Exceptions can occur in both user mode and kernel mode code and can be generated by either the processor (such as "general protection," "divide by zero," or debug exceptions) or by calling ZwRaiseException. Almost all exceptions eventually result in the kernel mode routine KiDispatchException being called. This routine is at the heart of the exception-handling and debugging support provided by the system, and its pseudocode appears in Example D.1.

**Example D.1: Pseudocode for KiDispatchException**

```c
enum CHANCE {
    FirstChance,
    LastChance
};

enum EVENT {
    ExceptionEvent,
    DebugEvent
};

VOID KiDispatchException(PEXCEPTION_RECORD Er, ULONG Reserved,
                        PKEVENT_ROUTINE_PTR 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;
```
### Exceptions and Debugging: Example D.1

```c
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);
}
```

*KiDebugRoutine* is a pointer to a function, and normally takes one of two values, depending on whether the system was rebooted with kernel mode debugging enabled (for example, /DEBUG was specified in boot.ini).

There are two main paths through *KiDispatchException* that are selected according to the previous execution mode.
Exceptions and Debugging: Example 20.2

If the previous mode was kernel, the following steps are taken:

- If frame-based exception-handling is allowed (\texttt{SearchFrames == TRUE}), the kernel debugger is given a first chance to handle the exception.
- If the kernel debugger does not handle the exception, then \texttt{RtlDispatchException} is invoked to search for and invoke a frame-based exception handler.
- If \texttt{RtlDispatchException} does not find a handler prepared to handle the exception or if \texttt{SearchFrames} is \texttt{FALSE}, the kernel debugger is given a last chance to handle the exception.
- Finally, if the exception has still not been handled, \texttt{KeBugCheckEx} is invoked to shut down the system with the bugcheck code \texttt{KMODE EXCEPTION NOT_HANDLED}.

If the previous mode was user, the following steps are taken:

- If frame-based exception-handling is allowed (\texttt{SearchFrames == TRUE}) and if the process is not being debugged by a user mode debugger (\texttt{DebugPort == 0}), the kernel debugger is given a first chance to handle the exception; otherwise, a description of the exception is forwarded to the user mode debugger via the LPC mechanism.
- If the exception is not handled by a debugger and the user mode stack appears to be still valid, the user mode context is adjusted so that upon return to user mode, the function \texttt{KiUserExceptionDispatcher} will be invoked.
- After returning to user mode, \texttt{KiUserExceptionDispatcher} invokes \texttt{RtlDispatchException} to search for a frame-based exception handler.
- If \texttt{RtlDispatchException} does not find a handler prepared to handle the exception, the exception is re-signaled, specifying \texttt{SearchFrames} as \texttt{FALSE}.
- \texttt{KiDispatchException} is entered again and, because \texttt{SearchFrames} is \texttt{FALSE}, the next step is to give a user mode debugger a last chance to handle the exception.
- If the debugger (if any) still does not handle the exception, a description of the exception is forwarded to the exception port (if any) of the process.
- The recipient (if any) of the message to the exception port can still handle the exception, but if it does not, \texttt{ZwTerminateThread} is called to terminate the current thread.
- If \texttt{ZwTerminateThread} fails for any reason, \texttt{KeBugCheckEx} is invoked to shut down the system with the bugcheck code \texttt{KMODE EXCEPTION NOT_HANDLED}.

Example D.2: Pseudocode for \texttt{KiUserExceptionDispatcher}

```c
VOID KiUserExceptionDispatcher(PEXCEPTION_RECORD ExceptionRecord, PCONTEXT Context)
{
    NTSTATUS rv = RtlDispatchException(ExceptionRecord, Context) == 1
        ? ZwContinue(Context, FALSE)
        : ZwRaiseException(ExceptionRecord, Context, FALSE);

    EXCEPTION_RECORD NestedExceptionRecord
```
Exceptions and Debugging: Example 20.2

```c
    = {rv, EXCEPTION_NONCONTINUABLE, ExceptionRecord};
    RtlRaiseException(&NestedExceptionRecord);
}
```

Example D.2 shows how KiUserExceptionDispatcher uses the two system services, ZwContinue and ZwRaiseException. As mentioned previously, KiUserExceptionDispatcher first calls RtlDispatchException to find and invoke a frame-based exception handler. An exception handler can modify the context structure (which it accesses by calling GetExceptionInformation). Therefore, if RtlDispatchException finds a handler, upon return from the handler, ZwContinue is invoked to modify the execution context of the current thread to make it the one that is specified by the handler. If a handler is not found, ZwRaiseException is called to re-signal the exception. If either ZwContinue or ZwRaiseException return, a nested, noncontinuable exception is raised.

All threads created by Win32 functions have a top-level frame-based exception handler; the behavior of this handler can be influenced by calling the Win32 function SetUnhandledExceptionFilter. This functionality allows a last-chance handler to be defined, which handles the unhandled exceptions of all threads in a process. There is no mechanism defined to provide a first-chance handler (which would have the chance to handle the exceptions of all threads before searching the thread’s stack for frame-based handlers), but by knowing how exception dispatching works, it is possible to provide this functionality by patching the binary code of KiUserExceptionDispatcher. (There are resource kit–like utilities that actually do this).

The Kernel Debugger

The principal link between the kernel debugger and the kernel itself are the call-outs to the kernel debugger (KiDebugRoutine) embedded in the kernel routine KiDispatchException. The only other essential link is the check performed by KeUpdateSystemTime for input from a remote debugger (for example, a Ctrl-C break-in); if input is detected, KeUpdateSystemTime generates an exception by calling DbgBreakPointWithStatus, which eventually results in the KiDispatchException kernel debugger call-outs being invoked.

Other kernel components that wish to inform the kernel debugger of some event call DebugService, which ultimately conveys the information to the kernel debugger by raising an exception.

Example D.3: Pseudocode for DebugService

```c
typedef enum _DEBUG_SERVICE_CODE {
    DebugPrint = 1,
    DebugPrompt,
    DebugLoadImageSymbols,
    DebugUnloadImageSymbols
} DEBUG_SERVICE_CODE;

NTSTATUS DebugService(DEBUG_SERVICE_CODE Opcode, PSTRING String, PVOID Data)
```
Exceptions and Debugging: User Mode Debuggers

```c
{ NTSTATUS rv;

    __asm {
        mov   eax, Opcode
        mov   ecx, String
        mov   edx, Data
        int   0x2D
        int   0x03
        mov   rv, eax
    }

    return rv;
}
```

As was mentioned in Appendix B, "Intel Platform-Specific Entry Points to Kernel Mode," the instruction "int 0x2D" invokes KiDebugService, which saves the values of selected registers in an EXCEPTION_RECORD structure and then raises a STATUS_BREAKPOINT exception. When KiDispatchException is invoked to handle the exception and KiDebugRoutine is called, the kernel debugger recognizes the exception as coming from KiDebugService (because the EXCEPTION_RECORD member ExceptionParameters[0] is non-zero) and responds accordingly.

Two kernel routines that inform the kernel debugger of events using this mechanism are MmLoadSystemImage and MmUnloadSystemImage. (This is how the kernel debugger learns of the loading and unloading of device drivers).

As was mentioned earlier, KiDebugService is a pointer to a function, and it normally points at one of two routines. If kernel debugging is enabled (by specifying /DEBUG in boot.ini, for example), KiDebugService points to KdpTrap; otherwise, it points to KdpStub.

KdpStub checks whether the exception is a STATUS_BREAKPOINT with a recognized DEBUG_SERVICE_CODE that can be ignored (all except DebugPrompt can be ignored) and, if so, returns one to KiDispatchException, indicating that the exception has been handled. KdpStub also does what is necessary to support ZwSystemDebugControl.

KdpTrap implements the full kernel debugger raising functionality and can, if necessary, freeze the operation of the system and interact with a remote debugger via the serial line.

User Mode Debuggers

At five points in the kernel (as described below), a check is made as to whether the current process has a debug port; if it does, then an LPC message is constructed describing the event that has just occurred. All threads (except the current) are frozen and the message is sent to the debug port. When a reply is received, the frozen threads are thawed.

The five points in the kernel at which checks are made are:

- Thread creation routine
- Thread termination routine
- Executable image-mapping routine
- Executable image-unmapping routine
- Exception dispatching routine (KiDispatchException, described earlier)
The message sent to the debug port is a DEBUG_MESSAGE structure, which bears a resemblance to the Win32 DEBUG_EVENT structure.

```c
typedef struct _DEBUG_MESSAGE {
    PORT_MESSAGE PortMessage;
    ULONG EventCode;
    ULONG Status;
    union {
        struct {
            EXCEPTION_RECORD ExceptionRecord;
            ULONG FirstChance;
        } Exception;
        struct {
            ULONG Reserved;
            PVOID StartAddress;
        } CreateThread;
        struct {
            HANDLE FileHandle;
            PVOID Base;
            ULONG PointerToSymbolTable;
            ULONG NumberOfSymbols;
            HANDLE Reserved2;
            PVOID EntryPoint;
        } CreateProcess;
        struct {
            ULONG ExitCode;
        } ExitThread;
        struct {
            HANDLE FileHandle;
            PVOID Base;
            ULONG PointerToSymbolTable;
            ULONG NumberOfSymbols;
        } LoadDll;
        struct {
            PVOID Base;
        } UnloadDll;
    } u;
} DEBUG_MESSAGE, *PDEBUG_MESSAGE;
```

Some of the messages include handles that are valid in the context of the debuggee. Example 20.4 demonstrates how to implement debugger-type functionality by directly receiving and replying to these messages.

**Debug Message Routing**

The debug port of Win32 processes being debugged is normally the general function port for the Win32 subsystem process (the port named "\Windows\ApiPort") rather than a port created by the debugger itself.
There are routines in ntdll.dll intended for use by environment subsystems to perform the bulk of debug message processing. By default, these routines repackage the message slightly and forward it to the port named “\DbgSsApiPort,” but they allow the subsystem to customize their behavior by registering callback functions. The Win32 subsystem process (csrss.exe) does not add any significant functionality to the forwarding process.

The process that listens to the port named “\DbgSsApiPort” is the Session Manager (smss.exe), which acts as a switch and monitor between applications and debuggers. Debuggers register with the Session Manager by connecting to the port named “\DbgUiApiPort.”

The Session Manager receives messages from the port named “\DbgSsApiPort,” repackages their contents again (duplicating any handles into the debugger) and forwards the message to the debugger.

When the debugger replies to the message specifying the “continue status,” the Session Manager forwards the reply to Win32 subsystem process, which forwards it in turn to the debuggee.

**Value Added by the Routing Process**

When a variant of Example D.4 that uses the Win32 debugging API (rather than the native API) is run, a consequence of the routing of the debug messages through various processes is that the CPU load is roughly evenly divided between the debuggee, the Session Manager, the Win32 subsystem, and the debugger. So it is worthwhile considering the value that each process adds.

The Win32 subsystem process does not add any significant value when debugging a newly created process, but it does provide important functionality in support of the Win32 DebugActiveProcess function: It fabricates process and thread creation debug messages for the existing threads and image-mapping events for the loaded DLLs of the debuggee.

The Session Manager ensures that the debuggee is terminated if the debugger terminates. A debuggee waiting for a debugger to reply to a debug message cannot be terminated, so if the debugger were to terminate and the debuggee were allowed to continue running, the next debug event to occur (as a result of thread creation, DLL loading, or exception) would cause the debuggee to enter a state from which it could not be continued or terminated.

The Session Manager also signals the availability of messages to the debugger by signaling a semaphore; this allows a debugger to timeout a wait for a debug event. This was necessary in Windows NT 4.0, because, as conventional ports are not waitable objects, it is not possible to use ZwWaitForSingleObject to wait on them. The waitable ports introduced with Windows 2000 or the new ZwReplyWaitReceivePortEx system service could also be used to tackle this problem, but in practice the Windows NT 4.0 architecture has been retained.
Exceptions and Debugging: OutputDebugString

**OutputDebugString**

OutputDebugString communicates its string to the debugger by raising an exception with a particular code (0x40010006); if not recognized and handled by a debugger, a frame-based exception handler is invoked, which makes the string available to debug string monitors (such as dbmon.exe) by copying it to a file mapping and signaling an event.

**Tracing Calls to Routines Exported by DLLs**

Example D.4 demonstrates the direct manipulation of the debug port of a process. The example traces calls to the exported routines of all the DLLs that are loaded in a process and runs in about 60 percent of the time required by a variant using the Win32 debugging API. The level of tracing is more detailed than that produced by utilities that patch the image export directories of the loaded DLLs, but the tracing consumes substantially more CPU time. An application being traced runs at about one twentieth of its normal speed.

**Example D.4: A Trace Utility**

```c
#include "ntdll.h"
#include <imagehlp.h>
#include <stdlib.h>
#include <stdio.h>
#include <vector>
#include <map>
#define elements(s) (sizeof(s) / sizeof *(s))
namespace NT {
    extern "C" {
        typedef struct _DEBUG_MESSAGE {
            PORT_MESSAGE PortMessage;
            ULONG EventCode;
            ULONG Status;
            union {
                struct {
                    EXCEPTION_RECORD ExceptionRecord;
                    ULONG FirstChance;
                } Exception;
                struct {
                    ULONG Reserved;
                    PVOID StartAddress;
                } CreateThread;
                struct {
                    ULONG Reserved;
                    HANDLE FileHandle;
                    PVOID Base;
                    ULONG PointerToSymbolTable;
                    ULONG NumberOfSymbols;
                    ULONG Reserved2;
                    PVOID EntryPoint;
                } CreateProcess;
                struct {
                    ULONG ExitCode;
                } CreateThread;
            }
        } DEBUG_MESSAGE;
```
Exceptions and Debugging: User Mode Debuggers

```c
} ExitThread;
struct {
    ULONG ExitCode;
} ExitProcess;
struct {
    HANDLE FileHandle;
    PVOID Base;
    ULONG PointerToSymbolTable;
    ULONG NumberOfSymbols;
} LoadDll;
struct {
    PVOID Base;
} UnloadDll;
} u;
} DEBUG_MESSAGE, *PDEBUG_MESSAGE;

typedef struct _DEBUG_STATUS {
    ULONG B0 : 1;
    ULONG B1 : 1;
    ULONG B2 : 1;
    ULONG B3 : 1;
    ULONG    : 9;
    ULONG BD : 1;
    ULONG BS : 1;
    ULONG BT : 1;
    ULONG    : 16;
} DEBUG_STATUS, *PDEBUG_STATUS;

typedef struct _DEBUG_CONTROL {
    ULONG L0 : 1;
    ULONG G0 : 1;
    ULONG L1 : 1;
    ULONG G1 : 1;
    ULONG L2 : 1;
    ULONG G2 : 1;
    ULONG L3 : 1;
    ULONG G3 : 1;
    ULONG LE : 1;
    ULONG GE : 1;
    ULONG    : 3;
    ULONG GD : 1;
    ULONG    : 2;
    ULONG RWE0 : 2;
    ULONG LEN0 : 2;
    ULONG RWE1 : 2;
    ULONG LEN1 : 2;
    ULONG RWE2 : 2;
    ULONG LEN2 : 2;
    ULONG RWE3 : 2;
    ULONG LEN3 : 2;
} DEBUG_CONTROL, *PDEBUG_CONTROL;

struct Error {
    ULONG line;
    ULONG code;
    Error(ULONG line, ULONG code) : line(line), code(code) {}
};

struct enter {
    PCSTR name;
    BYTE  opcode;
};
```
Exceptions and Debugging: User Mode Debuggers

```cpp
ULONG argc;
enter() : name(0), opcode(0), argc(0) {}
enter(PCSTR n, BYTE o = 0, ULONG a = 3) : name(n), opcode(o), argc(a) {};

struct leave {
    PVOID eip;
    ULONG esp;
    leave() : eip(0), esp(0) {}
    leave(PVOID ip, ULONG sp) : eip(ip), esp(sp) {}
};

#pragma warning(disable:4786)
typedef std::map<ULONG, std::vector<leave>, std::less<ULONG> > leaves_t;
typedef std::map<PVOID, enter, std::less<PVOID> > enters_t;
typedef std::map<ULONG, PVOID, std::less<ULONG> > steps_t;

enters_t enters;
leaves_t leaves;
steps_t steps;
std::map<ULONG, HANDLE, std::less<ULONG> > threads;
HANDLE hProcess;
ULONG StartTime;
BOOL Discard;
const int EXECUTE = PAGE_EXECUTE | PAGE_EXECUTE_READ
| PAGE_EXECUTE_READWRITE | PAGE_EXECUTE_WRITECOPY;

BYTE InsertBreakPoint(PVOID addr)
{
    MEMORY_BASIC_INFORMATION mbi;
    ULONG rv;
    BYTE op, bp = 0xcc;
    rv = VirtualQueryEx(hProcess, addr, &mbi, sizeof mbi);
    if (rv != sizeof mbi) return bp;
    if ((mbi.Protect & EXECUTE) == 0) return bp;
    rv = ReadProcessMemory(hProcess, addr, &op, sizeof op, 0);
    if (rv == FALSE) return bp;
    rv = WriteProcessMemory(hProcess, addr, &bp, sizeof bp, 0);
    if (rv == FALSE) return bp;
    return op;
}

VOID ReinsertBreakPoint(PVOID addr)
{
    BYTE bp = 0xcc;
    BOOL rv = WriteProcessMemory(hProcess, addr, &bp, sizeof bp, 0);
    if (rv != TRUE) throw Error(__LINE__, GetLastError());
}

VOID StepBreakPoint(PCONTEXT context, ULONG tid, PVOID addr, BYTE opcode)
{
    BOOL rv = WriteProcessMemory(hProcess, addr, &op, sizeof opcode, 0);
    if (rv != TRUE) throw Error(__LINE__, GetLastError());
    steps[tid] = addr;
```
Exceptions and Debugging: User Mode Debuggers

```c
context->EFlags |= 0x100;
context->Eip -= 1;
}

ULONG ReturnBreak(PCONTEXT context, PVOID addr, ULONG tid)
{
    std::vector<leave>& stack = leaves[tid];

    while (!stack.empty() && stack.back().esp < context->Esp) {
        stack.pop_back();
        printf("#");
    }

    if (addr == 0) return 0;
    stack.push_back(leave(addr, context->Esp));

    PDEBUG_CONTROL dr7 = PDEBUG_CONTROL(&context->Dr7);
    PDEBUG_STATUS dr6 = PDEBUG_STATUS(&context->Dr6);

    context->Dr0 = ULONG(addr);
    dr7->L0 = 1, dr7->RWE0 = 0, dr7->LEN0 = 0, dr6->RO = 0;
    return stack.size() - 1;
}

VOID AddFPO(PVOID base, PSTR name)
{
    PIMAGE_DEBUG_INFORMATION idi = MapDebugInformation(0, name, getenv("_NT_SYMBOL_PATH"), 0);
    if (idi == 0) return;

    for (ULONG i = 0; i < idi->NumberOfFpoTableEntries; i++) {
        PVOID func = PVOID(PBYTE(base) + idi->FpoTableEntries[i].ulOffStart);
        enters_t::iterator entry = enters.find(func);
        if (entry != enters.end())
            entry->second.argc = idi->FpoTableEntries[i].cdwParams;
    }

    UnmapDebugInformation(idi);
}

VOID InsertBreakPoints(PVOID base)
{
    IMAGE_DOS_HEADER dos;
    IMAGE_NT_HEADERS nt;
    BOOL rv;

    rv = ReadProcessMemory(hProcess, base,
        &dos, sizeof dos, 0);
    if (rv != TRUE) throw Error(__LINE__, GetLastError());

    rv = ReadProcessMemory(hProcess, PVOID(base) + dos.e_lfanew,
        &nt, sizeof nt, 0);
    if (rv != TRUE) throw Error(__LINE__, GetLastError());

    PIMAGE_DATA_DIRECTORY expdir = nt.OptionalHeader.DataDirectory + IMAGE_DIRECTORY_ENTRY_EXPORT;
    ULONG size = expdir->Size;
    ULONG addr = expdir->VirtualAddress;
```
Exceptions and Debugging: User Mode Debuggers

PIMAGE_EXPORT_DIRECTORY exports = PIMAGE_EXPORT_DIRECTORY(malloc(size));
rv = ReadProcessMemory(hProcess, PBYTE(base) + addr, exports, size, 0);
if (rv != TRUE) throw Error(__LINE__, GetLastError());

PULONG functions = PULONG(PBYTE(exports) + addr + ULONG(exports->AddressOfFunctions));
PUSHORT ordinals = PUSHORT(PBYTE(exports) + addr + ULONG(exports->AddressOfNameOrdinals));
PULONG fnames = PULONG(PBYTE(exports) + addr + ULONG(exports->AddressOfNames));
for (ULONG i = 0; i < exports->NumberOfNames; i++) {
    ULONG ord = ordinals[i];
    if (functions[ord] < addr || functions[ord] >= addr + size) {
        PBYTE func = PBYTE(base) + functions[ord];
        PSTR name = PSTR(PBYTE(exports) - addr + fnames[i]);
        BYTE op = InsertBreakPoint(func);
        if (enters.find(func) == enters.end())
            enters[func] = enter(name, op);
    }
}
AddFPO(base, PSTR(PBYTE(exports) + addr + exports->Name));
}

VOID RemoveDeadBreakPoints()
{
    enters_t dead(enters);
    BYTE op;
    for (enters_t::iterator entry = dead.begin();
        entry != dead.end(); entry++)
        if (ReadProcessMemory(hProcess, PVOID(entry->first), &op, sizeof op, 0) == FALSE)
            enters.erase(entry->first);
}

VOID ReportEntry(PCONTEXT context, NT::PDEBUG_MESSAGE dm)
{
    ULONG stack[17];
    CHAR buf[512];
    PVOID addr = dm->u.Exception.ExceptionRecord.ExceptionAddress;
    enter& entry = enters[addr];
    PCSTR s = entry.name;
    if (*s == '?' && UnDecorateSymbolName(s, buf, sizeof buf - 1, 0) > 0)
        s = buf;
    ULONG argc = min(ULONG(elements(stack)) - 1, entry.argc);
    BOOL rv = ReadProcessMemory(hProcess, PVOID(context->Esp),
        stack, sizeof stack[0] * (1 + argc), 0);
    ULONG now = GetTickCount() - StartTime;
Exceptions and Debugging: User Mode Debuggers

ULONG n = rv ? ReturnBreak(context, PVOID(stack[0]),
    ULONG(dm->PortMessage.ClientId.UniqueThread))
    : 0;
printf("\n%d.%02d %4x %*s (
    now / 1000, (now % 1000) / 10,
    ULONG(dm->PortMessage.ClientId.UniqueThread), n, "", s);
if (rv == TRUE) {
    switch (argc) {
    case 0: break;
    case 1: printf("%x", stack[1]); break;
    case 2: printf("%x, %x", stack[1], stack[2]); break;
    case 3: printf("%x, %x, %x", stack[1], stack[2], stack[3]); break;
    default:
        printf("%x, %x, %x", stack[1], stack[2], stack[3], stack[4]);
        for (ULONG i = 5; i <= argc; i++) printf("", %x, stack[i]);
    }
    printf(");
}
VOID ReportExit(PCONTEXT context)
{
    printf(" -> %x", context->Eax);
}
ULONG HandleBreakPoint(NT::PDEBUG_MESSAGE dm)
{
    PVOID addr = dm->u.Exception.ExceptionRecord.ExceptionAddress;
    enters_t::iterator entry = enters.find(addr);
    if (entry != enters.end() && entry->second.opcode != 0xcc) {
        HANDLE hThread = threads[ULONG(dm->PortMessage.ClientId.UniqueThread)];
        CONTEXT context;
        context.ContextFlags = CONTEXT_DEBUG_REGISTERS | CONTEXT_CONTROL;
        GetThreadContext(hThread, &context);
        ReportEntry(&context, dm);
        StepBreakPoint(&context, ULONG(dm->PortMessage.ClientId.UniqueThread),
            addr, entry->second.opcode);
        SetThreadContext(hThread, &context);
    }
    else {
        if (entry != enters.end() && entry->second.name != 0)
            printf("\n\nDebug exception at %s
", entry->second.name);
        else
            printf("\n\nDebug exception at %p
", addr);
    }
    return DBG_CONTINUE;
}
ULONG HandleSingleStep(NT::PDEBUG_MESSAGE dm)
452  **Exceptions and Debugging:** User Mode Debuggers

```
{
    CONTEXT context;
    steps_t::iterator step
        = steps.find(ULONG(dm->PortMessage.ClientId.UniqueThread));

    if (step != steps.end()) {
        if (!Discard) ReinsertBreakPoint(step->second);
        steps.erase(step);
        return DBG_CONTINUE;
    }

    PVOID eaddr = dm->u.Exception.ExceptionRecord.ExceptionAddress;
    std::vector<leave>& stack
        = leaves[ULONG(dm->PortMessage.ClientId.UniqueThread)];
    if (!stack.empty() && stack.back().eip == eaddr) stack.pop_back();
    PVOID iaddr = stack.empty() ? 0 : stack.back().eip;
    HANDLE hThread = threads[ULONG(dm->PortMessage.ClientId.UniqueThread)];

    context.ContextFlags
        = CONTEXT_DEBUG_REGISTERS | CONTEXT_CONTROL | CONTEXT_INTEGER;
    GetThreadContext(hThread, &context);
    PDEBUG_CONTROL dr7 = PDEBUG_CONTROL(&context.Dr7);
    PDEBUG_STATUS  dr6 = PDEBUG_STATUS(&context.Dr6);
    context.Dr0 = ULONG(iaddr);
    dr7->L0 = 1, dr7->RWE0 = 0, dr7->LEN0 = 0, dr6->B0 = 0;
    if (iaddr == eaddr) context.EFlags |= 0x100, dr7->L0 = 0;
    SetThreadContext(hThread, &context);
    ReportExit(&context);
    return DBG_CONTINUE;
}

ULONG HandleExceptionEvent(NT::PDEBUG_MESSAGE dm)
{
    switch (dm->u.Exception.ExceptionRecord.ExceptionCode) {
    case EXCEPTION_BREAKPOINT:
        return HandleBreakPoint(dm);
    case EXCEPTION_SINGLE_STEP:
        return HandleSingleStep(dm);
    default:
        printf("\nException %x at %p\n",
            dm->u.Exception.ExceptionRecord.ExceptionCode,
            dm->u.Exception.ExceptionRecord.ExceptionAddress);
    }
    return DBG_EXCEPTION_NOT_HANDLED;
}
```
ULONG HandleCreateProcessThreadEvent(NT::PDEBUG_MESSAGE dm)
{
    printf("Process %x, Thread create %x\n", 
        dm->PortMessage.ClientId.UniqueProcess, 
        dm->PortMessage.ClientId.UniqueThread);

    NT::OBJECT_ATTRIBUTES oa = { sizeof oa };
    HANDLE hThread;

    NT::ZwOpenThread(&hThread, THREAD_ALL_ACCESS, 
        &oa, &dm->PortMessage.ClientId);

    threads[ULONG(dm->PortMessage.ClientId.UniqueThread)]
        = hThread;

    leaves[ULONG(dm->PortMessage.ClientId.UniqueThread)]
        = std::vector<leave>();

    return DBG_CONTINUE;
}

ULONG HandleExitThreadEvent(NT::PDEBUG_MESSAGE dm)
{
    printf("Thread %x exit code %x\n", 
        dm->PortMessage.ClientId.UniqueThread, 
        dm->u.ExitThread.ExitCode);

    leaves.erase(ULONG(dm->PortMessage.ClientId.UniqueThread));

    return DBG_CONTINUE;
}

ULONG HandleExitProcessEvent(NT::PDEBUG_MESSAGE dm)
{
    printf("Process %x exit code %x\n", 
        dm->PortMessage.ClientId.UniqueProcess, 
        dm->u.ExitProcess.ExitCode);

    leaves.erase(ULONG(dm->PortMessage.ClientId.UniqueThread));

    return DBG_CONTINUE;
}

ULONG HandleLoadDllEvent(NT::PDEBUG_MESSAGE dm)
{
    InsertBreakPoints(dm->u.LoadDll.Base);

    return DBG_CONTINUE;
}

ULONG HandleUnloadDllEvent(NT::PDEBUG_MESSAGE)
{
    RemoveDeadBreakPoints();

    return DBG_CONTINUE;
}

BOOL WINAPI HandlerRoutine(ULONG event)
{
    if (event != CTRL_C_EVENT || Discard == TRUE)
        TerminateProcess(hProcess, 0);

    if (event == CTRL_C_EVENT)
**Exceptions and Debugging: User Mode Debuggers**

```c
Discard = TRUE;
return TRUE;
}

HANDLE StartDebuggee(HANDLE hPort)
{
    PROCESS_INFORMATION pi;
    STARTUPINFO si = {sizeof si};
    PSTR cmd = strchr(GetCommandLine(), ' ') + 1;
    CreateProcess(0, cmd, 0, 0, 0, CREATE_SUSPENDED, 0, 0, &si, &pi);
    NT::ZwSetInformationProcess(pi.hProcess, NT::ProcessDebugPort,
                               &hPort, sizeof hPort);
    ResumeThread(pi.hThread);
    CloseHandle(pi.hThread);
    return pi.hProcess;
}

int main(int argc, wchar_t *argv[])
{
    if (argc == 1) return 0;
    SetConsoleCtrlHandler(HandlerRoutine, TRUE);
    NT::OBJECT_ATTRIBUTES oa = {sizeof oa};
    HANDLE hPort;
    NT::ZwCreatePort(&hPort, &oa, 0, 0x78, 0);
    hProcess = StartDebuggee(hPort);
    StartTime = GetTickCount();
    NT::DEBUG_MESSAGE dm;
    do {
        NT::ZwReplyWaitReceivePort(hPort, 0, 0, &dm.PortMessage);
        try {
            switch (dm.EventCode + 1) {
                case EXCEPTION_DEBUG_EVENT:
                    dm.Status = HandleExceptionEvent(&dm);
                    break;

                case CREATE_THREAD_DEBUG_EVENT:
                    case CREATE_PROCESS_DEBUG_EVENT:
                    case EXIT_THREAD_DEBUG_EVENT:
                    case EXIT_PROCESS_DEBUG_EVENT:
                        dm.Status = HandleExitProcessEvent(&dm);
                        break;
            }
        } catch (const std::exception &e) {
            // Handle exception
        }
    } while (NT::ZwWaitForMultipleObjects(0, 0, 0, TRUE, INFINITE));
}
```
case LOAD_DLL_DEBUG_EVENT:
    dm.Status = HandleLoadDllEvent(&dm);
    break;

case UNLOAD_DLL_DEBUG_EVENT:
    dm.Status = HandleUnloadDllEvent(&dm);
    break;

default:
    dm.Status = DBG_CONTINUE;
    printf("nUnusual event %lx\n", dm.EventCode);
    break;
}

catch (Error e) {
    printf("Error %ld on line %ld\n", e.code, e.line);
    dm.EventCode = EXIT_PROCESS_DEBUG_EVENT - 1;
}

NT::ZwReplyPort(hPort, &dm.PortMessage);

} while (dm.EventCode + 1 != EXIT_PROCESS_DEBUG_EVENT);

return 0;

As a utility, Example D.4 is useful for understanding the relationship between Win32 functions and the native system services. By attempting to show the call nesting, this example makes it possible to see which system services are invoked during a call to a Win32 function.

Contrary to the advice of—"Don’t document bugs—fix them!" one known problem with Example D.4 is that it does not suspend all the other threads in the process while single stepping a thread over a breakpoint. This would potentially allow other threads to call an exported function when the breakpoint instruction at its entry point is temporarily removed.