操作系统实习报告

曹胜操 - 1500012838

Lab 3: User Environments

在这一Lab中，我们将会实现一个基本的用户模式运行环境。我们需要设置一些环境相关的数据结构、创建新的一个用户环境、载入程序然后把它运行起来。JOS也需要处理系统调用以及其它的一些异常。

Part A: User Environments and Exception Handling

JOS内核通过`Env`这个数据结构来追踪用户环境。有一些全局变量与此相关：

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

`Env`结构内部的定义如下：

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

JOS中的环境将“线程”和“地址空间”结合在一起，前者主要是由保存的寄存器(`env_tf`)来定义的，而后者主要是由页目录和页表(`env_pgdir`)来定义的。要运行一个环境，内核需要将两者都设置好。

> Exercise 1. 修改`kern/pmap.c`中的`mem_init()`来分配并映射`envs`数组。它包含了`NENV`个`Env`结构，应当被映射到`UENVS`，且是用户只读的。
>
> 保证`check_kern_pgdir()`运行成功。

类似于Lab2中对`pages`数组的操作。我们可以写出：

```
...
    uint32_t envs_size = NENV * sizeof(struct Env);
    envs = (struct Env*)boot_alloc(envs_size);
    memset(envs, 0, envs_size);
...
    boot_map_region(kern_pgdir, UENVS, ROUNDUP(envs_size, PGSIZE),
            PADDR(envs), PTE_U);

...
```

由于我们还没有一个文件系统，要运行用户环境，JOS只能把一个静态的二进制文件作为ELF可执行映像文件直接嵌入内核。`GNUmakefile`会生成一些二进制文件到`obj/user`下，而在`kern/Makefrag`中，这些二进制文件直接就被链接进了内核的可执行文件中。在`kern/init.c`的`i386_init()`中，我们可以看到：

```
...
    ENV_CREATE(user_hello, ENV_TYPE_USER);
...
    env_run(&envs[0]);
...
```

这就是创建一个环境然后运行它的代码。但是，其中的关键函数目前还没有被实现。

> Exercise 2. 在`env.c`中实现以下函数：
>
> `env_init()`：初始化`envs`数组中所有的`Env`，将它们添加进`env_free_list`。然后它会调用`env_init_percpu()`，这会设置段寄存器中的特权级。

这里我们只需要用一个循环把`envs`的元素依次初始化并加入链表即可。需要将`env_status`初始化为`ENV_FREE`，将`env_id`初始化为`0`。

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    for (int i = 0; i < NENV; ++i)
    {
        envs[i].env_status = ENV_FREE;
        envs[i].env_id = 0;
        envs[i].env_link = &envs[i + 1];
    }
    env_free_list = envs;
    envs[NENV - 1].env_link = NULL;

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

> `env_setup_vm()`：分配一个页目录给新的环境。初始化地址空间中的内核部分。

我们需要做两件事。一是分配一个物理页用于储存页目录，并把它的虚拟地址赋给`e->env_pgdir`，物理页的`pp_ref`增加。二是初始化页目录的内容。由于`UTOP`之上的内容是所有`Env`所共享的，可以直接根据`kern_pgdir`来复制。而`UTOP`之下的内容目前还没有分配，初始化为零。

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

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

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

    // LAB 3: Your code here.
    e->env_pgdir = (pde_t*)page2kva(p);
    ++p->pp_ref;
    for (i = 0; i < PDX(UTOP); ++i)
        e->env_pgdir[i] = 0;
    for (i = PDX(UTOP); i < NPDENTRIES; ++i)
        e->env_pgdir[i] = kern_pgdir[i];

    // 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()`：为环境分配并映射物理内存。

传入参数中的`va`和`len`可能不是页对齐的，但我们分配物理页的地址需要页对齐。因此我们先对虚拟地址的端点对齐，然后逐页加入页目录。对于用户来说，这些物理内存应是既可读又可写的。

```
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 *va_start = ROUNDDOWN(va, PGSIZE);
    void *va_end = ROUNDUP(va + len, PGSIZE);
    void *va_i;
    struct PageInfo *pp;
    for (va_i = va_start; va_i != va_end; va_i += PGSIZE)
    {
        if (!(pp = page_alloc(0)) ||
                page_insert(e->env_pgdir, pp, va_i, PTE_U | PTE_W))
            panic("region_alloc: Allocation fails!\n" );
    }
}
```

> `load_icode()`：解析ELF二进制映像文件，很类似于Boot loader所做的事情。然后将ELF文件的内容载入一个新的环境的用户地址空间中。

模仿Boot loader，通过传入的ELF文件头地址进行解析。首先通过`elfhdr->e_magic`判断是否为合法ELF文件，然后逐段载入。其中，`ph->p_type == ELF_PROG_LOAD`时才会载入，且应有`ph->p_filesz <= ph->p_memsz`。多余的部分占用内存但并不出现在ELF文件中，应在内存中初始化为零。在载入的前后，我们还应该设置`cr3`寄存器，因为我们是在用户地址空间中载入的，之前或之后都在内核地址空间中。此外，我们还需要初始化用户环境的入口地址和栈空间。

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
    // Hints:
    //  Load each program segment into virtual memory
    //  at the address specified in the ELF section header.
    //  You should only load segments with ph->p_type == ELF_PROG_LOAD.
    //  Each segment's virtual address can be found in ph->p_va
    //  and its size in memory can be found in ph->p_memsz.
    //  The ph->p_filesz bytes from the ELF binary, starting at
    //  'binary + ph->p_offset', should be copied to virtual address
    //  ph->p_va.  Any remaining memory bytes should be cleared to zero.
    //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
    //  Use functions from the previous lab to allocate and map pages.
    //
    //  All page protection bits should be user read/write for now.
    //  ELF segments are not necessarily page-aligned, but you can
    //  assume for this function that no two segments will touch
    //  the same virtual page.
    //
    //  You may find a function like region_alloc useful.
    //
    //  Loading the segments is much simpler if you can move data
    //  directly into the virtual addresses stored in the ELF binary.
    //  So which page directory should be in force during
    //  this function?
    //
    //  You must also do something with the program's entry point,
    //  to make sure that the environment starts executing there.
    //  What?  (See env_run() and env_pop_tf() below.)

    // LAB 3: Your code here.
    struct Elf *elfhdr = (struct Elf*)binary;
    if (elfhdr->e_magic != ELF_MAGIC)
        panic("load_icode: Invalid ELF!\n");
    struct Proghdr *ph, *eph;
    ph = (struct Proghdr*)(binary + elfhdr->e_phoff);
    eph = ph + elfhdr->e_phnum;
    lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ++ph)
        if (ph->p_type == ELF_PROG_LOAD)
        {
            if (ph->p_filesz > ph->p_memsz)
                panic("load_icode: Bad memory size!\n");
            region_alloc(e, (void*)ph->p_va, ph->p_memsz);
            memcpy((void*)ph->p_va, binary + ph->p_offset, ph->p_filesz);
            memset((void*)ph->p_va + ph->p_filesz, 0, ph->p_memsz - ph->p_filesz);
        }
    e->env_tf.tf_eip = elfhdr->e_entry;
    lcr3(PADDR(kern_pgdir));

    // 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()`：用`env_alloc`分配用户环境，再用`load_icode`载入ELF文件。

```
void
env_create(uint8_t *binary, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;
    int r = env_alloc(&e, 0);
    if (r)
        panic("env_create: env_alloc() fails with error %e!\n" , r);
    load_icode(e, binary);
    e->env_type = type;
}
```

> `env_run()`：开始在用户模式下运行一个环境。

主要是做以下几件事情：设置之前正在运行环境的状态、设置当前运行环境为给定、切换地址空间、恢复寄存器从而进入用户模式运行该环境。

```
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.

    // Hint: This function loads the new environment's state from
    //  e->env_tf.  Go back through the code you wrote above
    //  and make sure you have set the relevant parts of
    //  e->env_tf to sensible values.

    // LAB 3: Your code here.
    if (curenv && 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));
    env_pop_tf(&curenv->env_tf);

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

到用户代码被调用为止，我们依次调用了如下函数：

- `start`（`kern/entry.S`）
- `i386_init`（`kern/init.c`）
  - `cons_init`
  - `mem_init`
  - `env_init`
  - `trap_init`
  - `env_create`
  - `env_run`
    - `env_pop_tf`

编译内核，在QEMU中运行。系统会正常运行，直到`hello`程序通过`int`指令产生了一次系统调用。但是这时候JOS还没有设置硬件来允许这种系统调用，最终会触发Triple fault，导致系统直接重启。

通过GDB，我们可以观察到：进入`env_pop_tf`之后，处理器会通过`iret`指令进入用户模式，开始运行用户程序的代码。而到了`int $0x30`这一指令，它试图通过系统调用在控制台显示字符，而JOS还没有实现这一功能。接下来我们将会实现异常和系统调用的处理，这样内核才能从用户模式中恢复对处理器的控制。

> Exercise 3. 阅读80386 Programmer's Manual中的Chapter 9, Exceptions and Interrupts部分，或者IA-32 Developer's Manual的Chapter 5。

我们需要熟悉x86中的中断/异常处理机制。这部分内容将在接下来的内容中被用到。

异常和中断都属于“保护控制转移”，会导致处理器从用户模式切换为内核模式。按照一般的定义，中断是由外部事件异步产生的（例如I/O），而异常则是由正在执行的代码同步产生的（例如除零）。为了保证这种控制转移的保护性，中断/异常机制并不会使得正在运行的代码能够任意选择进入内核的位置和方式。在x86平台上有两种机制共同提供这种保护性。

- 中断描述符表(Interrupt Descriptor Table, IDT)：产生中断或异常时，只能进入由内核决定的某些特定的入口。x86允许有256个不同的这样的入口，每个都对应一个中断向量，这是由产生中断的来源所决定的。CPU会通过这个向量在中断描述符表中找到对应的入口，包括需要载入的`EIP`和`CS`寄存器值。
- 任务状态段(Task State Segment, TSS)：处理器需要一个被保护的区域，来储存中断或异常发生时旧的处理器状态，从而能够在处理完后恢复到原来的状态。当特权级被切换时，处理器也会转换到一个内核内存的栈中。任务状态段就会指定这个栈所在的段选择符和地址。发生中断或异常时，处理器会在栈中压入旧的`SS` `ESP` `EFLAGS` `CS` `EIP`，以及一个可能的错误码。然后它才会从中断描述符中载入`CS` `EIP`，把`SS` `ESP`设为新的栈。任务状态段其实有很多功能，JOS只是用其中的`ESP0` `SS0`域来定义应该切换到的内核栈。

接下来，我们将会扩展JOS来处理内部产生的x86异常（0到31号）以及软中断（48号）。

以除零异常为例，我们来看看JOS应当做什么。

- 处理器切换到由TSS中的`SS0` `ESP0`定义的内核栈（JOS中具体是`GD_KD` `KSTACKTOP`）。
- 处理器向栈中依次压入异常参数，包括`SS` `ESP` `EFLAGS` `CS` `EIP`。
- 除零对应中断向量0，处理器从IDT中读入这一项，将`CS` `EIP`设为对应的handler入口。
- handler函数接管并开始处理异常，例如终止用户环境。

对某些异常，除了上述5个字的异常状态外，处理器还可能压入一个错误码。80386手册上对此有详细的描述。

发生中断或异常时，如果处理器已经在内核模式下，那么它就只会在同一个栈中继续压入更多的值。这样内核就可以处理嵌套的异常了。在这种情况下，它并没有切换栈，所以也不需要压入`SS` `ESP`。如果因为某种原因，处理异常时无法压入旧的状态，那么处理器就没有办法恢复了，只能重启。

接下来我们需要为JOS设置IDT。`inc/trap.h` `kern/trap.h`中有中断/异常处理相关的重要定义。在`trapentry.S`中每个异常或中断都需要有自己的handler，`trap_init()`应当初始化这些handler的地址。每个handler会在栈中建立一个`struct Trapframe`保存旧的状态，然后调用`trap.c`中的`trap()`。`trap()`则会处理这些异常/中断，或者交给某个特定的处理函数。

> Exercise 4. 编辑`trapentry.S`和`trap.c`实现以上功能。`trapentry.S`中的宏`TRAPHANDLER`和`TRAPHANDLER_NOEC`以及`inc/trap.h`中的`T_*`定义会很有帮助。你需要在`trapentry.S`中用这些宏为`inc/trap.h`中的每个trap添加入口，以及提供`_alltraps`。还需要利用`SETGATE`宏修改`trap_init()`来将`idt`初始化，指向这些入口。
>
> `_alltraps`应当做这些事：利用`pushal`压入一些值，使之看起来像是一个`struct Trapframe`；将`GD_KD`载入`%ds` `%es`；`pushl %esp`来向`trap()`传入参数；调用`trap()`。
>
> 完成后，`make grade`应该会在`divzero` `softint` `badsegment`测试中成功。

首先看`trapentry.S`。其中宏定义`TRAPHANDLER`和`TRAPHANDLER_NOEC`可以定义一个全局的函数符号（方便后面在`trap.c`中找到对应的入口），然后前者会压入一个trap number，后者则会在之前先压入一个0来补位（对于CPU不会自动压入错误码的trap），再然后跳转到`_alltraps`做一些通用的事情，如上所述。

查询手册，我们可以确认哪些trap是会压入错误码的，哪些是不会压入错误码的，然后分别对应处理即可。`syscall`是我们会在后面实现的系统调用。

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
TRAPHANDLER_NOEC(t_divide, T_DIVIDE)
TRAPHANDLER_NOEC(t_debug, T_DEBUG)
TRAPHANDLER_NOEC(t_nmi, T_NMI)
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT)
TRAPHANDLER_NOEC(t_oflow, T_OFLOW)
TRAPHANDLER_NOEC(t_bound, T_BOUND)
TRAPHANDLER_NOEC(t_illop, T_ILLOP)
TRAPHANDLER_NOEC(t_device, T_DEVICE)
TRAPHANDLER(t_dblflt, T_DBLFLT)
TRAPHANDLER(t_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
```

在`trap.c`中我们会用到`SETGATE`宏，它是在`inc/mmu.h`中定义的。

```
// 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;        \
}
```

它用于设置中断描述符中的两种门：陷阱门(trap gate)和中断门(interrupt gate)，区别是否改变`IF`标志位（是否允许被中断）。另外的参数包括`sel` `off`（用于指定handler的入口）和`dpl`（用于指定可以触发该门的特权级）。接下来我们就根据具体的情况在`trap.c`中为之前所写的handler添加门。

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

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

    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();
}
```

有些特别的是对于断点和系统调用，我们可以允许用户模式的代码直接通过`int`指令来触发中断门。

> Question 1. 为什么每个异常/中断要有自己独立的handler？如果它们都用同一个handler，什么特性是无法实现的？

我们在`trapentry.S`中设置的handler会区分出不同的trap的编号（`struct Trapframde`中的`tf_trapno`）。如果没有它的话，后续的处理就根本无法区分不同的trap，更不能针对它们做不同的处理。这显然是不合理的。

> Question 2. 要使得`user/softint`正常工作，你需要做些什么吗？评测脚本期望它最终产生一个一般保护错误（13号trap），但代码中只有`int $14`。为什么它应该产生13号trap？如果内核允许它调用缺页异常的handler会怎么样？

我们为缺页异常设置的DPL是只有内核才能直接产生。如果在用户代码中`int $14`，由于权限不够，只会产生一个一般保护错误。

查阅资料后，了解到`int`指令并不会在内核栈中压入错误码，但是对于缺页异常是应当有这个错误码的。如果我们允许用户代码直接通过`int $14`调用缺页异常缺页异常的handler，由于这个错误码的错位，会导致一系列的问题。另一方面，内核也无法判断是否真的产生了一个缺页异常，用户的调用是否合理。因此，内核不应允许用户代码直接通过`int $14`指令来处理缺页异常。

Part B: Page Faults, Br eakpoints Ex ceptions, and System Calls

接下来我们继续完成一些重要的操作系统功能。

首先是缺页异常。处理器接受到时，它会将产生错误的虚拟地址存在一个特殊的控制寄存器`CR2`中。在`trap.c`中，我们用`page_fault_handler()`来处理缺页异常。

> Exercise 5. 修改`trap_dispatch()`，将缺页异常分派给`page_fault_handler()`。完成后，`make grade`应该会在`faultread` `faultreadkernel` `faultwrite` `faultwritekernel`测试中成功。

把`trap_dispatch()`中的判断改为`switch`语句，来处理各种不同的情况。部分代码会在之后的内容中说明。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
    int ret;
    switch(tf->tf_trapno)
    {
    case T_PGFLT:
        page_fault_handler(tf);
        break;
    case T_BRKPT:
        monitor(tf);
        break;
    case T_SYSCALL:
        ret = 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);
        if (ret < 0)
            panic("trap_dispatch: syscall fails with error %e!\n" , ret);
        tf->tf_regs.reg_eax = ret;
        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;
        }
    }
}
```

断点异常允许调试器来在程序代码中插入断点。JOS中，我们用它来调用内核的监视器（可以把它当做一个调试器）。用户模式下的`panic()`在显示消息后会有一个`int3`指令来作为断点。

> Exercise 6. 修改`trap_dispatch()`，使得断点异常调用内核监视器。完成后，`make grade`应该会在`breakpoint`测试中成功。

代码已经在上面展示了。

> Question 3. 这个测试会产生一个断点异常，或者一般保护错误。这取决于如何在IDT中初始化对应的那个门。为什么?

我们在前面通过`SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 3)`来进行初始化，DPL为用户模式，用户代码即可使用`int3`来触发断点。相反如果我们把这个DPL改为内核模式的话，就只能在内核代码中使用`int3`，用户代码中的`int3`会导致一般保护错误。

> Question 4. 这些机制的意义是什么？特别是考虑`user/softint`测试所做的事情？

通过IDT和TSS的设置，我们能够确保中断/异常的处理是在内核所控制的范围之内的。特别是对于权限的要求，使得我们避免了用户代码对于中断/异常机制的不正当利用，从而维护了操作系统的安全性（至少是目前看来）。

接下来我们再看系统调用。用户进程有时候会需要内核去帮它们做一些事情，这时就需要系统调用了。处理器会进入内核模式，软件和硬件共同保存下用户进程的状态，内核会执行适当的代码从而完成系统调用的功能，然后继续用户进程。在JOS中，我们使用`int $0x30`来产生系统调用。用户进程还应该通过寄存器传入系统调用的编号以及参数，其中`%eax`为调用号，`%edx` `%ecx` `%ebx` `%edi` `%esi`为至多五个的参数，返回值将存在`%eax`中。产生系统调用的汇编代码在`lib/syscall.c`的`syscall()`中。

```
static inline int32_t
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
uint32_t a5)
{
    int32_t ret;

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

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

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

    return ret;
}
```

> Exercise 7. 为中断`T_SYSCALL`在`kern/trapentry.S`和`kern/trap.c`中增加一个handler。修改`trap_dispatch()`，通过调用`syscall()`来处理系统调用。最后在`kern/syscall.c`中实现`syscall()`。
>
> 运行`user/hello`，应该可以看到系统调用的正确结果。完成后，`make grade`应该会在`testbss`测试中成功。

handler我们已经在前面的代码中见过了。在`trap.c`的`trap_dispatch()`中，我们需要正确地传递参数和处理返回值。

```
...
    case T_SYSCALL:
        ret = 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);
        if (ret < 0)
            panic("trap_dispatch: syscall fails with error %e!\n" , ret);
        tf->tf_regs.reg_eax = ret;
        break;
...
```

在`kern/syscall.c`中，我们只需要根据调用编号分别处理不同的函数即可。

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

那么，`kern/syscall.c`和`lib/syscall.c`有何区别呢？后者是提供给用户代码使用的，它通过汇编代码描述了调用编号和参数如何存放在寄存器中，如何通过`int $0x30`产生系统调用。而前者则是内核中在处理这个对应的trap所做的具体操作，由内核模式来实现系统调用的需求。

用户程序从`lib/entry.S`开始运行，经过一些初始化后它会调用`lib/libmain.c`中的`libmain()`。在这里需要初始化全局指针`thisenv`来指向当前环境。可以参见`inc/env.h`，使用`sys_getenvid`。之后，用户程序中的`umain()`会被调用。

> Exercise 8. 修改对应的代码，重启内核。应该可以看到`user/hello`有正确的输出结果，然后它会调用`sys_env_destroy()`来退出。完成后，`make grade`应该会在`hello`测试中成功。

修改`lib/libmain.c`中的`libmain()`即可。

```
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();
}
```

最后我们再来看看缺页异常和内存保护的关系。操作系统通常需要硬件支持才能实现对内存的保护，当程序试图访问非法的虚拟地址时，处理器会停止这个程序并且陷入内核。内核会根据这个错误是否可以被修复来进行具体的处理。例如，对于一个可扩展的栈来说，一开始它的大小只有一个物理页，如果程序超出了这个范围就会导致错误，而内核可以为它分配更多的内存，从而修复这一错误，继续执行程序。

然而，对于系统调用来说会有一些问题。用户程序可能将一些对于用户模式非法的指针交给内核去处理，但是在这样的系统调用中，内核必须区分出来缺页异常和权限问题是由内核本身造成的，还是由这个非法指针造成的。因此，内核需要谨慎对待用户程序提供的指针。我们接下来就实现对这种指针的检查。

> Exercise 9. 修改`kern/trap.c`，如果内核模式下发生缺页，就产生panic。我们可以通过`tf_cs`的低位来检查模式。
>
> 实现`kern/pmap.c`中的检查`user_mem_check`。修改`kern/syscall.c`来进行这个检查。
>
> 重启内核，运行`user/buggyhello`。用户环境应当被终止，但内核不能panic，因为这个错误是由用户程序造成的。
>
> 最后，修改`kern/kdebug.c`中的`debuginfo_eip`来调用`user_mem_check`进行检查。运行`user/breakpoint`，你应该可以运行`backtrace`来看到内核因缺页panic之前的backtrace。这个缺页是怎么产生的?

在`kern/trap.c`的`page_fault_handler()`中添加一个判断，即可产生内核模式下的panic。

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

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

    // 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);
}
```

在`kern/pmap.c`中，`user_mem_check()`和`user_mem_assert()`来检查用户模式所试图使用的内存。如果出现问题，则会输出错误信息，终止用户环境。需要注意的是页对齐和对地址大小、权限的检查。

```
//
// Check that an environment is allowed to access the range of memory
// [va, va+len) with permissions 'perm | PTE_P'.
// Normally 'perm' will contain PTE_U at least, but this is not required.
// 'va' and 'len' need not be page-aligned; you must test every page that
// contains any of that range.  You will test either 'len/PGSIZE',
// 'len/PGSIZE + 1', or 'len/PGSIZE + 2' pages.
//
// A user program can access a virtual address if (1) the address is below
// ULIM, and (2) the page table gives it permission.  These are exactly
// the tests you should implement here.
//
// If there is an error, set the 'user_mem_check_addr' variable to the first
// erroneous virtual address.
//
// Returns 0 if the user program can access this range of addresses,
// and -E_FAULT otherwise.
//
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    void *va_start = (void*)ROUNDDOWN(va, PGSIZE);
    void *va_end = (void*)ROUNDUP(va + len, PGSIZE);
    void *va_i;
    perm |= PTE_U | PTE_P;
    for (va_i = va_start; va_i != va_end; va_i += PGSIZE)
    {
        if ((uintptr_t)va_i >= ULIM)
        {
            if (va_i == va_start) va_i = (void*)va;
            user_mem_check_addr = (uintptr_t)va_i;
            return -E_FAULT;
        }
        pte_t *ptep = pgdir_walk(env->env_pgdir, va_i, 0);
        if (!ptep || (*ptep & perm) != perm)
        {
            if (va_i == va_start) va_i = (void*)va;
            user_mem_check_addr = (uintptr_t)va_i;
            return -E_FAULT;
        }
    }
    return 0;
}
```

在`kern/syscall.c`中，需要用到地址的是`sys_cputs()`。我们进行一个检查即可。

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
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);
}
```

运行`user/buggyhello`的结果如下，可见内核并没有问题。

```
[00001000] user_mem_check assertion failure for va 00000001
[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.
```

在`kern/kdebug.c`中修改`debuginfo_eip`进行内存检查。

```
...
        // 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;
...
```

运行`user/breakpoint`后，再试图调用`backtrace`会导致内核缺页异常。

```
K> backtrace
Stack backtrace:
  ebp efffff20  eip f0100df5  args 00000001 efffff38 f01b5000 00000000 f0173840
         kern/monitor.c:273: monitor+276
  ebp efffff90  eip f0103bda  args f01b5000 effffbc f03ff004 00000082 00000000
         kern/trap.c:192: trap+172
  ebp effffb0  eip f0103cfc  args effffbc 00000000 00000000 eebfdfd0 effffdc
         kern/syscall.c:69: syscall+0
  ebp eebfdfd0  eip 00800073  args 00000000 00000000 eebfdff0 00800049 00000000
         lib/libmain.c:26: libmain+58
  ebp eebfdff0  eip 00800031  args 00000000 00000000Incoming TRAP frame at 0xeffffea4
kernel panic at kern/trap.c:269: page_fault_handler: Kernel mode page fault!

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

> Exercise 10. 重启内核，运行`user/evilhello`。用户环境应当被终止，但内核不能panic。

与之前的类似，只是这里触发非法内存问题的原因是权限不足。运行结果是这样的。

```
[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.
```

> Challenge! 修改JOS的内核监视器，允许继续进程的执行（例如在`int3`之后），从而能够实现单步调试的功能。你需要理解`EFLAGS`中一些位的意义。

在我们之前的实现中，如果用户程序调用了`int3`指令产生一个断点异常，那么最终会在`trap_dispatch()`中调用`monitor()`，从而进入内核监视器。我们可以为监视器再添加几个命令，这样就可以像其它某些异常一样，经过处理之后返回到用户程序中继续执行。

首先我们需要关注`EFLAGS`寄存器中的`TF`位，这个位是第8位，对应着`0x100`。`TF`的意思是Trap flag，当该位被置为`1`时，代表进入单步调试模式，每执行一条指令就会产生一个异常，对应着我们在`inc/trap.h`中定义的`#define T_DEBUG 1`。

模仿我们曾经常用的调试器，可以在`kern/monitor.c`中，实现以下三个功能对应的函数：

- `mon_continue()`：继续执行，直到遇到下一个断点或者退出。
- `mon_stepinto()`：单步调试，在执行完下一条指令后暂停。
- `mon_kill()`：中止执行，退出用户程序。

需要注意的地方有几个。对于前两个函数，需要进行类似于`trap()`中对`curenv`的检查，然后通过`env_run(curenv)`来继续执行。我们还需要修改其中保存的`EFLAGS`寄存器的`TF`位。对于`mon_continue()`而言，不需要单步调试，清除该位即可。对于`mon_stepinto()`，我们要设置该位，使得在下一步的指令执行完成后触发异常，从而通过异常处理机制实现单步调试。而`mon_kill()`只需要调用`env_destroy(curenv)`。还有一点，`TF`造成的异常是`T_DEBUG`，而用户程序中的断点`int3`造成的异常是`T_BRKPT`。我们对两者的处理其实应该是一样的，需要在`trap_dispatch()`中把`T_DEBUG`也加进去。以下是`kern/monitor.c`中的具体实现：

```
int
mon_continue(int argc, char **argv, struct Trapframe *tf)
{
    if (curenv && curenv->env_status == ENV_RUNNING)
    {
        cprintf("Continue environment %x\n", curenv->env_id);
        curenv->env_tf.tf_eflags &= ~0x100;
        env_run(curenv);
    }
    else
    {
        cprintf("No suitable environment!\n");
        return 0;
    }
    return 0;
}

int
mon_stepinto(int argc, char **argv, struct Trapframe *tf)
{
    if (curenv && curenv->env_status == ENV_RUNNING)
    {
        cprintf("Stepinto environment %x\n", curenv->env_id);
        curenv->env_tf.tf_eflags |= 0x100;
        env_run(curenv);
    }
    else
    {
        cprintf("No suitable environment!\n");
        return 0;
    }
    return 0;
}

int
mon_kill(int argc, char **argv, struct Trapframe *tf)
{
    if (curenv)
    {
        cprintf("Kill environment %x\n", curenv->env_id);
        env_destroy(curenv);
    }
    return 0;
}
```

接下来，我们自己编写一个比`user/breakpoint.c`更长的测试文件`user/mybreakpoint.c`（需要加入到`kern/Makefrag`的规则中）：

```
void
umain(int argc, char **argv)
{
    cprintf("Hello\n");
    asm volatile("int $3");
    cprintf("world!\n");
    asm volatile("int $3");
    asm volatile("movl $0, %eax\n \
                  movl $1, %ebx\n \
                  movl $2, %ecx\n \
                  movl $3, %edx\n");
}
```

用户程序运行到第一个`int $3`时，产生断点异常，被处理后进入到监视器中：

```
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffffc
Incoming TRAP frame at 0xeffffffc
Hello
Incoming TRAP frame at 0xeffffffc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x00000006
  eax  0x00000006
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800044
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  ss   0x----0023
```

输入`continue`命令，使之继续执行到第二个`int $3`：

```
K> continue
Continue environment 1000
Incoming TRAP frame at 0xeffffbc
world!
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x00000007
  eax  0x00000007
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800051
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  ss   0x----0023
```

我们尝试一下单步调试，可以看到四个寄存器如何被逐个改变，以及Trap号是和之前的`Breakpoint`不同的`Debug`：

```
K> stepinto
Stepinto environment 1000
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x00000007
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800056
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb8
  ss   0x----0023
K> stepinto
Stepinto environment 1000
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000001
  edx  0xeebfde88
  ecx  0x00000007
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x0080005b
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb8
  ss   0x----0023
K> stepinto
Stepinto environment 1000
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
```

```
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000001
  edx  0xeebfde88
  ecx  0x00000002
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800060
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb8
  ss   0x----0023
K> stepinto
Stepinto environment 1000
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01bc000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000001
  edx  0x00000003
  ecx  0x00000002
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800065
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb8
  ss   0x----0023
```

输入`continue`，用户程序运行到结束为止：

```
K> continue
Continue environment 1000
Incoming TRAP frame at 0xeffffbc
[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.
```

或者也可以输入`kill`，直接中止该程序：

```
K> kill
Kill environment 1000
[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.
```

如果结合上一个Lab中实现的Challenge，我们还可以查看内存中的内容等等，已经可以说是实现了一个调试器的基本功能了。