OS Lab3 实验报告

（郑懿 1611361）

一、 实验准备

本次试验在 Lab2 基础上进行操作，在了解了用户进程相关知识后，对本次试验进行多次尝试，并结合网上内容，完成本次试验。
在实验准备阶段，不知道什么原因，在 monitor.c 和 pmap.c 中出现了很多 head 以及<<<<<之类的乱码，在随后百度过程中，可以得知，原来是本地代码与下拉代码差异所导致的，由于 lab2 做了一些挑战题，更改了部分本不需要更改的代码，具体结构如下图：

```
<<<<<<< HEAD
本地代码
=======
拉下来的代码
>>>>>>>
```

二、 实验过程

PART A

练习 1:
首先，我们打开 kern/pmap.c 重的 mem_init()，发现有一段注释写着：

LAB 3: Your code here.

要求是新建一个 envs 指针指向一个范围是 NENV 存放 Env 的数组，所以代码如下：

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

随后分配内存，要求 envs 数组要求用户只可读，根据 lab2 对于内存分配的知识可以写出如下代码：

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

练习 2:

env_init():

作用是初始化 envs 数组，构建 env_free_list 链表，由于 env[0]是链表头，所以从后往前循环，注释里也要求将 envs 置位 free，envs_ids 为 0，所以代码如下：

```
// 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_status = ENV_FREE;
                envs[i].env_link = env_free_list;
                env_free_list = &envs[i];
        }
        // Per-CPU part of the initialization
        env_init_percpu();
}

// Load GDT and segment descriptors.
void
env_init_percpu(void)
```

env_setup_vm():

该函数有一个参数 struct Env *e，初始化虚拟地址分布，如果返回值为 0，代表成功，如果返回小于 0，代表分配失败，失败可能包括没有足够地址分配。我们所写的代码只需要分配物理页作为页目录使用，并继承内核页目录，将 UVPT 映射到当前环境页目录物理地址 e->env_pgdir 处，因为新增关联所以 ref 需要加一，代码如下：

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

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

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

        // LAB 3: Your code here.
        e->env_pgdir = (pde_t *)page2kva(p);
        p->pp_ref++;
        memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
        // UVPT maps the env's own page table read-only.
        // Permissions: kernel R, user R
        e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

        return 0;
}
```

region_alloc():

该函数有三个参数：sturct Env *e 需要操作的用户环境；void *va 虚拟地址；size_t len 长度。目的是操作 e->env_pgdir，为[va,a+len]分配物理空间。还要求不可以初始化页表，要求可以被用户和内核写，并要 panic 如果分配失败。首先我们设置起点和终点，随后，分配一个物理页，如果分配失败，进行 panic，如果分配成功，借用 lab2 函数 page_insert 将我们物理页添加进去，并且根据要求进行添加，如果添加失败，再次恐慌。代码如下：

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
        // LAB 3: Your code here.
        // (But only if you need it for load_icode.)
        //
        // Hint: It is easier to use region_alloc if the caller can pass
        //   'va' and 'len' values that are not page-aligned.
        //   You should round va down, and round (va + len) up.
        //   (Watch out for corner-cases!)
        void* start = (void *)ROjUNDDOWN((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(" region alloc, allocation failed.");

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

load_icode():

该函数有两个参数：struct Env *e 需要操作的用户环境；uint8_t *binary：可执行用户代码的起始地址。目的是加载 binary 开始的 ELF 文件。通过上网了解，可以通过 ELF_MAGIC 来确定是否读取的事 ELF 文件，如果不是 panic 报错。随后检查可否打开该 ELF 文件，否则 panic 报错。然后通过 ELFHDR->e_phoff 获取程序头距离 ELF 文件的偏移，ph 指向的就是程序头的起始位置，相当于一个数组，随后通过设置 cr3 寄存器切换到该进程的页目录 env_pgdir。然后 env_tf->tf_eip 指向 header 的 e_entry，即程序初始位置。随后通过 region_alloc 分配每个程序段的内存，并按 segment 将代码存入相应内存，加载完成后，再次将 cr3 设置回 kern_pgdir 地址。

根据注释，我们还能得知，只有 ph->p_type==ELF_PROG_LOAD 时候，才可以加载 segments，他们虚拟地址和内存可以分别在 ph->p_va 和 ph->p_memsz 中找到，所以通过 region_alloc 分配。并且用 memmove 拷贝相关部分，清零剩余部分。如果 memsz<filesz，恐慌报错。代码如下：

```
..
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);
        }
}
```

随后，设置一页作为程序初始在指定虚拟地址，代码如下：

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

        // LAB 3: Your code here.
        region_alloc(e,(void *)(USTACKTOP-PGSIZE), PGSIZE);
}
```

env_create():

该函数有两个参数：unit8_t *binary 将要加载的可执行文件起始部分；enum EnvType type 用户环境类型。目的是从 env_free_list 链表取出一个 Env 结构，加载从 binary 地址开始处的 ELF 可执行文件的该 Env 结构。代码如下：

```
//
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.
        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;
}

//
// Frees env e and all memory it uses.
//
```

env_run():

该函数有一个参数：struct Env *e 当前执行的用户环境。作用是执行当前用户环境。如果这是第一次调用该函数，curenv 是 NULL。第一步：先判断现在是否有当前环境，如果有将当前环境返回到 ENV_RUNNABLE，

把当前环境置位 e，然后把当前环境状态设置为 ENV_RUNNING，更新 env_runs，然后把 lcr3 寄存器存储当前环境地址。第二步是用 env_pop_tf()还原环境的寄存器并在环境中进入用户模式。代码如下：

```
//
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.
        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;
}

//
// Frees env e and all memory it uses.
//
```

当我们做完练习 2 后，调用 qemu 会有提示，如图所示：

```
[00000000] new env 00001000
EAX=00000000 EBX=00000000 ECX=0000000d EDX=eebfde88
ESI=00000000 EDI=00000000 EBP=eebfde60 ESP=eebfde54
EIP=00800add EFL=00000092 [--S-A--] CPL=3 II=0 A20=1 SMM=0 HLT=0
ES =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =001b 00000000 ffffffff 00cffa00 DPL=3 CS32 [-R-]
SS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
DS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0028 f017da20 00000067 00408900 DPL=0 TSS32-avl
GDT=     f011b320 0000002f
IDT=     f017d200 000007ff
CR0=80050033 CR2=00000000 CR3=003bc000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

根据上网查询可以知道，这是由于用户程序 user/hello.c 中调用 cprintf 输出 hello world，会用到指令 int 0x30。而此时由于没有中断向量表，当 CPU 收到系统调用中断，会发现没有处理程序，于是会报 general protection 异常，于是最后成为 triple fault。然后会导致 CPU 不断重启，为了方便调试，增加了补丁，就会报错。

练习 3:

学习异常和中断的理论知识。

IDT 可以驻留在物理内存中的任何位置。 处理器通过 IDT 寄存（IDTR）定位 IDT。

Figure 9-1. IDT Register and Table

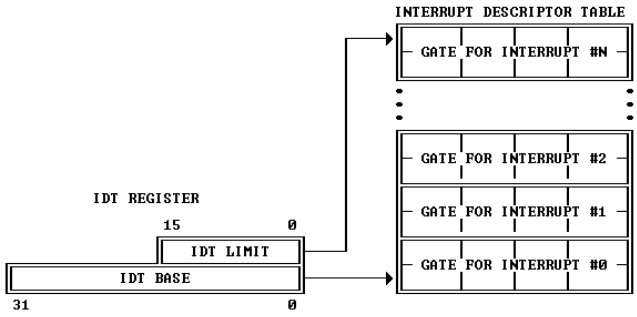
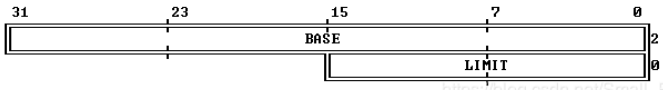
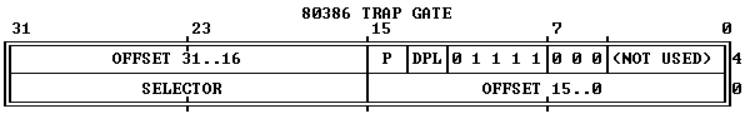


Figure 9-2. Pseudo-Descriptor Format for LIDT and SIDT



IDT 包含了三种描述子:任务门、中断门、陷阱门



每个 entry 为 8bytes，有以下关键 bit：
16~31：code segment selector
0~15 & 46-64：segment offset （根据以上两项可确定中断处理函数的地址）
Type （8-11）：区分中断门、陷阱门、任务门等
DPL：Descriptor Privilege Level， 访问特权级
P：该描述符是否在内存中

练习 4:
为了完成中断向量表初始化以及异常/中断处理，阅读了 inc/trap.h，发现了现在我们需要添加的有 0-31 号以及 48 号，其中 9 号和 15 号被保留，添加 48 号中断是为了方便系统调用，后面的题会用到，具体如图：

```
// 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          // aligment 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
```

所以，我们在 trap_init()中添加这些中断，代码如下：

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

随后，我们将这些中断对应相应 trapframe 结构，通过阅读 trap.h 中对结构 trapframe 了解，代码如下：

```
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));
```

可以看出 trapframe 存储的是当前寄存器的值，然后存入相应 IDT，GDT 值已经对应中断名称，这里的 SEGATA 找到一下定义：

#define SETGATE(gate, istrap, sel, off, dpl)

其中参数:

istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.

sel: 代码段选择子 for interrupt/trap handler

off: 代码段偏移 for interrupt/trap handler

dpl: 描述符特权级

代码如下：

```
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
```

紧接着，在 kern/trapentry.S 补充_alltraps，首先将 trap.c 和 trap.h 中的通过 TRAPHANDLE 压入中断向量和错误码，然后在_alltraps 中压入旧的 DS,ES 寄存次和通用寄存器的值，并将 DS，ES 的值设置为 GD_KD，又因为 DS，ES 是段寄存器，不支持立即数，所以 AX 寄存器中转数据。随后将 ESP 里的值压入内核栈，最后调用 trap（tf）函数。代码如下：

```
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)

TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)

/*
 * 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
```

至此完成练习 4，通过 make grade 确认完成 divzero，softint，basegment。如图所示：

```
divzero: OK (0.9s)
softint: OK (1.0s)
badsegment: OK (0.9s)
Part A score: 30/30
```

问题 1：

- 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？

答：因为不同的中断和异常需要不同的处理，这么做为了方便区分不同中断异常，可以根据压入的中断向量和错误码，来确定如何对应相应的中断和异常。

- 你有没有额外做什么事情让 user/softint 这个程序按预期运行？打分脚本希望它产生一个一般保护错(陷阱 13)，可是 softint 的代码却发送的是 int $14。为什么 这个产生了中断向量 13 ？如果内核允许 softint 的 int $14 指令去调用内核中断向量 14 所对应的的缺页处理函数，会发生什么？

答：因为当前系统运行在用户状态下，特权级别为 3，INT 是系统指令，特权为 0，会引起 General Protection Exception。

PART B

练习 5：

通过会看 trapframe 结构，我们可以发现有一个 uint32_t tf_trapno，所以再看中断类型，所以如果对应的是中断向量 14:T_PGFLT，那么调用 page_fault_handler()，代码如下：

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

结果如下，完成 faultread、faultreadkernal、faultwriet、faultwritekernal,：

```
faultread: OK (1.0s)
    (Old jos.out.faultread failure log removed)
faultreadkernel: OK (1.1s)
    (Old jos.out.faultreadkernel failure log removed)
faultwrite: OK (1.0s)
    (Old jos.out.faultwrite failure log removed)
faultwritekernel: OK (0.9s)
    (Old jos.out.faultwritekernel failure log removed)
```

练习 6：

类似于练习 5，代码如下：

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

结果如下，完成 breakpoint：

```
breakpoint: OK (1.7s)
    (Old jos.out.breakpoint failure log removed)
```

问题 2：

- 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？

答：我们在初始化时候，可以将 DPL 设置为 3，也就是防止如果用户程序跳转去执行内核程序，为了避免一般保护错误，也就是为了能让程序跳转到所指向的程序那里继续执行，因此，我们要将 DPL（规定访问该段的权限级别）设置为 3，防止在 CPL（当前进程的权限级别）值为 3 时候，DPL 小于 CPL 产生一般保护错误。

- 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。

DPL 的设置可以限制用户对指令的使用。

练习 7：

通过阅读 lib/syscall.c，其中内联汇编部分，如图所示：

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

“volatile”表示编译器不要优化代码，后面的指令 保留原样，其中”=a”表示”ret”是输出操作数；“i”=立即数。最后一个子句告诉汇编器这可能会改变条件代码和任意内存位置。memory 强制 gcc 编译器假设所有内存单元均被汇编指令修改，这样 cpu 中的 registers 和 cache 中已缓存的内存单元中的数据将作废。cpu 将不得不在需要的时候重新读取内存中的数据。这就阻止了 cpu 又将 registers，cache 中的数据用于去优化指令，而避免去访问内存。

随后，对 kern/trap.c 进行编辑。在刚才的 switch 语句中再加一个 T_SYSCALL 的分支，代码如下：

```
case 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);
        break;
```

随后，在 kern/syscall.c 中，根据阅读前面的函数以及联想 lib/syscall.c 中的函数，我们很容易写下代码，只需要找好对应参数个数即可，代码如下：

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

        //panic("syscall not implemented");

        switch (syscallno) {
        case SYS_cputs:
                sys_cputs((char *)a1, a2);
                return 0;
        case SYS_cgetc:
                return sys_cgetc();
        case SYS_getenvid:
                return sys_getenvid();
        case SYS_env_destroy:
                return sys_env_destroy(a1);
        default:
                return -E_NO_SYS;
        }
}
```

其实整个过程，就是用户调用 lib/syscall.c 中的各个函数，该函数会调用 syscall()函数，随后会发生 int 0x30 系统调用中断，进入 trap，并 dispatch 到了 kern/syscall.c 中，调用制定函数。所

以大部分 lib 中函数并没有实际操作，而是通过系统调用进入内核进行操作。

所以当我们 make run-hello 后，果然报错，如图所示：

```
[00001000] user fault va 00000048 ip 0080004a
TRAP frame at 0xf01a1000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  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  0x0080004a
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  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.
←[0;32;40m He110 World×=3 u=-268435626K>
```

随后再次用 make grade 测试，发现完成 testbss，如图所示：

```
testbss: OK (1.1s)
```

练习 8:

根据题目所说，我们又阅读 user/hello.c 代码，其中第二句，后面有一句：thisenv->env_id。发现以下：

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

我们应该修改 libmain()用来初始化全局指针 thisenv 以指向 envs[]中对应此时的环境。所以通过刚才实现的系统调用获得此环境对应的 Env，代码如下：

```
        // set thisenv to point at our Env structure in envs[].
        // LAB 3: Your code here.
        thisenv = &envs[ENVX(sys_getenvid())];
```

实现结果如图示：

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
hello, world
Incoming TRAP frame at 0xeffffbc
i am environment 00001000
Incoming TRAP frame at 0xeffffbc
[00001000] exiting gracefully
[00001000] free env 00001000
```

我们发现可以实现后面一句的输出，随后 make grade 测试，结果如图：

```
hello: OK (0.9s)
    (Old jos.out.hello failure log removed)
```

练习 9:

首先，我们要在 trap.c 中实现 panic，因为我们刚才在发生缺页情况下，调用 page_fault_handler()函数，所以去修改一下该函数，因为如果在内核状态下 CPL 为 0，所以增加代码如下：

```
if((tf->tf_cs & 3) == 0)
{
        panic("page_fault in kernel mode, fault address %d\n", fault_va);
}
```

随后，查看 pmap.c 增加 user_mem_check 内容，先阅读 user_mem_assert()内容，并没有增添对函数了解，所以阅读关于 check 的注释内容，我们将要测试范围内所有页面和“len/PGSIZE”，“len/PGSIZE+1”或“len/PGSIZE+2”页面。如果该页面满足：1、该地址低于 ULIM；2、页面表为其授予权限，则用户程序可以访问虚拟地址。如果有错误，请将“ user_mem_check_addr”变量设置为第一个错误的虚拟地址。所以代码如下：

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

随后，我们要在 syscall.c 调用 user_mem_assert()，代码如图所示：

```
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);
}
```

经过验证，结果如图所示：

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

随后，在 kern/kdebug.c 中加入语句，如图所示：

```
// 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))
        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))
        return -1;

if (user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U))
        return -1;
```

然后运行了 breakpoint，但没办法在监视器里使用 backtrace，没有添加过类似指令。

练习 10:

运行后，确实没有引起 panic，结果如图所示：

```
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
[00001000] user_mem_check assertion failure for va f010000c
```

至此，所有实验结束。

三、 实验结果

最后通过 make grade 完成全部实验，结果如下：

```
make[1]: Leaving directory `/home/brian/lab1/src/lab3'
divzero: OK (1.1s)
softint: OK (1.0s)
badsegment: OK (0.9s)
Part A score: 30/30

faultread: OK (1.4s)
faultreadkernel: OK (0.7s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.0s)
breakpoint: OK (1.1s)
testbss: OK (1.9s)
hello: OK (2.0s)
buggyhello: OK (2.0s)
buggyhello2: OK (1.0s)
evilhello: OK (0.9s)
Part B score: 50/50

Score: 80/80
```

通过本次试验加深了对用户进程的理解。加深了对中断和异常认识，并了解了操作系统如何对自己进行保护。