# 实 验 报 告

**Lab4**

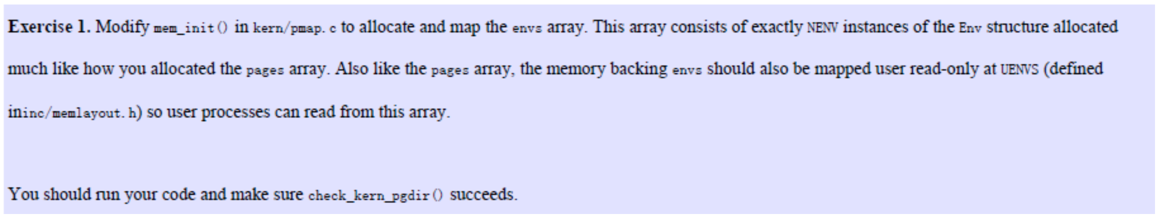
姓名：朱瑞媛

学号：**14307130373**

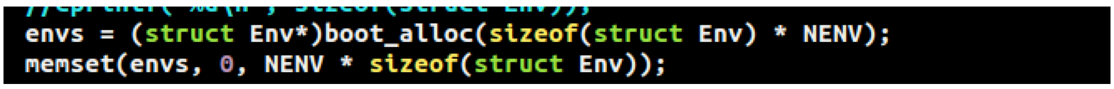
# 实验报告填写要求

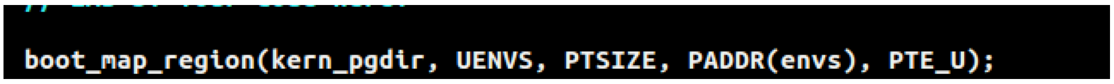
1.请在每个 exercise 之后简要叙述实验原理，详细描述实验过程。
2.请将你认为的关键步骤附上必要的截图。
3.有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。
4.你还可以列举包括但不局限于以下方面：实验过程中碰到的问题、你是如何解决的、实验之后你还留有哪些疑问和感想。
5.如果实验附有 question，请在每个 question 之后作答，这是实验报告评分的重要部分。
7.Challenge 为加分选作题。每个 lab 可能有多个 challenge，我们会根据完成情况以及难度适当加分。这部分的实验过程描述应该比 exercise 更加详细。

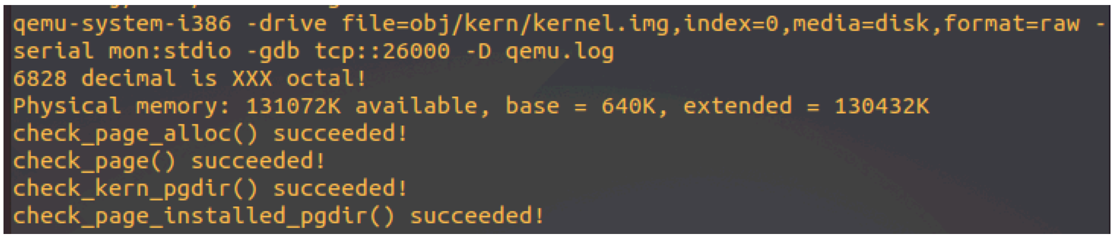
## Part A: User Environments and Exception Handling

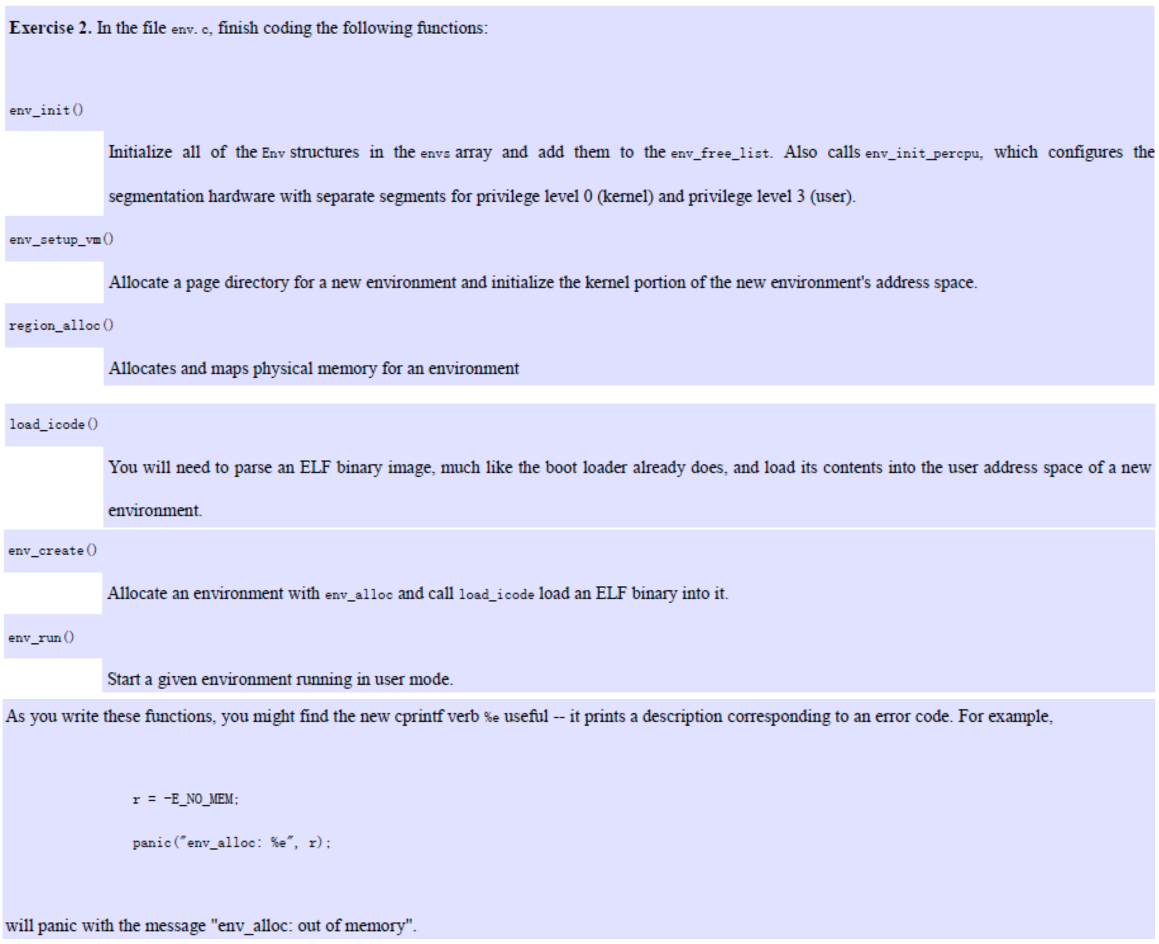

**Exercise 1.** Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user 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.


```
envs = (struct Env*)boot_alloc(sizeof(struct Env) * NENV);
memset(envs, 0, NENV * sizeof(struct Env));
```
调用 boot_alloc 函数为 envs 数组分配空间，并将其置 0.


```
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```
调用 boot_map_region 函数建立虚拟地址到物理地址的映射。


```
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is XXX octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
```
测试通过 check_kern_pgdir()函数。
//由于这个报告写于全部内容完成后，所以这里截了通过的函数示意，没有完整的 panic 信息==以下很多测试截图同理 QAQ



**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.

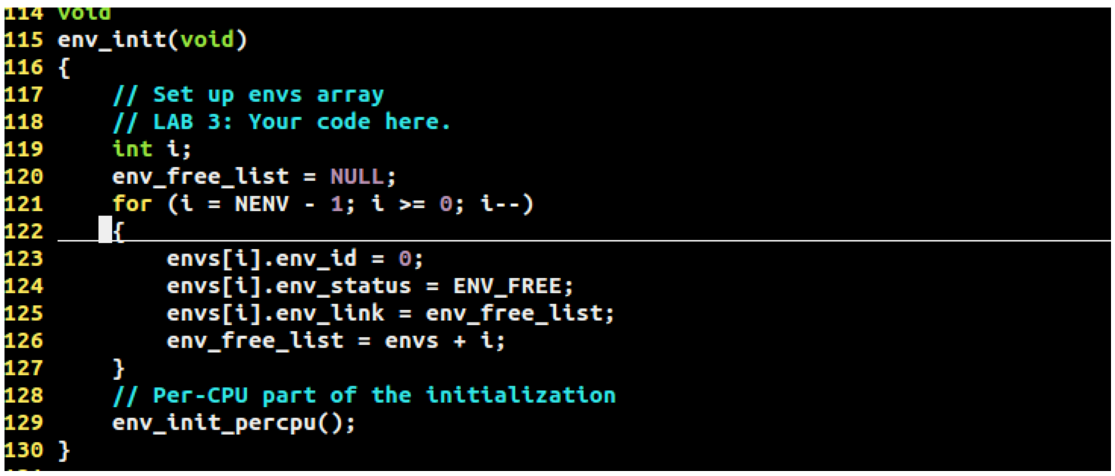
`env_run()`

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,

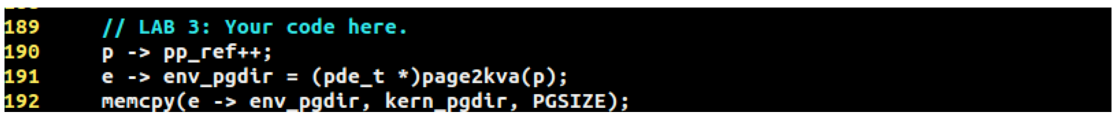
```
r = -E_NO_MEM;
panic("env_alloc: %e", r);
```

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



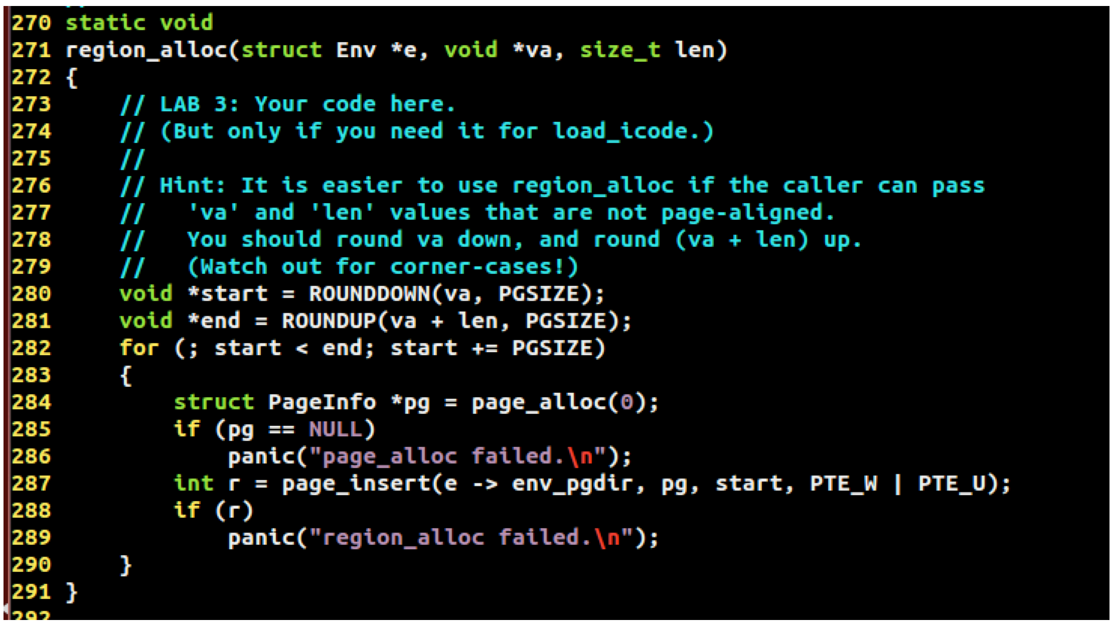
```
void
env_init(void)
{
	// Set up envs array
	// LAB 3: Your code here.
	int i;
	env_free_list = NULL;
	for (i = NENV - 1; i >= 0; i--)
	{
		envs[i].env_id = 0;
		envs[i].env_status = ENV_FREE;
		envs[i].env_link = env_free_list;
		env_free_list = envs + i;
	}
	// Per-CPU part of the initialization
	env_init_percpu();
}
```

env_init()函数。
初始化envs,与env_free_list反向。



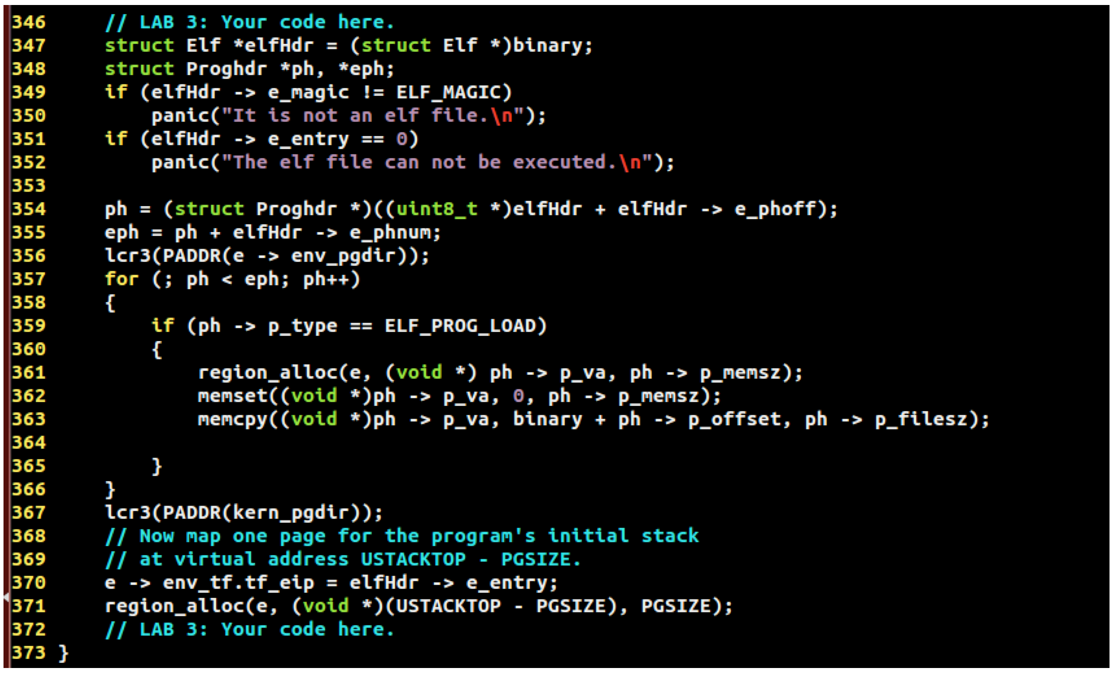
```
	// LAB 3: Your code here.
	p -> pp_ref++;
	e -> env_pgdir = (pde_t *)page2kva(p);
	memcpy(e -> env_pgdir, kern_pgdir, PGSIZE);
```

env_set_vm()函数。
分配 e 对应的页表空间，设置页 p 的信息。利用 kern 为模版初始化虚拟地址。



```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.)
    //
    // Hint: It is easier to use region_alloc if the caller can pass
    //   'va' and 'len' values that are not page-aligned.
    //   You should round va down, and round (va + len) up.
    //   (Watch out for corner-cases!)
    void *start = ROUNDDOWN(va, PGSIZE);
    void *end = ROUNDUP(va + len, PGSIZE);
    for (; start < end; start += PGSIZE)
    {
        struct PageInfo *pg = page_alloc(0);
        if (pg == NULL)
            panic("page_alloc failed.\n");
        int r = page_insert(e -> env_pgdir, pg, start, PTE_W | PTE_U);
        if (r)
            panic("region_alloc failed.\n");
    }
}
```

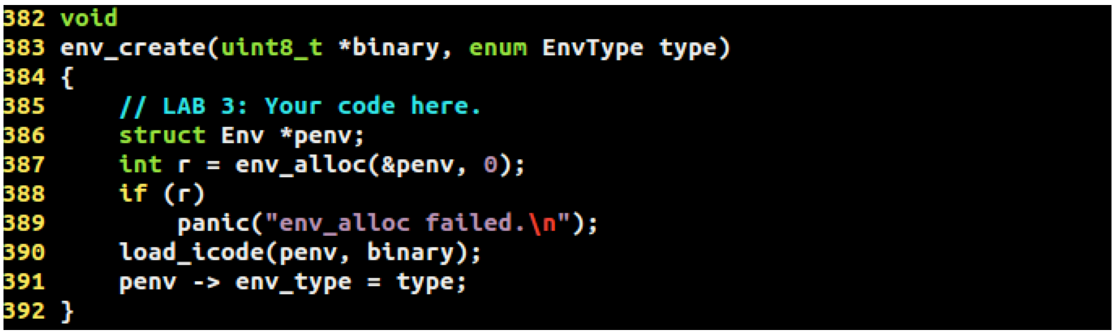
region_alloc()函数。
将 va 及 va + len 按页对齐。遍历这些地址并为它们分配页，调用 page_insert()建立虚拟地址到页的映射。



```c
    // LAB 3: Your code here.
    struct Elf *elfHdr = (struct Elf *)binary;
    struct Proghdr *ph, *eph;
    if (elfHdr -> e_magic != ELF_MAGIC)
        panic("It is not an elf file.\n");
    if (elfHdr -> e_entry == 0)
        panic("The elf file can not be executed.\n");

    ph = (struct Proghdr *)((uint8_t *)elfHdr + elfHdr -> e_phoff);
    eph = ph + elfHdr -> e_phnum;
    lcr3(PADDR(e -> env_pgdir));
    for (; ph < eph; ph++)
    {
        if (ph -> p_type == ELF_PROG_LOAD)
        {
            region_alloc(e, (void *) ph -> p_va, ph -> p_memsz);
            memset((void *)ph -> p_va, 0, ph -> p_memsz);
            memcpy((void *)ph -> p_va, binary + ph -> p_offset, ph -> p_filesz);

        }
    }
    lcr3(PADDR(kern_pgdir));
    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.
    e -> env_tf.tf_eip = elfHdr -> e_entry;
    region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
    // LAB 3: Your code here.
}
```

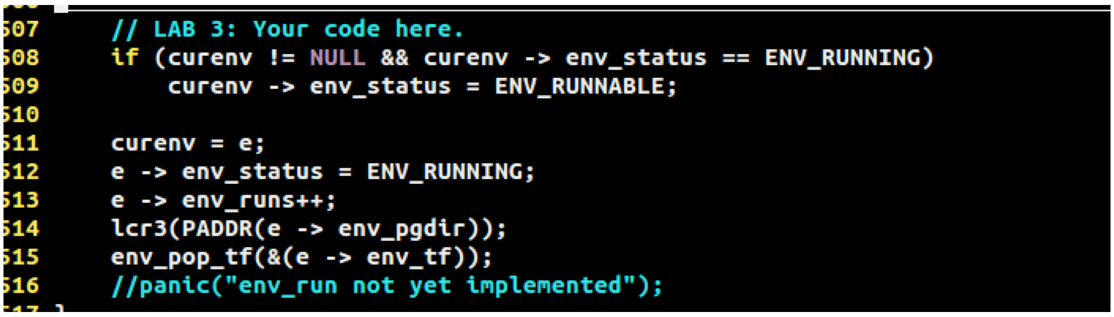
load_icode()函数。
将一个可执行 elf 文件（内存地址）载入一个 env 中，为每一个用户进程设置它的初始代码区，堆栈以及处理器标识位。
首先判断*elfHdr 是否是一个可执行的 elf 文件。
计算进程的起始地址。
加载 cr3 寄存器，存放当前页对应的目录表。
遍历进程，当该文件需要被加载时，分配空间并建立映射。
将 ph -> memsz 个虚拟地址全部置 0，再将 binary + ph -> p_offset 开始内容复制至前 ph -> p_filesz.

重新加载 cr3 寄存器，恢复至初始 kern_pgdir。
设置入口，为进程分配页并建立映射。



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

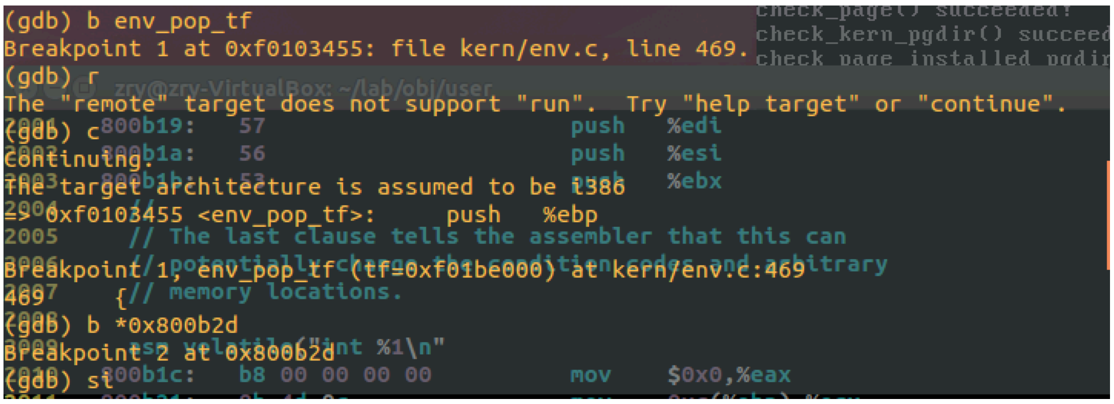
evn_create()函数。
加载 elf 文件至一个新分配的环境中。调用 env_alloc()分配环境，判定是否成功分配。调用 load_icode()加载，将参数 type 设置为环境的 type。



```c
    // LAB 3: Your code here.
    if (curenv != NULL && 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));
    //panic("env_run not yet implemented");
```

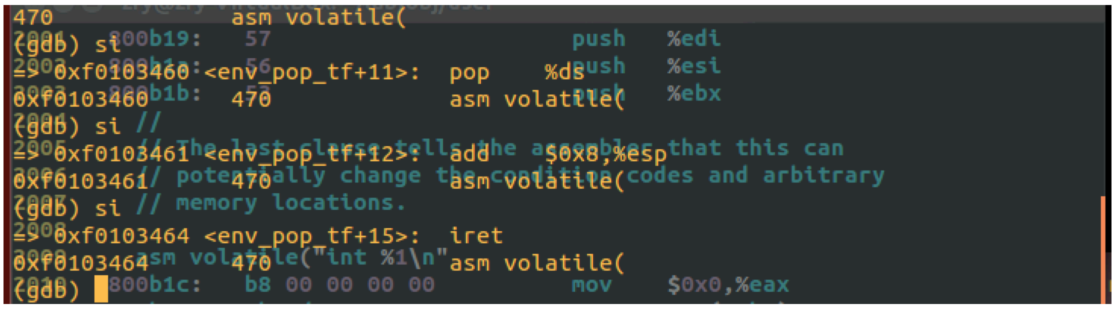
env_run()函数。
运行一个环境时直接调用的函数。
判断当前环境是否为空或者状态是否为 ENV_RUNNING，如果是状态置为 RUNNABLE.
当前环境设置为需要运行的环境 e。设置状态，增加环境被运行的次数。
重新加载 cr3 寄存器的值，进入用户模式。

此 execise 完成后，使用 gdb 进行测试。由于之后的一些部分没有完成，直接运行会出现 triple fault.



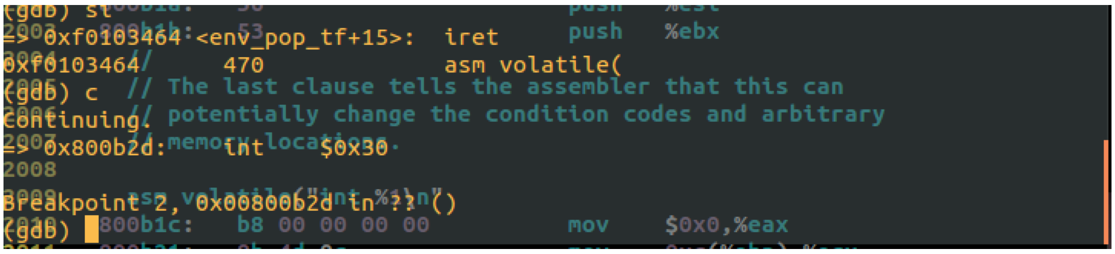
```
(gdb) b env_pop_tf
Breakpoint 1 at 0xf0103455: file kern/env.c, line 469.
(gdb) r
The "remote" target does not support "run".  Try "help target" or "continue".
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0103455 <env_pop_tf>:     push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01be000) at kern/env.c:469
469     {
(gdb) b *0x800b2d
Breakpoint 2 at 0x800b2d
(gdb) si
```

在最后一个函数 evn_pop_tf()函数处及 int 指令处设置断点。
运行至第一个断点后，单步执行。



```
470             asm volatile(
(gdb) si
=> 0xf0103460 <env_pop_tf+11>:  pop    %ds
0xf0103460      470             asm volatile(
(gdb) si
=> 0xf0103461 <env_pop_tf+12>:  add    $0x8,%esp
0xf0103461      470             asm volatile(
(gdb) si
=> 0xf0103464 <env_pop_tf+15>:  iret
0xf0103464      470             asm volatile(
(gdb) 
```

到达 iret 指令时，进入用户模式。



```
(gdb) si
=> 0xf0103464 <env_pop_tf+15>:  iret
0xf0103464      470             asm volatile(
(gdb) c
Continuing.
=> 0x800b2d:    int    $0x30

Breakpoint 2, 0x00800b2d in ?? ()
(gdb) 
```

继续运行后，会到达 sys_cputs()函数中的 int $0x30 指令。表示测试成功。

**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.

完成阅读。
了解了关于中断控制的知识。

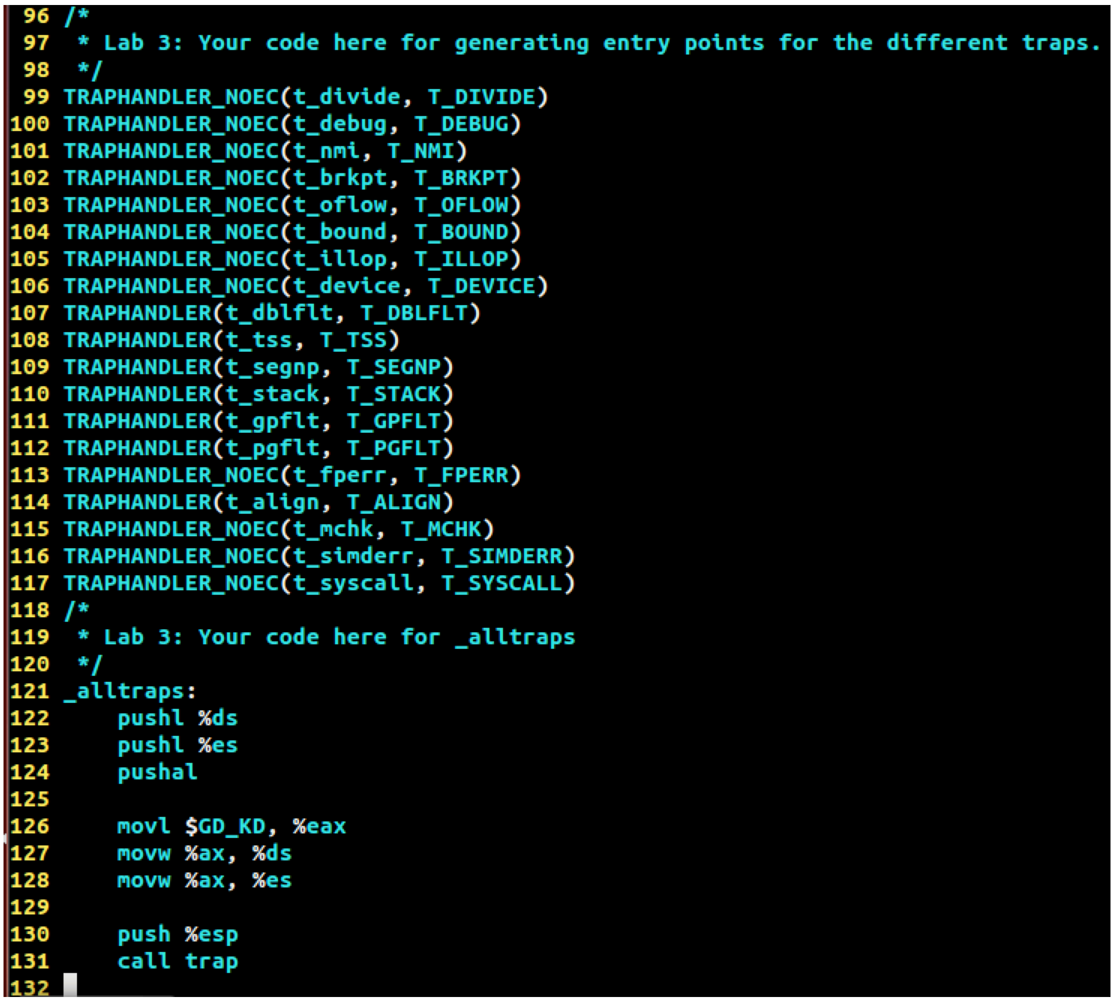
**Exercise 4.** Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S`should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapentry.S` (using those macros) for each trap 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.



```
/*
 * 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
```

编辑 trapentry.S 文件。
.text 中先定义一些类型的中断入口地址。利用之前定义的两个宏。
_alltraps 定义所有中断共同执行的内容。功能是为了能够让程序在之后调用 trap.c 中 trap 函数时，能够正确的访问到输入的参数。
压栈顺序为 ds， es, 其它。

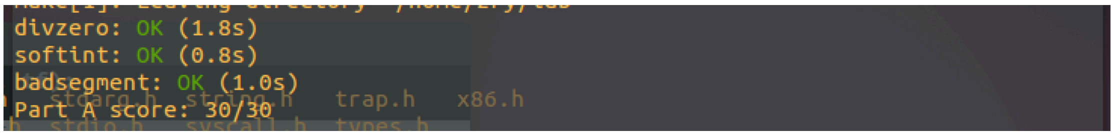
**Questions**

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)?

1. 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。

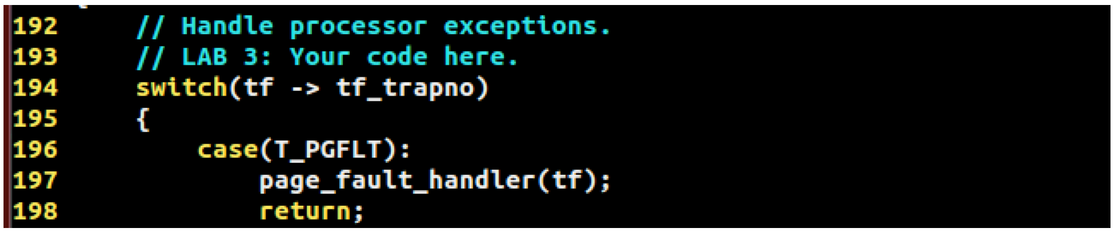
2. 因为当前的系统正在运行在用户态下，特权级为 3，而 INT 指令为系统指令，特权级为 0。特权级为 3 的程序不能直接调用特权级为 0 的程序，会引发一个 General Protection Exception，即 trap13。



```
divzero: OK (1.8s)
softint: OK (0.8s)
badsegment: OK (1.0s)
Part A score: 30/30
```

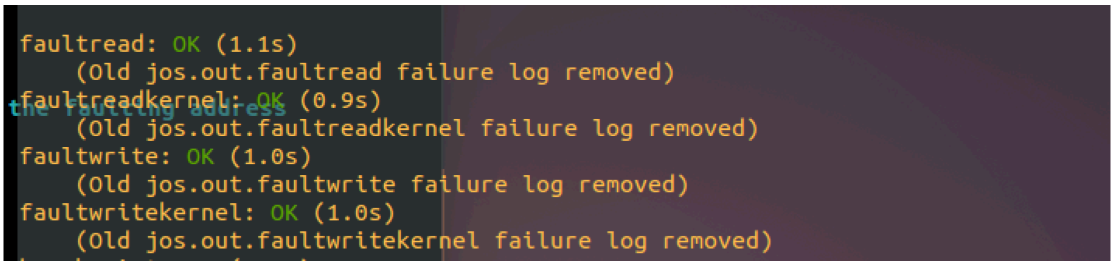
至此，Part A 完成。Score 30/30.

**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`.



```
    // Handle processor exceptions.
    // LAB 3: Your code here.
    switch(tf -> tf_trapno)
    {
        case(T_PGFLT):
            page_fault_handler(tf);
            return;
```

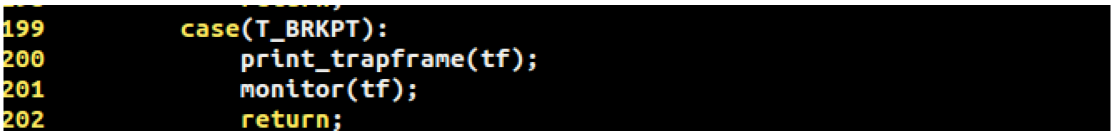
编辑 trap_dispatch()函数，添加以上内容。
如果中断的类型是缺页，则调用 page_fault_handler()函数。



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

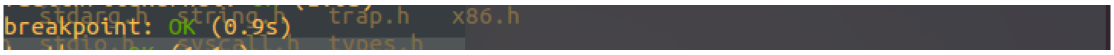
通过以上 4 个测试程序。

**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.



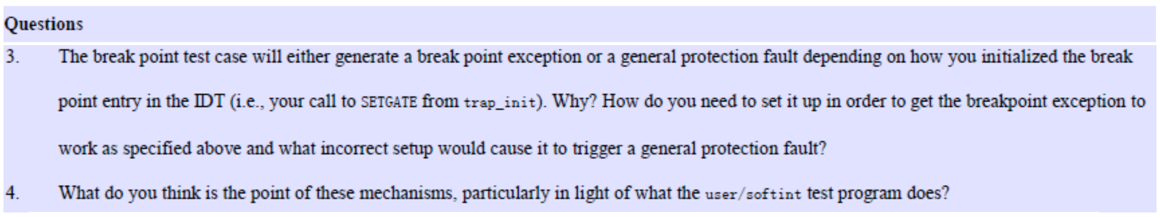
```
        case(T_BRKPT):
            print_trapframe(tf);
            monitor(tf);
            return;
```

编辑 trap_dispatch()函数，添加以上内容。
如果中断类型是断点中断，调用 kern/monitor.c 中的 monitor()函数。



```
breakpoint: OK (0.9s)
```

通过 breakpoint 测试程序。

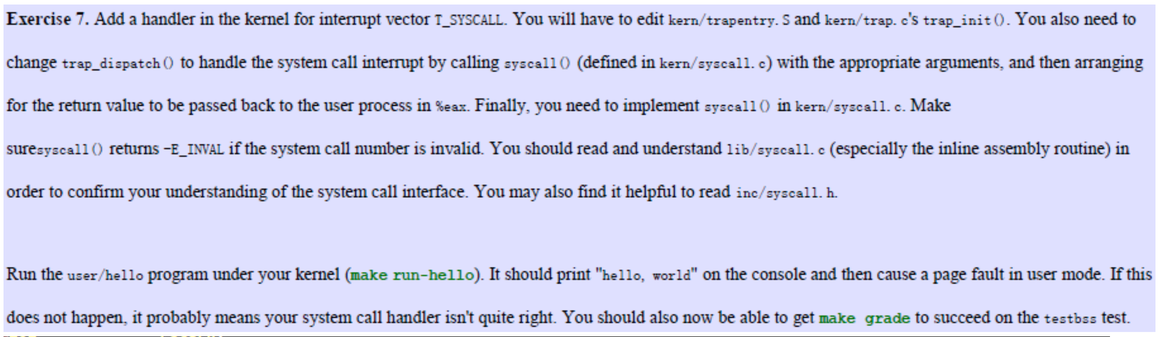


**Questions**

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?

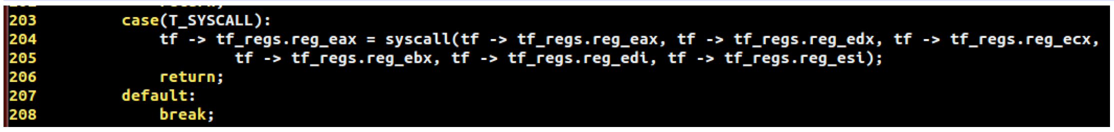
3.设置 IDT 表中的 breakpoint exception 的表项时，如果我们把表项中的 DPL 字段设置为 3，则会触发 break point exception，如果设置为 0，则会触发 general protection exception。DPL 字段代表的含义是段描述符优先级（Descriptor Privileged Level），如果我们想要当前执行的程序能够跳转到这个描述符所指向的程序哪里继续执行的话，有个要求，就是要求当前运行程序的 CPL，RPL 的最大值需要小于等于 DPL，否则就会出现优先级低的代码试图去访问优先级高的代码的情况，就会触发 general protection exception。那么我们的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，肯定是因为我们还没有编写处理 break point exception 的程序所引起的，所以是 break point exception。

4. 此权限机制是有必要的，有效地防止了一些程序恶意任意调用指令，引发一些危险的错误。
Softint 不允许用户的直接操作，保护内核。
Breakpoint 的权限机制给操作人员提供了便利。



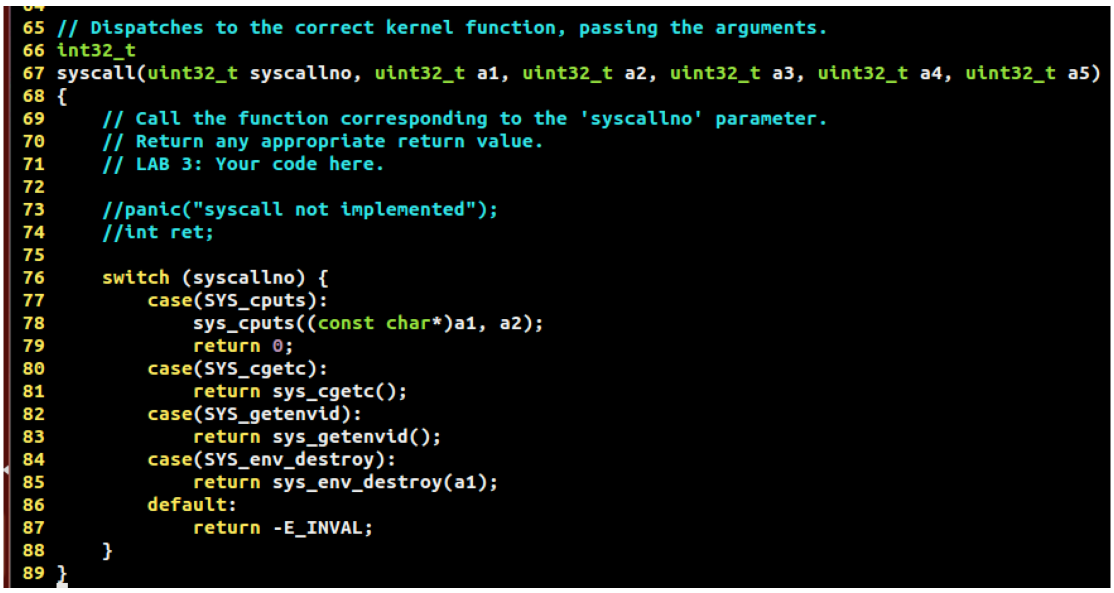
**Exercise 7.** Add a handler in the kernel for interrupt vector T_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap_init(). You also need to change trap_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, 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 suresyscall() 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.



```
        case(T_SYSCALL):
            tf -> tf_regs.reg_eax = syscall(tf -> tf_regs.reg_eax, tf -> tf_regs.reg_edx, tf -> tf_regs.reg_ecx,
                    tf -> tf_regs.reg_ebx, tf -> tf_regs.reg_edi, tf -> tf_regs.reg_esi);
            return;
        default:
            break;
```

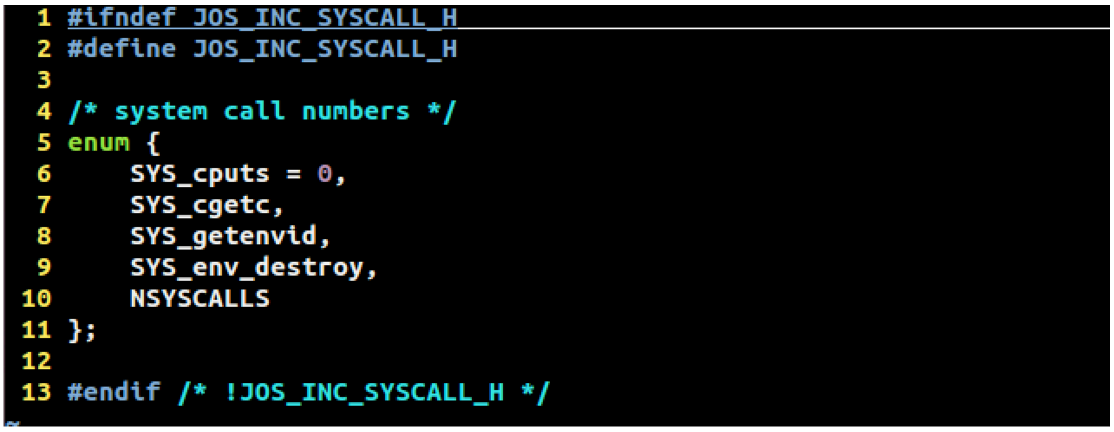
编辑trap_dispatch()函数。
调用syscall()函数。eax中存放的为call number.其余的参数依次传入。将结果存放至reg_eax.



```c
// 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");
    //int ret;

    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;
    }
}
```

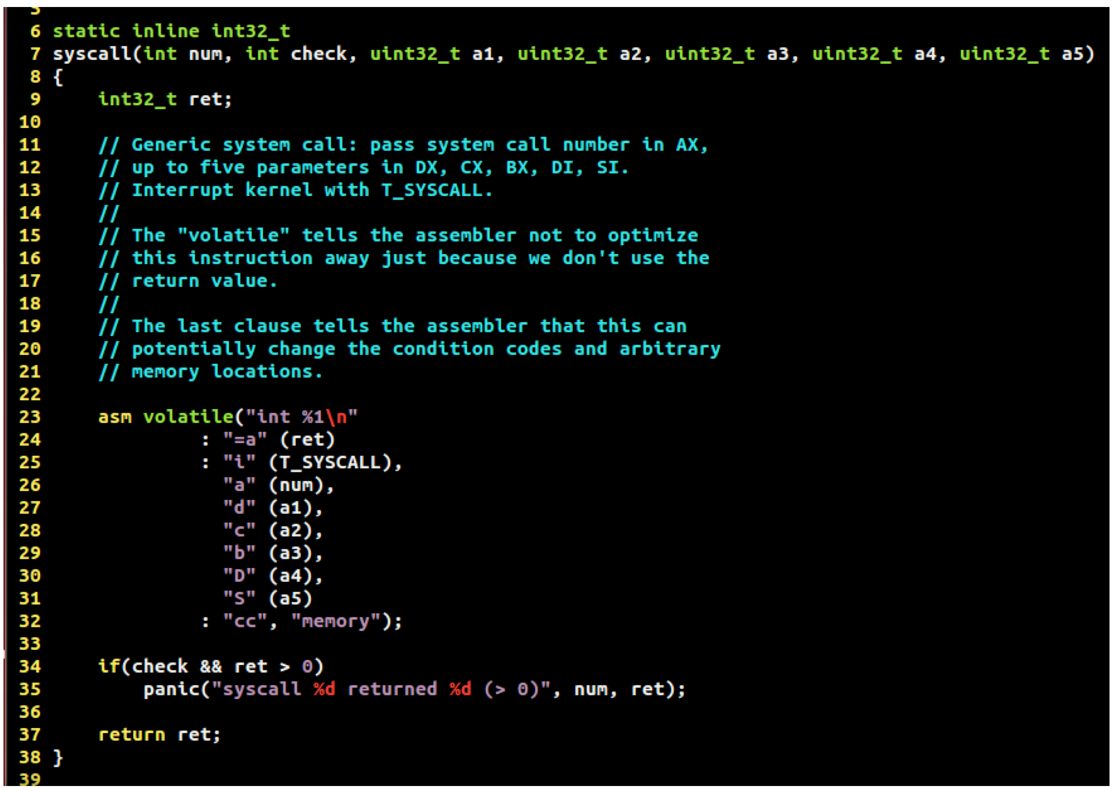
syscall()函数。

根据 call number 选择调用不同的函数。如果全部不符合返回-E_INVAL.



```c
#ifndef JOS_INC_SYSCALL_H
#define JOS_INC_SYSCALL_H

/* system call numbers */
enum {
    SYS_cputs = 0,
    SYS_cgetc,
    SYS_getenvid,
    SYS_env_destroy,
    NSYSCALLS
};

#endif /* !JOS_INC_SYSCALL_H */
```

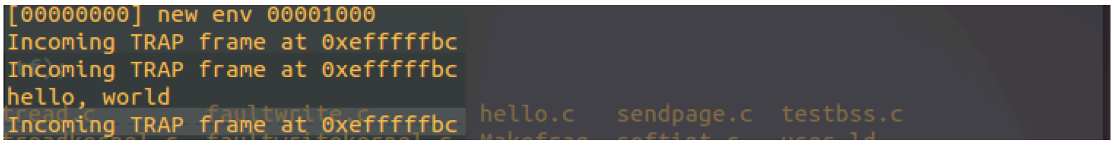
call number 的定义如上。



```c
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;
}
```

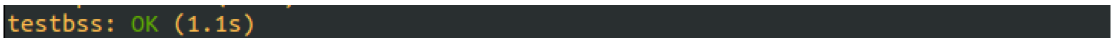
lib/syscall.c 中 syscall 函数的定义。

kern/syscall.c 中 syscall.c 中的函数可以看作是 lib/syscall.c 中函数的的一个外壳,kern/syscall 中的函数(例如 sys_cputs())通过调用一些函数(例如 cprintf()) 来实现调用 lib/syscall.c 中的函数。



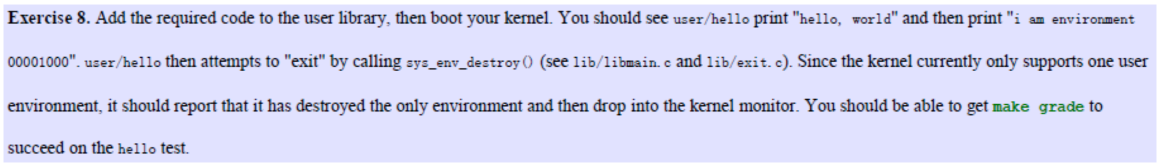
```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
```

输出 hello, world.并触发中断。

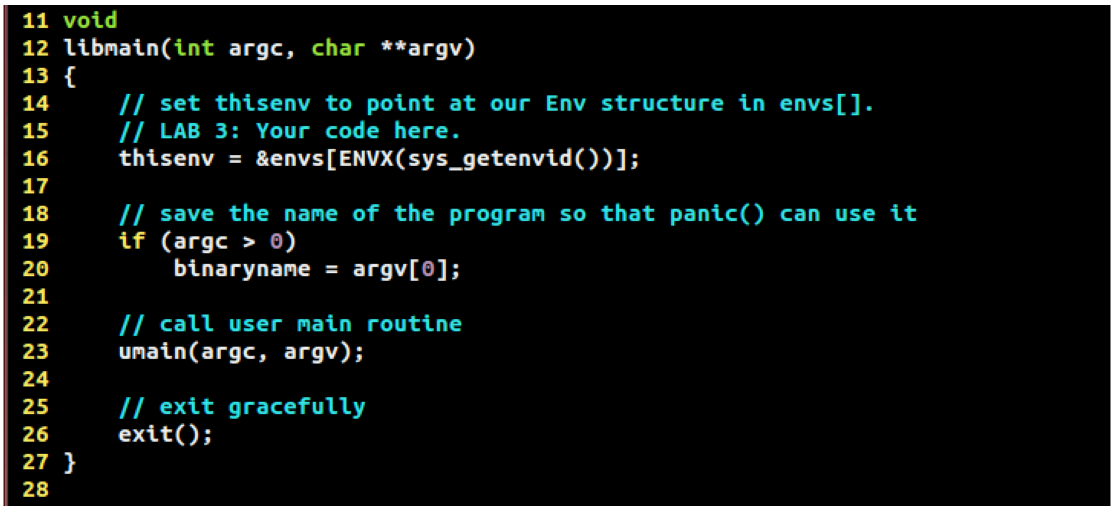


```
testbss: OK (1.1s)
```

通过测试。

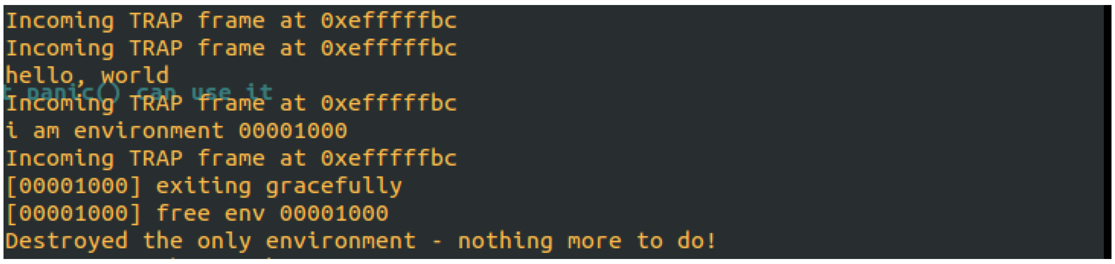


**Exercise 8.** Add the required code to the user library, then boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00001000`". `user/hello` then attempts to "exit" by calling `sys_env_destroy()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel 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.



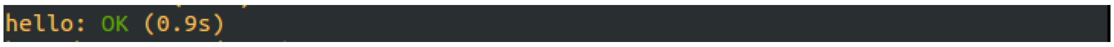
```c
void
libmain(int argc, char **argv)
{
    // set thisenv to point at our Env structure in envs[].
    // LAB 3: Your code here.
    thisenv = &envs[ENVX(sys_getenvid())];

    // 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();
}
```

初始化全局指针 thisenv ，让它指向当前用户环境的 Env 结构体。
env_id 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。可以通过获得 0～9 位得到这个用户环境对应的 Env 结构体。



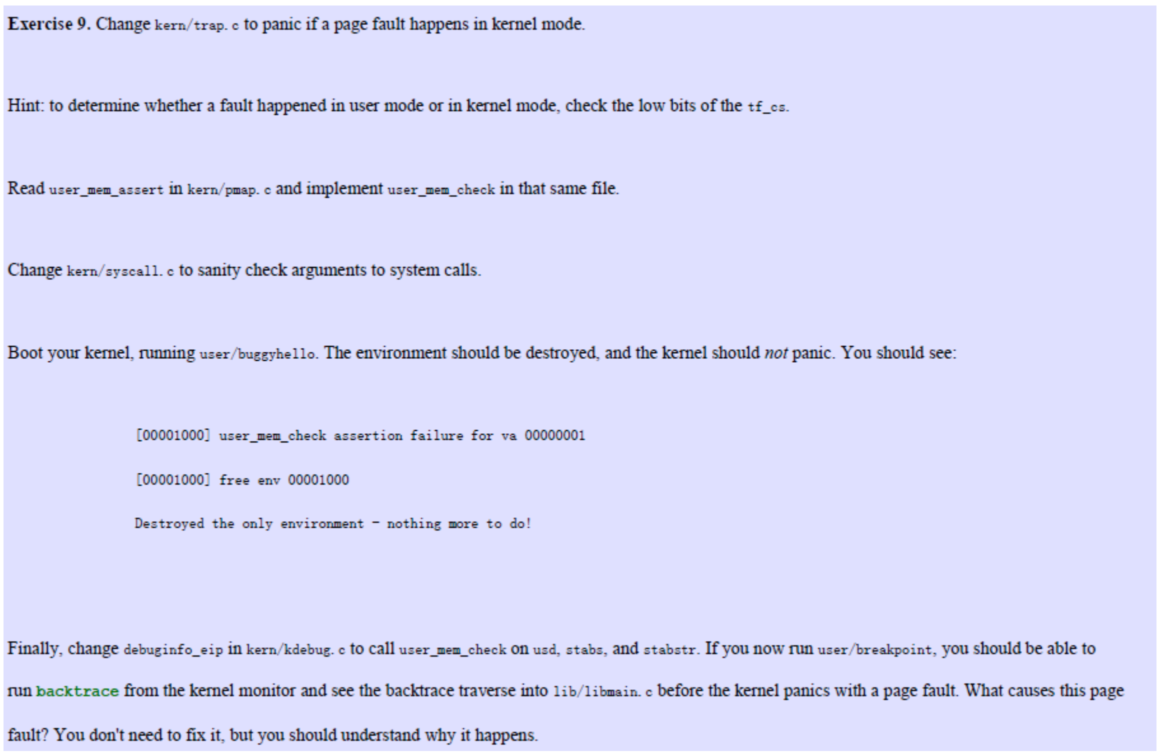
```
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
i am environment 00001000
Incoming TRAP frame at 0xefffffbc
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

输出题目的要求。



```
hello: OK (0.9s)
```

通过测试。



**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`.

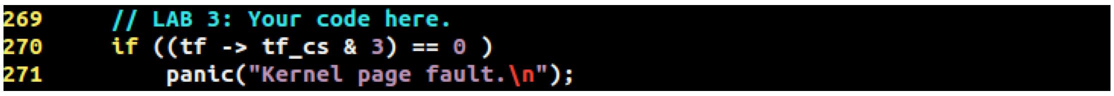
Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

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:

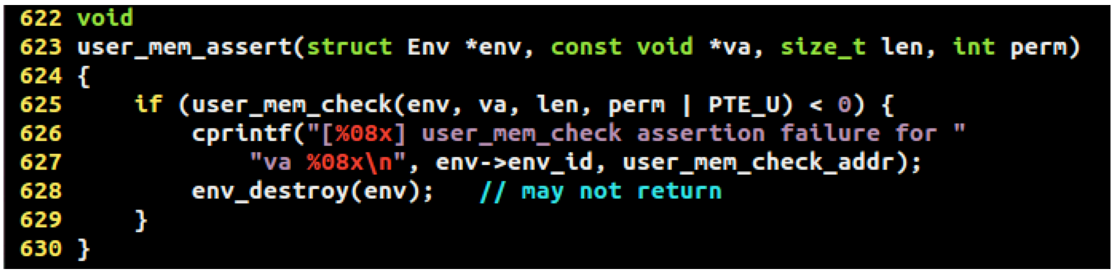
```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

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.



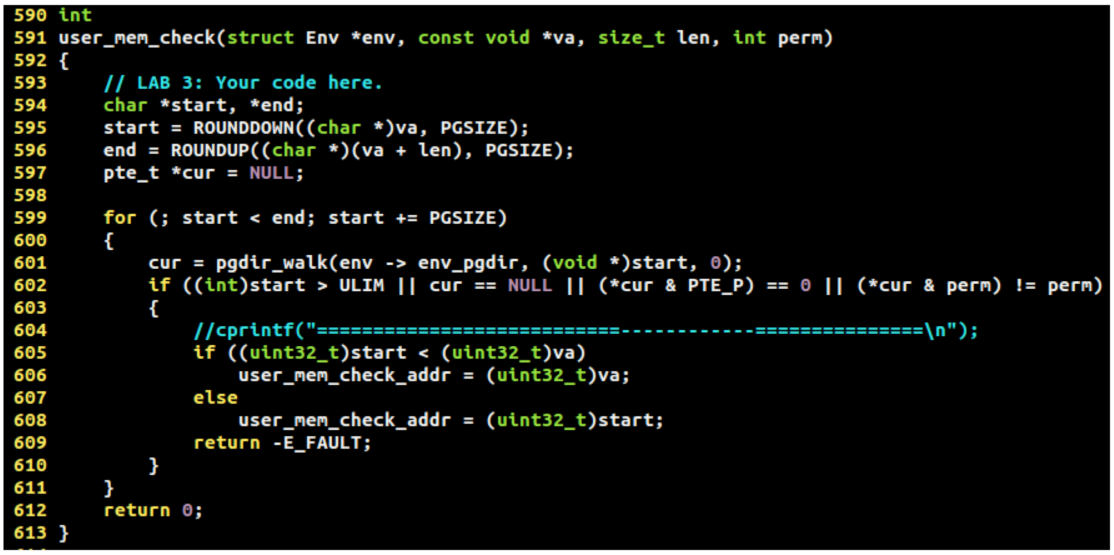
```
    // LAB 3: Your code here.
    if ((tf -> tf_cs & 3) == 0 )
        panic("Kernel page fault.\n");
```

在 page_fault_handler()函数中添加以上代码。
CS 段寄存器的低 2 位表示当前运行的代码的访问权限级别，0 代表是内核态，3 代表是用户态。如果检测到这个 page fault 是出现在内核态，将其 panic.



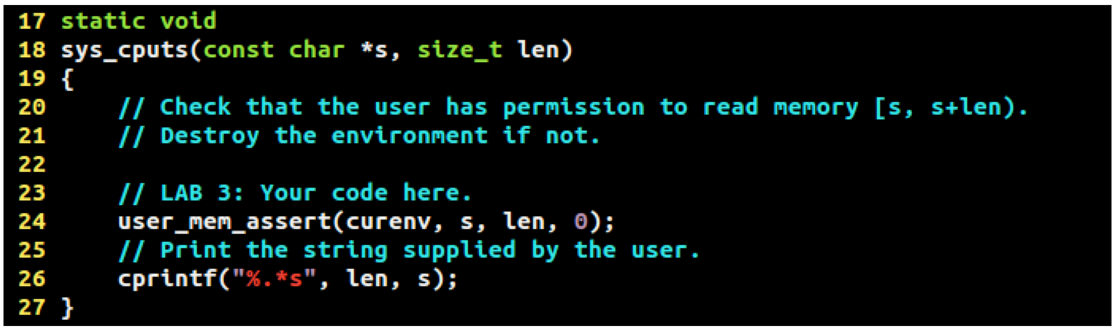
```
void
user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
{
    if (user_mem_check(env, va, len, perm | PTE_U) < 0) {
        cprintf("[%08x] user_mem_check assertion failure for "
            "va %08x\n", env->env_id, user_mem_check_addr);
        env_destroy(env);   // may not return
    }
}
```

观察 user_mem_assert()函数，发现它调用了 user_mem_check()函数。



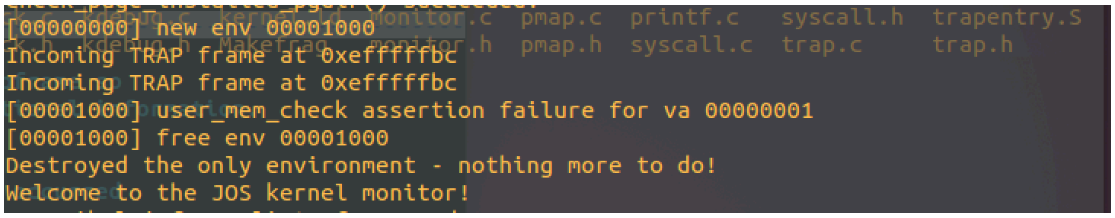
```c
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
    char *start, *end;
    start = ROUNDDOWN((char *)va, PGSIZE);
    end = ROUNDUP((char *)(va + len), PGSIZE);
    pte_t *cur = NULL;

    for (; start < end; start += PGSIZE)
    {
        cur = pgdir_walk(env -> env_pgdir, (void *)start, 0);
        if ((int)start > ULIM || cur == NULL || (*cur & PTE_P) == 0 || (*cur & perm) != perm)
        {
            //cprintf("==========================----------===============\n");
            if ((uint32_t)start < (uint32_t)va)
                user_mem_check_addr = (uint32_t)va;
            else
                user_mem_check_addr = (uint32_t)start;
            return -E_FAULT;
        }
    }
    return 0;
}
```

user_mem_check()函数。
检查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。
遍历所有地址。获得当前地址对应的页，如果无法取出或权限不正确，返回-E_FAULT.
如果没有错误则返回 0.



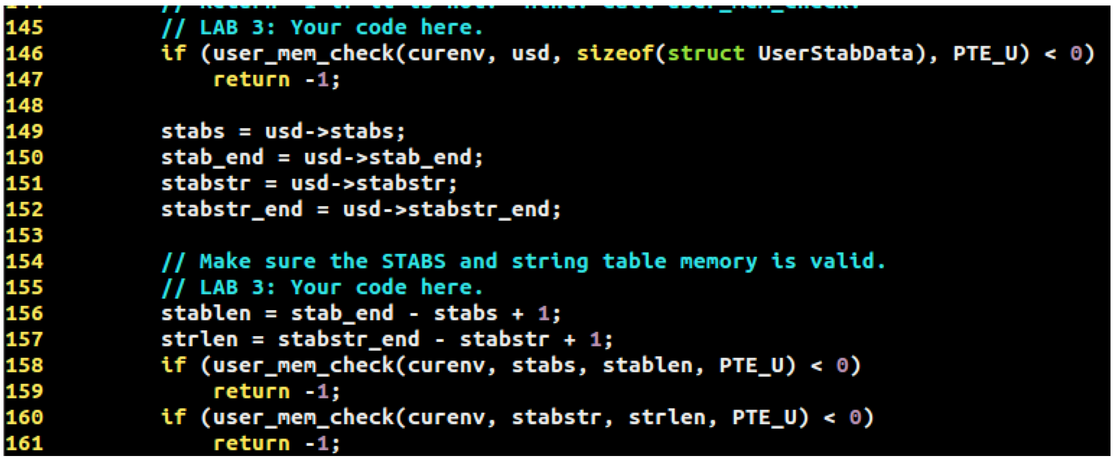
```c
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s, s+len).
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, 0);
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

sys_cputs()函数。
这个函数要求检查用户程序对虚拟地指空间 [s, s+len] 是否有访问权限。所以调用 user_mem_assert()函数来实现。



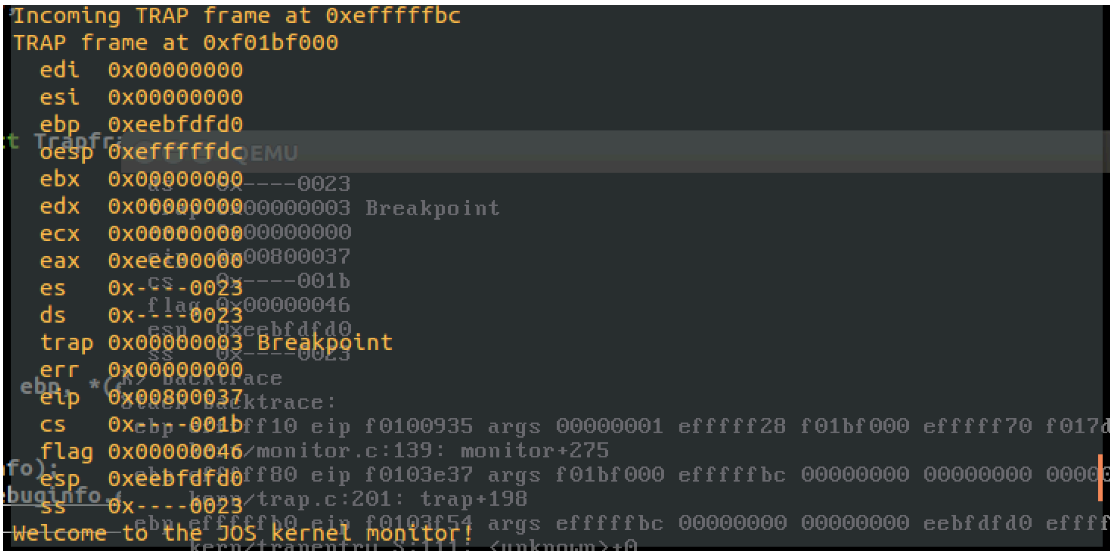
```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
```

make run-buggyhello 测试。可以看到触发了 user_mem_assert 中的错误信息。

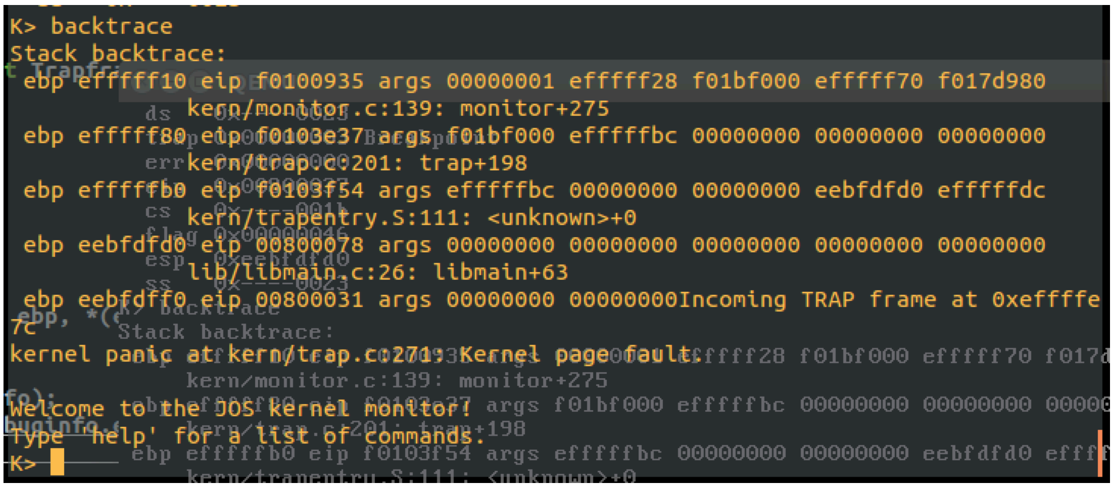


```
        // LAB 3: Your code here.
        if (user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U) < 0)
            return -1;

        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.
        stablen = stab_end - stabs + 1;
        strlen = stabstr_end - stabstr + 1;
        if (user_mem_check(curenv, stabs, stablen, PTE_U) < 0)
            return -1;
        if (user_mem_check(curenv, stabstr, strlen, PTE_U) < 0)
            return -1;
```

同理，使用 user_mem_check()检查 usd, stabs, stabstr.

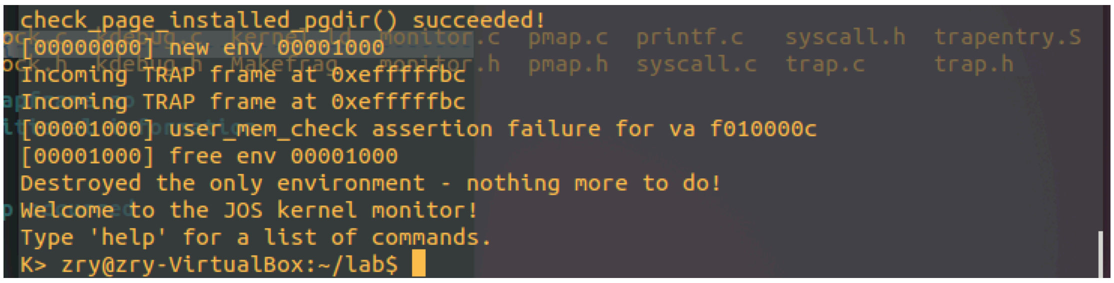


```
Incoming TRAP frame at 0xefffffbc
TRAP frame at 0xf01bf000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000046
  esp  0xeebfdfd0
  ss   0x----0023
Welcome to the JOS kernel monitor!
```



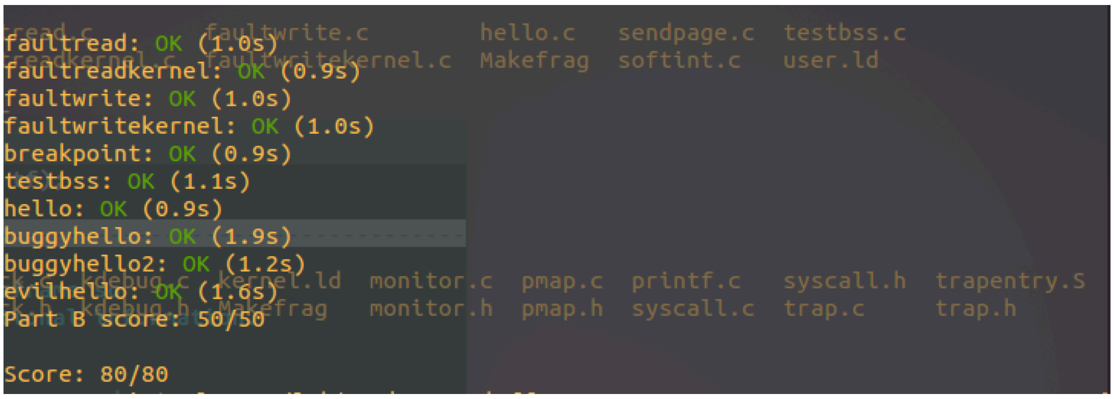
```
K> backtrace
Stack backtrace:
  ebp efffff10 eip f0100935 args 00000001 efffff28 f01bf000 efffff70 f017d980
         kern/monitor.c:139: monitor+275
  ebp efffff80 eip f0103e37 args f01bf000 efffffbc 00000000 00000000 00000000
         kern/trap.c:201: trap+198
  ebp efffffb0 eip f0103f54 args efffffbc 00000000 00000000 eebfdfd0 efffffdc
         kern/trapentry.S:111: <unknown>+0
  ebp eebfdfd0 eip 00800078 args 00000000 00000000 00000000 00000000 00000000
         lib/libmain.c:26: libmain+63
  ebp eebfdff0 eip 00800031 args 00000000 00000000Incoming TRAP frame at 0xeffffe
7c
kernel panic at kern/trap.c:271: Kernel page fault.
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

make run-breakpoint 及 backtrace 测试。



```
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> zry@zry-VirtualBox:~/lab$
```

make run-evilhello 测试。



```
faultread: OK (1.0s)
faultreadkernel: OK (0.9s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.0s)
breakpoint: OK (0.9s)
testbss: OK (1.1s)
hello: OK (0.9s)
buggyhello: OK (1.9s)
buggyhello2: OK (1.2s)
evilhello: OK (1.6s)
Part B score: 50/50

Score: 80/80
```

至此，Part B 完成。Score 80/80.

*Challenge!* 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.