Shanghai Jiaotong University

Multithreaded Program Debug Visualization Helper

Final Report

Documented By: Jessie Zhang

Table of Contents

1. Abstract 4

2. Introduction 4

3. The Goal – Project Scope 5

4. Technologies used in the solution's architecture 5
   - *4.1 Microsoft Research "Detours" Technology* 5
     - 4.1.1 Main idea 5
     - 4.1.2 Implementation 5
   - *4.2 Platform Invoke (P/Invoke)* 6
     - 4.2.1 Introduction 6
     - 4.2.2 Calling a DLL Export Directly from C# 6
     - 4.2.3 Example 7

5. The solution 7
   - *5.1 Architecture* 7
   - *5.2 Thread View* 8
   - *5.3 Implementation* 9
     - 5.3.1 Function List 9
     - 5.3.2 Code snippets 11

6. Project Evaluation 14
   - *6.1 Number of Lines of Code* 14
   - *6.2 Job Division* 15
   - *6.3 Problems Encountered* 15

7. References 16

Acknowledgement

First, I must thank my professor Dr. Zhengwei Qi. He gave me a lot of help, encouragement and support.

Second, I want to give credit to our TA Fuyuan Zhang. He gave me useful advice along the way. Without his help, this project would have never gone so smoothly.

Third, I thank all the teachers who are responsible for the whole arrangement of the summer projects at school. They have provided us with good environment for us to develop the software.

Finally, and most importantly, I thank all the team members, who worked with me during the past two months. It takes their diligent work, patience and support to accomplish this project.

Final Report

1. Abstract

This project is the summer project at software engineering school, SJTU in July 2008. This project provides a tool helping the programmer to debug their programs. It is based on a Microsoft-research technology called Detours.

The final product of this project is an add-in for Microsoft Visual Studio 2005 which visualizes the track of running threads with the help of interception of Win32 API function calls.

This project was created by:

Zhang Can	zhangcansjtu@yahoo.com.cn	Junior student of software engineering school, SJTU
Fang Wei	zhongguotu2005@yahoo.com.cn	Junior student of software engineering school, SJTU
Ji Xiaofei	alonewolf_1217@sina.com	Junior student of software engineering school, SJTU
Niu Xingman		Junior student of software engineering school, SJTU

2. Introduction

Nowadays, with the requirements of high efficiency and high performance, multithreaded programs are used more and more often in different kinds of projects. However, it also brings difficulty for programmers to debug these programs. As we all know, we may get different results if we run the multithreaded programs for a second time. Also, the Microsoft Visual Studio 2005(VS2005) does not provide an easy-use tool for the programmers to debug the multithreaded programs.

As a result, in this project we want to create an add-in for VS2005, which will help the programmer debug with multi-threaded programs by visualizing the Win32 API functions invoked by running threads in the program.

In this project we use “Detours”, which is a library for instrumenting arbitrary Win32 functions on x86 machines. Detours intercepts Win32 functions by re-writing target function images.

While prior researchers have used binary rewriting to insert debugging and profiling instrumentation, to our knowledge, Detours is the first package on any platform to logically preserve the non-instrmented target function (callable through a trampoline) as a subroutine for use by the instrumentation. Using the unique trampoline design is crucial for extending existing binary software.

Since the project’s scope is bounded by the academic research, we just implemented the interception of some of the Win32 API function calls. We mainly concentrated in understanding the technologies that we involved

and to produce a working prototype of an add-in for VS2005 that displays the track of running threads.

3. The Goal – Project Scope

The goal that we set was to create an add-in for VS2005, which intercepts the Win32 API function calls to display the track of running threads.
The main interest in this project was to get familiar with the Detours technology and to develop a working prototype of a real world, usable add-in application.

4. Technologies used in the solution's architecture

4.1 Microsoft Research "Detours" Technology

Reference: G. Hunt, D.B., Detours: Binary Interception of Win32 Functions. 1999, Microsoft Research.

4.1.1 Main idea

The Detours technology was conceived in the Microsoft Research labs. This is, yet, another method for intercepting method calls. Many techniques exist to uphold this task, though, this mechanism is no intrusive – the executable is not altered; only its memory image is changed. This way you do not need to compile the code again (no need for sources).

4.1.2 Implementation

The Detours library facilitates the interception of function calls. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the *target function* with an unconditional jump to the user-provided *detour function.* Instructions from the *target function* are preserved in a *trampoline function*. The trampoline consists of the instructions removed from the target function and an unconditional branch to the remainder of the *target function.*
Figure 4-1 shows the logical flow control for function invocation with and without interception.

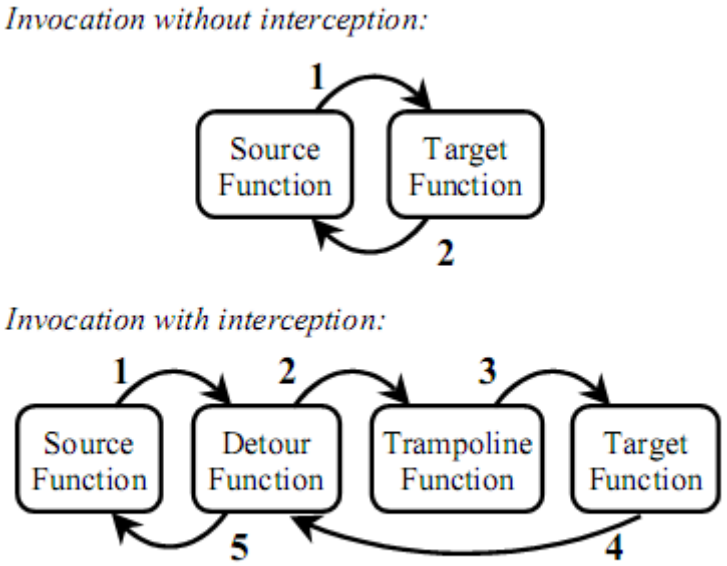


Figure. 4-1 Invocation with and without interception

4.2 Platform Invoke (P/Invoke)

Reference: http://msdn.microsoft.com/en-us/library/aa446536.aspx

http://msdn.microsoft.com/en-us/library/aa288468.aspx

4.2.1 Introduction

While Microsoft has incorporated much functionality into the .NET Framework class libraries, significant additional functionality resides outside of the managed world of .NET. COM interoperability is necessary in order to access native system APIs, such as shell integration, DirectX, Microsoft Office, and the Windows Registry, as well as custom legacy COM objects. COM interoperability in .NET can be a tricky issue for developers, who have to deal with issues such as figuring out the appropriate data types to use and marshalling data between managed and unmanaged code.

.NET provides access to COM components through its P/Invoke facility. P/Invoke allows developers to invoke native unmanaged methods from managed code.

4.2.2 Calling a DLL Export Directly from C#

To declare a method as having an implementation from a DLL export, do the following:

- Declare the method with the **static** and **extern** C# keywords.

- Attach the **DllImport** attribute to the method. The **DllImport** attribute allows you to specify the name of the DLL that contains the method. The common practice is to name the C# method the same as the exported method, but you can also use a different name for the C# method.
- Optionally, specify custom marshaling information for the method's parameters and return value, which will override the .NET Framework default marshaling.

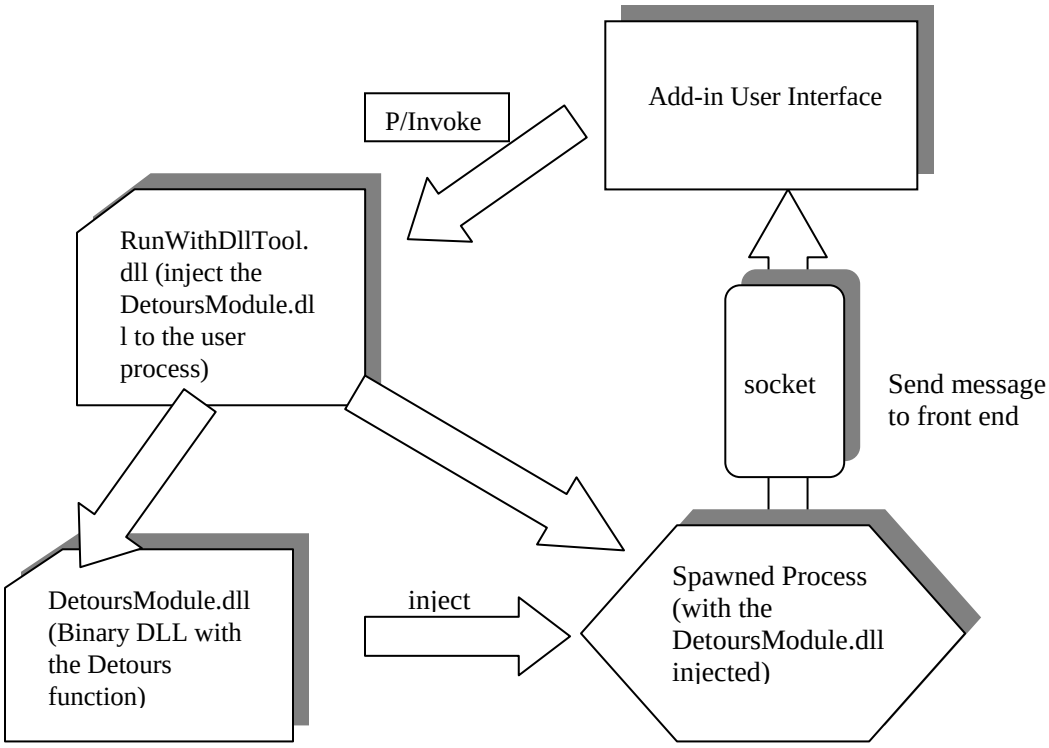
4.2.3 Example

```
//PInvokeTest.cs
using System;
using System.Runtime.InteropServices;

class PlatformInvokeTest
{
   [DllImport("msvcrt.dll")]
   public static extern int puts(string c);

   [DllImport("msvcrt.dll")]
   internal static extern int _flushall();

   public static void Main()
   {
      puts("Test");
      _flushall();
   }
}
```

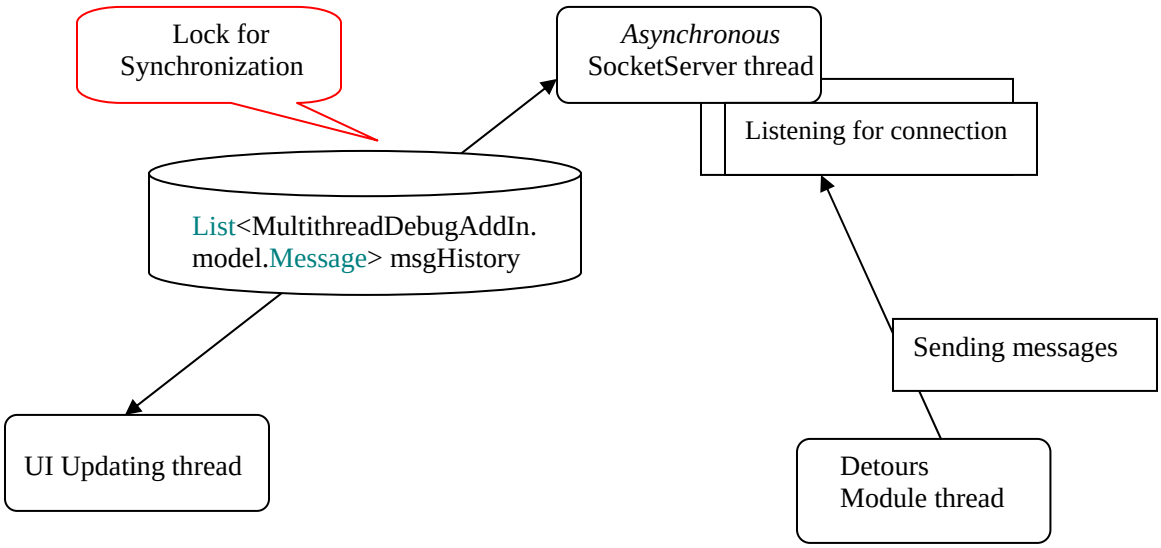
5. The solution

5.1 Architecture

The following diagram displays the grand picture.



5.2 Thread View



5.3 Implementation

5.3.1 Function List

The Win32 API function calls we intercept in this project are as follows:

CreateThread	WaitForSingleObject
Sleep	ExitThread
CreateEvent	CreateMutex
CreateSemaphore	TerminateThread
SuspendThread	ResumeThread
ReleaseMutex	ReleaseSemaphore
SetEvent	ResetEvent
CloseHandle	WaitForMultipleObjects

These functions are divided into six groups:

CreateThreadMsgType

CreateThread

CreateResourceMsgType

CreateEvent
CreateMutex
CreateSemaphore

HandleManipulateMsgType

TerminateThread
SuspendThread
ResumeThread
ReleaseMutex
ReleaseSemaphore
SetEvent
ResetEvent
CloseHandle

WaitMsgType

WaitForSingleObject
WaitForMultipleObjects

SelfManipulateMsgType

Sleep
ExitThread

MainThreadCreateMsgType

CreateThread

These six *Message Types* are the most important data structures of the whole project. They all inherit from one basic class – Message.

Each *Message Type* is defined as follows:

```
class Message
{
public:
        MessageName msgName;
        MessageType msgType;
        unsigned __int64 time; // in cpu cycle
        DWORD invokerThreadId; // the current invoker thread id
};
```

```
class CreateThreadMsg : public Message
{
public:
        HANDLE handle;
        HandleType handleType;
        DWORD threadId;
        ThreadState state;
};
```

```
class CreateResourceMsg : public Message
{
public:
        HANDLE handle;
        HandleType handleType;
        std::string name;
        ResourceState state;
};
```

```
class HandleManipulateMsg : public Message
{
public:
        HANDLE handle;
        int handleState; // ThreadState or ResourceState
};
```

```
class WaitMsg : public Message
{
public:
        int numOfObjects;
        const HANDLE* handles;
        ThreadState state;
};
```

```
class SelfManipulateMsg : public Message
{
public:
        ThreadState state;
};
```

5.3.2 Code snippets

- Using Detours

Basically, three steps are necessary in using Detours.

First, declare a new function pointer to point to the original Win32 API function.

```
static HANDLE (WINAPI * TrueCreateThread)(LPSECURITY_ATTRIBUTES lpThreadAttributes,
                                          DWORD dwStackSize,
                                          LPTHREAD_START_ROUTINE lpStartAddress,
                                          LPVOID lpParameter,
                                          DWORD dwCreationFlags,
                                          LPDWORD lpThreadId)
= CreateThread;
```

function pointer

original function

Second, define your own function.

```
//1.detoured method for Win32 API CreateThread
HANDLE WINAPI MyCreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
                                          DWORD dwStackSize,
                                          LPTHREAD_START_ROUTINE lpStartAddress,
                                          LPVOID lpParameter,
                                          DWORD dwCreationFlags,
                                          LPDWORD lpThreadId)
{
        printf("Detoured CreateThread function\n");
        return TrueCreateThread(lpThreadAttributes,
                                dwStackSize,
                                lpStartAddress,
                                lpParameter,
                                dwCreationFlags,
                                lpThreadId);
}
```

Call original function

Third, attach your function to replace the original function by calling *DetourAttach* method, and detach your function by calling *DetourDetach* method.

```
BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
   LONG error;
   (void)hinst;
   (void)reserved;

   if (dwReason == DLL_PROCESS_ATTACH) {
      DetourRestoreAfterWith();
      DetourTransactionBegin();
      DetourUpdateThread(GetCurrentThread());

      /*********************************************************/
      /* attach all the detoured Win32 API */
      DetourAttach(&(PVOID&)TrueCreateThread, MyCreateThread);
      error = DetourTransactionCommit();
   }
   else if (dwReason == DLL_PROCESS_DETACH) {
      DetourTransactionBegin();
      DetourUpdateThread(GetCurrentThread());

      /*********************************************************/
      /* detach all the detoured Win32 API */
      DetourDetach(&(PVOID&)TrueCreateThread, MyCreateThread);
      error = DetourTransactionCommit();
   }

   return true;

}
```

- Socket Communication

The user interface of the add-in is implemented in C#.net, and running in the *devenv.exe* process, while the detours interception module, which is implemented in C++, is attached to the user program as a dll. Therefore, there should be a way for them to communicate with each other. As mentioned above, we can call C++ dlls from C#.net code using P/Invoke. Also, we can use *socket* technology for the communication of two processes.

Add-in Interface (Socket Server):

*Use an **Asynchronous** socket to listen for connections:*

(if you want to know more, turn to the SocketServer.cs file in the MultithreadDebugAddIn project.)

```
public static void StartListening()
{
   // Data buffer for incoming data.
   byte[] bytes = new byte[1024];

   // Establish the local endpoint for the socket.
   IPAddress localAddr = IPAddress.Parse("127.0.0.1");
   IPEndPoint localEndPoint = new IPEndPoint(localAddr, PORT);
   // create a tcp/ip socket
   Socket listener = new Socket(AddressFamily.InterNetwork,
      SocketType.Stream, ProtocolType.Tcp);

   try
   {
      // Bind the socket to the local endpoint and listen for incoming connections.
      listener.Bind(localEndPoint);
      listener.Listen(100);

      while (!done)
      {
         // Set the event to nonsignaled state.
         allDone.Reset();

         // Start an asynchronous socket to listen for connections.
         // Begin establishing connection
         listener.BeginAccept(
            new AsyncCallback(AcceptCallback),
            listener);

         // Wait until a connection is made before continuing.
         allDone.WaitOne();
      }

   }
   catch (Exception e)
   {
      Console.WriteLine(e.ToString());
   }
}
```

Asynchronous socket

Detours Module (Socket Client):

Use <winsock2.h> APIs to establish socket connections.

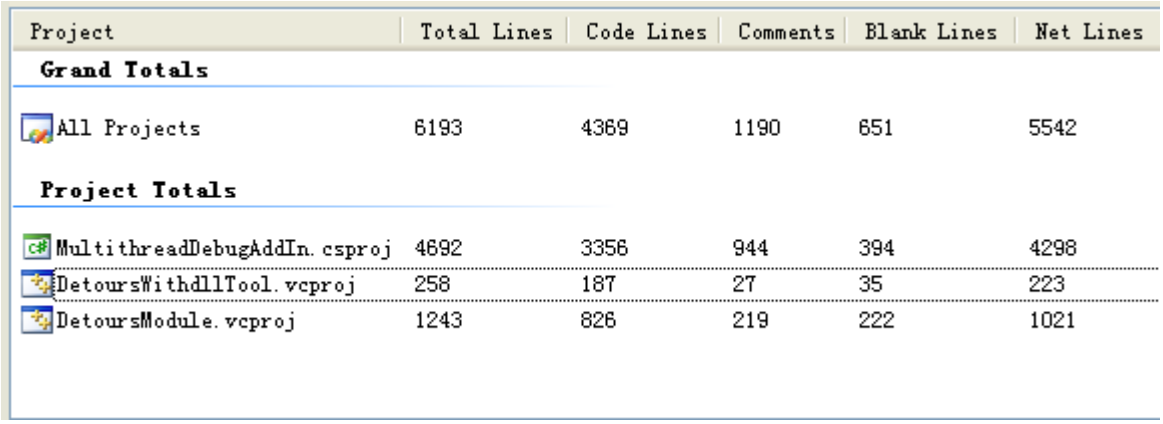
(if you want to know more, turn to the initSock.h and tools.cpp files in the DetoursModule project.)

```
bool sendData(std::string data)
{
	//set the flag variable to be 1
	socketFlag = 1;

	//first,set server address
	sockaddr_in servAddr;
	servAddr.sin_family = AF_INET;
	servAddr.sin_port = htons(PORT);
	servAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");

	//second,prepare the client socket
	SOCKET s = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	if(s == INVALID_SOCKET)
	{
		printf(" Failed socket() \n");
		return false;
	}
	if(::connect(s, (sockaddr*)&servAddr, sizeof(servAddr)) == -1)
	{
		printf(" Failed connect() \n");
		return false;
	}

	//third,send the data
	const char* dataSent = data.c_str();
	int dataLength = data.length();

	send(s,dataSent,dataLength,0);

	//forth,close the client socket
	closesocket(s);

	//reset the flag variable to be the default value
	socketFlag = 0;
	return true;
}
```

6. Project Evaluation

6.1 Number of Lines of Code

The result is calculated by Line Counter add-in for VS2005.



Project	Total Lines	Code Lines	Comments	Blank Lines	Net Lines
Grand Totals					
All Projects	6193	4369	1190	651	5542
Project Totals					
MultithreadDebugAddIn.csproj	4692	3356	944	394	4298
DetoursWithdllTool.vcproj	258	187	27	35	223
DetoursModule.vcproj	1243	826	219	222	1021

6.2 Job Division

Member	Main Job	Project	Total Lines	Comments
Fang Wei	Add-in GUI Design & Implementation	MultithreadDebugAddIn	1389	220
Ji Xiaofei	Detours Module Design & Implementation	DetoursModule	820	110
Niu Xingman	Detours Module Data Structure Design & Implementation	DetoursModule	717	170
Zhang Can	Architecture, Socket Communication, Testing	MultithreadDebugAddIn, DetoursWithdllTool	3267	690

6.3 Problems Encountered

- Time accuracy

Typically, we use functions like GetSystemTime() or GetLocalTime() to get the system time.
However, it's not suitable for our project, for the threads are running too fast so that the values returned by these methods are almost the same.

Finally, we use the GetCycleCount() function to get the number of CPU cycles passed since computer startup to represent how much time has passed.

- Process communication

At first, we didn't realize that there are two different processes running after starting up the add-in. Therefore, we simply use the P/Invoke technique to invoke method written in C++ from C#.net code.
However, we found that the function calls didn't succeed at all. At last, after we printed each process ID in the C++ code and the C# code, we found that they are actually in two different processes.
The two processes are:
Process 1: Add-in runs in the devenv.exe process when VS2005 starts up
Process 2: DetoursModule.dll attaches to user program when the add-in is activated
The difficulty is how to do inter-process communication?

Finally, we use socket techniques to make the communication possible. The solution seems easy and explicit, but it also takes us some effort to make it work. Synchronous socket server is not suitable for our project, for the messages received from the socket clients are in high speed. Only the asynchronous socket works.

- Distinguishing Function calls

We have to decide whether the function calls are from the debugging user program. For those from user program we have to notify the front end, while for those from system calls or other sources we have to ignore them.
In our program, we use socket to do the communication. Though in the code there are no explicit function calls to the detoured functions, the messages sent are not right. I think that the socket call some detoured functions implicitly when calling other functions. We use a TLS(thread local storage) variable - socketFlag to separate the calls from socket and from the real user program.

7. References

[1] G. Hunt, D.B., Detours: Binary Interception of Win32 Functions. 1999, Microsoft Research.
[2] An Introduction to P/Invoke and Marshaling on the Microsoft .NET Compact Framework
http://msdn.microsoft.com/en-us/library/aa446536.aspx
[3] Platform Invoke Tutorial
http://msdn.microsoft.com/en-us/library/aa288468.aspx