一、练习 1：

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.

题目要求：修改一下 mem_init()的代码，让它能够分配 envs 数组。这个数组是由 NENV 个 Env 结构体组成的。envs 数组所在的这部分内存空间也应该是用户模式只读的。被映射到虚拟地址 UENVS 处。

解答：只需要像在 Lab2 里面分配 pages 数组那样，分配一个 Env 数组给指针 envs 就可以了。首先要在 pmap.c 文件 page_init()之前为 envs 分配内存空间。代码编写如下图：

```
memset(pages, 0, npages * sizeof(struct PageInfo));

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

然后要在页表中设置它的映射关系，位于 check_page()函数之后，代码编写如下图：

```
// (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);
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
```

运行后调试结果如下，应该通过 check_page（）打印下列信息：

```
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!
```

二、练习二

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".

解答：首先 env_init()函数，env_init 函数就是遍历 envs 数组中的所有 Env 结构体，把每一个结构体的 env_id 字段置 0，因为要求所有的 Env 在 env_free_list 中的顺序要和它在 envs 中的顺序一致，所以需要采用头插法。

env.c/env_init()函数具体代码如下：

```
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();
}
```

env_setup_vm 函数功能主要是初始化新的用户环境的页目录表，不过只设置页目录表中和操作系统内核跟内核相关的页目录项，用户环境的页目录项不设置，因为所有用户环境的页目录表中和操作系统相关的页目录项都是一样的(除了虚拟地址 UVPT，这个也会单独进行设置)，所以我们可以参照 kern_pgdir 中的内

容来设置 env_pgdir 中的内容。具体代码如下所示：

```
        // LAB 3: Your code here.
        e->env_pgdir = (pde_t *)page2kva(p);
        p->pp_ref++;

        //Map the directory below UTOP.
        for(i = 0; i < PDX(UTOP); i++) {
                e->env_pgdir[i] = 0;
        }

        //Map the directory above UTOP
        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()为用户环境分配物理空间，这里注意我们要先把起始地址和终止地址进行页对齐，对其之后我们就可以以页为单位，为其一个页一个页的分配内存，并且修改页目录表和页表。代码如下图：

```
        //   (Watch out for corner-cases!)
        void* start = (void *)ROUNDDOWN((uint32_t)va, PGSIZE);
        void* end = (void *)ROUNDUP((uint32_t)va+len, PGSIZE);
        struct PageInfo *p = NULL;
        void* i;
        int r;
        for(i=start; i<end; i+=PGSIZE){
                p = page_alloc(0);
                if(p == NULL)
                        panic(" 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 功能是为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。每个用户程序都是 ELF 文件，所以我们要解析该 ELF 文件。具体代码实现如下：

```
        // LAB 3: Your code here.
        struct Elf* header = (struct Elf*)binary;
        if(header->e_magic != ELF_MAGIC) {
                panic("load_icode failed: The binary we load is not elf.\n");
        }
        if(header->e_entry == 0){
                panic("load_icode failed: The elf file can't be excuterd.\n");
        }

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

        lcr3(PADDR(e->env_pgdir));

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

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

        // Now map one page for the program's initial stack
        // at virtual address USTACKTOP - PGSIZE.
        region_alloc(e,(void *)(USTACKTOP-PGSIZE), PGSIZE);

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

env_create 是利用 env_alloc 函数和 load_icode 函数，加载一个 ELF 文件到用户环境中，具体代码如下：

```
//
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;
}
```

从 env_run 是真正开始运行一个用户环境，具体代码如下：

```
        //      e->env_tf to sensible values.

        // LAB 3: Your code here.
    if(curenv != NULL && curenv->env_status == ENV_RUNNING) {
        curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));
    env_pop_tf(&curenv->env_tf);

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

至此，已经完成了用户环境的大体配置，系统进入 hello 程序，在调用指令 int 之后触发中断，但尚未实现中断功能，所以在该指令处设置断点如果没报错说明我们的函数配置没有问题。在文件 obj/hello.asm 中的函数 sys_cputs 中，在如下图位置可以发现该指令的地址为 0x800aed，故在该地址设置断点并执行 continue 命令调试结果如下：

```
void
sys_cputs(const char *s, size_t len)
{
  800ad6:       55                      push   %ebp
  800ad7:       89 e5                   mov    %esp,%ebp
  800ad9:       57                      push   %edi
  800ada:       56                      push   %esi
  800adb:       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"
  800adc:       b8 00 00 00 00          mov    $0x0,%eax
  800ae1:       8b 4d 0c                mov    0xc(%ebp),%ecx
  800ae4:       8b 55 08                mov    0x8(%ebp),%edx
  800ae7:       89 c3                   mov    %eax,%ebx
  800ae9:       89 c7                   mov    %eax,%edi
  800aeb:       89 c6                   mov    %eax,%esi
  800aed:       cd 30                   int    $0x30
```

调试结果为：

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) b *0x800aed
Breakpoint 1 at 0x800aed
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x800aed:    int    $0x30

Breakpoint 1, 0x00800aed in ?? ()
(gdb)
```

输入 c 命令成功执行到 int 命令处，说明代码没有错误，继续后边实验。

四、练习四

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 `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.

该练习应实现下列功能：

1. trap_init() 先将所有中断处理函数的起始地址放到中断向量表 IDT 中。

2. 当中断发生时，不管是外部中断还是内部中断，处理器捕捉到该中断，进入核心态，根据中断向量去查询中断向量表，找到对应的表项

3. 保存被中断的程序的上下文到内核堆栈中，调用这个表项中指明的中断处理函数。

4. 执行中断处理函数。

5. 执行完成后，恢复被中断的进程的上下文，返回用户态，继续运行这个进程。

_alltraps 应该：

1. 把值压入堆栈使堆栈看起来像一个结构体 Trapframe

2. 加载 GD_KD 的值到 %ds, %es 寄存器中

3. 把%esp 的值压入，并且传递一个指向 Trapframe 的指针到 trap()函数中。

4. 调用 trap

解答：首先在 trapentry.S 文件，里面定义了两个宏定义，TRAPHANDLER，TRAPHANDLER_NOEC。他们的功能从汇编代码中可以看出：声明了一个全局符号 name，并且这个符号是函数类型的，代表它是一个中断处理函数名。其实这里就是两个宏定义的函数。这两个函数就是当系统检测到一个中断/异常时，需要首先完成的一部分操作，包括：中断异常码，中断错误码(error code)。正是因为有些中断有中断错误码，有些没有，所以利用两个宏定义函数。

在 trapentry.S 中，要根据这个中断是否有中断错误码，来选择调用 TRAPHANDLER，还是 TRAPHANDLER_NOEC，然后再统一调用_alltraps，其实目的就是为了能够让系统在正式运行中断处理程序之前完成必要的准备工作，比如保存现场等等。

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

具体代码如下：

```
.text

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

然后就会调用 _alltraps，_alltraps 函数其实就是为了能够让程序在之后调用 trap.c 中的 trap 函数时，能够正确的访问到输入的参数，即 Trapframe 指针类型的输入参数 tf。如下图：

```
/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
        pushl %ds
        pushl %es
        pushal

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

        push %esp
        call trap
```

而在 trap.c 文件中，应该继续完善 trap_init 函数，这个函数中将会对系统的 IDT 表进行初始化设置。在 trap.c 中实现 trap_init 函数，即在 idt 表中插入中断向量描述符，可以使用 SETGATE 宏实现：

SETGATE 宏的定义：

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

其中 gate 是 idt 表的 index 入口，istrap 判断是异常还是中断，sel 为代码段选择符，off 表示对应的处理函数地址，dpl 表示触发该异常或中断的用户权限。具体代码实现如下：

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

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

五、Question

Questions

Answer the following questions in your `answers-lab3.txt`:

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)?

Question1：
Answer：不同的中断或者异常当然需要不同的中断处理函数，因为不同的异常/中断可能需要不同的处理方式，比如有些异常是代表指令有错误，则不会返回被中断的命令。而有些中断可能只是为了处理外部 IO 事件，此时执行完中断函数还要返回到被中断的程序中继续运行。
Question2：
Answer：因为当前的系统正在运行在用户态下，特权级为 3，而 INT 指令为系统指令，特权级为 0。特权级为 3 的程序不能直接调用特权级为 0 的程序，会引发一个 General Protection Exception，即 trap 13。
六、Challenge 挑战部分：

Challenge! 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`.

该部分需要整理 kern/trapentry.S 和 kern/trap.c 中的代码，改变这种有坏代码的状态。

1、在 trapentry.S 中修改 TRAPHANDLER，利用.text 和.data 伪指令定义一块数据，这块数据保存了异常入口点以及它的属性。

2、在 trap.c 的 idt_init()函数中，利用循环来读取 trapentry.S 中定义的数据区，这样就减少了 idt_init()中的代码重复内容，这样可以有效防止程序不一致的情况。

具体修改如下：

1、在 TRAPHANDLER 修改

```
* Use ec = 1 for traps where the CPU automatically push an error code and ec =
0 for not.
 * Use user = 1 for a syscall; user = 0 for a normal trap.
 */
#define TRAPHANDLER(name, num, ec, user)                                \
        .globl name;            /* define global symbol for 'name' */   \
        .type name, @function;  /* symbol type is function */           \
        .align 2;               /* align function definition */         \
        name:                   /* function starts here */              \
        .if ec==0;                                                      \
                pushl $0;                                               \
        .endif;                                                         \
        pushl $(num);                                                   \
        jmp _alltraps                                                   \
.data;                                                                  \
        .long  name, num, user
```

2、修改 TRAPHANDLER 宏定义整合代码，在定义函数的同时(.text)定义相应的数据(.data)

然后定义一个全局数组，利用前面定义的宏实现该数组的填充。具体代码如下：

```
 */
data
        .globl  entry_data
        entry_data:
.text
TRAPHANDLER(divide_entry, T_DIVIDE, 0, 0);
TRAPHANDLER(debug_entry, T_DEBUG, 0, 0);
TRAPHANDLER(nmi_entry, T_NMI, 0, 0);
TRAPHANDLER(brkpt_entry, T_BRKPT, 0, 1);
TRAPHANDLER(oflow_entry, T_OFLOW, 0, 0);
TRAPHANDLER(bound_entry, T_BOUND, 0, 0);
TRAPHANDLER(illop_entry, T_ILLOP, 0, 0);
TRAPHANDLER(device_entry, T_DEVICE, 0, 0);
TRAPHANDLER(dblflt_entry, T_DBLFLT, 1, 0);
TRAPHANDLER(tts_entry, T_TSS, 1, 0);
TRAPHANDLER(segnp_entry, T_SEGNP, 1, 0);
TRAPHANDLER(stack_entry, T_STACK, 1, 0);
TRAPHANDLER(gpflt_entry, T_GPFLT, 1, 0);
TRAPHANDLER(pgflt_entry, T_PGFLT, 1, 0);
TRAPHANDLER(fperr_entry, T_FPERR, 0, 0);
TRAPHANDLER(align_entry, T_ALIGN, 1, 0);
TRAPHANDLER(mchk_entry, T_MCHK, 0, 0);
TRAPHANDLER(simderr_entry, T_SIMDERR, 0, 0);
TRAPHANDLER(syscall_entry, T_SYSCALL, 0, 1);
.data
        .long 0, 0, 0   // interupt end identify
/*
```

最后在 trap_init 函数中就可以使用该全局数组对 idt 进行初始化。如下所示：

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

        // LAB 3: Your code here.
        extern long entry_data[][3];
        int i;
        for (i = 0; entry_data[i][0] != 0; i++ )
                SETGATE(idt[entry_data[i][1]], 0, GD_KT, entry_data[i][0],
entry_data[i][2]*3);
        // Per-CPU setup
        trap_init_percpu();
}
```

//SETGATE 的第 2 个参数标示是异常，若是中断的话需要禁用中断，如果在陷入内核时，没禁用中断，会 panic 的。

七、练习五：

> **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`.

解答：本次练习要实现如下功能：修改 trap_dispatch 函数，使系统能够把缺页异常引导到 page_fault_handler() 上执行。在修改完成后，运行 make grade，出现的结果应该是你修改后的 JOS 可以成功运行 faultread，faultreadkernel，faultwrite，faultwritekernel 测试程序。

1、首先根据 trapentry.S 文件中的 TRAPHANDLER 函数可知，这个函数会把当前中断的中断码压入堆栈中，再根据 inc/trap.h 文件中的 Trapframe 结构体可以知道，Trapframe 中的 tf_trapno 成员代表这个中断的中断码。所以在 trap_dispatch 函数中我们需要根据输入的 Trapframe 指针 tf 中的 tf_trapno 成员来判断到来的中断是什么中断，这里判断是否是缺页中断，如果是则执行 page_fault_handler 函数。修改函数如下：

```
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:
            // 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;
            }
        }

        // 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;
        }
}
```

八、练习六

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.

解答：题目要求修改 trap_dispatch()使断点异常发生时，能够触发 kernel monitor。修改完成后运行 make grade，运行结果应该是你修改后的 JOS 能够正确运行 breakpoint 测试程序。

和前边的练习相似，这里处理断点中断 (T_BRKPT)，kernel monitor 就是定义在 kern/monitor.c 文件中的 monitor 函数，对 trap_dispatch 函数修改如下：

```
        // LAB 3: Your code here.
        switch(tf->tf_trapno) {
            case (T_PGFLT):
                page_fault_handler(tf);
                break;
            break;
            case (T_BRKPT):
                monitor(tf);
                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;
                }
        }

        // 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;
        }
}
```

问题 Questions

Questions

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

解答 Question3：

通过实验发现出现这个现象的问题就是在设置 IDT 表中的 breakpoint exception 的表项时，如果把表项中的 DPL 字段设置为 3，则会触发 break point

exception，如果设置为 0，则会触发 general protection exception。DPL 字段代表的含义是段描述符优先级（Descriptor Privileged Level），如果想要当前执行的程序能够跳转到这个描述符所指向的程序哪里继续执行的话，有个要求，就是要求当前运行程序的 CPL，RPL 的最大值需要小于等于 DPL，否则就会出现优先级低的代码试图去访问优先级高的代码的情况，就会触发 general protection exception。那么测试程序首先运行于用户态，它的 CPL 为 3，当异常发生时，它希望去执行 int 3 指令，这是一个系统级别的指令，用户态命令的 CPL 一定大于 int 3 的 DPL，所以就会触发 general protection exception，但是如果把 IDT 这个表项的 DPL 设置为 3 时，就不会出现这样的现象了，这时如果再出现异常，肯定是因为还没有编写处理 break point exception 的程序所引起的，所以是 break point exception。

七、练习七

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 `trap_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. Handle all the systems calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call.

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.

解答：题目要求给中断向量 T_SYSCALL 编写一个中断处理函数。编辑 kern/trapentry.S 和 kern/trap.c 中的 trap_init()函数，修改 trap_dispatch()函数，使其能够通过调用 syscall()（在 kern/syscall.c 中定义的）函数处理系统调用中断。最终实现 kern/syscall.c 中的 syscall()函数，确保这个函数会在系统调用号为非法值时返回-E_INVAL。理解 lib/syscall.c 文件。处理在 inc/syscall.h 文件中定义的所有系统调用。最后通过 make run-hello 指令来运行 user/hello 程序，它应该在控制台上输出 "hello, world" 然后出发一个页中断。如果没有发生的话，代表编写的系统调用处理函数是不正确的。

系统调用的整个流程如下;如果现在运行的是内核态的程序的话，此时调用了一个系统调用，比如 sys_cputs 函数时，此时不会触发中断，那么系统会直接执行定义在 lib/syscall.c 文件中的 sys_cputs，这个文件中定义了几个比较常用的系统调用，包括 sys_cputs，sys_cgetc 等等。他们都是统一调用一个 syscall 函数，通过这个函数的代码发现其实它是执行了一个汇编指令。所以最终是这个函数完成了系统调用。

以上是运行在内核态下的程序，调用系统调用时的流程。

如果是用户态程序，这个练习就是编写程序使用户程序在调用系统调用时，

最终也能经过一系列的处理最终去执行 lib/syscall.c 中的 syscall 指令。

这个具体过程就是，当用户程序中要调用系统调用时，比如 sys_cputs，从它的汇编代码中可以发现它会执行一个 int $0x30 指令，这个指令就是软件中断指令，这个中断的中断号就是 0x30，即 T_SYSCALL，所以题目中让我们首先为这个中断号编写一个中断处理函数，我们首先就要在 kern/trapentry.S 文件中为它声明它的中断处理函数，即 TRAPHANDLER_NOEC。在 trapentry.s 文件中代码中断处理函数如下：

然后在 trap.c 文件中声明 t_syscall() 函数。并且在 trap_init() 函数中为它注册，如下所示：

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

        // LAB 3: Your code here.
        extern long entry_data[][3];
        int i;
        for (i = 0; entry_data[i][0] != 0; i++ )
                SETGATE(idt[entry_data[i][1]], 0, GD_KT, entry_data[i][0],
entry_data[i][2]*3);

        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();
}
```

此时当系统调用中断发生时，系统就可以捕捉到这个中断了，中断发生时，系统会调用 _alltraps 代码块，并且最终来到 trap() 函数处，进入 trap 函数后，经过一系列处理进入 trap_dispatch 函数。题目中要求此时我们需要去调用 kern/syscall.c 中的 syscall 函数，这里注意，这个函数可不是 lib/syscall.c 中的 syscall 函数，但是通过阅读 kern/syscall.c 中的 syscall 程序我们发现，它的输入和 lib/syscall.c 中的 syscall 很像，如下

kern/syscall.c 中的 syscall ：

```
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
uint32_t a5)
```

lib/syscall.c 中的 syscall ：

```
syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
uint32_t a5)
```

可以发现 kern/syscall.c 中的所有函数和 lib/syscall.c 中的所有函数都是一样的。比如 在这两个文件中都有 sys_cputs 函数，但是其实实现方式却不一样。拿 sys_cputs 函数举例

在 kern/syscall.c 中的 sys_cputs 是这样的：

```
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);
}
```

而在 lib/syscall.c 中的 sys_cputs 是这样的

```
void
sys_cputs(const char *s, size_t len)
{
        syscall(SYS_cputs, 0, (uint32_t)s, len, 0, 0, 0);
}
```

可见在 lib/syscall.c 中，是直接调用 syscall 的，但是 kern/syscall.c 中的 sys_cputs，它调用了 cprintf，这个调用其实就是为了完成输出的功能，但是当我们程序运行到这里时，系统已经工作在内核态了，而 cprintf 函数其实就是通过调用 lib/syscall.c 中的 sys_cputs 来实现的，由于此时系统已经处于内核态了，所以这个 sys_cputs 可以被执行了。所以 kern/syscall.c 中的 sys_cputs 函数通过调用 cprintf 实现了调用 lib/syscall.c 中的 syscall 。

所以接下来要在 kern/syscall.c 中的 syscall() 函数中正确的调用 sys_cputs 函数，当然 kern/syscall.c 中其他的函数也能完成这个功能。所以必须根据触发这个系统调用的指令到底想调用哪个系统调用来确定该调用哪个函数。

根据 syscall 函数中的第一个参数，syscallno，这个值需要手动传递进去的，通过阅读 lib/syscall.c 中的 syscall 函数可以知道它存放在 eax 寄存器中。所以最后完成 trap_dispatch 和 kern/syscall.c 中的 syscall 函数的代码。Trap_dispatch 如下：

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

kern/syscall.c 中的 syscall()

```
// 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((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;
        }
}
```

调试，输入 make run_hello 后结果如下：

```
    ss   0x----0023
hello, world
Incoming TRAP frame at 0xefffffbc
TRAP frame at 0xf01a1000
  edi  0x00000000
  esi  0x00000000
```

八、练习八

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 00001000`". `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.

解答：本次练习就是要通过程序获得当前正在运行的用户环境的 env_id ， 以及这个用户环境所对应的 Env 结构体的指针。

（1）、env_id 可以通过调用 sys_getenvid() 这个函数来获得。

（2）、获得它对应的 Env 结构体指针：

通过阅读 lib/env.h 文件可以知道 env_id 的值包含三部分，第 31 位被固定为 0；第 10~30 这 21 位是标识符，标示这个用户环境；第 0~9 位代表这个用户环境所采用的 Env 结构体，在 envs 数组中的索引。所以只需知道 env_id 的低 0~9 位就可以获得这个用户环境对应的 Env 结构体了。

所以在 lib/libmain.c 中的 libmain()函数添加获取 env_id 代码，具体代码如下：

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

运行结果如下：

```
  ss   0x----0023
i am environment 00001000
Incoming TRAP frame at 0xeffffffc
TRAP frame at 0xf01a1000
  edi  0x00000000
```

九、练习九：

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.

解答：1、在本次练习中主要需要完善 kern/trap.c 文件，根据 CS 段寄存器叫做 CPL 位的低 2 位寄存器，表示当前运行的代码的访问权限级别，0 代表内核态，3 代表用户态，题目中要求检测到 page fault 是出现在内核态时，并把这个事件 panic 出来，所以需要修改 page_fault_handler 文件，具体修改如下所示：

```
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 && 0x01 == 0) {
                panic("page_fault in kernel mode, fault address %d\n",
fault_va);
        }

        // 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);
}
```

2、继续完善 kern/pmap.c 文件中的 user_mem_assert ，user_mem_check 函数，通过观察 user_mem_assert 函数，它调用了 user_mem_check 函数。而 user_mem_check 函数的功能是检查一下当前用户态程序是否有对虚拟地址空间[va, va+len] 的 perm| PTE_P 访问权限。所以应该先找到这个虚拟地址范围

对应于当前用户态程序的页表中的页表项，然后再去看一下这个页表项中有关访问权限的字段，是否包含 perm | PTE_P，只要有一个页表项是不包含的，就代表程序对这个范围的虚拟地址没有 perm|PTE_P 的访问权限。
具体代码实现如下：

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

3、接下来补全 kern/syscall.c 文件中 sys_cputs 函数，用来检查用户程序对虚拟地指空间 [s, s+len] 是否有访问权限，可以使用刚刚写好的函数 user_mem_assert() 来实现。

```
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);
}
```

4、在 kern/kdebug 中修改 debuginfo_eip 函数，对用户空间的数据使用 user_mem_check()函数检查当前用户空间是否对其有 PTE_U 权限。代码如下：

```
USTABDATA;

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

已经完成所有的函数。

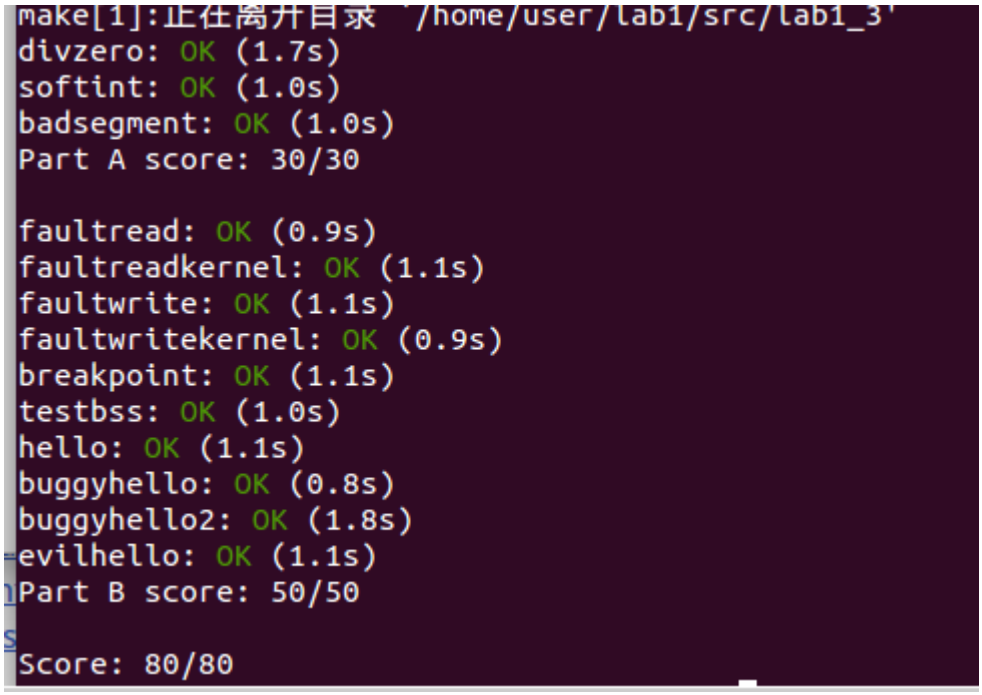
调试结果如下：

```
[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.
K>
```

在输入 backtace 时输出下列内容：

```
K> backtrace
Stack_backtrace:
 ebp:0xeffffea0 eip:0xf0100961 args:0x00000001 0xefffeb8 0x00000000 0xf01a1000
0xf017eaa0
             kern/monitor.c:147 monitor+275
 ebp:0xefffff10 eip:0xf01036f1 args:0x00000000 0xefffff3c 0xf01064f3 0xf01a1000
0xf01a1000
             kern/env.c:490 env_destroy+41
 ebp:0xefffff30 eip:0xf0102fb4 args:0xf01a1000 0x00001000 0x00000001 0x00000004
0xf01064f3
             kern/pmap.c:650 user_mem_assert+82
 ebp:0xefffff50 eip:0xf01040c8 args:0xf01a1000 0x00000001 0x00000001 0x00000004
0xf01064f3
             kern/syscall.c:26 syscall+72
 ebp:0xefffff80 eip:0xf0103f47 args:0x00000000 0x00000001 0x00000001 0x00000000
0x00000000
             kern/trap.c:190 trap+246
 ebp:0xefffffb0 eip:0xf0104076 args:0xefffffbc 0x00000000 0x00000000 0xeebfdfb0
0xefffffdc
             kern/trapentry.S:86 <unknown>+0
Incoming TRAP frame at 0xefffffdc
kernel panic at kern/trap.c:261: page_fault in kern
```

完美正如我们所料一样，至此终于完成了 lab3 的所有内容，最后 make grade ，所有工作完美结束。


make[1]:正在离开目录 '/home/user/lab1/src/lab1_3'
divzero: OK (1.7s)
softint: OK (1.0s)
badsegment: OK (1.0s)
Part A score: 30/30
faultread: OK (0.9s)
faultreadkernel: OK (1.1s)
faultwrite: OK (1.1s)
faultwritekernel: OK (0.9s)
breakpoint: OK (1.1s)
testbss: OK (1.0s)
hello: OK (1.1s)
buggyhello: OK (0.8s)
buggyhello2: OK (1.8s)
evilhello: OK (1.1s)
Part B score: 50/50
Score: 80/80