<https://www.cnblogs.com/panfeng412/archive/2011/11/06/2237857.html>

最近在Linux环境下做C语言项目，由于是在一个原有项目基础之上进行二次开发，而且项目工程庞大复杂，出现了不少问题，其中遇到最多、花费时间最长的问题就是著名的“段错误”（Segmentation Fault）。借此机会系统学习了一下，这里对Linux环境下的段错误做个小结，方便以后同类问题的排查与解决。

**1. 段错误是什么**

一句话来说，段错误是指访问的内存超出了系统给这个程序所设定的内存空间，例如访问了不存在的内存地址、访问了系统保护的内存地址、访问了只读的内存地址等等情况。这里贴一个对于“段错误”的准确定义（参考Answers.com）：

[复制代码](javascript:void(0);)

A segmentation fault (often shortened to segfault) is a particular error condition that can occur during the operation of computer software. In short, 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 (e.g., attempts to write to a read-only location, or to overwrite part of the operating system). Systems based on processors like the Motorola 68000 tend to refer to these events as Address or Bus errors.  
  
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 It has been superseded by paging for most purposes, but much of the terminology of segmentation is still used, "segmentation fault" being an example.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 invalid memory receives the SIGSEGV signal. On Microsoft Windows, a process that accesses invalid memory receives the STATUS\_ACCESS\_VIOLATION exception.

[复制代码](javascript:void(0);)

**2. 段错误产生的原因**

**2.1 访问不存在的内存地址**

[复制代码](javascript:void(0);)

#include<stdio.h>  
#include<stdlib.h>  
void main()  
{  
 int \*ptr = NULL;  
 \*ptr = 0;  
}

[复制代码](javascript:void(0);)

**2.2 访问系统保护的内存地址**

[复制代码](javascript:void(0);)

#include<stdio.h>  
#include<stdlib.h>  
void main()  
{  
 int \*ptr = (int \*)0;  
 \*ptr = 100;  
}

[复制代码](javascript:void(0);)

**2.3 访问只读的内存地址**

[复制代码](javascript:void(0);)

#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
void main()  
{  
 char \*ptr = "test";  
 strcpy(ptr, "TEST");  
}

[复制代码](javascript:void(0);)

**2.4 栈溢出**

[复制代码](javascript:void(0);)

#include<stdio.h>  
#include<stdlib.h>  
void main()  
{  
 main();  
}

[复制代码](javascript:void(0);)

等等其他原因。

**3. 段错误信息的获取**

程序发生段错误时，提示信息很少，下面有几种查看段错误的发生信息的途径。

**3.1 dmesg**

dmesg可以在应用程序crash掉时，显示内核中保存的相关信息。如下所示，通过dmesg命令可以查看发生段错误的程序名称、引起段错误发生的内存地址、指令指针地址、堆栈指针地址、错误代码、错误原因等。以程序2.3为例：

panfeng@ubuntu:~/segfault$ dmesg  
[ 2329.479037] segfault3[2700]: segfault at 80484e0 ip 00d2906a sp bfbbec3c error 7 in libc-2.10.1.so[cb4000+13e000]

**3.2 -g**

使用gcc编译程序的源码时，加上-g参数，这样可以使得生成的二进制文件中加入可以用于gdb调试的有用信息。以程序2.3为例：

panfeng@ubuntu:~/segfault$ gcc -g -o segfault3 segfault3.c

**3.3 nm**

使用nm命令列出二进制文件中的符号表，包括符号地址、符号类型、符号名等，这样可以帮助定位在哪里发生了段错误。以程序2.3为例：

[复制代码](javascript:void(0);)

panfeng@ubuntu:~/segfault$ nm segfault3  
08049f20 d \_DYNAMIC  
08049ff4 d \_GLOBAL\_OFFSET\_TABLE\_  
080484dc R \_IO\_stdin\_used  
 w \_Jv\_RegisterClasses  
08049f10 d \_\_CTOR\_END\_\_  
08049f0c d \_\_CTOR\_LIST\_\_  
08049f18 D \_\_DTOR\_END\_\_  
08049f14 d \_\_DTOR\_LIST\_\_  
080484ec r \_\_FRAME\_END\_\_  
08049f1c d \_\_JCR\_END\_\_  
08049f1c d \_\_JCR\_LIST\_\_  
0804a014 A \_\_bss\_start  
0804a00c D \_\_data\_start  
08048490 t \_\_do\_global\_ctors\_aux  
08048360 t \_\_do\_global\_dtors\_aux  
0804a010 D \_\_dso\_handle  
 w \_\_gmon\_start\_\_  
0804848a T \_\_i686.get\_pc\_thunk.bx  
08049f0c d \_\_init\_array\_end  
08049f0c d \_\_init\_array\_start  
08048420 T \_\_libc\_csu\_fini  
08048430 T \_\_libc\_csu\_init  
 U \_\_libc\_start\_main@@GLIBC\_2.0  
0804a014 A \_edata  
0804a01c A \_end  
080484bc T \_fini  
080484d8 R \_fp\_hw  
080482bc T \_init  
08048330 T \_start  
0804a014 b completed.6990  
0804a00c W data\_start  
0804a018 b dtor\_idx.6992  
080483c0 t frame\_dummy  
080483e4 T main  
 U memcpy@@GLIBC\_2.0

[复制代码](javascript:void(0);)

**3.4 ldd**

使用ldd命令查看二进制程序的共享链接库依赖，包括库的名称、起始地址，这样可以确定段错误到底是发生在了自己的程序中还是依赖的共享库中。以程序2.3为例：

panfeng@ubuntu:~/segfault$ ldd ./segfault3  
 linux-gate.so.1 => (0x00e08000)  
 libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x00675000)  
 /lib/ld-linux.so.2 (0x00482000)

**4. 段错误的调试方法**

**4.1 使用printf输出信息**

这个是看似最简单但往往很多情况下十分有效的调试方式，也许可以说是程序员用的最多的调试方式。简单来说，就是在程序的重要代码附近加上像printf这类输出信息，这样可以跟踪并打印出段错误在代码中可能出现的位置。

为了方便使用这种方法，可以使用条件编译指令#ifdef DEBUG和#endif把printf函数包起来。这样在程序编译时，如果加上-DDEBUG参数就能查看调试信息；否则不加该参数就不会显示调试信息。

**4.2 使用gcc和gdb**

**4.2.1 调试步骤**

 1、为了能够使用gdb调试程序，在编译阶段加上-g参数，以程序2.3为例：

panfeng@ubuntu:~/segfault$ gcc -g -o segfault3 segfault3.c

2、使用gdb命令调试程序：

[复制代码](javascript:void(0);)

panfeng@ubuntu:~/segfault$ gdb ./segfault3   
GNU gdb (GDB) 7.0-ubuntu  
Copyright (C) 2009 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i486-linux-gnu".  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>...  
Reading symbols from /home/panfeng/segfault/segfault3...done.  
(gdb)

[复制代码](javascript:void(0);)

3、进入gdb后，运行程序：

[复制代码](javascript:void(0);)

(gdb) run  
Starting program: /home/panfeng/segfault/segfault3   
  
Program received signal SIGSEGV, Segmentation fault.  
0x001a306a in memcpy () from /lib/tls/i686/cmov/libc.so.6  
(gdb)

[复制代码](javascript:void(0);)

从输出看出，程序2.3收到SIGSEGV信号，触发段错误，并提示地址0x001a306a、调用memcpy报的错，位于/lib/tls/i686/cmov/libc.so.6库中。

4、完成调试后，输入quit命令退出gdb：

[复制代码](javascript:void(0);)

(gdb) quit  
A debugging session is active.  
  
 Inferior 1 [process 3207] will be killed.  
  
Quit anyway? (y or n) y

[复制代码](javascript:void(0);)

**4.2.2 适用场景**

1、仅当能确定程序一定会发生段错误的情况下使用。

2、当程序的源码可以获得的情况下，使用-g参数编译程序。

3、一般用于测试阶段，生产环境下gdb会有副作用：使程序运行减慢，运行不够稳定，等等。

4、即使在测试阶段，如果程序过于复杂，gdb也不能处理。

**4.3 使用core文件和gdb**

在4.2节中提到段错误会触发SIGSEGV信号，通过man 7 signal，可以看到SIGSEGV默认的handler会打印段错误出错信息，并产生core文件，由此我们可以借助于程序异常退出时生成的core文件中的调试信息，使用gdb工具来调试程序中的段错误。

**4.3.1 调试步骤**

1、在一些Linux版本下，默认是不产生core文件的，首先可以查看一下系统core文件的大小限制：

panfeng@ubuntu:~/segfault$ ulimit -c  
0

2、可以看到默认设置情况下，本机Linux环境下发生段错误时不会自动生成core文件，下面设置下core文件的大小限制（单位为KB）：

panfeng@ubuntu:~/segfault$ ulimit -c 1024  
panfeng@ubuntu:~/segfault$ ulimit -c  
1024

3、运行程序2.3，发生段错误生成core文件：

panfeng@ubuntu:~/segfault$ ./segfault3  
段错误 (core dumped)

4、加载core文件，使用gdb工具进行调试：

[复制代码](javascript:void(0);)

panfeng@ubuntu:~/segfault$ gdb ./segfault3 ./core   
GNU gdb (GDB) 7.0-ubuntu  
Copyright (C) 2009 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i486-linux-gnu".  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>...  
Reading symbols from /home/panfeng/segfault/segfault3...done.  
  
warning: Can't read pathname for load map: 输入/输出错误.  
Reading symbols from /lib/tls/i686/cmov/libc.so.6...(no debugging symbols found)...done.  
Loaded symbols for /lib/tls/i686/cmov/libc.so.6  
Reading symbols from /lib/ld-linux.so.2...(no debugging symbols found)...done.  
Loaded symbols for /lib/ld-linux.so.2  
Core was generated by `./segfault3'.  
Program terminated with signal 11, Segmentation fault.  
#0 0x0018506a in memcpy () from /lib/tls/i686/cmov/libc.6

[复制代码](javascript:void(0);)

从输出看出，同4.2.1中一样的段错误信息。

5、完成调试后，输入quit命令退出gdb：

(gdb) quit

**4.3.2 适用场景**

1、适合于在实际生成环境下调试程序的段错误（即在不用重新发生段错误的情况下重现段错误）。

2、当程序很复杂，core文件相当大时，该方法不可用。

**4.4 使用objdump**

**4.4.1 调试步骤**

1、使用dmesg命令，找到最近发生的段错误输出信息：

panfeng@ubuntu:~/segfault$ dmesg  
... ...  
[17257.502808] segfault3[3320]: segfault at 80484e0 ip 0018506a sp bfc1cd6c error 7 in libc-2.10.1.so[110000+13e000]

其中，对我们接下来的调试过程有用的是发生段错误的地址：80484e0和指令指针地址：0018506a。

2、使用objdump生成二进制的相关信息，重定向到文件中：

panfeng@ubuntu:~/segfault$ objdump -d ./segfault3 > segfault3Dump

其中，生成的segfault3Dump文件中包含了二进制文件的segfault3的汇编代码。

3、在segfault3Dump文件中查找发生段错误的地址：

[复制代码](javascript:void(0);)

panfeng@ubuntu:~/segfault$ grep -n -A 10 -B 10 "80484e0" ./segfault3Dump   
121- 80483df: ff d0 call \*%eax  
122- 80483e1: c9 leave   
123- 80483e2: c3 ret   
124- 80483e3: 90 nop  
125-  
126-080483e4 <main>:  
127- 80483e4: 55 push %ebp  
128- 80483e5: 89 e5 mov %esp,%ebp  
129- 80483e7: 83 e4 f0 and $0xfffffff0,%esp  
130- 80483ea: 83 ec 20 sub $0x20,%esp  
131: 80483ed: c7 44 24 1c e0 84 04 movl $0x80484e0,0x1c(%esp)  
132- 80483f4: 08   
133- 80483f5: b8 e5 84 04 08 mov $0x80484e5,%eax  
134- 80483fa: c7 44 24 08 05 00 00 movl $0x5,0x8(%esp)  
135- 8048401: 00   
136- 8048402: 89 44 24 04 mov %eax,0x4(%esp)  
137- 8048406: 8b 44 24 1c mov 0x1c(%esp),%eax  
138- 804840a: 89 04 24 mov %eax,(%esp)  
139- 804840d: e8 0a ff ff ff call 804831c <memcpy@plt>  
140- 8048412: c9 leave   
141- 8048413: c3 ret

[复制代码](javascript:void(0);)

通过对以上汇编代码分析，得知段错误发生main函数，对应的汇编指令是movl $0x80484e0,0x1c(%esp)，接下来打开程序的源码，找到汇编指令对应的源码，也就定位到段错误了。

**4.4.2 适用场景**

1、不需要-g参数编译，不需要借助于core文件，但需要有一定的汇编语言基础。

2、如果使用了gcc编译优化参数（-O1，-O2，-O3）的话，生成的汇编指令将会被优化，使得调试过程有些难度。

**4.5 使用catchsegv**

catchsegv命令专门用来扑获段错误，它通过动态加载器（ld-linux.so）的预加载机制（PRELOAD）把一个事先写好的库（/lib/libSegFault.so）加载上，用于捕捉断错误的出错信息。

[复制代码](javascript:void(0);)

panfeng@ubuntu:~/segfault$ catchsegv ./segfault3  
Segmentation fault (core dumped)  
\*\*\* Segmentation fault  
Register dump:  
  
 EAX: 00000000 EBX: 00fb3ff4 ECX: 00000002 EDX: 00000000  
 ESI: 080484e5 EDI: 080484e0 EBP: bfb7ad38 ESP: bfb7ad0c  
  
 EIP: 00ee806a EFLAGS: 00010203  
  
 CS: 0073 DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 007b  
  
 Trap: 0000000e Error: 00000007 OldMask: 00000000  
 ESP/signal: bfb7ad0c CR2: 080484e0  
  
Backtrace:  
/lib/libSegFault.so[0x3b606f]  
??:0(??)[0xc76400]  
/lib/tls/i686/cmov/libc.so.6(\_\_libc\_start\_main+0xe6)[0xe89b56]  
/build/buildd/eglibc-2.10.1/csu/../sysdeps/i386/elf/start.S:122(\_start)[0x8048351]  
  
Memory map:  
  
00258000-00273000 r-xp 00000000 08:01 157 /lib/ld-2.10.1.so  
00273000-00274000 r--p 0001a000 08:01 157 /lib/ld-2.10.1.so  
00274000-00275000 rw-p 0001b000 08:01 157 /lib/ld-2.10.1.so  
003b4000-003b7000 r-xp 00000000 08:01 13105 /lib/libSegFault.so  
003b7000-003b8000 r--p 00002000 08:01 13105 /lib/libSegFault.so  
003b8000-003b9000 rw-p 00003000 08:01 13105 /lib/libSegFault.so  
00c76000-00c77000 r-xp 00000000 00:00 0 [vdso]  
00e0d000-00e29000 r-xp 00000000 08:01 4817 /lib/libgcc\_s.so.1  
00e29000-00e2a000 r--p 0001b000 08:01 4817 /lib/libgcc\_s.so.1  
00e2a000-00e2b000 rw-p 0001c000 08:01 4817 /lib/libgcc\_s.so.1  
00e73000-00fb1000 r-xp 00000000 08:01 1800 /lib/tls/i686/cmov/libc-2.10.1.so  
00fb1000-00fb2000 ---p 0013e000 08:01 1800 /lib/tls/i686/cmov/libc-2.10.1.so  
00fb2000-00fb4000 r--p 0013e000 08:01 1800 /lib/tls/i686/cmov/libc-2.10.1.so  
00fb4000-00fb5000 rw-p 00140000 08:01 1800 /lib/tls/i686/cmov/libc-2.10.1.so  
00fb5000-00fb8000 rw-p 00000000 00:00 0  
08048000-08049000 r-xp 00000000 08:01 303895 /home/panfeng/segfault/segfault3  
08049000-0804a000 r--p 00000000 08:01 303895 /home/panfeng/segfault/segfault3  
0804a000-0804b000 rw-p 00001000 08:01 303895 /home/panfeng/segfault/segfault3  
09432000-09457000 rw-p 00000000 00:00 0 [heap]  
b78cf000-b78d1000 rw-p 00000000 00:00 0  
b78df000-b78e1000 rw-p 00000000 00:00 0  
bfb67000-bfb7c000 rw-p 00000000 00:00 0 [stack]

[复制代码](javascript:void(0);)

**5. 一些注意事项**

1、出现段错误时，首先应该想到段错误的定义，从它出发考虑引发错误的原因。

2、在使用指针时，定义了指针后记得初始化指针，在使用的时候记得判断是否为NULL。

3、在使用数组时，注意数组是否被初始化，数组下标是否越界，数组元素是否存在等。

4、在访问变量时，注意变量所占地址空间是否已经被程序释放掉。

5、在处理变量时，注意变量的格式控制是否合理等。

**6. 参考资料列表**

1、http://www.docin.com/p-105923877.html

2、http://blog.chinaunix.net/space.php?uid=317451&do=blog&id=92412