Linux 与 SEH

Linux与SEH 1
前言 2
Windows的SEH与Linux的NSEH 3
NSEH的实现分析 19
NSEH在内核模块编程中的应用 39
总结 46
后记 47

前言

本文详细剖析了 Linux 内核中类似 Windows 下 SEH 的 exception handling 机制及在 Kernel Module 编程中的应用。

Windows的SEH与Linux的NSEH

前几天与一位朋友聊起 Linux 下开发 device driver,他对于 Linux 内核不支持类似 Windows 下的 SEH(Structure Exception Handling)机制感到很遗憾。他随手举了下面两个例子：

例一

```
int *pIntValue = 0;
static int   temp = 0;


__try
{
  *pIntValue = 1;
}
__except(EXCEPTION_CONTINUE_EXECUTION)
{
  pIntValue = &temp;
}

DbgPrint("temp = %d\n", temp);
```

例二

```
int *pIntValue = 0;


__try
{
  *pIntvalue = 1;
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
  return false;
}

return      true;
```

我想了好一阵，尽力想在 Linux 找到对应的功能，但实在找不到。其实一看到上面的代码，我就想起 Linux 中的一个类似功能，但由于实在太丑陋，我觉得用它来反驳好像太无力。

我先列出在 Linux 内核开发中要实现上面功能的例子代码：

例一

```
int *pIntValue = 0;
static int  temp = 0;

__asm__ __volatile__(
"1:  movl $1, (%eax)\n\t"
".section .fixup, \"ax\"\n\t"
"2:  movl %2, %%eax\n\t"
"jmp 1b\n\t"
".previous\n\t"
".section __ex_table, \"a\"\n\t"
".align 4\n\t"
".long 1b, 2b\n\t"
".previous\n\t"
:
:"a"(pIntValue),"m"(&temp)
);

printk("temp = %d\n", temp);
```

例二

```
int *pIntValue = 0;

__asm__ __volatile__(
"1:  movl $1, (%eax)\n\t"
"movl $1, %%eax\n\t"            作为返回值，令%eax = 1,即 return true
"2:  nop\n\t"
".section .fixup, \"ax\"\n\t"
"3: xorl %%eax, %%eax\n\t"      作为返回值，令%eax = 0,即 return false
```

```
  "jmp 2b\n\t"
  ".previous\n\t"
  ".section __ex_table, \"a\"\n\t"
  ".align 4\n\t"
  ".long 1b, 3b\n\t"
  ".previous\n\t"
  :
  :"a"(pIntValue)
  );
```

如果你不喜欢看上面的代码，那很正常。一个就象美丽的少女，另一个则象丑陋的老太婆，但两者实现的功能是完全一样的。gcc 并没有提供象 Microsoft C++那样的__try, __except, __finally 关键字来支持异常处理（exception handling）。对 c / c++ programmer 而言，异常处理需要两方面的支持：

1. c/c++编译器方面的支持
2. 操作系统内核的支持

Microsoft 两者都支持得非常好，在编译器方面是通过引入__try, __except, __finally 关键字来实现对 programmer 的友好支持。在内核方面的支持从它能达到的功能看，也比 Linux 强大。在 Linux 世界，则编译器方面（gcc）没有为 c programmer 做一点支持,以至于不得不用 assembly 来实现；在内核方面，异常处理机制功能没有 Windows 强大，只能勉强说有这样的功能。

对 Windows 的 SEH 的介绍，最经典的自然是 Matt Pietrek 大师的《A Crash Course on the Depths of Win32 Structured Exception Handling》一文。另外一些 Windows 的黑客们也零零碎碎的写过一些剖析的文章，但遗憾的是所有的这些都是基于用户态的数据结构的剖析，而没有涉及 Windows 内核对 SEH 的实现机制。我不知道是大师们故意遗漏还是觉得无关紧要而忽略了。

Windows SEH 的强大更显示出它内核支持机制的神秘（当然没有原代码可看才是它神秘的根本所在）。还好，Linux 是可以看到原代码的。它确实没有 Windows SEH 那样强大，但毕竟还是能完成类似 SEH 的大部分功能。虽然从代码易读性上来说，简直象天书一般。其实基于现在 Linux 的简陋的异常处理机制，gcc 也是能够实现类似 Microsoft C++那样的 SEH 语法的，关键是 gcc 的开发者也许觉得没必要。分析 Linux 内核的异常实现机制或许对理解 Windows 内核对 SEH 的支持有点帮助吧。

异常处理往复杂了说很复杂，但往简单了说又很简单。大白话就是当出错以后，跳到程序员指定的代码去执行。

比如

例一

```
int *pIntValue = 0;
static int   temp = 0;

__try
{
  *pIntValue = 1;          ①          ③
}
__except(EXCEPTION_CONTINUE_EXECUTION)
{
  pIntValue = &temp;       ②
}

DbgPrint("temp = %d\n", temp);    ④
```

上面代码的执行顺序是①②③④。如果在执行①时出现 exception，则跳砖到②去执行，EXCEPTION_CONTINUE_EXECUTION 是告诉系统在执行完②以后，恢复到出现 exception 的地方，即①的地方重新执行③，然后执行④。这里有两点是关键：

- ①与②是配对的，即当执行①出错时，就由②来处理
- 在跳转到 exception handler(②)后，通过参数（比如这里的 EXCEPTION_CONTINUE_EXECUTION）来告诉系统是执行③还是④

例二

```
int *pIntValue = 0;

__try
{
  *pIntvalue = 1;          ①
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
  return false;        ②
}

return   true;             ③
```

上面代码的执行顺序是①②或者①③。如果执行①时没有发生 exception,则接下来执行③，否则执行②。EXCEPTION_EXECUTE_HANDLER 告诉系统不用恢复①的执行。

而 Linux 下的对应例子只不过以 assembly 的形势来体现上面的思路。
例一

```
int *pIntValue = 0;
static int  temp = 0;


__asm__ __volatile__(
"1:  movl $1, (%eax)\n\t"
".section .fixup, \"ax\"\n\t"
"2:  movl %2, %%eax\n\t"
"jmp 1b\n\t"
".previous\n\t"
".section __ex_table, \"a\"\n\t"
".align 4\n\t"
".long 1b, 2b\n\t"
".previous\n\t"
:
:"a"(pIntValue),"m"(&temp)
);


printk("temp = %d\n", temp);
```

为了符合 gcc 的嵌入式汇编的语法，上面的代码才那么难懂。我来把它翻译一下，使它易懂一点。

```
.text 代码段
  int *pIntValue = 0;
  static int temp = 0;


  movl      pIntValue, %eax
1:   movl      $1, (%eax)
```

```
    printk("temp = %d\n", temp);
```

```
    .fixup 代码段
2:    movl      &temp, %eax
    jmp     .text 代码段的标号 1
```

```
  __ex_table 数据段
 （.text 代码段标号 1 的地址，.fixup 代码段标号 2 的地址）
```

上面短短几行汇编代码生成的指令和数据被链接器(linker)放在三个地方

- .text 代码段
- .fixup 代码段
- __ex_table 数据段

在.text 代码段中的是正常流程的代码，.fixup 代码段中的是异常处理代码，而__ex_table 数据段中放的是可能出异常的代码的地址于处理该异常的代码的地址对（address pair）。很显然，内核怎么知道当出现异常时应该调用哪儿的代码来处理，就是由这一对地址对来提供的。在 Windows 中，上面这些肮脏的细节都被 Microsoft 给隐藏起来了，而 Linux 则要你手工来完成这一切。

在 Linux 下 Programming 或许要比 Windows 下吃力，但也有难以抹杀的优势 --- Open Source。正基于此，我们才可以比较省力的剖析 Linux 内核是怎样实现 exception handling 机制的。Matt Pietrek 这样的大师出手才能写出 Windows 下 SEH 的经典剖析《A Crash Course on the Depths of Win32 Structured Exception Handling》，而非我等无名小辈可以乱言的。而由于 Linux 的开源性质，才使得我等无名小辈也“乱言”了。

由于 Linux 下对异常处理的简陋，我姑且把它命名为 NSEH --- **N**on-**S**tructured **E**xception **H**andling。

NSEH 的起源

NSEH 的简陋是有原因的。因为它的出现，一开始并不是象 Windows 下的 SEH 一样作为像模像样的通用异常处理器，而是为了针对避免在每次 kernel 空间访问 user 空间的内存时必不可少的合法性检查而造成的性能低下提出的解决方案。作为通用 exception handling 是它的一个“副产品”。

比如 write ()系统调用

```
  ssize_t write(int fd, const void *buf, size_t count);
```

对应到 kernel 中处理该函数的内核代码如下

在 Linux 2.0.40 版本中未引入 NSEH 的 write()系统调用的实现

```
fs/read_write.c


asmlinkage int sys_write(unsigned int fd,char * buf,unsigned int count)
{
  这里的 buf 是用户空间的 memory
        int error;
        struct file * file;
        struct inode * inode;


        error = -EBADF;
        file = fget(fd);
        if (!file)
                goto bad_file;
        inode = file->f_inode;
        if (!inode)
                goto out;
        if (!(file->f_mode & 2))
                goto out;
        error = -EINVAL;
        if (!file->f_op || !file->f_op->write)
                goto out;
        error = 0;
        /*
         *      If this was a development kernel we'd just drop the test
         *      its not so we do this for stricter compatibility both to
         *      applications and drivers.
         */
        if (!count && !IS_ZERO_WR(inode))
                goto out;
     error                                                          =
locks_verify_area(FLOCK_VERIFY_WRITE,inode,file,file->f_pos,count);
     if (error)
```

```
            goto out;
    在内核访问以前必须先 check,该用户态的内存是否可读，否则会 crash 在内核里
      error = verify_area(VERIFY_READ,buf,count);
      if (error)
            goto out;
        /*
         * If data has been written to the file, remove the setuid and
         * the setgid bits. We do it anyway otherwise there is an
         * extremely exploitable race - does your OS get it right |->
         *
         * Set ATTR_FORCE so it will always be changed.
         */
        if (!suser() && (inode->i_mode & (S_ISUID | S_ISGID))) {
               struct iattr newattrs;
               /*
                * Don't turn off setgid if no group execute. This special
                * case marks candidates for mandatory locking.
                */
               newattrs.ia_mode = inode->i_mode &
                     ~(S_ISUID | ((inode->i_mode & S_IXGRP) ? S_ISGID :
0));
               newattrs.ia_valid = ATTR_CTIME | ATTR_MODE | ATTR_FORCE;
               notify_change(inode, &newattrs);
        }

        down(&inode->i_sem);
        error = file->f_op->write(inode,file,buf,count);
        up(&inode->i_sem);
  out:
        fput(file, inode);
  bad_file:
        return error;
```

```
}
```

上面标红的代码即是对 user 空间内存的合法性检查。

mm/memory.c

```
/*
 * Ugly, ugly, but the goto's result in better assembly..
 */
int verify_area(int type, const void * addr, unsigned long size)
{
        struct vm_area_struct * vma;
        unsigned long start = (unsigned long) addr;

        /* If the current user space is mapped to kernel space (for the
         * case where we use a fake user buffer with get_fs/set_fs()) we
         * don't expect to find the address in the user vm map.
         */
        if (!size || get_fs() == KERNEL_DS)
                return 0;

        vma = find_vma(current->mm, start);
        if (!vma)
                goto bad_area;
        if (vma->vm_start > start)
                goto check_stack;

good_area:
        if (type == VERIFY_WRITE)
                goto check_write;
        for (;;) {
                struct vm_area_struct * next;
                if (!(vma->vm_flags & VM_READ))
                        goto bad_area;
```

```
                if (vma->vm_end - start >= size)
                        return 0;
                next = vma->vm_next;
                if (!next || vma->vm_end != next->vm_start)
                        goto bad_area;
                vma = next;
        }

  check_write:
        if (!(vma->vm_flags & VM_WRITE))
                goto bad_area;
        if (!wp_works_ok)
                goto check_wp_fault_by_hand;
        for (;;) {
                if (vma->vm_end - start >= size)
                        break;
                if      (!vma->vm_next      ||      vma->vm_end      !=
vma->vm_next->vm_start)
                        goto bad_area;
                vma = vma->vm_next;
                if (!(vma->vm_flags & VM_WRITE))
                        goto bad_area;
        }
        return 0;

  check_wp_fault_by_hand:
        size--;
        size += start & ~PAGE_MASK;
        size >>= PAGE_SHIFT;
        start &= PAGE_MASK;

        for (;;) {
```

```
                do_wp_page(current, vma, start, 1);
                if (!size)
                        break;
                size--;
                start += PAGE_SIZE;
                if (start < vma->vm_end)
                        continue;
                vma = vma->vm_next;
                if (!vma || vma->vm_start != start)
                        goto bad_area;
                if (!(vma->vm_flags & VM_WRITE))
                        goto bad_area;;
        }
        return 0;

  check_stack:
        if (!(vma->vm_flags & VM_GROWSDOWN))
                goto bad_area;
        if (expand_stack(vma, start) == 0)
                goto good_area;

  bad_area:
        return -EFAULT;
  }
```

代码蛮长的，其实在做如下的检查

1. 用户空间的[addr， addr + size)是否分配
2. 该空间虽然已经分配，但 type（读，写，执行等）所指定的操作允许吗

有如下应用程序代码

```
  char *mem = malloc(1000);
```

```
...

for(int i = 0; i < 1000; i++)
{
  write(fd, mem, 1000);
}
```

也就是说当 write 调用陷入内核后，对 mem 空间要检查 1000 次（运行 verify_area(VERIFY_READ, mem, 1000) 一千次），这显然是非常耗时的，很影响性能。但内核又不能不作检查，因为在普通应用程序里面访问非法空间，最多是该程序异常退出，而在 kernel 里面则会 Kernel Panic，整个机器都当掉。我想你不会希望如下的代码就能搞垮你的机器吧。

```
char *mem = NULL;
write(fd, mem, 1000);
```

如果没有 verify_area(),你只有重启机器了。所以内核代码对用户态的空间是极不信任的，都要经过 verify_area（）的“搜身”，才能放行。这倒有点象我国的司法制度（有罪推定），先把大家看成有罪的人，然后有司法来检查是否正真正有罪。好处当然是不放纵一个坏人，但坏处呢？

对 Linux 内核而言，坏处是会造成性能低下。于是 Linux 内核开发者引入了 NSEH 机制。思路是这样的，现假定传入的用户空间是安全的，不需要在访问以前要通过 verify_area（）来“搜身”，而是正常访问。如果这个假定是正确的，那太好了，没有任何性能的损失。但如果假定是错误的，当然不能 Kernel Panic，它必须恢复过来，然后以报错的形势优雅的退出。看一下引入 NSEH 以后的 write()系统调用的实现

在 Linux 2.4.20 版本中已经引入 NSEH 的 write()系统调用的实现

```
asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t
count)
{
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_WRITE) {
            struct inode *inode = file->f_dentry->d_inode;
            ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file,
                  file->f_pos, count);
```

```
            if (!ret) {
                ssize_t (*write)(struct file *, const char *, size_t,
loff_t *);
                ret = -EINVAL;
                if (file->f_op && (write = file->f_op->write) != NULL)
                    ret = write(file, buf, count, &file->f_pos);
            }
        }
        if (ret > 0)
                dnotify_parent(file->f_dentry, DN_MODIFY);
        fput(file);
    }
    return ret;
 }
```

这里的 write 由某个具体的文件系统 driver 来实现，我这里以 Linux 下最普通的 ext2 文件系统为例。

```
 struct file_operations ext2_file_operations = {
        llseek:        generic_file_llseek,
        read:          generic_file_read,
        write:         generic_file_write,
        ioctl:         ext2_ioctl,
        mmap:          generic_file_mmap,
        open:          generic_file_open,
        release:       ext2_release_file,
        fsync:         ext2_sync_file,
 };
```

响应 ext2 文件系统写文件的 handler 是 generic_file_write（）。

```
 ssize_t
 generic_file_write(struct file *file,const char *buf,size_t count,
loff_t *ppos)
 {
   ...
```

```
        kaddr = kmap(page);
        status = mapping->a_ops->prepare_write(file, page, offset,
offset+bytes);
        if (status)
            goto sync_failure;
        page_fault = __copy_from_user(kaddr+offset, buf, bytes);
        flush_dcache_page(page);
        status = mapping->a_ops->commit_write(file, page, offset,
offset+bytes);
        if (page_fault)
            goto fail_write;

    ...
  }
```

generic_file_write（）是一个通用文件系统写处理器。上面标红的地方即是从用户空间拷贝内容到内核空间。__copy_from_user 是一个宏，在 include/asm-i386/uaccess.h 中。

```
  __copy_from_user(to,from,size)
```

从宏的命名上就可以看出是从用户空间的 from 处拷贝 size 个 bytes 到内核空间的“to”处。如果返回值为零，表示 size 个 bytes 被成功拷贝；如果返回值大于零，表示只拷贝了(size – return value)个 byte。拷贝，这么简单的动作都会失败，自然是用户空间“from”非法，引起 exception。看一下代码。

__copy_from_user 宏被层层展开后，最终调用如下的宏：

```
  #define __copy_user_zeroing(to,from,size)              \
  do {                                  \
    int __d0, __d1;                         \
    __asm__ __volatile__(                       \
        "0: rep; movsl\n"               \   可能引发异常的指令
        "   movl %3,%0\n"               \
        "1: rep; movsb\n"               \   可能引发异常的指令
        "2:\n"                          \
```

```
        ".section .fixup,\"ax\"\n"              \
        "3: lea 0(%3,%0,4),%0\n"            \ 处理 0 处的异常
        "4: pushl %0\n"                 \     处理 1 处的异常
        "   pushl %%eax\n"                  \
        "   xorl %%eax,%%eax\n"             \
        "   rep; stosb\n"               \
        "   popl %%eax\n"               \
        "   popl %0\n"              \
        "   jmp 2b\n"               \
        ".previous\n"                   \
        ".section __ex_table,\"a\"\n"             \
        "   .align 4\n"                 \
        "   .long 0b,3b\n"                  \
        "   .long 1b,4b\n"                  \
        ".previous"                     \
        : "=&c"(size), "=&D" (__d0), "=&S" (__d1)     \
        : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)   \
        : "memory");                    \
    } while (0)
```

把上面的代码按位置整理一下：

```
    .text section

    "0:   rep; movsl\n"
    "     movl %3,%0\n"
    "1:   rep; movsb\n"
    "2:   \n"
```

```
    .fixup section


    "3:   lea 0(%3,%0,4),%0\n"
    "4:   pushl %0\n"
    "     pushl %%eax\n"
```

```
"     xorl %%eax,%%eax\n"
"     rep; stosb\n"
"     popl %%eax\n"
"     popl %0\n"
"     jmp 2b\n"
```

```
__ex_table section


".long 0b,3b\n"
".long 1b,4b\n"
```

text section 中的是主流程代码，完成从 from 到 to 的拷贝动作，如果失败（用户空间无效）则跳转到.fixup section 中的异常处理。__copy_from_user()封装了整个过程，从外部看如果一切正常，则返回值为 0；如果失败，则返回未拷贝的字节数（非零）。也就是内核总是假定用户是守法好公民，只有你确实犯罪了（引发异常），他才会把你逮住。不会象以前一样，一上来就不分青红皂白的搜身，检查。

```
char *mem = malloc(1000);


...


for(int i = 0; i < 1000; i++)
{
  write(fd, mem, 1000);
}
```

对于上面的代码，由于 mem 是合法的，就像内核里面没有检查这一关一样。这对性能当然有很大帮助。

NSEH的实现分析

include/asm-i386/uaccess.h

```
  /*
   * The exception table consists of pairs of addresses: the first is the
   * address of an instruction that is allowed to fault, and the second
is
   * the address at which the program should continue.  No registers are
   * modified, so it is entirely up to the continuation code to figure
out
   * what to do.
   *
   * All the routines below use bits of fixup code that are out of line
   * with the main instruction path.  This means when everything is well,
   * we don't even have to jump over them.  Further, they do not intrude
   * on our cache or tlb entries.
   */

  struct exception_table_entry
  {
    unsigned long insn, fixup;
  };
```

上面的注释已经解释的够清楚了。内核本身的异常处理地址对（exception handling address pair）被集中放在系统文件的__ex_table section 中。让我们来验证一下（以 Redhat 8 的内核文件/boot/vmlinux-2.4.18-14 为例）。

```
  $readelf –S /boot/vmlinux-2.4.18-14

  There are 27 section headers, starting at offset 0x2846fc:


  Section Headers:
    [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
    [ 0]                   NULL            00000000 000000 000000 00      0   0  0
    [ 1] .text             PROGBITS        c0100000 001000 14b8de 00  AX  0   0 16
```

```
  [ 2] .rodata           PROGBITS        c024b8e0 14c8e0 02aea1 00   A  0   0 32
  [ 3] .kstrtab          PROGBITS        c02767a0 1777a0 009cfd 00   A  0   0 32
  [ 4] __ex_table        PROGBITS        c02804a0 1814a0 0027c0 00   A  0   0  4
  [ 5] __ksymtab         PROGBITS        c0282c60 183c60 002de8 00   A  0   0  4
  [ 6] __kallsyms        PROGBITS        c0285a48 186a48 07c6c9 00   A  0   0  4
  [ 7] .data             PROGBITS        c0302120 203120 043684 00  WA  0   0 32
  [ 8] .data.init_task   PROGBITS        c0346000 247000 002000 00  WA  0   0 32
  [ 9] .text.init        PROGBITS        c0348000 249000 01c8fc 00  AX  0   0 16
  [10] .data.init        PROGBITS        c0364900 265900 018318 00  WA  0   0 32
  [11] .setup.init       PROGBITS        c037cc20 27dc20 0001c8 00  WA  0   0  4
  [12] .initcall.init    PROGBITS        c037cde8 27dde8 0000a4 00  WA  0   0  4
  [13] .data.page_aligne PROGBITS        c037d000 27e000 000800 00  WA  0   0 32
  [14] .data.cacheline_a PROGBITS        c037d800 27e800 000fe0 00  WA  0   0 32
  [15] .bss              NOBITS          c037e7e0 27f7e0 0533a0 00  WA  0   0 32
  [16] .comment          PROGBITS        00000000 27f7e0 004ad2 00      0   0  1
  [17] .debug_aranges    PROGBITS        00000000 2842b2 000028 00      0   0  1
  [18] .debug_pubnames   PROGBITS        00000000 2842da 000022 00      0   0  1
  [19] .debug_info       PROGBITS        00000000 2842fc 0000bd 00      0   0  1
  [20] .debug_abbrev     PROGBITS        00000000 2843b9 0000a9 00      0   0  1
  [21] .debug_line       PROGBITS        00000000 284462 000088 00      0   0  1
  [22] .debug_frame      PROGBITS        00000000 2844ec 000024 00      0   0  4
  [23] .debug_str        PROGBITS        00000000 284510 0000bb 01  MS  0   0  1
  [24] .shstrtab         STRTAB          00000000 2845cb 00012f 00      0   0  1
  [25] .symtab           SYMTAB          00000000 284b34 045e40 10     26 2cd3  4
  [26] .strtab           STRTAB          00000000 2ca974 04b8e2 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

上面标红的是.text section 与__ex_table section。我们没看到.fixup section（Why）。

```
  [ 4] __ex_table        PROGBITS        c02804a0 1814a0 0027c0 00   A  0
0  4
```

__ex_table section 从文件的偏移 0x1814a0 开始。

```
$hexdump –C –s 0x1814a0 /boot/vmlinux-2.4.18-14

001814a0  fa 72 10 c0 fd 72 10 c0  7c 78 10 c0 e4 86 24 c0  |.r...r..|x....$.|

001814b0  7f 78 10 c0 ed 86 24 c0  45 7e 10 c0 f6 86 24 c0  |.x....$.E~....$.|

001814c0  4c 7e 10 c0 00 87 24 c0  57 7e 10 c0 0a 87 24 c0  |L~....$.W~....$.|

001814d0  61 7e 10 c0 14 87 24 c0  78 7e 10 c0 1e 87 24 c0  |a~....$.x~....$.|

001814e0  97 7e 10 c0 28 87 24 c0  a1 7e 10 c0 32 87 24 c0  |.~..(.$..~..2.$.|

001814f0  ab 7e 10 c0 3c 87 24 c0  b7 7e 10 c0 46 87 24 c0  |.~..<.$..~..F.$.|

...
```

以__ex_table section 中的第一项为例来研究一下。

fa 72 10 c0 fd 72 10 c0 = (0xc01072fa, 0xc01072fd)

构成一队地址对，其中 0xc01072fa 为可能发生异常的代码地址，而 0xc01072fd 则是异常处理器的地址。

```
  [ 1] .text             PROGBITS        c0100000 001000 14b8de 00  AX  0   0
16
```

由上可知.text section 所占地址空间为（0xc0100000, 0xc024b8de）,显然上面两个地址都落在此区间。由此也可推知在链接(link)阶段，.fixup section 被合并入.text section。所以我们为什么会看不到.fixup section。

让我们看一下(0xc01072fa, 0xc01072fd)到底指向什么代码。

```
$objdump –d/boot/vmlinux-2.4.18-14

...

c0107230 <machine_power_off>:

c0107230:    a1 c8 ee 37 c0       mov    0xc037eec8,%eax

c0107235:    85 c0                test   %eax,%eax

c0107237:    75 07                jne                          c0107240
<machine_power_off+0x10>

c0107239:    c3                   ret

c010723a:    8d b6 00 00 00 00    lea    0x0(%esi),%esi

c0107240:    ff d0                call   *%eax

c0107242:    eb f5                jmp                          c0107239
<machine_power_off+0x9>

c0107244:    8d b6 00 00 00 00    lea    0x0(%esi),%esi

c010724a:    8d bf 00 00 00 00    lea    0x0(%edi),%edi
```

```
c0107250 <show_regs>:
c0107250:   56                      push   %esi
c0107251:   31 f6                   xor    %esi,%esi
c0107253:   53                      push   %ebx
c0107254:   83 ec 14                sub    $0x14,%esp
c0107257:   8b 5c 24 20             mov    0x20(%esp,1),%ebx
c010725b:   8b 43 28                mov    0x28(%ebx),%eax
c010725e:   c7 44 24 08 00 02 00 movl   $0x200,0x8(%esp,1)
c0107265:   00
c0107266:   c7 44 24 04 c0 ec 37 movl   $0xc037ecc0,0x4(%esp,1)
c010726d:   c0
c010726e:   89 04 24                mov    %eax,(%esp,1)
c0107271:   e8 7a 19 02 00          call   c0128bf0 <lookup_symbol>
c0107276:   c7 04 24 f5 ba 26 c0 movl   $0xc026baf5,(%esp,1)
c010727d:   e8 5e 3d 01 00          call   c011afe0 <printk>
c0107282:   b8 00 e0 ff ff          mov    $0xffffe000,%eax
c0107287:   21 e0                   and    %esp,%eax
c0107289:   c7 04 24 68 3a 25 c0 movl   $0xc0253a68,(%esp,1)
c0107290:   8d 90 3e 02 00 00       lea    0x23e(%eax),%edx
c0107296:   8b 40 78                mov    0x78(%eax),%eax
c0107299:   89 54 24 08             mov    %edx,0x8(%esp,1)
c010729d:   89 44 24 04             mov    %eax,0x4(%esp,1)
c01072a1:   e8 3a 3d 01 00          call   c011afe0 <printk>
c01072a6:   c7 04 24 7d 3a 25 c0 movl   $0xc0253a7d,(%esp,1)
c01072ad:   0f b7 43 2c             movzwl 0x2c(%ebx),%eax
c01072b1:   89 44 24 04             mov    %eax,0x4(%esp,1)
c01072b5:   8b 43 28                mov    0x28(%ebx),%eax
c01072b8:   c7 44 24 0c 00 00 00 movl   $0x0,0xc(%esp,1)
c01072bf:   00
c01072c0:   89 44 24 08             mov    %eax,0x8(%esp,1)
c01072c4:   e8 17 3d 01 00          call   c011afe0 <printk>
c01072c9:   c7 04 24 99 3a 25 c0 movl   $0xc0253a99,(%esp,1)
```

```
c01072d0:   c7 44 24 04 c0 ec 37 movl   $0xc037ecc0,0x4(%esp,1)
c01072d7:   c0
c01072d8:   e8 03 3d 01 00       call   c011afe0 <printk>
c01072dd:   f6 43 2c 03          testb  $0x3,0x2c(%ebx)
c01072e1:   0f 85 b9 00 00 00    jne    c01073a0 <show_regs+0x150>
c01072e7:   e8 74 36 01 00       call   c011a960 <print_tainted>
c01072ec:   c7 04 24 b4 3a 25 c0 movl   $0xc0253ab4,(%esp,1)
c01072f3:   8b 53 30             mov    0x30(%ebx),%edx
c01072f6:   89 44 24 08          mov    %eax,0x8(%esp,1)
c01072fa:   0f 20 e6             mov    %cr4,%esi
c01072fd:   89 54 24 04          mov    %edx,0x4(%esp,1)
c0107301:   e8 da 3c 01 00       call   c011afe0 <printk>
c0107306:   c7 04 24 80 4f 25 c0 movl   $0xc0254f80,(%esp,1)
c010730d:   8b 43 18             mov    0x18(%ebx),%eax
c0107310:   89 44 24 04          mov    %eax,0x4(%esp,1)
c0107314:   8b 03                mov    (%ebx),%eax
…
```

(0xc01072fa, 0xc01072fd)为于 show_regs（）

```
c01072fa:   0f 20 e6             mov    %cr4,%esi
c01072fd:   89 54 24 04          mov    %edx,0x4(%esp,1)
```

反正有原码，查一下吧

在 ***linux/arch/i386/kernel/process.c*** 中

```
void show_regs(struct pt_regs * regs)
{
  unsigned long cr0 = 0L, cr2 = 0L, cr3 = 0L, cr4 = 0L;


  printk("\n");
  printk("Pid: %d, comm: %20s\n", current->pid, current->comm);
  printk("EIP: %04x:[<%08lx>] CPU: %d\n",0xffff & regs->xcs,regs->eip,
smp_processor_id());
  print_symbol("EIP is at %s\n", regs->eip);


  if (user_mode_vm(regs))
```

```
        printk(" ESP: %04x:%08lx",0xffff & regs->xss,regs->esp);
    printk(" EFLAGS: %08lx    %s  (%s %.*s)\n",
          regs->eflags, print_tainted(), init_utsname()->release,
          (int)strcspn(init_utsname()->version, " "),
          init_utsname()->version);
    printk("EAX: %08lx EBX: %08lx ECX: %08lx EDX: %08lx\n",
        regs->eax,regs->ebx,regs->ecx,regs->edx);
    printk("ESI: %08lx EDI: %08lx EBP: %08lx",
        regs->esi, regs->edi, regs->ebp);
    printk(" DS: %04x ES: %04x GS: %04x\n",
          0xffff & regs->xds,0xffff & regs->xes, 0xffff & regs->xgs);

    cr0 = read_cr0();
    cr2 = read_cr2();
    cr3 = read_cr3();
    cr4 = read_cr4_safe();   这一行是宏
    printk("CR0: %08lx CR2: %08lx CR3: %08lx CR4: %08lx\n", cr0, cr2, cr3,
cr4);
    show_trace(NULL, regs, &regs->esp);
  }

  #define read_cr4_safe() ({            \
    unsigned int __dummy;               \
    /* This could fault if %cr4 does not exist */ \
    __asm__("1: movl %%cr4, %0      \n"   \
        "2:           \n"   \
        ".section __ex_table,\"a\"  \n"   \
        ".long 1b,2b         \n"   \
        ".previous       \n"   \
        : "=r" (__dummy): "0" (0));      \
    __dummy;                  \
  })
```

read_cr4_safe()宏中的注释已经说明问题了，即如果当前运行的 CPU 没有 cr4 寄存器，则 mov %cr4,%esi 当然会出错（我想应该是非法指令异常吧），而该异常处理器只是跳过产生异常的指令。

整个__ex_table section 就是由异常处理地址对构成的表。这个表在 Linux 内核编译期间生成。在系统初始化时，该表被载入内存，并由两个全局符号来表示该表的头与尾（__start___ex_table 和__stop___ex_table）。当异常发生时，内核用产生异常的代码地址作为关键字来搜索该表，以得到对应的异常处理器的地址。具体代码如下：

linux/arch/i386/mm/extable.c

```
unsigned long
search_exception_table(unsigned long addr)
{
  unsigned long ret = 0;

#ifndef CONFIG_MODULES
  /* There is only the kernel to search.  */
  ret = search_one_table(__start___ex_table, __stop___ex_table-1, addr);
  return ret;
#else
  unsigned long flags;
  /* The kernel is the last "module" -- no need to treat it special. */
  struct module *mp;

  spin_lock_irqsave(&modlist_lock, flags);
  for (mp = module_list; mp != NULL; mp = mp->next) {
      if (mp->ex_table_start == NULL || !(mp->flags&(MOD_RUNNING|MOD_INITIALIZING)))
          continue;
      ret = search_one_table(mp->ex_table_start,
                  mp->ex_table_end - 1, addr);
      if (ret)
          break;
```

```
  }
  spin_unlock_irqrestore(&modlist_lock, flags);
  return ret;
#endif
}
对__ex_table 进行二分查找
static inline unsigned long
search_one_table(const struct exception_table_entry *first,
       const struct exception_table_entry *last,
       unsigned long value)
{
       while (first <= last) {
      const struct exception_table_entry *mid;
      long diff;

      mid = (last - first) / 2 + first;
      diff = mid->insn - value;
              if (diff == 0)
                     return mid->fixup;
              else if (diff < 0)
                     first = mid+1;
              else
                     last = mid-1;
       }
       return 0;
}
```

我们搞清楚了异常处理的数据结构和怎样由产生异常的地址跳转到对应的异常处理器，但还有关键的一点没搞清楚。内核是怎么感知异常的？

异常处理（exception handing）的“异常”，在各个 CPU 平台上各有差异，但大致上是相同的。比如访问不存在的内存，写只读的内存，除零错，非法指令错，等等。只有 CPU 理解的异常，而 OS 又关心的异常才能被捕捉，除此之外，捕捉都不能，更遑论处理。在内核的核心代码里有如下代码：

arch/i386/kernel/traps.c

系统通过如下代码为 CPU 安装了“异常“处理器 ———— 这里的异常是 CPU 的异常，第一级异常。

```
DO_VM86_ERROR_INFO( 0, SIGFPE,  "divide error", divide_error, FPE_INTDIV, regs->eip)
DO_VM86_ERROR( 3, SIGTRAP, "int3", int3)
DO_VM86_ERROR( 4, SIGSEGV, "overflow", overflow)
DO_VM86_ERROR( 5, SIGSEGV, "bounds", bounds)
DO_ERROR_INFO( 6, SIGILL,  "invalid operand", invalid_op, ILL_ILLOPN, regs->eip)
DO_VM86_ERROR( 7, SIGSEGV, "device not available", device_not_available)
DO_ERROR( 8, SIGSEGV, "double fault", double_fault)
DO_ERROR( 9, SIGFPE,  "coprocessor segment overrun", coprocessor_segment_overrun)
DO_ERROR(10, SIGSEGV, "invalid TSS", invalid_TSS)

DO_ERROR(11, SIGBUS,  "segment not present", segment_not_present)
DO_ERROR(12, SIGBUS,  "stack segment", stack_segment)
DO_ERROR_INFO(17, SIGBUS, "alignment check", alignment_check, BUS_ADRALN, get_cr2())
```

比如：

```
DO_ERROR_INFO( 6, SIGILL,  "invalid operand", invalid_op, ILL_ILLOPN, regs->eip)
```

即当 CPU 的指令处理单元执行一条非法指令（它不认识的指令）时，它即会产生 exception, 然后从 CPU 与内核约定好的表中找到该异常的处理器（也就是这里的 invalid_op）。注意这里的异常处理器是指最顶层的 exception hander，不是本文主要探讨的 exception handler。在这些系统的异常处理器中有能探知我们关心和探讨的 exception 的“传感器”。内核在很多系统异常处理器中都安装了“传感器”。比如：

```
asmlinkage void do_general_protection(struct pt_regs * regs, long error_code)
{
  if (regs->eflags & VM_MASK)
      goto gp_in_vm86;

  if (!(regs->xcs & 3))
      goto gp_in_kernel;

  current->thread.error_code = error_code;
```

```
  current->thread.trap_no = 13;
  force_sig(SIGSEGV, current);
  return;

gp_in_vm86:
  handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
  return;

gp_in_kernel:
  {
      unsigned long fixup;
      fixup = search_exception_table(regs->eip);    “传感器”
      if (fixup) {
          regs->eip = fixup;
          return;
      }
      die("general protection fault", regs, error_code);
  }
}
```

```
static void inline do_trap(int trapnr, int signr, char *str, int vm86,
           struct pt_regs * regs, long error_code, siginfo_t *info)
{
  if (regs->eflags & VM_MASK) {
      if (vm86)
          goto vm86_trap;
      else
          goto trap_signal;
  }

  if (!(regs->xcs & 3))
```

```
        goto kernel_trap;

    trap_signal: {
        struct task_struct *tsk = current;
        tsk->thread.error_code = error_code;
        tsk->thread.trap_no = trapnr;
        if (info)
            force_sig_info(signr, info, tsk);
        else
            force_sig(signr, tsk);
        return;
    }

    kernel_trap: {
        unsigned long fixup = search_exception_table(regs->eip);
    “传感器”
        if (fixup)
            regs->eip = fixup;
        else
            die(str, regs, error_code);
        return;
    }

    vm86_trap: {
        int ret = handle_vm86_trap((struct kernel_vm86_regs *) regs,
error_code, trapnr);
        if (ret) goto trap_signal;
        return;
    }
  }
```

这里还是用一个例子来说明整个流程吧。就以前述例二来举例：

```
int *pIntValue = 0;


__asm__ __volatile__(
"1:   movl $1, (%eax)\n\t"
"movl $1, %%eax\n\t"          作为返回值，令%eax = 1,即 return true
"2:   nop\n\t"
".section .fixup, \"ax\"\n\t"
"3: xorl %%eax, %%eax\n\t"    作为返回值，令%eax = 0,即 return false
"jmp 2b\n\t"
".previous\n\t"
".section __ex_table, \"a\"\n\t"
".align 4\n\t"
".long 1b, 3b\n\t"
".previous\n\t"
:
:"a"(pIntValue)
);
```

1. 1: movl $1, (%eax) 即 *pIntValue = 1;
由于 pIntValue 为 NULL 指针，自然该指令会引发内存访问违规，从而抛出 x86 CPU 的 trap 0Eh 异常。

2. CPU 跳转到如下代码：

```
arch/i386/kernel/entry.S


ENTRY(page_fault)             这是 CPU 出现内存异常后的最初始入口
  pushl $ SYMBOL_NAME(do_page_fault)
  jmp error_code


  ...


error_code:
  pushl %ds
  pushl %eax
```

```
    xorl %eax,%eax
    pushl %ebp
    pushl %edi
    pushl %esi
    pushl %edx
    decl %eax          # eax = -1
    pushl %ecx
    pushl %ebx
    cld
    movl %es,%ecx
    movl ORIG_EAX(%esp), %esi   # get the error code
    movl ES(%esp), %edi         # get the function address
    movl %eax, ORIG_EAX(%esp)
    movl %ecx, ES(%esp)
    movl %esp,%edx
    pushl %esi         # push the error code
    pushl %edx         # push the pt_regs pointer
    movl $(__KERNEL_DS),%edx
    movl %edx,%ds
    movl %edx,%es
    GET_CURRENT(%ebx)
    call *%edi         调用 do_page_fault（）
    addl $8,%esp
从 do_page_fault（）返回，这时通过 search_exception_table（）已经找到 fixup 代码，regs->eip = fixup，在下面的 RESTORE_ALL 宏中，regs 结构中的各个 register 将被恢复给 CPU，而 CPU 的 eip 则会被设置成 fixup。
    jmp ret_from_exception
```

call *%edi 即是调用在 arch/i386/mm/fault.c 文件中的

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
```

而在 call *%edi 前的那么多 push 指令，都在为该 C 函数准备调用参数，也就是 struct pt_regs *regs。

3. 进入真正内存 exception 处理的核心函数

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long
error_code)
  {
    struct task_struct *tsk;
    struct mm_struct *mm;
    struct vm_area_struct * vma;
    unsigned long address;
    unsigned long page;
    unsigned long fixup;
    int write;
    siginfo_t info;

    /* get the address */
    __asm__("movl %%cr2,%0":"=r" (address));

    /* It's safe to allow irq's after cr2 has been saved */
    if (regs->eflags & X86_EFLAGS_IF)
        local_irq_enable();

    tsk = current;

    /*
     * We fault-in kernel-space virtual memory on-demand. The
     * 'reference' page table is init_mm.pgd.
     *
     * NOTE! We MUST NOT take any locks for this case. We may
     * be in an interrupt or a critical region, and should
     * only copy the information from the master page table,
     * nothing more.
     *
     * This verifies that the fault happens in kernel space
     * (error_code & 4) == 0, and that the fault was not a
```

```
     * protection error (error_code & 1) == 0.
     */
    if (address >= TASK_SIZE && !(error_code & 5))
        goto vmalloc_fault;

    mm = tsk->mm;
    info.si_code = SEGV_MAPERR;

    /*
     * If we're in an interrupt or have no user
     * context, we must not take the fault..
     */
    if (in_interrupt() || !mm)
        goto no_context;

    down_read(&mm->mmap_sem);

    vma = find_vma(mm, address);
    if (!vma)
        goto bad_area;
    if (vma->vm_start <= address)
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))
        goto bad_area;
    if (error_code & 4) {
        /*
         * accessing the stack below %esp is always a bug.
         * The "+ 32" is there due to some instructions (like
         * pusha) doing post-decrement on the stack and that
         * doesn't show up until later..
         */
        if (address + 32 < regs->esp)
```

```
        goto bad_area;
  }
  if (expand_stack(vma, address))
      goto bad_area;
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:
  info.si_code = SEGV_ACCERR;
  write = 0;
  switch (error_code & 3) {
      default:   /* 3: write, present */
#ifdef TEST_VERIFY_AREA
        if (regs->cs == KERNEL_CS)
            printk("WP fault at %08lx\n", regs->eip);
#endif
        /* fall through */
     case 2:       /* write, not present */
        if (!(vma->vm_flags & VM_WRITE))
            goto bad_area;
        write++;
        break;
     case 1:       /* read, present */
        goto bad_area;
     case 0:       /* read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
            goto bad_area;
  }

 survive:
  /*
```

```
     * If for any reason at all we couldn't handle the fault,
     * make sure we exit gracefully rather than endlessly redo
     * the fault.
     */
    switch (handle_mm_fault(mm, vma, address, write)) {
    case 1:
        tsk->min_flt++;
        break;
    case 2:
        tsk->maj_flt++;
        break;
    case 0:
        goto do_sigbus;
    default:
        goto out_of_memory;
    }

    /*
     * Did it hit the DOS screen memory VA from vm86 mode?
     */
    if (regs->eflags & VM_MASK) {
        unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
        if (bit < 32)
            tsk->thread.screen_bitmap |= 1 << bit;
    }
    up_read(&mm->mmap_sem);
    return;

/*
 * Something tried to access memory that isn't in our memory map..
 * Fix it, but check if it's kernel or user first..
 */
```

```
bad_area:
  up_read(&mm->mmap_sem);

  /* User mode accesses just cause a SIGSEGV */
  if (error_code & 4) {
      tsk->thread.cr2 = address;
      tsk->thread.error_code = error_code;
      tsk->thread.trap_no = 14;
      info.si_signo = SIGSEGV;
      info.si_errno = 0;
      /* info.si_code has been set above */
      info.si_addr = (void *)address;
      force_sig_info(SIGSEGV, &info, tsk);
      return;
  }

  /*
   * Pentium F0 0F C7 C8 bug workaround.
   */
  if (boot_cpu_data.f00f_bug) {
      unsigned long nr;

      nr = (address - idt) >> 3;

      if (nr == 6) {
          do_invalid_op(regs, 0);
          return;
      }
  }

no_context:
  /* Are we prepared to handle this kernel fault?  */
```

```
    if ((fixup = search_exception_table(regs->eip)) != 0) {  ②
        regs->eip = fixup;
        return;
    }

...

}
```

① /* get the address */
__asm__("movl %%cr2,%0":"=r" (address));
该处的嵌入汇编即获得访问违例时的代码地址，也就是 1: movl $1, (%eax)中的标号“1”。

②处的 regs->eip 即是 address，用它来在_ex_table 表中搜索对应的处理器地址。返回值 fixup 自然应该是下面代码中的标号“3”。

".section .fixup, \"ax\"\n\t"
"3: xorl %%eax, %%eax\n\t" 作为返回值，令%eax = 0,即 return false
"jmp 2b\n\t"

使得 regs->eip 指向我们定义的异常处理器。

4. 这是从 do_page_fault（）函数返回

```
    call *%edi          调用 do_page_fault（）
    addl $8,%esp
    jmp ret_from_exception

    ...

ret_from_exception:
    movl EFLAGS(%esp),%eax      # mix EFLAGS and CS
    movb CS(%esp),%al
    testl $(VM_MASK | 3),%eax   # return to VM86 mode or non-supervisor?
    jne ret_from_sys_call
    jmp restore_all

restore_all:
```

```
    RESTORE_ALL



  #define RESTORE_ALL\
    popl %ebx; \
    popl %ecx; \
    popl %edx; \
    popl %esi; \
    popl %edi; \
    popl %ebp; \
    popl %eax; \
1:  popl %ds;  \
2:  popl %es;  \
    addl $4,%esp; \
3:  iret;      \
```

从上面的代码可知出了 do_page_fault（）后，经过几个跳转，执行 RESTORE_ALL 宏，也就是恢复被记录在 regs 中的发生异常时的现场。关键是这时的现场中的 eip 已经被修改过了，由原始的指向

1: movl $1, (%eax)

变成了

3: xorl %%eax, %%eax

完成了从异常代码跳转到给该异常的处理代码。

NSEH在内核模块编程中的应用

上面分析了内核本身是支持 exception handling 的，那么对于开发 Linux 设备驱动程序的 programmer 而言，是否支持呢？从一个函数就可看出是支持的。

linux/arch/i386/mm/extable.c

```
unsigned long
search_exception_table(unsigned long addr)
{
   unsigned long ret = 0;

#ifndef CONFIG_MODULES
   /* There is only the kernel to search.  */
   ret = search_one_table(__start___ex_table, __stop___ex_table-1, addr);
   return ret;
#else
   unsigned long flags;
   /* The kernel is the last "module" -- no need to treat it special.  */
   struct module *mp;

   spin_lock_irqsave(&modlist_lock, flags);
   for (mp = module_list; mp != NULL; mp = mp->next) {
        if (mp->ex_table_start == NULL || !(mp->flags&(MOD_RUNNING|MOD_INITIALIZING)))
            continue;
        ret = search_one_table(mp->ex_table_start,
                       mp->ex_table_end - 1, addr);
        if (ret)
            break;
   }
   spin_unlock_irqrestore(&modlist_lock, flags);
   return ret;
```

```
#endif

}
```

上面的预编译宏 CONFIG_MODULES 可以用来控制 Linux 内核是否支持动态载入内核模块（Kernel Module）－－－ Linux 设备驱动程序的最通常方式。该函数的下面标红部分即是用来支持 Kernel Module 中的异常处理的。

下面举个例子

```
$cat exception/exception.c
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Walter Zhou");

static int __init generate_exception_init(void)
{
  unsigned int temp = 0;
  unsigned int *ptemp = &temp;

  __asm__ __volatile__(
  "xorl %%eax, %%eax\n\t"
  "1: movl $1, (%%eax)\n\t"
  ".section .fixup, \"ax\"\n\t"
  "2: movl %0, %%eax\n\t"
  "jmp 1b\n\t"
  ".previous\n\t"
  ".section __ex_table, \"a\"\n\t"
  ".align 4\n\t"
  ".long 1b, 2b\n\t"
  ".previous\n\t"
  :
  :"m"(ptemp)
```

```
    :"%eax", "%ecx"
    );

    printk(KERN_ALERT "If you could see the message, one read NULL pointer
in kernel mode exception has been handled!\n");
    printk(KERN_ALERT "The temp = %d\n", temp);
    return 0;
  }

  static void __exit generate_exception_exit(void)
  {
    printk(KERN_ALERT "exit generate_exception module\n");
  }

  module_init(generate_exception_init);
  module_exit(generate_exception_exit);
```

上面的例子代码用 Window 下的 SEH 来重写的话，同如下 c/c++代码等价。

```
  unsigned int temp = 0;
  unsigned int *ptemp = &temp;

  unsigned int *null = 0;
  __try
  {
    *null = 1;        ①
  }
  __except(EXCEPTION_CONTINUE_EXECUTION)
  {
    null = ptemp;     在异常处理器中使得 null 指向合法空间
  }

  printk(KERN_ALERT "If you could see the message, one read NULL pointer
in kernel mode exception has been handled!\n");
  printk(KERN_ALERT "The temp = %d\n", temp); temp 变量被改写成第①行改写
```

```
成 1
  return   0;
```

```
$cat exception/Makefile
```

```
WARN     := -W -Wall -Wstrict-prototypes -Wmissing-prototypes
INCLUDE  := -isystem /usr/src/linux-`uname -r`/include
CFLAGS   := -O2 -DMODULE -D__KERNEL__ ${WARN} ${INCLUDE}
CC       := gcc
OBJS     := ${patsubst %.c, %.o, ${wildcard *.c}}

all: ${OBJS}

.PHONY:  clean

clean:
  rm -rf *.o
```

```
$readelf –S exception/exception.o
There are 16 section headers, starting at offset 0x238:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00004a 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000680 000030 08     14   1  4
  [ 3] .data             PROGBITS        00000000 000080 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000080 000000 00  WA  0   0  4
  [ 5] .modinfo          PROGBITS        00000000 000080 000035 00   A  0   0  1
  [ 6] .rodata.str1.32   PROGBITS        00000000 0000c0 0000a3 01 AMS  0   0 32
  [ 7] .rodata.str1.1    PROGBITS        00000000 000163 000012 01 AMS  0   0  1
  [ 8] .fixup            PROGBITS        00000000 000175 000008 00  AX  0   0  1
  [ 9] .rel.fixup        REL             00000000 0006b0 000008 08     14   8  4
```

```
  [10] __ex_table       PROGBITS        00000000 000180 000008 00   A  0   0  4
  [11] .rel__ex_table   REL             00000000 0006b8 000010 08     14   a  4
  [12] .comment         PROGBITS        00000000 000188 000033 00      0   0  1
  [13] .shstrtab        STRTAB          00000000 0001bb 00007b 00      0   0  1
  [14] .symtab          SYMTAB          00000000 0004b8 000130 10     15   f  4
  [15] .strtab          STRTAB          00000000 0005e8 000098 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

```
$objdump –d exception/exception.o

exception.o:     file format elf32-i386


Disassembly of section .text:


00000000 <init_module>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8d 45 fc                lea    0xfffffffc(%ebp),%eax
   6:   83 ec 14                sub    $0x14,%esp
   9:   c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
  10:   89 45 f8                mov    %eax,0xfffffff8(%ebp)
  13:   31 c0                   xor    %eax,%eax
  15:   c7 00 01 00 00 00       movl   $0x1,(%eax)     对 NULL 指针赋值 1，会触发异常
  1b:   68 00 00 00 00          push   $0x0
  20:   e8 fc ff ff ff          call   21 <init_module+0x21>
  25:   58                      pop    %eax
  26:   5a                      pop    %edx
  27:   ff 75 fc                pushl  0xfffffffc(%ebp)
  2a:   68 00 00 00 00          push   $0x0
  2f:   e8 fc ff ff ff          call   30 <init_module+0x30>
```

```
  34:   31 c0                   xor    %eax,%eax
  36:   c9                      leave
  37:   c3                      ret


00000038 <cleanup_module>:
  38:   55                      push   %ebp
  39:   89 e5                   mov    %esp,%ebp
  3b:   83 ec 14                sub    $0x14,%esp
  3e:   68 80 00 00 00          push   $0x80
  43:   e8 fc ff ff ff          call   44 <cleanup_module+0xc>
  48:   c9                      leave
  49:   c3                      ret
Disassembly of section .fixup:


00000000 <.fixup>:              异常处理器代码在.fixup section
   0:   8b 45 f8                mov    0xfffffff8(%ebp),%eax
   3:   e9 11 00 00 00          jmp    19 <__module_license+0x3>
```

运行的结果：

```
#/sbin/insmod exception/exception.o

# If you could see the message, one read NULL pointer in kernel mode exception has been handled!

The temp = 1
```

你或许想知道，如果在 Kernel Mode 没有上面的异常处理器，那会怎么样呢？下面就是你会看到的所谓 Kernel Panic 画面，类似于你肯定在 Windows 下碰到过的蓝屏死机。

```
EIP:    0010:[<d0889065>]    Tainted: PF
EFLAGS: 00010246
EIP is at init_module+0x5/0x10 [exception]
eax: 00000000   ebx: fffffffea   ecx: c0365390   edx: d0889000
esi: 00000000   edi: d088906a   ebp: ce3fdf24   esp: ce3fdf24
ds: 0018   es: 0018   ss: 0018
Process insmod (pid: 454, stackpage=ce3fd000)
Stack: ce3fdfbc c01171d2 d0889060 0806cdf0 00000268 00000060 00000009 cf287ab4
       d0889000 ce852000 ce853000 00000060 d087e000 d0889060 000002c8 00000000
       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
Call Trace:
 [<c01171d2>] sys_init_module+0x56c/0x60e [kernel]
 [<d0889060>] init_module+0x0/0x10 [exception]
 [<d0889060>] init_module+0x0/0x10 [exception]
 [<c0108de7>] system_call+0x33/0x38 [kernel]

Code: c7 00 01 00 00 00 c9 31 c0 c3 90 55 89 e5 c9 c3 65 78 63 65

Entering kdb (current=0xce3fc000, pid 454) Oops: Oops
due to oops @ 0xd0889065
eax = 0x00000000 ebx = 0xfffffffea ecx = 0xc0365390 edx = 0xd0889000
esi = 0x00000000 edi = 0xd088906a esp = 0xce3fdf24 eip = 0xd0889065
ebp = 0xce3fdf24 xss = 0x00000018 xcs = 0x00000010 eflags = 0x00010246
xds = 0x00000018 xes = 0x00000018 origeax = 0xffffffff &regs = 0xce3fdef0
kdb> _
```

总结

由于没有 gcc 的支援，现在 Linux exception handling 对 programmer 而言是非常不友好的。对极少数 Linux 内核的开发者而言，这不成问题。但相对于数量庞大的设备驱动开发者而言，显然很有可以改善的余地。毕竟 gcc 的嵌入式汇编实在是“汇编中的汇编”。 或许 Linux 内核有关异常处理的部分在不久的将来会有所改善，毕竟 Linux 是以善于学习著称的。Linux 的内核开发者们虽然或许有点瞧不起 Microsoft，但勿容置疑，Windows 的 SEH 架构实在很值得 Linux 学习的。

后记

此篇文章是根据我几年前的笔记整理而成，所以参考的主流 Linux 版本是 **2.4.20**。为了解当前 Linux 内核中 exception handling 的最新状况，我看了当前最新内核版本(2.6.20),发觉我这篇整理自几年前读 Linux source 的笔记而成的文章还有价值。整个异常处理机制几乎没变。唉，真是遗憾！

Walter Zhou
mailto:z-l-dragon@hotmail.com