实验二 单步调试器

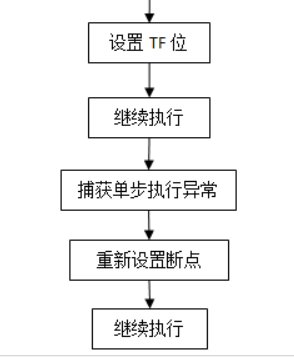
一 实验目的：在Debuger Main Loop示例代码的基础上，完成一个具备单步跟踪功能的调试器开发

二 实验内容：

1. 主进程通过创建远程进程来进行调试
2. 设置eFlag寄存器中的tf标志位，使程序执行一条指令后，产生单步执行异常。
3. 在异常处理过程中，记录程序执行时所有数据（每一步的EIP、八个寄存器的值等）。

三 实验过程

\*流程图：



1. 通过主进程创建子进程（被调试进程）

\*所用函数：CreateProcess()

\*使用说明：

CreateProcess(

TEXT("xx.exe"),//被调试进程

NULL, // 命令行字符串

NULL, // 指向一个NULL结尾的、用来指定可执行模块的宽字节字符串

NULL, // 指向一个SECURITY\_ATTRIBUTES结构体，这个结构体决定是否返回的句柄可以被子进程继承。

false,// 指示新进程是否从调用进程处继承了句柄。

DEBUG\_PROCESS,//如果这个标志被设置，调用进程将被当做一个调试程序，并且新进程会被当做被调试的进程（DEBUG\_EVENT）。系统把被调试程序发生的所有调试事件通知给调试器。调用进程可以调用WaitForDebugEvent函数接收调试信息。

NULL, // 指向一个新进程的环境块。如果此参数为空，新进程使用调用进程的环境

NULL, // 指定子进程的工作路径

&si, // 决定新进程的主窗体如何显示的STARTUPINFO结构体

&pi // 接收新进程的识别信息的PROCESS\_INFORMATION结构体

)

1. 单步调试目标进程
2. 获取被调试进程的主线程的上下文环境

\*所用函数：GetThreadContext()

\*说明：

GetThreadContext(pi.hThread, pContext)//可以获得获取当前CPU寄存器状态的集合

pContext->ContextFlags = CONTEXT\_FULL;//设置CONTEXT变量结构体，指明想要收回哪些寄存器，CONTEXT\_FULL：包含CPU的控制寄存器,比如指今指针,堆栈指针,标志和函数返回地址、用于标识CPU的整数寄存器、用于标识CPU的段寄存器

1. 将eFlag寄存器中的tf标志位设置为1，使被调试进程产生单步异常

\*所用函数：SetThreadContext()

\*说明：

context.EFlags |= 0x100;**//16进制的100转化为二进制为1 0000 0000，而TF位是EFLAGS寄存器中的第8位（从0开始算），通过异或运算可以使TF位为1**

SetThreadContext(pi.hThread, pContext)**//重新设置线程寄存器的值**

1. 捕获异常

\*所用函数：WaitForDebugEvent()

\*说明：

WaitForDebugEvent(DebugEv, INFINITE);//等待调试事件通知的到来，INFINITE表示持续等待（the function does not return until a debugging event has occurred.）

switch (DebugEv->dwDebugEventCode)//描述了调试事件的类型，总共有9类调试事件

{

case EXCEPTION\_DEBUG\_EVENT://被调试事件发生异常

switch (DebugEv->u.Exception.ExceptionRecord.ExceptionCode)

{

case EXCEPTION\_SINGLE\_STEP://单步调试，tf位为1时单步中断

single = 1;

step\_through(context);//将进程上下文的相关信息输出

break;

...

}

...

}

1. 在异常处理过程中，记录程序执行时数据（如EIP和八个寄存器的值）

\*所用函数：void step\_through(CONTEXT context)- 自编

\*说明：

void step\_through(CONTEXT context)

{

cout << "EIP :"<< uppercase << hex << context.Eip << endl; //输出EIP，存储的是下次要执行的指令的地址

cout << endl;

//八个寄存器值

cout << "寄存器值 :" << endl;

cout << "eax：" << uppercase << hex << context.Eax << endl;//通用寄存器（eax、ebx、ecx、edx）

cout << "ebx：" << uppercase << hex << context.Ebx << endl;

cout << "ecx：" << uppercase << hex << context.Ecx << endl;

cout << "edx：" << uppercase << hex << context.Edx << endl;

cout << "esi：" << uppercase << hex << context.Esi << endl;//索引寄存器

cout << "edi：" << uppercase << hex << context.Edi << endl;

cout << "esp：" << uppercase << hex << context.Esp << endl;//栈顶指针

cout << "ebp：" << uppercase << hex << context.Ebp << endl;

cout << endl;

cout << "显示内存内容："<<endl;

unsigned int address = 0;

unsigned int length = 128;

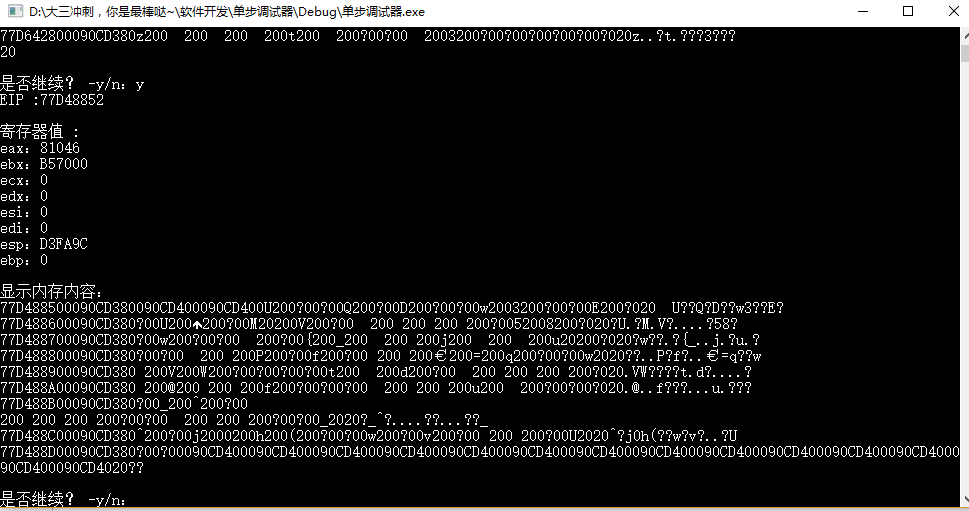
address = context.Eip;

DumpHex(address, length);

cout << endl << endl;

}

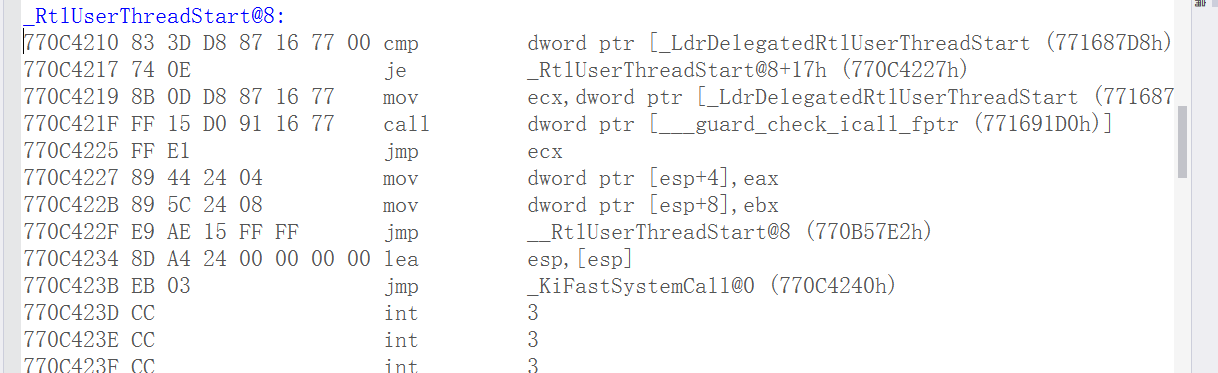
四 实验效果



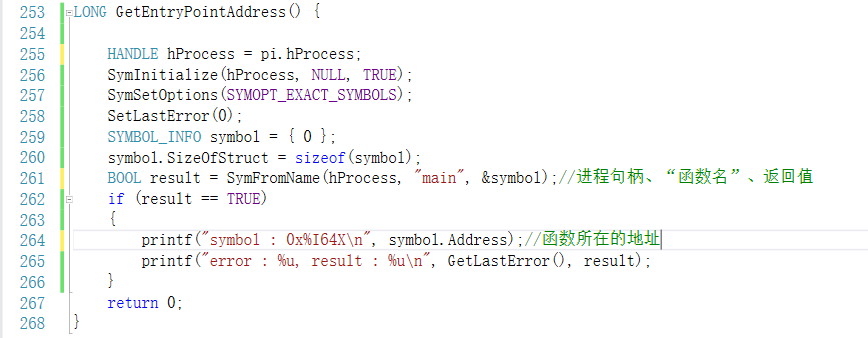
五 问题

1. 被调试程序main()函数地址的确定

Q：调试时下的第一个断点位置如下：

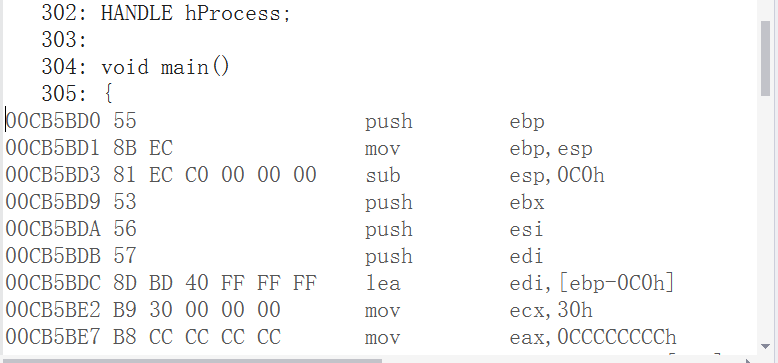


并没有定位到被调试程序的入口函数处，当有符号表文件时，考虑利用API函数SymFromName 获取被调试程序main函数地址，进而从main函数地址开始下断点：



存在的问题是SymFromName函数中当传入被调试进程句柄时无法获取main地址，而当进程句柄通过GetCurrentProcess()获得当前进程句柄时，可以获得调试进程的main函数地址





A：解决思路：可能是缺少被调试程序符号表？

五 实验收获

在本实验中，初步掌握了调试器的基本原理，学会了以调试状态创建目标进程，并循环监听调试事件，遇到断点后挂起进程进行对应操作。另外，通过学习进程产生单步中断的原理，挂起进程后对上下文进行修改，达到单步调试的效果。

六 附录（实验代码）

见附件

七 参考资料

https://www.cnblogs.com/zplutor/archive/2011/04/02/2004097.html