

# 南京农业大学

## 操作系统课程设计 技术问题分析报告



题目：操作系统课程设计  
技术问题分析报告

姓名：邱日

学院：信息科学与技术学院

专业：计算机科学与技术

班级：计科 151

学号：19215116

指导教师：姜海燕 职称：教授

2018 年 4 月 30 日

# 目 录

目录 .....	I
第 1 章 技术问题分析报告 .....	1
1.1 问题 01: 混合编程 .....	1
1.1.1 问题描述 .....	1
1.1.2 问题具体样式 .....	1
1.1.3 问题分析及解决方案 .....	1
1.1.4 问题解决后的结果样式 .....	4
1.2 问题 02: 指针使用 .....	5
1.2.1 问题描述 .....	5
1.2.2 问题具体样式 .....	5
1.2.3 问题分析及解决方案 .....	5
1.2.4 问题解决后的结果样式 .....	6
1.3 问题 03: 中断的 EOI 信号 .....	6
1.3.1 问题描述 .....	6
1.3.2 问题具体样式 .....	6
1.3.3 问题分析及解决方案 .....	6
1.3.4 问题解决后的结果样式 .....	7
1.4 问题 04: triple fault .....	7
1.4.1 问题描述 .....	7
1.4.2 问题具体样式 .....	7
1.4.3 问题分析及解决方案 .....	7
1.4.4 问题解决后的结果样式 .....	8
1.5 问题 05: 打开的文件不对应 .....	8
1.5.1 问题描述 .....	8
1.5.2 问题具体样式 .....	8
1.5.3 问题分析及解决方案 .....	8
1.5.4 问题解决后的结果样式 .....	8
1.6 问题 06: 同步性 .....	9
1.6.1 问题描述 .....	9
1.6.2 问题分析及解决方案 .....	9
1.7 问题 07: 编译选项 .....	9
1.7.1 问题描述 .....	9
1.7.2 问题具体样式 .....	9
1.7.3 问题分析及解决方案 .....	9
1.7.4 问题解决后的结果样式 .....	9
1.8 问题 08: 键盘支持 .....	9
1.8.1 问题描述 .....	9
1.8.2 问题具体样式 .....	10
1.8.3 问题分析及解决方案 .....	10
1.8.4 问题解决后的结果样式 .....	10
1.9 问题 09: 对大文件的支持 .....	10
1.9.1 问题描述 .....	10
1.9.2 问题具体样式 .....	10
1.9.3 问题分析及解决方案 .....	10
1.9.4 问题解决后的结果样式 .....	10
1.10 问题 10: 少括号 .....	11

## 第 1 章 技术问题分析报告

### 1.1 问题 01: 混合编程

#### 1.1.1 问题描述

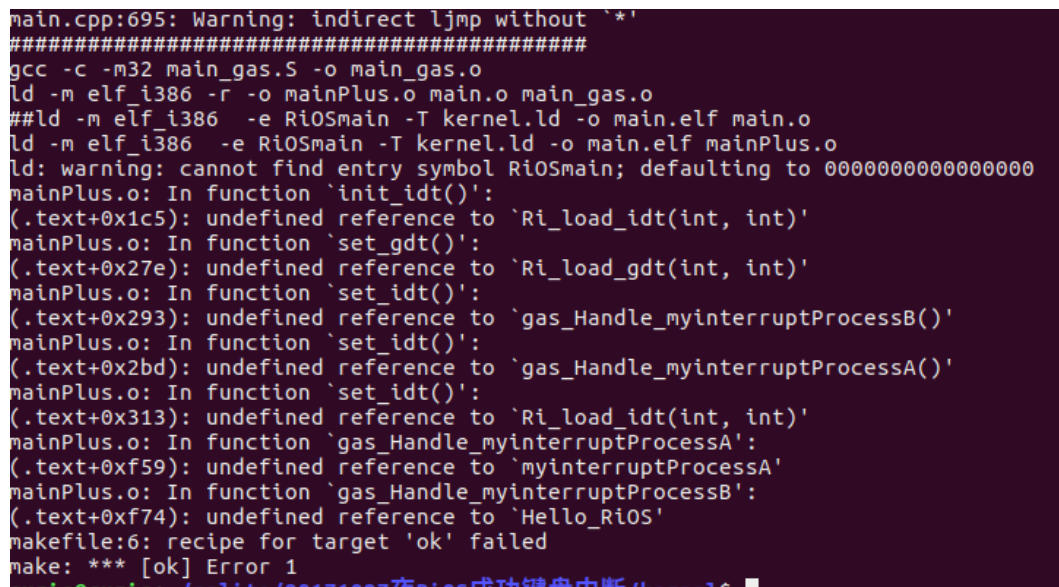
这个问题可以追溯到上学期做第一次操作系统实验的时候,当时找到了原因,但没有找到解决问题的方法,导致我一直只能用汇编和 C,无法使用 C++,现在这个问题已经解决.简单地说,就是不会汇编和 C++ 的相互调用.

#### 1.1.2 问题具体样式

问题的背景是我采用 C 语言开发本系统的过程中,代码没有问题,编译通过可以正常运行的情况下,我将 main.c 改为 main.cpp.当然 Makefile 里也作相应变化比如用 g++ 来编译 main.cpp 什么的.但是,却发现一些通过标号调用汇编代码的地方全不行了,报错 undefined.

#### 1.1.3 问题分析及解决方案

问题 01-1 如图 1.1 所示.



```
main.cpp:695: Warning: indirect ljmp without '*'
#####
gcc -c -m32 main_gas.S -o main_gas.o
ld -m elf_i386 -r -o mainPlus.o main.o main_gas.o
##ld -m elf_i386 -e RiOSmain -T kernel.ld -o main.elf main.o
ld -m elf_i386 -e RiOSmain -T kernel.ld -o main.elf mainPlus.o
ld: warning: cannot find entry symbol RiOSmain; defaulting to 0000000000000000
mainPlus.o: In function `init_idt()':
(.text+0x1c5): undefined reference to `Ri_load_idt(int, int)'
mainPlus.o: In function `set_gdt()':
(.text+0x27e): undefined reference to `Ri_load_gdt(int, int)'
mainPlus.o: In function `set_idt()':
(.text+0x293): undefined reference to `gas_Handle_myinterruptProcessB()'
mainPlus.o: In function `set_idt()':
(.text+0x2bd): undefined reference to `gas_Handle_myinterruptProcessA()'
mainPlus.o: In function `set_idt()':
(.text+0x313): undefined reference to `Ri_load_idt(int, int)'
mainPlus.o: In function `gas_Handle_myinterruptProcessA':
(.text+0xf59): undefined reference to `myinterruptProcessA'
mainPlus.o: In function `gas_Handle_myinterruptProcessB':
(.text+0xf74): undefined reference to `Hello_RiOS'
makefile:6: recipe for target 'ok' failed
make: *** [ok] Error 1
```

图 1.1 问题 01-1

一直找不到原因,我将编译后的机器码反汇编成.asm 文件找原因,如图 1.2 所示.

Linux 命令: objdump -S -D main.elf >main.asm

之前编译正常.c 文件时的编译结果如图 1.2 所示.

可以看到,我的函数名 RiOSmain 和 init\_8259 在编译之后是保留原来的字的.

Linux 命令:

g++ -nostdinc -I. -fpermissive -fno-stack-protector -c main.cpp -m32 -o main.o

objdump -S main.o >main.asm

```

main.elf:      file format elf32-i386

Disassembly of section .text:

00000000 <RiOSmain>:
  0:      55                      push    %ebp
  1:      89 e5                  mov     %esp,%ebp
  3:      83 ec 18              sub     $0x18,%esp
  6:      e8 ff 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
  b:      05 b5 24 00 00       add     $0x24b5,%eax
10:      e8 18 01 00 00       call    12d <init_gdt>
15:      e8 d7 01 00 00       call    1f1 <set_gdt>
1a:      e8 63 01 00 00       call    182 <init_idt>
1f:      e8 63 02 00 00       call    287 <set_idt>
24:      e8 19 00 00 00       call    42 <init_8259>
29:      fb                    sti
2a:      e8 a7 00 00 00       call    d6 <init_8253>
2f:      e8 d3 00 00 00       call    107 <KeyboardMouseintEnable>
34:      e8 e3 06 00 00       call    71c <console_cls>
39:      c7 45 f4 00 00 00 00 movl    $0x0,-0xc(%ebp)
40:      eb fe                  jmp     40 <RiOSmain+0x40>

00000042 <init_8259>:
42:      55                      push    %ebp
43:      89 e5                  mov     %esp,%ebp
45:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
4a:      05 76 24 00 00       add     $0x2476,%eax
4f:      b8 ff 00 00 00       mov     $0xff,%eax
54:      ba 21 00 00 00       mov     $0x21,%edx
59:      ee                      out     %al,(%dx)
5a:      b8 ff 00 00 00       mov     $0xff,%eax
5f:      ba a1 00 00 00       mov     $0xa1,%edx
64:      ee                      out     %al,(%dx)
65:      b8 11 00 00 00       mov     $0x11,%eax
6a:      ba 20 00 00 00       mov     $0x20,%edx
6f:      ee                      out     %al,(%dx)
70:      b8 20 00 00 00       mov     $0x20,%eax
75:      ba 21 00 00 00       mov     $0x21,%edx
7a:      ee                      out     %al,(%dx)
7b:      55                      push    %ebp
7c:      89 e5                  mov     %esp,%ebp
7e:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
83:      05 76 24 00 00       add     $0x2476,%eax
88:      b8 ff 00 00 00       mov     $0xff,%eax
8d:      ba 21 00 00 00       mov     $0x21,%edx
92:      ee                      out     %al,(%dx)
93:      55                      push    %ebp
94:      89 e5                  mov     %esp,%ebp
96:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
9b:      05 76 24 00 00       add     $0x2476,%eax
a0:      b8 ff 00 00 00       mov     $0xff,%eax
a5:      ba 21 00 00 00       mov     $0x21,%edx
aa:      ee                      out     %al,(%dx)
ab:      55                      push    %ebp
ac:      89 e5                  mov     %esp,%ebp
ae:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
b3:      05 76 24 00 00       add     $0x2476,%eax
b8:      b8 ff 00 00 00       mov     $0xff,%eax
bd:      ba 21 00 00 00       mov     $0x21,%edx
c2:      ee                      out     %al,(%dx)
c3:      55                      push    %ebp
c4:      89 e5                  mov     %esp,%ebp
c6:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
cb:      05 76 24 00 00       add     $0x2476,%eax
d0:      b8 ff 00 00 00       mov     $0xff,%eax
d5:      ba 21 00 00 00       mov     $0x21,%edx
da:      ee                      out     %al,(%dx)
db:      55                      push    %ebp
dc:      89 e5                  mov     %esp,%ebp
de:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
e3:      05 76 24 00 00       add     $0x2476,%eax
e8:      b8 ff 00 00 00       mov     $0xff,%eax
ed:      ba 21 00 00 00       mov     $0x21,%edx
f2:      ee                      out     %al,(%dx)
f3:      55                      push    %ebp
f4:      89 e5                  mov     %esp,%ebp
f6:      e8 c0 0f 00 00       call    100a <__x86.get_pc_thunk.ax>
fb:      05 76 24 00 00       add     $0x2476,%eax

```

图 1.2 编译.c 文件时的结果

```

main.o:      file format elf32-i386

Disassembly of section .text:

00000000 <_Z8RiOSmainv>:
  0:  55                push    %ebp
  1:  89 e5             mov     %esp,%ebp
  3:  83 ec 18          sub     $0x18,%esp
  6:  e8 fc ff ff ff    call    7 <_Z8RiOSmainv+0x7>
  b:  e8 fc ff ff ff    call    c <_Z8RiOSmainv+0xc>
 10:  e8 fc ff ff ff    call    11 <_Z8RiOSmainv+0x11>
 15:  e8 fc ff ff ff    call    16 <_Z8RiOSmainv+0x16>
 1a:  e8 fc ff ff ff    call    1b <_Z8RiOSmainv+0x1b>
 1f:  fb               sti
 20:  e8 fc ff ff ff    call    21 <_Z8RiOSmainv+0x21>
 25:  e8 fc ff ff ff    call    26 <_Z8RiOSmainv+0x26>
 2a:  c7 45 f4 00 00 00 00 movl    $0x0,-0xc(%ebp)
 31:  eb fe             jmp     31 <_Z8RiOSmainv+0x31>

00000033 <_Z9init_8259v>:
 33:  55                push    %ebp
 34:  89 e5             mov     %esp,%ebp
 36:  b8 ff 00 00 00    mov     $0xff,%eax
 3b:  ba 21 00 00 00    mov     $0x21,%edx
 40:  ee               out     %al,(%dx)
 41:  b8 ff 00 00 00    mov     $0xff,%eax
 46:  ba a1 00 00 00    mov     $0xa1,%edx
 4b:  ee               out     %al,(%dx)
 4c:  b8 11 00 00 00    mov     $0x11,%eax
 51:  ba 20 00 00 00    mov     $0x20,%edx
 56:  ee               out     %al,(%dx)
 57:  b8 20 00 00 00    mov     $0x20,%eax
 5c:  ba 21 00 00 00    mov     $0x21,%edx
 61:  ee               out     %al,(%dx)
 62:  b8 04 00 00 00    mov     $0x4,%eax
 67:  ba 21 00 00 00    mov     $0x21,%edx
 6c:  ee               out     %al,(%dx)
 6d:  b8 04 00 00 00    mov     $0x4,%eax

```

图 1.3 编译同样内容的.cpp 文件时的结果

此时我惊讶地发现，在 gcc 或 g++ 对 .cpp 文件编译时，原来 RiOSmain 的函数名变成了 Z 数字 RiOSmainv，即 gcc 是这样处理的函数到标号的映射是标号：Z 数字函数名 v，加上了前缀后缀，这样根据名字去链接必然失败。

解决方法是将 C++ 源程序保存为 .cc 文件，在头文件的把函数声明包在 extern "C" 之中，这样 C++ 的函数编译后就能和 C 编译后的函数兼容。

#### 1.1.4 问题解决后的结果样式

```
/*this is hello.h*/
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
void hello_cplusplus();
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

extern "C" 的意思，是让 C++ 编译器（不是 C 编译器，而且是编译阶段，不是链接阶段）在编译 C++ 代码时，为被 extern "C" 所修饰的函数在符号表中按 C 语言方式产生符号名（比如前面的 add），而不是按 C++ 那样的增加了参数类型和数目信息的名称（\_Z3addii）。

展开来细说，就是：如果是 C 调用 C++ 函数，在 C++ 一侧对函数声明加了 extern "C" 后符号表内就是 add 这样的名称，C 程序就能正常找到 add 来调用；如果是 C++ 调用 C 函数，在 C++ 一侧在声明这个外部函数时，加上 extern "C" 后，C++ 产生的 obj 文件符号表内也就是标记为它需要一个名为 add 的外部函数，这样配合 C 库，就一切都好。

特地写了一个简单的程序来验证一下，加深印象，如图 1.4 所示。

```
#include <stdio.h>
int main()
{
    int a[100]={0};
    for(int i=0;i<100;i++)a[i]=i;
    int *p=(int *)&a;
    int *q=(int *)&a;
    printf("X  &p:%d,&q:%d\n", (int)&p, (int)&q);
    printf("OK p:%d,q:%d\n", (int)p, (int)q);
    printf("p[13]%d,q[13]:%d\n", p[13], q[13]);

    int *pointer =(int *)&a; /* 指针指向数组的方法 */
```

```
#include <stdio.h>
int main()
{
    int a[100]={0};
    for(int i=0;i<100;i++)a[i]=i;
    int *p=(int *)&a;
    int *q=(int *)&a;
    printf("X  &p:%d,&q:%d\n",(int)&p,(int)&q);
    printf("OK  p:%d,q:%d\n",(int)p,(int)q);
    printf("p[13]%d,q[13]:%d\n",p[13],q[13]);

    int *pointer =(int *)&a; /*指针指向数组的方法*/
    printf("pointer[13]:%d\n",pointer[13]);
    return 0;
}
```

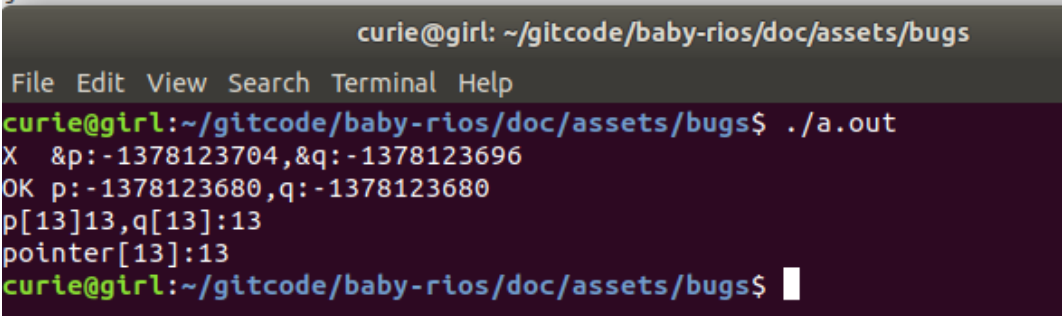


图 1.4 问题 02-1

```
printf("pointer[13]:%d\n",pointer[13]);
return 0;
}
```

## 1.2 问题 02: 指针使用

### 1.2.1 问题描述

指针使用错误.

### 1.2.2 问题具体样式

代码见下.

联合体相当于是个实体, 而数组 `int a[10]` 中的 `a` 就是个指针, 若在指针前面, 再加上取地址符 `&` 就成了指针的地址, 在联合体前面加取地址符就是指向联合体的指针. 我之前错误地在指针前面加了取地址符, 导致间接寻址都无效.

### 1.2.3 问题分析及解决方案

/\* 这里是修改好后的正确写法 \*/

```
u8 two_sectors[1024]={0};
```

```
IDE_read_sector((void *)two_sectors, DATA_BLK_NR_TO_SECTOR_NR(p_ft->f_inode->i
```

/\* 之前这里错误地写成了,IDE\_read\_sector((void \*)&two\_sectors, 多了个 & 造成大错 \*/

```
union free_space_grouping_head g_head;
u8 * p1 = (u8 *)&g_head ;IDE_write_sector((void *)p1,sector_num );
```

#### 1.2.4 问题解决后的结果样式

正常编译运行.

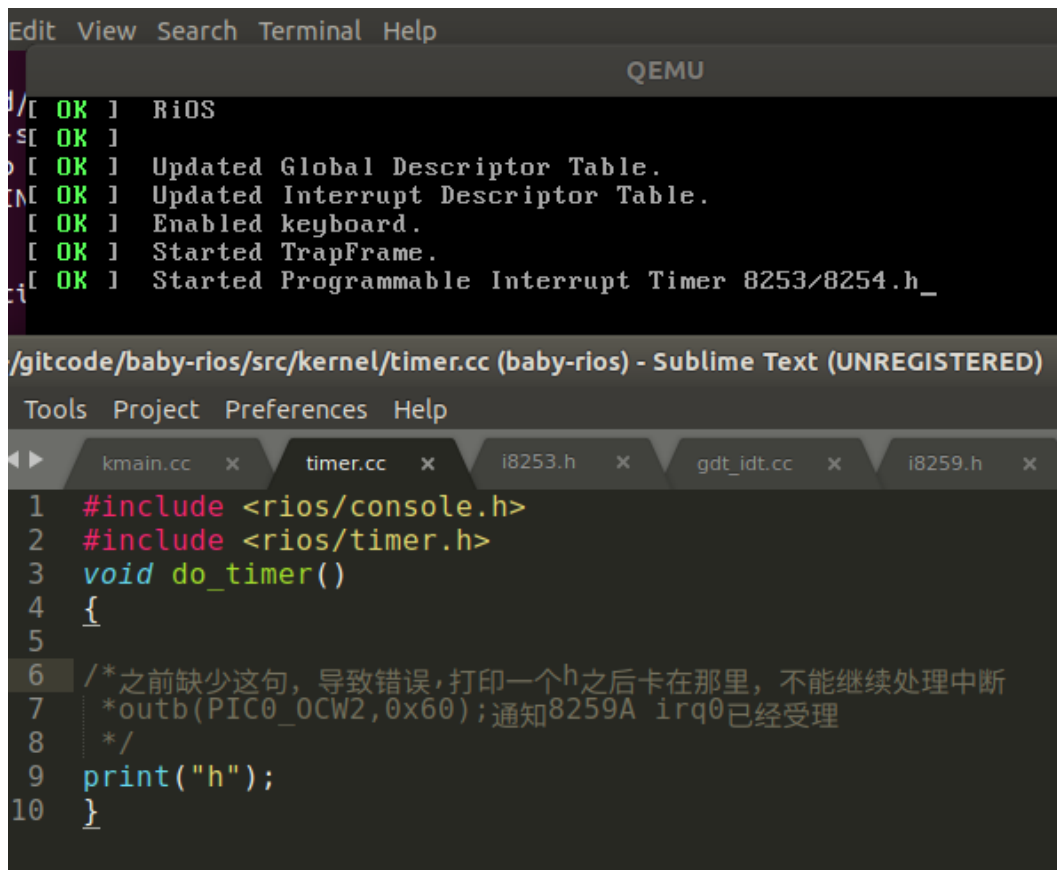
### 1.3 问题 03: 中断的 EOI 信号

#### 1.3.1 问题描述

中断只能执行一次.

#### 1.3.2 问题具体样式

比如键盘中断, 按下一个键, 可以在屏幕上显示相应字符, 但是再按键盘一点也没有反应. 如图 1.5 所示.



The image shows a QEMU terminal window at the top with the following output:

```
QEMU
d/[ OK ] RIOS
s/[ OK ]
b/[ OK ] Updated Global Descriptor Table.
M/[ OK ] Updated Interrupt Descriptor Table.
/[ OK ] Enabled keyboard.
/[ OK ] Started TrapFrame.
t/[ OK ] Started Programmable Interrupt Timer 8253/8254.h_
```

Below the terminal is a Sublime Text editor window titled "/gitcode/baby-rios/src/kernel/timer.cc (baby-rios) - Sublime Text (UNREGISTERED)". The editor shows the following code:

```
Tools Project Preferences Help
kmain.cc x timer.cc x i8253.h x gdt_idt.cc x i8259.h x
1 #include <rios/console.h>
2 #include <rios/timer.h>
3 void do_timer()
4 {
5
6 /*之前缺少这句, 导致错误, 打印一个h之后卡在那里, 不能继续处理中断
7 *outb(PIC0_OCW2,0x60);通知8259A irq0已经受理
8 */
9 print("h");
10 }
```

图 1.5 问题 03-1

#### 1.3.3 问题分析及解决方案

在中断处理函数中通过 in out 指令告知 8259A 中断已经被处理, 不然将会一直卡死在那里, 不会受理下一次中断.



解决方案：方法一是给 8259A 发 EOI 信号，即为 `outb(PIC0_OCW2,0x60)`，即 `outb(0x20,0x60)`；方法二是采用自动 EOI 方式，目前 RiOS 采用后者，一下这两句分别给主片和从片设为自动 EOI 模式。

```
outb_wait(0x20 + 1, 0x3); // Auto EOI in 8086/88 mode
outb_wait(0xa0 + 1, 0x3); // Auto EOI in 8086/88 mode
```

#### 1.3.4 问题解决后的结果样式

正常响应中断，不会按下一个键就卡住了。

### 1.4 问题 04:triple fault

#### 1.4.1 问题描述

这个问题可能是'triple fault'。

#### 1.4.2 问题具体样式

早期写内核时遇到一个 bug，用 grub 引导过去之后一按键盘就重启，一按找不到原因。

#### 1.4.3 问题分析及解决方案

初步猜测两个原因一是在 gdt 没有正确设置的情况下去写中断处理的代码，二可能是没有把所有陷阱门搞个处理函数。

在网上找到大致符合我症状的描述 (<http://www.osdever.net/bkerndev/Docs/gdt.htm>)。

Note that GRUB already installs a GDT for you, but if we overwrite the area of memory that GRUB was loaded to, we will trash the GDT and this will cause what is called a 'triple fault'. In short, it'll reset the machine. What we should do to prevent that problem is to set up our own GDT in a place in memory that we know and can access. This involves building our own GDT, telling the processor where it is, and finally loading the processor's CS, DS, ES, FS, and GS registers with our new entries. The CS register is also known as the Code Segment. The Code Segment tells the processor which offset into the GDT that it will find the access privileges in which to execute the current code. The DS register is the same idea, but it's not for code, it's the Data segment and defines the access privileges for the current data. ES, FS, and GS are simply alternate DS registers, and are not important to us.

我们利用 GRUB 进入保护模式，这里 GRUB 应该给我们默认设置了 GDT，但是如果我们把 gdt 的地方覆盖了，将会引发异常，具体叫做'triple fault'，这或许就是原因。

想到这里，我觉得在设置 IDT 之前，我们应当自己先把 GDT 设置好，GRUB 默认设的可能会覆盖，因此这是有必要的。

#### 1.4.4 问题解决后的结果样式

当我先设置 GDT 再设置 IDT 之后, 不再出现一按键盘就重启的现象, 一切正常了.

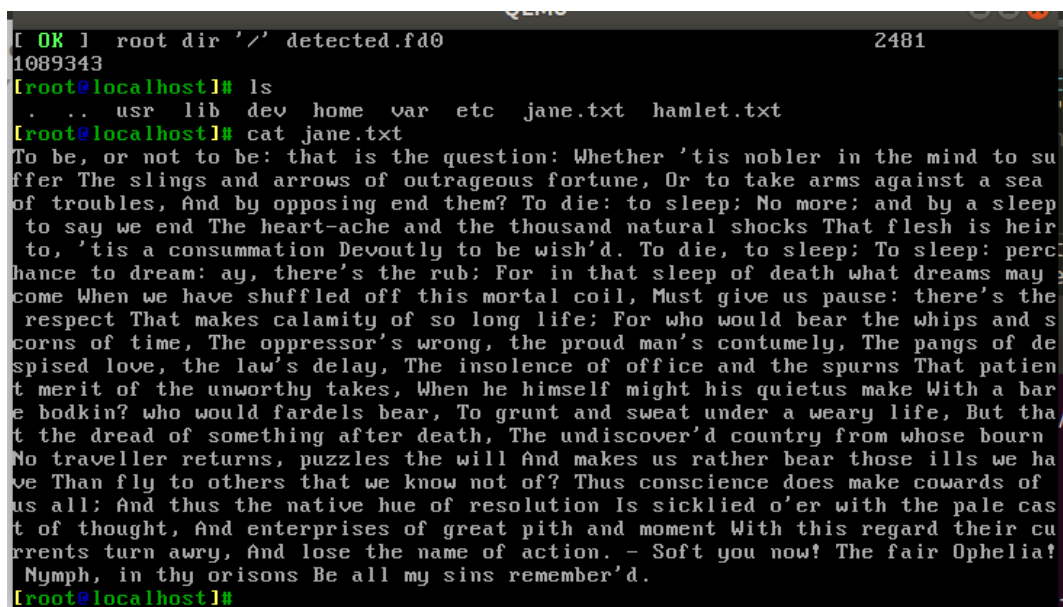
### 1.5 问题 05: 打开的文件不对应

#### 1.5.1 问题描述

. open 函数不能正确打开文件.

#### 1.5.2 问题具体样式

通过 open 返回的 fd 对应了其他文件, 错乱了. 如图 1.6 所示. 本来要显示的是大文件《简爱》(jane.txt), 然而打印出来的是 hamlet.txt 这么一个小段的话.



```
[ OK ] root dir '/' detected.fid0 2481
1089343
[root@localhost]# ls
. .. usr lib dev home var etc jane.txt hamlet.txt
[root@localhost]# cat jane.txt
To be, or not to be: that is the question: Whether 'tis nobler in the mind to su
ffer The slings and arrows of outrageous fortune, Or to take arms against a sea
of troubles, And by opposing end them? To die: to sleep; No more; and by a sleep
to say we end The heart-ache and the thousand natural shocks That flesh is heir
to, 'tis a consummation Devoutly to be wish'd. To die, to sleep; To sleep: perc
hance to dream: ay, there's the rub; For in that sleep of death what dreams may
come When we have shuffled off this mortal coil, Must give us pause: there's the
respect That makes calamity of so long life; For who would bear the whips and s
corns of time, The oppressor's wrong, the proud man's contumely, The pangs of de
spised love, the law's delay, The insolence of office and the spurns That patien
t merit of the unworthy takes, When he himself might his quietus make With a bar
e bodkin? who would fardels bear, To grunt and sweat under a weary life, But tha
t the dread of something after death, The undiscover'd country from whose bourn
No traveller returns, puzzles the will And makes us rather bear those ills we ha
ve Than fly to others that we know not of? Thus conscience does make cowards of
us all; And thus the native hue of resolution Is sicklied o'er with the pale cas
t of thought, And enterprises of great pith and moment With this regard their cu
rrents turn awry, And lose the name of action. - Soft you now! The fair Ophelia!
Nymph, in thy orisons Be all my sins remember'd.
[root@localhost]#
```

图 1.6 问题 05-1

#### 1.5.3 问题分析及解决方案

文件系统中, 我们三个表, 这三个表联系紧密, 他们之间的关系类似于索引, 使用时要确保这种索引关系要时刻保持, 不然这个链就断了, 之前没有 current->filp[fd]->f\_inode 指向活动 inode 表, 导致虽然已经正确写入磁盘, 活动 inode 表里也有, 但是甲 => 乙 => 丙两个指针中有一个没有指向正确的地方, 这就导致最终指不到正确的 inode, 造成错误.

错误原因在于在 open 函数中我在最后漏写了这一句:

```
current->filp[fd]->f_inode = &active_inode_table.inode_table[active_inode_table_nr];
```

#### 1.5.4 问题解决后的结果样式

正常.

下面几个问题都比较小，比较容易解决，有些就合起来描述了。

## 1.6 问题 06: 同步性

### 1.6.1 问题描述

close 函数也存在和 bug09 中 open 函数类似的问题, 那就是没有保持那三个表的同步性.

### 1.6.2 问题分析及解决方案

```
/* 更正时, 在 close 新添加的语句 */  
for(j=0;j<MAX_ACTIVE_INODE;j++)  
    if(active_inode_table.inode_table[j].i_ino==filp->f_inode->i_ino)  
        break;  
memset(&active_inode_table.inode_table[j],0x00,sizeof(m_inode));
```

这样在 close 时也把活动 inode 清理, 这就不会出差错了.

## 1.7 问题 07: 编译选项

### 1.7.1 问题描述

编译报错. 如图 1.7 所示.

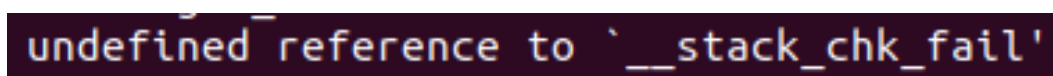


图 1.7 问题 07-1

### 1.7.2 问题具体样式

undefined reference to `\_\_stack\_chk\_fail`

### 1.7.3 问题分析及解决方案

解决方法: 在编译时, 在 CFLAGS 后面加上 -fno-stack-protector  
-fno-stack-protector

### 1.7.4 问题解决后的结果样式

正常编译.

## 1.8 问题 08: 键盘支持

### 1.8.1 问题描述

在 qemu 虚拟机中没问题, 按一个键打印一个字符, 但到了实体机按一次打印两个相同字符

### 1.8.2 问题具体样式

原理：按一个键打印两个字符，因为一次是 Make code，一次是 Break Code. 键盘的扫描码 Scan Code，通码 Make code，断码 Break Code 用户按键盘上的字母，硬件底层会产生对应的 Scan Code，而且是按下那一刻产生一个通码 Make code，释放的时候产生一个断码 Break code。即你从按下一个键盘上的字母，到手松开，实际上对应着一个通码 Make Code 和一个断码 Break Code，两者概念上都属于扫描码 Scan Code。

### 1.8.3 问题分析及解决方案

解决方法：判断键盘状态

### 1.8.4 问题解决后的结果样式

正常.

## 1.9 问题 09: 对大文件的支持

### 1.9.1 问题描述

支持不了大文件

### 1.9.2 问题具体样式

大一点的文件只能找到它的前面小部分, 后面就找不到、显示不出来了.

### 1.9.3 问题分析及解决方案

i\_size 当时设的值太小了, 当时没有考虑到后来文件变得很大的时候.

当时开始设的文件大小 i\_size 类型是 u8, 也就是 8 位二进制数, 但可以看到我的 jane.txt 内容相当多, 是简爱一本书的内容, 大小有 1089343, 用十六进制表示是 0x109f3f, 明显 8 位二进制数早已不能表示它, 发生溢出. 当我的文件大小没有超过 8 位时, cat jane.txt 和 cat hamlet.txt 都能正常工作, 但是此时文件大小比较大, i\_size 这一项错误了, 导致显示 jane.txt 时错误地显示了 hamlet.txt 的内容. 故需要修改 i\_size 类型, 使其为 32 位, 同时, 我的 inode 大小是固定好的, 还要调整其他项, 使 inode 大小不发生改变.

这只是其中原因之一, 之二是我还没有支持二级间地址访问, 所以一个文件最大容量受限, 容纳几 kB 还行, 但尚不能容纳几 MB 大小的单个文件.

### 1.9.4 问题解决后的结果样式

完善代码后支持了大文件.

### 1.10 问题 10: 少括号

这是个非常简单的错误, 但找起来却不轻松我本要一个数除以  $(80*25)$ , 结果错误地写成 `int times = len/80 * 25;` 这样实际上变成了  $(len/80)25$ , 导致错误. 应该写成 `int times = len/(80*25);`