Linux 下的段错误（Segmentation fault）产生的原因及调试方法（经典）

2009-04-05 11:25

简而言之,产生段错误就是访问了错误的内存段，一般是你没有权限，或者根本就不存在对应的物理内存,尤其常见的是访问 0 地址.

一 般来说,段错误就是指访问的内存超出了系统所给这个程序的内存空间，通常这个值是由 gdtr 来保存的，他是一个 48 位的寄存器，其中的 32 位是保存由它指 向的 gdt 表，后 13 位保存相应于 gdt 的下标，最后 3 位包括了程序是否在内存中以及程序的在 cpu 中的运行级别,指向的 gdt 是由以 64 位为一个单位的 表，在这张表中就保存着程序运行的代码段以及数据段的起始地址以及与此相应的段限和页面交换还有程序运行级别还有内存粒度等等的信息。一旦一个程序发生了 越界访问，cpu 就会产生相应的异常保护，于是 segmentation fault 就出现了.

在编程中以下几类做法容易导致段错误,基本是是错误地使用指针引起的

1)访问系统数据区，尤其是往 系统保护的内存地址写数据

最常见就是给一个指针以 0 地址

2)内存越界(数组越界，变量类型不一致等) 访问到不属于你的内存区域

解决方法

我 们在用 C/C++语言写程序的时侯，内存管理的绝大部分工作都是需要我们来做的。实际上，内存管理是一个比较繁琐的工作，无论你多高明，经验多丰富，难 免会在此处犯些小错误，而通常这些错误又是那么的浅显而易于消除。但是手工“除虫”（debug），往往是效率低下且让人厌烦的，本文将就"段错误"这个 内存访问越界的错误谈谈如何快速定位这些"段

错误"的语句。

下面将就以下的一个存在段错误的程序介绍几种调试方法：

```
 1 dummy_function (void)
 2 {
 3         unsigned char *ptr = 0x00;
 4         *ptr = 0x00;
 5 }
 6
 7 int main (void)
 8 {
 9         dummy_function ();
10
11          return 0;
12 }
```

作为一个熟练的 C/C++程序员，以上代码的 bug 应该是很清楚的，因为它尝试操作地址为 0 的内存区域，而这个内存区域通常是不可访问的禁区，当然就会出错了。我们尝试编译运行它:

```
xiaosuo@gentux test $ ./a.out
段错误
```

果然不出所料，它出错并退出了。

1.利用 gdb 逐步查找段错误:

这种方法也是被大众所熟知并广泛采用的方法，首先我们需要一个带有调试信息的可执行程序，所以我们加上“-g -rdynamic"的参数进行编译，然后用 gdb 调试运行这个新编译的程序,具体步骤如下:

```
xiaosuo@gentux test $ gcc -g -rdynamic d.c
xiaosuo@gentux test $ gdb ./a.out
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".
```

```
(gdb) r
Starting program: /home/xiaosuo/test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x08048524 in dummy_function () at d.c:4
4               *ptr = 0x00;
(gdb)
```

哦？!好像不用一步步调试我们就找到了出错位置 d.c 文件的第 4 行，其实就是如此的简单。

从这里我们还发现进程是由于收到了 SIGSEGV 信号而结束的。通过进一步的查阅文档(man 7 signal)，我们知道 SIGSEGV 默认 handler 的动作是打印”段错误"的出错信息，并产生 Core 文件，由此我们又产生了方法二。

2.分析 Core 文件：

Core 文件是什么呢？

The default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing an image of the process's memory at the time of termination. A list of the signals which cause a process to dump core can be found in signal(7).

以 上资料摘自 man page(man 5 core)。不过奇怪了，我的系统上并没有找到 core 文件。

后来，忆起为了渐少系统上的拉圾文件的数量（本人有些洁癖，这也是我喜欢 Gentoo 的原因 之一），禁止了 core 文件的生成，查看了以下果真如此，将系统的 core 文件的大小限制在 512K 大小，再试:

```
xiaosuo@gentux test $ ulimit -c
0
xiaosuo@gentux test $ ulimit -c 1000
xiaosuo@gentux test $ ulimit -c
1000
xiaosuo@gentux test $ ./a.out
段错误 (core dumped)
xiaosuo@gentux test $ ls
a.out core d.c f.c g.c pango.c test_iconv.c test_regex.c
```

core 文件终于产生了，用 gdb 调试一下看看吧:

```
xiaosuo@gentux test $ gdb ./a.out core
GNU gdb 6.5
```

```
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".


warning: Can't read pathname for load map: 输入/输出错误.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0 0x08048524 in dummy_function () at d.c:4
4               *ptr = 0x00;
```

哇，好历害，还是一步就定位到了错误所在地，佩服一下 Linux/Unix 系统的此类设计。

接着考虑下去，以前用 windows 系统下的 ie 的时侯，有时打开某些网页，会出现“运行时错误”，这个时侯如果恰好你的机器上又装有 windows 的编译器的话，他会弹出来一个对话框，问你是否进行调试，如果你选择是，编译器将被打开，并进入调试状态，开始调试。

Linux 下如何做到这些呢？我的大脑飞速地旋转着，有了，让它在 SIGSEGV 的 handler 中调用 gdb，于是第三个方法又诞生了:

3.段错误时启动调试:

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

void dump(int signo)
{
        char buf[1024];
        char cmd[1024];
        FILE *fh;

        snprintf(buf, sizeof(buf), "/proc/%d/cmdline", getpid());
```

```
        if(!(fh = fopen(buf, "r")))
                exit(0);
        if(!fgets(buf, sizeof(buf), fh))
                exit(0);
        fclose(fh);
        if(buf[strlen(buf) - 1] == '\n')
                buf[strlen(buf) - 1] = '\0';
        snprintf(cmd, sizeof(cmd), "gdb %s %d", buf, getpid());
        system(cmd);

        exit(0);
}

        void
dummy_function (void)
{
        unsigned char *ptr = 0x00;
        *ptr = 0x00;
}

        int
main (void)
{
        signal(SIGSEGV, &dump);
        dummy_function ();

        return 0;
}
```

编译运行效果如下:

```
xiaosuo@gentux test $ gcc -g -rdynamic f.c
xiaosuo@gentux test $ ./a.out
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".

Attaching to program: /home/xiaosuo/test/a.out, process 9563
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
```

```
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0 0xffffe410 in __kernel_vsyscall ()
#1 0xb7ee4b53 in waitpid () from /lib/libc.so.6
#2 0xb7e925c9 in strtold_l () from /lib/libc.so.6
#3 0x08048830 in dump (signo=11) at f.c:22
#4 <signal handler called>
#5 0x0804884c in dummy_function () at f.c:31
#6 0x08048886 in main () at f.c:38
```

怎么样？是不是依旧很酷？

以上方法都是在系统上有 gdb 的前提下进行的，如果没有呢？其实 glibc 为我们提供了此类能够 dump 栈内容的函数簇，详见/usr/include/execinfo.h（这些函数都没有提供 man page，难怪我们找不到），另外你也可以通过 gnu 的手册§进行学习。

4.利用 backtrace 和 objdump 进行分析:

重写的代码如下:

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/* A dummy function to make the backtrace more interesting. */
	void
dummy_function (void)
{
	unsigned char *ptr = 0x00;
	*ptr = 0x00;
}

void dump(int signo)
{
	void *array[10];
	size_t size;
	char **strings;
	size_t i;

	size = backtrace (array, 10);
```

```
        strings = backtrace_symbols (array, size);

        printf ("Obtained %zd stack frames.\n", size);

        for (i = 0; i < size; i++)
                printf ("%s\n", strings[i]);

        free (strings);

        exit(0);
}

        int
main (void)
{
        signal(SIGSEGV, &dump);
        dummy_function ();

        return 0;
}
```

编译运行结果如下：

```
xiaosuo@gentux test $ gcc -g -rdynamic g.c
xiaosuo@gentux test $ ./a.out
Obtained 5 stack frames.
./a.out(dump+0x19) [0x80486c2]
[0xffffe420]
./a.out(main+0x35) [0x804876f]
/lib/libc.so.6(__libc_start_main+0xe6) [0xb7e02866]
./a.out [0x8048601]
```

这次你可能有些失望,似乎没能给出足够的信息来标示错误,不急,先看看能分析出来什么吧,

用 objdump 反汇编程序,找到地址 0x804876f 对应的代码位置:

```
xiaosuo@gentux test $ objdump -d a.out

8048765:       e8 02 fe ff ff          call   804856c <signal@plt>
804876a:       e8 25 ff ff ff          call   8048694 <dummy_function>
804876f:       b8 00 00 00 00          mov    $0x0,%eax
8048774:       c9                      leave
```

我们还是找到了在哪个函数(dummy_function)中出错的,信息已然不是很完整,不过有总比

没有好的啊!

后记:

本文给出了分析"段错误"的几种方法,不要认为这是与孔乙己先生的"回"字四种写法一样的哦,因为每种方法都有其自身的适用范围和适用环境,请酌情使用,或遵医嘱。