

实验 1 进程/线程

班级：__谢志鹏__ 姓名：__王傲__ 学号：__15300240004__

请于 2017 年 10 月 18 日实验课后当天内，上传到 <ftp://10.141.251.211>。

(用户名: *stu*, 密码: *os*)

实验 1. [进程数据结构]

阅读材料，熟悉 `kern/env.c`、`inc/env.h` 中的代码。根据注释修改 `kern/pmap.c` 中的 `mem_init()` 函数，为内核分配 `envs` 数组，并设定适当权限。

meminit() in kern/pmap.c:

为内核分配 `envs` 数组：

与分配 `page` 类似，分配好后用 `memset` 填充 0

```
envs = (struct Env *) boot_alloc(sizeof(struct Env) * NENV);
```

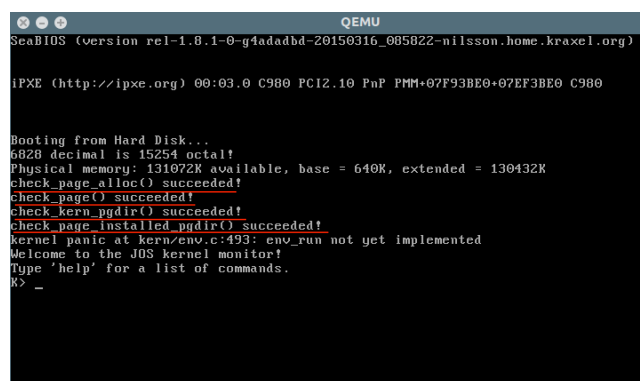
```
memset(envs, 0, sizeof(struct Env) * NENV);
```

设定权限：

仍然与对 `page` 的权限设定类似，使用 `boot_map_region` 函数进行映射并设定权限

```
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

运行结果：



实验 2. [进程初始化]

参考相关资料，根据注释补全 kern/env.c 中的以下函数:env_init()、env_create()和 env_run()，为进程启动做好准备工作。

env_init() in kern/env.c:

```
int count;//is used to count

struct Env* next_addr=NULL;//record the address of the next struct Env,
forming a linked table

//the distribution of the array envs is done in the question 1

for (count=NENV-1;count>=0;count--)//start from the end, so that the head
of the link table is envs[0]
{
    envs[count].env_id = 0;//initiallize
    envs[count].env_parent_id = 0;
    envs[count].env_type = ENV_TYPE_USER;
    envs[count].env_status = 0;
    envs[count].env_runs = 0;
    envs[count].env_pgdir = NULL;
    envs[count].env_link = next_addr;//to form a link table
    next_addr=&envs[count];//to record the address of temporal envs
}

env_free_list=&envs[0];//set the head of the link table
```

注：初始化 env_free_list 为链表，每个 Env 相应的初始化，利用 env_link 连接，注意 env_free_list 指向 envs 数组的第一个元素

env_create() in kern/env.c:

```
int state = 0;

struct Env *env_pointer = NULL;

state = env_alloc(&env_pointer, 0); //to allocate an Env

if(state < 0)
{
    panic("env create fail");
}

load_icode(env_pointer, binary); //to parse an ELF binary image
//and load its contents into the user address space of the new environment

env_pointer->env_type = type; //set the EnvType as given
```

注：利用 env_alloc 分配一个 Env，用 load_icode 划分一个 ELF 二进制文件并加载到相应的用户地址空间，更改 env_type 属性。

env_run() in kern/env.c:

```
//Step 1

if(curenv && curenv->env_status == ENV_RUNNING)
{
    curenv->env_status = ENV_RUNNABLE;
}

curenv = e; //e is given as the parameter of the function
e->env_status = ENV_RUNNING;
e->env_runs++;

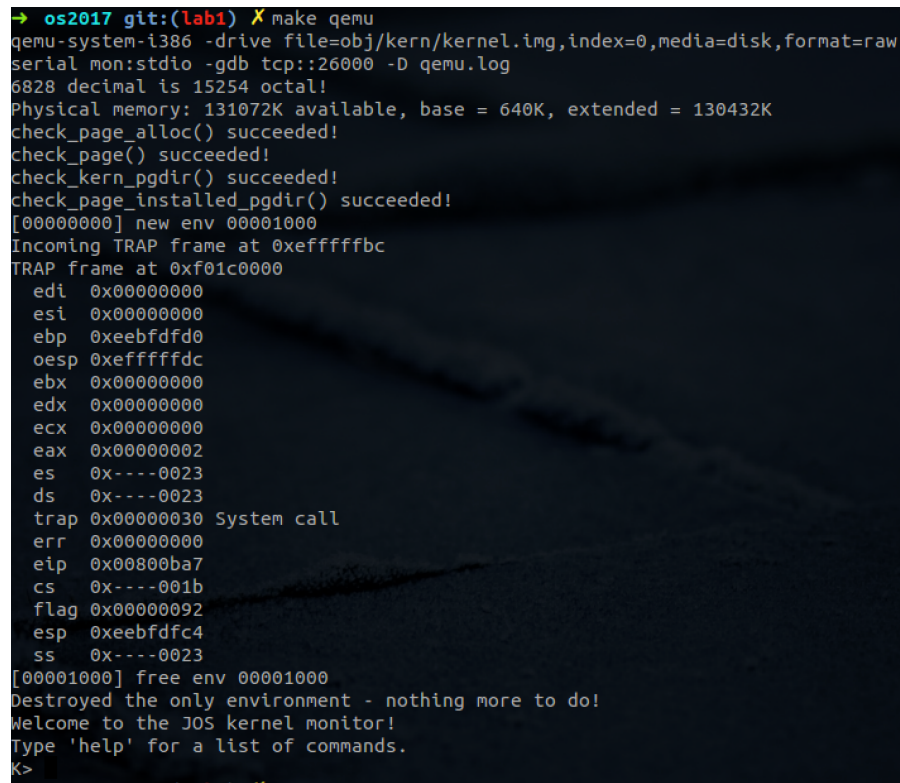
lcr3(PADDR(e->env_pgdir));

//Step 2
```

```
env_pop_tf(&(e->env_tf)); //use the register parameters stored in the trap
frame to complete env switch
```

注：依照源文件的要求逐步完成。这里 env_run 函数实际上起着替换进程的作用，只不过初始化时 curenv 是 NULL。

运行结果：



```
→ os2017 git:(lab1) X make qemu
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 15254 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!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffb4
TRAP frame at 0xf01c0000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfdc0
  oesp 0xefffffb4
  ebx 0x00000000
  edx 0x00000000
  ecx 0x00000000
  eax 0x00000002
  es 0x---0023
  ds 0x---0023
  trap 0x00000030 System call
  err 0x00000000
  eip 0x00800ba7
  cs 0x---001b
  flag 0x00000092
  esp 0xeebdfdc4
  ss 0x---0023
[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>
```

使用 make grade 命令，发现一些任务的前几条变为了 GOOD，但没有能完全完成的。

实验 3. [系统调用处理]

阅读材料中关于中断、异常处理、系统调用的内容。完成：

- (1) 修改 kern/trap.c 中的 trap_dispatch() 方法，在其中调用 kern/syscall.c 中的 syscall() 方法处理系统调用，注意参数和返回值。

代码解释见注释

syscall() in kern/trap.c:

```

    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;

    //in the description of lib/syscall.c, the order of the five
parameters of the function syscall is DX, CX, BX, DI, SI, while the first one
is syscallno

```

注：syscall 的第一个属性为系统调用编号，对应相应的系统调用，后面按照 DX, CX, BX, DI, SI 的寄存器顺序加载参数，参数顺序根据 lib/syscall.c 确定。

(2) 实现 kern/syscall.c 中的 syscall() 方法，调用 lib/syscall.c 中相应方法正确处理 inc/syscall.h 中列出的所有系统调用。

syscall() in kern/syscall.c:

```

switch (syscallno)
{
    //possible states of syscallno are in inc/syscall.h

    case SYS_cputs:
        sys_cputs((const char *) a1, a2); //function impleted in
lib/syscall.c, to output character

        return 0;

    case SYS_cgetc:
        return sys_cgetc(); //to input character

    case SYS_getenvid:
        return sys_getenvid(); //to get the id of the env

    case SYS_env_destroy:
        return sys_env_destroy(a1); //to destroy the env

    default:
        return -E_INVALID;
}

```

```
}
```

注：根据 inc/syscall.h 确定 syscallno 的可能取值，即相应的 syscall 情况，其余函数在 lib/syscall.c 中定义，进行相应调用。

运行结果：

```
→ os2017 git:(lab1) X make qemu
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 15254 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!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbfc
Incoming TRAP frame at 0xeffffbfc
hello, world
Incoming TRAP frame at 0xeffffbfc
I am environment 00001000
Incoming TRAP frame at 0xeffffbfc
[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>
→ os2017 git:(lab1) X
```

```
divzero: OK (1.5s)
softint: OK (0.9s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (1.0s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.0s)
breakpoint: OK (1.0s)
testbss: OK (1.9s)
hello: OK (1.1s)
buggyhello: OK (1.8s)
buggyhello2: OK (2.2s)
evilhello: OK (2.0s)
Part B score: 50/50

Score: 80/80
```

实验 4. [程序分析]

选择 user 目录下的两个程序(对于程序 xxx, 使用 `makerun-xxx` 命令运行), 分别对照代码分析其运行过程(涉及权限、异常、系统调用等)

首先需要说明 JOS 的启动过程。

JOS 使用结构体 `Env` 来存储相应的进程(环境)信息, 具体内容见上。JOS 使用进程表 `env_free_list` 和 `envs` 来管理所有的进程, 如同链表一样操作, 完成进程的申请, 释放等操作。

启动的具体过程如下:

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

首先, 调用 `cons_init` 函数, 启动 console。然后, 调用 `mem_init` 和 `env_init` 函数, 进行初始化, 分配内存空间、进程空间并进行虚拟空间的映射。

再然后, 调用 `trap_init` 函数, 对 `trap` 进行初始化。`trap_init` 函数对 IDT 信息做了处理, 将所有中断处理函数的起始地址放到中断向量表 IDT, 然后调用 `trap_init_percpu` 函数。在 `trap_init_percpu` 函数中, 设置了进程陷入内核态时的相关处理。对于 JOS, 处理器由用户态转变到内核态时, 会从当前进程的堆栈通过 TSS 切换到内核的堆栈, 并存储相应寄存器信息。在 `trap_init_percpu` 函数中, 我们可以看见, TSS 的 `ts_esp0` 为 `KSTACKTOP`, 即指向内核堆栈的栈顶, 而 `ts_ss0` 为 `GD_KD`, 指明了内核堆栈的位置和大小。当发生从用户态到内核态的切换时, 处理器会切换到内核态的堆栈, 将相关寄存器的信息压入内核态堆栈, 并调用相关的 `trap` 处理函数。

初始化 `trap` 的相关信息后, 通过调用 `env_create` 函数, 加载用户程序的 ELF 文件, 并最终通过 `env_run` 函数运行该进程。`env_run` 函数实际起到的作用是将当前进程切换为输入的参数, 即进程 `e`。启动时, 当前进程 (`curenv`) 为 `NULL`。其中, `env_pop_tf` 函数给寄存器赋值, 并通过 `iret` 指令退出中断程序, 退出内核态, 开始或继续用户程序的运行。

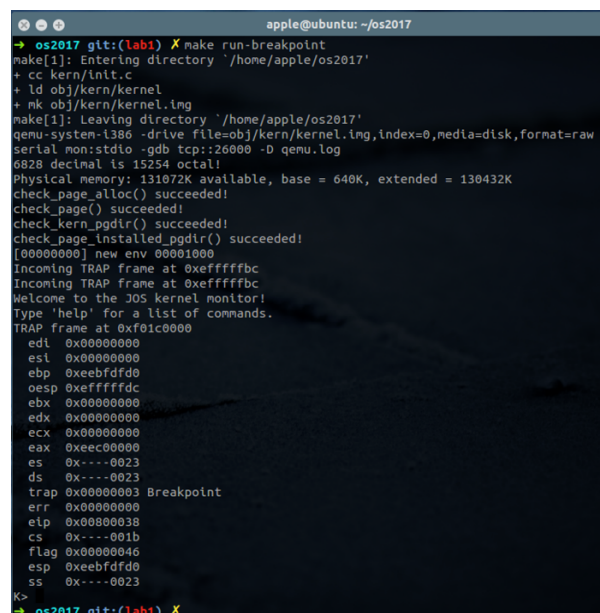
发生内核态和用户态之间的切换时，会发生内核栈和用户栈的切换，CS 的值也会发生变化（0 为内核态，3 为用户态）。

(1) 程序 A:

breakpoint.c, 断点程序。

产生中断的语句为 `asm volatile("int $3");` 即发送中断向量为 3 的中断指令。从 `inc/trap.h` 中定义的宏和 `kern/trap.c` 中定义的中断向量表可以看出，3 号中断向量号对应着断点中断 Breakpoint。

程序的运行结果如图：



```
apple@ubuntu: ~/os2017
→ os2017 git:(lab1) X make run-breakpoint
make[1]: Entering directory '/home/apple/os2017'
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/apple/os2017'
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 15254 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!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01c0000
  edi 0x00000000
  esi 0x00000000
  ebp 0xeebdfdd0
  oesp 0xeffffdc
  ebx 0x00000000
  edx 0x00000000
  ecx 0x00000000
  eax 0xeec00000
  es  0x---0023
  ds  0x---0023
  trap 0x00000003 Breakpoint
  err 0x00000000
  eip 0x00800038
  cs  0x---001b
  flg 0x00000046
  esp 0xeebdfdd0
  ss  0x---0023
K>
→ os2017 git:(lab1) X
```

首先需要注意的几点：1. Trapframe 是一个 CPU 执行任务（执行任务可能只是一小段汇编代码，中间调用了 `int` 中断）中当发生切换时（产生中断）的当前执行状态（CPU 状态），包括一些比较重要的寄存器的值和状态位等（从 `inc/trap.h` 中 Trapframe 的定义可以看出）。但与中断控制中的 TSS 不同，虽然他们都是类似的状态保存结构，但是 TSS 是在引入用户进程（Env）之后产生的概念，即当一个用户进程切换到另一进程时，不仅 CPU 的运行状态需要保存以便切换和恢复，并且它们的虚拟地址环境（页目录，CR3）等等也需要保存，这是不同的两个概念。2. 此时 JOS 并没有文件系统，现在的做法是将文件编译后和内核连接到一起（通过上面的 `load_icode` 函数）。

首先，查看 obj/user/breakpoint.asm 源码，发现在 0x00800037 处调用 int 0x3 指令产生中断，因此在此处打上断点，查看这条指令前后寄存器的变化。

```
Breakpoint 1, 0x00800037 in ?? ()
(gdb) p $esp
$1 = (void *) 0xeebdfd0
(gdb) si
=> 0xf0103e9c: push    $0x3
0xf0103e9c    52      TRAPHANDLER_NOEC(th3, T_BRKPT)
(gdb) p $esp
$2 = (void *) 0xfffffe8
(gdb) x/6x 0xfffffe8
0xfffffe8: 0x00000000    0x00800038    0x0000001b    0x00000046
0xfffffe8: 0xeebdfd0     0x00000023
```

可以看出，在经过 int 指令后，esp 寄存器的值发生了变化，表明了发生了栈的切换。事实上，int 指令产生了系统调用，切换到了内核堆栈，并压入了相关数据。具体的，查看执行 int 指令前的寄存器情况：

```
Breakpoint 1, 0x00800037 in ?? ()
(gdb) info r
eax            0xeec00000    -289406976
ecx            0x0          0
edx            0x0          0
ebx            0x0          0
esp            0xeebdfd0     0xeebdfd0
ebp            0xeebdfd0     0xeebdfd0
esi            0x0          0
edi            0x0          0
eip            0x800037     0x800037
eflags         0x46        [ PF ZF ]
cs             0x1b        27
ss             0x23        35
ds             0x23        35
es             0x23        35
fs             0x23        35
gs             0x23        35
```

可以看出，压入的寄存器顺序是 ss, esp, eflags, cs, eip, 最终是一个错误码。所以，调用 int 指令后发生的事情是：1. 处理器会首先切换自己的堆栈，切换到由 TSS 的 SS0, ESP0 字段所指定的内核堆栈区，这两个字段分别存放着 GD_KD 和 KSTACKTOP 的值，同时完成用户态到内核态的转换 2. 把相关寄存器信息压入内核堆栈中。3. 通过 IDT 加载 CS 和 EIP，控制转移至中断处理函数。

之后，进入中断处理函数。在 kern/trapentry.s 中定义了两个宏：TRAPHANDLER_NOEC 和 TRAPHANDLER。两者功能比较类似，即在内核栈中设置好一个 Trapframe 的布局，同时定义出一个相应的中断处理程序，即向栈里压入相关错误码和中断号。两者区别在于没有错误号时 TRAPHANDLER_NOEC 压入一个 0。

随后，调用 _alltrap，接着调用 trap 函数。其中，if ((tf->tf_cs & 3) == 3) 用来判断进入 trap 的进程是否是内核态，因为 cs 寄存器的最后两位是 RPL，用来表明进程的特权级。

随后，执行 `trap_dispatch` 函数，完成具体的中断处理程序的分发。在 `trap_dispatch` 中，会根据中断类型进行相应处理。JOS 的中断一共有三种：page fault，中断和系统调用。

仍然使用 GDB 逐步调试，结果如图：

```
194         if (tf->tf_trapno == T_BRKPT) {
(gdb) s
=> 0xf0103dd3 <trap+172>:      mov     %esi,%esp
195         monitor(tf);
(gdb) s
=> 0xf0100851 <monitor+9>:      movl    $0xf0105500,%esp
monitor (tf=0xf01c0000) at kern/monitor.c:133
133         cprintf("Welcome to the JOS kernel monitor!\n");
```

可以看出，在 `trap_dispatch` 中对 `trapno` 进行判断时，符合中断类型（`T_BRKPT`），即中断，使用 `monitor` 函数对 `Trapframe` 进行处理，调用 `print_trapframe` 函数输出 `Trapframe` 信息。之后，`trap` 函数调用 `env_run(curenv)`，执行用户进程，完成进程的切换，同时切换回用户态。注意，由于不是异常，进程最终没有销毁。

（2） 程序 B:

`divzero.c`，除数为 0 程序。

产生中断的语句为 `cprintf("1/0 is %08x!\n", 1/zero);` 查看 `obj/user/divzero.asm` 源码，发现在 `0x00800053` 处，`idiv %ecx` 指令导致抛出异常：

```
(gdb) si
=> 0x800053:      idiv     %ecx
0x00800053 in ?? ()
(gdb) si
=> 0xf0103e8a:      push     $0x0
0xf0103e8a      49      TRAPHANDLER_NOEC(th0, T_DIVIDE)
(gdb)
```

与上面类似，`TRAPHANDLER_NOEC` 宏定义一个相应的中断处理程序，即向栈里压入相关错误码和中断号。可以看出，除数为 0 对应的 IDT 号码为 0。

从 GDB 的分析结果得知，与上面不同的是，在 `trap` 函数中，由于 `tf_trapno` 与 `T_PGFLT`，`T_BRKPT`，`T_SYSCALL` 均不同，且 `tf->tf_cs` 与 `GD_KT` 不同，最终直接通过 `env_destroy(curenv)` 将当前进程销毁。

最终结果如下：

```

=> 0xf0103dbc <trap+149>:      mov     0x28(%esi),%eax
190      if (tf->tf_trapno == T_PGFLT) {
(gdb) si
=> 0xf0103dbf <trap+152>:      cmp     $0xe,%eax
0xf0103dbf 190      if (tf->tf_trapno == T_PGFLT) {
(gdb) si
=> 0xf0103dc2 <trap+155>:      jne     0xf0103dce <trap+167>
0xf0103dc2 190      if (tf->tf_trapno == T_PGFLT) {
(gdb) si
=> 0xf0103dce <trap+167>:      cmp     $0x3,%eax
194      if (tf->tf_trapno == T_BRKPT) {
(gdb) si
=> 0xf0103dd1 <trap+170>:      jne     0xf0103ddd <trap+182>
0xf0103dd1 194      if (tf->tf_trapno == T_BRKPT) {
(gdb) si
=> 0xf0103ddd <trap+182>:      cmp     $0x30,%eax
198      if (tf->tf_trapno == T_SYSCALL) {
(gdb) si
=> 0xf0103de0 <trap+185>:      jne     0xf0103e14 <trap+237>
0xf0103de0 198      if (tf->tf_trapno == T_SYSCALL) {
(gdb) si
=> 0xf0103e14 <trap+237>:      mov     %esi,(%esp)
206      print_trapframe(tf);
(gdb) n
=> 0xf0103e1c <trap+245>:      cmpw    $0x8,0x34(%esi)
207      if (tf->tf_cs == GD_KT)
(gdb) n
=> 0xf0103e3f <trap+280>:      mov     0xf017df88,%eax
210      env_destroy(curenv);

```

需要额外强调一下：divzero 由于先计算的 $1/0$ 并直接抛出异常，所以没有调用 `cprintf` 函数（从 `obj/user/divzero.asm` 可以看出，`cprintf` 的调用在 `0x00800060`，而程序在执行到 `0x00800053` 处就进入异常处理，并最终直接销毁该进程，没有返回到用户程序，即没有调用 `cprintf`），因而没有发生系统调用。

事实上，如果程序正常进行的话，流程会如下所示：

首先，在 `0x800060` 处调用 `cprintf` 函数；`cprintf` 函数在 `0x800186` 处调用 `vcprintf` 函数；`vcprintf` 函数在 `0x800166` 处调用 `sys_cputs` 函数；`sys_cputs` 函数在 `0x800ae8` 处产生 `int $0x30` 中断。通过在 `inc/trap.h` 中查找，`T_SYSCALL` 的宏为 `48`，即 `0x30`，所以此时产生系统调用，为中断。

系统调用 `syscall` 函数的输入为系统调用序号 `syscallno` 和 5 个相应寄存器的值，根据 `syscallno` 的值的不同分为 5 种情况，这里是调用 `sys_cputs` 函数，向屏幕输出字符。其他系统调用情况类似。输出后通过 `iret` 指令正常返回。