

操作系统 LAB2

# 给 OS 感受和反应的能力

Report

姓名： 肖丹妮

学号： 201220199

邮箱： [3165372798@qq.com](mailto:3165372798@qq.com)

## 目录

|                            |   |
|----------------------------|---|
| 1. 23 个 exercise.....      | 1 |
| 2. 6 个 task.....           | 6 |
| 3. 2 个 challenge.....      | 7 |
| challenge1 challenge2(第一问) |   |
| 4. 2 个 conclusion.....     | 9 |
| conclusion1 conclusion2    |   |

## 一、Exercises

### exercise1:

如果把连续的信息存在同一柱面号同一扇区号的连续的盘面上，在每次读完一个扇区，要读取下一个盘面的同一扇区号的扇区之前，都需要等待该扇区转到读写磁头下面。并且盘面的转动是机械动作，速度较慢。在最差的情况下可能磁头恰好要等待一整圈，平均情况下每次磁头都需要等待半圈。

但是将连续的信息存在同一柱面号同一磁道的连续的盘面上，就可以大大减少磁头等待的时间，提升读写效率。

### exercise2

设每个柱面有  $h$  个磁道，每个磁道有  $s$  个扇区，那么对应 LBA 表示法的编号（块地址）是  $C \cdot h \cdot s + H \cdot s + S - 1$ 。

### exercise3

使用 `readelf -l xxx` 可以查看程序头表，如下所示

```
danni@danni-VirtualBox:~/down$ readelf -l lb1
Elf 文件类型为 EXEC (可执行文件)
Entry point 0x8049060
There are 11 program headers, starting at offset 52

程序头:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
INTERP         0x000194 0x08048194 0x08048194 0x00013 0x00013 R   0x1
               [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD           0x000000 0x08048000 0x08048000 0x002e8 0x002e8 R   0x1000
LOAD           0x001000 0x08049000 0x08049000 0x00264 0x00264 R E 0x1000
LOAD           0x002000 0x0804a000 0x0804a000 0x00254 0x00254 R   0x1000
LOAD           0x002f0c 0x0804bf0c 0x0804bf0c 0x00264 0x00268 RW 0x1000
DYNAMIC        0x002f14 0x0804bf14 0x0804bf14 0x000e8 0x000e8 RW   0x4
NOTE           0x0001a8 0x080481a8 0x080481a8 0x00044 0x00044 R   0x4
GNU_EH_FRAME   0x0020e0 0x0804a0e0 0x0804a0e0 0x0004c 0x0004c R   0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW   0x10
GNU_RELRO      0x002f0c 0x0804bf0c 0x0804bf0c 0x000f4 0x000f4 R   0x1

Section to Segment mapping:
段节...
00
01 .interp
02 .interp.note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
03 .init .plt .text .fini
04 .rodata .eh_frame_hdr .eh_frame
05 .init_array .fini_array .dynamic .got .got.plt .data .bss
06 .dynamic
07 .note.gnu.build-id .note.ABI-tag
08 .eh_frame_hdr
09
10 .init_array .fini_array .dynamic .got
```

说明:

共有 11 个表项，其中有四个是可装入段（**Type=LOAD**）对应的表项信息。以第一个可装入段（程序头表中第三个表项）为例，对应可执行目标文件中第 `0x000000~0x002e8` 字节的内容，被映射到从虚拟地址 `0x8048000` 开始的长度为 `0x002e8` 字节的区域，按 `0x1000=212=4KB` 对齐，具有只读权限（**Flg=R**），是一个只读代码段。

### exercise4

第一步：加载 OS 部分——在 `bootloader` 文件的 `boot.c` 中由 `bootMain()`

完成；

第二步：进行系统的各种初始化工作——在 `kernel/main.c` 的 `kEntry()` 中被调用，对应函数的具体实现所在位置如下所示：

- i. `initSerial`——`kernel/kernel/serial.c`
- ii. `initIdt`——`kernel/kernel/idt.c`
- iii. `initIntr`——`kernel/kernel/i8259.c`
- iv. `initSeg`——`kernel/kernel/kvm.c`
- v. `initVga`——`kernel/kernel/vga.c`
- vi. `initKeyTable`——`kernel/kernel/keyboard.c`
- vii. `loadUMain`——`kernel/kernel/kvm.c`

第三步：进入用户空间进行输出

- viii. `enterUserSpace`——在 `kernel/kernel/kvm.c` 中实现，在 `kernel/kernel/kvm.c` 的 `loadUMain()` 中被调用，进入用户空间。
- ix. 调用库函数，输出各种内容——`app/main.c`

## exercise5

中断分为外部中断和内中断。

外部中断可分为“可屏蔽中断”和“不可屏蔽中断”，“可屏蔽中断”是通过可屏蔽中断请求线 `INTR` 向 CPU 进行请求的中断，主要来自 I/O 设备的中断请求；“不可屏蔽中断”是通过不可屏蔽中断请求线 `NMI` 向 CPU 发出的中断请求，如电源掉电、硬件故障等。

内中断包括“软中断”和“异常”。“软中断”是执行 `INT n` 指令引起的；“异常”是指软件运行过程中 CPU 检测到的和指令执行相关的事件，如除零错误、段错误等。

## exercise6

IRQ 和中断号不一样。

IRQ 指中断请求（IRQ 号是中断请求号）。在 Linux 中，每一个能够发送中断信号的硬件设备控制都有一根输出线，与中断控制器 8259A 的输入引脚相连。如果硬件设备想要向 CPU 发送中断信号，需要向 8259A 申请一条可用的中断请求线（申请一个 IRQ 号）。

中断号指中断向量号，是 8 位的无符号整数，对应 256 种中断信号源。

默认情况下，IRQ 号对应的中断向量号由 BIOS 初始化，IRQ0-IRQ7 被设置为对应的向量号 0x08-0x0F。由于中断控制器是可编程的，因此 IRQ 和中断号之间的对应关系可以被修改。

## exercise7

从内核态返回用户态时，需要利用 `iret` 指令获得原来的 `EIP`、`CS` 和 `EFLAGS`，因此需要把 `EIP`、`CS` 和 `EFLAGS` 保存进内核态的内核堆栈。如果使用一些函数或者指令序列，这些函数或者指令序列容易被用户程序使用，使得安全性降低。

## exercise8

若选择将信息保存到一个固定的地方，当发生中断嵌套，比如要执行 `printf`，就会在处理中断之前将寄存器信息都保存到这个固定的地方，在 `printf` 的过程中又发生了除零异常，在处理除零之前会将 `printf` 过程的寄存器信息也保存到固定的地方，导致原来的状态信息被覆盖。

## exercise9

比较 `GDT[old_CS].DPL` 和 `GDT[CS].DPL`，如果 `GDT[old_CS].DPL < GDT[CS].DPL`，说明在用户态发生中断，中断处理在内核态。中断处理前，进行硬件堆栈切换，将 `old_SS` 和 `old_EIP` push 进栈；中断处理完之后，需要进行硬件堆栈切换，将原来 push 进堆栈的 `old_SS` 和 `old_EIP` 释放到 `ESP` 和 `SS` 中，切换到用户态的用户栈。

## exercise10

会。

如果不保存和恢复中断现场，可能导致处理完中断之后回到用户态时用户程序的执行受到影响；另一方面，若不恢复成原程序的信息，当返回用户态，用户程序可以直接根据寄存器获得内核的信息，对于内核来说安全性不够。

## exercise11

一方面将软中断的实现设计成和硬件中断类似的流程，提供了统一的中断响应和处理接口，处理更加方便有效，也避免了用户使用资源时出错，提高了效率；另一方面，这样可以对系统进行“保护”，内核可以检查是否发生 `#GP` 异常，保证系统的安全性。

## exercise12

- (1) 如果不设置 `esp`，`esp` 就是一个未知的值，可能会发生 `#GP` 异常、破坏用户空间等等问题。
- (2) `esp` 不能设置在用户和内核的代码段和数据段所在地址范围内，比如设置在 `0x100900`，因为访问了内核空间所以会发生 `#GP` 异常。

```
movl $0x100900,%eax
# movl $0xffffffff,%eax      # setting esp
movl %eax,%esp
```

```
void GProtectFaultHandle(struct TrapFrame *tf){
    assert(0);
    return;
}
```

```
Assertion failed: kernel/irqHandle.c:43
```

## exercise13

正确的装载 ELF 可执行文件的过程是：

1. 先通过 `readSect` 函数把 ELF 文件整体读入固定的位置。
2. 找到 ELF 可执行文件的 ELF 头。
3. 通过 ELF 可执行文件的头，找到程序头表的位置，并获知表项的多少
4. `type` 为 `LOAD` 的段，需要被加载到内存里去。它相对于 ELF 起始位置的偏移量是 `off`，应当被加载到 `PhysAddr` 的物理内存中去。文件的大小是 `filesz`，在内存中占据的大小是 `memsz`。

框架代码已经实现了步骤 1，错误代码只正确实现了 2，但 3 和 4 均没有实现。

```
for (i = 0; i < 200 * 512; i++) {
    *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
}
```

这几句没有根据程序头表的内容进行装载，最终导致数据段的装载是错的。

## exercise14

- (1) `kMainEntry` 函数和 Kernel 的 `main.c` 里的 `kEntry` 的值相等。
- (2) `kMainEntry` 函数的参数和返回类型都是空类型，表示 `kMainEntry` 是一个指向函数入口的指针变量。`kMainEntry = (void (*)(void))(eh->entry);` 这一句将程序入口 `eh->entry` 强制类型转换为函数指针类型并赋给 `kMainEntry`，使得 `kMainEntry` 指向 `kernel/main.c` 的入口。  
由 `Makefile` 里的 `ld` 链接命令指定 `kEntry` 为程序入口，故两者值相同。

## exercise15

- (1) 都会跳到 `asmDoIrq`，在 `pushal` 和 `pushl %esp` 之后再跳到 `irqHandle`。
- (2) 因为中断的响应都涉及保护现场，转向中断事件处理程序执行，恢复现场这几个一致的步骤，因此可以将这几个步骤抽象成一个函数，各种中断的响应都可以统一执行这一个函数，方便且有效。

## exercise16

```
struct TrapFrame {
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    int32_t irq;
};
```

在 `push esp` 之前，先用 `pushl $xxx` 将 `irq` 压栈，再用 `pusha` 将通用寄存器都压栈，这些是之后 `irqHandle` 以及中断处理程序需要用到的参数，构成了 `TrapFrame` 结构体，`push esp` 将 `TrapFrame` 的首地址压栈，随后 `call irqHandle`，因此 `esp` 作为 `TrapFrame` 结构体的指针、`irqHandle` 的参数传给了 `irqHandle`。

## exercise17

如果 keyboard 中断出现嵌套，比如连续按键 'A' 'B'，'A' 引起的中断正在处理

的时候发生了'B'的中断，就会先处理'B'的中断，将'B'打印在显存上，然后返回处理'A'的程序，将'A'打印在显存上。屏幕上显示出来的不是正常的按键顺序 AB 而是倒序 BA，堆栈会先保存原来用户程序的现场，再保存 A 的现场，然后 A 的现场信息出栈，原来用户程序的现场信息出栈。

### exercise18

占 0x100000 字节=1MB

### exercise19

因为将字符打印到显存需要先获得字符，字符存储在用户数据段，但是此时程序位于内核态，令 `sel` 变量等于用户数据段的段选择子，将 `sel` 的值放进 `es` 寄存器，`str+i` 是字符串第  $(i+1)$  位字符的偏移量，用 `es:(str+i)` 就可找到在用户数据段的字符串。

### exercise20

- (1) `paraList` 初始值设置为 `&format`，即找到了 `printf` 参数表的首地址。
- (2) 数字 2 的地址是 `&format+sizeof(format)`
- (3) `index` 初始化为 0，

之后解析 `format` 遇到 % 的时候：

`index=index+1;`

`para = paraList + sizeof(format) * index;`

则 `para` 就是此时需要格式化打印的参数的地址。

### exercise21

`SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0`

这些参数在 `syscall()` 中都对应存进了 `eax`、`ecx`、`edx`、`ebx`、`esi`、`edi`，再通过 `pushal` 存进了 `TrapFrame` 结构体，由 `TrapFrame` 指针 `tf` 传给了 `irqHandle()`，后续再传给 `syscallHandle()`。

`SYS_WRITE` 是系统调用号，在 `syscallHandle()` 会用到，由 `SYS_WRITE` 知需调用 `syscallWrite()`；

`STD_OUT` 是文件文件描述符，表示写入的文件，在 `syscallWrite()` 会用到，由 `STD_OUT` 知接下来调用 `syscallPrint()`；

`(uint32_t)buffer` 是写入字符串的首地址，在 `syscallPrint()` 会用到；

`(uint32_t)count` 是写入字符串的长度，在 `syscallPrint()` 会用到；

后面的两个 0 在此处没有实际意义。

### exercise22

- (1) 串口的两个函数 `putChar`、`putStr` 是通过 `outByte` 写 I/O 端口，此处的 `getChar` 和 `getStr` 是分别调用 `syscallGetChar()` 和 `syscallGetStr()`，通过

等待外部中断键盘输入, 利用 `inByte` 读 I/O 端口, 将数据存进 `keyBuffer`, 通过 `keyBuffer` 获得数据。

- (2) `getChar` 和 `getStr` 可以在用户程序中被调用, 但是 `putChar` 和 `putStr` 只能在内核态被调用。

## exercise23

因为 `gdt` 初始化时就将用户代码段和用户数据段的基址设置成了 `0x200000`。

## 二、Tasks

### task1:

利用 `readSect()` 将 `kMain.elf` 从磁盘读到内存起始地址为 `0x200000` 的地方;

```
ELFHeader * eh = (ELFHeader *)elf;
ProgramHeader *ph = (ProgramHeader *) (elf + eh->phoff);
ProgramHeader *eph = (ProgramHeader *) (ph + eh->phnum);
```

`Eh` 指向 ELF 头, `ph` 指向第一个程序头表表项的开始, `eph` 指向程序头表最后一个表项的结束。

利用 `for` 循环对每一个程序头表表项进行判断, 如果 `type` 是可装载段, 就将 `filesz` 个字节装载到以 `ph->paddr` 为起始地址的地方, 如果 `ph->memsz > ph->filesz`, 说明有未初始化全局变量, 用 0 填充。

令 `kMainEntry = (void (*)(void))(((ELFHeader *)elf)->entry)`, 使 `kMainEntry` 指向 `kernel` 入口。调用 `kMainEntry` 从而跳到 `kernel` 入口执行。

### task2:

调用对应的初始化函数即可。

### task3:

- (1) 对于门描述符, 判断 `dpl` 是 `DPL_USER` 还是 `DPL_KERN`,  
    `if(dpl == DPL_USER)`  
        `ptr->segment = USEL(selector);`  
    `else`  
        `ptr->segment = KSEL(selector);`
- (2) 将 `keyboard` 中断设置成 `Intr`, `syscall` 也设置为 `Intr`, 其他均为 `Trap`;
- (3) `syscall` 从用户态陷入内核态, 故 `syscall` 对应的 `dpl` 是 `DPL_USER=3`, 其他的 `dpl` 均为 `DPL_KERN`。

### task4:

`irq = -1, break;`  
`irq = 0xd`, 对应中断处理程序是 `GProtectFaultHandle()`;



irq = 0x21, 对应中断处理程序是 KeyboardHandle();  
 irq = 0x80, 对应中断处理程序是 syscallHandle();

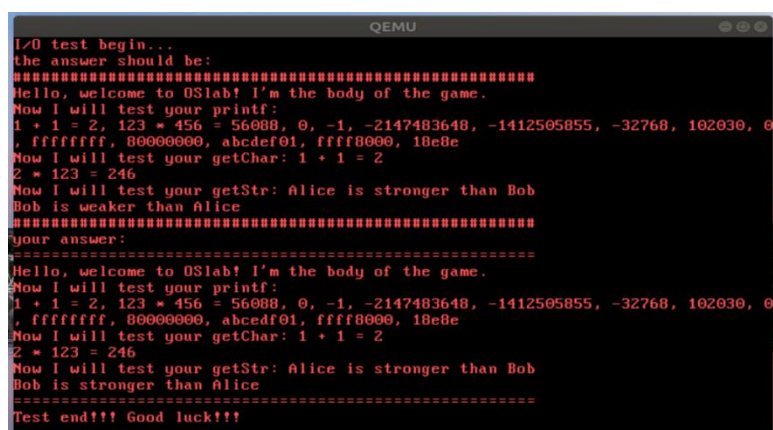
### task5:

- (1) getChar:调用 syscall(), 相关参数为 SYS\_READ, STD\_IN, 0, 0, 0, 0;
- (2) getStr: 调用 syscall(), 相关参数为 SYS\_READ, STD\_STR, (uint32\_t)str, (uint32\_t)size, 0, 0;
- (3) printf:借助状态机实现:  
 state=0: 可能会遇到合法的字面字符, 这时直接将字符存进 buffer;  
 或者遇到'%', 这时进入 state1, 并且 index 加一, 表示当前需要格式化的 printf 的参数向后移一个。  
 state=1: 首先令 para = paraList + sizeof(format) \* index;得到参数的地址, 再根据格式转换说明符进行分类讨论, 分别调用已经实现好的辅助函数。  
 另外, 每往 buffer 里面存一个字符都需要注意判断 count 是否已经到了 buffer 的最大长度 MAX\_BUFFER\_SIZE, 如果是, 就需要先调用 syscall 将当前 buffer 字符串输出, 令 count=0, 再继续。

### task6:

基本与加载 kernel 类似, 首先将磁盘第 201 块-400 块扇区的内容读到内存起始地址为 0x200000 的地方, 然后定义 ELF 头指针、程序头指针等等。  
 令 uint32\_t uMainEntry = ((ELFHeader \*)elf)->entry - 0x200000;使得程序入口偏移量为 0x0-0x200000, 对应 cs 为 USEL(SEG\_UCODE), 这样的物理地址就是 0x200000+(0x0-0x200000)=0x0。

结果如下所示:



```

I/O test begin...
the answer should be:
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
your answer:
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getChar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is stronger than Alice
Test end!!! Good luck!!!
    
```

## 三、Challenge

### challenge1:

错误代码从程序头表中的第一个表项开始, 连续装载了 200\*512 个字节大小的



文件内容。

```
oslab@oslab-VirtualBox:~/Lab2$ readelf -l kernel/kMain.elf
Elf file type is EXEC (Executable file)
Entry point 0x1000a8
There are 3 program headers, starting at offset 52

Program Headers:
Type           Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
LOAD           0x001000  0x00100000  0x00100000  0x01414 0x01414  R E  0x1000
LOAD           0x003000  0x00103000  0x00103000  0x00120 0x01f00  RW  0x1000
GNU_STACK      0x000000  0x00000000  0x00000000  0x00000 0x00000  RWE  0x10

Section to Segment mapping:
Segment Sections...
00      .text .rodata .eh_frame
01      .got.plt .data .bss
02
```

由 kernel 的程序头表知，前两个段是可装载段，需要装载到内存，错误代码将包括前两个段在内的内容全部从从 elf+offset 开始的地方平移到了从 elf 开始的地方，elf = 0x100000，只读代码段的起始 Physaddr=0x100000，只读代码段恰好移到了正常装载的目的地址处。对于可读可写数据段，虽然装载到了错误的地方，因为全局变量是在初始化之后才使用，不需要直接访问，因此即使写进错误的地址后续也可以正常访问。

## challenge2:

### (1) KeyboardHandle 函数处理键盘中断:

先用 getKeyCode() 从 I/O 端口获得输入字符的 uint32\_t code，再对 code 进行讨论:

是空格，在显存打印空格；是回车，将显存的当前行数加一（判断是否需要滚屏）；是其他正常字符，将 code 转为 char 型，调用 putchar() 在串口输出，用内联汇编在显存打印字符。asm volatile("movw %0, (%1)":"r"(data),"r"(pos+0xb8000));

### (2) syscallPrint 函数对应于“写”系统调用:

获取参数字符串首地址为 tf->eax，长度为 tf->ecx，将用户数据段选择子存进 es 寄存器，利用 es:(str+i) 获得字符地址，将字符依次打印到显存里地址为 pos+0xb8000 的地方，并完成光标的维护。

### (3) syscallGetChar 和 syscallGetStr 对应于“read”系统调用:

- a) syscallGetChar: 第一个 while 循环是，开中断，令 c 等于 keyBuffer 第一个字符，关中断，直到 keyBuffer 获得用户输入、c 不为 0 退出循环。将 c 从串口输出，利用 tf->eax=c; 将获得的字符传给 syscall、作为 syscall 的返回值。下一个循环等待按键，无论按什么键都结束程序。
- b) syscallGetStr: 初始化 keyBuffer，while 循环里面循环执行开中断，等待获得键盘输入，关中断的过程，直到键盘按键为换行或者已经获得了目标 size=tf->ebx 个字符，将用户数据段选择子存进 es 寄存器，利用 es:(str+k) 获得字符地址，将 keyBuffer 里的字符依次存进 es:(str+k)，并在最后一个字符的下一个地址单元存 0x00，表示字符的结束'\0'。

## 四、 Conclusion

### conclusion1:

syscall 中：先将各寄存器的值保存到变量 `eax`, `ecx`, `edx`, `ebx`, `esi`, `edi` 中，再将各个参数分别赋值给 `EAX`, `EBX`, `ECX`, `EDX`, `EDI`, `ESI`（约定将返回值放入 `EAX` 中），接着利用 `int $0x80` 指令陷入内核。

硬件：从 TSS 中将内核态对应的栈寄存器内容和栈指针装入 `SS` 和 `ESP`，将 `SS`、`ESP`、`EFLAGS`、`CS`、`EIP` 保存到内核栈中，从 IDT 中的第 0x80 个表项中取出门描述符，检查 `CPL` 和门描述符的 `DPL`，`CPL=DPL=3`，将段选择符装入 `CS`，偏移地址装入 `EIP`，`CS:EIP` 就是 `irqSyscall` 的第一条指令的地址，并修改 `EFLAGS` 的 `IF` 位为 0，跳转到 `irqSyscall`。

`irqSyscall`：压入 `error code`、中断向量号 0x80，跳转到 `asmDoIrq`；

`asmDoIrq`：将通用寄存器的值压栈，将 `esp` 压栈（传递 `TrapFrame` 结构体的首地址），跳转到 `irqHandle`；

`irqHandle`：根据 `irq=0x80`，调用 `syscallHandle()`；

`syscallHandle`：根据参数 `SYS_WRITE`，调用 `syscallWrite()`；

`syscallWrite`：根据参数 `STD_OUT`，调用 `syscallPrint()`；

`irqHandle` 执行结束，`TrapFrame` 结构体出栈，`error code` 出栈，使用 `iret` 指令恢复 CPU 的执行，返回 `syscall` 执行。

`syscall`：将 `eax` 中的返回值存进 `ret` 变量，回复 `EAX`, `EBX`, `ECX`, `EDX`, `EDX`, `EDI`, `ESI` 寄存器的值，重置 `DS` 寄存器的值，返回 `ret`。

### conclusion2:

CPU 被告知有一个中断向量号为 0xd 的中断到来，硬件查 IDT 表，从 IDT 中的第 0xd 个表项中取出门描述符，进行特权级检查，发现目标代码段的 `DPL = 0`，`GDT[old_CS].DPL = 3`，因此硬件进行堆栈切换，从 TSS 中将内核态对应的栈寄存器内容和栈指针装入 `SS` 和 `ESP`，将 `SS`、`ESP`、`EFLAGS`、`CS`、`EIP` 保存到内核栈中。将段选择符装入 `CS`，偏移地址装入 `EIP`，`CS:EIP` 就是 `irqGProtectFault` 的第一条指令的地址，跳转到 `irqGProtectFault`。

`irqGProtectFault`：将 `irq=0xd` 压栈，跳转到 `asmDoIrq`；

`asmDoIrq`：将通用寄存器的值压栈，将 `esp` 压栈（传递 `TrapFrame` 结构体的首地址），跳转到 `irqHandle`；

`irqHandle`：根据 `irq=0xd`，调用 `GProtectFaultHandle ()`；

`GProtectFaultHandle`：处理保护错误（本实验中是 `assert(0)` 使程序中断）；

`irqHandle` 执行结束，`TrapFrame` 结构体出栈，使用 `iret` 指令恢复 CPU 的执行，返回 `old_CS:old_EIP` 执行。