## 调试技巧

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **更改历史** | | | | |
| 次数 | 版本号 | 更改内容 | 时间 | 作者 |
| 1 | V0.1 | First draft。 | 2013-1-5 | Braveyly |
| 2 | V0.2 | 增加动态库依赖关系察看一章 | 2013-1-26 | Braveyly |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |
| 6 |  |  |  |  |
| 7 |  |  |  |  |
| 8 |  |  |  |  |
| 9 |  |  |  |  |

[调试技巧 1](#_Toc378537449)

[一、Linux调试工具汇总 3](#_Toc378537450)

[1、MEMWATCH 3](#_Toc378537451)

[2、YAMD 3](#_Toc378537452)

[3、electric fence 3](#_Toc378537453)

[4、strace 3](#_Toc378537454)

[5、gdb 3](#_Toc378537455)

[6、Oops 4](#_Toc378537456)

[7、mtrace 4](#_Toc378537457)

[8、binutil 4](#_Toc378537458)

[9、Purify 5](#_Toc378537459)

[10、pc-lint 5](#_Toc378537460)

[11、splint 5](#_Toc378537461)

[12、boundercheck 5](#_Toc378537462)

[13、valgrind 5](#_Toc378537463)

[二、软件调试资料汇总 5](#_Toc378537464)

[三、GDB贴士 6](#_Toc378537465)

[1、GDB调试基本命令 6](#_Toc378537466)

[2、多线程程序调试 6](#_Toc378537467)

[3、程序core掉调试 7](#_Toc378537468)

[4、GDB远程调试 8](#_Toc378537469)

[四、断点类型 8](#_Toc378537470)

[0、概述 8](#_Toc378537471)

[1、各种断点 8](#_Toc378537472)

[2、进阶应用 9](#_Toc378537473)

[五、Windows调试工具及调试思路 10](#_Toc378537474)

[1、调试工具 10](#_Toc378537475)

[2、调试思路 10](#_Toc378537476)

[六、VS调试小技巧 11](#_Toc378537477)

[1、vs函数重复定义也生成exe 11](#_Toc378537478)

[2、vs调试变量显示格式控制 11](#_Toc378537479)

[3、VS2005断点失效问题解决方案 11](#_Toc378537480)

[4、使用windbg替代vs进行断点调试 12](#_Toc378537481)

[七、Windows用户态调试技术 12](#_Toc378537482)

[1、工具使用 12](#_Toc378537483)

[2、测试实例 13](#_Toc378537484)

[八、动态库依赖关系察看 16](#_Toc378537485)

一、Linux调试工具汇总

“工欲善其事 必先利其器”现在将一些常见的调试工具进行一下总结.

1、MEMWATCH

功能：

（1）MEMWATCH 支持 ANSI C

（2）它提供结果日志纪录

（3）能检测双重释放（double-free）

（4）错误释放（erroneous free）

（5）没有释放的内存（unfreed memory）、

（6）溢出和下溢等等。

参考文献：

（1）http://blog.csdn.net/lengxingfei/archive/2006/08/09/1040800.aspx

2、YAMD

功能：

（1）查找 c 和 c++ 中动态的、与内存分配有关的问题

参考文献：

（1）<http://www.cs.hmc.edu/~nate/yamd/>

（2）http://www.ppsql.com/server/p262/J26232224\_2.shtml

3、electric fence

功能：

（1）检测内存泄露

（2）检查数据越界

（3）分配受保护的内存

参考文献：

（1）

<http://book.opensourceproject.org.cn/embedded/oreillybuildembed/index.html?page=opensource/belinuxsys-chp-11-sect-4.html>

（2）

<http://book.opensourceproject.org.cn/embedded/oreillybuildembed/opensource/belinuxsys-chp-11-sect-4.html>

（3）<http://www.ppsql.com/server/p262/J26232224_3.shtml>

4、strace

功能：

（1）显示用户空间程序发出的系统调用

（2）显示调用的参数和返回值

5、gdb

功能：

（1）启动程序

（2）使程序能够停止在指定文件的指定位置

（3）查看程序变量值

（4）改变程序执行期间的相关变量、路径等

6、Oops

功能： 显示系统错误信息

参考文献：

（1）http://www.ibm.com/developerworks/cn/linux/sdk/l-debug/

7、mtrace

功能：检测一些内存分配和泄漏的失败等

参考文献：

（1）

<http://www.vbaccelerator.com/home/VB/Code/Techniques/RunTime_Debug_Tracing/mTrace_bas.asp>

（2）<http://www.yuanma.org/data/2006/0612/article_789.htm>  
（3）http://www.mylinux.com.cn/HTML/1183705958487.html

8、binutil

说明：binutil为一个工具集合，包含如下的工具。

addr2line 把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行

文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。

ar 建立、修改、提取归档文件。归档文件是包含多个文件内容的一个大文件，其

结构保证了可以恢复原始文件内容。

as 主要用来编译GNU C编译器gcc输出的汇编文件，产生的目标文件由连接器ld

连接。

c++filt 连接器使用它来过滤 C++ 和 Java 符号，防止重载函数冲突。

gprof 显示程序调用段的各种数据。

ld 是连接器，它把一些目标和归档文件结合在一起，重定位数据，并链接符号引

用。通常，建立一个新编译程序的最后一步就是调用ld。

nm 列出目标文件中的符号。

objcopy 把一种目标文件中的内容复制到另一种类型的目标文件中.

objdump 显示一个或者更多目标文件的信息。显示一个或者更多目标文件的信息。

使用选项来控制其显示的信息。它所显示的信息通常只有编写编译工具的人才感兴趣。

ranlib 产生归档文件索引，并将其保存到这个归档文件中。在索引中列出了归档文

件各成员所定义的可重分配目标文件。

readelf 显示ebf格式可执行文件的信息。

size 列出目标文件每一段的大小以及总体的大小。默认情况下，对于每个目标文

件或者一个归档文件中的每个模块只产生一行输出。

strings 打印某个文件的可打印字符串，这些字符串最少4个字符长，也可以使用

选项-n设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其它类型的文件 它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。

strip 丢弃目标文件中的全部或者特定符号。

libiberty 包含许多GNU程序都会用到的函数，这些程序有： getopt, obstack, strerror, strtol 和 strtoul.

libbfd 二进制文件描述库.

libopcodes 用来处理opcodes的库, 在生成一些应用程序的时候也会用到它, 比如objdump.Opcodes是文本格式可读的处理器操作指令.

功能：程序调试、归档等

参考文献：

http://man.chinaunix.net/linux/

9、Purify

功能：内存检查工具

参考文献：

http://blog.csdn.net/haoel/archive/2003/12/16/2904.aspx

10、pc-lint

功能：C/C++的静态代码检测工具

参考文献：

（1）http://dev.yesky.com/134/2593634.shtml

11、splint

说明：splint是一个动态检查C语言程序安全弱点和编写错误的程序

功能：

（1）空引用错误

（2）未定义的变量错误

（3）类型错误

（4）内存检查

参考文献：

（1）  
<http://www.turbolinux.com.cn/turbo/wiki/doku.php?id=splint&DokuWiki=e3b300af2d101a8bec5a4c805e678823>

（2）http://www.splint.org/

12、boundercheck

功能：检查内存泄露

13、valgrind

功能：

检查内存错误

参考文献:

<http://blog.donews.com/idlecat511/archive/2006/02/21/736877.aspx>

二、软件调试资料汇总

《软件调试》及补编 张银奎 （比较系统全面）

《软件调试实战》 （德）格勒特克，赵俐译 （介绍了几款性能测试的工具，包括quantity）

《Windows用户态程序高效排错》熊力 （由几篇博客拼凑而成，并且以互联网应用为示例）

《Windows高级调试》 Mario Hewardt Daniel Pravat 著

《日臻完善:软件调试与优化典型应用》

Windbg简明教程.doc Sucsor on 2006.06.11 （介绍了windbg最常用的命令）

通往WinDbg的捷径 博客 （清晰实用的步骤）

三、GDB贴士

1、GDB调试基本命令

（1）开启调试：gdb filename（启动一个程序来调试）或者attach pid（调试正在运行程序）或者file filename（开启gdb后，然后启动一个程序开始执行）  
（2）l（list）：查看文件  
（3）b 6 | b sum | b 8 if i == 10：设置文件行断点；设置函数断点；设置条件断点  
（4）info b：查看断点信息  
（5）r（run）：运行代码  
（6）c（continue）：继续运行指导遇到断点或退出程序  
（7）n（next）：下一步，遇到函数，则跳过函数  
（8）s（step）：下一步，遇到函数，则进入该函数  
（9）delete 断点号：删除指定断点  
（10）disable 断点号：停止指定断点  
（11）p（print） 变量 | 表达式：查看程序运行时对应表达式和变量的值  
（12）display 表达式 | info display | undisplay num：自动显示  
（13）x显示内存  
（14）bt：显示调用栈  
（15）set：设置运行中程序变量的值

2、多线程程序调试

（1）info threads 显示当前可调试的所有线程，每个线程会有一个GDB为其分配的ID，后面操作线程的时候会用到这个ID。 前面有\*的是当前调试的线程。   
（2）thread ID 切换当前调试的线程为指定ID的线程。   
（3）break thread\_test.c:123 thread all 在所有线程中相应的行上设置断点  
（4）thread apply ID1 ID2 command 让一个或者多个线程执行GDB命令command。   
（5）thread apply all command 让所有被调试线程执行GDB命令command。   
（6）set scheduler-locking off|on|step 估计是实际使用过多线程调试的人都可以发现，在使用step或者continue命令调试当前被调试线程的时候，其他线程也是同时执行的，怎么只让被调试程序执行呢？通过这个命令就可以实现这个需求。off 不锁定任何线程，也就是所有线程都执行，这是默认值。 on 只有当前被调试程序会执行。  
注意在set scheduler-locking on，设置只运行当前线程后，需要c一下，代码才会继续往下执行

多线程调试示例程序

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  void print\_xs(void\* a)  {  while(1)  {  sleep(1);  printf("thread %d,x\n",pthread\_self());  }  return NULL;  }  void print\_ys(void\* a )  {  while(1)  {  sleep(1);  printf("thread %d,y\n",pthread\_self());  }  return NULL;  }  int main()  {  pthread\_t thread\_id1,thread\_id2;  pthread\_create(&thread\_id1,NULL,&print\_xs,NULL);  pthread\_create(&thread\_id2,NULL,&print\_ys,NULL);  while(1);  return 0;  } |

3、程序core掉调试

（1）设定core文件和文件大小为无限大  
ulimit -c unlimited  
ulimit unlimited  
（2）用gdb查看core文件  
gdb ./test test.core  
（3）bt查看发生段错误segment error的地方

|  |
| --- |
| #include <stdio.h>  int sum()  {  int i = 100;  int a = i/0;  }  int main()  {  sum();  return 0;  } |

4、GDB远程调试

 环境：两台centos虚拟机，Host（199.168.128.1），Target（199.168.128.5）；Host装有GDB、Target装有GDBserver。  
 步骤：（1）Host上-g编译生成测试程序test  
    （2）将测试程序拷贝到Target机上，并执行：gdbserver 199.168.128.1:2345 ./test  
    （3）在Host机上执行：gdb test;target remote 199.168.128.5:2345  
    （4）开始打断点，并display查看各个变量的值  
注意： （1）第三步时出现 No route to host，关闭Host和Target的防火墙，执行：service iptables stop  
    （2）如果Target为Arm体系，则需要在Host将GDB源码自行编译成针对Arm平台的程序（暂命名为GDBARM），并在Host上用GDBARM代替GDB来调试程序  
    （3）同样，对于Target上产生的core，则可将其拷贝到Host上，执行：GDBARM ./test core  
    （4）参考<http://blog.chinaunix.net/uid-20680966-id-3475824.html>

四、断点类型

0、概述

总共有如下几种断点类型：

普通的位置断点：在程序的固定位置产生断点；

函数断点：在函数定义的第一行产生断点，优点是可以不用打开源代码即可设置断点，并可以用来找main函数；

数据断点：有时候需要弄清楚哪段代码改变了变量，即可使用此断点，当变量值发生改变时，产生此断点；

条件断点--根据命中次数：在一个循环中，可能需要在特定的循环次数中产生中断；

条件断点—根据具体条件：可以用任何变量来定义特定的条件满足时，产生断点，实际上也包括了“根据命中次数”；

断点的进阶应用：

当产生断点时，执行一段代码；

针对特定线程进行调试；

1、各种断点

#include <iostream>

using namespace::std;

int main()

{

int i =0;

int j;

for( i = 0; i < 1000; i++ )

{

cout << "i is "<<i<<endl;

if( i == 650 )

j = 100;

}

}

1.1普通断点

在特定行按F9，则产生断点，只要程序有运行到此行，则产生断点

1.2函数断点

“工具”->”自定义”->”命令”->”调试”->”在函数处中断”

将”在函数处中断”拖到工具栏，然后点击，输入”main”进行测试。

1.3数据断点

“工具”->”自定义”->”命令”->”调试”->”新建数据断点”

首先将”新建数据断点”拖到工具栏，然后在数据变量所在函数入口处，设置普通断点，然后按F5，程序停留在函数入口处，则”新建数据断点”使能，点击”新建数据断点”，输入&j，再按F5，则下个断点发生在i==650时，此时j发生改变。

1.4条件断点--根据命中次数

先在循环内部的特定行插入普通断点，然后在右键此行，选择”断点””命中次数”，则当循环特定次数时，发生中断

1.5条件断点—根据具体条件

先在循环内部的特定行插入普通断点，然后在右键此行，选择”断点””条件”，则当在循环中，此条件满足时发生中断

2、进阶应用

2.1当断点发生时，继续运行，并执行特定操作

先设置上述任意一种断点。然后在断点行，点击右键->”断点”->”命中条件”

可以打印一段消息：这点对应socket编程，可能发生超时的API来说，当断点发生时，不能够中断程序，而是需要让程序继续跑下去；

可以运行宏：可以运行段宏来改变程序中的变量；

可以继续运行：勾选；

2.2限定在特定线程中使用断点

先设置上述任意一种断点，然后在断点行，点击右键->”断点”->”筛选器”

参考资料：

Visual Studio调试之断点基础篇

[http://www.cnblogs.com/killmyday/archive/2009/09/26/1574311.html]

Visual Studio调试之断点技巧篇[http://blog.csdn.net/Donjuan/article/details/4618717]

Visual Studio调试之断点进阶篇[http://blog.csdn.net/Donjuan/article/details/4600546]

理解First Chance和Second Chance避免单步调试

[http://blog.csdn.net/Donjuan/article/details/3859160]

五、Windows调试工具及调试思路

1、调试工具

（1）Windbg , vc6,vc8, Ntsd, cdb

分析挂死,死锁,CPU占用,异常的最佳工具,提供dump文件生成,源代码查看,线程栈打印,资源调用跟踪等功能,是定位问题的神兵利器

（2）DebugDiag

分析多种类型内存泄漏,异常,监控进程的执行状态的好工具,特点是泄漏代码调用栈清晰,给出的修改建议可操作性强,同时较其他的分析工具线程栈更加全面. 更强大的功能是能够选择异常类型自动生成dump文件,解决随机问题很有效

（3）Application verifier

辅助工具,通过设置开关,能够跟踪多种资源的使用,并给出最近的调用栈.是分析泄漏,死锁的好工具.

（4）Umdh

分析堆泄漏的好工具

（5）Mr.Watson, Adplus.vbs

进程监控工具,能够记录进程运行过程出现的异常并自动产生可供分析的dump文件

2、调试思路

2.1指针访问违例等异常类问题

(1)要处理此类问题,请首先熟悉windbg调试器的的用法.

(2)要分析异常问题,找到异常代码,首先要有异常情况下的core dump文件,操作系统处理过程中,有硬件中断,到操作系统的SHE处理,最终到分发给调试器,分析core dump,找到异常线程栈,找到线程栈后,开始从windows的SHE相关接口往上看,找到出错的自己模块和代码.

相关可以分享的调试命令:

~\*kn 100 //切换到指点线程.

!analyze –v/ -hang/-crash //调试器直接进行异常分析,找出线程,并展示异常码和异常堆栈

.frame x //在有源代码的情况下,直接查看接口的栈变量,参数和返回值等信息

Kn kb kd.dump .logopen dc //更多的命令请大家学习windbg的帮助文档.

2.2线程死锁等挂死类问题

(1)通过!analyze –hang找出互锁的线程组,切换到其中的一个线程中.

(2)通过windbg的元命令!CS等找到windows核心对象的句柄,找到拥有核心对象的线程. 参考命令

(3)~\*kn 切换到拥有锁的线程, 通过kn100 开始分析线程堆栈,找到导致死锁的代码流程.并进行修改.

请参照windbg 用户手册进行操作.

2.3内存/句柄泄漏类问题

(1)要快速定位内存和句柄泄漏,并迅速找到泄漏堆栈和代码行,请首先熟悉工具DebugDiag的使用.

(2)通过debugDiag监控出现性能问题的进程.

(3)当产生内存和句柄泄漏的时候,通过debugDiag抓去此时的进程的coredump文件,

(4)配置代码的pdb访问路径,启用DebugDiag的分析功能开始分析,通过windows分析报告的堆栈提示

找到泄漏代码行,问题解决.DebugDiag的使用方法,请参照文档: 性能类问题定位工具DebugDiag使用说明

2.4CPU占用内存高问题

(1)进程的CPU占用高,最好在CPU占用高时,抓去当前进程的core dump文件.

(2)通过Windbg的元命令!runaway分析core dump文件,找出累计占用CPU时间最多的几个线程.

(3)结合代码分析可疑线程,找出调用最频繁的接口和函数.或者可以直接永windbg调试进程采用wt系列命令追踪栈上各个接口调用的次数和时间,找出性能瓶颈.

六、VS调试小技巧

1、vs函数重复定义也生成exe

有时候平台库（不可修改）中有一个函数的定义，而自己会打桩（会修改）也定义

了这个函数，这时候就会出现重复定义的情况。使用/FORCE MULTIPLE强制输出文件。

而具体使用哪个函数，则将链接库的顺序改一下，就可以选择性的使用平台库函数

还是桩函数

http://msdn.microsoft.com/zh-cn/library/70abkas3(v=vs.90).aspx

2、vs调试变量显示格式控制

由autoexp.dat文件控制。详情见Improving Visual C++ Debugging with Better Data Display

http://www.codeguru.com/cpp/v-s/debug/debuggers/article.php/c15291/Improving-Visual-C-Debugging-with-Better-Data-Display.htm#page-1

3、VS2005断点失效问题解决方案

其实问题在于，在空项目中不生成调试文件pdb，所以无法调试。

要让项目生成pdb 文件，需要更改：

项目属性，configuration properties->linker->Generate Debug Info 从 no 改为 yes

(项目属性:属性->连接器->调试->生成连接信息 改为"是")

但这样还是不够的，还需要更改：

项目属性，configuration properties->c/c++->debug information format 为/ZI

(项目属性:属性->c++ -> 调试信息格式 为/ZI )

项目属性，configuration properties->c/c++->optimization为Disabled

(项目属性:属性->c++ -> 优化 改为禁用 （这点不是必须的）

因为为了生成这个文件，需要设定debug信息的格式并关掉O2，还要更改linker生成调

试信息的开关

如果有兴趣diff 项目的.vcproj文件，会发现更改了里面的三行配置信息：

<Tool

Name="VCCLCompilerTool"

Optimization="0" //原来是2

DebugInformationFormat="4" //原来是0

/>

<Tool

Name="VCLinkerTool"

GenerateDebugInformation="true" //原来是false

/>

4、使用windbg替代vs进行断点调试

使用windbg替代vs进行断点调试

Vs作为一个IDE，对于源代码的断点单步调试，已经非常成熟了，但有时候各方面的原因（例如vs出现了问题），则windbg可以作为替代，大材小用。

（1） windbg的安装、symbol文件的装载等基本配置操作

（2） 打开executable文件

（3） 按F5运行程序，等待程序把初始化工作做完

（4） 按Ctrl+break暂停程序，以便设置断点

（5） 打开源文件，找到指定行，按F9进行断点设置

（6） 再次按F5重启程序

（7） 触发断点

（8） F10单步 F11进入函数 shift+F11跳出函数 F5继续执行

参考资料：

windbg使用入门--开始调试【http://blog.csdn.net/fly\_here/article/details/2045229】

七、Windows用户态调试技术

1、工具使用

1.1产生dump工具比较

有5种产生用户态dump文件的工具，包括ADPlus、Dr.Watson、CDB 和、WinDbg和userDump。

其中ADPlus是目前最好的工具。抓crash的具体命令为：

adplus.vbs -crash -pn TMSACSDm.exe -fullonfirst -o c:\dumps

注意必须在进程起起来后，并在crash之前启动ADPlus。

1.2WinDbg设置symbol环境变量和路径\_NT\_SYMBOL\_PATH=srv\*c:\mysymbols\*http://msdl.microsoft.com/download/symbols;cache\*c:\mysymbols

同时，在WinDbg中设置Symbol File Path为同样的值。

1.3PDB文件

PDB文件为符号文件，包含了二进制文件的调试信息，符号文件包括全局变量、局部变量、函数名和它们的入口地址、FPO（Frame Pointer Omisson）数据。Frame Pointer是一种用来在调用堆栈中找到下一个将要被调用的函数的数据结构源代码的行序号。1.2节设置的路径就是用来查找PDB文件的，包括系统的符号文件，将从微软网站自动下载。【url：http://www.vckbase.com/index.php/wv/1418.html符号文件——Windows 应用程序调试必备】

WinDbg系列工具中symchk.exe可用于检查exe和pdb文件是否匹配

symchk.exe TMSACSDm.exe /v /s ./

[url: 关于PDB与EXE/DLL 文件的匹配问题：

http://blogold.chinaunix.net/u/8681/showart\_2217695.html]

2、测试实例

2.1进程Crash实例

（1）将软件对应pdb文件，包括dll和lib的pdb文件拷贝到c:\mysymbols中

（2）开启进程

（3）运行adplus.vbs -crash -pn TMSACSDm.exe -fullonfirst -o c:\dumps

（4）进程crash

（5）查看c:\dumps下dump文件

（6）windbg->file->Open Crash Dump

（7）!analyze –v综合分析crash情况

（8）kb显示堆栈

（9）ln addressA,显示addressA地址附近的符号

2.2利用map和cod文件找到崩溃地址对应代码

（1）设置linker-debugging-Generate Map File 为 YES

（2）设置C/C++-Output Files-Assembler Output 为 Assembly, Machine Code and Source (/FAcs)

（3）从map文件中，找到比崩溃地址（0x00411a84）小的最大的地址（0x00411a30），从而找到崩溃的函数名

（4）用崩溃地址减去崩溃函数地址，得到偏移地址（0x54），从cod文件中找到偏移地址为0x54的行，发现对应的代码行为（int b = 123/a;）

|  |
| --- |
| // test123.cpp : 定义控制台应用程序的入口点。  //  #include "stdafx.h"  int \_tmain(int argc, \_TCHAR\* argv[])  {  int a = 0;  int n = 30000;  while( n )  {  printf( "n is %d\n",n );  n--;  }  int b = 123/a;  return 0;  } |

2.3进程消耗CPU实例

（1）通过任务管理器，找到消耗CPU多的进程

（2）用windbg附上这个进程

（3）!runaway 3显示每个线程用户模式和内核时间

（4）g运行一段时间，然后ctrl+c再次暂停程序

（5）再次运行!runaway 3，显示每个线程消耗的时间。对比两次!runaway 3命令的运行结果，可以得出消耗CPU最多的线程是哪个

测试程序如下：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <windows.h>  DWORD WINAPI thread\_fun1(LPVOID n)  {  while(1)  {  printf( "thread\_fun1\n" );  }  }  DWORD WINAPI thread\_fun2(LPVOID n)  {  Sleep(10000);  }  DWORD WINAPI thread\_fun3(LPVOID n)  {  Sleep(5000);  }  int main()  {  HANDLE hThrd;  DWORD threadId;    hThrd = CreateThread(NULL,  0,  thread\_fun1,  (LPVOID)1,  0,  &threadId);  printf("Start the first thread［%d］\n",threadId);    hThrd = CreateThread(NULL,  0,  thread\_fun2,  (LPVOID)1,  0,  &threadId);  printf("Start the second thread[%d]\n",threadId);  hThrd = CreateThread(NULL,  0,  thread\_fun3,  (LPVOID)1,  0,  &threadId);  printf("Start the third thread[%d]\n",threadId);  while(1)  {  }  } |

2.4判断死锁hang住问题

（1）将windbg附到挂住的进程

或者使用ADPlus -hang -pn dead\_mutex\_lock.exe –o C:\dumps将dump保存下来

（2）用命令~显示所有的线程

（3）然后~ns命令将第n个线程设定为当前线程

（4）最后kb显示当前线程的调用栈

具体例子见：

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  HANDLE hMutex1;  HANDLE hMutex2;  DWORD WINAPI thread\_fun1(LPVOID n)  {  WaitForSingleObject(hMutex1,INFINITE);  Sleep(1000);  WaitForSingleObject(hMutex2,INFINITE);  printf("mutex 1 --> mutex 2 is running\n");  ReleaseMutex(hMutex2);  ReleaseMutex(hMutex1);  }  DWORD WINAPI thread\_fun2(LPVOID n)  {  WaitForSingleObject(hMutex2,INFINITE);  WaitForSingleObject(hMutex1,INFINITE);  printf("mutex 2 ---> mutex 1 is running\n");  ReleaseMutex(hMutex1);  ReleaseMutex(hMutex2);  }  int main(int argc, char argv[] )  {  HANDLE hThrd;  DWORD threadId;  hMutex1=CreateMutex(NULL,FALSE,"tickets1");  hMutex2=CreateMutex(NULL,FALSE,"tickets2");  hThrd = CreateThread(NULL,  0,  thread\_fun1,  (LPVOID)1,  0,  &threadId);  hThrd = CreateThread(NULL,  0,  thread\_fun2,  (LPVOID)1,  0,  &threadId);  Sleep(100000);  } |

八、动态库依赖关系察看

windows下，进程查看器（ProcessExplorer）可以用来查看进程（实时运行）依赖的dll文件；DependencyWalker可以用来查看dll或exe依赖的dll文件。

linux下，ldd可以用来查看bin文件或dll文件依赖的dll