

操作系统第二次上机作业实验报告

成员1:

专业：计算机科学与技术

学号：1310617

姓名：刘丹

成员2:

专业：计算机科学与技术

学号：1310636

姓名：赵婧涵

一、用户环境及异常处理

1. 用户环境(User Environments)

1.1 理论准备

(1) Env 数据结构

在 inc/env.h 中可以看到:

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

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

env_tf

其定义在 inc/trap.h, 当内核或者其他的用户环境在运行时, 该结构保存着当这个环境不运行时寄存器的值, 内核当由用户态转向内核态时保存这些值, 以便之后还可以 resume 该环境。

env_link

指向 env_free_list 的 next Env 的指针, env_free_list 指向空闲列表的 the first free environment。

env_id

该变量保存一个环境的 id, 这个 id 唯一地标识了当前使用这个 Env 结构的用户环境。当一个用户环境终止之后, 内核可能会将这个 Env 结构分配给新的用户环境, 但是这时这个用户环境就会有一个不同的 env_id, 即使新的用户环境和旧的使用的是 envs 数组中的同一个位置。

env_parent_id

该变量保存着创建这个环境的环境的 env_id(parent), 通过这种方式, 内核可以生产一个家谱(family tree), 这在决定哪些环境可以对别的环境做出何种操作时十分有用。

env_type

该变量用以区分环境的类型, 对大部分环境来说, 它是 ENV_TYPE_USER, 空闲的环境是 ENV_TYPE_IDLE。我们会在后续试验介绍更多的类型。

env_status

这个变量有如下值:

ENV_FREE: 表明当前的 Env 是没有激活的(inactive), 也就是在 env_free_list 上的。

ENV_RUNNABLE: 表明当前 Env 代表的环境正在等待处理器。

ENV_RUNNING: 表示该环境正在运行。

ENV_NOT_RUNNABLE: 表明这个 Env 是活动的(active)并且当前还没有准备好在处理器上运行, 比如它正在其他环境的 IPC(进程通信)。

env_pgdir

该变量保存这个环境页目录(page directory)的虚拟地址。

在文件 kern/env.c 中, 内核维护着如下三个重要的全局变量:

```
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)
```

一旦 JOS 启动并开始运行, envs 就会指向一个保存系统中所有用户环境的 Env 数组。在当前的设计中, JOS 内核将支持最多 NENV 个同时活动的(active)环境, 当然, 实际运行时的个数将比 NENV 少很多的(NENV 是在文件 inc/env.h 中使用#define 定义的常量)。一旦 envs 数组被分配之后, 它就会为每一个存在的环境保存一份 Env 数据。

JOS 内核将所有未激活的(inactive) Env 结构保存在 env_free_list 中。这个设计使得系统可以非常快速的分配和释放用户环境, 因为这个链表很少进行添加和删除操作。

内核使用变量 curenv 来保存当前运行的环境(currently executing environment)。在启动的过程中, 在第一个环境运行之前, curenv 将被初始化为 NULL。

(2) 总体过程

在建立进程之前首先要建立进程的管理机制, 包括分配进程描述符的空间、建立空闲进程列表、把进程描述符数组相应内存进行映射等。这是进程描述符管理的一些工作。

其次初始化进程的虚拟地址空间, 也就是给其页目录赋值过程, 主要是将内核空间映射

到虚拟地址的高位上。

然后加载进程的代码，代码以二进制形式和内核编译到一起了，所谓“加载”就是将这段代码从内存（内核区往上一一点的位置）复制到某个物理位置，接着将这个位置和相应虚拟地址进行映射。

最后将进程的页目录加载进 cr3，然后将各寄存器压栈，通过 iret 跳到用户态执行。

(3)注：当前 JOS 没有文件系统，所以用户态的程序是编译在 kernel 里面的，通过编译器的导出符号来进行访问。

1.2 问题与作业

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 `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.

设计思路:给数组 `envs` 分配空间，然后将分配的页面从 `page_free_list` 里剔除掉，最后将 `UENVS` 映射到 `envs`，整个过程跟 `Page` 数组的处理完全一样

代码实现:(见 `kern/pmap.c`)

在 `mem_init()` 中添加以下代码:

```
////////////////////////////////////
// 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)); //给envs分配空间

////////////////////////////////////
// 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.
//UENVS的映射
boot_map_region(kern_pgdir,UENVS,ROUNDUP(NENV* sizeof(struct Env),PGSIZE),PADDR
(envs),PTE_U | PTE_P);
```

在 `page_init()` 中修改原来代码, 将 `envs` 所占用的页面也从 `age_free_list` 里剔除掉:

```
//剔除两块地址

//第一块是0地址开始的第一个页面 (包括IDT等)
extern char end[];
pages[1].pp_link=0; //只需让第二个页面 (下标为1) 的pp_link域指向空, 其原本其指向的是第一个页面

//第二块是io hole开始的向上的一组连续的页面: IO hole和kernel之上部分(kernel本身+kern_pgdir+pages+envs)

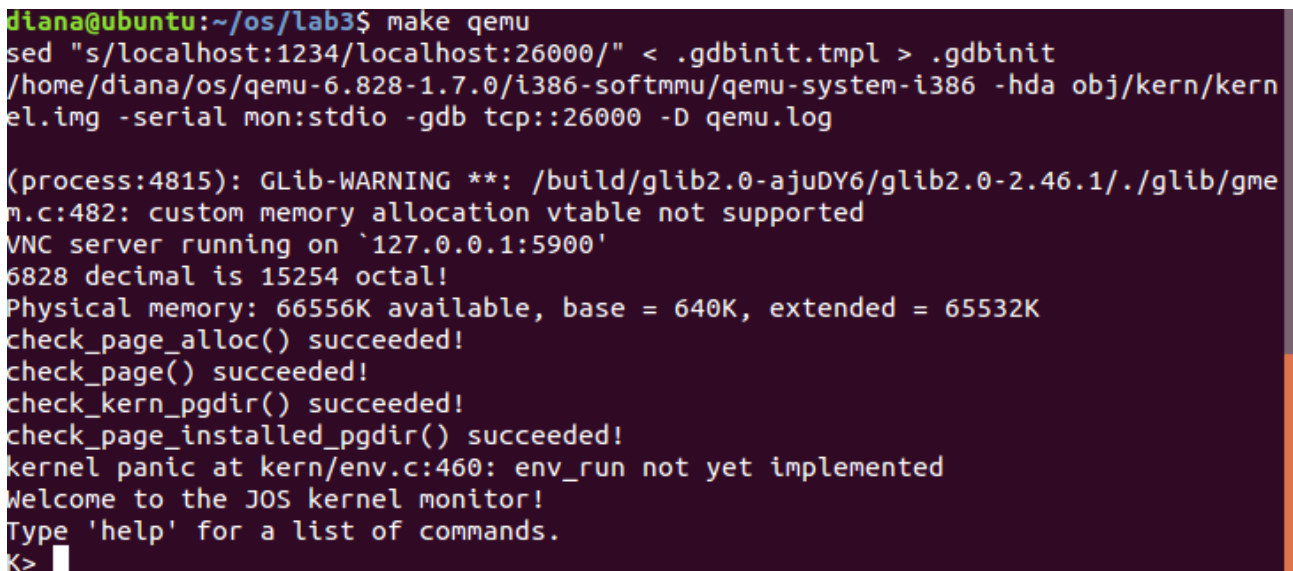
//先计算出要剔除的Page的地址

//首地址, 低地址, IOPHYMEM已经是物理地址了, 所以只需要使用pa2page得到Page结构即可。
struct Page* pgstart=pa2page((physaddr_t)IOPHYMEM);
/*尾, 高地址, 首先将end符号地址转化成物理地址(-KERNBASE), 然后再加上刚才分配的kern_pgdir (一个PGSIZE)
和pages数组以及envs所占用的空间即可。*/
struct Page* pgend=pa2page((physaddr_t)(end-KERNBASE+PGSIZE+npages*sizeof(struct Page)+NENV* sizeof(struct Env)));

//pgstart和pgend这两个Page也要剔除
pgend++;
pgstart--;

pgend->pp_link=pgstart; //改变pp_link域, 跳过中间的区域 (要剔除的部分)
```

检测结果:



```
diana@ubuntu:~/os/lab3$ make qemu
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
/home/diana/os/qemu-6.828-1.7.0/i386-softmmu/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log

(process:4815): GLib-WARNING **: /build/glib2.0-ajuDY6/glib2.0-2.46.1/./glib/gmain.c:482: custom memory allocation vtable not supported
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
kernel panic at kern/env.c:460: env_run not yet implemented
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
k>
```

可以看到 `check_kern_pgdir()` succeeded!

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,

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

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

(1)env_init()

函数作用: 进程描述符数组 `envs` 的初始化。

设计思路: 将数组中所有的进程 `id` 置0, 状态设置为未激活, 同时将各元素串起来, 并把数组地址赋给 `env_free_list`。

代码实现:

```

void
env_init(void)//进程描述符数组envs的初始化
{
    // Set up envs array
    // LAB 3: Your code here.
    int i=0;
    for(i=0;i<NENV;i++)//将数组中所有的进程id置0，状态设置为未激活,同时将各元素串起来
    {
        envs[i].env_id=0;
        envs[i].env_status = ENV_FREE;
        if(i!=NENV-1)
            envs[i].env_link=&envs[i+1];
    }
    env_free_list=envs; //把数组地址赋给env_free_list
    // Per-CPU part of the initialization
    env_init_percpu();
}

```

(2)env_setup_vm()

函数作用:初始化进程虚拟地址空间,不同的进程有不同的虚拟地址空间,进而就必须有自己的页目录和页表,利用该函数初始化页目录。

设计思路:首先分配一个空闲页当做页目录,并增加分配的物理页的引用,将此页的虚拟地址赋值给进程的 pgdir,使进程有权限操作自己的 pgdir,然后将这个页目录映射内核地址空间(UTOP 之上的部分),不需要映射页表,因为可以和内核共用页表。

代码实现:

```

// LAB 3: Your code here.

e->env_pgdir=page2kva(p); //将此页的虚拟地址赋值给进程的pgdir,使进程有权限操作自己的pgdir。
memset(e->env_pgdir,0x0,PGSIZE);

p->pp_ref++; //增加分配的物理页的引用
for(i=PDX(UTOP);i< NPENTRIES;i++) //将这个页目录映射内核地址空间(UTOP之上的部分) NPENTRIES=1024
    e->env_pgdir[i]=kern_pgdir[i];

```

(3)region_alloc()

函数作用:辅助映射,映射虚拟地址 va 及之后的 len 字节到进程 e 的虚拟地址空间里。

设计思路:首先将 va 向下4K 对齐, va+len 向上4k 对齐,以保证分配的是整数页面。然后利用 page_insert() 进行页面分配,为了使页面用户态可写,设置权限 PTE_W|PTE_U。

代码实现:

```

static void
region_alloc(struct Env *e, void *va, size_t len) //映射虚拟地址va及之后的len字节到进程e的虚拟地址空间里
{
    // LAB 3: Your code here.
    // (But only if you need it for load_icode.)
    //
    // Hint: It is easier to use region_alloc if the caller can pass
    // 'va' and 'len' values that are not page-aligned.
    // You should round va down, and round (va + len) up.
    // (Watch out for corner-cases!)

    void* i;
    //将va向下4k对齐, va+len向上4k对齐, 以保证分配的是整数页面 |
    for(i=ROUNDDOWN(va,PGSIZE);i<ROUNDUP(va+len,PGSIZE);i+=PGSIZE)
    {
        struct Page* p=(struct Page*)page_alloc(1);
        if(p==NULL)
            panic("region_alloc:%e",-E_NO_MEM); //out of memory
        page_insert(e->env_pgdir,p,i,PTE_W|PTE_U); //页面分配,使页面用户态可写,设置权限PTE_W|PTE_U
    }
}

```

(4)load_icode()

函数作用:给相应的进程加载可执行代码。

设计思路:使用类似加载 kernel 的方式将代码加载到相应内存中。为了往进程对应的虚拟空间映射到的物理内存中写数据,首先必须要加载进程相应的页目录(在调用此函数之前页目录是已经初始 化过的)。接着仿照 kernel 的方式去分析 elf 文件,加载类型为 ELF_PROG_LOAD 的段到其要求的虚拟地址中, 使用 region_alloc()进行地址的映射。最后将程序入口放入进程的 eip 中,并映射进程堆栈,然后重新加载 kern_pgdir。

代码实现:

```
// LAB 3: Your code here.

lcr3(PADDR(e->env_pgdir)); //加载进程相应的页目录

struct Elf * ELFHDR=(struct Elf *)binary;
struct Proghdr *ph, *eph;
int i;
if (ELFHDR->e_magic != ELF_MAGIC)
    panic("Not a elf binary");

ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;
for (; ph < eph; ph++)
{
    // p_pa是该段的加载地址 (也是物理地址)
    if(ph->p_type==ELF_PROG_LOAD) //加载类型为ELF_PROG_LOAD的段到其要求的虚拟地址中
    {
        if(ph->p_filesz>ph->p_memsz)
            panic("load_icode:file size larger than memory size");
        region_alloc(e,(void*)ph->p_va,ph->p_memsz); //使用 region_alloc()进行地址的映射
        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));
    }
}

e->env_tf.tf_eip=ELFHDR->e_entry; //将程序入口放入进程的eip中

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

// LAB 3: Your code here.

struct Page* p=(struct Page*)page_alloc(1);
if(p==NULL)
    //panic("Not enough mem for user stack!");
    panic("page_alloc:e%\n",p);
page_insert(e->env_pgdir,p,(void*)(USTACKTOP-PGSIZE),PTE_W|PTE_U); //映射进程堆栈

lcr3(PADDR(kern_pgdir)); //重新加载kern_pgdir
```

(5)env_create()

函数作用:建立进程。

设计思路:首先分配一个进程描述符,然后加载可执行代码。

代码实现:

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env* env;
    int err;
    err = env_alloc(&env, 0);
    if(err== -E_NO_FREE_ENV || err== -E_NO_MEM)
        panic("env_alloc() failed: %e\n", err);
    if(err==0) //分配一个进程描述符
    {
        env->env_type=type;
        load_icode(env, binary,size); //加载可执行代码
    }
}
```

(6)env_run()

函数作用:运行进程。

设计思路:首先设置当前进程的一些信息,然后更改当前进程指针指向要运行的进程,之后加载进程页目录,跳转到 env_pop_tf() 真正使进程执行。

代码实现:

```
// LAB 3: Your code here.

if(curenv!=NULL)
    if(curenv->env_status==ENV_RUNNING)
        |    curenv->env_status=ENV_RUNNABLE;

e->env_status=ENV_RUNNING; //设置当前进程的一些信息
e->env_runs++;
curenv=e; //更改当前进程指针指向要运行的进程
lcr3(PADDR(e->env_pgdir));
env_pop_tf(&e->env_tf); //加载进程页目录跳转到使用env_pop_tf真正使进程执行
```

下面使用 debugger 检查是否进入了用户态(user mode):

(1)使用“make qemu-gdb”并在 env_pop_tf() 处设置 GDB 断点,它是进入用户态之前的最后一个函数:

在 obj/kern/kernel.asm 中可以看到:

```
env_pop_tf(&e->env_tf); //加载进程页目录跳转到使用env_pop_tf真正使进程执行
f0102e2f:      83 ec 0c          sub    $0xc,%esp
f0102e32:      50              push   %eax
f0102e33:      e8 7a ff ff ff  call   f0102db2 <env_pop_tf>
```

```
f0102db2 <env_pop_tf>:
//
// This function does not return.
//
void
env_pop_tf(struct Trapframe *tf)
{
f0102db2:      55              push   %ebp
f0102db3:      89 e5          mov    %esp,%ebp
f0102db5:      83 ec 0c          sub    $0xc,%esp
```

可以看到 env_pop_tf() 的地址为0xf0102db2, 在此处设置断点:

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000ffff in ?? ()
(gdb) b *0xf0102db2
Breakpoint 1 at 0xf0102db2
(gdb) c
Continuing.

Breakpoint 1, 0xf0102db2 in ?? ()
(gdb) █
```

(2) 使用 si 命令做单步调试, 在 iret 指令后进入用户态:

```
(gdb) x/10i 0xf0102db2
=> 0xf0102db2: push    %ebp
0xf0102db3: mov    %esp,%ebp
0xf0102db5: sub    $0xc,%esp
0xf0102db8: mov    0x8(%ebp),%esp
0xf0102dbb: popa
0xf0102dbc: pop    %es
0xf0102dbd: pop    %ds
0xf0102dbe: add    $0x8,%esp
0xf0102dc1: iret
0xf0102dc2: push   $0xf01052ab
(gdb) █
```

执行到 `iret` 后，单步执行：

```
(gdb) si
0xf0102dc1 in ?? ()
(gdb) x/10i 0xf0102dc1
=> 0xf0102dc1:  iret
    0xf0102dc2:  push    $0xf01052ab
    0xf0102dc7:  push    $0x1f6
    0xf0102dcc:  push    $0xf010521e
    0xf0102dd1:  call    0xf01000a5
    0xf0102dd6:  push    %ebp
    0xf0102dd7:  mov     %esp,%ebp
    0xf0102dd9:  sub     $0x8,%esp
    0xf0102ddc:  mov     0x8(%ebp),%eax
    0xf0102ddf:  mov     0xf017c244,%edx
(gdb) si
0x00800020 in ?? ()
(gdb) x/10i 0x00800020
=> 0x800020:    cmp     $0xeebfe000,%esp
    0x800026:    jne     0x80002c
    0x800028:    push    $0x0
    0x80002a:    push    $0x0
    0x80002c:    call    0x80005e
    0x800031:    jmp     0x800031
    0x800033:    push    %ebp
    0x800034:    mov     %esp,%ebp
    0x800036:    sub     $0x14,%esp
    0x800039:    push    $0x800e94
(gdb) █
```

看到进入用户环境的第一条指令是在 `0x800020`,

在 `obj/user/hello.asm` 中可以看到如下内容：

```
00800020 <_start>:
// 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
800020:    81 fc 00 e0 bf ee    cmp     $0xeebfe000,%esp
    jne args_exist
800026:    75 04                jne     80002c <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
800028:    6a 00                push    $0x0
    pushl $0
80002a:    6a 00                push    $0x0
```

即进入用户环境的第一条指令是 `hello.asm` 中 `start` 标签的 `cmpl` 指令。

(3)在 hello.asm 的 sys_cputs() 的 int \$0x30 处设置断点, int 这条指令是打印一个字符到控制台的系统调用。

在 obj/user/hello.asm 中可以看到:

```
void
sys_cputs(const char *s, size_t len)
{
    800b13:      55                push    %ebp
    800b14:      89 e5            mov     %esp,%ebp
    800b16:      57                push    %edi
    800b17:      56                push    %esi
    800b18:      53                push    %ebx
    //
    // The last clause tells the assembler that this can
    // potentially change the condition codes and arbitrary
    // memory locations.

    asm volatile("int %1\n"
    800b19:      b8 00 00 00 00    mov     $0x0,%eax
    800b1e:      8b 4d 0c            mov     0xc(%ebp),%ecx
    800b21:      8b 55 08            mov     0x8(%ebp),%edx
    800b24:      89 c3                mov     %eax,%ebx
    800b26:      89 c7                mov     %eax,%edi
    800b28:      89 c6                mov     %eax,%esi
    800b2a:      cd 30                int     $0x30
    |
```

看到 int \$0x30的用户态地址是0x800b2a。

在 0x800b2a 处设置断点, 并执行到这一步:

```
(gdb) b *0x800b2a
Breakpoint 2 at 0x800b2a
(gdb) c
Continuing.

Breakpoint 2, 0x00800b2a in ?? ()
(gdb) x/i 0x800b2a
=> 0x800b2a:    int    $0x30
(gdb)
```

可以看到可以执行到 int。

综合以上, 作业2成功完成。

2. 异常处理(Exception Handling)

2.1 理论准备

2.1.1 控制权转移(Basics of Protected Control Transfer)

异常和中断都是控制权转移(Protected Control Transfer),它将使得处理器从用户模式转移到内核模式而不给用户代码任何的机会去干预内核或者其他的进程。在 Intel 的术语中,中断(Interrupt)是一种保护下的控制权转移,通常是由外部的异步事件例如外部设备的 I/O 行为通知处理器所造成的;异常(Exception)则是由当前正在运行的代码的同步事件所造成的控制权转移,比如除零错误或者非法的内存访问。

为了保证这些控制权转移是受保护的(protected),处理器的中断/异常机制被设计为在中断或者异常出现时,当前正在运行的代码不需要知道内核以何种方式在何处被载入。相反,处理器保证一定会在严格控制的条件下进入内核。在 x86 体系中,这种保护以如下两种方式体现:

(1) 中断描述符表(The Interrupt Descriptor Table, IDT)

处理器保证中断或者异常只会使内核在少数特殊的、完善设计的、被内核自己所预先指定的地方载入,而不是中断或者异常出现时所正在运行的代码决定。

特别的说,在 x86 中,中断和异常最多被区分为 256 种类型,每一种都被一个特殊的中断号(interrupt number,有时候也称为异常号(exception number)或者陷阱号(trap number))所关联。一旦处理器确认一个特定的中断或者异常发生了,它就将中断号作为在 IDT 中的索引。IDT 是内核设立在内核的私有内存中的,类似于 GDT。在这个表的某项中,处理器可以获得:

要载入指令指针寄存器(EIP)的值,也就是用来处理这种类型异常的内核代码;

要载入代码段寄存器(CS)的值,包括了将要执行的 exception handler 的权限等级(privilege),不过在 JOS 中,所有的中断都在内核中进行处理,所以它的权限是 0。

(2) 任务状态段(The Task State Segment, TSS)

为了保证在内核中保证中断处理程序能够拥有一个完善的载入点(entry-point),在处理器执行中断处理程序之前,也需要一个地方来存储旧的处理器状态,比如 EIP 和 CS 的值,这样就可以在执行完中断处理程序之后,处理器还可以继续执行被中断的程序。当然,这部分区域必须保证不能被未授权的代码访问,否则,有错误的程序或者恶意代码就可以简单的危害到内核(现在使用这种方式进行攻击的依然不在少数)。

因此, 当 x86 处理器响应一个中断或陷阱时, 会引起从用户态到内核态权限的改变, 也会切换到内核的栈。TTS 指定了新栈的 the segment selector and address。处理器将 SS, ESP, EFLAGS, CS, EIP and an optional error code 压入新栈, 然后从 IDT 加载 CS 和 EIP, 设置 SS 与 ESP 为新栈。

尽管 TTS 是一个非常庞大且支持多种用途的结构, 在 JOS 中, 它将只被用于定义当处理器从用户态转移到核心态时的内核栈。由于 JOS 中的内核模式在 x86 的特权等级体系中是 0, 当进入内核模式时处理器使用 TTS 中的 ESP0 和 SS0 两部分来定义内核栈, TS 的其他部分在 JOS 中并不被使用。

2.1.2 中断和异常的类型 (Types of Exceptions and interrupts)

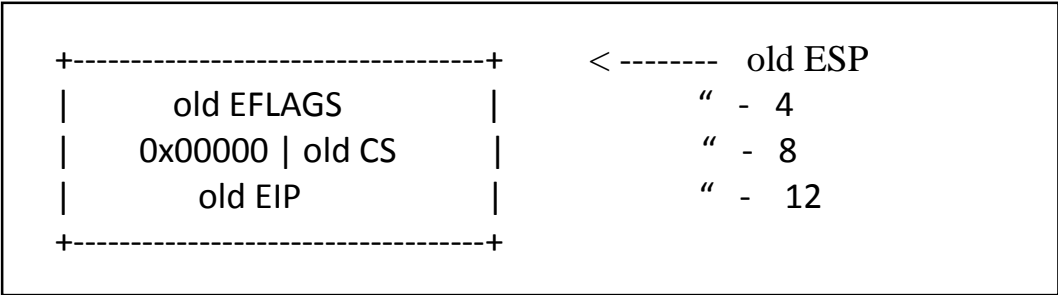
X86 处理器所能生成的所有同步异常在内部都是使用中断号 0 到 31 的, 并且映射到 IDT 中的 0 到 31 号表项。例如, Intel 从硬件上将缺页中断处理 (page fault handler) 设置为 14。高于 31 的中断号只被用于处理被 INT 指令引起的软中断 (software interrupt), 或者是由外部设置在需要时所产生的异步硬中断 (hardware interrupt)

在本节中, 将扩展 JOS 的中断处理功能, 使其能够处理 0 到 31 号由 Intel 定义的中断。在下一节中, 将扩展 JOS 能够处理软中断 0x30, JOS 使用该软中断进行系统调用的处理。在后续的实验, 将继续扩展 JOS, 最终使其可以支持诸如时钟中断 (clock interrupt) 之类的硬中断。

2.1.3 中断和异常的嵌套 (Nested Exceptions and Interrupts)

处理器在用户态和内核态中都可以响应异常和中断, 但是, 只有从用户态转入内核态时, x86 处理器才会在将原有的寄存器值入栈前转换栈, 并且从 IDT 中读取相应的异常处理函数入口。当中断或异常发生时, 处理器已经处于内核 (CS 的最低两个 bit 已经是 0 时), 内核只需要将更多的值压入当前的内核栈。这样, 内核就可以低代价的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模式并产生一个嵌套的异常, 由于它不需要转换栈, 因此就不再保存旧的 SS 或 ESP 寄存器的值。对于不需要压入错误码的异常, 内核堆栈就是如下:



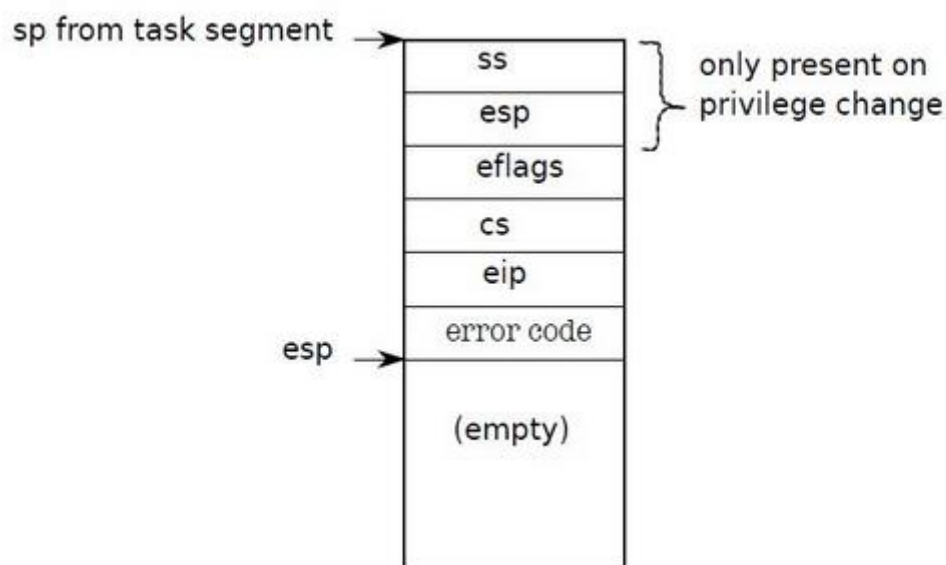
对于需要压入错误码的, 处理器在旧的 EIP 之后压入一个 error code。

关于嵌套异常还有一点重要的说明, 如果处理器在内核模式下又处理了一个异常, 并且由于诸如栈空间不足等原因无法将原有的处理器状态压入栈, 那么处理器除了进行重置(reset)之外别无选择。也就是说, 任何正常的内核都应该在设计时就保证这种情况永远都不会发生。

2.1.4 int 指令

int 指令是一个较为复杂的指令, 其做了很多事情, 按顺序包括以下几步:

- 1、查找 idtr 里面的 idt 地址, 根据这个地址找到 idt, 然后根据 idt 找到中断向量的表项;
- 2、检查 cpl 和表项的 dpl, 如果 $cpl > dpl$ 产生保护异常, 否则继续;
- 3、根据 tssr 寄存器里的 tss 地址找到 tss 在内存中的位置, 读取其中的 ss 和 esp 并装载 (tss 结构是 x86 定义好的, 其内存中存放的位置需要 os 去决定, 并对其中内容的赋值也要 os 实现, 这部分内容在 trap.c 的 trap_init_percpu 函数中, 同时还包括着加载 idt 的逻辑);
- 4、如果是一个用户态到内核态的陷入操作, 则像堆栈中压入 ss 和 esp, 注意这个 ss 和 esp 是之前用户态的数据, 而不是新装载的数据;
- 5、压入 esp, eflags, eip;
- 6、修改 eflags 中的某些位 (比如关中断);
- 7、如果有必要, 压入 errorcode, 某些中断需要 errorcode 以及不同中断的 errorcode 含义可查看 handout;
- 8、根据 idt 表项设置 cs 和 eip, 也就是跳转到处理函数执行。idt 内容相关可查看 handout。



压入后的堆栈就应该是这个样子的，跳转到相应中断处理函数的时候就是这样一个堆栈。接着中断处理函数处理相应操作，然后根据目前堆栈里有的内容和当前寄存器的内容恢复现场，继续程序执行。

2.1.5 CPL、RPL、DPL

CPL 是当前进程的权限级别 (Current Privilege Level)，是当前正在执行的代码所在的段的特权级，存在于 cs 寄存器的低两位。

RPL 说明的是进程对段访问的请求权限 (Request Privilege Level)，是对于段选择子而言的，每个段选择子有自己的 RPL，它说明的是进程对段访问的请求权限，有点像函数参数。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。RPL 可能会削弱 CPL 的作用，例如当前 CPL=0 的进程要访问一个数据段，它把段选择符中的 RPL 设为 3，这样它对该段仍然只有特权为 3 的访问权限。

DPL 存储在段描述符中，规定访问该段的权限级别 (Descriptor Privilege Level)，每个段的 DPL 固定。

当进程访问一个段时，需要进程特权级检查，一般要求 $DPL \geq \max \{CPL, RPL\}$ 。

2.2 问题与作业

作业 3

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.

代码阅读:

(1) IDT 的数据结构, 定义在 `kern/trap.c` 中:

```
/* 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
};
```

`Idt_pd` 是系统寄存器 `IDTR` 的对应结构。

(2) 门描述符数据结构 struct Gatedesc 定义在 inc/mmu.h 中:

```
// 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
};

// 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; \
} \
```

其中提供一个很好用的宏 SETGATE 用来设置一个特定的描述符。

(3) TRAPHANDLER 和 TRAPHANDLER_NOEC

TRAPHANDLER in kern/trapentry.S:

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

TRAPHANDLER 定义一个全局方法来处理陷阱:将陷阱号压栈,然后跳到_alltraps。被用来处理 CPU 自动将错误代码压入的中断。

TRAPHANDLER_NOEC in kern/trapentry.S

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

\\
\\
\\
\\
\\
\\
\\

TRAPHANDLER_NOEC 被用于处理 CPU 不压入错误代码,而是压入0的中断,所以在任何情况下 trap frame 有一样的格式。

这两个宏定义的功能就是接受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的以该函数名命名的中断处理程序。这样的中断向量程序的执行流程就是向栈里压入相关错误码和中断号,然后跳转到_alltraps 来执行共有的部分(把 Trapframe 剩下的那些结构在栈中设置好)。

这里牵涉到一个重要的问题,就是错误代码,如果是系统运行中产生的中断,根据不同的中断类型,在切换完栈以后,处理器会向栈中放入一个错误代码。比如8号中断 Double Fault,但是比如0号 Divide Zero 就不会放。特别注意,当用户使用 int 指令手动调用中断时,处理器是不会放入错误代码的,在系统没有放入错误码时,我们的中断处理程序就要手动补齐这个空间了。TRAPHANDLER_NOEC 宏就是帮我们完成这个事情的。

(4) struct 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));
```

某些 padding 开头的变量是为了让数据补齐4Byte。

Trapframe 保存的都是一些系统关键的寄存器。需要特别关注4个寄存器,涉及到程序执行的控制流问题:

- EFLAGS:状态寄存器;
- EIP:Instruction Pointer, 当前执行的汇编指令的地址;
- ESP:当前的栈顶地址;
- EBP:辅助用, 当前过程的帧在栈中的开始地址(高地址)即 EBP 到 EIP 中就是此过程的帧;

的帧;

其中 EBP 由程序自行操作,而其他三者都会被在执行汇编指令时被改变。push 和 pop 指令都是以 ESP 指针为操作目标的。

(5) the T_*defines in inc/trap.h

```
// 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

#define IRQ_OFFSET   32      // IRQ 0 corresponds to int IRQ_OFFSET
```

设计思路：

题目要求完成 trapentry.S 和 trap.c 的部分内容，使之能正确的调用 trap 函数，并将一个正确的 trapframe 结构指针当做函数的参数。

在这个实验时，还需要完成很多操作才能进行系统调用。因为加载 idt 的工作 JOS 已经帮我们做了，我们需要做的就是初始化 idt，给不同的中断分配不同的处理函数。

简单分析 JOS 的逻辑可知，JOS 是先将所有中断都跳转到 trap 函数，再在 trap 函数里调用 trap_dispatch 来进行分发，再在 trap_dispatch 中调用具体的处理函数，这意味着 idt 里指向的函数只需要调用 trap 函数就可以了。

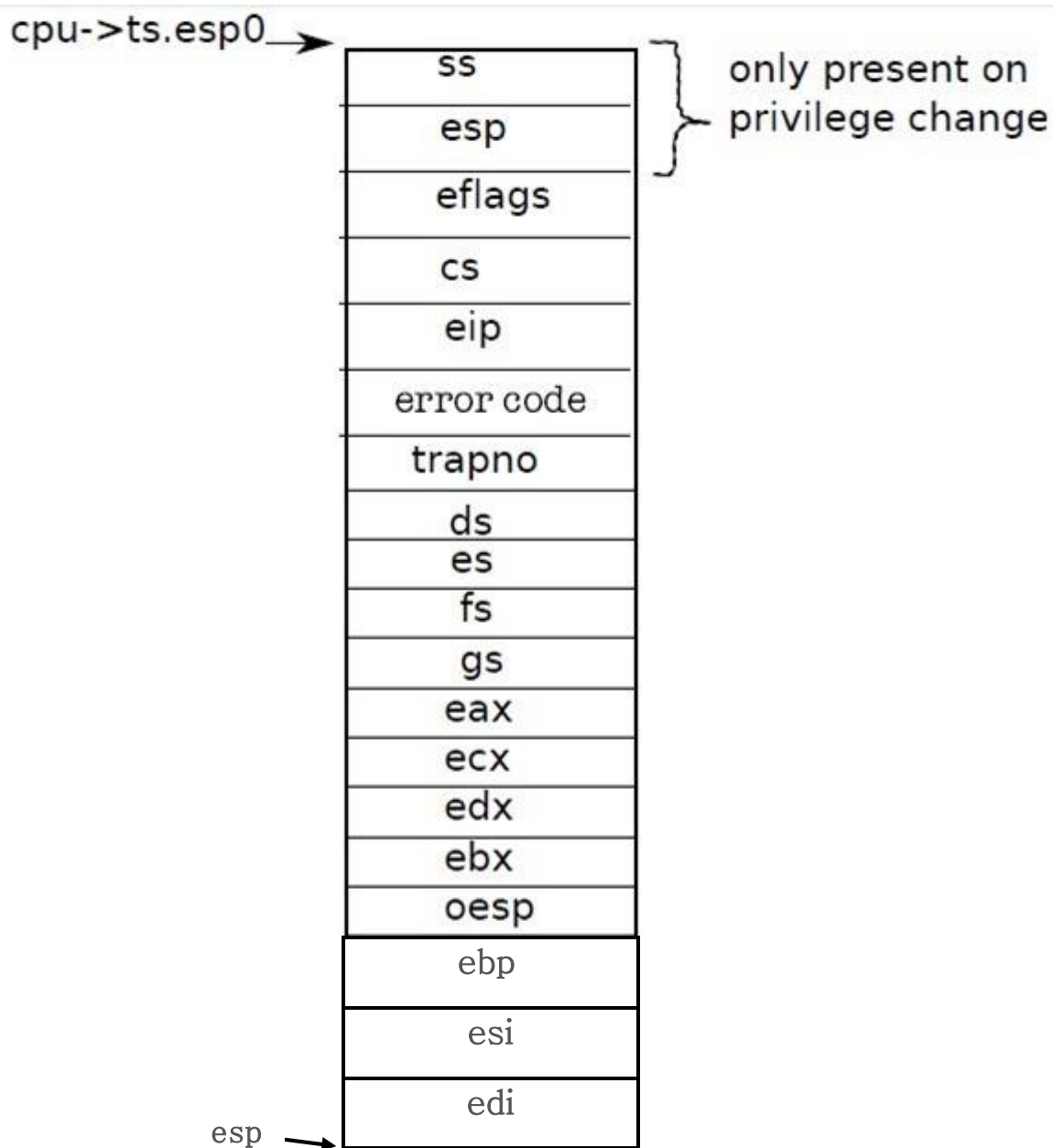
在 trapentry.S 中，根据定义的两个宏，参考 handout 里面 errorcode 所对应的中断号，可以为不同的中断号定义处理函数，名字随便取，然后再在 trap.c 中声明这些函数，并获取函数地址填充进 idt 中（使用 setgate 宏，使用 GD_KT 段，也就是 OS 的代码段）。

此时 idt 初始化代码就完成了，发现所有中断处理函数跳转到 _alltrap 处，在 trapentry.S 中定义 _alltrap 符号，接着往堆栈里压入某些值，使堆栈看起来像是一个 trapframe，此时堆栈栈顶的指针就是指向这个 trapframe 的指针（结构体内存从低向高增

长), 将这个指针也就是 esp 压入堆栈, 调用 trap 函数, 进入函数时, 会取栈顶元素当做参数, 正好就是这个 trapframe, 这样就完成了我们想要的功能。

接下来考虑如何压栈才能让堆栈看起来像 trapframe。

trapframe 看起来应该是这个样子的:



在每一个中断处理函数里, 已经压入了 trapno, 所以在_alltrap 处, 还需要压入 trapno 以下的所有内容。接着根据要求加载内核数据段, 最后再压入 esp, 执行 call trap 即可转入 c 执行。

代码实现：

kern/trapentry.S 中，根据定义的两个宏，参考 handout 里面 errorcode 所对应的中断号，可以为不同的中断号定义处理函数。

```
.text
```

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
TRAPHANDLER_NOEC(divide_error, T_DIVIDE)//divide error
TRAPHANDLER_NOEC(debug, T_DEBUG)//debug exception
TRAPHANDLER_NOEC(nmi, T_NMI)//non-maskable interrupt
TRAPHANDLER_NOEC(break_point, T_BRKPT)//breakpoint
TRAPHANDLER_NOEC(overflow, T_OFLOW)//overflow
TRAPHANDLER_NOEC(bounds, T_BOUND)//bounds check
TRAPHANDLER_NOEC(illegal_op, T_ILLOP)//illegal opcode
TRAPHANDLER_NOEC(device_not_available, T_DEVICE)//device not available

TRAPHANDLER(double_fault, T_DBLFLT)//double fault
TRAPHANDLER_NOEC(coprocessor_segment_overrun, T_COPROC)//reserved (not generated by recent processors)
TRAPHANDLER(invalid_TSS, T_TSS)//invalid task switch segment
TRAPHANDLER(segment_not_present, T_SEGNP)// segment not present
TRAPHANDLER(stack_exception, T_STACK)// stack exception
TRAPHANDLER(general_protection_fault, T_GPFLT)// general protection fault
TRAPHANDLER(page_fault, T_PGFLT)//page fault
TRAPHANDLER_NOEC(reserved, T_RES) //reserved
TRAPHANDLER_NOEC(floating_point_error, T_FPERR)//floating point error
TRAPHANDLER(alignment_check, T_ALIGN)//alignment check
TRAPHANDLER_NOEC(machine_check, T_MCHK)//machine check
TRAPHANDLER_NOEC(SIMD_float_point_error, T_SIMDERR)//SIMD floating point error
TRAPHANDLER_NOEC(system_call, T_SYSCALL)// system call
```

_alltraps:往堆栈里压入某些值，使堆栈看起来像是一个 trapframe，此时堆栈栈顶的指针就是指向这个 trapframe 的指针（结构体内存从低向高增长），将这个指针也就是 esp 压入堆栈，调用 trap 函数，进入函数时，会取栈顶元素当做参数，正好就是这个 trapframe：


```

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

_alltraps:
    pushl %ds
    pushl %es
    pushal
    pushl $GD_KD
    popl %ds
    pushl $GD_KD
    popl %es
    pushl %esp
    call trap

```

kern/trap.c 中:在 trap.c 中声明这些函数, 并获取函数地址填充进 idt 中 (使用 setgate 宏, 使用 GD_KT 段, 也就是 OS 的代码段)

trap_init() 中添加如下代码:

```

extern void divide_error();
extern void debug();
extern void nmi();
extern void break_point();
extern void overflow();
extern void bounds();
extern void illegal_op();
extern void device_not_available();
extern void double_fault();
extern void coprocessor_segment_overrun();
extern void invalid_TSS();
extern void segment_not_present();
extern void stack_exception();
extern void general_protection_fault();
extern void page_fault();
extern void reserved();
extern void floating_point_error();
extern void alignment_check();
extern void machine_check();
extern void SIMD_float_point_error();
extern void system_call();

```

```
SETGATE(idt[T_DIVIDE],0,GD_KT,divide_error,0);
SETGATE(idt[T_DEBUG],0,GD_KT,debug,0);
SETGATE(idt[T_NMI],0,GD_KT,nmi,0);
SETGATE(idt[T_BRKPT],0,GD_KT,break_point,0);
SETGATE(idt[T_OFLOW],0,GD_KT,overflow,0);
SETGATE(idt[T_BOUND],0,GD_KT,bounds,0);
SETGATE(idt[T_ILLOP],0,GD_KT,illegal_op,0);
SETGATE(idt[T_DEVICE],0,GD_KT,device_not_available,0);
SETGATE(idt[T_DBLFLT],0,GD_KT,double_fault,0);
SETGATE(idt[T_COPROC],0,GD_KT,coprocessor_segment_overrun,0);
SETGATE(idt[T_TSS],0,GD_KT,invalid_TSS,0);
SETGATE(idt[T_SEGNP],0,GD_KT,segment_not_present,0);
SETGATE(idt[T_STACK],0,GD_KT,stack_exception,0);
SETGATE(idt[T_GPFLT],0,GD_KT,general_protection_fault,0);
SETGATE(idt[T_PGFLT],0,GD_KT,page_fault,0);
SETGATE(idt[T_RES],0,GD_KT,reserved,0);
SETGATE(idt[T_FPERR],0,GD_KT,floating_point_error,0);
SETGATE(idt[T_ALIGN],0,GD_KT,alignment_check,0);
SETGATE(idt[T_MCHK],0,GD_KT,machine_check,0);
SETGATE(idt[T_SIMDERR],0,GD_KT,SIMD_float_point_error,0);
SETGATE(idt[T_SYSCALL],0,GD_KT,system_call,0);|
```

检测结果：

```
gmake[1]: Entering directory '/home/diana/os/lab3'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/diana/os/lab3'
divzero: OK (1.5s)
softint: OK (1.5s)
badsegment: OK (1.5s)
Part A score: 30/30
```

至此，partA 成功完成！

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)?

Answer:

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?)

因为 JOS 的中断处理程序在真正的处理之前要将中断号放入内核栈以组织成 Trapframe 的结构，目前的中断机制不能给出足够的信息来在一个函数里面识别不同的中断，如果所有中断都跳到同一个处理程序，那么就无法区分是哪个中断调用进来的，也就无法正确设置它们的中断号了，除非 CPU 可以自动保存中断向量这样可以分辨中断的数据给函数使用。

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)?

将 kern/init.c 中载入的第一个程序设置为 user_softint:

```

    #if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE(TEST, ENV_TYPE_USER);
    #else
        // Touch all you want.
        //ENV_CREATE(user_hello, ENV_TYPE_USER);
        ENV_CREATE(user_softint, ENV_TYPE_USER);
    #endif

```

int 14 Page Fault 并非一个用户可以直接调用的中断，在实现的时候是给与其内核级别 DPL(在中断向量里设置的14号 Page fault 的调用权限是0)，所以用户直接访问的时候会产生保护错误，从而转到 int 13 General Protection。

如下图所示：

```

diana@ubuntu: ~/os/lab3
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefbffffbc
TRAP frame at 0xf019f000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfdf0
  oesp 0xefbffffdc
  ebx 0x00000000
  edx 0x00000000
  ecx 0x00000000
  eax 0x00000000
  es 0x----0023
  ds 0x----0023
  trap 0x0000000d General Protection
  err 0x00000072
  eip 0x00800036
  cs 0x----001b
  flag 0x00000082
  esp 0xeebdfdf0
  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>

```

如果将14号 Page fault 权限打开，即中断向量中权限设置为3给用户调用，`SETGATE(idt[T_PGFLT], 0, GD_KT, page_fault, 3)`

这样 QEMU 成功抛出了 Page Fault：

```

diana@ubuntu: ~/os/lab3
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfff0
TRAP frame at 0xefbfff0
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xefbfff0
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0x00000000
es 0x----0023
ds 0x----0023
trap 0x0000000e Page Fault
cr2 0x00000000
err 0x00800038 [kernel, read, not-present]
eip 0x0000001b
cs 0x----0082
flag 0xeebdfd0
esp 0x00000023
Incoming TRAP frame at 0xefbfff48
TRAP frame at 0xefbfff48
edi 0x00000000
esi 0xf0105872
ebp 0xefbfff94
oesp 0xefbfff68
ebx 0xefbfff0
edx 0x000003d5
ecx 0x000003d4
eax 0x00000000
es 0x----0010
ds 0x----0010
trap 0x0000000e Page Fault
cr2 0xefc00000
err 0x00000000 [kernel, read, not-present]
eip 0xf010347c
cs 0x----0008
flag 0x00000096
kernel panic at kern/trap.c:194: unhandled trap in kernel

```

Page fault 中断是需要压入错误代码的,但是用户用 `int` 指令调用中断是不会压入错误代码的。而在 `kern/trapentry.S` 中为 Page fault 指定的中断处理程序默认认为系统为我们放入了错误码,所以不会补齐。那么当我们用 `int` 调用中断处理程序造成的后果就是栈中没有放入错误码,注意上面打印出信息的关于 `err` 开始,就发生了错位。

内核栈的压入结构要对应 `Trapframe`,如果少了一个成员,我们再把这个 `Trapframe` 传到 `trap()` 中进行处理,那么在访问 `Trapframe` 中的最后一个 `DWORD`(也就是访问 `ss` 寄存器时),肯定就访问到 `KSTACKTOP` 之之上的空间上去了,打印 `ss` 时候不可以正常打印出东西,故在第一个 Page fault 处理程序中(由 `softint` 中使用 `int` 指令调用),在打印最后一个 `ss` 时,又发生了 Page fault 中断(由系统产生)。

即如果内核允许用户直接调用 `int 14`,用户调用时不会压入错误码,会导致栈错位。

二、缺页中断、断点异常, 系统调用

理论准备

(1) 缺页中断 (Page Faults)

缺页中断, 又称为页面失效, 中断号为 14 (T_PGFLT), 当处理器遇到一个缺页中断时, 它将引起错误的线性地址存储到一个特殊的寄存器 CR2 中。

(2) 断点异常 (Breakpoint Exception)

断点异常, 中断号为 3 (T_BRKPT), 通常是用来使调试器可以在程序的任意位置设置断点, 调试器临时将该处的指令替换为一字节的 int3 软中断指令。在 JOS 中, 我们扩大它的使用范围, 将其设置为原始的伪系统调用 (primitive pseudo-system call), 使得任何用户都可以调用它来引用 JOS 的内核监视器 (JOS kernel monitor)。在用户模式下实现的 lib/panic.c 中的 panic() 函数在输出信息之后就会使用 int3 中断。

(3) 系统调用 (System Call)

用户进程通过使用系统调用来要求内核做一些事情。当用户进程调用系统调用时, 处理器进入内核模式, 处理器和内核协作保存用户进程的状态, 内核执行适当的代码来启动系统调用, 然后恢复到用户进程。关键在于: 在于用户进程如何获得内核的注意, 以及根据系统的不同它如何设定哪一个调用是要执行的。

在 JOS 内核, 我们将使用 int 指令, 它可以引起处理器中断。也就是说, 使用 int \$0x30 作为系统调用的中断。代码中已经将 T_SYSCALL 定义为 0x30。之后, 必须设置中断描述符使用户进行可以引发中断。注意中断 0x30 不能从硬件产生, 所以应使用用户代码产生。

在寄存器中传递系统调用的编号和参数。这样, 就不需要在用户进程的堆栈或者指令序列中搜寻了。系统调用的编号在 %eax 中, 参数 (最多 5 个) 则依次保存在 %edx、%ecx、%ebx、%edi 和 %esi 中。内核则将返回值保存在 %eax 中。lib/syscall.c 中的 syscall() 函数中, 用户调用系统调用的汇编代码已经完成。

(4) 启动用户模式 (User-mode Startup)

用户程序在 lib/entry.S 的顶部开始运行。进行一些设置之后, 这部分代码会调用 lib/libmain.c 中的 libmain() 函数。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 结构的 env 指针 (注意 lib/entry.S 已经定义了 envs 来指向 UENVS) 可查看 inc/env.h, 使用 sys_getenvid。

之后, `libmain()` 调用 `umain`, 假设当前是运行在 `user/hello.c` 中的 `hello` 程序。注意在打印出 “hello, world” 之后, 它会尝试访问 `thisenv->env_id`。这也是它为什么会在上面的实验中出现缺页中断的原因。下面设置了 `env` 后, 就不会出错了。

(5) 缺页和存储保护 (Page faults and memory protection)

存储保护是操作系统的一项至关重要的功能。通过存储保护, 操作系统可以保证一个错误的程序不会影响到其他的程序或者操作系统本身。

技术上讲, 操作系统提供存储保护也依赖于硬件的支持。OS 保存着哪些虚拟地址可用而哪些不可用。如果一个程序访问了不可用的虚拟地址, 或者是它无权访问的地址, 处理器就会停止执行该程序的这条指令并带着这些信息陷入内核。如果这个问题是可以被修复的, 内核就修复该问题并继续运行原来的程序; 如果是不可修复的, 那么该程序就无法继续运行了, 因为这条指令永远无法执行过去。

作为一个可修复错误的例子, 想象一下可自动扩展的栈。在许多系统中, 内核初始时只会分配一个单独的页, 如果程序错误的访问栈以下的页, 内核就会自动分配这些页并让程序继续。这样, 内核只需要尽可能的分配程序所需要的栈空间, 而程序则可以假想拥有一个无限大的堆栈。

系统调用对存储保护也提出了一个有趣的问题。大多数系统调用接口允许用户程序给内核传递指针。这些指针指向需要被读写的用户内存区。内核在执行系统调用时就会用到这些指针。这里, 主要有如下两个问题:

在内核的缺页错误要比在用户程序中更为严重。如果内核出现了缺页错误, 那么通常是内核的 bug, 而且这些处理函数会停止内核 (以及整个系统)。在系统调用中, 当内核引用指向用户空间的指针时, 我们需要一种方法来确认这些引用所造成的缺页错误都是位于用户程序的内核比用户程序有着更高的访问特权。用户程序会请求内核读取或写入内核中某些用户程序不能读写的区域。如果内核不够仔细的话, 一个错误的或者是恶意的程序就可以欺骗内核而进行各种非法的操作, 甚至完全破坏内核的完整性。

基于以上这些原因, 内核在引用用户程序所传递的指针时必须十分仔细。

JOS 需要仔细审核所有由用户程序传递到内核的指针, 然后在内核中执行。也就是说, 当指针被传递时, 内核就会检查该地址是否是用户程序可访问的, 以及用户程序的页表是否允许这些内存操作。

由此, 内核就永远不会因为引用了一个用户提供的指针和出现缺页错误。如果出现缺页错误, 它将会立刻停止。

问题与作业

作业4

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`.

设计思路:

根据中断号把 `page fault` 分发到 `trap.c` 里的 `page_fault_handler()` 函数，在此函数里把这个唯一的 `env` 给毁掉了，即用户程序出了 `page fault` 直接销毁。

代码实现:

在 `trap.c` 函数 `trap_dispatch()` 中添加如下代码:

```
if (tf->tf_trapno == T_PGFLT)
{ //根据中断号把page fault分发到trap.c里的trap_fault_handler函数即可
    page_fault_handler(tf);
    return;
}
```

检测结果:

```
faultread: OK (1.5s)
faultreadkernel: OK (1.4s)
faultwrite: OK (1.5s)
faultwritekernel: OK (1.5s)
```


作业 5

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.

代码实现:

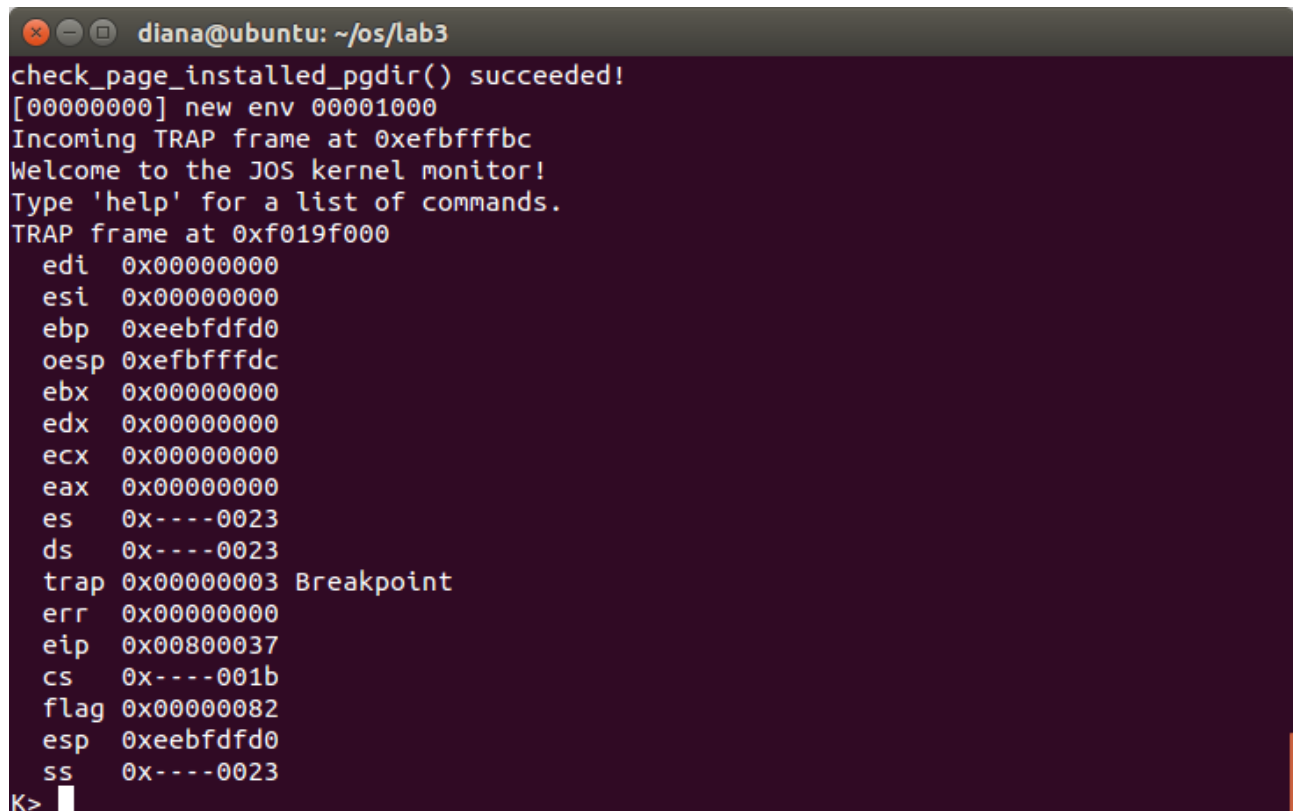
在 `trap.c` 的 `trap_init()` 中修改 breakpoint 权限:

```
SETGATE(idt[T_DIVIDE], 0, GD_KT, divide_error, 0);
SETGATE(idt[T_DEBUG], 0, GD_KT, debug, 0);
SETGATE(idt[T_NMI], 0, GD_KT, nmi, 0);
SETGATE(idt[T_BRKPT], 0, GD_KT, break_point, 3);
SETGATE(idt[T_OFLOW], 0, GD_KT, overflow, 0);
SETGATE(idt[T_BOUND], 0, GD_KT, bounds, 0);
```

在 `trap.c` 函数 `trap_dispatch()` 中添加如下代码:

```
if (tf->tf_trapno==T_BRKPT) //breakpoint
{
    monitor(tf);
    return;
}
```

检测结果:



```
diana@ubuntu: ~/os/lab3
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefbffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf019f000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfd0
  oesp 0xefbffffdc
  ebx 0x00000000
  edx 0x00000000
  ecx 0x00000000
  eax 0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000082
  esp  0xeebdfd0
  ss   0x----0023
K>
```

make grade:

```
faultread: OK (1.5s)
faultreadkernel: OK (1.4s)
faultwrite: OK (1.5s)
faultwritekernel: OK (1.5s)
breakpoint: OK (1.5s)
```

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?

Answer:

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?

IDT 的 DPL 为0时为内核级别的调用权限,所以用户直接访问的时候会产生保护错误,从而转到 int 13 General Protection。DPL 设置为3赋予其用户调用权限,则可以由用户直接调用,产生的是 int 3 break point exception。

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

中断权限机制避免了某些中断直接被用户调用而因此产生栈错位(如上面介绍的 user/softint 执行过程),它是一种保护机制。

CPU 会根据当前 cs 寄存器里的 CPL 和 GDT 的段描述符的 DPL, 以确保中断服务程序是高于当前程序的, 如果这次中断是编程异常(如: int 80h 系统调用), 那么还要检查 CPL

和 IDT 表中中断描述符的 DPL，以保证当前程序有权限使用中断服务程序，这可以避免用户应用程序访问特殊的陷阱门和中断门。

作业 6

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。

代码实现:

在 trap.c 的 trap_init() 中修改 syscall 权限:

```
SETGATE(idt[T_ALIGN], 0, GD_KT, alignment_check, 0);
SETGATE(idt[T_MCHK], 0, GD_KT, machine_check, 0);
SETGATE(idt[T_SIMDERR], 0, GD_KT, SIMD_float_point_error, 0);
SETGATE(idt[T_SYSCALL], 0, GD_KT, system_call, 3);
```

在 trap.c 的 trap_dispatch() 中添加如下代码来处理系统调用中断:

```
//在寄存器中传递系统调用的编号和参数。这样,就不需要在用户进程的堆栈或者指令序列中搜寻了.
//系统调用的编号在%eax中,参数(最多5个)则依次保存在%edx、%ecx、%ebx、%edi和%esi中.
//内核则将返回值保存在%eax中
if (tf->tf_trapno == T_SYSCALL)
{
    tf->tf_regs.reg_eax =
        syscall(tf->tf_regs.reg_eax, tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx,
            tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
    return;
}
```

完成 kern/syscall.c 文件中的 syscall() 函数:

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
    // Call the function corresponding to the 'syscallno' parameter.
    // Return any appropriate return value.
    // LAB 3: Your code here.
    int ret = 0;
    switch (syscallno)
    {
        case SYS_cputs:
            sys_cputs((char*)a1, a2);
            ret = 0;
            break;
        case SYS_cgetc:
            ret = sys_cgetc();
            break;
        case SYS_getenv:
            ret = sys_getenv(a1);
            break;
        case SYS_env_destroy:
            sys_env_destroy(a1);
            ret = 0;
            break;
        default:
            ret = -E_INVAL; //系统调用号是无效的
    }
    return ret;
    //panic("syscall not implemented");
}
```

检测结果:

在内核运行 user/hello 程序(make run-hello):

```

diana@ubuntu: ~/os/lab3
[00000000] new env 00001000
Incoming TRAP frame at 0xefbffffbc
hello, world
Incoming TRAP frame at 0xefbffffbc
[00001000] user fault va 00000048 ip 00800048
TRAP frame at 0xf019f000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebdfdf0
  oesp 0xefbffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000048
  err  0x00000004 [user, read, not-present]
  eip  0x00800048
  cs   0x----001b
  flag 0x00000092
  esp  0xeebdfdb8
  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>

```

可以看到在控制台输出“hello world”，然后会引起一个用户模式下的缺页中断。

运行测试可以通过 testbss 测试：

```

faultread: OK (1.5s)
faultreadkernel: OK (1.4s)
faultwrite: OK (1.5s)
faultwritekernel: OK (1.5s)
breakpoint: OK (1.5s)
testbss: OK (1.5s)

```

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。

设计思路：

在 libmain 用刚才写好的系统调用取得该 env 的 envid，然后根据 envid 得到 Env 结构（使用 ENVX 宏获取 UENVS 数组的索引）。

代码实现：

在 lib/libmain.c 中：

```
void
libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = 0;
    //用刚才写好的系统调用取得该env的envid，然后根据envid得到Env结构即可
    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();
}
```

检测结果：

make run-hello 后：

```

diana@ubuntu: ~/os/lab3
diana@ubuntu:~/os/lab3$ make run-hello
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
/home/diana/os/qemu-6.828-1.7.0/i386-softmmu/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log

(process:18962): GLib-WARNING **: /build/glib2.0-ajuDY6/glib2.0-2.46.1/./glib/gmem.c:482: custom memory allocation vtable not supported
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
Incoming TRAP frame at 0xefbfffbc
hello, world
Incoming TRAP frame at 0xefbfffbc
i am environment 00001000
Incoming TRAP frame at 0xefbfffbc
[00001000] exiting gracefully
[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>

```

可以看到未发生缺页中断。

Make grade 后:

```

faultread: OK (1.5s)
faultreadkernel: OK (1.4s)
faultwrite: OK (1.5s)
faultwritekernel: OK (1.5s)
breakpoint: OK (1.5s)
testbss: OK (1.5s)
hello: OK (1.5s)

```

可以看到通过 hello 测试.

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。

代码实现及检测结果：

首先是在 `kern/trap.c` `page_fault_handler()` 中处理内核态中抛出页面错误：

```
// LAB 3: Your code here.
if ((tf->tf_cs&3) == 0)
    panic("Kernel-mode page fault!");
```

然后是检测线性地址的页面是否有效: 在 `kern/pmap.c: user_mem_check()` 中

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    uint32_t begin = (uint32_t) ROUNDDOWN(va, PGSIZE);
    uint32_t end = (uint32_t) ROUNDUP(va+len, PGSIZE);
    uint32_t i;
    for (i = (uint32_t)begin; i < end; i+=PGSIZE)
    {
        pte_t *pte = pgdir_walk(env->env_pgdir, (void*)i, 0);

        if ((i>=ULIM) || !pte || !(*pte & PTE_P) || ((*pte & perm) != perm))
        {
            user_mem_check_addr = (i<(uint32_t)va?(uint32_t)va:i);
            return -E_FAULT;
        }
    }

    return 0;
}
```


然后在 kern/syscall.c 的 sys_cputs() 中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, PTE_U);

    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

此时运行 user/buggyhello:make run-buggyhello

看到如下输出:

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

Make grade 后:

```
gmake[1]: Entering directory '/home/diana/os/lab3'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/diana/os/lab3'
divzero: OK (2.3s)
softint: OK (1.5s)
badsegment: OK (1.5s)
Part A score: 30/30

faultread: OK (1.5s)
faultreadkernel: OK (1.4s)
faultwrite: OK (1.5s)
faultwritekernel: OK (1.5s)
breakpoint: OK (1.5s)
testbss: OK (1.5s)
hello: OK (1.5s)
buggyhello: OK (1.4s)
buggyhello2: OK (1.5s)
evilhello: OK (1.4s)
Part B score: 50/50

Score: 80/80
```

到此为止是成功的。

修改 kern/kdebug.c 中的 debuginfo_eip(), 在 usd、stabs、stabstr 调用 user_mem_check。完成对 stab 的检查。

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

make run-breakpoint:

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
Incoming TRAP frame at 0xefbfffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01a0000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfd0
  oesp 0xefbfffdc
  ebx 0x00000000
  edx 0x00000000
  ecx 0x00000000
  eax 0xeec00000
  es 0x----0023
  ds 0x----0023
  trap 0x00000003 Breakpoint
  err 0x00000000
  eip 0x00800037
  cs 0x----001b
  flag 0x00000082
  esp 0xeebdfd0
  ss 0x----0023
K> 
```

使用 backtrace 命令

```
K> backtrace
Stack backtrace:
  ebp efbfff20 eip f01008f3 args 00000001 efbfff38 f01a0000 00000000 f017db00
    kern/monitor.c:156: monitor+283
  ebp efbfff90 eip f0103688 args f01a0000 efbfffb0 00111000 00000082 00000000
    kern/trap.c:199: trap+162
  ebp efbfffb0 eip f010379f args efbfffb0 00000000 00000000 eebfd0d0 efbfffdc
    kern/syscall.c:70: syscall+0
  ebp eebfd0d0 eip 0080007d args 00000000 00000000 eebfd0ff 00800053 00000000
    lib/libmain.c:28: libmain+68
Incoming TRAP frame at 0xefbffe9c
kernel panic at kern/trap.c:272: Kernel-mode page fault!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

可以看到出现页错误。

查看 breakpoint 程序函数调用栈出现页错误原因是:用户栈显示到了最顶端,再往上是
没有映射的空间,访问这些空间会导致页错误。

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
```

make run-evilhello:

```
[00000000] new env 00001000
Incoming TRAP frame at 0xefbfffbc
Incoming TRAP frame at 0xefbfffbc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

至此,作业完成!

三、组员分工

第一周：

阅读作业中涉及的相关资料，进行理论准备。

第二、三周：

两人完成实验的完整过程，遇到问题一起探讨解决，过程中每人均完成一份报告的草本。

第四周：

两人相互交流，完善改进代码，总结实验收获，完成实验最终报告。