

16 级计科 7 班: 操作系统原理实验 #5

Due on Friday, April 13, 2018

凌应标 周一 9-10 节

颜彬

16337269

Content

	Page
1 实验目的	3
2 实验方案	3
2.0.1 中断调用原理	3
2.0.2 软硬中断简介	3
2.0.3 用户自定义中断	3
2.1 实验环境	4
2.1.1 系统与虚拟机	4
2.1.2 相关工具、指令	4
3 实验过程	4
3.0.1 GCC 语言拓展的运用	4
3.1 修改时钟中断	6
3.2 修改键盘中断	6
3.3 安装用户自定义中断	7
3.4 自定义的底层输出函数	7
4 实验结果	7
4.1 终端界面	7
4.2 用户程序	9
4.3 特色中断	10
5 实验总结	10
5.1 亮点介绍	10
5.1.1 极其详尽的文档	10
5.1.2 ASM 拓展语法	10
5.1.3 格式化 ls 输出	10
5.1.4 光标滚屏自主控制	11
5.2 更新细节	11
5.2.1 printf 进一步完善	11
5.2.2 目录树结构变更	11
5.3 心得体会	11
5.4 BUG 汇总	11
5.4.1 扇区加载错误	11
5.4.2 时钟中断未正确保存上下文	12
5.4.3 双次安装中断的错误	12
A 参考文献	13

1 实验目的

掌握 PC 微机的实模式硬件中断系统原理和中断服务程序设计方式, 实现对时钟、键盘/鼠标等硬件中断的简单服务处理程序编程和调试, 让你的原型操作系统在运行以前已有的用户程序时, 能对异步事件正确地捕捉和响应。

掌握操作系统的系统调用原理, 实现原型操作系统中的系统调用框架, 提供若干简单功能的系统调用。

学习掌握 C 语言库的实际方法, 为自己的原型操作系统配套一个 C 程序开发环境, 实现用自建的 C 语言开发简单的输入/输出的用户程序, 展示封装的系统调用。

2 实验方案

2.0.1 中断调用原理

在实模式下, 内存 0x0000 起始处维护着一张中断服务程序入口的表。每个表项站 4 字节, 共同代表服务程序的入口地址。其中的低 16 位代表代码段地址, 高 16 位代表段内偏移地址。

当调用中断时, 中断号代表着服务程序地址在地址表中的索引。所以当执行 `int id` 时, 会将 `0x0000 : id×4` 起始的 32 bits 作为入口地址。一般寄存器 `ah` 中的值会作为中断的“功能号”。其他寄存器的值依据文档, 相应地作为参数或作为返回值。

当实模式下触发中断时, 会首先将 `FLAGS` 寄存器压入栈中 (32 位模式下是 `EFLAGS`), 随后压入 `cs:ip`, 并根据地址表中的信息, 跳转到相应地址。在中断返回时, 要首先向硬件端口输出相应参数, 告知中断处理硬件中断完成。随后, `iret` 指令将把 `FLAGS` 和 `cs:ip` 出栈, 并跳转回相应的用户程序地址。

这些中断调用的原理就给了自定义中断的可能。用户只需修改中断地址表, 并按照中断的行为书写自己的代码, 就能实现自己的中断。

2.0.2 软硬中断简介

软件和硬件都能引发中断。其中软件中断又称为系统调用。其与调用一个函数 (在程序员可见角度) 类似, 有输入的参数和相应的返回值。程序员在系统调用前应手动保存需要保存的寄存器, 并能预计到中断会修改相应寄存器的值。系统调用都是不可屏蔽的。

硬件中断有可屏蔽和不可屏蔽两种。其中时钟中断 (`watchdog`) 是不可屏蔽中断。由于硬件中断可以发生在任何一个时间点, 所以硬件中断应能保证绝对不修改任何寄存器的值。即硬件中断对程序员是透明的。

2.0.3 用户自定义中断

用户自定义中断可以采用如下的方式。首先查阅中断地址表, 找到未被占用的中断号 (例如本项目使用的 `0x2B` 号)。将地址表的相应表项修改用户自定义的地址。则当中断发生时, 程序控制流可以到达用户自定义地址。

标准的中断实现中, 用户自定义中断应该提取 `ah` 作为功能号, 建立中断自己的功能地址表, 根据 `ah` 的值在功能地址表中提取相应功能的中断入口地址, 并跳转到该地址。

中断返回时, 要保证将栈退到“刚进入中断”时的状态, 向硬件端口传送一些数据, 并用 `iret` 指令返回。

2.1 实验环境

2.1.1 系统与虚拟机

- 操作系统
本实验在 Linux 下完成。采用 Ubuntu 16.04
- 虚拟机
bochs. 它是一款开源且跨平台的 IA-32 模拟器。

2.1.2 相关工具、指令

- 汇编器
NASM. NASM 是一个轻量级的、模块化的 80x86 和 x86-64 汇编器。它的语法与 Intel 原语法十分相似, 但更加简洁和易读。它对宏有十分强大的支持。
- 编译器
GCC. GNU/GCC 是开源的 C 语言编译器。其产生的伪 16 位代码可以与 NASM 结合, 混编生成伪 16 位程序。
- 镜像文件产生工具
bxiimage. 该命令允许生成指定大小的软件镜像。
- 二进制写入命令
dd. dd 允许指定源文件和目标文件, 将源文件的二进制比特写入目标文件中的指定位置。
- 二进制文件查看命令
xxd. xxd 允许将二进制文件中的内容按地址顺序依次输出, 可读性强
- 反汇编器
objdump. objdump 可以查看目标文件和二进制文件的反汇编代码, 还能指令 intel 或 at&t 格式显示。
- 代码生成脚本
makefile. makefile 脚本具有强大的功能, 其可以识别文件依赖关系, 自动构筑文件, 自动执行 shell 脚本等。

3 实验过程

3.0.1 GCC 语言拓展的运用

为了简化代码的书写, 本项目进一步地使用了 GCC 的语言拓展, `__asm__` 语法。代码1举出了两个比较有说明性的道理, 列举其用法。内嵌汇编的初衷是, 若 C 语言不得不调用汇编, 而汇编语句又很短 (10 行以内), 为了汇编而新建一个文件并编译链接显得很麻烦。内嵌汇编可以做到“细粒度”的 C 与汇编交互功能。

`volatile` 语句的作用是“禁止 GCC 优化 (删除) 掉汇编语句”。由于 GCC 在编译时无法解析汇编的内容, 汇编的内容是在 C 语言代码完全编译后, 再由内置汇编器解释的。故 GCC 无法在编译期确定内嵌的汇编代码是否有副作用。若 GCC (错误地) 认为这些汇编代码没有副作用, 会把代码完全优化 (删

除) 掉。volatile 可以禁止这种优化。

整个拓展语法为 `__asm__ volatile (assembly : output : input : clobber);`

其中 assembly 为字符串表示的汇编代码。以回车结尾以分隔汇编代码。output 显式地指明了汇编代码会“输出”到 C 变量, 即汇编代码会修改某个 C 变量的值。input 显式地指明了汇编代码会从 C 语言“输入”值, 即把 C 的某个变量赋给寄存器。clobber 显式地指明了汇编语句会修改到哪些内容 (如标志寄存器, 内存等)。如果不指明, C 语言会 (为了速度) 而不刷新缓存, 导致汇编语句的修改没有更新到 C 语言环境中。

代码1中, “c”, “D” 分别代表寄存器 CX 和 DX, “CC” 表示标志寄存器, 其他标志可从字面意思看懂。这些标志与架构密切相关, 可以从 GCC 官网的 ASM 语言拓展查到详细的使用手册。

代码1的两个函数中, 第一个函数通过中断的方式在屏幕的某个特定的位置显示字符。C 语言对它做了便利的封装。第二个函数用于得到光标位置, 并自动导出到 cursor_pos 中。

代码 1: GCC 内嵌汇编拓展语法详解

```
static inline int16_t _draw_char(char ch, int offset, uint8_t style) {
2   uint16_t written_data = ch | (style << 8);
   __asm__ volatile (
4       "movb $1, %%ah\n"
       "int $0x2B\n"
6       : /* no output */
       : "c"(written_data), "D"(offset)
8       : "cc", "ax", "memory"
   );
10  return 1;
}

12 static inline uint16_t get_cursor(){
   uint16_t cursor_pos;
14   __asm__ volatile(
       "pushw %%ax\n"
16       "pushw %%bx\n"
       "movb $0x03, %%ah\n"
18       "movb $0, %%bh\n"
       "int $0x10\n"
20       "popw %%bx\n"
       "popw %%ax\n"
22       : "=d"(cursor_pos)
       : /* no input */
24       : "cc", "cx"
   );
26  return cursor_pos;
}
```

3.1 修改时钟中断

本步骤的重点在于，保证时钟中断结束后不修改任何寄存器的值，且进入时钟中断的时立即将段寄存器初始化到正确的状态。

由于时钟中断可以发生在代码执行的任何一个时刻，用户程序在执行到任何一个状态时，都有可能被突然间中断。如果时钟中断没有保存（哪怕一个）寄存器的值，当中断返回时，用户程序会直接崩溃。更可怕的是，有可能用户程序不立即崩溃，带着错误的状态继续执行下去，错误像滚雪球般地越积越多。当程序错误地退出时，很难定位到 BUG 的具体发生位置。

更需要注意的是，在时钟中断进入后，要尽快地将 ds 的值设置为 cs 的值。虽然本项目使用平坦模型 (flatten model)，段寄存器的值都是 0，但是由于时钟中断可以发生在任何时候（哪怕当前正在运行另外一个中断）。而另外的中断（例如 int 16H）的代码中很可能临时地修改段寄存器。在时钟中断突然来临后，ds 的值将是错误的值。这同样会导致程序的异常出现。

代码2展示了自定义时钟中断的实现。假设代码已经成功地将中断向量表地址修改为 timeOut 汇编函数的首地址。timeOut 后，程序会先保存一些寄存器的值，将 ds 同步为 cs，随后使用 call 指令调用 C 的真正处理函数。当从 C 函数中返回后，恢复相关寄存器，随后跳转到原时钟中断 (0F000H:0FEA5H) 的地址。由于不清楚原时钟中断做了什么，所以最好在程序最后跳转回原中断，让它完成它想完成的事情。

代码 2: 自定义时钟中断的部分代码

```
timeOut:
2
4     pusha
4     push gs
4     push ds
6     push es

8     mov ax, cs
8     mov ds, ax
10    mov es, ax

12    push cs
12    call timeout ; timeout is a C function
14

16    pop es
16    pop ds
16    pop gs
18    popa

20    jmp 0F000H:0FEA5H ; 为了展示目的，这里采用硬编码的方式
```

3.2 修改键盘中断

键盘中断的实现和时钟中断十分类似。首先修改中断向量表，保证中断发生时控制权限能到达用户自定义程序。只要保证能恢复到中断前的状态，程序一般就能正确运行。

本个实现的小技巧是, 采用 `pushf` 和 `call far` 的方式先调用旧键盘中断处理程序。由于旧中断处理程序的最后一条指令必为 `iret`, 刚好可以把 `pushf` 和 `call far` 中压栈的内容正确地恢复到原有的寄存器中。随后自定义中断服务程序再通过一个 `iret` 把栈完全复原。

3.3 安装用户自定义中断

为了让用户自定义中断的行为和系统自带的中断保持类似, 自定义中断也采用 `ah` 作为功能号标志。具体实现如代码3所示。

`global_custom_int_install` 函数安装用户自定义中断, 把 `2BH` 号中断链接到 `custom_int_handler` 函数中。该函数是个中断处理函数, 它负责解析功能号 (`ah`) 的值, 在一个向量表中查询服务程序的地址, 并将控制权移交给服务程序。值得注意的是, 所有的服务程序都通过 `retf` 的方式先返回到 `custom_int_handler`, 由其统一地使用 `iret` 结束中断。也就是 `custom_int_handler` 是一个统一的入口和出口, 做了封装和“分发处理”。

`custom_int_table` 地址表。功能号 `ah` 代表着地址在表中的位置。本项目定义了三个简单的中断程序。第一个叫 `testcase`, 它做了若干入栈和出栈操作。它用来检测“从中断安装到中断执行”的过程中, 栈是否正确地返回了。`_int_draw_char` 是重写的底层输出函数。所有的 IO 函数几乎都依赖于这一中断调用。见3.4 节对输出的工作有其他的介绍。`_int_draw_my_info` 中断会在程序的特定位置显示 3-D 的学号。视觉效果较好。

3.4 自定义的底层输出函数

为了给屏幕四周的边框预留位置, 本实验实现了自定义的底层输出函数, 只需要实现该底层输出函数, 高层的 IO 函数可以几乎不经任何修改, 就能自动地腾出边框的位置。

如代码4所示, 要满足上述要求, 首先需要具有手动设置光标的能力。当检测到光标的位置在倒数第二列时 (因为要把最后一列空出来给边框), 在输出后需要手动设置光标到下一行的第一列。当检测到光标在倒数第二行时, (同理, 让出最后一行给边框), 需要手动将屏幕向上滚动一行。

还有一些细节需要考虑, 例如输出回车, 输出换行, 输出退格等的光标操作。若输出回车, 则将光标设置到行首, 行号不变。如果输出换行, 则需要将光标向下移动一行, 但光标的列号不变。注意, 换行和回车是不一样的。若输出退格, 则需要将光标的列号-1。这需要在输出函数中加入若干 `if` 语句做手动判断。

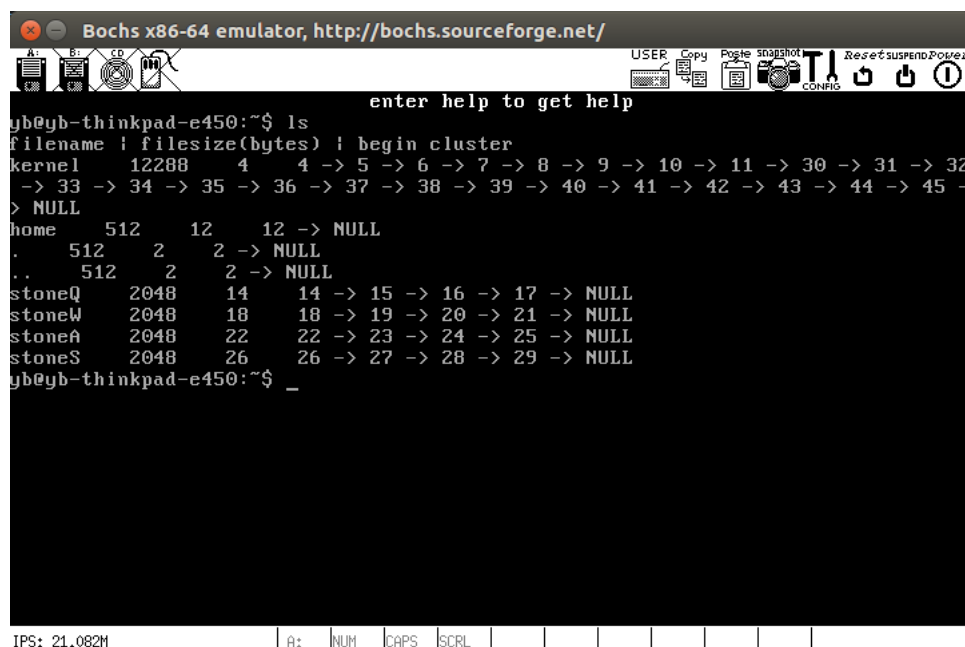
函数中调用的 `_draw_char` 函数是 C 函数, 它的函数体完全由内嵌汇编组成。该内嵌汇编只做了一件事情, 调用中断。该中断真正完成了输出字符到屏幕的操作。

如何实现滚屏和光标设置不再赘述。可直接阅读源码。在 `stdio.h` 和 `utilities.h`, 两个文件中。也可在文档网页中搜索函数名看具体实现。

4 实验结果

4.1 终端界面

图1展示了终端界面的外观。由于四周围的边框需要有闪动的字符, 所以终端的所有提示信息都应自动地为边框腾出空间。3.4 节详细介绍了如何为边框腾出空间。为了完成这个效果, 本项目结合了滚屏中



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
enter help to get help
yb@yb-thinkpad-e450:~$ ls
filename  filesize(bytes)  begin cluster
kernel    12288            4  -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 30 -> 31 -> 32
-> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 41 -> 42 -> 43 -> 44 -> 45 -
> NULL
home      512             12  -> NULL
.         512             2  -> NULL
..        512             2  -> NULL
stoneQ    2048            14  -> 15 -> 16 -> 17 -> NULL
stoneW    2048            18  -> 19 -> 20 -> 21 -> NULL
stoneA    2048            22  -> 23 -> 24 -> 25 -> NULL
stoneS    2048            26  -> 27 -> 28 -> 29 -> NULL
yb@yb-thinkpad-e450:~$
```

图 2: 带有文件所有簇号列表的 ls 指令

4.2 用户程序

图3实现了带按键反馈的用户程序。在进入用户程序时，安装自定义中断，键盘对每次按下都会输出“OUCH!”，而在离开用户程序后，中断复原，不再对键盘按下响应“OUCH!”。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G
d
c
b
a
Z
Y
X
W
V
U
T
S
R
Q
ouch! ou h ou ch! ouc ! ouch! ouch! ouch! ouch!
o c ! ouc ! ouch! o c ! ouch! ouch! ouch! ouch!
uch!
XWUUTSRQPONMLKJ HGFEDCBAzyxwvut rqp onmlkjihgfed
```

图 3: 带按键反馈的用户程序

4.3 特色中断

本实现实现了特色中断，可以以 3D 的形式输出本人的学号。如图4 所示。

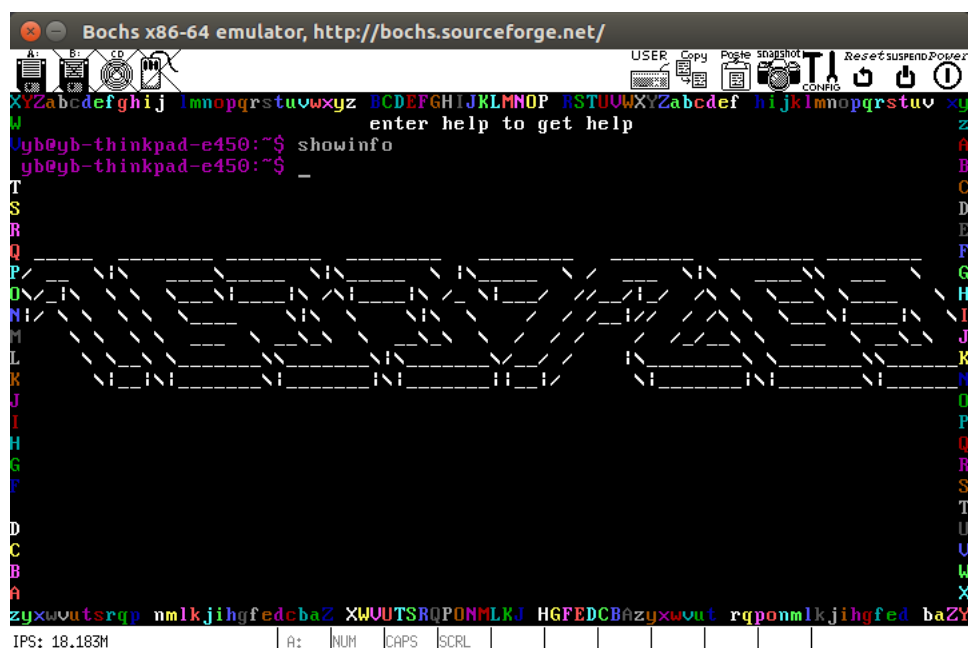


图 4: 特色中断: 个人学号的 3D 显示

5 实验总结

5.1 亮点介绍

5.1.1 极其详尽的文档

本次项目为所有文件添加了足够的注释，并自动生成了说明文档。说明文档可以见网页

<https://yanb25.github.io/OperatingSystem/files.html>。(可点击)

其上极其详细地为几乎所有函数增加说明，列举了函数间的依赖关系以及调用方法等。

同时，本项目中附属的项目文档 pdf 文件也由代码和注释自动生成。

5.1.2 ASM 拓展语法

本实验在上个实验的基础上，更深入地使用了 GCC 的 ASM 语言拓展。ASM 语言拓展分为 basic asm 和 extend asm。以往的项目只使用 basic asm，从本项目起加入 extend asm 语法的内容，为 C 和汇编的交互带来了更大的便利。见3.0.1节。

5.1.3 格式化 ls 输出

本实验实现了格式化的 ls 输出，在输出文件名的同时，格式化地输出文件大小和文件首簇位置。本实验还实现了函数，可以将文件的所有簇按序输出。见图2和图1。其他寄存器

5.1.4 光标滚屏自主控制

为了完成终端中闪烁的边框,本项目将 IO 函数的一个底层中断重写了一遍。并结合了光标控制和自动滚屏。见3.4节的具体细节。

5.2 更新细节

5.2.1 printf 进一步完善

实现了定宽输出的功能。即支持 `printf("%4d", 1)` 功能,在输出的 1 前输出 3 个空格。同时,所有的 IO 函数重构成返回 `int16_t` 类型,代表他们输出到屏幕的字符数。这为格式化输出提供了巨大的帮助。

5.2.2 目录树结构变更

将 C 程序库进一步细分,分成了 `utilities.h`, `stdio.h`, `ctype.h`, `mystring.h` 等基本库。`utilities.h` 负责内核所需的最基本的操作,例如扇区读写和光标控制等。其他头文件作用类似与 ANSI C 的实现。

5.3 心得体会

本次实验稍微晚交了一天,因为这次实验中我又再次整理微调了目录树,并给所有的函数增加了详尽的文档。以后的实验中,代码量会逐渐加大,翔实的文档应该能带来很大的帮助(也能给后来的人一些参考)。

这个实验遇到的 BUG 比上次文件系统的实验中还多。这主要是因为中断的 DEBUG 并不好进行。例如,我无法预知下一次时钟中断发生的时间,不能在中断发生前(有预谋地)提前设置断点。两次时钟中断的时间间隔虽短,但中间可能间隔着无数行代码了。这些因素都让我的这次 DEBUG 感到举步维艰。对 BUG 的具体讨论见 5.4 节。

这次实验的一部分工作量还在于完善底层的函数。例如 IO 函数和字符串处理函数等。这些改动在本项目中还难以体现(唯一能体现的就是对齐的 `ls` 输出了),但在以后的实验中,相信能成为我最重要的工具。

5.4 BUG 汇总

5.4.1 扇区加载错误

首先是发生了扇区加载不足的状况。操作系统内核共 10 个扇区长度,但在最开始的实现中只向内存加载了 8 个扇区。其实这个 BUG 并没有带来多大的影响。这可能是因为最后两个扇区中的代码都是些辅助函数,而这些函数恰好没有被调用到。但我依然在发现了这个问题后第一时间把它改了过来。这也是为什么,内核的簇在文件系统中是不连续的:内核的后半部分是我后来才加入文件系统的。图2中也能看出, `ls` 指令正确地显示了内核中每个簇的编号。也能看到他们确实是不连续的。

第二个问题是磁面号计算错误。我之前错误地将磁面的计算写成了

$$logic - sector \div 18$$

类似的写法。这个写法对 36 号前的逻辑扇区换算都是正确的。但是当扇区号高于 36 时,会算出磁面号为 2! 这个隐秘的 BUG 很容易捕捉:当我发现我最后一个用户程序一运行就卡死时,我就很快发现了它根本没有被加载入内存。但是找到我的磁面数计算错误却花了我很多的时间。因为一开始我没有相通,为什么前面的扇区能载入,后面的扇区就无法载入了!

5.4.2 时钟中断未正确保存上下文

这个 BUG 花了我很多的时间, 因为当时我几乎是把“程序化简到最简”再运行, 程序却始终出错。有时是卡死, 有时是程序屏幕变花, 有时甚至出现了如图5所示的错误。这个错误极其让我困惑: 中断 16H 越界? 我根本没有动过 16H 啊。我几乎只是把时钟中断安装了, 自定义的时钟中断没有做任何事, 立即就返回了。

后来我才想到, 我在中断中没有把 DS 设置为 CS。我的所有程序暂时还使用平坦内存模型 (flatten model), 所有段寄存器都设置为 0, 我以为不可能出现 DS 被修改的状况。然而因为时钟中断可以发生在程序运行的任何时刻, 包括其他中断正在运行时。其他的中断有可能临时地修改了段寄存器, 所以当进入时钟中断时, 段寄存器的值是错的。

这个错误很难被找到。我确信我自己写的代码绝对不会修改到段寄存器, 所以我潜意识地默认了, 段寄存器一定不会被修改, 所以我很自然地就没有初始化 DS 寄存器。为了找到这个 BUG, 我用 bochs 跟着程序控制流一路走, 走过了 6/7 层嵌套的函数调用, 最终在某刻突然发现段寄存器的值被改变了。最后发现是中断 16H 在程序中会有修改 DS 的操作。虽然 16H 保证在返回前把 DS 复原, 但时钟中断可能在 DS 复原前发生。

DS 错误是一个很难捕捉的 BUG。它不会让程序立即崩溃。相反, 它让程序进入一个错误的状态, 它任由错误日积月累, 让错误像滚雪球一样地越积越多, 直到最后程序崩溃。程序崩溃的位置和错误发生的位置相隔十万八千里了。

这也让我理解到了软件开发的一些定律, 例如: 尽力让错误在编译期捕获; 当程序无法处理该异常时不要捕获该异常 (let it crash 定律)。这些定律的目标都是一致的: 让程序尽早地展现出它的错误, 而不是把错误隐藏, 带着错误的状态继续运行程序。

我会考虑在下一个版本的实验中增加 assert 函数。这个函数的参数是一个条件表达式, 当表达式运算为 False 时直接停止程序的继续运行, 并报出文件名和行号 (如果可以的话)。

5.4.3 双次安装中断的错误

这个错误完全是个人不小心。我在用户程序进入前, 把键盘中断安装成自定义中断, 但没有在退出时把旧中断复原 (我还没有实现复原中断的逻辑, 我习惯写完一部分代码就先做测试, 测试完了再继续开发)。

当我两次进入用户程序时, 自定义中断就会被安装两次! 第一次安装时, 旧中断被缓存到 old_int_address 变量中。第二次进入中断时, 用户自定义中断就被写入到 old_int_address 中了! 旧中断的地址就被用户程序的地址改写了!

这是个很蠢的 BUG, 但当时的注意力没有放在这里, 一直怀疑自己是其他地方出了错, 所以没有即时地找到这个 BUG。

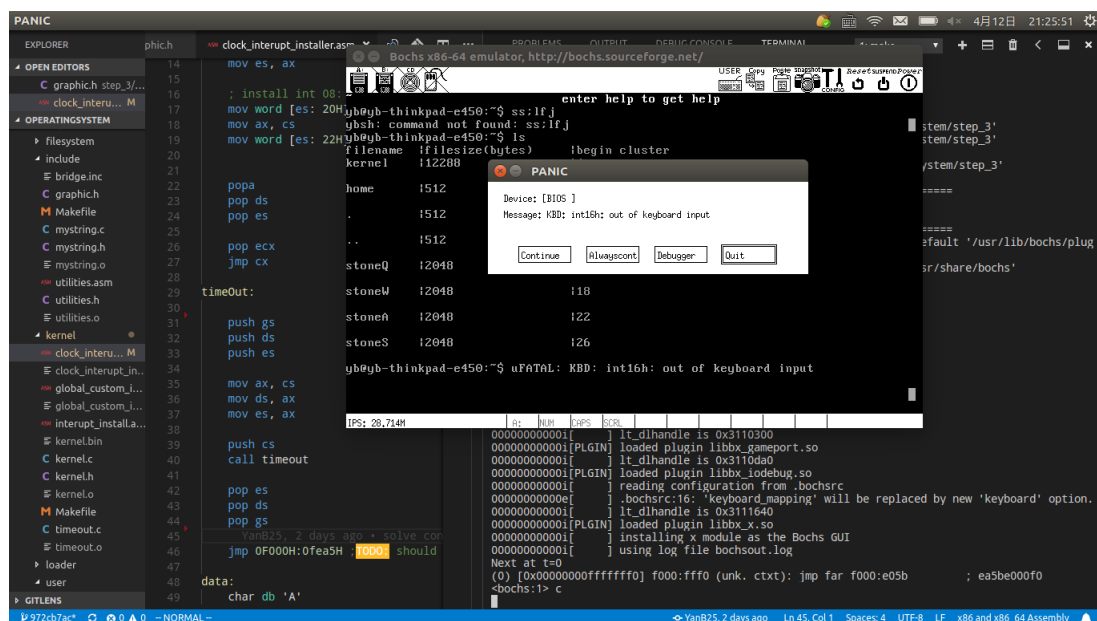


图 5: 中断未正确保存上下文时遇到的诡异的 BUG

附录 A 参考文献

1. <https://blog.csdn.net/longintchar/article/details/50602851>
16 位和 32 位汇编指令的不同 (尤其是 push 指令)
2. <https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1>
GCC 内嵌汇编的书写。

代码 3: 用户自定义中断的实现

```
global custom_int_install:
2   push ds
   mov ax, 0
4   mov ds, ax

   mov word [ds:2BH * 4], custom_int_handler
   mov word [ds:2BH * 4 + 2], cs
8
   pop ds
10  retl
custom_int_handler:
12  ; ah=01H: _draw_char
   push bx
14  push ax
   push dx
16
   mov dl, 4
18  mov al, ah
   xor ah, ah
20  mul dl
   add ax, custom_int_table
22  mov bx, ax

   pop dx
24  pop ax

   call far [bx]
28
   pop bx
30  iret

32  testcase:
   ...
34
   retf
36  _int_draw_char:
   ...
38
   retf
40
_int_draw_my_info:
42  ...

44  retf
custom_int_table:
46  dw testcase
   dw 0x0000
48  dw _int_draw_char
   dw 0x0000
50  dw _int_draw_my_info
   dw 0x0000
```

代码 4: 自定义的底层输出函数的实现

```
static inline int16_t putch_style(char ch, uint8_t style) {  
2   uint16_t cursor = get_cursor();  
   uint8_t crow = cursor >> 8;  
4   uint8_t ccol = cursor & 0b11111111;  
   if (ch == '\r') {  
6       set_cursor(crow, 1);  
   } else if (ch == '\n') {  
8       if (crow >= 23) {  
           scroll_up_one_line();  
10      } else {  
           set_cursor(crow + 1, ccol);  
12      }  
   } else if (ch == '\b') {  
14      set_cursor(crow, ccol-1);  
   }  
16   else if (ch == '\t') {  
       set_cursor(crow, ccol + 4);  
18   }  
   else {  
20       _draw_char(ch, (crow * 80 + ccol)*2, style);  
       ccol++;  
22       if (ccol == 79) {  
           crow++;  
24           ccol = 1;  
       }  
26       set_cursor(crow, ccol);  
   }  
28   return 1;  
}
```