分配了虚拟的地址，所以这里我们使用 memcpy 继承内核页目录。

(3) region_alloc() 给长度为 len 的虚拟内存中的地址分配页表，并映射到 e->env_pgdir 所对应的页目录中。

参数 struct Env *e 为用户环境，void *va 为虚拟地址，size_t len 为长度。函数作用：操作 e->env_pgdir，为[va, va+len) 分配物理空间。

注意这里的 va 不一定是按照 PGSIZE 进行对齐的，所以要进行处理；这里插入点时候，让用户可以写，设置 PTE_U, PTE_W 为 1。

```
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.)
    //地址对齐
    void* start = (void *)ROUNDDOWN((uint32_t)va, PGSIZE);
    void* end = (void *)ROUNDUP((uint32_t)va+len, PGSIZE);
    struct PageInfo *p = NULL;
    void* i;
    int r;
    for(i=start; i<end; i+=PGSIZE){
        p = page_alloc(0);
        if(p == NULL)
            panic("allocation failed.");

        r = page_insert(e->env_pgdir, p, i, PTE_W | PTE_U);
        if(r != 0) {
            panic("region alloc:%e", r);
        }
    }
    // 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!)
}
```

p = page_alloc(0); // 分配一个物理页。page_insert(e->env_pgdir, p, i, PTE_W | PTE_U); // 建立逻辑地址 i 到物理页 p 的映射关系，PTE_W | PTE_U 保证页对于用户和内核是可写的。

(4) `load_icode()`:这个函数是向 `env` 中加载一个 `elf` 文件, 仿照 `boot/main.c` 中加载 `elf` 的方法完成该部分代码;

```
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 */;
}
```

`load_icode()`这个函数设置了 ELF 的二进制映像, 并且将映像内容读入地址空间。因为题目说明中, JOS 没有文件系统, 所以为了测试用户程序, 我们要将用户的程序和我们的内核地址链接到一起, 即地址要紧挨在内核的后面, 所以该函数就是将嵌入到内核的用户程序取出来并映射到相应的虚拟空间, 这里的 `binary` 指针就是内核开始位置的虚拟地址; 我们首先介绍一下 ELF。


```

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;
};

```

`e_type` 它标识的是该文件的类型。

`e_machine` 表明运行该程序需要的体系结构。

`e_version` 表示文件的版本。

`e_entry` 程序的入口地址。

`e_phoff` 表示 Program header table 在文件中的偏移量（以字节计数）。

`e_shoff` 表示 Section header table 在文件中的偏移量（以字节计数）。

`e_flags` 对 IA32 而言，此项为 0。

`e_ehsize` 表示 ELF header 大小（以字节计数）。

`e_phentsize` 表示 Program header table 中每一个条目的大小。

`e_phnum` 表示 Program header table 中有多少个条目。

`e_shentsize` 表示 Section header table 中的每一个条目的大小。

`e_shnum` 表示 Section header table 中有多少个条目。

`e_shstrndx` 包含节名称的字符串是第几个节（从零开始计数）。

开头的 `e_magic` 必须要和 `ELF_MAGIC` 相等，否则将不是有效的 ELF 文件，所以在函数开始需要我们先进行判断。

之后我们看一下在 `bootmain()` 中用到的结构 `Proghdr`。


```

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;
};

```

`p_type` 表示该段的类型。

`P_offset` 表示该段在文件镜像中的偏移量。

`p_va` 表示该段在内存中的虚拟地址。

`p_pa` 表示在使用相对物理地址的系统中，表示该段在内存中的物理地址。

`p_filesz` 表示该段在文件镜像中的大小（以 bytes 计，可以为 0）。

`p_memsz` 表示该段在内存中的大小（以 bytes 计，可以为 0）。

这里面注意，`p_filesz` 和 `p_memsz` 的区别为：`bss` 段在 `elf` 文件之中，不会被分配内存，但是在实际载入的时候需要被初始化为 0，所以这里的意思比较明确就是前 `p_filesz` 的内存存放 `bss` 段以外的内容，而 `p_filesz` 到 `p_memsz` 表示 `bss` 段的内容，在这里需要被置为 0；

在一个用户环境中（Env），`,env_tf_eip` 表示用户程序的入口地址；

操作系统会存储一张全局描述符表 `GDT` 在内存中，而这张 `GDT` 是通过 `GDTR` 寄存器指出它的位置的。但是，`GDTR` 中的地址是线性地址，也就是说，如果开启了分页机制的话，需要经过 `CR3` 寄存器的帮助，才能映射到物理内存地址，从而找到 `GDT`。也就是说，我们需要使用 `lcr3()` 函数来切换到内核态对存储进行改写；

下面来看具体的函数内容，函数中几个参数的定义参照了 `bootmain()`，比如 `ph` 和 `eph`。因为需要我们将每个程序段都加载到 `ELF` 标题中指定的地址的虚拟内存中，所以我们需要先判断其是否是一个有效的 `ELF` 文件。之后确定要映射的段为 `ph-eph`。同时我们需要利用 `lcr3` 函数将页目录物理地址加载到 `cr3` 寄存器，实现页目录的切换。根据要求只加载 `ph->p_type == ELF_PROG_LOAD` 的段，并且映射的范围也在注释中要求了，利用 `memset` 函数将剩余内存字节清 0。同

时参考下面的代码更改函数的入口点。最后利用 `region_alloc()` 在虚拟地址 `USTACKTOP-PGFSIZE` 处映射程序初始堆栈的一个页面。

```
// LAB 3: Your code here.
struct Elf* header = (struct Elf*)binary;

if(header->e_magic != ELF_MAGIC) {
    panic("load_icode failed: The binary we load is not elf.\n");
}

if(header->e_entry == 0){
    panic("load_icode failed: The elf file can't be excuterd.\n");
}

e->env_tf.tf_eip = header->e_entry;

lcr3(PADDR(e->env_pgdir));

struct Proghdr *ph, *eph;
ph = (struct Proghdr*)((uint8_t *)header + header->e_phoff);
eph = ph + header->e_phnum;
for(; ph < eph; ph++) {
    if(ph->p_type == ELF_PROG_LOAD) {
        if(ph->p_memsz - ph->p_filesz < 0) {
            panic("load_icode failed : p_memsz < p_filesz.\n");
        }

        region_alloc(e, (void *)ph->p_va, ph->p_memsz);
        memmove((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
        memset((void *) (ph->p_va + ph->p_filesz), 0, ph->p_memsz - ph->p_filesz);
    }
}

// Now map one page for the program's initial stack
// at virtual address USTACKTOP - PGFSIZE.
// LAB 3: Your code here.
region_alloc(e, (void *) (USTACKTOP-PGFSIZE), PGFSIZE);
```

`env_creat()`调用 `env_alloc` 和 `load_icode` 创建一个 `env` 并加载到 `elf` 文件，这函数调用了 `env_alloc()`和 `load_icode()`，来实现新进程的创建。接受内核传入的用户程序所在的内核地址 `binary`，然后为其创建用户进程空间，并且将其载入到相应的虚拟地址上。其中 `env_alloc` 中的参数 0，是实现创建新的 `env`，保证其用户环境为 0。

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

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

`env_run()`函数的功能完成将 `e` 这个环境设置为开始运行，并将现有运行的进程放入 `runnable` 中；该函数首先确定是否当前环境有其他的 `env` 在运行中，如果有就将其状态设置为 `ENV_RUNNABLE`，之后将当前环境设置为 `e`，将 `e` 的状态设置为运行中，更新计数器 `env_runs`。并且利用 `lcr3()`把页目录加载到 `cr3` 寄存器。最后使用 `env_pop_tf()`恢复环境的寄存器(从 `env_tf` 中加载新环境)

```
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.
    if(curenv != NULL && curenv->env_status == ENV_RUNNING) {
        curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));
    // 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.
    env_pop_tf(&curenv->env_tf);

    panic("env_run not yet implemented");
}
```

三、作业三

首先 IDT 的数据结构，定义在 `kern/trap.c` 中

```
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
    sizeof(idt) - 1, (uint32_t) idt
};
```

IDT-pd 是系统寄存器 IDTR 的对应结构，门描述符 `struct Gatedesc` 定义在 `inc/mmu.h` 中，使用 `setgate` 来设置一个特定的描述符。

```
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
};

#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; \
}

// Set up a call gate descriptor.
#define SETCALLGATE(gate, 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 = STS_CG32; \
    (gate).gd_s = 0; \
    (gate).gd_dpl = (dpl); \
    (gate).gd_p = 1; \
    (gate).gd_off_31_16 = (uint32_t) (off) >> 16; \
}

```

在 kern/trapentry.S 中定义好每个中断处理程序，在 trap.c 的 idt_init() 中将第一步定义好的中断处理程序安装 IDT。

每个 interrupt handler 都必须在内核栈中设置 Trapframe 的布局结构，然后将这个结构传给 trap() 进行进一步处理，最后 trap_dispatch() 中进行具体的中断处理程序的分发。

JOS 提供了两个宏：接收一个函数名和对应处理的中断向量编号，然后定义出一个相应同名的中断处理程序。这样的中断向量程序的执行流程就是向栈里压入相关的错误和中断编号，跳转到 _alltraps 来执行共有的部分。（把 Trapframe

剩下的那些结构在栈中设置好)，我们的中断处理程序需要在系统没有放入错误码时手动不起这个空间。TRAPHANDLER_NOEC 就是将如此操作的。

```
#define TRAPHANDLER(name, num) \
    .globl name;          /* define global symbol for 'name' */ \
    .type name, @function; /* symbol type is function */ \
    .align 2;             /* align function definition */ \
    name:                 /* function starts here */ \
    pushl $(num);         \
    jmp _alltraps

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

.text
TRAPHANDLER_NOEC(t_divide, T_DIVIDE)
TRAPHANDLER_NOEC(t_debug, T_DEBUG)
TRAPHANDLER_NOEC(t_nmi, T_NMI)
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT)
TRAPHANDLER_NOEC(t_oflow, T_OFLOW)
TRAPHANDLER_NOEC(t_bound, T_BOUND)
TRAPHANDLER_NOEC(t_illop, T_ILLOP)
TRAPHANDLER_NOEC(t_device, T_DEVICE)
TRAPHANDLER(t_dblflt, T_DBLFLT)
TRAPHANDLER(t_tss, T_TSS)
TRAPHANDLER(t_segnp, T_SEGNP)
TRAPHANDLER(t_stack, T_STACK)
TRAPHANDLER(t_gpflt, T_GPFLT)
TRAPHANDLER(t_pgflt, T_PGFLT)
TRAPHANDLER_NOEC(t_fperr, T_FPERR)
TRAPHANDLER(t_align, T_ALIGN)
TRAPHANDLER_NOEC(t_mchk, T_MCHK)
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR)
/*
```

```

* Lab 3: Your code here for _alltraps
*/

_alltraps:
    pushl %ds
    pushl %es
    pushal

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

    push %esp
    call trap

```

TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)

我们安装 IDT 表，看到 kern/trap.c 中的 idt_init()

第三个参数 cs，设置为内核的代码段 GD_KT,最后一个参数为用户特权级的设置。

```

static struct Trapframe *last_tf;

/* Interrupt descriptor table. (Must be built at run time because
 * shifted function addresses can't be represented in relocation record
 * s.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
    sizeof(idt) - 1, (uint32_t) idt
};

void t_divide();
void t_debug();
void t_nmi();
void t_brkpt();
void t_oflow();
void t_bound();
void t_illop();
void t_device();
void t_dblflt();
void t_tss();

```

```

void t_segnp();
void t_stack();
void t_gpflt();
void t_pgflt();
void t_fperr();
void t_align();
void t_mchk();
void t_simderr();
void t_syscall();

static const char *trapname(int trapno)
{
    static const char * const excnames[] = {
        "Divide error",
        "Debug",
        "Non-Maskable Interrupt",
        "Breakpoint",
        "Overflow",
        "BOUND Range Exceeded",
        "Invalid Opcode",
        "Device Not Available",
        "Double Fault",
        "Coprocessor Segment Overrun",
        "Invalid TSS",
        "Segment Not Present",
        "Stack Fault",
        "General Protection",
        "Page Fault",
        "(unknown trap)",
        "x87 FPU Floating-Point Error",
        "Alignment Check",
        "Machine-Check",
        "SIMD Floating-Point Exception"
    };

    if (trapno < sizeof(excnames)/sizeof(excnames[0]))
        return excnames[trapno];
    if (trapno == T_SYSCALL)
        return "System call";
    return "(unknown trap)";
}

void

```



```

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

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, t_nmi, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 3);
    SETGATE(idt[T_OFLOW], 0, GD_KT, t_oflow, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, t_bound, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, t_illop, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, t_device, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, t_dblflt, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, t_tss, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, t_segnp, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, t_stack, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, t_gpflt, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, t_pgflt, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, t_fperr, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, t_align, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, t_mchk, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT, t_simderr, 0);
    SETGATE(idt[T_SYSCALL], 0, GD_KT, t_syscall, 3);
    // Per-CPU setup
    trap_init_percpu();
}

// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;

    // Initialize the TSS slot of the gdt.
    gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                               sizeof(struct Taskstate) - 1, 0);
    gdt[GD_TSS0 >> 3].sd_s = 0;

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    ltr(GD_TSS0);

```

```

// Load the IDT
lidt(&idt_pd);
}

```

中断被捕获后会转到 Kern/trapentry.S 中的 routine divide(),然后跳到 alltraps,kern/trap.c 中的 trap();程序会进入 trap_dispatch()打印出寄存器信息。

```

static void
trap_dispatch(struct Trapframe *tf)
{
    int32_t ret_code;
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch(tf->tf_trapno) {
        case (T_PGFLT):
            page_fault_handler(tf);
            break;
        case (T_BRKPT):
            print_trapframe(tf);
            monitor(tf);
            break;
        case (T_DEBUG):
            monitor(tf);
            break;
        case (T_SYSCALL):
            ret_code = syscall(
                tf->tf_regs.reg_eax,
                tf->tf_regs.reg_edx,
                tf->tf_regs.reg_ecx,
                tf->tf_regs.reg_ebx,
                tf->tf_regs.reg_edi,
                tf->tf_regs.reg_esi);
            tf->tf_regs.reg_eax = ret_code;
            break;
        default:
            // Unexpected trap: The user process or the kernel has a bug.

            print_trapframe(tf);
            if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
            else {
                env_destroy(curenv);
                return;
            }
    }
}

```

```

    }
}

void
trap(struct Trapframe *tf)
{
    // The environment may have set DF and some versions
    // of GCC rely on DF being clear
    asm volatile("cld" ::: "cc");

    // Check that interrupts are disabled. If this assertion
    // fails, DO NOT be tempted to fix it by inserting a "cli" in
    // the interrupt path.
    assert(!(read_eflags() & FL_IF));

    cprintf("Incoming TRAP frame at %p\n", tf);

    if ((tf->tf_cs & 3) == 3) {
        // Trapped from user mode.
        assert(curenv);

        // Copy trap frame (which is currently on the stack)
        // into 'curenv->env_tf', so that running the environment
        // will restart at the trap point.
        curenv->env_tf = *tf;
        // The trapframe on the stack should be ignored from here on.
        tf = &curenv->env_tf;
    }

    // Record that tf is the last real trapframe so
    // print_trapframe can print some additional information.
    last_tf = tf;

    // Dispatch based on what type of trap occurred
    trap_dispatch(tf);

    // Return to the current environment, which should be running.
    assert(curenv && curenv->env_status == ENV_RUNNING);
    env_run(curenv);
}

```

在 inc/memlayout.h 中可以看到 KSTACKTOP 上的空间为系统页目录 VPT, JOS 在载入 softint 时会分配物理页放入 elf 文件, 我们查看一下知道其 stab 映射

发生了，使 VPT 上在发生 Page fault 中断的时候，VPT 上就没有相应的映射页了。

```
/* Place debugging symbols so that they can be found by
 * the kernel debugger.
 * Specifically, the four words at 0x200000 mark the beginning of
 * the stabs, the end of the stabs, the beginning of the stabs
 * string table, and the end of the stabs string table, respectively.
 */

.stab_info 0x200000 : {
    LONG(__STAB_BEGIN__);
    LONG(__STAB_END__);
    LONG(__STABSTR_BEGIN__);
    LONG(__STABSTR_END__);
}

.stab : {
    __STAB_BEGIN__ = DEFINED(__STAB_BEGIN__) ? __STAB_BEGIN__ : .;
    *(.stab);
    __STAB_END__ = DEFINED(__STAB_END__) ? __STAB_END__ : .;
    BYTE(0)      /* Force the linker to allocate space
                  for this section */
}

.stabstr : {
```

```

    __STABSTR_BEGIN__ = DEFINED(__STABSTR_BEGIN__) ?
__STABSTR_BEGIN__ : .;

    *(.stabstr);

    __STABSTR_END__ = DEFINED(__STABSTR_END__) ? __STABSTR_END__ : .;

    BYTE(0)    /* Force the linker to allocate space
                for this section */

}

/DISCARD/ : {

    *(.eh_frame .note.GNU-stack .comment)

}
}

```

#结论

在这个练习中定义了一系列中断入口函数，用好 JOS 提供的宏可以轻松完成这些工作。

1. 使用宏 TRAPHANDLER 和 TRAPHANDLER_NOEC 定义一系列中断入口地址。
1. 因为要切换到内核代码，所以需要将 ES 和 DS 设置成内核的 ES 和 DS。
2. 使用 pushal 命令来压入中断调用帧(trapframe)，然后将这栈中的这个帧的地址传给函数 trap。
3. 在 trap_init 中需要设置 IDT。

#分析

下面这两个结构体定义对于理解中断很有用：

```

...
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;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel
    */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
...

```

从 `TrapFrame` 的注释中可以得知，在进入中断入口时，内核栈中已经由 CPU 压入了 `ESP` (用户态进入内核态才有)，`EFLAGS`，`CS`，`EIP`，`ERR_NO`。

进入中断入口之后，如果该中断没有错误码，则压入一个 `0` 以保持 `TrapFrame` 的致，接着是压入中断号方便后面 `trap_dispatch` 函数处理不同的中断。因为可能要更改代码段和数据段描述符，所以将 `es` 和 `ds` 也压入堆栈。最后是调用 `pushal` 压入当前的寄存器值。

读代码的时候需要注意的是堆栈的增长方向是从大地址到小地址，而结构体的 `field` 从上到下是从小地址到大地址。因此最先压入堆栈的是结构体中的最后一个字段。且压栈和出栈都是以 4 个字节为一个块进入操作的，对于不足 4 字节的数据，会被填充。

中断入口的代码如下：

```

...

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

```

```

...

.globl _alltraps
_alltraps:
pushl %ds

pushl %es
pushal

movw $GD_KD, %ax
movw %ax, %ds
movw %ax, %es

pushl %esp /* trap(%esp) */
call trap

```

四、问题一

问题 1

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)?

对每一个中断/异常都分别给出中断处理函数的目的是什么？换句话说，如果所有的中断都交给同一个中断处理函数处理，现在我们实现的哪些功能就没办法实现了？

打分脚本希望它产生一个一般保护错(trap13)，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？

(1) 不同的中断或异常不能只由一个处理函数处理，因为每个中断或异常都是不同的，有些是错误引起的（如除 0 的错误）、有些可能是由 I/O 操作引起的，像 I/O 操作引起的中断处理完后程序还要继续接着执行，而错误引起的可能就导致程序结束了。如果仅用一个处理函数，是无法兼顾各种情况的。

(2) softint.c 代码：


```

softint.c (~OS/src/lab3/user) - gedit
Open Save
// buggy program - causes an illegal software interrupt
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    asm volatile("int $14");    // page fault
}

```

从上述代码可以看出，程序希望产生一个缺页异常(int \$14)。

grade-lab3 代码如下：

```

grade-lab3 (~OS/src/lab3) - gedit
Open Save
@test(10)
def test_softint():
    r.user_test("softint")
    r.match('Welcome to the JOS kernel monitor!',
            'Incoming TRAP frame at 0xeffffbc',
            'TRAP frame at 0xf.....',
            'trap 0x0000000d General Protection',
            'eip 0x008.....',
            'ss 0x----0023',
            '.00001000. free env 0000100')

```

就上述评分标准的代码看来，测试 softint 时产生的是 General Protection Exception——trap13。

当前的系统运行在用户态下，权限级别为 3，而 INT 指令为系统指令，权限级别为 0。当用户态调用内核态的系统指令，会引发 General Protection Exception。如果允许调用，那么这就存在巨大的漏洞，若是被恶意程序一直调用该中断，就会不断地处理缺页异常，导致可用的内存页变少，浪费了资源。

五、作业四

作业 4：

在 trap_dispatch 中判断一下当前的异常是不是缺页错误，如果是的话，就把这个异常传递给 page_fault_handler 来处理。

```

static void
trap_dispatch(struct Trapframe *tf)
{
    int32_t ret_code;
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch(tf->tf_trapno) {
        case (T_PGFLT):
            page_fault_handler(tf);

```

```

        break;
    case (T_BRKPT):
        print_trapframe(tf);
        monitor(tf);
        break;
    case (T_DEBUG):
        monitor(tf);
        break;
    case (T_SYSCALL):
        ret_code = syscall(
            tf->tf_regs.reg_eax,
            tf->tf_regs.reg_edx,
            tf->tf_regs.reg_ecx,
            tf->tf_regs.reg_ebx,
            tf->tf_regs.reg_edi,
            tf->tf_regs.reg_esi);
        tf->tf_regs.reg_eax = ret_code;
        break;
    default:
        // Unexpected trap: The user process or the kernel has a bug.

        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
            panic("unhandled trap in kernel");
        else {
            env_destroy(curenv);
            return;
        }
    }
}
}

```

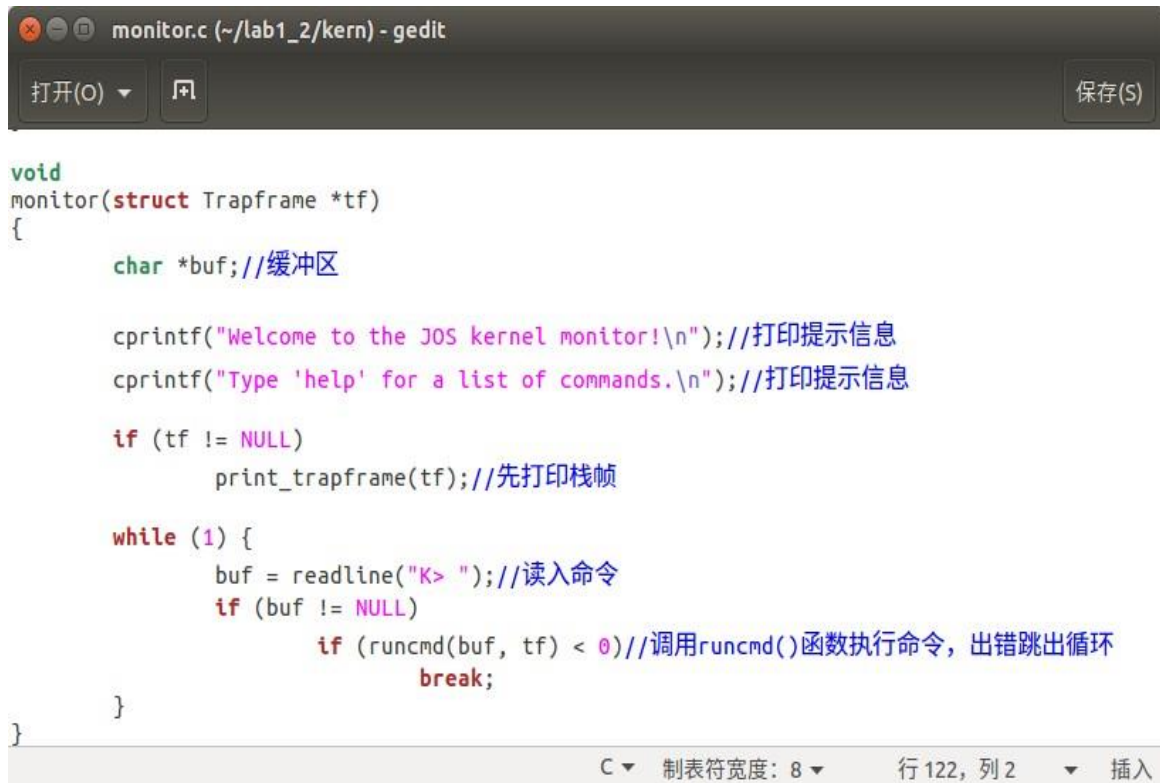
六、作业五

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.

题目要求修改 `trap_dispatch()` 函数来让断点异常唤醒内核监视器。思路和作业 4 一样的，需要在 `trap_dispatch()` 函数中增加一个断点异常的分支。如果是断点异常那么就唤醒内核监视器。增加的代码如下：

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

调用内核监视器的 `monitor()` 函数定义在 `kern/monitor.c` 中，代码如下：



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

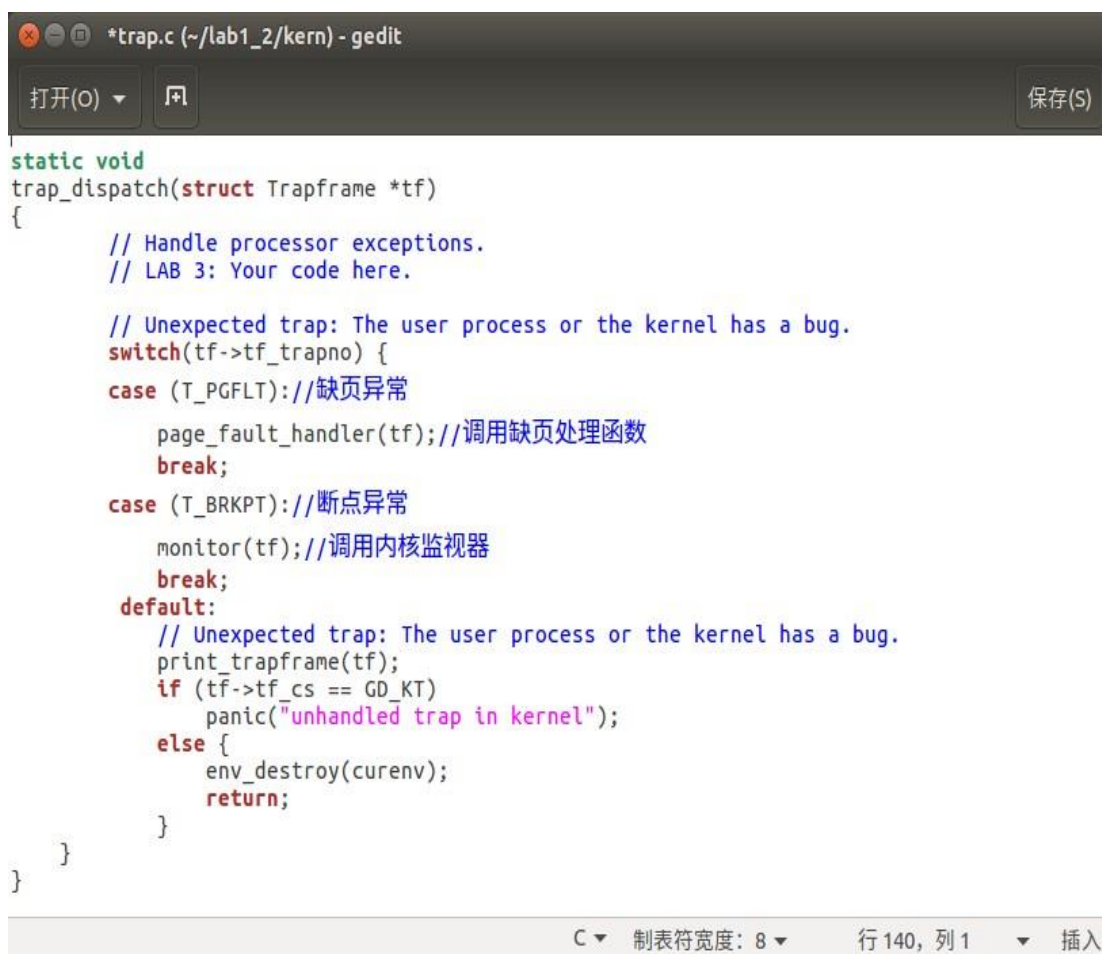
    cprintf("Welcome to the JOS kernel monitor!\n"); //打印提示信息
    cprintf("Type 'help' for a list of commands.\n"); //打印提示信息

    if (tf != NULL)
        print_trapframe(tf); //先打印栈帧

    while (1) {
        buf = readline("K> "); //读入命令
        if (buf != NULL)
            if (runcmd(buf, tf) < 0) //调用runcmd()函数执行命令，出错跳出循环
                break;
    }
}
```

图 11: `monitor()` 函数

修改后的 `trap_dispatch()` 函数：



```

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

    // Unexpected trap: The user process or the kernel has a bug.
    switch(tf->tf_trapno) {
        case (T_PGFLT): //缺页异常
            page_fault_handler(tf); //调用缺页处理函数
            break;
        case (T_BRKPT): //断点异常
            monitor(tf); //调用内核监视器
            break;
        default:
            // Unexpected trap: The user process or the kernel has a bug.
            print_trapframe(tf);
            if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
            else {
                env_destroy(curenv);
                return;
            }
    }
}

```

图 12: trap_dispatch() 函数

七、作业六

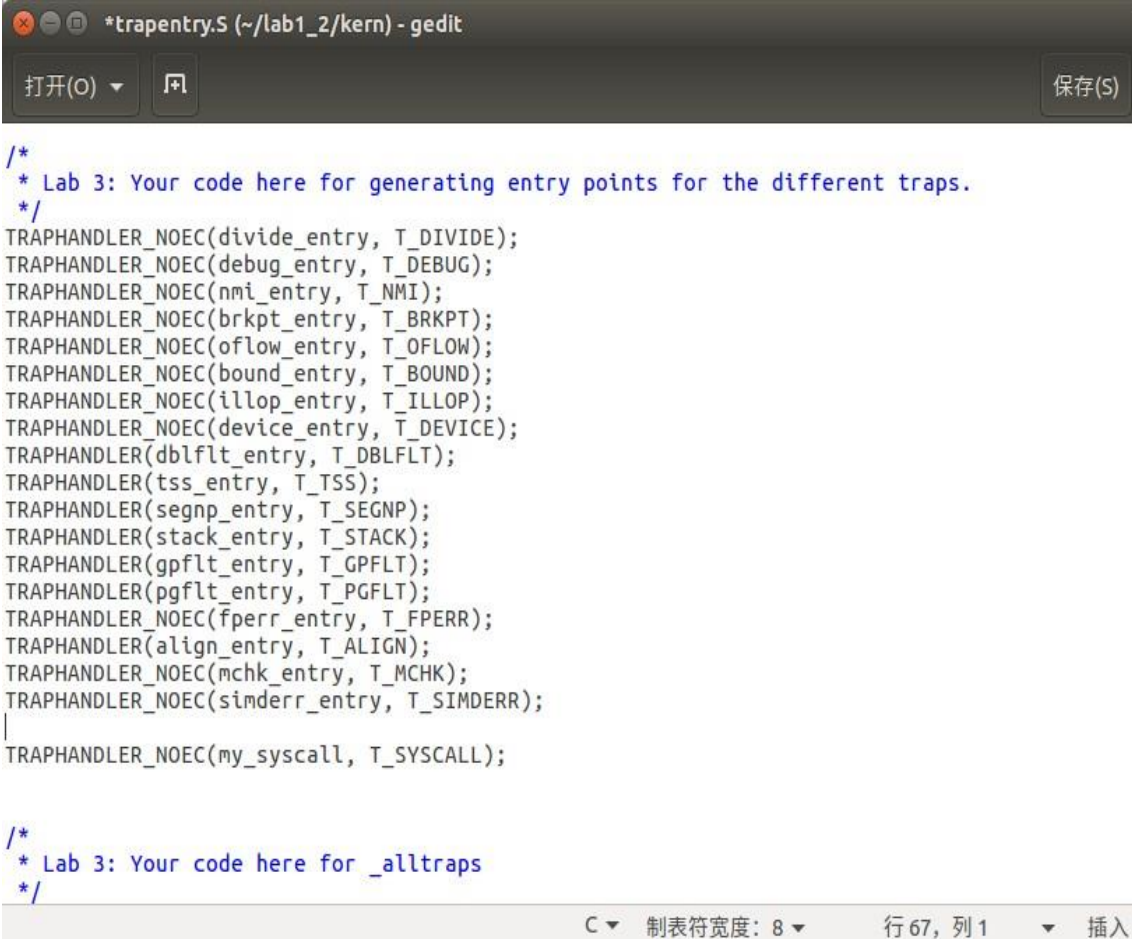
在内核中为中断 T_SYSCALL 添加一个处理程序。

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_INVALID。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。系统调用的中断是 T_SYSCALL, 即 0x30, 每当用户态需要系统调用时, 就会执行 int 0x30 指令。所以我们需要为这个中断号去写一个处理函数。

首先需要在 kern/trapentry.S 中为这个中断号声明一个处理函数。结果如图



```

/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
TRAPHANDLER_NOEC(divide_entry, T_DIVIDE);
TRAPHANDLER_NOEC(debug_entry, T_DEBUG);
TRAPHANDLER_NOEC(nmi_entry, T_NMI);
TRAPHANDLER_NOEC(brkpt_entry, T_BRKPT);
TRAPHANDLER_NOEC(oflow_entry, T_OFLOW);
TRAPHANDLER_NOEC(bound_entry, T_BOUND);
TRAPHANDLER_NOEC(illop_entry, T_ILLOP);
TRAPHANDLER_NOEC(device_entry, T_DEVICE);
TRAPHANDLER(dblflt_entry, T_DBLFLT);
TRAPHANDLER(tss_entry, T_TSS);
TRAPHANDLER(segnp_entry, T_SEGNP);
TRAPHANDLER(stack_entry, T_STACK);
TRAPHANDLER(gpflt_entry, T_GPFLT);
TRAPHANDLER(pgflt_entry, T_PGFLT);
TRAPHANDLER_NOEC(fperr_entry, T_FPERR);
TRAPHANDLER(algn_entry, T_ALIGN);
TRAPHANDLER_NOEC(mchk_entry, T_MCHK);
TRAPHANDLER_NOEC(simderr_entry, T_SIMDERR);
|
TRAPHANDLER_NOEC(my_syscall, T_SYSCALL);

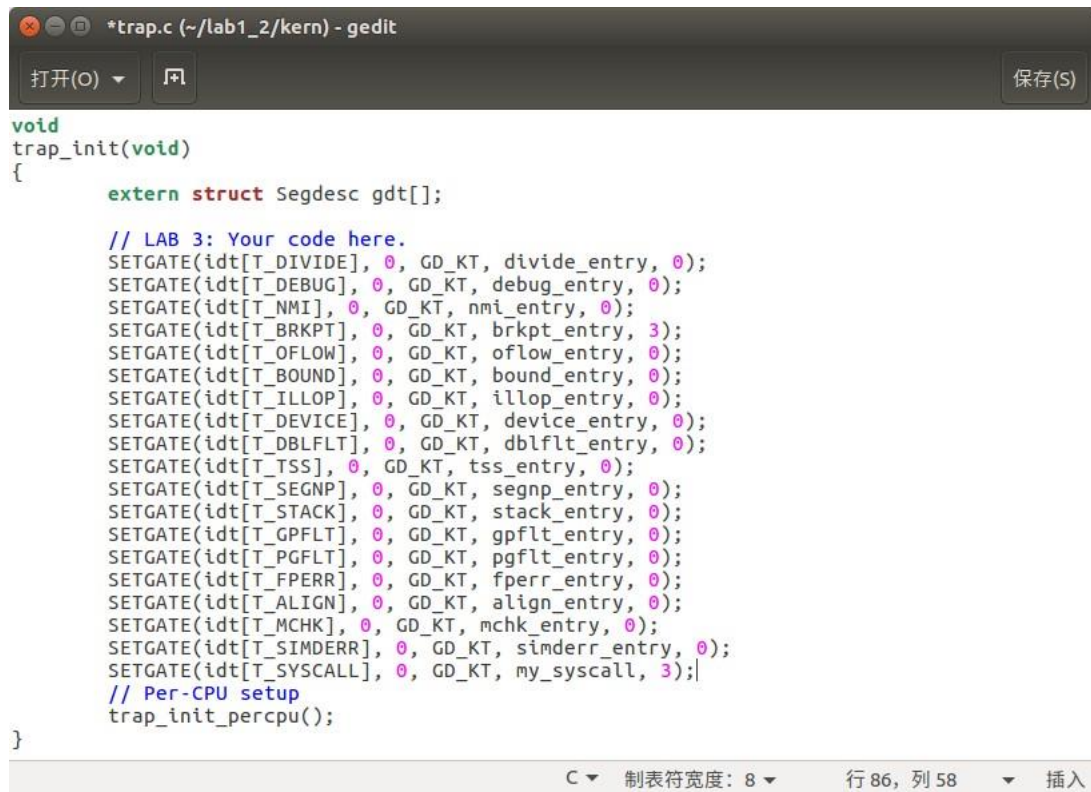
/*
 * Lab 3: Your code here for _alltraps
 */

```

图 13: 为 T_SYSCALL 声明中断处理函数 my_syscall

然后在 trap.c 中声明这个函数, 并在 trap_init() 函数中为它注册。增加的代码如下:

```
void my_syscall();
```



```

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

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, divide_entry, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, debug_entry, 0);
    SETGATE(idt[T_NMI], 0, GD_KT, nmi_entry, 0);
    SETGATE(idt[T_BRKPT], 0, GD_KT, brkpt_entry, 3);
    SETGATE(idt[T_OFLOW], 0, GD_KT, oflow_entry, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT, bound_entry, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT, illop_entry, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT, device_entry, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT, dblflt_entry, 0);
    SETGATE(idt[T_TSS], 0, GD_KT, tss_entry, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT, segnp_entry, 0);
    SETGATE(idt[T_STACK], 0, GD_KT, stack_entry, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT, gpflt_entry, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT, pgflt_entry, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT, fperr_entry, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT, align_entry, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT, mchk_entry, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT, simderr_entry, 0);
    SETGATE(idt[T_SYSCALL], 0, GD_KT, my_syscall, 3);
    // Per-CPU setup
    trap_init_percpu();
}

```

实现系统调用，需要修改 trap_dispatch() 函数。

```

static void
trap_dispatch(struct Trapframe *tf)
{
    int32_t ret_code; //返回值
    switch(tf->tf_trapno)
    {
        case T_PGFLT:
            page_fault_handler(tf);
            break;
        case T_BRKPT:
            monitor(tf);
            break;
        case T_SYSCALL:
            ret_code = syscall(//获取返回值
                               tf->tf_regs.reg_eax, //传入系统调用编号
                               tf->tf_regs.reg_edx, //传入参数
                               tf->tf_regs.reg_ecx, //传入参数
                               tf->tf_regs.reg_ebx, //传入参数
                               tf->tf_regs.reg_edi, //传入参数
                               tf->tf_regs.reg_esi); //传入参数
            tf->tf_regs.reg_eax = ret_code; //返回值写回%eax
            break;
        default:
            // Unexpected trap: The user process or the kernel has a bug.
            print_trapframe(tf);
            if (tf->tf_cs == GD_KT)
                panic("unhandled trap in kernel");
            else {
                env_destroy(curenv);
                return;
            }
    }
}

```

C 制表符宽度: 8 行 161, 列 68 插入

然后修改 kern/syscall.c 中的 syscall() 函数

八、作业七

作业 7

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。

从文档前述内容中可以知道，启动用户模式的流程如下。

首先，用户程序在 lib/entry.S 的顶部开始运行，该文件中已经定义了 envs 来指向 UENVS，lib/entry.S 代码如下。代码中.set envs, UENVS 即为定义 envs 指向 UENVS。

```

#include <inc/mmu.h>
#include <inc/memlayout.h>

.data
// Define the global symbols 'envs', 'pages', 'uvpt', and 'uvpd'
// so that they can be used in C as if they were ordinary global
arrays.
    .globl envs
    .set envs, UENVS
    .globl pages
    .set pages, UPAGES
    .globl uvpt
    .set uvpt, UVPT
    .globl uvpd
    .set uvpd, (UVPT+(UVPT>>12)*4)

```

其次，进行一些设置之后，这部分代码会调用 lib/libmain.c 中的 libmain() 函数，如 pdf 中所述，这个函数负责初始化全局的指向这个程序在 envs[] 数组中的 Env 结构 env 指针，对应 lib/libmain.c 中 libmain() 函数代码如下。

```

void
libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = 0;

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

    // call user main routine
    umain(argc, argv);

    // exit gracefully
    exit();
}

```

之后，libmain 会调用 umain()，运行用户程序并打印结果之后，会尝试访问 thisenv->env_id。值得指出，umain() 函数在 inc/lib.h 文件中声明，声明方法如下：

```

// main user program
void    umain(int argc, char **argv);

```

在 libmain() 函数中，argc 为整型变量，用来统计程序运行时发送给函数的命令行参数的个数，char** 型的 argv[]，为字符串数组，用来存放指向的字符串参数的指针数组，每一个元素指向一个参数。各成员含义如下：

argv[0] 指向程序运行的全路径名；

`argv[1]`指向在 DOS 命令行中执行程序名后的第一个字符串；

`argv[2]`指向执行程序名后的第二个字符串；

`argv[3]`指向执行程序名后的第三个字符串；

`argv[argc]`为 NULL。

在代码中，如果 `argc` 大于 0，`argv[0]`将指向程序运行的全路径名传入到 `binaryname` 中，`binaryname` 变量在 `libmain.c` 文件中定义，个人认为该变量作用是在为了后续程序中，将 utf-8 编码的全路径名转化成二进制格式，便于后续系统识别该路径并对路径下文件进行读写操作。之后将 `argc` 和 `argv` 传入到 `umain()` 函数中，作用是唤起用户主线程。编译用户目录下的文件。

编译 `user/hello.c` 后，会尝试调用 `sys_env_destroy()`，将用户环境 `env` 数组中的内容释放掉，然后执行 `exit` 退出。

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

void
exit(void)
{
    sys_env_destroy(0);
}
```

明确上述过程之后，显而易见，若要输出 `hello world` 以及用户环境，需要对 `libmain()`函数进行修改，原先代码中 `thisenv = 0`，由于之前没有设置 `thisenv` 的值，所以运行到 `hello` 的第二句时会出现错误，这里根据 `id` 取出索引，然后找到相应 `env`。代码修改如下。

```
/* 该函数负责初始化全局的指向这个程序在envs[]数组中的Env结构的env指针 */
void
libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = &envs[ENVX(sys_getenvid())];

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

    // call user main routine
    umain(argc, argv);

    // exit gracefully
    exit();
}
```

执行 make qemu 编译文件，运行结果如下。

```
hello, world
Incoming TRAP frame at 0xeffffbfc
i am environment 00001000
Incoming TRAP frame at 0xeffffbfc
```

九、作业八

作业 8

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：

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
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。

首先，当内核中出现缺页错误时，系统调用的缺页处理函数为 kern/trap.c 中的 page_fault_handler()，该函数代码如下。

```
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.

    // 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);
}
```

分析这段代码，代码中 `uint32_t` 定义了一个由 `typedef unsigned int uint32_t` 声明类型的变量 `fault_va`，用于保存报错的地址。下一句代码 `fault_va = rcr2()` 意思是读取处理器的 CR2 寄存器，找到出现错误的地址。下面就需要对内核缺页错误进行处理。也是第一个子问题需要修改的地方。

修改后的代码为：

```
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.
    if(tf->tf_cs & 3 == 0) { //tf->tf_cs && 0x01 == 0
        panic("page_fault in kernel mode, fault address %d\n", fault_va);
    }
    // LAB 3: Your code here.

    // 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);
}
```

由于 `cs` 寄存器的低 2 位的值与 `CPL` 相等，所以可以根据 `cs` 寄存器判断是否在内核态。通过检查 `tf_cs` 最低几位检查缺页中断是发生在用户模式还是内核模式。在该函数中抛出页面错误。

阅读 `kern/pmap.c` 中的 `user_mem_assert()` 函数，函数代码如下。

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

该函数的作用是检查环境“`env`”是否被允许以“`perm | PTE_U | PTE_P`”的权限访问内存范围`[va, va+len)`。如果可以，那么函数直接返回。如果不能，则销毁“`env`”，如果 `env` 是当前环境，此函数将不会返回。

之后，对 `kern/pmap.c` 中的 `user_mem_check()` 函数进行修改，以检测线性地址的页面是否有效。

```

int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    char * end = NULL;
    char * start = NULL;
    start = ROUNDDOWN((char *)va, PGSIZE);
    end = ROUNDUP((char *)va + len, PGSIZE);
    pte_t *cur = NULL;

    for(; start < end; start += PGSIZE) {
        cur = pgdir_walk(env->env_pgdir, (void *)start, 0);
        if((int)start > ULIM || cur == NULL || ((uint32_t)(*cur) & perm) != perm) {
            if(start == ROUNDDOWN((char *)va, PGSIZE)) {
                user_mem_check_addr = (uintptr_t)va;
            }
            else {
                user_mem_check_addr = (uintptr_t)start;
            }
            return -E_FAULT;
        }
    }

    return 0;
}

```

检查环境是否被允许访问具有权限“perm | PTE_P”的内存范围[va, va+len)。通常“perm”至少包含 PTE_U，但这不是必需的。“va”和“len”不需要页面对齐；但是必须测试包含任何该范围的每一页。比如测试“len/PGSIZE”、“len/PGSIZE + 1”或“len/PGSIZE + 2”页面。如果(1)地址低于 ULIM，并且(2)页表授予它权限，用户程序可以访问虚拟地址。如果有错误，需要将“user_mem_check_addr”变量设置为第一个错误的虚拟地址。如果用户程序可以访问此范围的地址，则返回 0；否则返回-E_FAULT。

通过页表找到相应的 cur，然后判断是否具有权限，这里需要记录第一个出错的虚拟地址，所以一开始不能将 va 对齐，还有这种写法的 end 不能向下对齐，因为结果可能会比 start 还小。

下一步，修改 kern/syscall.c 文件来检查传递给内核的参数。在 sys_cputs()函数中添加一行代码 user_mem_assert(curenv, s, len, 0)。以加入相应对用户空间地址的检查。

```
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);
}
```

最后，修改 kern/kdebug.c 中的 debuginfo_eip，在 usd、stabs、stabstr 调用 user_mem_check。代码如下。加入内存检查。

```
// Make sure this memory is valid.
// Return -1 if it is not. Hint: Call user_mem_check.
// LAB 3: Your code here.

if(user_mem_check (curenv, usd, sizeof (struct UserStabData), PTE_U) < 0)
    return -1;

stabs = usd->stabs;
stab_end = usd->stab_end;
stabstr = usd->stabstr;
stabstr_end = usd->stabstr_end;

// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.

if (user_mem_check (curenv, stabs, stab_end - stabs, PTE_U) < 0
|| user_mem_check (curenv, stabstr, stabstr_end - stabstr, PTE_U) < 0)
    return -1;
```

接下来，运行 make run-breakpoint 并在内核中输入 backtrace 得到输出如下。

```
Stack backtrace:
  ebp ffffff20  eip f0100a75  args 00000001 effffff38 f01b4000 00000000 f0172840
    kern/monitor.c:187: monitor+276
  ebp effffff90  eip f0103833  args f01b4000 effffffbc f0105c64 00000082 00000000
    kern/trap.c:196: trap+169
  ebp effffffb0  eip f010393b  args effffffbc 00000000 00000000 eebfdfd0 effffffdc
    kern/syscall.c:68: syscall+0
  ebp eebfdfd0  eip 800073  args 00000000 00000000 eebfdff0 00800049 00000000
    lib/libmain.c:27: libmain+58
Incoming TRAP frame at 0xeffffeac
kernel panic at kern/trap.c:268: Page fault in kernel-mode
```

从输出的 ebp 寄存器内容可以看出，ffffff20, effffff90, effffffb0 都位于内核栈上，仅有 eebfdfd0 位于用户栈上。至此，问题 8 结束。

十、作业九

作业 9

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，你将看见一下信息：

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

最后，打开 `lab1_3` 目录，输入命令 `make run-evilhello`，环境崩溃，出现以下报错信息，作业完成。庆祝一下。

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

十一、问题二**问题 2**

- 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?

1. 断点那个测试样例可能会生成一个断点异常，或者生成一个一般保护错，这取决于你是怎样在 IDT 中初始化它的入口的（换句话说，你是怎样在 `trap_init` 中调用 `SETGATE` 方法的）。为什么？你应该做什么才能让断点异常像上面所说的那样工作？怎样的错误配置会导致一般保护错？

我们先去查看一下宏定义 `SETGATE` (`mmu.h`)。

```

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   see section 9.6.1.3 of the i386 reference: "The difference between
//   an interrupt gate and a trap gate is in the effect on IF (the
//   interrupt-enable flag). An interrupt that vectors through an
//   interrupt gate resets IF, thereby preventing other interrupts from
//   interfering with the current interrupt handler. A subsequent IRET
//   instruction restores IF to the value in the EFLAGS image on the
//   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//       the privilege level required for software to invoke
//       this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl) \
{ \
    (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff; \
    (gate).gd_sel = (sel); \
    (gate).gd_args = 0; \
    (gate).gd_rsv1 = 0; \
    (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).gd_s = 0; \
    (gate).gd_dpl = (dpl); \
    (gate).gd_p = 1; \
    (gate).gd_off_31_16 = (uint32_t) (off) >> 16; \
}

```

注释说明：/设置正常的中断/陷阱门描述符。

- istrap: 1 表示陷阱 (=异常) 门, 0 表示中断门。

参见 i386 参考的第 9.6.1.3 节：“中断门和陷阱门之间的区别在于 IF（中断使能标志）的影响。通过中断门引入的中断复位 IF，从而防止 其他中断干扰当前的中断处理程序。随后的 IRET 指令将 IF 恢复为堆栈上 EFLAGS 映像中的值。通过陷阱门的中断不会改变 IF。”

- sel: 中断/陷阱处理程序的代码段选择器

- off: 中断/陷阱处理程序的代码段偏移量

- dpl: Descriptor Privilege Level-软件使用 int 指令显式调用此中断/陷阱门所需的权限级别。

IDT 的数据结构定义如下：

```

/* Interrupt descriptor table. (Must be built at run time because
 * shifted function addresses can't be represented in relocation records.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
    sizeof(idt) - 1, (uint32_t) idt
};

```

所以如果我们调用该宏定义，需要知道其中的参数意义。也如上面注释中说的，gate 是一个 Gatedesc 结构，使中断描述符表。istrap 表示是陷阱还是中断

门（如果是 `trap` 那么就是 1，如果是 `interrupt` 那么就是 0），按照其宏定义看大多数应该是 `trap`。`sel` 是代码段选择器，按照 `_alltraps` 中实现的，这些都是需要进入内核的，也就是说参数应该是 `GD_KD`。`off` 是代码段偏移量，也就是函数地址，此处填入的就是相应的 `handler` 函数。最后的是 `dpl`，代表的是权限级别，其中要注意的是 `T_BRKPT`，必须置为 3，这是可以使用户自发调用的，否则会出现错误。

`istrap` 表示是陷阱还是中断门（如果是 `trap` 那么就是 1，如果是 `interrupt` 那么就是 0），按照其宏定义看大多数应该是 `trap`。所以此处对于断点异常需要设置为 1。而权限位 `dpl` 的设置在此处也很重要，如果权限位设置的是 0，那么就会出现权限保护错误。因为断点是用户可以设置的，所以应该置为 3。

通过实验发现出现这个现象的问题就是在设置 IDT 表中的 `breakpoint exception` 的表项时，如果我们把表项中的 `DPL` 字段设置为 3，则会触发 `break point exception`，如果设置为 0，则会触发 `general protection exception`。`DPL` 字段代表的含义是段描述符优先级（Descriptor Privileged Level），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`。那么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`，但是如果把 IDT 这个表项的 `DPL` 设置为 3 时，就不会出现这样的现象了，这时如果再出现异常，肯定是因为我们还没有编写处理 `break point exception` 的程序所引起的，所以是 `break point exception`。

2. 你认为这样的机制意义是什么？尤其要想想测试程序 `user/softint` 的所作所为 / 尤其要考虑一下 `user/softint` 测试程序的行为。

这样的机制可以实现操作系统内部的软中断保护机制。一个中断产生之后，内核在中断处理函数中可能需要完成很多工作。但是中断处理函数的处理是关闭了中断的。也就是说在响应中断时，系统不能再次响应外部的其它中断。这样的后果会造成有可能丢失外部中断。于是，`linux` 内核设计出了一种架构，中断函

数需要处理的任务分为两部分，一部分在中断处理函数中执行，这时系统关闭中断。另外一部分在软件中断中执行，这个时候开启中断，系统可以响应外部中断。所以此处中断源发出软中断的信号，cpu 接收到了之后可以进行对应的中断处理功能。