Segmentation Fault in Linux

—— 原因与避免

Author: ZX_WING(xing5820@163.com)

写在前面的话

最近 CU（chinaunix）出现了很多问 segmentation fault 的帖子，其实这也是个“月经贴”了，泡 CU 几年，每个月都有人问。为了减少重复回帖，笔者结合自己的经验，总结了 SIGSEGV 在 Linux 中产生的机理，并用实际例子概括哪些编程错误容易引发 SIGSEGV。由于本人经验有限，文中难免有疏漏和错误，请发现的朋友发信到 xing5820@163.com 指正，笔者好即使修改。

内容提要

本文简单介绍了 Segmentation fault 发生的原因，结合实际例子描述了内核向用户态程序发送 SIGSEGV 信号的流程。文中以实例回答了常见的一些 SIGSEGV 问题，例如“为什么函数返回了栈还可以访问？”、“为什么 free()后的内存仍然可以使用”、“为什么我遇到的是 SIGSEGV 而不是 SIGILL 信号”等。最后笔者结合自己的经验，列举了一些预防 SIGSEGV 的编程习惯，供大家参考。SIGSEGV 严格依赖操作系统、编译器、硬件平台，本文基于 Linux、GCC、32bit IA32 架构，但对其他平台操作系统也有借鉴意义。

Revision History

日期	版本	描述
2009.12.21	1.0	初次发表版本

1.什么是“Segmentation fault in Linux”？

我们引用 wiki 上的一段话来回答这个问题。

*A **segmentation fault** (often shortened to **SIGSEGV**) is a particular error condition that can occur during the operation of computer software. A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a read-only location, or to overwrite part of the operating system).*

Segmentation is one approach to memory management and protection in the operating system. It has been superseded by paging for most purposes, but much of the terminology of segmentation is still used, "segmentation fault" being an example. Some operating systems still have segmentation at some logical level although paging is used as the main memory management policy.

On Unix-like operating systems, a process that accesses an invalid memory address receives the SIGSEGV signal. On Microsoft Windows, a process that accesses invalid memory receives the STATUS_ACCESS_VIOLATION exception.

上述文字没有给出 SIGSEGV 的定义，仅仅说它是“计算机软件操作过程中的一种错误情况”。文字描述了 SIGSEGV 在何时发生，即“当程序试图访问不被允许访问的内存区域（比如，尝试写一块属于操作系统的内存），或以错误的类型访问内存区域（比如，尝试写一块只读内存）。这个描述是准确的。为了加深理解，我们再更加详细的概括一下 SIGSEGV。

- SIGSEGV 是在访问内存时发生的错误，它属于内存管理的范畴
- SIGSEGV 是一个用户态的概念，是操作系统在用户态程序错误访问内存时所做出的处理。
- 当用户态程序访问（访问表示读、写或执行）不允许访问的内存时，产生 SIGSEGV。
- 当用户态程序以错误的方式访问允许访问的内存时，产生 SIGSEGV。

从用户态程序开发的角度，我们并不需要理解操作系统复杂的内存管理机制，这是和硬件平台相关的。但是，了解内核发送 SIGSEGV 信号的流程，对我们理解 SIGSEGV 是很有帮助的。在《Understanding Linux Kernel Edition 3》和《Understanding the Linux Virtual Memory Manager》相关章节都有一幅总图对此描述，对比之下，笔者认为 ULK 的图更为直观。

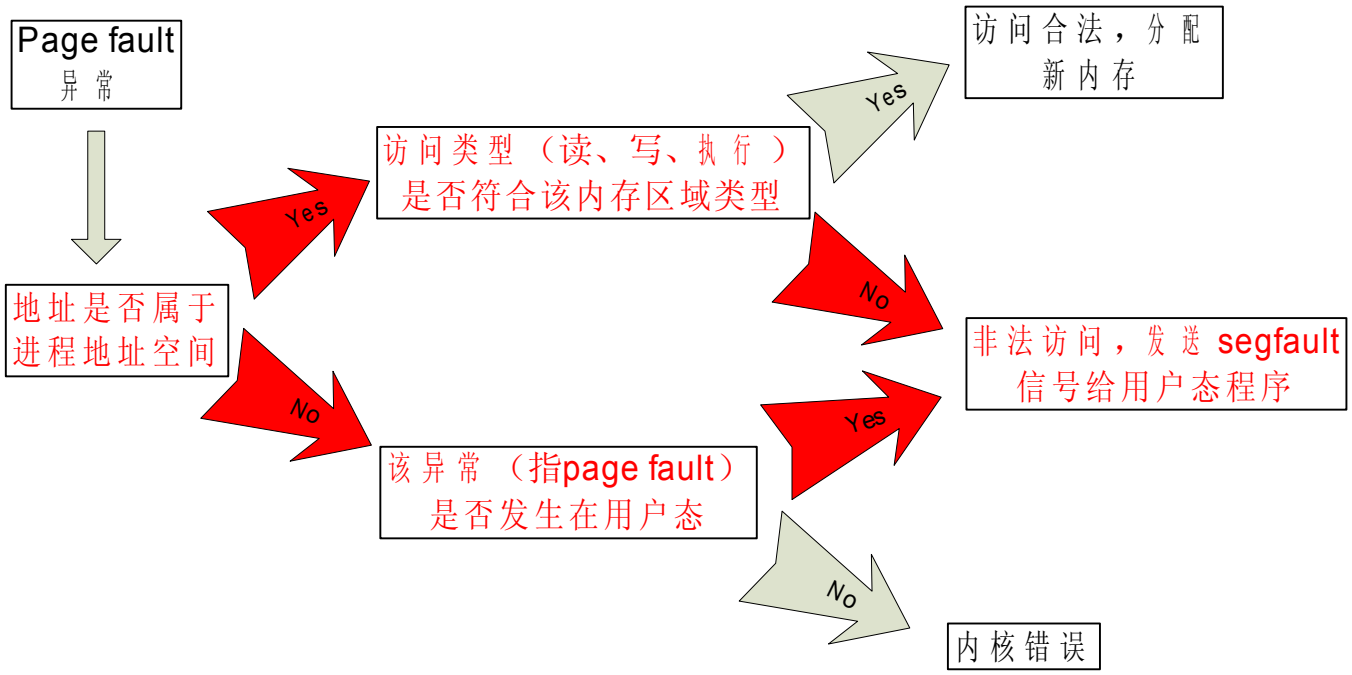

图 1. SIGSEGV Overview

图 1 红色部分展示了内核发送 SIGSEGV 信号给用户态程序的总体流程。当用户态程序访问一个会引发 SIGSEGV 的地址时，硬件首先产生一个 page fault，即“缺页异常”。在内核的 page fault 处理函数中，首先判断该地址是否属于用户态程序的地址空间[*]。以 Intel 的 32bit IA32 架构的 CPU 为例，用户态程序的地址空间为[0，3G]，内核地址空间为[3G，4G]。如果该地址属于用户态地址空间，检查访问的类型是否和该内存区域的类型是否匹配，不匹配，则发送 SIGSEGV 信号；如果该地址不属于用户态地址空间，检查访问该地址的操作是否发生在用户态，如果是，发送 SIGSEGV 信号。

[*]这里的用户态程序地址空间，特指程序可以访问的地址空间范围。如果广义的说，一个进程的地址空间应该包括内核空间部分，只是它不能访问而已。

图 2 更为详细的描绘了内核发送 SIGSEGV 信号的流程。在这里我们不再累述图中流程，在后面章节的例子中，笔者会结合实际，描述具体的流程。

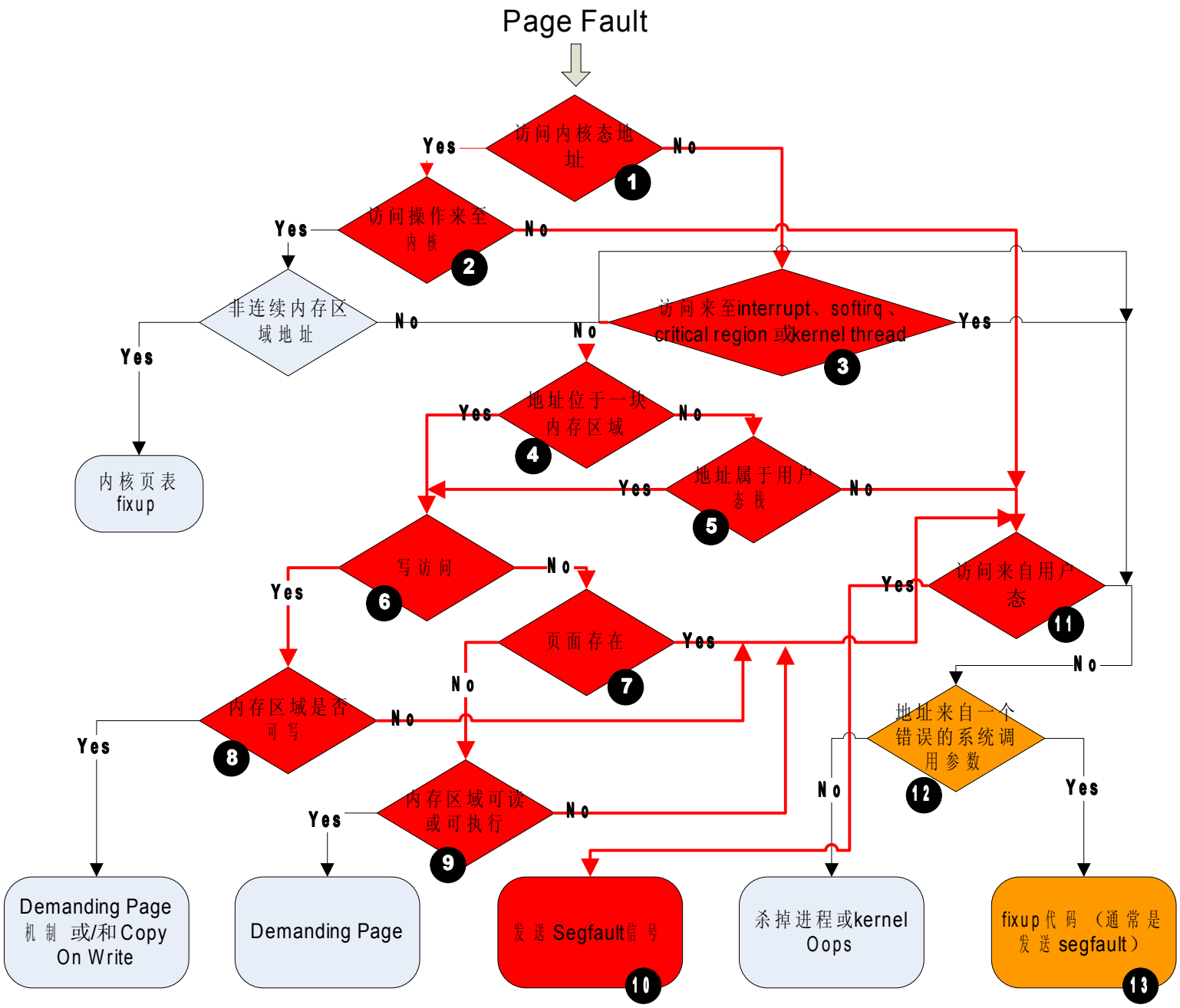

图 2 SIGSEGV detailed flow

2.指针越界和 SIGSEGV

经常看到有帖子把两者混淆，而这两者的关系也确实微妙。在此，我们把指针运算（加减）引起的越界、野指针、空指针都归为指针越界。SIGSEGV 在很多时候是由于指针越界引起的，但并不是所有的指针越界都会引发 SIGSEGV。一个越界的指针，如果不解引用它，是不会引起 SIGSEGV 的。而即使解引用了一个越界的指针，也不一定会引起 SIGSEGV。这听上去让人发疯，而实际情况确实如此。SIGSEGV 涉及到操作系统、C 库、编译器、链接器各方面的内容，我们以一些具体的例子来说明。

2.1 错误的访问类型引起的 SIGSEGV

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5         char* s = "hello world";
6
7         s[1] = 'H';
8 }
```

这是最常见的一个例子。想当年俺对 C 语言懵懂的时候，也在校内的 BBS 上发帖问过，当时还以为这是指针和数组的区别。此例中，”hello world”作为一个常量字符串，在编译后会被放在.rodata 节（GCC），最后链接生成目标程序时.rodata 节会被合并到 text segment 与代码段放在一起，故其所处内存区域是只读的。这就是错误的访问类型引起的 SIGSEGV。

其在图 2 中的顺序为：

1 -→ 3 -→ 4 -→ 6 -→ 8 -→ 11 -→10

2.2 访问了不属于进程地址空间的内存

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5         int* p = (int*)0xC0000fff;
6
7         *p = 10;
8 }
```

在这个例子中，我们访问了一个属于内核的地址（IA32，32bit）。当然，很少会有人这样写程序，但你的程序可能在不经意的情况下做出这样的行为（这个不经意的行为在后面讨论）。此例在图 2 的流程：

1 -→ 2 -→ 11 -→ 10

2.3 访问了不存在的内存

最常见的情况不外乎解引用空指针了，如：

```
1 #include <stdio.h>
```

```
 2 #include <stdlib.h>
 3
 4 int main () {
 5         int *a = NULL;
 6
 7         *a = 1;
 8 }
```

在实际情况中，此例中的空指针可能指向用户态地址空间，但其所指向的页面实际不存在。其产生 SIGSEGV 在图 2 中的流程为：

1 → 3 -→ 4 -→ 5 -→ 11 -→10

2.4 栈溢出了，有时 SIGSEGV，有时却啥都没发生

这也是 CU 常见的一个月经贴。大部分 C 语言教材都会告诉你，当从一个函数返回后，该函数栈上的内容会被自动“释放”。“释放”给大多数初学者的印象是 free()，似乎这块内存不存在了，于是当他访问这块应该不存在的内存时，发现一切都好，便陷入了深深的疑惑。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int* foo() {
 5         int a = 10;
 6
 7         return &a;
 8 }
 9
10 int main() {
11         int* b;
12
13         b = foo();
14         printf ("%d\n", *b);
15 }
```

当你编译这个程序时，会看到“warning: function returns address of local variable”，GCC 已经在警告你栈溢的可能了。实际运行结果一切正常。原因是操作系统通常以“页”的粒度来管理内存，Linux 中典型的页大小为 4K，内核为进程栈分配内存也是以 4K 为粒度的。故当栈溢的幅度小于页的大小时，不会产生 SIGSEGV。那是否说栈溢出超过 4K，就会产生 SIGSEGV 呢？看下面这个例子：

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
```

```
 4 char* foo() {
 5         char buf[8192];
 6
 7         memset (buf, 0x55, sizeof(buf));
 8         return buf;
 9 }
10
11 int main() {
12         char* c;
13
14         c = foo();
15         printf ("%#x\n", c[5000]);
16 }
```

虽然我们的栈溢已经超出了 4K 大小，可运行仍然正常。这是因为 C 教程中提到的“栈自动释放”实际上是改变栈指针，而其指向的内存，并不是在函数返回时就被回收了。在我们的例子中，所访问的栈溢处内存仍然存在。无效的栈内存（即栈指针范围外未被回收的栈内存）是由操作系统在需要时回收的，这是无法预测的，也就无法预测何时访问非法的栈内容会引发 SIGSEGV。

好了，在上面的例子中，我们的栈溢例子，无论是大于一个页尺寸还是小于一个页尺寸，访问的都是已分配而未回收的栈内存。那么访问未分配的栈内存，是否就一定会引发 SIGSEGV 呢？答案是否定的。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main() {
 5         char* c;
 6
 7         c = (char*)&c – 8192 *2;
 8         *c = 'a';
 9         printf ("%c\n", *c);
10 }
```

在 IA32 平台上，栈默认是向下增长的，我们栈溢 16K，访问一块未分配的栈区域（至少从我们的程序来看，此处是未分配的）。选用 16K 这个值，是要让我们的溢出范围足够大，大过内核为进程分配的初始栈大小（初始大小为 4K 或 8K）。按理说，我们应该看到期望的 SIGSEGV，但结果却非如此，一切正常。

答案藏在内核的 page fault 处理函数中：

```
    if (error_code & PF_USER) {
        /*
```

```
            * Accessing the stack below %sp is always a bug.
            * The large cushion allows instructions like enter
            * and pusha to work.  ("enter $65535,$31" pushes
            * 32 pointers and then decrements %sp by 65535.)
            */
           if (address + 65536 + 32 * sizeof(unsigned long) < regs->sp)
                goto bad_area;
    }
    if (expand_stack(vma, address))
           goto bad_area;
```

内核为 enter[*]这样的指令留下了空间，从代码来看，理论上栈溢小于 64K 左右都是没问题的，栈会自动扩展。令人迷惑的是，笔者用下面这个例子来测试栈溢的阈值，得到的确是 70K ~ 80K 这个区间，而不是预料中的 65K ~ 66K。

[*]关于 enter 指令的详细介绍，请参考《Intel(R) 64 and IA-32 Architectures Software Developer Manual Volume 1》6.5 节“PROCEDURE CALLS FOR BLOCK-STRUCTURED LANGUAGES”

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define GET_ESP(esp) do {     \
 5        asm volatile ("movl %%esp, %0\n\t" : "=m" (esp));    \
 6 }    while (0)
 7
 8
 9 #define K 1024
10 int main() {
11        char* c;
12        int i = 0;
13        unsigned long esp;
14
15        GET_ESP (esp);
16        printf ("Current stack pointer is %#x\n", esp);
17        while (1) {
18              c = (char*)esp -    i * K;
19              *c = 'a';
20              GET_ESP (esp);
21              printf ("esp = %#x, overflow %dK\n", esp, i);
22              i ++;
23        }
24 }
```

笔者目前也不能解释其中的魔术，这神奇的程序啊！上例中发生 SIGSEGV 时，在图 2

中的流程是：

1 -→ 3 -→ 4 -→ 5 -→ 11 -→ 10 （注意，发生 SIGSEGV 时，该地址已经不属于用户态栈了，所以是 5 → 11 而不是 5 -→ 6）

到这里，我们至少能够知道 SIGSEGV 和操作系统（栈的分配和回收），编译器（谁知道它会不会使用 enter 这样的指令呢）有着密切的联系，而不像教科书中“函数返回后其使用的栈自动回收”那样简单。

2.5 我们知道栈了，那么堆呢？

看了栈的例子，举一反三就能知道，SIGSEGV 和堆的关系取决于你的内存分配器，通常这意味着取决于 C 库的实现。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define K 1024
 5 int main () {
 6         char* c;
 7         int i = 0;
 8
 9         c = malloc (1);
10         while (1) {
11                 c += i*K;
12                 *c = 'a';
13                 printf ("overflow %dK\n", i);
14                 i ++;
15         }
16 }
```

上面这个例子在笔者机器上于 15K 时产生 SIGSEGV。让我们改变初次 malloc 的内存大小，当初次分配 16M 时，SIGSEGV 推迟到了溢出 180K；当初次分配 160M 时，SIGSEGV 推迟到了溢出 571K。我们知道内存分配器在分配不同大小的内存时通常有不同的机制，这个例子从某种角度证明了这点。此例 SIGSEGV 在图 2 中的流程为：

1 -→ 3 -→ 4 -→ 5 -→ 11 -→ 10

用一个野指针在堆里胡乱访问很少见，更多被问起的是“为什么我访问一块 free()后的内存却没发生 SIGSEGV”，比如下面这个例子：

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
```

```
 4 #define K 1024
 5 int main () {
 6         int* a;
 7
 8         a = malloc (sizeof(int));
 9         *a = 100;
10         printf ("%d\n", *a);
11         free (a);
12         printf ("%d\n", *a);
13 }
```

SIGSEGV 没有发生，但 free()后 a 指向的内存被清零了，一个合理的解释是为了安全。相信不会再有人问 SIGSEGV 没发生的原因。是的，free()后的内存不一定就立即归还给了操作系统，在真正的归还发生前，它一直在那儿。

2.6 如果是指向全局区的野指针呢？

看了上面两个例子，我觉得这实在没什么好讲的。

2.7 函数跳转到了一个非法的地址上执行

这也是产生 SIGSEGV 的常见原因，来看下面的例子：

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 void foo () {
 6         char c;
 7
 8         memset (&c, 0x55, 128);
 9 }
10
11 int main () {
12         foo();
13 }
```

通过栈溢出，我们将函数 foo 的返回地址覆盖成了 0x55555555，函数跳转到了一个非法地址执行，最终引发 SIGSEGV。非法地址执行，在图 2 中的流程中的可能性就太多了，从 1-→3 -→4 -→ … -→10，从 4 到 10 之间，几乎每条路径都可能出现。当然对于此例，0x55555555 所指向的页面并不在内存之中，其在图 2 的流程为:

1-→3 -→4 -→5--→11-→10

如果非法地址对应的页面（页面属于用户态地址空间）存在于内存中，它又是可执行的[*]，则程序会执行一大堆随机的指令。在这些指令执行过程中一旦访问内存，其产生 SIGSEGV 的流程几乎就无法追踪了（除非你用调试工具跟进）。看到这里，一个很合理的问题是：为什么程序在非法地址中执行的是随机指令，而不是非法指令呢？在一块未知的内存上执行，遇到非法指令可能性比较大吧，这样应该收到 SIGILL 信号啊？

[*]如果不用段寄存器的 type checking，只用页表保护，传统 32bit IA32 可读即可执行。在 NX 技术出现后页级也可以控制是否可以执行。

事实并非如此，我们的 IA32 架构使用了如此复杂的指令集，以至于找到一条非法指令的编码还真不容易。在下例子中：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5                       char    buf[128]    =    "asdfaowerqoweurqwuroahfoasdbaoseur20
234123akfhasbfqower53453";
6       sleep(1);
7 }
```

笔者在 buf 中随机的敲入了一些字符，反汇编其内容得到的结果是：

```
0xbffa9e00:     popa
0xbffa9e01:     jae     0xbffa9e67
0xbffa9e03:     popaw
0xbffa9e05:     outsl   %ds:(%esi),(%dx)
0xbffa9e06:     ja      0xbffa9e6d
0xbffa9e08:     jb      0xbffa9e7b
0xbffa9e0a:     outsl   %ds:(%esi),(%dx)
0xbffa9e0b:     ja      0xbffa9e72
0xbffa9e0d:     jne     0xbffa9e81
0xbffa9e0f:     jno     0xbffa9e88
0xbffa9e11:     jne     0xbffa9e85
0xbffa9e13:     outsl   %ds:(%esi),(%dx)
0xbffa9e14:     popa
0xbffa9e15:     push    $0x73616f66
0xbffa9e1a:     bound   %esp,%fs:0x6f(%ecx)
0xbffa9e1e:     jae     0xbffa9e85
0xbffa9e20:     jne     0xbffa9e94
0xbffa9e22:     xor     (%eax),%dh
0xbffa9e24:     and     %ah,(%eax)
0xbffa9e26:     and     %dh,(%edx)
0xbffa9e28:     xor     (%ecx,%esi,1),%esi
```

```
0xbffa9e2b:     xor     (%ebx),%dh
0xbffa9e2d:     popa
0xbffa9e2e:     imul    $0x61,0x68(%esi),%esp
0xbffa9e32:     jae     0xbffa9e96
0xbffa9e34:     data16
0xbffa9e35:     jno     0xbffa9ea6
0xbffa9e37:     ja      0xbffa9e9e
0xbffa9e39:     jb      0xbffa9e70
0xbffa9e3b:     xor     0x33(,%esi,1),%esi
0xbffa9e42:     add     %al,(%eax)
0xbffa9e44:     add     %al,(%eax)
0xbffa9e46:     add     %al,(%eax)
0xbffa9e48:     add     %al,(%eax)
0xbffa9e4a:     add     %al,(%eax)
0xbffa9e4c:     add     %al,(%eax)
0xbffa9e4e:     add     %al,(%eax)
0xbffa9e50:     add     %al,(%eax)
0xbffa9e52:     add     %al,(%eax)
0xbffa9e54:     add     %al,(%eax)
0xbffa9e56:     add     %al,(%eax)
0xbffa9e58:     add     %al,(%eax)
0xbffa9e5a:     add     %al,(%eax)
0xbffa9e5c:     add     %al,(%eax)
0xbffa9e5e:     add     %al,(%eax)

.....................................................................
```

一条非法指令都没有!大家也可以自己构造一些随机内容试试,看能得到多少非法指令。故在实际情况中，函数跳转到非法地址执行时，遇到 SIGSEGV 的概率是远远大于 SIGILL 的。

我们来构造一个遭遇 SIGILL 的情况，如下例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define GET_EBP(ebp)       \
    do {       \
        asm volatile ("movl %%ebp, %0\n\t" : "=m" (ebp));    \
    } while (0)

char buf[128];
```

```
void foo () {
    printf ("Hello world\n");
}

void build_ill_func() {
    int i = 0;

    memcpy (buf, foo, sizeof(buf));
    while (1) {
        /*
         * Find *call* instruction and replace it with
         * *ud2a* to generate a #UD exception
         */
        if ( buf[i] == 0xfffffffe8 ) {
            buf[i] = 0x0f;
            buf[i+1] = 0x0b;
            break;
        }
        i ++;
    }
}

void overflow_ret_address () {
    unsigned long ebp;
    unsigned long addr = (unsigned long)buf;
    int i;

    GET_EBP (ebp);

    for ( i=0; i<16; i++ )
        memcpy ((void*)(ebp + i*sizeof(addr)), &addr, sizeof(addr));
    printf ("ebp = %#x\n", ebp);
}

int main() {
    printf ("%p\n", buf);
    build_ill_func ();
    overflow_ret_address ();
}
```

我们在一块全局的 buf 里填充了一些指令，其中有一条是 ud2a，它是 IA32 指令集中用来构造一个非法指令陷阱。在 overflow_ret_address()中，我们通过栈溢出覆盖函数的返回地

址，使得函数返回时跳转到 buf 执行，最终执行到 ud2a 指令产生一个 SIGILL 信号。注意此例使用了 ebp 框架指针寄存器，在编译时不能使用-fomit-frame-pointer 参数，否则得不到期望的结果。

2.8 非法的系统调用参数

这是一种较为特殊的情况。特殊是指前面的例子访问非法内存都发生在用户态。而此例中，对非法内存的访问却发生在内核态。通常是执行 copy_from_user()或 copy_to_user()时。其流程在图 2 中为：

1 -→ …. -→ 11 -→ 12 -→ 13

内核使用 fixup[*]的技巧来处理在处理此类错误。ULK 说通常的处理是发送一个 SIGSEGV 信号，但实际大多数系统调用都可以返回 EFAULT（bad address）码，从而避免用户态程序被终结。这种情况就不举例了，笔者一时间想不出哪个系统调用可以模拟此种情况而不返回 EFAULT 错误。

[*] 关于 fixup 的技巧可以参考笔者另一篇文章《Linker Script in Linux》，http://linux.chinaunix.net/bbs/viewthread.php?tid=1032711&extra=page%3D2%26amp%3Bfilter%3Ddigest

2.9 还有什么？

我们已经总结了产生 SIGSEGV 的大多数情况，在实际编程中，即使现象不一样，最终发生 SIGSEGV 的原因都可以归到上述几类。掌握了这些基本例子，我们可以避免大多数的 SIGSEGV。

3.如何避免 SIGSEGV

良好的编程习惯永远是最好的预防方法。良好的习惯包括：

尽量按照 C 标准写程序。之所以说是尽量，是因为 C 标准有太多平台相关和无定义的行为，而其中一些实际上已经有既成事实的标准了。例如 C 标准中，一个越界的指针导致的是无定义的行为，而在实际情况中，一个越界而未解引用的指针是不会带来灾难后果的。借用 CU 的一个例子，如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
 3
 4 int main () {
 5         char a[] = "hello";
 6         char* p;
 7
 8         for ( p = a+5; p>=a; p-- )
 9                 printf ("%c\n", *p);
10
11 }
```

虽然循环结束后，p 指向了数组 a 前一个元素，在 C 标准中这是一个无定义的行为，但实际上程序却是安全的，没有必要为了不让 p 成为一个野指针而把程序改写为：

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main () {
 5         char a[] = "hello";
 6         char* p;
 7
 8         for ( p = a+5; p!=a; p-- ) {
 9                 printf ("%c\n", *p);
10         }
11         printf ("%c\n", *p);
12 }
```

当然，或许世界上真有编译器会对“越界但未解引用”的野指针进行处理，例如引发一个 SIGSEGV。笔者无法 100%保证，所以大家在实践中还是各自斟酌吧。

彻底的懂得你的程序。和其它程序员不同的是，C 程序员需要对自己的程序完全了解，做到精确控制。尤其在内存的分配和释放方面。在操作每一个指针前，你都应该清楚它所指向内存的出处（栈、堆、全局区），并清楚此内存的生存周期。只有明白的使用内存，才能最大限度的避免 SIGSEGV 的产生。

大量使用 assert。笔者偏好在程序中使用大量的 assert，凡是有认为不该出现的情况，笔者就会加入一个 assert 做检查。虽然 assert 无法直接避免 SIGSEGV，但它却能尽早的抛出错误。离错误越近，就越容易 root cause。很多时候出现 SIGSEGV 时，程序已经跑飞很远了 。

打开-Wall –Werror 编译选项。如果程序是自己写的，0 warning 应该始终是一项指标（0 warning 不包括因为编译器版本不同而引起的 warning）。一种常见的 SIGSEGV 来源于向函数传入了错误的参数类型。例如：

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
```

```
 3 #include <string.h>
 4
 5 int main () {
 6         char buf[12];
 7         int buff;
 8
 9         strcpy (buff, "hello");
10
11 }
```

这个例子中，本意是要向 buf 拷贝一个字符串，但由于有一个和 buf 名称很相近的 buff 变量，由于一个笔误（这个笔误很可能就来自你编辑器的自动补全，例如 vim 的 ctrl – p, ctrl – n），strcpy 如愿的引发了 SIGSEGV。实际在编译期间，编译器就提示我们 warning: passing argument 1 of `strcpy' makes pointer from integer without a cast，但我们忽略了。

这就进一步要求我们尽量使用编译器的类型检查功能，包括多用函数少用宏（特别是完成复杂功能的宏），函数参数多用带类型的指针，少用 void*指针等。此例就是我们在 2.2 节提到的不经意的行为。

少用奇技淫巧，多用标准方法。好的程序应该逻辑清楚，干净整洁，像一篇朗朗上口的文章，让人一读就懂。那种充满晦涩语法、怪异招数的试验作品，是不受欢迎的。很多人喜欢把性能问题做为使用不标准方法的借口，实际上他们根本不知道对性能的影响如何，拿不出具体指标，全是想当然尔。笔者曾经在项目中，将一个执行频繁的异常处理函数用汇编重写，使该函数的执行周期从 2000 多个机器周期下降到 40 多个。满心欢喜的提交了一个 patch 给该项目的 maintainer，得到的答复是：“张，你具体测试过你的 patch 能带来多大的性能提升吗？如果没有明显的数据，我是不愿意将优雅的 C 代码替换成这晦涩的汇编的。”于是我做了一个内核编译来测试 patch，耗时 15 分钟，我的 patch 带来的整体性能提升大约为 0.1%。所以，尽量写清楚明白的代码，不仅有利于避免 SIGSEGV，也利于在出现 SIGSEGV 后进行调试。

当你的一个需求，标准的方法不能满足时，只有两种可能：1.从一开始的设计就错了，才会导致错误的需求；2.你读过的代码太少，不知道业界解决该问题的标准方法是什么。计算机已经发展了几十年，如果你不是在做前沿研究，遇到一定得用非标准方法解决的问题的机会实在太小了。正如我们经常用 gdb 跟踪发现 SIGSEGV 发生在 C 库里，不要嚷嚷说 C 库有 bug，大部情况是一开始你传入的参数就错了。

小结

无论如何我们应该感谢 SIGSEGV，是它让我们能在不重启机器的情况下调试程序。相比那些由于内存使用错误而不得不一次又一次重启机器来 debug 的内核工程师，SIGSEGV 让我们的生活变得轻松。理解 SIGSEGV 同时，我们也更加理解程序。希望这篇文档对初学 C 语言的同志有些许帮助。