一，什么是coredump

我们经常听到大家说到程序core掉了，需要定位解决，这里说的大部分是指对应程序由于各种异常或者bug导致在运行过程中异常退出或者中止，并且在满足一定条件下（这里为什么说需要满足一定的条件呢？下面会分析）会产生一个叫做core的文件。

通常情况下，core文件会包含了程序运行时的内存，寄存器状态，堆栈指针，内存管理信息还有各种函数调用堆栈信息等，我们可以理解为是程序工作当前状态存储生成第一个文件，许多的程序出错的时候都会产生一个core文件，通过工具分析这个文件，我们可以定位到程序异常退出的时候对应的堆栈调用等信息，找出问题所在并进行及时解决。

二，coredump文件的存储位置

core文件默认的存储位置与对应的可执行程序在同一目录下，文件名是core，大家可以通过下面的命令看到core文件的存在位置：

cat /proc/sys/kernel/core_pattern

缺省值是core

注意：这里是指在进程当前工作目录的下创建。通常与程序在相同的路径下。但如果程序中调用了chdir函数，则有可能改变了当前工作目录。这时core文件创建在chdir指定的路径下。有好多程序崩溃了，我们却找不到core文件放在什么位置。和chdir函数就有关系。当然程序崩溃了不一定都产生 core文件。

如下程序代码：则会把生成的core文件存储在/data/coredump/wd，而不是大家认为的跟可执行文件在同一目录。

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

int main(){
   //change the current working directory to "/data/coredump/wd"
   const char *wdir = "/data/coredump/wd";
   int ret = -1;
   ret = chdir(wdir);
   if(0 != ret) {
      printf("chdir fails, ret : %d", ret);
      return 0;
   }

   char *ptr = "linux.xxx";
   *ptr = 0;

   return 0;

}
```

通过下面的命令可以更改coredump文件的存储位置，若你希望把core文件生成到/data/coredump/core目录下：

echo “/data/coredump/core”> /proc/sys/kernel/core_pattern

注意，这里当前用户必须具有对/proc/sys/kernel/core_pattern的写权限。

缺省情况下，内核在coredump时所产生的core文件放在与该程序相同的目录中，并且文件名固定为*core*。很显然，如果有多个程序产生core文件，或者同一个程序多次崩溃，就会重复覆盖同一个core文件，因此我们有必要对不同程序生成的core文件进行分别命名。

我们通过修改kernel的参数，可以指定内核所生成的coredump文件的文件名。例如，使用下面的命令使kernel生成名字为*core.filename.pid*格式的core dump文件：

echo “/data/coredump/core.%e.%p” >/proc/sys/kernel/core_pattern

这样配置后，产生的core文件中将带有崩溃的程序名、以及它的进程ID。上面的%e和%p会被替换成程序文件名以及进程ID。

如果在上述文件名中包含目录分隔符“/”，那么所生成的core文件将会被放到指定的目录中。 需要说明的是，在内核中还有一个与coredump相关的设置，就是/proc/sys/kernel/core_uses_pid。如果这个文件的内容被配置成1，那么即使core_pattern中没有设置%p，最后生成的core dump文件名仍会加上进程ID。

三，如何判断一个文件是coredump文件?

在类unix系统下，coredump文件本身主要的格式也是ELF格式，因此，我们可以通过readelf命令进行判断。

```
TEG_23_76_sles10_64:/data/coredump # readelf -h core
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              CORE (Core file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         25
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

可以看到ELF文件头的Type字段的类型是：CORE (Core file)

可以通过简单的file命令进行快速判断：

```
TEG_23_76_sles10_64:/data/coredump # file core
core: ELF 64-bit LSB core file AMD x86-64, version 1 (SYSV), SVR4-style, from 'coremain
```

四，产生coredum的一些条件总结

1， 产生coredump的条件，首先需要确认**当前会话**的ulimit –c，若为0，则不会产生对应的coredump，需要进行修改和设置。

ulimit -c unlimited (可以产生coredump且不受大小限制)

若想甚至对应的字符大小，则可以指定：

ulimit –c [size]

```
tenfyguo@TEG_23_76_sles10_64:~/test/coredumpTest> ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 32064
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 32064
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

可以看出，这里的size的单位是blocks,一般1block=512bytes

如：

ulimit –c 4 (注意，这里的size如果太小，则可能不会产生对应的core文件，笔者设置过ulimit –c 1的时候，系统并不生成core文件，并尝试了1，2，3均无法产生core，至少需要4才生成core文件)

但当前设置的ulimit只对当前会话有效，若想系统均有效，则需要进行如下设置：

Ø 在/etc/profile中加入以下一行，这将允许生成coredump文件

ulimit-c unlimited

Ø 在rc.local中加入以下一行，这将使程序崩溃时生成的coredump文件位于/data/coredump/目录下:

echo /data/coredump/core.%e.%p> /proc/sys/kernel/core_pattern

注意rc.local在不同的环境，存储的目录可能不同，susu下可能在/etc/rc.d/rc.local

更多ulimit的命令使用，可以参考：http://baike.baidu.com/view/4832100.htm

这些需要有root权限, 在ubuntu下每次重新打开中断都需要重新输入上面的ulimit命令, 来设置core大小为无限.

2， 当前用户，即执行对应程序的用户具有对写入core目录的写权限以及有足够的空间。

3， 几种不会产生core文件的情况说明：

The core file will not be generated if

(a) the process was set-user-ID and the current user is not the owner of the program file, or

(b) the process was set-group-ID and the current user is not the group owner of the file,

(c) the user does not have permission to write in the current working directory,

(d) the file already exists and the user does not have permission to write to it, or

(e) the file is too big (recall the RLIMIT_CORE limit in Section 7.11). The permissions of the core file (assuming that the file doesn't alrea read and user-write, although Mac OS X sets only user-read.

五，coredump产生的几种可能情况

造成程序coredump的原因有很多，这里总结一些比较常用的经验吧：

1，内存访问越界

a) 由于使用错误的下标，导致数组访问越界。

b) 搜索字符串时，依靠字符串结束符来判断字符串是否结束，但是字符串没有正常的使用结束符。

c) 使用strcpy, strcat, sprintf, strcmp,strcasecmp等字符串操作函数，将目标字符串读/写爆。应该使用strncpy, strlcpy, strncat, strlcat, snprintf, strncmp, strncasecmp等函数防止读写越界。

2，多线程程序使用了线程不安全的函数。

应该使用下面这些可重入的函数，它们很容易被用错：

asctime_r(3c) gethostbyname_r(3n) getservbyname_r(3n)ctermid_r(3s) gethostent_r(3n) getservbyport_r(3n) ctime_r(3c) getlogin_r(3c)getservent_r(3n) fgetgrent_r(3c) getnetbyaddr_r(3n) getspent_r(3c)fgetpwent_r(3c) getnetbyname_r(3n) getspnam_r(3c) fgetspent_r(3c)getnetent_r(3n) gmtime_r(3c) gamma_r(3m) getnetgrent_r(3n) lgamma_r(3m) getauclassent_r(3)getprotobyname_r(3n) localtime_r(3c) getauclassnam_r(3) etprotobynumber_r(3n)nis_sperror_r(3n) getauevent_r(3) getprotoent_r(3n) rand_r(3c) getauevnam_r(3)getpwent_r(3c) readdir_r(3c) getauevnum_r(3) getpwnam_r(3c) strtok_r(3c) getgrent_r(3c)getpwuid_r(3c) tmpnam_r(3s) getgrgid_r(3c) getrpcbyname_r(3n) ttyname_r(3c)getgrnam_r(3c) getrpcbynumber_r(3n) gethostbyaddr_r(3n) getrpcent_r(3n)

3，多线程读写的数据未加锁保护。

对于会被多个线程同时访问的全局数据，应该注意加锁保护，否则很容易造成coredump

4，非法指针

a) 使用空指针

b) 随意使用指针转换。一个指向一段内存的指针，除非确定这段内存原先就分配为某种结构或类型，或者这种结构或类型的数组，否则不要将它转换为这种结构或类型的指针，而应该将这段内存拷贝到一个这种结构或类型中，再访问这个结构或类型。这是因为如果这段内存的开始地址不是按照这种结构或类型对齐的，那么访问它时就很容易因为bus error而core dump。

5，堆栈溢出

不要使用大的局部变量（因为局部变量都分配在栈上），这样容易造成堆栈溢出，破坏系统的栈和堆结构，导致出现莫名其妙的错误。

六，利用gdb进行coredump的定位

其实分析coredump的工具有很多，现在大部分类unix系统都提供了分析coredump文件的工具，不过，我们经常用到的工具是gdb。

这里我们以程序为例子来说明如何进行定位。

1， 段错误 – segmentfault

Ø 我们写一段代码往受到系统保护的地址写内容。

```
tenfyguo@TEG_23_76_sles10_64:~/test/coredumpTest> cat ./coremain.cpp
#include <stdio.h>

void core_test1() {
    int i = 0;
    scanf("%d", i); /*should have used & to get addr othen than access directly*/
    printf("%d/n", i);
}

void core_test2() {
    char *ptr = "myname is tenfyguo";
    *ptr = 0;
}


int main(){
    core_test1();
    return 0;

}
```

Ø 按如下方式进行编译和执行，注意这里需要-g选项编译。

```
tenfyguo@TEG_23_76_sles10_64:~/test/coredumpTest> g++ coremain.cpp -g -o coremain
tenfyguo@TEG_23_76_sles10_64:~/test/coredumpTest> ./coremain
12
Segmentation fault (core dumped)
```

可以看到，当输入12的时候，系统提示段错误并且core dumped

Ø 我们进入对应的core文件生成目录，优先确认是否core文件格式并启用gdb进行调试。

```
tenfyguo@TEG_23_76_sles10_64:/data/coredump> file core.coremain.19879
core.coremain.19879: ELF 64-bit LSB core file AMD x86-64, version 1 (SYSV), SVR4-style, from 'coremain'
tenfyguo@TEG_23_76_sles10_64:/data/coredump> gdb /data/home/tenfyguo/test/coredumpTest/coremain core.coremain.19879
GNU gdb 6.6
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 "x86_64-suse-linux"...
Using host libthread_db library "/lib64/libthread_db.so.1".
Reading symbols from /usr/lib64/libstdc++.so.6...done.
Loaded symbols for /usr/lib64/libstdc++.so.6
Reading symbols from /lib64/libm.so.6...done.
Loaded symbols for /lib64/libm.so.6
Reading symbols from /lib64/libgcc_s.so.1...done.
Loaded symbols for /lib64/libgcc_s.so.1
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./coremain'.
Program terminated with signal 11, Segmentation fault.
#0  0x00002b94ac8e1ec0 in _IO_vfscanf_internal () from /lib64/libc.so.6
(gdb) bt
#0  0x00002b94ac8e1ec0 in _IO_vfscanf_internal () from /lib64/libc.so.6
#1  0x00002b94ac8eadfc in scanf () from /lib64/libc.so.6
#2  0x000000000040062f in core_test1 () at coremain.cpp:5
#3  0x000000000040064d in main () at coremain.cpp:16
(gdb)
```

从红色方框截图可以看到，程序中止是因为信号11，且从bt(backtrace)命令（或者where）可以看到函数的调用栈，即程序执行到coremain.cpp的第5行，且里面调用scanf 函数，而该函数其实内部会调用_IO_vfscanf_internal()函数。

接下来我们继续用gdb，进行调试对应的程序。

记住几个常用的gdb命令：

l(list)，显示源代码，并且可以看到对应的行号；

b(break)x, x是行号，表示在对应的行号位置设置断点；

p(print)x, x是变量名，表示打印变量x的值

r(run), 表示继续执行到断点的位置

n(next),表示执行下一步

c(continue),表示继续执行

q(quit)，表示退出gdb

启动gdb,注意该程序编译需要-g选项进行。

```
tenfyguo@TEG_23_76_sles10_64:~/test/coredumpTest> gdb ./coremain
GNU gdb 6.6
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 "x86_64-suse-linux"...
Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) l
7       }
8
9       void core_test2() {
10         char *ptr = "myname is tenfyguo";
11         *ptr = 0;
12      }
13
14
15      int main(){
16         core_test1();
(gdb) l 7
2
3       void core_test1() {
4          int i = 0;
5          scanf("%d", i); /*should have used & to get addr othen than access directly*/
6          printf("%d/n", i);
7       }
8
9       void core_test2() {
10         char *ptr = "myname is tenfyguo";
11         *ptr = 0;
(gdb) b 5
Breakpoint 1 at 0x40061d: file coremain.cpp, line 5.
(gdb) r
Starting program: /data/home/tenfyguo/test/coredumpTest/coremain

Breakpoint 1, core_test1 () at coremain.cpp:5
5          scanf("%d", i); /*should have used & to get addr othen than access directly*/
(gdb) p i
$1 = 0
(gdb) n
12

Program received signal SIGSEGV, Segmentation fault.
0x00002ae435e70ec0 in _IO_vfscanf_internal () from /lib64/libc.so.6
(gdb) p i
No symbol "i" in current context.
(gdb) c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb) q
```

注： SIGSEGV 11 Core Invalid memoryreference

七，附注：

1， gdb的查看源码

显示源代码

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码。还是来看一看查看源代码的GDB命令吧。

list<linenum>

显示程序第linenum行的周围的源程序。

list<function>

显示函数名为function的函数的源程序。

list

显示当前行后面的源程序。

list -

显示当前行前面的源程序。

一般是打印当前行的上5行和下5行，如果显示函数是是上2行下8行，默认是10行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

setlistsize <count>

设置一次显示源代码的行数。
showlistsize
查看当前listsize的设置。
list命令还有下面的用法：
list<first>, <last>
显示从first行到last行之间的源代码。
list ,<last>
显示从当前行到last行之间的源代码。
list +
往后显示源代码。
一般来说在list后面可以跟以下这些参数：

<linenum> 行号。
<+offset> 当前行号的正偏移量。
<-offset> 当前行号的负偏移量。
<filename:linenum> 哪个文件的哪一行。
<function> 函数名。
<filename:function>哪个文件中的哪个函数。
<*address> 程序运行时的语句在内存中的地址。

2， 一些常用signal的含义
SIGABRT：调用abort函数时产生此信号。进程异常终止。
SIGBUS：指示一个实现定义的硬件故障。
SIGEMT：指示一个实现定义的硬件故障。EMT这一名字来自PDP-11的emulator trap 指令。
SIGFPE：此信号表示一个算术运算异常，例如除以0，浮点溢出等。
SIGILL：此信号指示进程已执行一条非法硬件指令。4.3BSD由abort函数产生此信号。SIGABRT现在被用于此。
SIGIOT：这指示一个实现定义的硬件故障。IOT这个名字来自于PDP-11对于输入 / 输出TRAP(input/outputTRAP)指令的缩写。系统V的早期版本，由abort函数产生此信号。SIGABRT现在被用于此。
SIGQUIT：当用户在终端上按退出键（一般采用Ctrl-/）时，产生此信号，并送至前台进
程组中的所有进程。此信号不仅终止前台进程组（如SIGINT所做的那样），同时产生一个core文件。
SIGSEGV：指示进程进行了一次无效的存储访问。名字SEGV表示“段违例（segmentationviolation）”。
SIGSYS：指示一个无效的系统调用。由于某种未知原因，进程执行了一条系统调用指令，但其指示系统调用类型的参数却是无效的。
SIGTRAP：指示一个实现定义的硬件故障。此信号名来自于PDP-11的TRAP指令。
SIGXCPUSVR4和4.3+BSD支持资源限制的概念。如果进程超过了其软C P U时间限制，则产生此信号。
SIGXFSZ：如果进程超过了其软文件长度限制，则SVR4和4.3+BSD产生此信号。

3， Core_pattern的格式
可以在core_pattern模板中使用变量还很多，见下面的列表：
%% 单个%字符
%p 所dump进程的进程ID
%u 所dump进程的实际用户ID
%g 所dump进程的实际组ID
%s 导致本次core dump的信号
%t core dump的时间 (由1970年1月1日计起的秒数)
%h 主机名
%e 程序文件名