利用Core Dump调试程序

[描述]

       这里介绍Linux环境下使用gdb结合core dump文件进行程序的调试和定位。

[简介]

       当用户程序运行，可能会由于某些原因发生崩溃（crash），这个时候可以产生一个Core Dump文件，记录程序发生崩溃时候内存的运行状况。这个Core Dump文件，一般名称为core或者core.pid(pid就是应用程序运行时候的pid号)，它可以帮助我们找出程序崩溃的原因。

       对于一个运行出错的程序，我们可以有多种方法调试它，以便发生错误的原因：a)通过阅读代码；b)通过在代码中设置一些打印语句（插旗子）；c)通过使用gdb设置断点来跟踪程序的运行。但是这些方法对于调试程序运行崩溃这样类似的错误，定位都不够迅速，如果程序代码很多的话，显然前面的方法有很多缺陷。在后面，我们来看看另外一种可以定位错误的方法：d)使用gdb结合Core Dump文件来迅速定位到这个错误。这个方法，如果程序运行崩溃，那么可以迅速找到导致程序崩溃的原因。

       当然，调试程序，没有哪个方法是最好的，这里只对最后一种方法重点讲解，实际过程中，往往根据需要和其他方法结合使用。

[举例]

       下面，给出一个实际的操作过程，描述我们使用gdb调试工具，结合Core Dump文件，定位程序崩溃的位置。

一、程序源代码

       下面是我们的程序源代码：

         1 #include <iostream>

         2 using std::cerr;

         3 using std::endl;

         4

         5 void my\_print(int d1, int d2);

         6 int main(int argc, char \*argv[])

         7 {

         8     int a = 1;

         9     int b = 2;

       10     my\_print(a,b);

       11     return 0;

       12 }

       13

       14 void my\_print(int d1, int d2)

       15 {

       16     int \*p1=&d1;

       17     int \*p2 = NULL;

       18     cerr<<"first is:"<<\*p1<<",second is:"<<\*p2<<endl;

       19 }

       这里，程序代码很少，我们可以直接通过代码看到这个程序第17行的p2是NULL，而18行却用\*p2来进行引用，明显这样访问一个空的地址是一个错误（也许我们的初衷是使用p2来指向d2）。

       我们可以有多种方法调试这个程序，以便发生上面的错误：a)通过阅读代码;b)通过在代码中设置一些打印语句（插旗子）;c)通过使用gdb设置断点来跟踪程序的运行。但是这些方法对于这个程序中类似的错误，定位都不够迅速，如果程序代码很多的话，显然前面的方法有很多缺陷。在后面，我们来看看另外一种可以定位错误的方法：d)使用gdb结合Core Dump文件来迅速定位到这个错误。

二、编译程序

       编译过程如下：

       [root@lv-k test]# ls

       main.cpp

       [root@lv-k test]# g++ -g main.cpp

       [root@lv-k test]# ls

       a.out  main.cpp

       这样，编译main.cpp生成了可执行文件a.out，一定注意，因为我们要使用gdb进行调试，所以我们使用'g++'的'-g'选项。

三、运行程序

       运行过程如下：

       [root@lv-k test]# ./a.out

       段错误

       [root@lv-k test]# ls

       a.out  main.cpp

       这里，如我们所期望的，会打印段错误的信息,但是并没有生成Core Dump文件。

       配置生成Core Dump文件的选项,并生成Core Dump：

       [root@lv-k test]# ulimit -c unlimited

       [root@lv-k test]# ./a.out

       段错误 (core dumped)

       [root@lv-k test]# ls

       a.out  core.30557  main.cpp

       这里，我们看到，使用'ulimit'配置之后，程序崩溃的时候就会生成Core Dump文件了，这里的文件是core.30557，文件名称不同的系统生成的名称有一点不同，这里linux生成的名称是:"core"+".pid"。

四、调试程序

       使用Core Dump文件，结合gdb工具进行调试，过程如下：

       1)初步定位：

       [root@lv-k test]# gdb a.out core.30557

       GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5\_5.2)

       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 "i386-redhat-linux-gnu".

       For bug reporting instructions, please see:

       <http://www.gnu.org/software/gdb/bugs/>...

       Reading symbols from /root/test/a.out...done.

       Reading symbols from /usr/lib/libstdc++.so.6...(no debugging symbols found)...                done.

       Loaded symbols for /usr/lib/libstdc++.so.6

       Reading symbols from /lib/libm.so.6...(no debugging symbols found)...done.

       Loaded symbols for /lib/libm.so.6

       Reading symbols from /lib/libgcc\_s.so.1...(no debugging symbols found)...done.

       Loaded symbols for /lib/libgcc\_s.so.1

       Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.

       Loaded symbols for /lib/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 `./a.out'.

       Program terminated with signal 11, Segmentation fault.

       #0  0x0804870e in my\_print (d1=1, d2=2) at main.cpp:18

       18              cerr<<"first is:"<<\*p1<<",second is:"<<\*p2<<endl;

       这里，我们就进入了gdb的调试交互界面，看到gdb直接定位到导致程序出错的位置了。我们还可以使用如下命令："#gdb a.out --core=core.30557"。

       通过错误，我们知道程序由于"signal 11"导致终止，如果想要大致了解"signal 11"，那么我们可查看signal的man手册：

       #man 7 signal

       这样，在输出的信息中我们可以看见“SIGSEGV      11       Core    Invalid memory reference”这样的字样，意思是说，signal(信号)11表示非法内存引用。注意这里使用"man 7 signal"而不是"man signal"，因为我们要查看的不是signal函数或者signal命令，而是signal的其他信息，其他的信息在man手册的第7节，具体需要了解一些使用man的命令。

       2）查看具体调用关系

       (gdb) bt

       #0  0x0804870e in my\_print (d1=1, d2=2) at main.cpp:18

       #1  0x08048799 in main (argc=<value optimized out>, argv=<value optimized out>) at main.cpp:10

       这里，我们通过backtrace(简写为bt)命令可以看到，导致崩溃那条语句是通过什么调用途径被调用到的。

       3)设置断点,并进行调试等：

       (gdb) b main.cpp:10

       Breakpoint 1 at 0x8048787: file main.cpp, line 10.

       (gdb) r

       Starting program: /root/test/a.out

       Breakpoint 1, main (argc=<value optimized out>, argv=<value optimized out>) at                 main.cpp:10

       10              my\_print(a,b);

       (gdb) s

       my\_print (d1=1, d2=2) at main.cpp:16

       16              int \*p1=&d1;

       (gdb) n

       17              int \*p2 = NULL;

       (gdb) n

       18              cerr<<"first is:"<<\*p1<<",second is:"<<\*p2<<endl;

       (gdb) p d1

       $1 = 1

       (gdb) p d2

       $2 = 2

       (gdb) p \*p1

       $1 = 1

       (gdb) p \*p2

       Cannot access memory at address 0x0

       (gdb) n

       Program received signal SIGSEGV, Segmentation fault.

       0x0804870e in my\_print (d1=1, d2=2) at main.cpp:18

       18              cerr<<"first is:"<<\*p1<<",second is:"<<\*p2<<endl;

       这里，我们在开始初步的定位的基础上，通过设置断点（break)，运行（run），gdb的单步跟进（step)，单步跳过(next)，变量的打印(print)等各种gdb命令，来了解产生崩溃时候的具体情况，确定产生崩溃的原因。

       4）退出gdb：

       (gdb) q

       A debugging session is active.

               Inferior 3 [process 30584] will be killed.

               Inferior 1 [process 1] will be killed.

       Quit anyway? (y or n) y

       Quitting: Couldn't get registers: 没有那个进程.

       [root@lv-k test]#

       [root@lv-k test]# ls

       a.out core.30557 core.30609 main.cpp

       这里，我们看到又产生了一个core文件。因为刚才调试，导致又产生了一个core文件。实际，如果我们只使用"gdb a.out core.30557"初步定位之后，不进行调试就退出gdb的话，就不会再生成core文件。

五、修正错误

       1）通过上面的过程我们最终修正错误，得到正确的源代码如下：

         1 #include <iostream>

         2 using std::cerr;

         3 using std::endl;

         4

         5 void my\_print(int d1, int d2);

         6 int main(int argc, char \*argv[])

         7 {

         8     int a = 1;

         9     int b = 2;

       10     my\_print(a,b);

       11     return 0;

       12 }

       13

       14 void my\_print(int d1, int d2)

       15 {

       16     int \*p1=&d1;

       17     //int \*p2 = NULL;//lvkai-

       18     int \*p2 = &d2;//lvkai+

       19     cerr<<"first is:"<<\*p1<<",second is:"<<\*p2<<endl;

       20 }

       2）编译并运行这个程序，最终产生结果如下：

       [root@lv-k test]# g++ main.cpp

       [root@lv-k test]# ls

       a.out  main.cpp

       [root@lv-k test]# ./a.out

       first is:1,second is:2

       这里，得到了我们预期的结果。

       另外，有个小技巧，如果对Makefile有些了解的话可以充分利用make的隐含规则来编译单个源文件的程序，

       过程如下：

       [root@lv-k test]# ls

       main.cpp

       [root@lv-k test]# make main

       g++     main.cpp   -o main

       [root@lv-k test]# ls

       main  main.cpp

       [root@lv-k test]# ./main

       first is:1,second is:2

       这里注意，make的目标参数必须是源文件"main.cpp"去掉后缀之后的"main"，等价于"g++ main.cpp -o main"，这样编译的命令比较简单。

[其它]

       其它内容有待添加。

       认真地工作并且思考，是最好的老师。在工作的过程中思考自己所缺乏的技术，以及学习他人的经验，才能在工作中有所收获。这篇文章原来是工作中我的一个同事加朋友的经验，我站在这样的经验的基础上，进行了这样总结。

       以上文章，如果有什么问题或者更好的建议，都可以联系我，谢谢。^\_^

作者：QuietHeart

Email:quiet\_heart000@126.com

时间：2011年4月23日