limbioliong

limbioliong.wordpress.com/2022/01/22/understanding-windows-structured-exception-handling-part-3-under-the-hood

January 22, 2022

//

you're reading...

C++, SEHException

Understanding Windows Structured Exception Handling Part 3 – Under The Hood

Posted by Lim Bio Liong · January 22, 2022 · Leave a comment

Introduction

1. In parts 1 and 2, we have studied how SEH is constructed in a function by the Visual C++ compiler. We have also seen how __try/__except/__finally blocks are tracked and their means of execution made possible via a SCOPETABLE_ENTRY array.

2. In this part 3 of this multi-part series of articles on Win32 Structured Exception Handling, we will look under the hood and study how the OS performs SEH when an exception is raised.

3. We shall be studying the Visual C++ provided __except_handler3() function which is the default exception handler put in place by the compiler for a function which contains one or more __try/__except/__finally blocks.

4. We shall also talk about a complex procedure known as stack unwinding.

__except_handler3()

1. The following are the characteristics of __except_handler3() :

- It is a generic SEH handler provided by the Visual C++ compiler.
- It is not a Windows API.
- When called to action, it works with the SEH Frame of one function only.
- This is so even though __except_handler3() may be installed in multiple functions.
- It calls __except filter functions of a SEH Frame and evaluates their return values.
- It executes __except and __finally blocks.
- Because it is generic in nature, it cannot assume intimate knowledge of any filter functions, or __except and __finally blocks.

2. A pseudocode for __except_handler3() is given in Matt Pietrek's article. However, I personally found James McNellis' pseudocode much better. It can be found in this YouTube video at 26:57. The slides of the CppCon Presentation can be found here.

3. I have revised James McNellis' pseudocode based on my understanding of how __except_handler3() works. This is listed below :

```
// The pseudocode for __except_handler3().
EXCEPTION_DISPOSITION NTAPI __except_handler3_pseudocode
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    PEH4_EXCEPTION_REGISTRATION_RECORD RegistrationNode = NULL;

    // Obtain the RegistrationNode of the Current SEH Frame via offset from the
EstablisherFrame pointer.
    RegistrationNode
        = (PEH4_EXCEPTION_REGISTRATION_RECORD)
          ((PCHAR)EstablisherFrame - FIELD_OFFSET(EH4_EXCEPTION_REGISTRATION_RECORD,
SubRecord));
    // Declare a EXCEPTION_POINTERS structure,
    // fill its members with actual values from the function parameter,
    // and then assign its address to RegistrationNode's ExceptionPointers member.
    EXCEPTION_POINTERS ExceptionPointers{ pExceptionRecord, pContextRecord };
    RegistrationNode->ExceptionPointers = &ExceptionPointers;

    if (pExceptionRecord->ExceptionFlags includes EXCEPTION_UNWINDING == false)
    {
        // __except_handler3() is being called to perform Exception Handling
        // in the function that it is registered for.

        // Get a pointer to teh SCOPETABLE_ENTRY array of the Current Function
        // in which __except_handler3() has been registered as the SEH Handler.
        SCOPETABLE_ENTRY* pScopeTable = (SCOPETABLE_ENTRY*)(RegistrationNode-
>EncodedScopeTable);

        // Loop through the SCOPETABLE_ENTRIES of the Current Function
        // in which __except_handler3() has been registered as the SEH Handler.
        //
        // We are now searching for any available __except block Filter Function
        // starting from the current TryLevel.
        for
        (
            int i = RegistrationNode->TryLevel; // Start from the current TryLevel
            i != -1;                            // and work our way downwards
            i = pScopeTable[i].EnclosingLevel   // until we reach TryLevel == -1.
        {
            if (pScopeTable[i].lpfnFilter == NULL)
            {
                // The current TryLevel does not have an __except Filter Function.
                // We skip this TryLevel and continue to the TryLevel of the
                // Enclosing __try block.
                continue;
            }
```

```
            // If there is a Filter Function, call it and get the result.
            int iFilterResult = pScopeTable[i].lpfnFilter();

            switch (iFilterResult)
            {
                case EXCEPTION_CONTINUE_SEARCH:
                {
                    // Move on to the next enclosing TryLevel's Scope Table.
                    continue;
                }

                case EXCEPTION_CONTINUE_EXECUTION:
                {
                    // The Filter has resolved the Exception Cause.
                    // We can now continue execution at the point
                    // of the original Exception.
                    return ExceptionContinueExecution;
                }

                case EXCEPTION_EXECUTE_HANDLER:
                {
                    // First do a Global Unwind. This is to inform all SEH Exception
Handlers
                    // which have been installed -AFTER- the current SEH Handler to
do Unwinding.
                    //
                    // Here, RegistrationNode->SubRecord is the TIB's ExceptionList
Item
                    // which points to the current SEH Exception Handler.
                    //
                    // DoGlobalUnwind() will perform the following :
                    // 1. Get each of these handlers to do Local Unwinding.
                    // 2. Uninstall each of these handlers off the TIB's
ExceptionList.
                    //
                    // Note that the first parameter indicates to DoGlobalUnwind() to
                    // do Unwinding for all SEH Handlers -UP TO- the current SEH
Handler.
                    //
                    DoGlobalUnwind(&(RegistrationNode->SubRecord), pExceptionRecord);

                    // Next, we do a Local Unwind. This is to ensure that if there
are any
                    // __finally blocks installed -AFTER- the current TryLevel (i.e.
of a
                    // greater TryLevel value), they are all to be executed.
                    DoLocalUnwind(&(RegistrationNode->SubRecord), RegistrationNode-
>TryLevel);

                    // Do a Non-Local-Goto to call the __except handler (i.e.
pScopeTable[i].lpfnHandler())
                    // This call must not return here.
```

```
                    CallAndNeverReturn(pScopeTable[i].lpfnHandler());

                    break;
                }
            }
        }
    }
    else
    {
        // __except_handler3() is being called to perform Local Unwind
        // in the function that it is registered for.
        // The Local Unwind() will call the __finally blocks from the
        // highest TryLevel down to -1.
        DoLocalUnwind(&(RegistrationNode->SubRecord), -1);
    }

    return ExceptionContinueSearch;
}
```

4. __except_handler3() is a generic exception handler which serves more than one purpose :

- It can be called to perform Exception Handling for the Current SEH Frame.
- It can be called to perform local unwinding.

Note well that when an exception occurs inside a function, the OS activates the first SEH Exception Handler Registered in the TIB->ExceptionList. The OS does not know in which function this SEH Exception Handler was registered in.

Recall from Part 1 that a SEH Exception Handler is to return a value from the EXCEPTION_DISPOSITION enum :

```
// Exception disposition return values
typedef enum _EXCEPTION_DISPOSITION
{
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

Hence __except_handler3() must eventually return one of the above values.

5. The following is a summary analysis of the pseudocode :

- __except_handler3() first obtains a pointer to the EH4_EXCEPTION_REGISTRATION_RECORD of the Current SEH Frame. This pointer is set to the local pointer RegistrationNode.

- The Current SEH Frame being setup inside the C/C++ function in which a __try/__except/__finally block is defined and for which the __except_handler3() function is invoked.
- Next, a local EXCEPTION_POINTERS structure is defined and its members filled with actual pointers from the parameters. A pointer to this structure is then set as the ExceptionPointers member of RegistrationNode (see point 6 below).
- ExceptionFlags is then checked to see if it includes the EXCEPTION_UNWINDING flag. If so, it means that __except_handler3() is being invoked to do Local Unwinding for the Current SEH Frame. DoLocalUnwind() is called to perform this (see point 7 below).
- If Unwinding is not called for, we assume that __except_handler3() is being invoked to perform Exception Handling for the current SEH Frame.
- Exception Handling will involve iterating through the SCOPETABLE_ENTRY items, calling the associated Filter Functions of __except blocks and evaluating their return values.
- The iteration of SCOPETABLE_ENTRY items begins at the index signified by the current RegistrationNode->TryLevel. This is the TryLevel which is most relevant when the Exception occurred (see point 8 below).
- If the Filter Function pointer is NULL, it means that the current __try block is a __try/__finally block. It is skipped since we are now searching for an __except filter.
- The iteration of the SCOPETABLE_ENTRY array continues until we find an __except Filter Function. Once one is found, it is called.
- When a Filter Function returns, its return value is evaluated.
    - If EXCEPTION_CONTINUE_SEARCH is returned, __except_handler3() moves onto the next SCOPETABLE_ENTRY to check for and call another Filter Function in the same SEH Frame (see point 9 and point 10 below).
    - If EXCEPTION_CONTINUE_EXECUTION is returned, it means that the Filter Function has fixed the cause of the Exception and is instructing the Handler to continue execution at the original point of the Exception. In the case __except_handler3() will return ExceptionContinueExecution. __except_handler3() is deemed to have performed its job and the OS will take over from there.
    - If EXCEPTION_EXECUTE_HANDLER is returned, things get a little more complicated. See Executing an __except Block for more details.

6. The pExceptionRecord and pContextRecord parameters must be set in the RegistrationNode->ExceptionPointers member in order that it can be referenced in later calls to Global and Local Unwinding.

7. Local Unwinding is a procedure to call the __finally blocks of a SEH Frame. It can be performed during 2 occasions :

- When an exception is being handled by a SEH Handler and there are higher level SEH handlers which have declined the exception handling.
- When a SEH Handler has decided to handle an exception (at a TryLevel) and there are higher TryLevels further on which may contain __finally blocks that will need to be executed.

See an Demo Global Unwinding and Demo Local Unwinding for more details.

8. The Current RegistrationNode->TryLevel is the most relevant TryLevel in a SEH Frame when an exception occurs. Take the following sample code :

```
void ExceptionCausingFunction()
{
    int* pInt = NULL;
    *pInt = 100;
}

void DemoMostRelevantTryLevel()
{
    printf("TryLevel == -1. No associated ScopeEntry.\r\n");

    __try
    {
        printf("TryLevel == 0. ScopeEntry[0].\r\n");

        __try
        {
            printf("TryLevel == 1. ScopeEntry[1].\r\n");

            ExceptionCausingFunction();

            __try
            {
                printf("TryLevel == 2. ScopeEntry[2].\r\n");
            }
            __except (EXCEPTION_EXECUTE_HANDLER)
            {
                printf("__except for TryLevel == 2. ScopeEntry[2].\r\n");
            }

        }
        __except (EXCEPTION_EXECUTE_HANDLER)
        {
            printf("__except for TryLevel == 1. ScopeEntry[1].\r\n");
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");
    }

    printf("TryLevel == -1. No associated ScopeEntry.\r\n");
}
```

In the above code, when DemoMostRelevantTryLevel() runs, it calls ExceptionCausingFunction(). We know that an exception will be raised in ExceptionCausingFunction(). As there are no exception handlers registered for ExceptionCausingFunction(), the OS will invoke the __except_handler3() registered for DemoMostRelevantTryLevel().

When the __except_handler3() of DemoMostRelevantTryLevel() is run, the most relevant TryLevel is 1. This is because 1 is the TryLevel when the exception occurred. It is certainly not necessarily the highest TryLevel value.

9. Note that as the SCOPETABLE_ENTRY array is traversed, it is iterated from a higher TryLevel to a lower one. However, we cannot assume that the TryLevel is always decremented by 1. Observe the following code snippet :

```
...
...
...
__try
{
    printf("TryLevel == 3. ScopeEntry[3].\r\n");

    __try
    {
        printf("TryLevel == 4. ScopeEntry[4].\r\n");

        int* pInt = NULL;
        *pInt = 100;
    }
    __finally
    {
        printf("__finally for TryLevel == 4. ScopeEntry[4].\r\n");
    }
}
__finally
{
    printf("__finally for TryLevel == 3. ScopeEntry[3].\r\n");
}

printf("TryLevel == -1. No associated ScopeEntry.\r\n");
```

In the above code snippet, the __try block which is associated with TryLevel 3 has an Enclosing Level value of -1 and not 2. Hence it is important to traverse to the next *Enclosing Level*.

10. Also note that the EXCEPTION_EXECUTE_HANDLER, EXCEPTION_CONTINUE_SEARCH and EXCEPTION_CONTINUE_EXECUTION are values returned by the Filter Function and used internally by the Exception Handler (e.g. __except_handler3()). These are not returned to the OS. The OS is only interested in a value from the EXCEPTION_DISPOSITION enum :

```
// Exception disposition return values
typedef enum _EXCEPTION_DISPOSITION
{
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

11. EXCEPTION_EXECUTE_HANDLER, EXCEPTION_CONTINUE_SEARCH and EXCEPTION_CONTINUE_EXECUTION may be considered constant values defined by the Visual C++ compiler (as is __except_handler3()). Other compilers may define other constants and exception handlers and may even define language syntax different from __try/__except/__finally.

Executing an __except Block

1. When an __except filter returns EXCEPTION_EXECUTE_HANDLER, its associated block is to be executed. But before the __except block can be called, several things need to first be performed :

- First, Global Unwinding must take place.
- This must be followed by Local Winding.

Only after the above 2 actions are performed can the __except block be called.

2. We will be going in-depth on Global and Local Unwinding in the next sections. For now, I want to make certain concepts clear.

3. When the Exception Handler of a SEH Frame decides to handle an exception, it may not be the first Handler listed in the current TIB's ExceptionList. The TIB's ExceptionList is a Last-In-First-Out (LIFO) linked list. A SEH Frame registered higher in the TIB ExceptionList is a Frame that is installed sequentially later than a lower one.

4. In other words, a lower SEH Frame belongs to a function which either directly or indirectly calls a later function with a higher SEH Frame. We shall see this in more detail in the next section.

5. After Global and Local Unwinding has been performed, we call the exception handler, i.e. SCOPETABLE_ENTRY[i].lpfnHandler() where "i" is the appropriate TryLevel.

6. This call is special in the sense that it will never return. This is logical since after an __except block has been executed, the flow of the program must continue after the block and not return to the exception handler.

7. At the destination address, the stack pointer is immediately updated to the saved one which was setup in the EH4_EXCEPTION_REGISTRATION_RECORD at the construction of the SEH Frame. Recall in Part 2, the EH4_EXCEPTION_REGISTRATION_RECORD structure is defined as :

```
typedef struct _EH4_EXCEPTION_REGISTRATION_RECORD
{
    PVOID                           SavedESP;
    PEXCEPTION_POINTERS             ExceptionPointers;
    EXCEPTION_REGISTRATION_RECORD   SubRecord;
    UINT_PTR                        EncodedScopeTable;
    ULONG                           TryLevel;
} EH4_EXCEPTION_REGISTRATION_RECORD, *PEH4_EXCEPTION_REGISTRATION_RECORD;
```

This SavedESP member is given its value when the SEH Frame is setup :

```
?DemoLocalUnwind@@YAXXZ PROC                            ; DemoLocalUnwind

; 115  : {

        push    ebp
        mov     ebp, esp
        push    -1
        push    OFFSET __sehtable$?DemoLocalUnwind@@YAXXZ
        push    OFFSET __except_handler3
        mov     eax, DWORD PTR fs:0
        push    eax
        mov     DWORD PTR fs:0, esp
        sub     esp, 8
        push    ebx
        push    esi
        push    edi
        mov     DWORD PTR __$SEHRec$[ebp], esp
```

8. This SavedESP is then used to restore the esp register at the start of the __except block :

```
; 140  :     __except (EXCEPTION_EXECUTE_HANDLER)

        mov     eax, 1
$LN9@DemoLocalU:
$LN21@DemoLocalU:
        ret     0
$LN8@DemoLocalU:
        mov     esp, DWORD PTR __$SEHRec$[ebp]

; 141  :     {
; 142  :         printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");

        push    OFFSET ??_C@_0CN@NNAANJNA@__except?5for?5TryLevel?5?$DN?$DN?50?4?
5Sco@
        call    _printf
        add     esp, 4

; 138  :         }
; 139  :     }
```

This esp restoration is necessary in order that the __except block executes in the correct Stack Frame.

9. However, how the flow of control is passed to the start of the __except block is not entirely clear to me. __except_handler3() calls a mysterious and undocumented function named _NLG_Notify() which causes control to be transferred to a destination address which is specified in the eax register.

10. Another matter that baffles me is how the base pointer register ebp is updated to the stack frame when the __except block is entered (the ebp register is not saved in the EH4_EXCEPTION_REGISTRATION_RECORD structure). The ebp register is just as important as the esp register in order for the low-level assembly language code to access local variables.

11. However, one thing clear is that something known as a “Non-Local-Goto” is performed (connected with the “NLG” in _NLG_Notify()). For this, the C/C++ standard provides the setjmp() and longjmp() functions. While setjmp() and longjmp() are well documented and easily tested and used, it is not clear to me how _NLG_Notify() works.

12. I do hope that any reader who happens to be familiar with the workings of _NLG_Notify() to contact me and enlighten me.

Demo Global Unwinding

1. Global unwinding means only 3 things :

- The SEH Handlers of Frames higher up the TIB's ExceptionList are called with the EXCEPTION_UNWINDING flag.
    This is to enable these Frames to do Local Unwinding (i.e. calling the __finally blocks).
- These SEH Frames are then unregistered from the TIB's ExceptionList.
    This is because they are no longer relevant as far as the Exception that has occurred is concerned.
- When the eventual __except block is run, the stack frames of all functions associated with these unregistered SEH Frames are discarded.
    This is achieved by a simple modification of the stack pointer esp.

2. The following code is adapted from Matt Pietrek's code (MYSEH2.CPP) in his article. It is also an modified version of the TestDivisionByZero01SEH() and MyDivisionByZero01ExceptionRoutine() functions that we saw in Part 1 :

```
#define DISPLAY_EXCEPTION_INFO(pExceptionRecord) \
    printf("An excepton occured at address : [0x%p]. Exception Code : [0x%08X].
Exception Flags : [0x%08X]\r\n", \
        pExceptionRecord->ExceptionAddress, \
        pExceptionRecord->ExceptionCode, \
        pExceptionRecord->ExceptionFlags); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_NONCONTINUABLE) \
            printf(" EXCEPTION_NONCONTINUABLE\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_UNWINDING) \
            printf(" EXCEPTION_UNWINDING\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_EXIT_UNWIND) \
            printf(" EXCEPTION_EXIT_UNWIND\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_STACK_INVALID) \
            printf(" EXCEPTION_STACK_INVALID\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_NESTED_CALL) \
            printf(" EXCEPTION_NESTED_CALL\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_TARGET_UNWIND) \
            printf(" EXCEPTION_TARGET_UNWIND\r\n"); \
\
        if (pExceptionRecord->ExceptionFlags & EXCEPTION_COLLIDED_UNWIND) \
            printf(" EXCEPTION_COLLIDED_UNWIND\r\n");

int g_iDividend = 1000;
int g_iDivisor = 0;

EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyDivisionByZero01ExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    return ExceptionContinueSearch;
}

int TestDivisionByZero01SEH()
{
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();

    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyDivisionByZero01ExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
```

```
    TIB->ExceptionList = &Registration;

    int iValue = g_iDividend / g_iDivisor;

    TIB->ExceptionList = TIB->ExceptionList->Next;

    return iValue;
}

void DemoGlobalUnwinding()
{
    __try
    {
        TestDivisionByZero01SEH();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Caught the exception in DemoGlobalUnwinding()\n");
    }
}
```

3. In DemoGlobalUnwinding(), the presence of the __try/__except block signals to the compiler to emit a standard SEH Frame with __except_handler3().

4. Then, inside the __try block, TestDivisionByZero01SEH() is called and a separate SEH Frame is setup with a custom handler MyDivisionByZero01ExceptionRoutine().

5. When DemoGlobalUnwinding() runs, a Division by Zero Exception will be thrown. Since the highest SEH Frame in the TIB's ExceptionList is that created from TestDivisionByZero01SEH(), its handler MyDivisionByZero01ExceptionRoutine() will have first shot at handling the exception.

6. MyDivisionByZero01ExceptionRoutine() returns ExceptionContinueSearch meaning that it has declined the offer to handle the exception. The Windows OS moves to the next SEH frame which was setup in the DemoGlobalUnwinding() function and calls its filter function.

7. The filter function returns EXCEPTION_EXECUTE_HANDLER which indicates that the __except handler is to be invoked. But before this happens, MyDivisionByZero01ExceptionRoutine() **will be called a second time**.

8. This second call is part of something known as Global Unwinding which we will cover later in this article. Meantime, understand that when MyDivisionByZero01ExceptionRoutine() is called a second time, the pExceptionRecord->ExceptionFlags will contain EXCEPTION_UNWINDING.

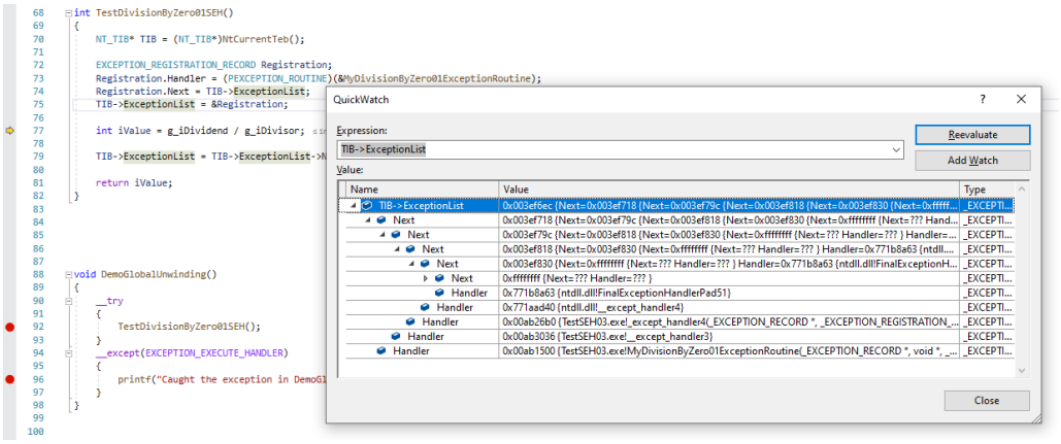
9. This flag signals to the exception handler that it is to perform unwinding. When a SEH Frame is told to Unwind, it only means one thing : do a local unwind. We will look into local unwinding next.

10. For now, observe the console output when DemoGlobalUnwinding() completes :

```
An excepton occured at address : [0x0001160B]. Exception Code : [0xC0000094]. Exception Flags : [0x00000000]
An excepton occured at address : [0x79F530D9]. Exception Code : [0xC0000027]. Exception Flags : [0x00000002]
 EXCEPTION_UNWINDING
Caught the exception in DemoGlobalUnwinding()
```

Notice that the unwinding is done before the __except handler in DemoGlobalUnwinding() is performed.

11. Now, to illustrate some of the points mentioned in Executing an __except Block about SEH Frames setup in a sequentially later function having its Handler positioned higher up the TIB's ExceptionList, observe the function TestDivisionByZero01SEH() at runtime :



12. In the above screenshot, we use the Visual Studio QuickWatch Window to observe the TIB's ExceptionList linked list. This is at line 77 where the EXCEPTION_REGISTRATION_RECORD defined in TestDivisionByZero01SEH() has just been inserted as the latest SEH Frame.

13. The QuickWatch window shows the following :

- MyDivisionByZero01ExceptionRoutine() is the latest SEH Exception Handler.
- Next below it is the __except_handler3() exception handler which was installed by DemoGlobalUnwinding() (the caller of MyDivisionByZero01ExceptionRoutine()).
- And below that it's _except_handler4() installed inside __scrt_common_main_seh() which indirectly calls DemoGlobalUnwinding().

14. James McNellis provided in the CPP Con a great summary pseudocode which I present below with some of my own comments :

```
void RtlUnwindPseudocode
(
    EXCEPTION_REGISTRATION_RECORD* TargetFrame,
    void* TargetIp,
    EXCEPTION_RECORD* ExceptionRecord,
    void* ReturnValue
)
{
    // Include the EXCEPTION_UNWINDING in ExceptionFlags
    ExceptionRecord->ExceptionFlags |= EXCEPTION_UNWINDING;

    // Obtain the TIB so that we can traverse the ExceptionList.
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();
    // Traverse the ExceptionList from the topmost SEH Frame
    // and move downwards until we reach the TargetFrame.
    while (TIB->ExceptionList != TargetFrame)
    {
        // For each inner SEH Frame, call its Exception Handler.
        // The Exception Handler is supposed to call LocalUnwind().
        TIB->ExceptionList->Handler(ExceptionRecord, TIB->ExceptionList);
        // Not only move downwards. Also change the ExceptionList
        // so that it no longer point to the current SEH Frame.
        TIB->ExceptionList = CurrentRecord->Next;
    }
}
```

Demo Local Unwinding

1. Local unwinding is done primarily to perform the relevant __finally blocks contained in a function. Observe the following function :

```
void DemoLocalUnwind()
{
    __try
    {
        printf("TryLevel == 0. ScopeEntry[0].\r\n");

        __try
        {
            printf("TryLevel == 1. ScopeEntry[1].\r\n");

            __try
            {
                printf("TryLevel == 2. ScopeEntry[2].\r\n");

                ExceptionCausingFunction();
            }
            __finally
            {
                printf("__finally for TryLevel == 2. ScopeEntry[2].\r\n");
            }
        }
        __finally
        {
            printf("__finally for TryLevel == 1. ScopeEntry[1].\r\n");
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");
    }
}
```

2. In DemoLocalUnwind(), an exception will be caused by the ExceptionCausingFunction() which will be called when the TryLevel is 2. However, the exception is handled when the TryLevel is 0. Hence in between TryLevels 2 and 0, there *may be* SCOPETABLE_ENTRY items with __finally blocks that will have to be called. Note the italicized *may be*. This is because instead of __finally blocks, TryLevels 2 and 1 may contain __except blocks that decline to take action.

3. Before the exception handler code at TryLevel 0 is executed, local unwinding is to be performed. This is done by iterating through the SCOPETABLE_ENTRY array of the SEH Frame of the function starting from **the current TryLevel when the exception occurred** and then working down until a **stop point**. The stop point here being TryLevel 0 because it is not part of the unwinding process.

4. In the DemoLocalUnwind() function above, when the exception occurred, the TryLevel is 2. Hence the following __finally block is executed :

```
            __finally
            {
                printf("__finally for TryLevel == 2. ScopeEntry[2].\r\n");
            }
```

The TryLevel is then set to the EnclosingLevel of SCOPETABLE_ENTRY[2] which is 1 and the following __finally block will be executed :

```
        __finally
        {
            printf("__finally for TryLevel == 1. ScopeEntry[1].\r\n");
        }
```

The TryLevel is then set to the EnclosingLevel of SCOPETABLE_ENTRY[1] which is 0 (the stop point). The local unwinding process thus stops.

4. I present below the pseudocode of the local unwind function provided by James McNellis with some modifications and additional comments from me :

```
void _local_unwind_pseudcode
(
    EH4_EXCEPTION_REGISTRATION_RECORD* RN,
    int Stop
)
{
    // We assume that RN's TryLevel is currently at the
    // TryLevel at which the exception occurred.
    while (RN->TryLevel != Stop)
    {
        // Access the SCOPETABLE_ENTRY of the TryLevel.
        SCOPETABLE_ENTRY* CurrentEntry = &RN->ScopeTable[RN->TryLevel];
        // CurrentEntry->Filter == NULL means this is a __finally block
        // which is what we want.
        if (CurrentEntry->Filter == nullptr)
        {
            // Call the __finally block code.
            CurrentEntry->Handler();
        }
        // Move onto the next EnclosingLevel.
        RN->TryLevel = CurrentEntry->EnclosingLevel;
    }
}
```

Source Codes

1. The source codes for this part can be found in GitHub.

2. It is compiled in Visual Studio Community 2019.

3. Be sure to set the compilation target to x86 and not x64.

4. Note that I have deliberately set some settings in order to simplify our study of the assembly language code generated for the C++ source codes.

5. These settings include :

- No optimization.
- C/C++ | Code Generation | Basic Runtime Checks : Default.
- C/C++ | Code Generation | Security Checks : Disable Security Checks (/GS-)

6. Optimization has been turned off to enable the Visual C++ compiler to produce template assembly codes for our C++ functions and SEH constructs. These non-optimized code will be easier to understand and follow.

7. The use of Default Basic Runtime Checking code will avoid the inclusion of additional runtime code which will clutter up our assembly language code.

8. By Disabling of Security Checks, the compiler will use the __except_handler3() exception handler which we will use in later parts to study show exception handling code works internally. If Security Checks is enabled, the compiler will emit code to use __except_handler4 instead which is a more advanced exception handler that includes security features. This will be beyond the scope of this series of articles.

Summary

1. In this part 3, we have done some rigorous study of the __except_handler3() function.

2. We have studied Global and Local Unwind and also made a limited attempt to try to understand how the flow of control is passed to an __except block.

3. Thus far, we have created custom SEH Frames by directly modifying the TIB's ExceptionList. However, we have not provided any equivalent custom exception filter functions nor custom finally functions.

4. In the next part, I shall use all that we have learned so far and attempt to create custom exception filters, except and finally functions.

About Lim Bio Liong

I've been in software development for nearly 20 years specializing in C , COM and C#. It's truly an exicting time we live in, with so much resources at our disposal to gain and share knowledge. I hope my blog will serve a small part in this global knowledge sharing network.

For many years now I've been deeply involved with C development work. However since circa 2010, my current work has required me to use more and more on C# with a particular focus on COM interop. I've also written several articles for CodeProject. However, in recent years I've concentrated my time more on helping others in the MSDN forums. Please feel free to leave a comment whenever you have any constructive criticism over any of my blog posts.
View all posts by Lim Bio Liong »