Inside Exception Handling

By ITh4cker

In this topic,I will introduce the detial of exception handling on Windows to you.Okay,Let's start and go>>

As we all know that SEH(Structured Exception Handling) is an build-in exception handling mechasim provided by Widdows.So it can be used and extended by any programming language and compiler in theory,for which they should accomplish mainly two task:

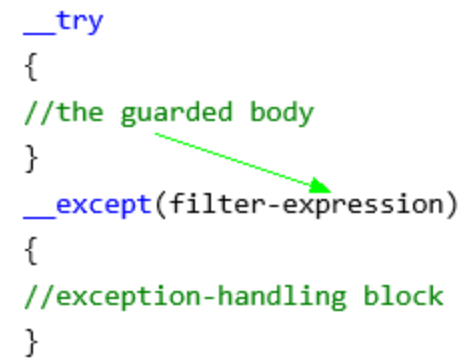
1. define the necessary key words to represent the logic of exception handling,which will be used by programmer as an interface.For example,VC compiler define the __try, __finally,__except key words.
2. Implement the compling of the key words and link up the exception handling code with SEH.In other words,any other extended exception handling is all based on the build-in SEH on windows.

In VC++,the normal exception code is like following:

```
__try
{
//the guarded body
}
__except(filter-expression)
{
//exception-handling block
}
```

When the code in __try{}(guarded body) occur an exception,the system will execute the code in filter-expression, which will decided what to do next.

The execution line like as following:



It seems like a leap from the guaded body to filter-expression in the execution line.It's uneasy to perform the leap for it

should know the exact loaction of filter-expressions and keep a balance on stack.To implement the leap,the complier will first analyze the structure of the exception handling code and *package and mark* the all parts in the structure and then register the exception handling function in case of being called when exception occurred.
Okay,Okay..Before introducing the detail of exception handling,Let's have a look at the process of exception dispatching(*Here you only need to know the user-mode,for kernel-mode I will have a deep introduction in the subsequent sofeware debugging or windows kernel learning series*)

When there is exception happened,CPU wwill search the IDT table for the exception handler(like KiTrapXX),which will finally call the kernel function **KiDispatchException** to dispatch exception
KiDispatchException is the hub for dispatching various exceptions.Its prototype is:
VOID KiDispatchException (IN PEXCEPTION_RECORD ExceptionRecord, IN PKEXCEPTION_FRAME ExceptionFrame, IN PKTRAP_FRAME TrapFrame, IN KPROCESSOR_MODE PreviousMode, IN BOOLEAN FirstChance)
The parameter PreviousMode is an constant of enumation:
Typedef enum _MODE{KernelMode, UserMode, MaximumMode} MODE;
When PreviousMode is 0, it represent the KernelMode and 1 for UserMode.
The parameter FirstChance represent whether it's the first time dispatching exception.System will dispatch at most twice for an exception

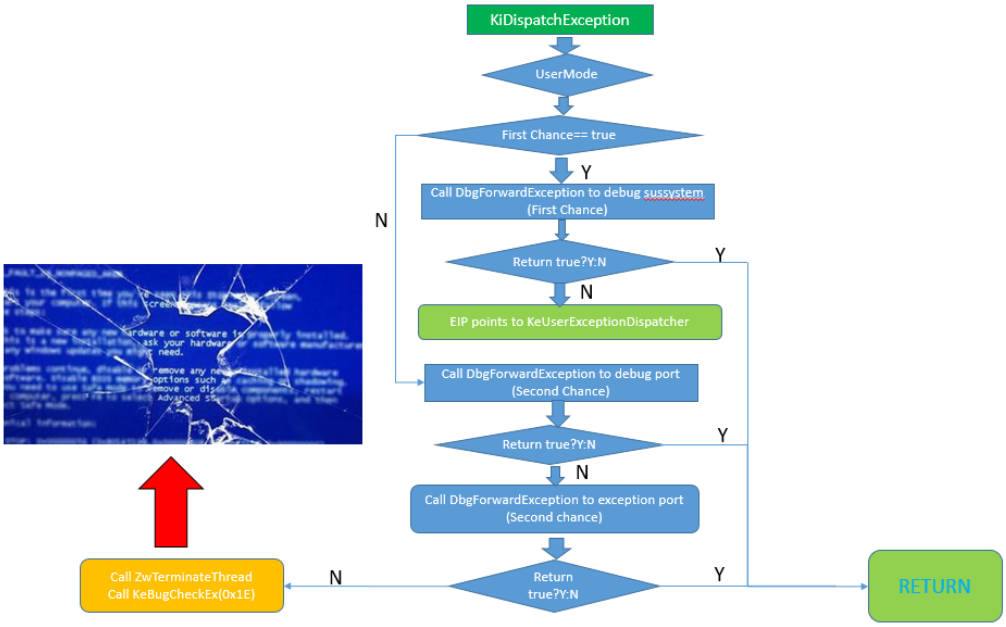


Figure 1.The process of KiDispatchException in user-mode

```
//pseudocode for KiDispatchException
VOID KiDispatchException(PEXCEPTION_RECORD Er, ULONG Reserved,
                    PKTRAP_FRAME Tf, MODE PreviousMode,
                    BOOLEAN SearchFrames)
{
   PCR->KeExceptionDispatchCount++;
   CONTEXT Context
      = {CONTEXT_FULL | (PreviousMode == UserMode ?
CONTEXT_DEBUG : 0)};
   KeContextFromKframes(Tf, Reserved, &Context);
   if (Er->ExceptionCode == STATUS_BREAKPOINT) Context.Eip–;
   do {
      if (PreviousMode == KernelMode) {
         if (SearchFrames) {
            if (KiDebugRoutine &&
               KiDebugRoutine(Tf, Reserved, Er, &Context,
                  PreviousMode, FirstChance) != 0) break;
               if (RtlDispatchException(Er, &Context) == 1)
break;
         }
         if (KiDebugRoutine &&
            KiDebugRoutine(Tf, Reserved, Er, &Context,
            PreviousMode, LastChance) != 0) break;
      }
      else {
         if (SearchFrames) {
            if (PsGetCurrentProcess()->DebugPort == 0
               || KdIsThisAKdTrap(Tf, &Context)) {
                  if (KiDebugRoutine &&
                     KiDebugRoutine(Tf, Reserved, Er,
&Context,
                        PreviousMode, FirstChance) != 0) break;
            }
            if (DbgkForwardException(Tf, DebugEvent,
               FirstChance) != 0) return;
            if (valid_user_mode_stack_with_enough_space) {
               // copy EXCEPTION_RECORD and CONTEXT to user
mode stack;
               // push addresses of EXCEPTION_RECORD and
CONTEXT
               // on user mode stack;
               Tf->Eip = KeUserExceptionDispatcher;
               return;
            }
```

```
            }
            if (DbgkForwardException(Tf, DebugEvent,
               LastChance) != 0) return;
            if (DbgkForwardException(Tf, ExceptionEvent,
               LastChance) != 0) return;
            ZwTerminateThread(NtCurrentThread(),
Er->ExceptionCode);
         }
         KeBugCheckEx(KMODE_EXCEPTION_NOT_HANDLED,
Er->ExceptionCode,
            Er->ExceptionAddress, Er->ExceptionInformation[0],
            Er->ExceptionInformation[1]);
   } while (false);
   KeContextToKframes(Tf, Reserved, &Context,
      Context.ContextFlags, PreviousMode);
}
```

As showed in figure 1, there are twice exception dispatching.For the first chance,KiDispatchException will dispatch the exception to the user-mode debugger by calling the kernel instance DbgFowardException,which has three parameters: the first is exception record,the second is a boolean to specify the debug port or the exception port,the third specify whether the first chance,for the first chance, It's called like:
DbgkForwardExcepton(ExceptionRecord,TRUE,FALSE); means sending to debug port and DbgkForwardExcepton will check whether the DebugPort of current process is NULL,if not NULL then calling DbgkpSendApiMessage tto send exception to debugging-subsystem ,which will send exception to debugger again.If DbgkpSendApiMessage return STATUS_SUCCESS and debugger handlered the exception(ReturnedStatus == DBG_CONTINUE),then DbgForwardException will return true,the dispatching get over. If the debugger didn't handle the exception(ReturenedStatus ==DBG_EXCEPTION_NOT_HANDLER), DbgkpSendApiMessage will return false.What KiDispatchException next to do is to search the exception handling block to handle it,because the exception occurred in user-mode code and exception handling block should be in user-mode function,so KiDispatchException will go back to user-mode to execute,in which EIP will points to the KiUserExceptionDispatcher function in NTDLL.dll.The KiUserExceptionDispatcher will call the RtlDispatchException to search the exception handling block,and if it return true,representing the exception has been handled , KiUserExceptionDispatcher will call ZwContinue(system service) to continue to execute at where exception occurred.If

RtlDispatchException return false,representing that no exception handling blcok can handle it, and the current process being debugged, KiUserExceptionDispatcher will set the parameter FirstChance to false by calling ZwRaiseException then starting the second dispatching(ZwRaiseException pass the exception to KiDispatchException by calling NtRaiseException)For the second dispathcing, KiDispatchException will still dispatch the exception to the DebugPort of the process first,if it return false ,then dispatch to ExceptionPort(now the user-mode debugger has no any chance to handler the exception ☹),if DbgFowardException return false, representing the exception is unhandled,then terminate the current thread and call the KeBugCheckEx to make a BSOD(Blue Sreen Of Death)☺as showed in figure 1

(*The process of KiUserExceptionDispatcher will be introduced later*☺)

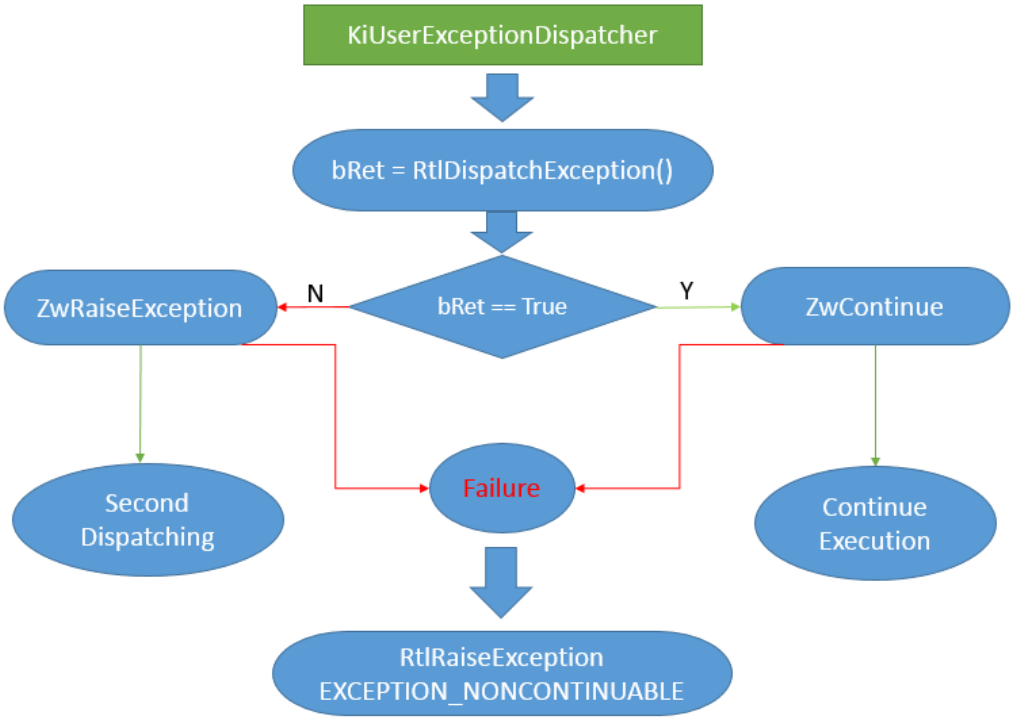


Figure 2 The process of KiUserExceptionDispatcher

As is showed in figure 2,KiUserExceptionDispatcher will call RtlDispatchException to dispatch exception,if it return true,then it call NtContinue,which will makes the program executed from where exception occurred,if false,then it call NtRaiseException to start the second dispatching,if both NtContinue and NtRaiseException return false,it will execute the left code in KiUserExceptionDispatcher,otherwise it will never

return to KiUserExceptionDispatcher.
The pseudocode for KiUserExceptionDispatcher as following:

```
KiUserExceptionDispatcher( PEXCEPTION_RECORD pExcptRec,
CONTEXT * pContext )
{
    DWORD retValue;
    // Note: If the exception is handled,
RtlDispatchException() never returns
    if ( RtlDispatchException( pExceptRec, pContext ) )
        retValue = NtContinue( pContext, 0 );
    else
        retValue = NtRaiseException( pExceptRec, pContext, 0 );
    EXCEPTION_RECORD excptRec2;

    excptRec2.ExceptionCode = retValue;
    excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    excptRec2.ExceptionRecord = pExcptRec;
    excptRec2.NumberParameters = 0;

    RtlRaiseException( &excptRec2 );
}
```

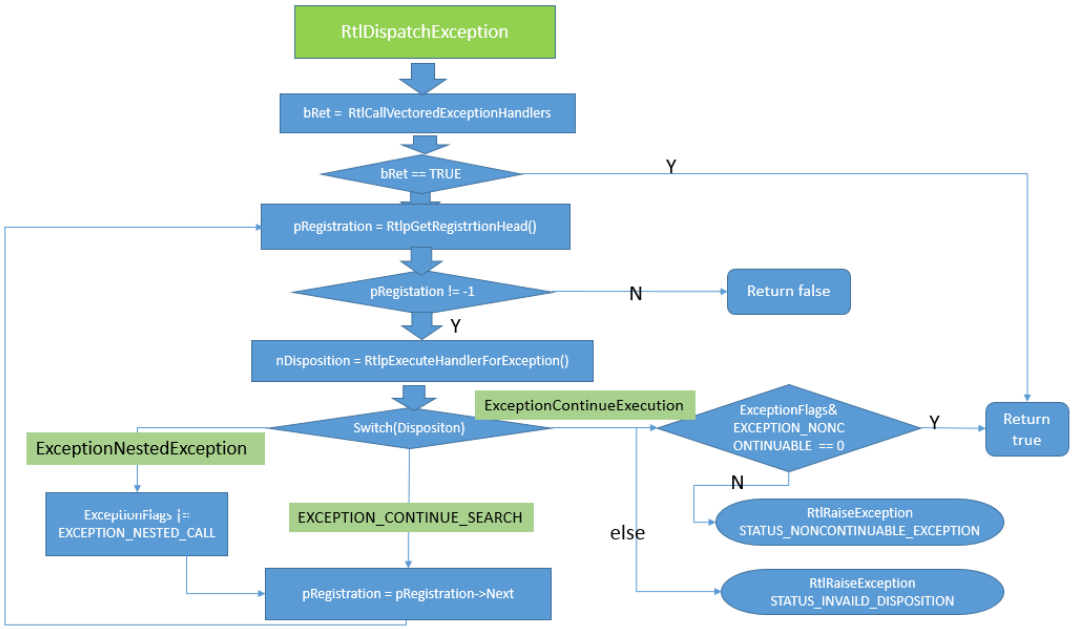


Figure 3. The process of RtlDispatchException

The pseudocode for RtlDispatchException :

```
   int RtlDispatchException( PEXCEPTION_RECORD pExcptRec,
CONTEXT * pContext )
   {
       DWORD   stackUserBase;
```

```
        DWORD   stackUserTop;
        PEXCEPTION_REGISTRATION pRegistrationFrame;
        DWORD hLog;

        // Get stack boundaries from FS:[4] and FS:[8]
        RtlpGetStackLimits( &stackUserBase, &stackUserTop );
        pRegistrationFrame = RtlpGetRegistrationHead();
        while ( -1 != pRegistrationFrame )
        {
            PVOID justPastRegistrationFrame =
&pRegistrationFrame + 8;
            if ( stackUserBase > justPastRegistrationFrame )
            {
                pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
                return DISPOSITION_DISMISS; // 0
            }
            if ( stackUsertop < justPastRegistrationFrame )
            {
                pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
                return DISPOSITION_DISMISS; // 0
            }
            if ( pRegistrationFrame & 3 )   // Make sure stack
is DWORD aligned
            {
                pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
                return DISPOSITION_DISMISS; // 0
            }
            if ( someProcessFlag )
            {
                // Doesn't seem to do a whole heck of a lot.
                hLog = RtlpLogExceptionHandler( pExcptRec,
pContext, 0, pRegistrationFrame, 0x10 );
            }
            DWORD retValue, dispatcherContext;
            retValue= RtlpExecuteHandlerForException(pExcptRec,
pRegistrationFrame,pContext, &dispatcherContext,
pRegistrationFrame->handler );

            // Doesn't seem to do a whole heck of a lot.
            if ( someProcessFlag )
                RtlpLogLastExceptionDisposition( hLog,
retValue );

            if ( 0 == pRegistrationFrame )
```

```
            {
                pExcptRec->ExceptionFlags &= ~EH_NESTED_CALL;
// Turn off flag
            }
            EXCEPTION_RECORD excptRec2;
            DWORD yetAnotherValue = 0;
            if ( DISPOSITION_DISMISS == retValue )
            {
                if ( pExcptRec->ExceptionFlags &
EH_NONCONTINUABLE )
                {
                    excptRec2.ExceptionRecord = pExcptRec;
                    excptRec2.ExceptionNumber =
STATUS_NONCONTINUABLE_EXCEPTION;
                    excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
                    excptRec2.NumberParameters = 0
                    RtlRaiseException( &excptRec2 );
                }
                else
                    return DISPOSITION_CONTINUE_SEARCH;
            }
            else if ( DISPOSITION_CONTINUE_SEARCH == retValue )
            {
            }
            else if ( DISPOSITION_NESTED_EXCEPTION == retValue )
            {
                pExcptRec->ExceptionFlags |= EH_EXIT_UNWIND;
                if ( dispatcherContext > yetAnotherValue )
                    yetAnotherValue = dispatcherContext;
            }
            else    // DISPOSITION_COLLIDED_UNWIND
            {
                excptRec2.ExceptionRecord = pExcptRec;
                excptRec2.ExceptionNumber =
STATUS_INVALID_DISPOSITION;
                excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
                excptRec2.NumberParameters = 0
                RtlRaiseException( &excptRec2 );
            }

            pRegistrationFrame = pRegistrationFrame->prev;  //
Go to previous frame
        }
```

```
        return DISPOSITION_DISMISS;
    }
```

As is showed in figure 3,the RtlDispatchException will call RtlCallVectoredExceptionHandlers(VEH,here we don't discuss VEH),if it return true,after several function related to VEH calling, it will return true,if RtlCallVectoredExceptionHandlers return false,then next call RtlpGetRegistrationHead to retrive the start head of SEH chain,starting the traversing of SEH chain.if the pRegistrationFrame != 0xFFFFFFFF(the end of SEH chain),call RtlpExecuteHandlerForException to execute the SEH handler: RtlpExecuteHandlerForException -> ExecuteHandler -> ExecuteHandler2 -> call ecx(SEH handler / _except_handler4) -> _except_handler4_common
the return value of RtlpExecuteHandlerForException is an constant of enum EXCEPTION_DISPOSITION:

```
typedef enum _EXCEPTION_DISPOSITION {
ExceptionContinueExecution, //0 retore to execute the code that
                            //trigger exception
ExceptionContinueSearch,    //1  continue to search the next EH
            structure
ExceptionNestedException,   //2  the nested exception occured
ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION
```

Inside _except_handler4 ,you can see the detailed process of compiler-enhanced exception handling(including global unwinding and local unwinding of EH etc..) The routine is complex but not difficult, if you debug and tracing it all the way, you can see the nature landscape☺ *For the time being,I have no plan to write the detail about it,maybe I will write it in my sussequent Software Debugging series of topics.*

Reference:

1.*Book:*<Software Debugging> Booked by YinKui,Zhang China.
2.*Paper:*A Crash Course on the Depths of Win32™ Structured Exception Handling
https://www.microsoft.com/msj/0197/Exception/Exception.aspx
3.Google ☺

ITh4cker 2015/12/09 BeiJing China