Gemini Lite: A Non-intrusive Debugger for Windows NT

Ryan S. Wallach
Lucent Technologies

Abstract

It is frequently useful to debug a running software system in a production environment with a symbolic debugger without interfering with the operation of the system. The user of such a debugger may want to inspect data or trigger some data collecting operations whenever the running program hits an arbitrary address. Terminating or initiating the debugging session must also be transparent to users of the system. Debuggers available for Windows NT (or any debugger written with the Win32 debugging API) cannot detach from a running process without killing it, so they are unsuitable for debugging live systems. This paper presents the design of Gemini Lite, a debugger written without the Win32 debugger API, which has the capabilities needed to debug running production systems.

1 Introduction

Lucent Technologies' DEFINITY® Enterprise Communications System is a highly reliable (99.999% uptime) large communications server. The software base contains several million lines of code. Like any large software system, each release of DEFINITY contains software bugs that are not discovered until the system has been installed at a customer's premises. Many bugs are easily reproducible in the development lab from a customer's description. It is impossible, however, to precisely reproduce the conditions of an installed system, and therefore some problems cannot be reproduced in the lab. When this happens, it is necessary to debug the system at the customer's premises without disturbing its operation.

DEFINITY is implemented as a collection of processes in a multitasking proprietary operating system. It contains a proprietary client/server debugger, Gemini, which lets support engineers at a Lucent site securely connect to a customer's system and non-intrusively debug the software. Gemini consists of a small DEFINITY server process (known as the *agent*) which controls the target processes, and a UNIX client process (known as the *host*) that accepts commands and sends messages to the agent to execute them. The host process runs on the support engineer's workstation. It has access to symbolic information for the DEFINITY processes, so it translates symbol names into addresses for the agent.

Gemini supports the traditional model of debugging, that is, the user can set a breakpoint, wait for the process to halt, then single step it and examine data to find the cause of the bug. These features are often used during product development under controlled conditions. However, DEFINITY processes are interrelated and time-dependent. If a process is halted for more than a few milliseconds, the system could (in the course of error recovery) reinitialize itself, and this could disrupt the customer's business. When debugging a live system, then, special debugging capabilities are needed. Gemini provides four key features:

1. Gemini breakpoints can have lists of commands (called action lists) associated with them, and the breakpoints can remain active even when the host isn't running (the agent is always running). When a process hits a breakpoint with an action list, the agent runs the commands, logging any output to an internal buffer, and resumes execution of the debugged process. This mechanism is frequently used to determine where processes are executing and what the relevant data looked like when the breakpoint was hit. Lucent support engineers typically use the host to set breakpoints with action lists and then they exit the debugger and come back hours or days later and view the output buffer to determine if the breakpoint was hit and gather the output from the action list commands.
2. Gemini can debug all the processes in the system in the same session of the debugger. It is even possible to set up breakpoints with action list commands that can manipulate other processes when the first process hits the breakpoint.
3. Gemini is non-intrusive. Users can read and write memory, set and clear breakpoints, and look at the status of DEFINITY processes without halting them. Most of the important data in DEFINITY is global, so developers frequently need to read from (and write to) these data structures without halting the process they're debugging.
4. Gemini can attach to running processes and detach from them cleanly, without interfering with their operation.

In February 2000, Lucent introduced its DEFINITY ONE™ Communications System. DEFINITY ONE contains DEFINITY plus other co-resident applications running on one system under Windows NT 4.0 (all references to Windows NT in this paper refer to this version). DEFINITY ONE required a debugger to run on the platform with the same capabilities as Gemini as well as the ability to debug multithreaded processes. Because no suitable off-the-shelf debugger could be found, we developed our own debugger, Gemini Lite, to provide these capabilities for DEFINITY ONE.

2 The Search for a Debugger on Windows NT

In order to understand the rationale behind Gemini Lite's design, it is important to understand the Win32 Debugging API and how this affects off-the-shelf debuggers for Windows NT.

2.1 The Win32 Debugging API

Windows NT provides an API for developers to create their own debugger [1]. Typically, a debugger attaches to a running process by calling DebugActiveProcess() with the target process id as an argument. This registers the debugger with the operating system. The debugger then calls WaitForDebugEvent() which makes the calling thread of the debugger block until a debugging event is sent to it by the operating system. Windows initially sends events to the debugger to give it handles to each thread in the process being debugged. The debugger then receives events when threads in the target process hit a breakpoint, generate an unhandled exception, etc.[2]

The Win32 debugging API is similar to the ptrace() system call interface used by some UNIX debuggers such as GDB [3] (some versions of UNIX do debugging through the /proc filesystem instead of through ptrace(), and there is nothing similar in Windows NT). To initiate a debug session, a UNIX debugger can call ptrace with a PTRACE_ATTACH request, which allows it to control the target process as if it were its parent. This is analogous to NT's DebugActiveProcess() call. After the debugger has attached to the process, it receives SIGCHLDs when something happens to the target, or it can do a wait() or waitpid() to receive notification of events from the target. The wait() or waitpid() calls are analogous to NT's WaitForDebugEvent() calls.

The substantial difference between the Windows NT and UNIX APIs is that a UNIX debugger can call ptrace() with a PTRACE_DETACH request to disconnect the debugger from the target. The target continues to run after the debugger is disconnected, and the parent-child relationship between the debugger and the target is destroyed. Windows NT does not provide a clean way to detach a target, i.e., there is no call to undo a DebugActiveProcess() request. Furthermore, once the process that has initiated a debugging session exits, Windows NT kills the processes that it was debugging [4]. Microsoft plans to address this issue in a later release of Windows NT (in NT 6.0 or later) [5], but for now there is no workaround.

2.2 Debuggers Investigated

We investigated several Microsoft debuggers for Windows NT (WinDbg, Visual C++ IDE, and ntsd) [6] [7][8] to determine if they could be used for DEFINITY ONE. Each of these appears to use the Win32 API and kills the debugged process when it exits. Since the Microsoft provided debuggers could not satisfy our requirement that they be able to cleanly detach, we investigated commercially available debuggers such as GDB, NuMega's SoftICE [9], and Oasys MULTI [10]. These exhibited the same problem.

Many other debuggers have been built for multithreaded applications on different operating systems [11][12][13], and multi-process, non-intrusive debuggers have also been built [14]. These debuggers have all been built either by using the native debugger API provided by the operating system or extending it to meet the needs of the debugger. Building a debugger (or adopting one of these debuggers) on Windows NT using the debugging API is not acceptable for the reasons discussed above, and since Windows NT is not an open source operating system, it would not be possible to extend it to support one of these debuggers.

GDB is perhaps the most common open-source debugger available, and we considered adapting it. Besides the fact that GDB uses the Win32 debugging API, there were other reasons that we chose not to use it. First, GDB can only debug a single process at a time and is intrusive [15]. This behavior stems from the core of GDB; modifying this would be, says the Cygnus White Paper on GDB, "a daunting task because of its complexities…". Furthermore, our project used the Microsoft Visual C++ 5.0 compiler, and the version of GDB available during our development cycle only supported COFF format symbolic information in the executables. The Microsoft compiler only emits CodeView symbolic information in executable files (and DLLs).

Because no suitable off-the-shelf debugger could be found, we developed Gemini Lite. Gemini Lite is a general-purpose Windows NT debugger. It can be used to debug any NT process (not just DEFINITY) assuming the process is properly linked. Gemini Lite is non-intrusive and does not use the Win32 debugging API. It has the basic features of other debuggers, but its architecture permits it to have unattended action lists and to debug processes without killing them once it exits.

3 Design of Gemini Lite

3.1 Overview

The Win32 API provides the basic mechanisms to implement basic debugging features in Gemini Lite. Table 1 shows which Win32 functions can be used to implement the core features of the debugger [16].

Feature	Win32 API Used
Read/Write Memory, set/clear breakpoints	ReadProcessMemory(), WriteProcessMemory()
Read/Write Registers, control single stepping	GetThreadContext(), SetThreadContext()
Halt/Resume a thread	SuspendThread(), ResumeThread()
Determine when a thread hits a breakpoint, steps, or has some other exception	Structured Exception Handling mechanisms

Table 1. Win32 support for debugger features

Win32 calls that refer to the target's memory require a handle to the target process, which can be obtained from a call to OpenProcess(). Similarly, calls that refer to threads (e.g., SuspendThread()), require a handle to the target thread, which can only be obtained by that thread (or the thread which created it) [17] because there is no OpenThread() call in Windows NT 4.0 (Microsoft has added this to the Win32 API in Windows 2000). Using structured exception handling mechanisms for breakpoints presents a similar problem; a process cannot change the exception mechanisms of threads in other processes.

When DebugActiveProcess() is used to implement a debugger, Windows NT sends the debugger the handles to the desired threads. Without using this API, the only way for the debugger to have access to the thread handles is for it (or the part of it that actually controls other processes) to be integrated into the application code. Due to the size of the DEFINITY code base, it was not feasible to change the application code to accommodate the debugger. We used a client/server approach to separate the portion of Gemini Lite that interacts with the user from the portion that controls processes, which must be somehow linked into the application.

The first part of Gemini Lite, the debugger process, is what the user runs to access the debugger. It acts like DEFINITY's Gemini host, accepting input from the user and sending the input to the server to be parsed and executed. The server part of the architecture, which is the core of Gemini Lite, is a DLL that is linked with the applications that can be debugged (for the rest of this paper, "the DLL" refers to this). The DLL takes the place of the Gemini agent and is responsible for parsing and executing the commands sent by the debugger process. The DLL must be linked with both the debugger process and all the target processes.

To force the application processes to link with the DLL without changing their code, they must be linked (with the Visual C++ linker) using the **–include <symbol>** directive and the appropriate export library for the DLL. The **–include** directive places a reference to the specified symbol (which is some globally exported symbol in the DLL) into the executable, which forces the DLL to be loaded when the process is run [18]. This limits the utility of the debugger somewhat, as it can only debug processes that are linked with the DLL, but for DEFINITY ONE this was an acceptable constraint.

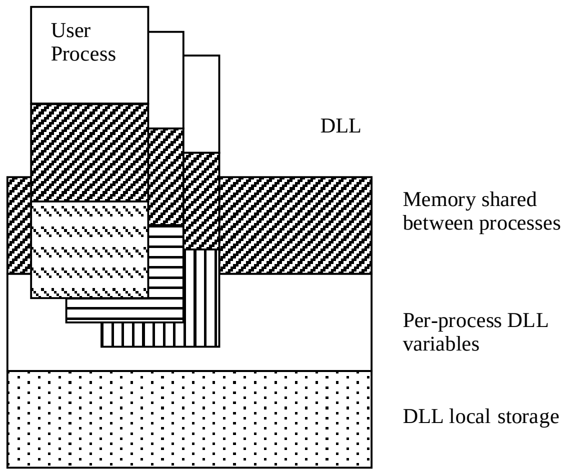

Figure 1. Overview of Gemini Lite's Architecture

A general overview Gemini Lite's architecture appears in Figure 1. The figure illustrates that all processes linked with the DLL share some common memory. The contents of the shared memory are defined by the

DLL; it contains exported functions as well as shared data. The DLL defines another set of exported variables; this set of variables has unique copies in each of the debugged processes. The DLL also has non-exported code and data that are not visible to them.

Windows NT forces each process linked with the DLL to call the DllMain() function in the DLL when the process and its threads are created and when they exit [19]. The Gemini Lite DLL uses this property to obtain handles to each debuggable thread (i.e., a thread in a process linked with the DLL) by having the thread store a handle to itself in the shared memory area of the DLL. DllMain also has each thread set its unhandled exception filter to a routine inside the DLL. This exception filter is used to catch breakpoint and single step exceptions, which would typically not be caught by the application. All other exceptions that are caught by this handler are sent back to Windows NT to resolve, although they could just as easily be handled by the debugger and reported to the user.

The debugger itself is just another user process that can call routines inside the DLL that implement its functions. These routines are exported to all processes, not just the debugger, and this architecture makes it possible for processes to call debugger routines when they hit breakpoints, which will be discussed in detail below.

3.2 *Shared Memory in the DLL*

The shared memory area of the DLL contains data that must be shared between the debugger and the user processes. The data structures must be statically allocated at compile time because any memory dynamically allocated by one process would not be in the address space of other processes, including the debugger. Furthermore, there is no guarantee that the DLL will be mapped to the same address space in each user process, so any traditional data structure that uses pointers (e.g., linked lists or hash tables) is not suitable for the DLL. Array-based hash tables, lists, and queue template classes were defined to hold the shared data.

The data structures contained in the shared memory area include:

1. Information about each process registered with the DLL (e.g., pid and creation time)
2. Information about each thread registered (e.g., thread id, handle to the thread, state of the thread)
3. Information about each breakpoint set (e.g., process in which it's set, address, commands to run when hit)
4. A queue that contains messages generated from functions in the DLL to be displayed by the debugger.

These data structures are protected by mutexes to ensure correctness.

The shared memory area also contains variables that track whether the debugger is running. Since the debugger is a user process, it also calls DllMain() when it starts. DllMain() checks the name of each calling process, and when it finds the name of the debugger process (a predefined name), it notes that the debugger is running. Another mechanism could also be used in DllMain() to determine which user process is the debugger. The DLL needs to know whether the debugger is attached because certain status messages (e.g., breakpoints being hit) are written to a queue in the shared memory area by functions in the DLL. The debugger has a thread that looks at this queue and displays the messages to the user.

3.3 *Process and Thread Registration in the DLL*

As mentioned before, Windows NT forces each process and thread linked with the DLL to call DllMain() when they are created. NT passes a parameter to DllMain() that indicates the reason for the call. When new processes start up, they call DllMain() one or more times. A process's first call to DllMain() has this parameter set to DLL_PROCESS_ATTACH. This notifies the DLL that the process (and its primary thread) has attached to the DLL. Subsequent calls by threads in the process to DllMain() set the parameter to DLL_THREAD_ATTACH and inform the DLL that additional threads in the process have been created.

When the Gemini Lite DLL's DllMain() is called with a DLL_PROCESS_ATTACH message, the DLL determines whether the process is the debugger or an application process. As mentioned above, if the process is the debugger, the DLL stores its pid in shared memory and sets a status variable in the DLL to reflect that the debugger is active. For the primary thread (and other threads) of user processes, the DLL creates an object to represent the thread in its shared memory area. The thread id and handle are stored in the object. DllMain() then sets the thread's unhandled exception filter to point to a routine inside the DLL.

Processes and threads also notify the DLL when they exit normally. When a thread exits, NT forces it to call DllMain() with a reason of DLL_THREAD_DETACH. Similarly, when a process exits, NT forces it to call DllMain() with a reason of DLL_PROCESS_DETACH. Note that the call with DLL_THREAD_DETACH is not made for all threads that are running when the process exits; only the DLL_PROCESS_DETACH call is made. When the

DLL gets these calls, it frees the object and associated data structures that were allocated for the thread (or threads, if the process detached) including its breakpoints.

In some circumstances (e.g., a call to TerminateProcess() or TerminateThread()), it is possible that processes and threads can be terminated without calling DllMain(). When this happens, the DLL does not know that the process or thread is gone, so it cannot free the related data structures. Because the tables holding the data are statically allocated and were sized to accommodate the number of threads and processes running in DEFINITY ONE, it is possible that they may fill up with information about processes and threads that no longer exist.

If the tables are full when the DLL attempts to register a process, the DLL checks all registered threads to make sure that they are still valid, and it frees up entries that are no longer valid.

Because the debugger is just another user process, the DLL can detect when it exits through its call to DllMain(). In order to prevent any application processes that have breakpoints set from stopping, the DLL disables all breakpoints that may have been set in other processes and it resumes execution of any threads that may have been stopped when it detects that the debugger exited.

3.4 Debugger Process

The Gemini Lite debugger process has two threads. The main thread runs in a loop which prompts the user for commands, reads the command line, and calls the appropriate functions in the DLL to parse and execute the command. The second thread repeatedly locks the mutex protecting the message queue in the DLL shared memory, removes and displays any messages found in the queue, then releases the mutex. As a result, the user is immediately informed of events such as breakpoints being hit regardless of what he or she may be doing in the debugger (typing commands or viewing output).

4 Implementation of Debugging Features

4.1 Symbolic Debugging

Debuggers like GDB typically read symbolic information for the process they are debugging from the executable file and build internal symbol tables for use by the debugger. Gemini Lite does not directly read the symbolic information for the processes and threads that it debugs. Instead, it relies on the Win32 symbol handling routines contained in IMAGEHLP.DLL [20]. These routines provide the capability to obtain an address in a running process from the name of a global symbol and vice-versa. The first time the user issues a command for a thread in a process that takes an address as a parameter, Gemini Lite calls SymInitialize() and passes it the handle to the process to initialize the symbol handler. It then loads the symbols for the process by enumerating all its modules and calling SymLoadModule() for each of them. Once the symbols have been loaded, Gemini Lite uses SymGetSymFromName() to translate global symbol names into an address or SymGetSymFromAddress() to translate an address into a global name.

In Windows NT 4.0, IMAGEHLP.DLL does not provide facilities for translating a file name and line number into an address and vice versa. Microsoft has added the SymGetLineFromAddr() and SymGetLineFromName() functions to the Win32 API in Windows 2000 to accomplish this. In order to perform this function in Windows NT 4.0, a program would have to directly examine the CodeView debugging information in the executables (or in separate .DBG files). Time constraints only permitted us to display file and line number information in Gemini Lite's disassembly routines. Other Gemini Lite commands (such as for setting breakpoints) cannot accept a file and line number in place of a text address.

Use of the IMAGEHLP.DLL symbol handling functions requires that the DLLs and EXEs that make up the processes being debugged are compiled with debugging information. The debugging information must be compiled into the objects, not placed in a program database (PDB) file. However, it is usually undesirable to ship production code without stripping debugging information. To avoid this, we used the **rebase** tool shipped with Visual C++ to strip the debugging information from the compiled objects and place it in separate .DBG files. When the application needs to be debugged, the .DBG files are copied to the target machine, and then the _NT_SYMBOL_PATH environment variable is set before running Gemini Lite. This environment variable tells the IMAGEHLP.DLL symbol handling routines where to find the symbols. We ship the symbol files (in an encrypted form) with the DEFINITY ONE system. When support engineers need to debug, they use Windows RAS or a TCP/IP network to establish a connection to the system. They then decrypt the symbol files and run the debugger in a window directly on the target system.

4.2 Stopping and Restarting Execution

The debugger can force threads to halt execution by calling SuspendThread() with the handle to the thread. The debugger obtains the handle from the shared memory area in the DLL. Before using the handle, the debugger must call DuplicateHandle() to obtain a handle in its context; the handle stored in the DLL is a handle in the context of the process that registered with the DLL. To resume execution of a thread, the debugger calls ResumeThread(), passing it the handle to the thread.

4.3 Reading and Writing Memory

The debugger commands that need to read or write memory do so by calling ReadProcessMemory() and WriteProcessMemory(). The debugger must have permission to read the memory of the processes it's debugging. When the debugger is debugging processes started by the same user, this is not a problem. For DEFINITY ONE, we require that the debugger can be started only by a privileged user. Some processes we need to debug are started by a system service. Ordinarily, a user process does not have permission to access a system service. We modified the default discretionary access control lists (DACLs) of our system level processes to give the account that can run the debugger full access to them.

4.4 Reading and Writing Registers

Registers in a thread can only be read by reading the thread context. Debugger commands that need to read registers first use SuspendThread() to stop the thread unless it is already halted. They then call GetThreadContext() to retrieve the context of the thread, which includes the contents of the registers. After the context is obtained, ResumeThread() is called if the thread needs to continue execution.

To write to a register, the context image returned by GetThreadContext() is modified to contain the updated register value, then SetThreadContext() is used to write the modified context back to the thread.

4.5 Breakpoints

Like most debuggers for software running on x86 processors, Gemini Lite sets breakpoints in a process by replacing the first byte of the instruction at the breakpoint address with 0xcc (INT3). The original instruction byte is saved in the record for the breakpoint in the shared memory area of the DLL so that it can be restored later. Because all threads in a process have the same address space, a breakpoint set in a process will affect all the threads in the process.

Figure 2 illustrates the sequence of events that occurs when a thread hits a breakpoint. First, the thread raises a breakpoint exception when it executes the instruction at the breakpoint address. The system stores the thread's context in a context record (the value of the EIP register in the record is set to the address where the thread encountered the exception) and forces the thread to call the appropriate exception filter. If no other exception filter handles breakpoint exceptions (which is a requirement for processes linked with the DLL), then the exception filter in the DLL (which was set as the unhandled exception for the process when it attached to the DLL) will be called. The exception filter receives a pointer to the context record as well as a pointer to an exception record, which contains the exception code, the address at which the exception occurred, and other information.

In the exception filter in the DLL, the thread first examines the exception record to determine the cause of the exception. If the type is not EXCEPTION_BREAKPOINT or EXCEPTION_SINGLE_STEP, then the exception filter will return EXCEPTION_CONTINUE_SEARCH, which will force NT to handle the exception. This will either terminate the process or invoke the system debugger, depending on the system's registry settings.

If the exception type is EXCEPTION_BREAKPOINT, then the thread checks the list of breakpoints in shared memory of the DLL to determine if a breakpoint was set at the exception address. If no breakpoint is found, there is no way for the thread to continue, so the filter will return EXCEPTION_CONTINUE_SEARCH.

If a breakpoint is found, the thread determines if it should halt. Breakpoints may have a threshold stored in the object representing them that specifies the number of times the breakpoint is to be hit before a thread will stop. Also, the thread will only halt if the debugger is running, as indicated by the variable that the debugger sets when it registers with the DLL.

If the thread determines that it must halt, it creates a message notifying the user that the breakpoint has been hit, and it puts it in the message queue for the debugger. It sets a variable in its record in the DLL's shared memory indicating that it is suspended due to the breakpoint, and then it suspends itself by calling SuspendThread() with its thread handle as a parameter. The debugger process that the user is running, meanwhile, contains two threads. One reads and executes commands from the user, and the other checks the message queue from the DLL. After the target thread puts the message into the queue indicating that it hit the breakpoint, this thread of the debugger displays it to the user.

When the user decides to resume execution of the thread, he or she gives the appropriate command to the debugger, which calls a function in the DLL. This function examines the record for the thread in shared memory. If the state of the thread indicates that it has

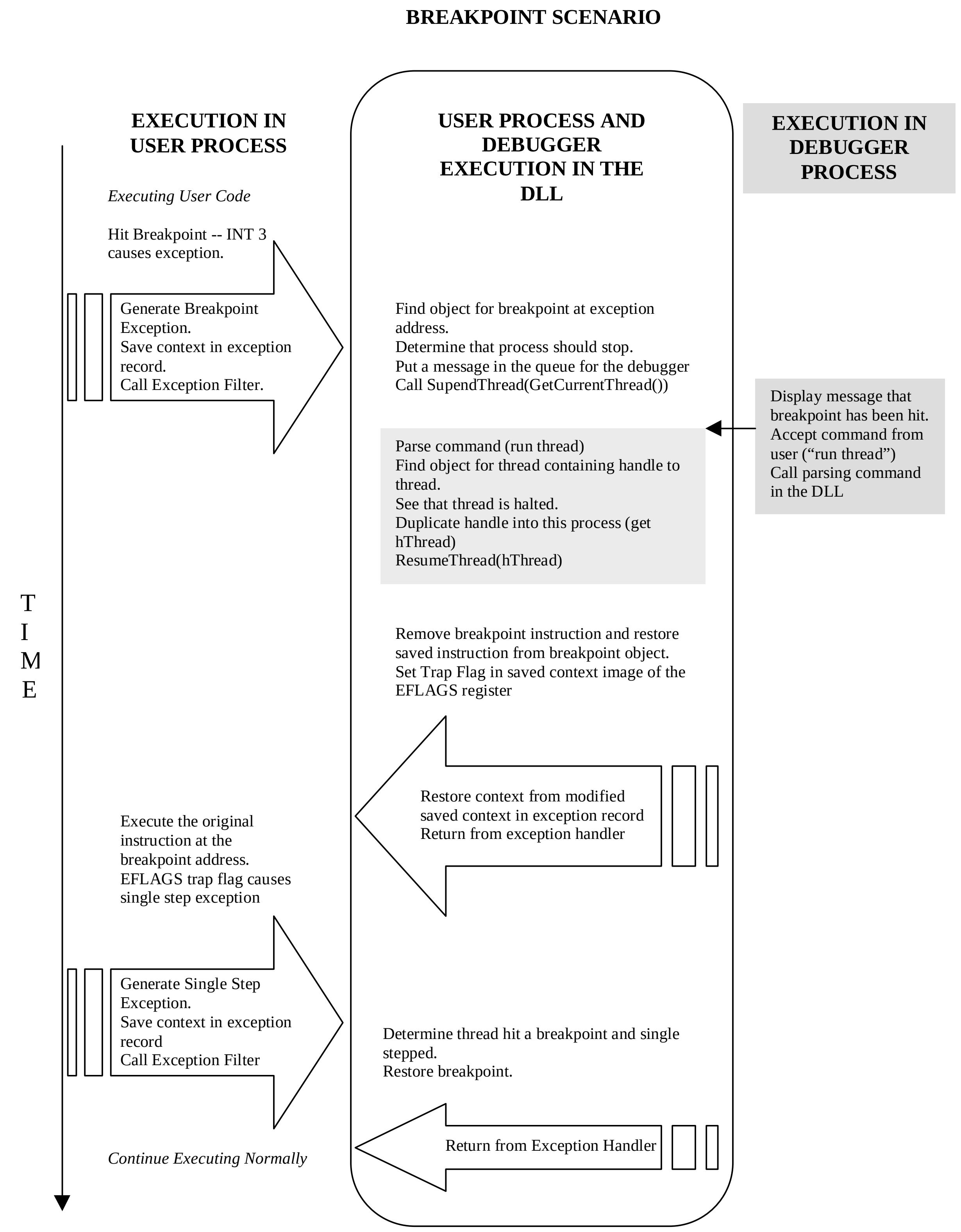
BREAKPOINT SCENARIO
EXECUTION IN USER PROCESS
USER PROCESS AND DEBUGGER EXECUTION IN THE DLL
EXECUTION IN DEBUGGER PROCESS
T I M E
Executing User Code
Hit Breakpoint -- INT 3 causes exception.
Generate Breakpoint Exception.
Save context in exception record.
Call Exception Filter.
Find object for breakpoint at exception address.
Determine that process should stop.
Put a message in the queue for the debugger
Call SupendThread(GetCurrentThread())
Display message that breakpoint has been hit.
Accept command from user ("run thread")
Call parsing command in the DLL
Parse command (run thread)
Find object for thread containing handle to thread.
See that thread is halted.
Duplicate handle into this process (get hThread)
ResumeThread(hThread)
Remove breakpoint instruction and restore saved instruction from breakpoint object.
Set Trap Flag in saved context image of the EFLAGS register
Restore context from modified saved context in exception record
Return from exception handler
Execute the original instruction at the breakpoint address.
EFLAGS trap flag causes single step exception
Generate Single Step Exception.
Save context in exception record
Call Exception Filter
Determine thread hit a breakpoint and single stepped.
Restore breakpoint.
Return from Exception Handler
Continue Executing Normally

Figure 2. View of execution when a thread hits a breakpoint.

been halted at a breakpoint, then the function gets the handle to the thread that is stored in shared memory and passes it to ResumeThread(). This wakes up the thread that hit the breakpoint.

When the thread wakes up it is still in the exception handler, and the breakpoint instruction is still at the breakpoint address. The thread replaces the breakpoint instruction with the original instruction, sets status variables in its record in shared memory to indicate that it has been resumed after a breakpoint, and sets the Trap Flag in the image of the EFLAGS register stored in the saved thread context that was passed to the exception handler. It then returns EXCEPTION_CONTINUE_EXECUTION. This forces NT to restore its context from the saved image (with the modified EFLAGS register) and continue executing where the exception occurred.

The thread executes the real instruction at the breakpoint address, and then, because the Trap Flag in the EFLAGS register is set, it generates a single step exception. Again, Windows NT forces the thread to call the unhandled exception filter. In the exception filter, the thread sees that the exception code is EXCEPTION_SINGLE_STEP. It checks the status variables in shared memory and figures out that it has single stepped after the previously hit breakpoint. The thread then saves the instruction at the breakpoint address, reinserts the breakpoint, and again the exception filter returns EXCEPTION_CONTINUE_EXECUTION, which lets the thread continue executing at the instruction after the breakpoint. The thread then continues executing until some other event occurs.

Gemini Lite’s handling of breakpoints differs from the traditional implementation. In a debugger written using the Win32 API, for example, when a thread hit a breakpoint, the *system* would suspend it. A debugger doing a WaitForDebugEvent() would be woken up, and it would decide whether to keep the process halted or restart it with a call to ContinueDebugEvent() (after replacing the breakpoint instruction with the original instruction and setting EFLAGS appropriately). In Gemini Lite, the *thread* decides itself whether it should be suspended, and it suspends itself. In both cases, the debugger causes the thread to resume execution. In the Win32 case, the thread resumes where it was stopped by the system, in the application code. In Gemini Lite, the thread resumes execution in the exception handler. When it returns from the handler, the system causes it to resume executing where the exception was raised.

4.6 Action Lists

An action list is a list of debugger commands to be executed when a breakpoint is hit. When the Gemini Lite user sets a breakpoint in a thread, he or she may also supply the action list. The action list is stored with the breakpoint information in the shared memory area in the DLL. In the exception filter, if the thread determines that a breakpoint has an action list, instead of calling SuspendThread(), it reads the list of action list commands from shared memory and passes them to the same function in the DLL that the debugger executable runs to execute commands that are input by the user. Since all the functions of Gemini Lite are also in the DLL, they can be executed just as if the user were giving them on the command line. The output from the commands is directed to a large circular buffer, also in shared memory. The output will stay in the buffer until the Gemini Lite user clears it. Note that if a breakpoint has an action list, Gemini Lite does not have to be running in order for the action list commands to execute, because the thread automatically resumes execution after the action list commands are run. This makes action lists very useful for unattended debugging. The user can set up the breakpoints with action list commands to dump data of interest, exit Gemini Lite, and come back later to examine the data.

The user can also set a flag in the DLL’s shared memory area that the thread will check after it executes the action list commands. If the flag is set and the debugger is running, the thread will generate a message for the debugger that tells the user that the breakpoint was hit. This feature can be used in combination with an empty action list to let the user know that the thread executed code at a particular instruction without having to halt the thread.

4.7 Single-Stepping

When a thread is stopped, either after being halted by the user or by hitting a breakpoint, the user may wish to step through the execution of the program being debugged. Because Gemini Lite relies only on the IMAGEHLP.DLL symbol handler to read debugging information, it does not have access to the information that links an address to the program source file and line number. Consequently, Gemini Lite can only step through a program any number of assembly-language instructions at a time.

The implementation of single stepping was seen above in the discussion of breakpoints. When the target thread is halted, the user’s single step command sets the Trap Flag in the EFLAGS register (by getting the thread context, modifying, and writing it back, if the thread is not halted after a breakpoint, or by modifying the saved context in the exception record, if it is), and resumes execution of the thread (by calling ResumeThread()). The user can specify the number of instructions to single-step; this number is stored in shared memory in the DLL.

After resuming execution, the target thread executes one instruction and generates an exception, sending it into the exception filter in the DLL. If the thread has stepped the desired number of instructions, it puts a message in the queue for the debugger to inform the user that it halted, then it calls SuspendThread() on itself. Otherwise, the thread decrements the step count, returns from the exception filter, and continues stepping.

After the thread is finally halted, the user can resume execution of the thread or single-step it again. As with breakpoints, when the thread is in the exception filter it checks to see if Gemini Lite is running before calling SuspendThread(). If the debugger is not present, the thread will not stop. This avoids the situation where a user requests a single step of a large number of instructions, but then exits the debugger before the stepping is completed.

5 Related Tools

Since the IMAGEHLP.DLL routines only locate global symbol names, we needed a set of tools to use with Gemini Lite which could show us the layout of structures in memory, addresses of individual array elements, and addresses of global functions and variables. In the UNIX environment, these functions are provided by tools like objdump (from GNU) and nm. On Windows NT, the Microsoft provided tools to do these things (such as dumpbin) are part of Visual C++ and cannot be run without it. We developed a standalone set of tools to do these things. The development was difficult, in part, because Microsoft's compilers emit symbolic information in a proprietary format (CodeView), and Microsoft does not provide any libraries for manipulating this information. We generated our own set of routines from Microsoft's symbolic debugging information specification [21].

6 Conclusion

Gemini Lite was used during the development of DEFINITY ONE to solve some difficult problems. In one case, an uninitialized variable was causing incorrect information to be displayed on DEFINITY's administration terminal. We set breakpoints with empty action lists both where we knew the code had executed and where we thought it should be executing. When these breakpoints are hit, Gemini Lite puts a message into its output buffer. By looking at the buffer, we were able to see where the code failed to branch as we thought it should. At that point, we used an action list to display a variable that determined where the code branched. After seeing that the value in this variable could not have been set by the code that had executed, a close examination of the code showed that the variable had not been initialized.

Our experience with Gemini Lite suggests some enhancements. First, Gemini Lite could be enhanced to read CodeView information from the processes it's debugging and maintain its own symbol table. With this information, Gemini Lite would have a knowledge of variable type information, mapping of source files and line numbers to addresses, locations and names of local variables in functions, and more information that would enable it to be a source level debugger instead of an assembly level debugger. A networked client-server approach to Gemini Lite has also been proposed which would eliminate the need to keep the symbol files (.DBG files) on the system being debugged.

7 Acknowledgements

I would like to thank Bhavesh Davda, Bill Lyford, and David Walters for their assistance during the development of Gemini Lite.

8 References

[1] "Platform SDK: Debugging and Error Handling: Debugging Reference: Debugging Functions", *MSDN Library*, Microsoft Corp., October, 1999.

[2] Kath, Randy. "The Win32 Debugging Application Programming Interface", Microsoft Corp., November 5, 1992.

[3] Stallman, Richard and Pesch, Roland. *Debugging with GDB, the GNU Source-Level Debugger, Fifth Edition*, Free Software Foundation, April, 1998.

[4] "PRB: Debugee Exits When the Debugger Exits, ID Q164205", *Microsoft Knowledge Base,* Microsoft Corp., February 28, 1997.

[5] Private Conversations with Microsoft Premier Support Representatives.

[6] "Platform SDK: Tools: Symbolic Debuggers", *MSDN Library*, Microsoft Corp., October, 1999.

[7] "Platform SDK: Tools: WinDbg", *MSDN Library,* Microsoft Corp., October, 1999.

[8] "Visual C++ User's Guide: Debugger", *MSDN Library,* Microsoft Corp., October, 1999.

[9] SoftICE is a trademark of NuMega Technologies, Inc.

[10] MULTI is a trademark of Green Hills Software and XEL, Inc.

[11] Buhr, Peter, Karsten, Martin, and Shih, Jun. "KDB: A Multi-threaded Debugger for Multi-threaded Applications". Proceedings of the

SIGMETRICS Symposium on Parallel and Distributed Tools, ACM, 1996, pp. 80 – 87.

[12] Caswell, Deborah, and Black, David. “Implementing a Mach Debugger for Multithreaded Applications”. Proceedings of the Winter 1990 USENIX Conference.

[13] Redell, David. “Experience with Topaz TeleDebugging”. Digital Equipment Corporation, Systems Research Center.

[14] Himelstein, Mark and Rowell, Peter. “Multi-process Debugging”. USENIX Conference Proceedings, Summer, 1985.

[15] Shebs, Stan. “GDB: An Open Source Debugger for Embedded Development”, Cygnus Support White Paper available at http://www.redhat.com/support/wpapers/cygnus_gdb.

[16] See the appropriate page for each function in *MSDN Library*, Microsoft Corp., October, 1999.

[17] “Platform SDK: DLLs, Processes, and Threads: Thread Handles and Identifiers”, *MSDN Library*, Microsoft Corp., October, 1999.

[18] “Visual C++ Programmer’s Guide: Compiler Reference”, *MSDN Library*, Microsoft Corp., October, 1999.

[19] Sarma, Debabrata. “DLLs for Beginners”, Microsoft Developer Support, November, 1996.

[20] Pietrek, Matt. “Under the Hood”, *Microsoft Systems Journal*, May, 1997.

[21] Smith, Steve, and Spalding, Dan, et al. “Visual C++ Symbolic Debug Information Specification, Revision 5, 32-Bit only September, 30,1996”, *MSDN Library*, Microsoft Corp., October, 1999.