实验报告 Lab3

10300240039 章超

November 4, 2013

1 User Environments and Exception Handling

因为 JOS 中对于 Environment 提供的语义与传统 UNIX 中的 Process 并不相同。所以在 JOS 中，Environment 与 Process 可以看作是等价的。所以第一部分也就是要我们实现用户的进程。

Kernel 使用`Env`来跟踪每个用户进程（与 OS 中的进程控制块的概念类似）。而且在 Lab3 里面，我们只会创建一个进程。这就是说，Kernel 加载完成后，将控制权交给某个用户进程，用户进程运行结束后就将控制权还给 Kernel。JOS 使用`envs`数组来跟踪所有用户进程，`curenv`来表示当前执行的进程，初始时为空。`env_free_list`来表示空闲进程链表（与空闲页表`page_free_list`表示类似)。

在`kern/env.c`中，`env_init`用于被`i386_init`调用，初始化用户进程环境。`env_create`用于创建用户进程。`env_run`用于 Kernel 将控制权交给执行用户进程，`env_destroy`用于终止用户进程，并将控制权交给 Kernel。

1.1 Environment State

`Env`，也就是进程中，保存了如下信息：

- `env_tf`，保存了进程的寄存器等信息，以便从其他进程或者 Kernel 恢复时能够继续执行，这部分在 Part B 中会用到。
- `env_link`，在`env_free_list`链表中的下一个空闲进程的指针。
- `env_id`，用于标识进程。
- `env_parent_id`，用于标识这个进程的父进程。
- `env_type`，用于表示这个进程的类型，在 Lab3 中只有`ENV_TYPE_USER`这个类型，表示用户进程。
- `env_status`，用于表示这个进程的执行状态。OS 课上讲到过。
- `env_pgdir`，用于表示这个进程虚拟空间的页目录。`env_tf`与`env_pgdir`耦合在一起，用于表示线程和地址空间。

1.2 Allocating the Environments Array

Exercise 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.

这一部分的目的就是为进程在物理内存中分配空间。这个与 Lab2 中分配页目录几乎一样，我们代码如下：

kern/pmap.c

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

一开始做的时候我把权限不小心打错了，于是在 Exercise 8 里面中枪，调试了好久才发现原来 Lab3 描述里就有说需要重新检查一下`pmap.c`的代码。

1.3 Creating and Running Environments

Exercise 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”.

1.3.1 env_init

这个函数所要完成的功能与page_init类似，为每一个Env进行初始化。注释中已经写明了，每个envs中的Env都必须是可用，env_id为 0，而且要加入空闲链表中。值得注意的是，因为i386_init中通过env_run(&envs[0]);执行了第一个用户进程，所以我们必须保证第一个用户进程应该要存放在envs[0]中，也就是说空闲链表在初始化完成以后的头指针为&envs[0]。

代码如下:

kern/env.c

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

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

1.3.2 env_setup_vm

这个函数的功能是，为传入的Env分配页目录，设置env_pgdir并初始化页目录。

根据提示，我们知道在UTOP之上，除了UVPT，所有的用户进程的虚拟空间映射是一致的。在UTOP之下，因为是用户进程的私有空间，所以一开始都置为 0。而对于UVPT，我们可以从kern_pgdir拷贝一份页目录来。这是因为在 Kernel 初始化时，kern_pgdir的相应权限已经设置好，用户是没有写权限的。不用担心用户会破坏这些物理地址的内容。

对于黄睿哲报告中的第一个问题，用户进程是否会恶意修改 Kernel 的页目录，我觉得这个问题是不成立的。首先这个函数的最后一行并没有设置用户可写的权限。另外即使用户可写，我们也只是从 Kernel 的页目录拷贝一份过来，而且相应的权限已经设置好，即使用户进程修改了自己的页目录，也不会直接知道 Kernel 的页目录在哪里。第二个问题，缺少 Kernel 中`kern_pgdir`更改之后的更新机制，我觉得这个倒是有必要考虑的。如果 Kernel 申请了更多的页，那么用户进程是不知道 Kernel 的操作，于是在系统调用的时候，用户进程可能会需要访问 Kernel 新申请的页，从而导致权限错误。

拷贝时候我们需要用的函数有两个选择，一个是`memmove`，另外一个是`memcpy`。通常我们在用户程序中会写`memcpy`，但后来我在查阅资料时发现库函数（也就是我们实现的 JOS）应该尽量使用`memmove`，因为这个函数能够处理解决源区域与目标区域重叠的情况。我一开始写的是`memcpy`，没有发现任何问题，这是因为在`lib/string.c`中,`memcpy`直接调用了`memmove`。所以为了排除隐患，我最后还是改成了`memmove`。

kern/env.c

```c
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.

  // LAB 3: Your code here.
  e->env_pgdir = page2kva(p);
  memmove(e->env_pgdir, kern_pgdir, PGSIZE);
  memset(e->env_pgdir, 0, PDX(UTOP) * sizeof(pde_t));
  p->pp_ref++;

  // 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;
}
```

1.3.3 env_alloc

这个函数不需要我们实现，而且又与`page_alloc`类似。不过这个函数有两点值得我们注意。

一个是`env_id`的设置。一个合理的`env_id`有如下的结构：

inc/env.h

```c
// +1+---------------21-----------------+---------10--------+
// |0|          Uniqueifier             |   Environment     |
// | |                                  |      Index        |
// +------------------------------------+-------------------+
```

也就是说，除了 Index 外，一个env_id还用 21 位来作为标识，用来区分在不同时间创建的相同 Index 的用户进程。而env_id的生成方式为

kern/env.c

```
  generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
  if (generation <= 0)  // Don't create a negative env_id.
    generation = 1 << ENVGENSHIFT;
  e->env_id = generation | (e - envs);
```

我们可以看到generation就是原先的env_id的高 21 位再增加 1，所以generation不断自增，且保证为正。

Env中的段寄存器在这个env_alloc中设置了 DPL = 3，也就是设置了用户模式。

1.3.4 region_alloc

这个函数的功能是为用户进程分配从虚拟地址va开始，长度为len自节的物理页面。需要注意的是，我们需要将va和len分别与PGSIZE对齐。

另外，打印错误的时候可以用%e，这是因为printfmt中实现了基于error_string数组的输出。

kern/env.c

```
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.)

  struct PageInfo* p = NULL;
  uint32_t i = 0;
  int r = 0;
  for (i = ROUNDDOWN((uint32_t)va, PGSIZE); i < ROUNDUP((uint32_t)va + len,
     PGSIZE); i+=PGSIZE) {
    p = page_alloc(0);
    if (p == NULL)
      panic("region_alloc: Out of memory!\n");
    r = page_insert(e->env_pgdir, p, (void*)i, PTE_W | PTE_U | PTE_P);
    if (r < 0)
      panic("region_alloc: %e\n", r);
  }
}
```

1.3.5 load_icode

这个函数的功能是将用户进程的可执行代码读入内存。由于 JOS 目前还没有文件系统，所以采用的方法是将希望运行的用户程序编译后和 Kernel 链接到一起，也就是通过读入 ELF 可执行文件的方式载入。这段代码将 ELF 读入到 ELF 头中指定的虚拟地址。同时将

ELF 头中指定但是不在 ELF 存储空间中的区域设为 0，比如程序的 bss 段（包含静态变量等）。读入的代码可以参考 Bootloader。

我的实现如下：

kern/env.c

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
  // LAB 3: Your code here.
  struct Elf *elf = (struct Elf*)binary;
  struct Proghdr *ph, *eph;
  if (elf->e_magic != ELF_MAGIC)
    panic("load_icode: Elf header is not correct");
  ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
  eph = ph + elf->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: File size is larger than memory size");
      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)
    ;
    }

  // Hint in env_alloc
  e->env_tf.tf_eip = elf->e_entry;

  // 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);
  lcr3(PADDR(kern_pgdir));
}
```

其中有几点需要注意的：

一个是`p_memsz`与`p_filesz`不一样。前者是 ELF 文件在内存中实际占据的空间大小，后者是代码占据的空间大小。所以前者必须大于或等于后者，我在这里先加了这个判断。然后将代码载入，再将前者大于后者的部分置为 0。

第二个是我们需要在读入代码的时候先将`e->envpgdir`设为当前页目录。这是因为我们读入的代码是要载入到用户进程中的，由用户去执行的。在读入完成后我们再恢复 Kernel 的页目录。我不知道黄睿哲为什么会觉得这个设置是没有用的。而且如果将这一步骤去掉，`make grade`是无法通过的。

最后一个是为用户进程分配栈空间，这里暂时只分配一页。

1.3.6 env_create

这个函数的作用非常明确，新建一个用户进程。首先初始化这个用户进程，然后读入代码，最后设置相应的类型。

kern/env.c

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

1.3.7 env_run

这个函数的功能是运行一个用户进程。只需要按照注释一步一步来就好了：如果当前进程与要执行的进程不同，就设置一下当前进程为`ENV_RUNNABLE`，然后设置当前进程为要执行的进程，增加计数，设置页目录。最后调用`env_pop_tf()`。实现如下：

kern/env.c

```c
void
env_run(struct Env *e)
{
  if (curenv != e) {
    if (curenv && curenv->env_status == ENV_RUNNING)
      curenv->env_status = ENV_RUNNABLE;
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs ++;
    lcr3(PADDR(e->env_pgdir));
  }
  env_pop_tf(&(e->env_tf));
}
```

最后一行`env_pop_tf()`的调用值得一看。这里用`%esp`设置为`tf`，用 |popal| 来恢复所有的寄存器值，用`popl`来恢复所有段寄存器，最后用`iret`设置 CS,IP 和 FLAGS。然后函数就不用返回了，因为已经设置了新的 CS,IP。最后一个`panic`只是用来通过编译器的检查。

kern/env.c

```c
void
env_pop_tf(struct Trapframe *tf)
{
  __asm __volatile("movl %0,%%esp\n"
    "\tpopal\n"
    "\tpopl %%es\n"
    "\tpopl %%ds\n"
    "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
    "\tiret"
    : : "g" (tf) : "memory");
  panic("iret failed");  /* mostly to placate the compiler */
}
```

1.4 Handling Interrupts and Exceptions

我们通过 Exercise 3 来了解中断机制。

Exercise 3 Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

中断是外部产生的，有可屏蔽中断 (INTR) 和不可屏蔽中断 (NMI)。异常是内部产生的，处理器异常包括 Fault/Trap/Abort，可编程异常（软件异常），比如`INTO`, `INT 3`, `INT n`等。

9.1 节主要说明了中断和异常由数字表示标识，NMI 和处理器异常标识在 0-31 之间，其他中断和异常标识在 32-255 之间。

9.2 节主要说明了如何开启和屏蔽中断，一个 NMI 会屏蔽其他 NMI，设置 IF 位可以屏蔽 INTR，设置 RF 位可以屏蔽 DEBUG Fault，对`SS`的`MOV`和`POP`操作也会屏蔽一些中断和异常（这是由于要维持`SS`和`ESP`的一致性）

9.3 节介绍了当多个中断和异常同时发生时，中断和异常的优先级和 CPU 响应顺序。

9.4 节介绍了中断描述符表 IDT。IDT 每一项是 8 字节，至多有 256 项，而且可以在物理内存中的任意位置。

9.5 节介绍了中断描述符的具体结构。

9.6 节介绍了中断任务和中断过程，JOS 只涉及到中断过程 (Interrupt Procedure)。中断任务与第 7 章的多道任务有关，这里不详述。

首先，一个中断过程调用 Handler 与`CALL`调用一个 Call Gate 类似。CPU 会将`EFLAGS`, `CS`, `EIP`, `ERROR CODE`, `SS`, `ESP`等压入中断过程栈（中断过程有一个专门的栈）。

然后，从中断过程返回时，调用了`IRET`，它比`RET`还要多弹出一个`EFLAGS`。

再次，中断发生以后 CPU 会自己重置`TF`的值, 防止单步跟踪调试对中断服务的影响。只有当中断过程完成以后,CPU 才会将 |TF| 重新设置成原来的值。从 Interrupt Gate 进入的中断程序还会重新设置 IF 使得其他中断不能打断当前中断过程。而从 Trap Gate 进入的则不会（嵌套中断）。

最后，中断过程的权限必须超过引发中断程序的权限，不然会导致一个保护异常被触发。

9.7 节介绍了错误码的格式。

9.8 节详细描述了每个处理器异常。

9.9 和 9.10 节分别总结了每个处理器异常和错误码。

1.5 Basics of Protected Control Transfer

受保护的控制转移意思是，中断和异常都有可能使得 CPU 从用户态切换到内核态，但不能使用户进程对 Kernel 和其他进程进行干扰。简单地说, 当用户进程需要执行特权指令

或者内核功能时, 或者中断和异常事件发生时, 用户进程不能够自己决定它要进入 Kernel 的哪里去执行; 而是 CPU 提供 IDT 和 TSS（任务状态段）一起保证从用户态进入内核态是受约束的。

1.6 Setting Up the IDT

Exercise 4 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 `idt_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.

这个练习要求我们建立 IDT。IDT 的本质是一个数组。中断处理的流程为：CPU->IDT->handler in `trapentry.S`->`trap(Trapfram* tf)`。

我们的实现分为两个部分，首先要实现 handler。在进入 handler 之前，CPU 已经压入了`EFLAGS`,`CS`,`IP`, 对于一些指令还压入了 Error code。`trapentry.S`中有两个宏，一个是`TRAPHANDLER`和`TRAPHANDLER_NOEC`。后者通过多压入了一个 0 以保证结构一致，然后二者都跳到`_alltraps`进行后续处理。

这里我们需要区别哪些 handler 要用`TRAPHANDLER`，而哪些要用`TRAPHANDLER`。`http://pdos.csail.mit.edu/6.828/2012/readings/i386/s09_10.htm`这里给出了 0-16 的 handler 的 error code 描述，但还是不够。因为`trap.h`中定义了 19 个，我们需要在 IA-32 手册中才能找到 `http://pdos.csail.mit.edu/6.828/2012/readings/ia32/IA32-3A.pdf`。

这里我们还定义了`trap_handlers`，用于形成一个`trap_handlers`数组，从而使`trap.c`的代码更简洁。这些内容都是 Challenge 1 的要求。

`_alltraps`的实现在题目描述中已经有了，先将寄存器值压入栈，使得栈与`Trapframe`结构一致；然后设置好段寄存器；再送入栈顶指针；最后调用`trap`。

`trapentry.S`上的工作如下：

kern/trapentry.S

```
.data
  .globl trap_handlers
trap_handlers:
    .long trhdlr0
    .long trhdlr1
    .long trhdlr2
    .long trhdlr3
    .long trhdlr4
    .long trhdlr5
    .long trhdlr6
    .long trhdlr7
    .long trhdlr8
    .long trhdlr9
    ...
.text
  TRAPHANDLER_NOEC(trhdlr0, 0)
  TRAPHANDLER_NOEC(trhdlr1, 1)
  TRAPHANDLER_NOEC(trhdlr2, 2)
  TRAPHANDLER_NOEC(trhdlr3, 3)
  TRAPHANDLER_NOEC(trhdlr4, 4)
  TRAPHANDLER_NOEC(trhdlr5, 5)
  TRAPHANDLER_NOEC(trhdlr6, 6)
  TRAPHANDLER_NOEC(trhdlr7, 7)
  TRAPHANDLER(trhdlr8, 8)
  TRAPHANDLER_NOEC(trhdlr9, 9)
  TRAPHANDLER(trhdlr10, 10)
  TRAPHANDLER(trhdlr11, 11)
  TRAPHANDLER(trhdlr12, 12)
  TRAPHANDLER(trhdlr13, 13)
  TRAPHANDLER(trhdlr14, 14)
  TRAPHANDLER(trhdlr15, 15)
  TRAPHANDLER_NOEC(trhdlr16, 16)
  TRAPHANDLER(trhdlr17, 17)
  TRAPHANDLER_NOEC(trhdlr18, 18)
  TRAPHANDLER_NOEC(trhdlr19, 19)
    ...
_alltraps:
  pushw $0x0
  pushw %ds
  pushw $0x0
  pushw %es
  pushal

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

  pushl %esp
  call trap
```

对于`trap.c`，我们通过`SET_GATE`来在 IDT 中注册相关的 handler。如果没有使用`trap_handlers`数组，我们就需要为每一个 handler 写一个`extern`声明，就显得不简洁

了。

kern/trap.c

```
void
trap_init(void)
{

  // LAB 3: Your code here.
  extern uint32_t trap_handlers[];
  uint32_t i;
  // Init handlers
  for (i = 0; i <= 255; i++)
    SETGATE(idt[i], 0, GD_KT, trap_handlers[i], 0);
  // Init breakpoint
  SETGATE(idt[T_BRKPT], 0, GD_KT, trap_handlers[T_BRKPT], 3);
  // Init syscall
  SETGATE(idt[T_SYSCALL], 0, GD_KT, trap_handlers[T_SYSCALL], 3);

  // Per-CPU setup
  trap_init_percpu();
}
```

Question

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)?

1. 因为调用 handler 的时候，CPU 并不会显示地“传入”当前中断的标识序号，所以我们必须为每一个异常/中断设置一个独立的 handler。并且我们也不知道 CPU 是否压入了 Error code.

2. 我们 IDT 中设置的 page fault(14) 的中断的 DPL 权限为 0，也就是说只有内核才能产生该中断。因此用户程序越权了，CPU 产生一个 general protection fault(13) 进行保护。如果允许用户程序产生 page fault 中断, 那么恶意用户程序可能会不断地调用, 将物理内存占满。

2 Page Faults, Breakpoints Exceptions, and System Calls

2.1 Handling Page Faults

Exercise 5 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`.

这个练习要求我们完成`page_fault_handler`的处理情况。我们在`trap_dispatch()`中，只需要将中断过程引入到`page_fault_handler()`就可以了，实现如下:

kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
  // Handle processor exceptions.
  // LAB 3: Your code here.
  int r;
  if (tf->tf_trapno == T_PGFLT) {
    page_fault_handler(tf);
    return;
  }
  ...
  // 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;
  }
}
```

2.2 The Breakpoint Exception

Exercise 6 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.

这个练习也比较简单，在 `trap_dispatch()`中，将中断过程引入到 Kernel monitor 就好了，实现如下:

kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
  ...
  if (tf->tf_trapno == T_BRKPT) {
    monitor(tf);
    return;
  }
  ...
}
```

Question

3. 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 `idt_init`). Why? How did you need to set it in order to get the breakpoint exception to work as specified above?

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

3. 产生 break point exception 还是 protection exception 取决于 IDT 的设置。如果在初始化 IDT 时把中断描述符的 DPL 权限级别设为 3，那么用户态下可以产生这个 break point 中断。如果中断描述符的 DPL 设为 0, 那么将产生 protection exception。

4. 这涉及到了 Protected Control Transfer(受保护的控制转移机制) 问题。必须保证不让用户进程对 Kernel 进行干扰。因为 softint 试图产生 page fault 异常，但我们之前设置 IDT 时把其 DPL 设为 0，也就是说这是一个内核级中断/异常，用户程序越权了，所以有 protection fault.

2.3 System calls

Exercise 7 Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `idt_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read `inc/syscall.h`.

Run the `user/hello` program under your kernel (`make run-hello`). It should print ”hello, world” on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get make grade to succeed on the testbss test.

用户可以通过系统调用来使用系统服务，但需要注意的是，用户调用的是`lib/syscall.c`。这里面会产生 system call 中断，从而触发`kern/syscall.c`。

这个练习要求我们实现系统调用。首先，我们在`kern/trapentry.S`中已经加入了 256 个 handler。然后，我们在`trap_init`中设置相应权限。

kern/trap.c

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

再然后，我们在trap_dispatch中添加相应的 handler，并且传入参数:

kern/trap.c

```
  if (tf->tf_trapno == T_SYSCALL) {
    r = syscall(tf->tf_regs.reg_eax,
      tf->tf_regs.reg_edx,
      tf->tf_regs.reg_ecx,
      tf->tf_regs.reg_ebx,
      tf->tf_regs.reg_edi,
      tf->tf_regs.reg_esi);
    tf->tf_regs.reg_eax = r;
    return;
  }
```

最后在kern/syscall.c中实现syscall:

kern/syscall.c

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

2.4 User-mode startup

Exercise 8 Add the required code to the user library, then boot your kernel. You should see user/hello print ”hello, world” and then print ”i am environment 00000800”. user/hello then attempts to ”exit” by calling `sys_env_destroy()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get `make grade` to succeed on the hello test.

这个练习也很简单，只需要加上一行。

```
  thisenv = &envs[ENVX(sys_getenvid())];
```

但不幸的是，我在这里抛出了一个异常。原因是因为在`map_init`时候没有设置好相应页的权限。

2.5 Page faults and memory protection

Exercise 9 Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change kern/syscall.c to sanity check arguments to system calls.

Boot your kernel, running `user/buggyhello`. The environment should be destroyed, and the kernel should not panic. You should see:

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

Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd` , `stabs`, and `stabstr`. If you now run user/breakpoint, you should be able to run `backtrace` from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don’t need to fix it, but you should understand why it happens.

这个练习要求我们实现内存保护，也就是说，当用户程序出错时，比如访问了没有权限的内存，其他用户程序和操作系统都不能出错。首先，我们需要对 page fault 进行判断，如果是在 Kernel mode 下抛出了 page fault，则说明操作系统发生了问题，需要`panic`:

kern/trap.c

```
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) != 3) {
    // My hint: from comments in trap()
    print_trapframe(tf);
    panic("page_fault_handler: Kernel Page Fault");
  }
  ...
}
```

这里我们的判断可以借鉴trap(struct Trapframe *tf)。

然后我们要实现user_mem_check检查函数，实现如下：

kern/pmap.c

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

  return 0;
}
```

需要注意的第一个问题是，我们需要将va和len分别对齐，然后分别调用pgdir_walk获取页表项。判断用户是否有权限访问内存时，先判断内存是否大于ULIM，然后判断对应页是否存在，最后判断权限是否相符。

还要注意的一点时，如果用户没有对虚拟地址va的权限，那么直接返回va就可以，不需要对齐。这是因为这个 Lab 的测试程序就是这样子的！buggyhello.c中有sys_cputs((char*)1, 1);，而测试的时候就直接判断是否输出为r.match('.00001000. user_mem_check assertion failure for va 00000001','.00001000. free env 00001000')。

然后，我们要在kern/syscall.c中的系统调用函数中引入user_mem_check。因为涉及到内存访问的系统调用只有sys_cputs()函数一个，所以我们的实现如下:

kern/syscall.c

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

最后，我们还要回到 Lab 1 的函数，在`debuginfo_eip()`中进行权限检查：

kern/kdebug.c

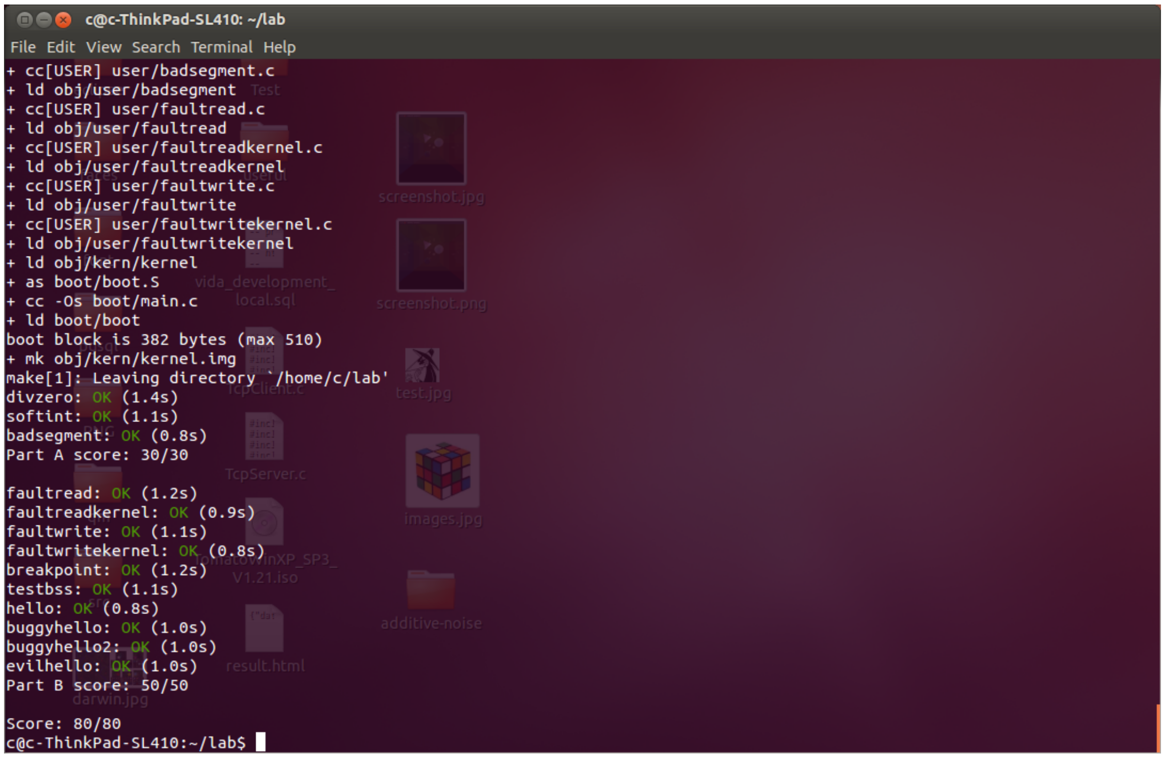
```
int
debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info) {
  ...
    // Make sure this memory is valid.
    // Return -1 if it is not.  Hint: Call user_mem_check.
    // LAB 3: Your code here.
    if (user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U) <
   0)
      return -1;

    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)
      return -1;
    if (user_mem_check(curenv, stabstr, stabstr_end - stabstr, PTE_U) < 0)
      return -1;
  ...
}
```

Exercise 10 Boot your kernel, running `user/evilhello`. The environment should be destroyed, and the kernel should not panic. You should see:

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

最后一个练习只是要我们检验一下是否能通过 evilhello 的测试，只要前面的代码都正确，这一步是不用修改任何代码的。

测试结果如下:



c@c-ThinkPad-SL410: ~/lab
File Edit View Search Terminal Help

```
+ cc[USER] user/badsegment.c
+ ld obj/user/badsegment
+ cc[USER] user/faultread.c
+ ld obj/user/faultread
+ cc[USER] user/faultreadkernel.c
+ ld obj/user/faultreadkernel
+ cc[USER] user/faultwrite.c
+ ld obj/user/faultwrite
+ cc[USER] user/faultwritekernel.c
+ ld obj/user/faultwritekernel
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/c/lab'
divzero: OK (1.4s)
softint: OK (1.1s)
badsegment: OK (0.8s)
Part A score: 30/30

faultread: OK (1.2s)
faultreadkernel: OK (0.9s)
faultwrite: OK (1.1s)
faultwritekernel: OK (0.8s)
breakpoint: OK (1.2s)
testbss: OK (1.1s)
hello: OK (0.8s)
buggyhello: OK (1.0s)
buggyhello2: OK (1.0s)
evilhello: OK (1.0s)
Part B score: 50/50

Score: 80/80
c@c-ThinkPad-SL410:~/lab$
```

3 Challenges

3.1 Challenge 1

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

我们在 Exercise 4 里面就已经这样实现了。

代码再贴一次：

trapentry.S

```
.data
  .globl trap_handlers
trap_handlers:
    .long trhdlr0
    .long trhdlr1
    .long trhdlr2
    .long trhdlr3
    .long trhdlr4
    .long trhdlr5
    .long trhdlr6
    .long trhdlr7
    .long trhdlr8
    .long trhdlr9
    .long trhdlr10
    .long trhdlr11
    .long trhdlr12
    .long trhdlr13
    .long trhdlr14
    .long trhdlr15
    .long trhdlr16
    .long trhdlr17
    .long trhdlr18
    .long trhdlr19
    .long trhdlr20
...

.text

/*
 * Lab 3: Your code here for generating entry points for the different
   traps.
 */
/*
 * http://pdos.csail.mit.edu/6.828/2010/readings/i386/s09_10.htm
 */
TRAPHANDLER_NOEC(trhdlr0, 0)
TRAPHANDLER_NOEC(trhdlr1, 1)
TRAPHANDLER_NOEC(trhdlr2, 2)
TRAPHANDLER_NOEC(trhdlr3, 3)
TRAPHANDLER_NOEC(trhdlr4, 4)
TRAPHANDLER_NOEC(trhdlr5, 5)
TRAPHANDLER_NOEC(trhdlr6, 6)
TRAPHANDLER_NOEC(trhdlr7, 7)
TRAPHANDLER(trhdlr8, 8)
TRAPHANDLER_NOEC(trhdlr9, 9)
TRAPHANDLER(trhdlr10, 10)
TRAPHANDLER(trhdlr11, 11)
TRAPHANDLER(trhdlr12, 12)
TRAPHANDLER(trhdlr13, 13)
TRAPHANDLER(trhdlr14, 14)
TRAPHANDLER(trhdlr15, 15)
TRAPHANDLER_NOEC(trhdlr16, 16)
TRAPHANDLER(trhdlr17, 17)
TRAPHANDLER_NOEC(trhdlr18, 18)
TRAPHANDLER_NOEC(trhdlr19, 19)
TRAPHANDLER_NOEC(trhdlr20, 20)
...

/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
  pushw $0x0
  pushw %ds
  pushw $0x0
  pushw %es
  pushal

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

  pushl %esp
  call trap
```

注：我这里把 0-255 的中断和异常全部绑定了，由于篇幅有限，所以用省略号代替。

kern/trap.c

```
void
trap_init(void)
{

  // LAB 3: Your code here.
  extern uint32_t trap_handlers[];
  uint32_t i;
  // Init handlers
  for (i = 0; i <= 255; i++)
    SETGATE(idt[i], 0, GD_KT, trap_handlers[i], 0);
  // Init breakpoint
  SETGATE(idt[T_BRKPT], 0, GD_KT, trap_handlers[T_BRKPT], 3);
  // Init syscall
  SETGATE(idt[T_SYSCALL], 0, GD_KT, trap_handlers[T_SYSCALL], 3);

  // Per-CPU setup
  trap_init_percpu();
}
```

3.2 Challenge 2

Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the int3, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the `EFLAGS` register in order to implement single-stepping.

Optional: If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 2, this is the stuff of which real kernel debuggers are made.

这个 Challenge 要求我们实现类似于 gdb 的功能，要求实现'continue' 和'step' 功能。对于 continue，我们需要再次调用`env_run`，并且需要将 TF 置为 0，关闭调试中断。对于 step，我们需要将 TF 置为 1，开启调试中断。实现如下：

kern/monitor.c

```c
static struct Command commands[] = {
...
  { "c", "Continue process", mon_c },
  { "si", "Step", mon_si },
};
...
int mon_c(int argc, char **argv, struct Trapframe *tf) {
  extern struct Env* curenv;
  if (tf == NULL || (tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG))
     {
    cprintf("Invalid Trapframe\n");
    return -1;
  }
  tf->tf_eflags &= ~FL_TF;
  env_run(curenv);
  return 0;
}

int mon_si(int argc, char **argv, struct Trapframe *tf) {
  extern struct Env* curenv;
  struct Eipdebuginfo info;
  if (tf == NULL || (tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG))
     {
    cprintf("Invalid Trapframe\n");
    return -1;
  }
  debuginfo_eip(tf->tf_eip, &info);
  cprintf("0x%08x %s:%d: %.*s+%d\n", tf->tf_eip, info.eip_file, info.
    eip_line, info.eip_fn_namelen, info.eip_fn_name, tf->tf_eip-info.
    eip_fn_addr);
  tf->tf_eflags |= FL_TF;
  env_run(curenv);
  return 0;
}
```

其中我们在 step 时模仿 backtrace 输出调试信息。

我们还需要写一个自己的测试程序：

user/cha2.c

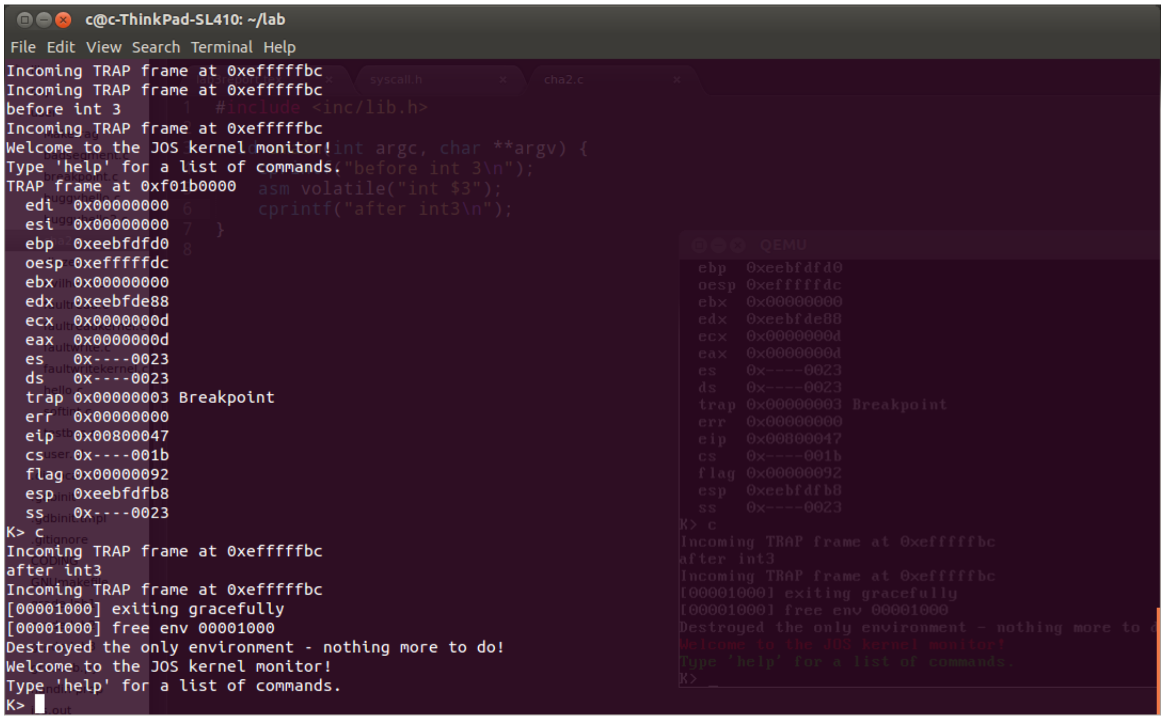
```
#include <inc/lib.h>
void umain(int argc, char **argv) {
  cprintf("before int 3\n");
  asm volatile("int $3");
  cprintf("after int3\n");
}
```

还需要改一下 Makefrag

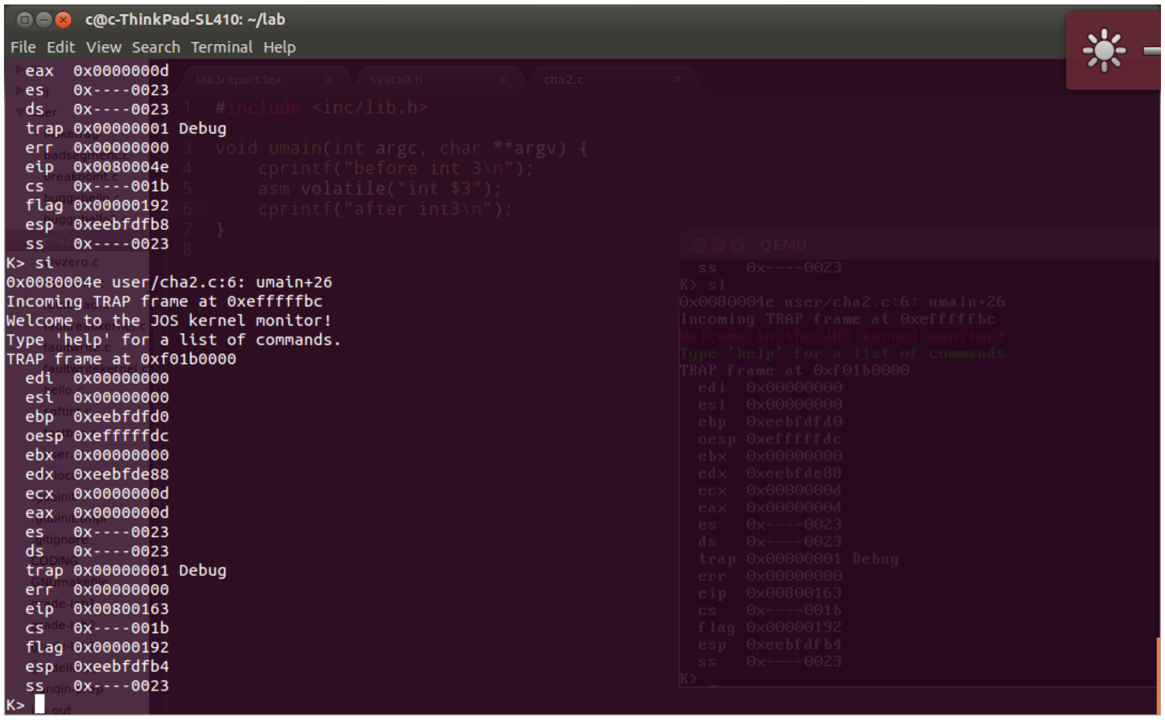
kern/Makefrag

```
KERN_BINFILES :=  user/hello \
      user/buggyhello \
      user/buggyhello2 \
      user/evilhello \
      user/testbss \
      user/divzero \
      user/breakpoint \
      user/softint \
      user/badsegment \
      user/faultread \
      user/faultreadkernel \
      user/faultwrite \
      user/faultwritekernel \
      user/cha2
```

运行结果如下：continue:



```
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
before int 3
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x0000000d
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800047
  cs   0x----001b
  flag 0x00000092
  esp  0xeebfdfb8
  ss   0x----0023
K> c
Incoming TRAP frame at 0xeffffbc
after int3
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.
K>
```



```
  eax  0x0000000d
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x0080004e
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb8
  ss   0x----0023
K> si
0x0080004e user/cha2.c:6: umain+26
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b0000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x0000000d
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800163
  cs   0x----001b
  flag 0x00000192
  esp  0xeebfdfb4
  ss   0x----0023
K>
```

4 Tips

1. 汇编里面的操作数长度一定要看清！
2. 参考的实验报告有很多结论是错的，一定要冷静分析。

3. Trapframe 本身就带有很多信息，调试的时候一定要多加利用！

5 References

北京大学操作系统实习 (实验班) 报告, 黄睿哲.