Segfault错误定位方法总结

以前只要出现Segfault错误就不知道怎么入手，经过几个问题单的处理，只要掌握方法，Segfault错误也可以像我们定位普通错误一样，快速的定位到Segfault错误的地方！一般我们有一下两种方法进行定位：

1. 在出现Segfault错误的进程主函数中添加SIGSEGV信号处理函数，并在处理函数中打印调用栈信息，这样当出现Segfault错误时，内核就会发送一个信号SIGSEGV给该进程，该进程收到这个信号就会进入到我们定义的信号处理函数，并打印出调用栈信息，可以马上定位到出现Segfault错误的函数。

当进程出现Segfault错误时，内核会捕捉到错误信息，并打印在/var/log/messags中，因为我们首先通过messages的信息找到出现Segfault错误的目标进程，然后在目标进程的入口main函数中注册SIGSEGV信号处理函数，并定义实现该信号处理函数。具体代码如下所示：

/\* add by j01842, define the process segfault handler to print call stack info to

\* analysis the segfault error.

\*/

#define SEGFAULT\_TRACE\_SIZE 30

void process\_segfault\_handler (int sig)

{

void \*trace\_array[SEGFAULT\_TRACE\_SIZE];

int trace\_size;

char \*\*trace\_strings = NULL;

int line;

syslog (LOG\_NOTICE, "Process: segfault trace start:\n");

trace\_size = backtrace(trace\_array, SEGFAULT\_TRACE\_SIZE);

if (trace\_size > 0 && trace\_size <= SEGFAULT\_TRACE\_SIZE)

{

trace\_strings = backtrace\_symbols (trace\_array, trace\_size);

if (NULL != trace\_strings)

{

for(line = 0; line < trace\_size; line++)

{

syslog(LOG\_NOTICE, "Process: segfault: %s\n", trace\_strings[line]);

}

gsh\_free(trace\_strings);

trace\_strings = NULL;

}

}

syslog(LOG\_NOTICE, "Ganesha: segfault trace end.\n");

exit(-1);

}

int main()

{

……………………………………………..

/\* Add by j01842, catch the SIGSEGV and print the call stack info to analysis the segfault error.

\*/

signal(SIGSEGV, process\_segfault\_handler);

…………………………………………….

}

出现Segfault错误之后，打印的调用栈信息如下所示（举例）：

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault trace start:

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /usr/local/hstor/sms\_install/usr/bin/cluster\_volume\_perf\_daemon() [0x4014a9]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /lib64/libc.so.6(+0x32900) [0x7f40b3cad900]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /usr/local/hstor/sms\_install/usr/lib/libcluster\_nas.so(cluster\_volume\_perf\_data\_read+0x134) [0x7f40b42bd5c4]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /usr/local/hstor/sms\_install/usr/bin/cluster\_volume\_perf\_daemon() [0x401c16]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /usr/local/hstor/sms\_install/usr/bin/cluster\_volume\_perf\_daemon(main+0x3d) [0x401e1d]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /lib64/libc.so.6(\_\_libc\_start\_main+0xfd) [0x7f40b3c99cdd]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault: /usr/local/hstor/sms\_install/usr/bin/cluster\_volume\_perf\_daemon() [0x4013a9]

Jun 17 16:38:09 cluster-77 cluster\_volume\_perf\_daemon: Cluster\_volume\_perf\_daemon: segfault trace end.

我们可以通过打印的调用栈信息快速定位到 出现Segfault错误的定位为libcluster\_nas.so库中cluster\_volume\_perf\_data\_read函数出现了段错误，那我们就直接可以到cluster\_volume\_perf\_data\_read函数中添加相应的调试信息，并按照问题环境进行复现，从而可以快速定位到该函数具体出现Segfault错误的代码，然后进行修改即可。

1. 打开系统dump开关，当出现Segfault错误时，会在指定或者默认目录下生成core文件，而这个文件就记录了详细的错误信息。不过我们要在编译该目标程序时添加 -g 选项，这样 用gdb调试时才能准确地显示定位出出错的函数名和具体参数。否则为乱码。
2. 打开系统dump开关（如果是关闭的话）

在/etc/profile文件中修改如下设置：

ulimit -S -c 0 > /dev/null 2>&1   
 修改为：

ulimit -S -c unlimited > /dev/null 2>&1

然后 source /etc/profile 使之生效。

注意： 生成core dump文件的位置，默认位置与可执行程序在同一目录下，文件名为core.\*\*\*,其中\*\*\*是一个数字。core dump文件名的模式保存在/proc/sys/kernel/core\_pattern中，缺省值是core。通过以下命令可以更改core dump文件的位置（如希望生成到/tmp/cores目录下）

echo “/tmp/cores/core-%e-%p-%t” > /proc/sys/kernel/core\_pattern

至此，当出现程序崩溃的时候就会在相应目录下生成core文件。我们下面就可以用gdb工具对其进行查看。

1. 用gdb工具定位core文件中错误信息

按以下格式运行gdb：

gdb 目标程序 core文件

然后 输入 bt， where

就可以指定显示出出现错误的 具体调用函数和详细参数！！！

最后总结一下常见会出现Segfault错误的地方

1. 使用非法的内存地址（指针）,包括使用未经初始化及已经释放的指针、不存在的地址、受系统保护的地址，只读的地址等，特别要注意对指针值引用前的非空判断，这一类也是最常见和最好解决的段错误问题。
2. 内存读/写越界。包括数组访问越界，或在使用一些写内存的函数时，长度指定不正确或者这些函数本身不能指定长度，典型的函数由strcpy(strncpy)，sprintf(snprintf)等等。
3. 对于C++对象，应该通过相应类的接口去进行内存操作，禁止通过其返回的指针对内存进行写操作，典型的如string类的c\_str()接口，如果你强制往其返回的指针写操作肯定会段错误的，因为其返回的地址是只读的。
4. 函数不要返回其中局部对象的引用或地址，当函数返回时，函数栈弹出，局部对象的地址将失效，改写或读这些地址都会造成未知的错误。
5. 避免在栈中定义过大的数组，否则可能导致进程的栈空间不足，此时也会出现段错误，同样的，在创建进程/线程时如果不知道此线程/进程最大需要多少栈空间时最好不要在代码中指定栈大小，应该使用系统默认的，这样问题比较好查，ulimit一下即可知道。这类问题也是为什么我们的程序在其他平台上跑的好好的，一移植到某些平台就段错误了。
6. 操作系统的相关限制，如：进程可以分配的最大内存，进程可以打开的最大文件描述符个数等，在Linux下这些需要同ulimit、setrlimit、sysctl等来解除相关限制，这类段错误问题在系统移植上经常发生。
7. 多线程的程序，涉及到多个线程同时操作一块内存时必须进行互斥，否则内存中的内容将不可预料。
8. 在多环境下使用非线程安全的函数调用，例如strerror函数等。
9. 在有信号的环境中，使用不可重入函数调用，而这些函数内部会读或写某片内存区，当信号中断时，内存写操作将被打断，而下次进入时将无法避免的出错。
10. 跨进程传递某个地址，传递的都是经过映射的虚拟地址，对另外一个进程是不通用的。
11. 某些有特殊要求的系统调用，例如epoll\_wait，正常情况下使用close关闭一个套接字后，epoll会不再返回这个socket上的事件，但是如果你使用dup或dup2操作，将导致epoll无法进行移除操作，此时再进行读写操作肯定是段错误。