Lab 3: User Environment

December 24, 2019

摘要

本次实验取自 MIT 6.828 Lab3，共分为两个部分。第一部分为用户环境与异常处理。第二部分为缺页中断、断点异常和系统调用。在本次实验中，将实现使保护模式下的用户进程得以运行的基础内核功能，并对 JOS 内核建立用于追踪用户进程的数据结构，创建用户进程，读入程序映像并运行。同时，要使 JOS 内核有能力响应用户进程的任何系统调用，并处理用户进程所造成的异常。

关键词：JOS、用户环境、中断、异常、系统调用

1 作业一

1.1 全局变量

kern/env.c 中定义的三个全局变量 envs、curenv、env_free_list 分别表示所有进程、当前正在运行的进程和空闲进程链表。

JOS 启动并开始运行，envs 指针便指向了一个 Env 结构体链表，表示系统中所有的用户环境的 env。JOS 内核把所有不活跃的 Env 结构体，用 env_free_list 链接起来，方便进行用户环境 env 的分配和回收。内核也会把 curenv 指针指向在任意时刻正在执行的用户环境的 Env 结构体。在内核启动时，并且还没有任何用户环境运行时，curenv 的值为 NULL。

1.2 ENV 结构体

inc/env 中定义了 Env 结构体，各部分功能如下：

```
struct Env
{
    struct Trapframe env_tf;
    struct Env *env_link;
    envid_t env_id;
    envid_t env_parent_id;
    enum EnvType env_type;
    unsigned env_status;
    uint32_t env_runs;

    pde_t *env_pgdir;
};
```

env_tf: 存储当进程没有在运行时被保存下来的寄存器的值。具体来说，当内核或者另一个不同的进程在运行的时候。内核在从用户模式切换到内核模式时将上一个进程的寄存器的值保存下来，从而进程可以从它被暂停的时间点恢复过来。

env_link: 指向下一个位于 env_free_list 中的 Env 结构体的指针。env_free_list 指向链表中第一个空闲进程。

env_id: 存储可以唯一确定正在使用这个 Env 数组的进程标识符。

env_parent_id: 存储创建这个进程的 env_id。通过这种方式，进程可以形成一个进程树 (family tree)，可用于在做一些安全决定时来确定某个进程能对哪些进程所执行的动作。

env_type: 用来区别一些特殊的进程。对于大多数进程，它会是 ENV_TYPE_USER。也可引入一些其他类型，来表示特殊的系统服务进程。

env_status: 这个变量的值可能是下面这些中的一个:

ENV_FREE: 这个 Env 结构未被利用，因此在 env_free_list 中。

ENV_RUNNABLE: 表示一个正在等待进入处理器运行的进程。

ENV_RUNNING: 表示目前正在运行的进程。

ENV_NOT_RUNNABLE: 表示目前已经激活的进程，但是它还没有准备

好运行,例如,它可能在等其他进程的进程间通信 (interprocess communication, IPC)。

ENV_DYING: 表示一个僵尸进程。僵尸进程将在下一次陷入内核的时候被释放。

env_pgdir: 储存这个进程的页目录的内核虚拟地址

1.3 作业一实现

作业 1 要求修改 kern/pmap.c 中的 mem_init() 函数来分配并映射 envs 数组。则对函数进行修改，首先要在 page_init() 之前对 envs 进行初始化并分配内存空间。然后要在 check_page() 函数之后的页表中设置它的映射关系。程序如下修改内容所示:

```
1 envs = (struct Env*)boot_alloc(NENV*sizeof(struct Env));
2 memset(envs, 0, NENV*sizeof(struct Env));
3 boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
```

make qemu 时发现 boot_map_region 函数中在 for 循环里定义变量类型，可以通过 gcc src.c -std=c99 -o src 将 GCC 换成 C99 标准或者将变量类型在 for 循环外面定义，从而可以看到 check_kern_pgdir() succeeded!

2 作业二

2.1 作业要求

在 env.c 中，完成以下函数

env_init() 初始化全部 envs 数组中的 Env 结构体，并将它们加入到空闲进程链表 env_free_list 中。

env_setup_vm() 为新的进程分配一个页目录，并初始化新进程的地址空间对应的内核部分。

region_alloc() 为进程分配和映射物理内存。

load_icode() 处理 ELF 二进制映像，并将映像内容读入新进程的用户地址空间。

env_create() 通过调用 env_alloc 分配一个新进程，并调用 load_icode 读入 ELF 二进制映像。

env_run() 启动给定的在用户模式运行的进程。

2.2 env_init()

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，并将它们插入到 env_free_list 中。envs 中的 env_link 指向的是下一个 env，为确保环境在空闲列表中的顺序与它们在 envs 数组中的顺序相同，也就是 envs[0] 在链表头部位置，插入时从后往前插入。

插入结束后调用 env_init_percpu 通过配置段硬件将其分隔为特权等级 0 (内核) 和特权等级 3（用户）两个不同的段。

```
1  void
2  env_init(void)
3  {
4      // Set up envs array
5      // LAB 3: Your code here.
6      int i=NENV;
7      env_free_list = NULL;
8      while (i>0) {
9           i--;
10         envs[i].env_id = 0;
11         envs[i].env_status = ENV_FREE;
12         envs[i].env_link = env_free_list;
13         env_free_list = &envs[i];
14     }
15     // Per-CPU part of the initialization
16     env_init_percpu();
17 }
```

2.3 env_setup_vm()

设置 e->env_pgdir 并初始化页目录，可参照 kern_pgdir 进行设置，将 p 转化为内核虚拟地址，然后以 kern_pgdir 为模板进行复制，注意要增加页引用。如果无法分配页目录或表返回-E_NO_MEM。

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

    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    e->env_pgdir = page2kva(p);
    memcpy(e->env_pgdir, kern_pgdir, PGSIZE);
    p->pp_ref++;

    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P |
       PTE_U;
     return 0;
}
```

2.4 region_alloc

分配 len 字节的物理内存给 env 环境，并将其映射到环境中的虚拟地址 va。先把 va 和 len 进行对齐，之后以页为单位，为其一个页一个页的分配内存，并且修改页目录表和页表。可利用 page_alloc() 完成内存页分配，page_insert() 完成虚拟地址到物理页的映射。

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
```

```
4       size_t pgnum = ROUNDUP(len, PGSIZE) / PGSIZE;
5       uintptr_t va_start = ROUNDDOWN((uintptr_t)va, PGSIZE);
6       struct PageInfo *pginfo = NULL;
7       cprintf("Allocate size: %d, Start from: %08x\n", len, va);
8       for (size_t i=0; i<pgnum; i++) {
9           pginfo = page_alloc(0);
10          if (! pginfo) {
11              int r = -E_NO_MEM;
12              panic("region_alloc: %e", r);
13              }
14          int r = page_insert(e->env_pgdir, pginfo, (void *)
                va_start, PTE_W | PTE_U | PTE_P);
15          if (r < 0) {
16              panic("region_alloc: %e", r);
17          }
18          cprintf("Va_start = %08x\n",va_start);
19          va_start += PGSIZE;
20      }
21  }
```

2.5 page_alloc()

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，处理 ELF 二进制映像，将映像内容读入新进程的用户地址空间，可参照 boot loader。

首先根据 ELF header 得出 Programm header。遍历所有 Programm header，分配好内存，加载类型为 ELF_PROG_LOAD 的段, 之后对用户栈进行分配。可通过 lcr3([页目录物理地址]) 将地址加载到 cr3 寄存器进行页目录切换。将 env->env_tf.tf_eip 设置为 elf->e_entry，等待之后的 env_pop_tf() 调用从而更改函数入口。

```
1   static void
```

```
2  load_icode(struct Env *e, uint8_t *binary)
3  {
4      // LAB 3: Your code here.
5      struct Elf* header = (struct Elf*)binary;
6
7      if(header->e_magic != ELF_MAGIC) {
8          panic("load_icode failed: The binary we load is not elf
               .\n");
9      }
10
11     if(header->e_entry == 0){
12          panic("load_icode failed: The elf file can't be
                excuterd.\n");
13     }
14
15     e->env_tf.tf_eip = header->e_entry;
16
17     lcr3(PADDR(e->env_pgdir));
18     struct Proghdr *ph, *eph;
19     ph = (struct Proghdr* )((uint8_t *)header + header->e_phoff
          );
20     eph = ph + header->e_phnum;
21     for(; ph < eph; ph++) {
22         if(ph->p_type == ELF_PROG_LOAD) {
23             if(ph->p_memsz - ph->p_filesz < 0) {
24                 panic("load icode failed : p_memsz < p_filesz.\
                       n");
25             }
26
27             region_alloc(e, (void *)ph->p_va, ph->p_memsz);
28             memmove((void *)ph->p_va, binary + ph->p_offset, ph
                   ->p_filesz);
```

```
29                memset((void *)(ph->p_va + ph->p_filesz), 0, ph->
                      p_memsz - ph->p_filesz);
30            }
31        }
32        lcr3(PADDR(kern_pgdir));
33        region_alloc(e,(void *)(USTACKTOP-PGSIZE), PGSIZE);
34    }
```

2.6 env_create

通过调用 env_alloc 分配一个新进程 env，调用 load_icode 读入 ELF 二进制映像，并设置 env_type，注意要将 env 的父 id 设置为 0。

```
1  void
2  env_create(uint8_t *binary, enum EnvType type)
3  {
4      // LAB 3: Your code here.
5      struct Env *e;
6      int r = env_alloc(&e, 0);
7      if (r)
8          panic("env_alloc failed");
9      load_icode(e, binary);
10     e->env_type = type;
11 }
```

2.7 env_run

启动给定的在用户模式运行的进程。进行上下文切换，首先判断当前环境是否为空，环境状态是不是 ENV_RUNNING，之后将 curenv 指向新的环境，状态设为 ENV_RUNNING，更新 env_runs 计数器，用 lcr3 切换到它的地址空间，使用 env_pop_tf() 储存环境计算器。

```
void
env_run(struct Env *e)
{
	// 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");
}
```

3 作业三

在 trapentry.S 中，根据是否需要压入错误码，使用 TRAPHANDLER 和 TRAPHANDLER_NOEC 为每个中断添加入口点，代码如下：

```
TRAPHANDLER_NOEC(t_divide, T_DIVIDE)     // 0
TRAPHANDLER_NOEC(t_debug, T_DEBUG)       // 1
TRAPHANDLER_NOEC(t_nmi, T_NMI)           // 2
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT)       // 3
TRAPHANDLER_NOEC(t_oflow, T_OFLOW)       // 4
TRAPHANDLER_NOEC(t_bound, T_BOUND)       // 5
TRAPHANDLER_NOEC(t_illop, T_ILLOP)       // 6
TRAPHANDLER_NOEC(t_device, T_DEVICE)     // 7
TRAPHANDLER(t_dblflt, T_DBLFLT)          // 8
                                         // 9
TRAPHANDLER(t_tss, T_TSS)                // 10
```

```
12 TRAPHANDLER(t_segnp, T_SEGNP)                // 11
13 TRAPHANDLER(t_stack, T_STACK)                // 12
14 TRAPHANDLER(t_gpflt, T_GPFLT)                // 13
15 TRAPHANDLER(t_pgflt, T_PGFLT)                // 14
16                                                 // 15
17 TRAPHANDLER_NOEC(t_fperr, T_FPERR)            // 16
18 TRAPHANDLER(t_align, T_ALIGN)                // 17
19 TRAPHANDLER_NOEC(t_mchk, T_MCHK)             // 18
20 TRAPHANDLER_NOEC(t_simderr, T_SIMDERR)       // 19
21
22 TRAPHANDLER_NOEC(t_syscall, T_SYSCALL)
```

而公共代码 _alltraps 则压入旧的 ds 和 es，还有通用寄存器的值，将 GD_KD 赋给 ds 和 es，再将 esp 压栈，作为 trap 的参数：

```
1  _alltraps:
2      pushl %ds
3      pushl %es
4      pushal
5
6      movw $GD_KD, %eax
7      movw %ax, %ds
8      movw %ax, %es
9
10     pushl %esp
11     call trap
```

trap.init() 用来初始化 IDT：

```
1      void t_divide();
2      void t_debug();
3      void t_nmi();
```

```
4       void t_brkpt();
5       void t_oflow();
6       void t_bound();
7       void t_illop();
8       void t_device();
9       void t_dblflt();
10      void t_tss();
11      void t_segnp();
12      void t_stack();
13      void t_gpflt();
14      void t_pgflt();
15      void t_fperr();
16      void t_align();
17      void t_mchk();
18      void t_simderr();
19      void t_syscall();
20      SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
21      SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
22      SETGATE(idt[T_NMI], 0, GD_KT, t_nmi, 0);
23      SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 3);
24      SETGATE(idt[T_OFLOW], 0, GD_KT, t_oflow, 0);
25      SETGATE(idt[T_BOUND], 0, GD_KT, t_bound, 0);
26      SETGATE(idt[T_ILLOP], 0, GD_KT, t_illop, 0);
27      SETGATE(idt[T_DEVICE], 0, GD_KT, t_device, 0);
28      SETGATE(idt[T_DBLFLT], 0, GD_KT, t_dblflt, 0);
29      SETGATE(idt[T_TSS], 0, GD_KT, t_tss, 0);
30      SETGATE(idt[T_SEGNP], 0, GD_KT, t_segnp, 0);
31      SETGATE(idt[T_STACK], 0, GD_KT, t_stack, 0);
32      SETGATE(idt[T_GPFLT], 0, GD_KT, t_gpflt, 0);
33      SETGATE(idt[T_PGFLT], 0, GD_KT, t_pgflt, 0);
34      SETGATE(idt[T_FPERR], 0, GD_KT, t_fperr, 0);
35      SETGATE(idt[T_ALIGN], 0, GD_KT, t_align, 0);
```

```
36      SETGATE(idt[T_MCHK], 0, GD_KT, t_mchk, 0);
37      SETGATE(idt[T_SIMDERR], 0, GD_KT, t_simderr, 0);
38      SETGATE(idt[T_SYSCALL], 0, GD_KT, t_syscall, 3);
```

4 问题一

4.1 第一问

每种异常/中断都有一个单独的 handler 函数是为每个中断实现不同的处理函数是为了区分，以便后续做相应的处理。

4.2 第二问

触发的是中断向量 13 是因为我们在 SETGATE 中对中断向量 14 设置的 DPL 为 0，从而由于用户程序 CPL=3，触发了 13 异常。如果内核允许 softint 的 int $14 指令去调用内核中断向量 14 所对应的缺页处理函数，可以设置中断向量 14 的 DPL 为 3，但是我们是不希望用户程序来操作内存的。

5 作业四

大内核锁的实现：

```
1  void
2  spin_lock(struct spinlock* lk)
3  {
4      #ifdef DEBUG_SPINLOCK
5      if (holding(lk))
6          panic("CPU %d cannot acquire %s: already holding",
               cpunum(), lk->name);
7      #endif
8
9      // The xchg is atomic.
```

```
10      // It also serializes, so that reads after acquire are not
11      // reordered before it.
12
13      while (xchg(&lk->locked, 1) != 0)
14          asm volatile ("pause");
15
16      // Record info about lock acquisition for debugging.
17      #ifdef DEBUG_SPINLOCK
18      lk->cpu = thiscpu;
19      get_caller_pcs(lk->pcs);
20      #endif
21  }
```

其中，在 inc/x86.h 中可以找到 xchg() 函数的实现，使用它而不是用简单的 if + 赋值是因为它是一个原子性的操作。

```
1   static inline uint32_t
2   xchg(volatile uint32_t* addr, uint32_t newval)
3   {
4       uint32_t result;
5       // The + in "+m" denotes a read-modify-write operand.
6       asm volatile("lock; xchgl %0, %1"
7           : "+m" (*addr), "=a" (result)
8           : "1" (newval)
9           : "cc");
10      return result;
11  }
```

这是一段内联汇编。lock 确保了操作的原子性，其意义是将 addr 存储的值与 newval 交换，并返回 addr 中原本的值。于是，如果最初 locked = 0，即未加锁，就能跳出这个 while 循环。否则就会利用 pause 命令自旋等待。确保了当一个 CPU 获得了 BKL，其他 CPU 如果要获得就只能自旋等待。

在这几处加大内核锁为了避免多个 CPU 同时运行内核代码，保证独立性。由于分页机制的存在，内核以及每个用户进程都有自己的独立空间。而多进程并发的时候，如果两个进程同时陷入内核态，就无法保证独立性了。例如内核中有某个全局变量 A，cpu1 让 A=1，而后 cpu2 却让 A=2，显然会互相影响。为了使系统尽快支持 SMP，直接在内核入口加大内核锁锁，保证其独立性。其流程大致为：BPS 启动 AP 前，获取内核锁，所以 AP 会在 mp_main 执行调度之前阻塞，在启动完 AP 后，BPS 执行调度，运行第一个进程，env_run() 函数中会释放内核锁，这样一来，其中一个 AP 就可以开始执行调度，运行其他进程。

```
// i386_init()
// Your code here:
lock_kernel();
boot_aps();
```

在唤醒其他 CPU 前需要 lock，防止唤醒的 CPU 启动进程

```
// mp_main()
// Your code here:
lock_kernel();
sched_yield();
```

初始化 AP 后，在调度之前需要 lock，防止其他 CPU 干扰进程的选择

```
// trap()
// LAB 4: Your code here.
lock_kernel();
assert(curenv);
```

用户态引发中断陷入内核态时，需要 lock

```
// env_run()
lcr3(PADDR(e->env_pgdir));
```

```
3 unlock_kernel();
4 env_pop_tf(&(e->env_tf));
```

离开内核态之前，需要 unlock；BPS 启动 AP 前，获取内核锁，所以 AP 会在 mp_main 执行调度之前阻塞，在启动完 AP 后，BPS 执行调度，运行第一个进程，之后释放内核锁，这样一来，其中一个 AP 就可以开始执行调度，若有的话运行进程。

6 问题二

6.1 第一问

出现一般保护异常的原因是将断点的权限级别设置为 0（内核级别），因此用户访问（CPL=3）肯定会出现保护错误（因为出现 CPL > DPL 的情况）。若将断点异常的权限级别设置为 3，保护错误就将消失。

6.2 第二问

如果内核栈中留下不同 CPU 之后需要使用的数据，可能会造成混乱。例如在某进程即将陷入内核态的时候（尚未获得锁），系统在 trap() 函数之前已经在 trapentry.S 中对内核栈进行了操作，压入了寄存器信息。如果共用一个内核栈，那显然会导致信息错误。

7 作业五

填充函数 trap_dispatch() ，让断点中断启动 monitor() 函数在 kern/trap.c 文件中，trap_dispatch() 函数，对于参数代表断点中断的情况，调用 monitor 函数在 inc/trap.h 文件中，找到断点中断名为 T_BRKPT，值为 3 在 kern/monitor.c 文件中能找到 monitor 函数：void monitor(struct Trapframe tf) 添加代码为：

```
1 if(tf->tf_trapno==T_BRKPT)
2 {
3     monitor(tf);
```

```
4       return;
5   }
```

8 作业六

8.1 实验要求

在内核中为中断 T_SYSCALL 添加一个处理程序。完成在 kern/trapentry.S 和 kern/trap.c 文件中的 trap_init() 修改 trap_dispathch() 来处理系统调用中断，通过调用 syscall() 在文件 kern/syscall.c 中），使用合适的参数并将返回值写回%eax 完成 kern/syscall.c 文件中的 syscall() 函数。如果系统调用号是无效的，syscall() 函数返回-E_INVAL。阅读并理解 lib/syscall.c 文件可以让你更加理解系统调用。在你的内核运行 user/hello 程序（make run hello）。它将在控制台输出“hello world”然后会引起一个用户模式下的缺页中断。

8.2 解决方案

在 Kern/Trapentry.S 中添加以下代码，为 T_SYSCALL 分配异常处理函数，名为 system_call。

修改 trap.c 中的函数 trap_dispatch( )。

添加对 T_SYSCALL 的控制语句。当异常编号为 T_SYSCALL 时，调用 syscall 函数，参数为当前各寄存器状态，并将返回值保存到寄存器 eax 中。

在 trap.c 的 trap_init( ) 函数中添加代码

完善 syscall.c 中的 syscall( ) 函数，根据不同的 syscallno 调用不同的函数。如果调用号是无效的，则返回 -E_INVAL。Run hello.c：

9 作业七

subsection 实验要求 添加所需代码到用户字典中，然后启动你的内核。让它可以使 user/hello 打印出 “hello，world” 然后打印 “i am environment 00001000”。然后 user/hello 会尝试调用 sys_env_destroy() 退出（详见 lib/libmain.c 和 lib/exit.c）。由于内核当前只支持一个程序，所以当前程序退出后，内核就会显示当前唯一的程序已经退出并且陷入内核监视器。此时，make grade 就可以通过 hello 测试了。

9.1 解决方案

修改 libmain.c 中的函数 libmain( ) ，添加如下代码：

使用宏 ENVS 从 envied 得到 env 在 envs 中的偏移量，将地址赋给 thisenv。Make grade：

10 作业八

首先根据文档，要判断缺页是发生在用户模式还是内核模式下，需检查 tf_cs 的低位，在 kern/trap.c 的函数 page_fault_handler 中添加代码如下：

```
1 if ((tf->tf_cs&3) == 0)
2     panic("Kernel page fault!");
```

在 kern/pmap.c 实现函数 user_mem_check，检测线性地址是否有效：

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

根据注释，用户程序可获得一个线性地址仅当：1）该地址低于 ULIM；2）页表给与其权限 perm | PTE_P。

接着，在 kern/syscall.c 的 sys_cputs 中检查用户空间地址：

```
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s, s+
```

```
        len).
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, PTE_U);

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

此时，执行 make run-buggyhello，进程会被销毁，内核不会恐慌。

最后，修改在 kern/kdebug.c 的 debuginfo_eip，对 usd,stabs,stabstr 分别调用 user_mem_check。

```
// Make sure this memory is valid.
// Return -1 if it is not.  Hint: Call user_mem_check.
// LAB 3: Your code here.
if (user_mem_check(curenv, usd, sizeof(struct UserStabData),
    PTE_U))
    return -1;

    ...
// Make sure the STABS and string table memory is valid.
// LAB 3: Your code here.
if (user_mem_check(curenv, stabs, sizeof(struct Stab), PTE_U))
    return -1;
if (user_mem_check(curenv, stabstr, stabstr_end-stabstr, PTE_U)
    )
    return -1;
```

11 作业九

根据文档，上面实现的机制对恶意的用户程序也同样有效，进程会被销毁，内核不会恐慌。

12 实验结果

执行 make grade，得到结果如下：

```
+ 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 380 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]:正在离开目录 `/home/zxj/lab1/src/lab3'
divzero: OK (1.7s)
softint: OK (1.0s)
badsegment: OK (0.9s)
Part A score: 30/30

faultread: OK (0.9s)
faultreadkernel: OK (1.0s)
faultwrite: OK (1.1s)
faultwritekernel: OK (0.9s)
breakpoint: OK (0.9s)
testbss: OK (1.1s)
hello: OK (0.9s)
buggyhello: OK (1.1s)
buggyhello2: OK (0.9s)
evilhello: OK (1.8s)
Part B score: 50/50

Score: 80/80
```

图 12.1: lab3 实验结果-make grade