Jacob R. Lorch and Alan Jay Smith

The VTrace Tool
Building a System Tracer for Windows NT and Windows 2000

This article assumes you're familiar with C, Windows NT, and Windows 2000

Level of Difficulty 1 2 3

SUMMARY This article describes the techniques used to construct VTrace, a system tracer for Windows NT and Windows 2000. VTrace collects data about processes, threads, messages, disk operations, network operations, and devices. The technique uses a DLL loaded into the address space of every process to intercept Win32 system calls; establishes hook functions for Windows NT kernel system calls; modifies the context switch code in memory to log context switches; and uses device filters to log accesses to devices.

For related articles see: Peering Inside the PE at http://msdn.microsoft.com/library/techart/msdn_peeringpe.htm and Learn System-Level Win32 Coding Techniques by Writing an API Spy Program (*MSJ*, December 1994).

For background information see: *Inside Windows NT* (Microsoft Press), *Windows NT File System Internals* (O'Reilly Associates), and *Programming the Microsoft Windows Driver Model* (Microsoft Press).

Writing a tracer for an operating system can be a nightmare. The size and complexity of an operating system complicates debugging, and it's especially tricky to debug code that runs before the system has fully started up. Many runs require rebooting the computer, and failed runs can require reinstalling the operating system or even reformatting the hard drive. Writing a tracer for Windows NT® and Windows® 2000 is even more difficult because source code isn't available, descriptions of internal operations are incomplete, and even the interface isn't always well documented.

In building VTrace, the tracer we created for Windows NT and Windows 2000, we had to deal with all these problems. We needed time-stamped traces of certain activities in Windows NT and Windows 2000 to study new energy management techniques for laptop computers. Because we were studying the effects of varying the CPU voltage and clock speed and of powering down various system components, we needed to know when power-consuming components (such as the CPU, the disk, and the network interface card) were active and what they were doing at each instant. This required traces of many different types of system objects: processes, threads, messages, waitable objects, key presses, file systems, disks, and the network. We wanted the tracer to be unintrusive and to respect the confidentiality of user data so that users would let us trace their systems. VTrace contains over 30,000 lines of code in C, C++, and assembler.

How did we do it? We set up a debugging environment and wrote what we call an initialization driver (which we'll explain later) for VTrace and a logger driver that time-stamped and logged all events. Then we modified the Windows keyboard filter to log key presses, Russinovich and Cogswell's Filemon filter driver (from their SysInternals Web site at http://www.sysinternals.com) to monitor file system activity, and a physical disk filter to log physical disk activity, in addition to writing a network filter driver. Finally, we wrote code to log context switches, to log system calls to the Win32® subsystem and the kernel, to monitor the file system, and to log the beginning and end of all processes and threads.

We developed VTrace for Windows NT and later ported it to Windows 2000. We'll explain the issues we encountered in that exercise, and also discuss our performance benchmarks.

Creating a Debugging Environment

Because a tracer contains and interacts with a lot of code that runs in kernel mode, we needed a kernel-mode debugger. Since much of the code in a tracer is executed before the system has completely started up, it runs before a debugger program can be launched. To address this we used a two-system debugging environment: the debugger runs on the host machine (the development machine), and the software that is being tested runs on the target machine. The debugger monitors and controls the target machine through a serial cable connecting the two machines. This setup was useful for another reason: if we did something that led to a reboot or reinstallation of the operating system, the development environment was unaffected.

Unfortunately, setting up kernel debugging with the DDK debugger, WinDbg, is notoriously difficult. Some of the most difficult tasks include configuring the debugger program settings and making the target machine communicate with a remote debugger. Because the documentation included with the debugger is insufficient for these tasks, we checked the Web and Usenet for answers. For example, when we found that Windows 2000 would inexplicably hang while starting up with the debugger, a Usenet post had the solution: use the Break command in WinDbg's Debug menu.

Once we had the debugger for Windows NT set up, it worked very well, enabling us to easily set breakpoints in source code, step through source code, examine and change runtime variable values, and even view operating system code (only in uncommented assembler, of course). Be warned, though: we've found early versions of WinDbg for Windows 2000 to be buggy.

Filter Drivers

VTrace uses filter drivers. A filter driver implements a filter device, which is extremely helpful in tracing system events in Windows NT and Windows 2000. (For more information on drivers, see the sidebar "Drivers for Windows NT and Windows 2000.") After a filter device attaches to an existing device, it starts intercepting any requests sent to that device. Typically, it modifies the request in some way, and then passes it on to the device. This lets it add functionality to the device; for instance, it could turn a traditional file system into an encrypted file system. A filter can also simply record information about requests, pass those requests unchanged to the device they were meant for, then record information about the results of those requests. This was the primary manner in which we used filters.

Figure 1 Driver Initialization Routine

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath)
{
  NTSTATUS status;
  int i;

  // Read driver-shared state from initialization driver.

  status = GetSharedState(&sharedState);
  if (!NT_SUCCESS(status))
    return status;

  // Create dispatch points for all routines that must be handled.
  // All entry points are registered since we might filter a
  // file system that processes all of them.

  for (i = IRP_MJ_CREATE; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    DriverObject->MajorFunction[i] = VTrcFSPassOnNormally;
    DriverObject->MajorFunction[IRP_MJ_CREATE] = VTrcFSDispatchCreate;
    DriverObject->MajorFunction[IRP_MJ_READ] = VTrcFSDispatchReadWrite;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = VTrcFSDispatchReadWrite;

  // Set up the fast I/O dispatch table. (See the section on fast I/O
  // for details.)

    DriverObject->FastIoDispatch = &VTrcFSFastIoDispatchTable;

  // Note: It would be unwise to unload this driver, so we don't set an
  // unload routine. Otherwise, we would set DriverObject->DriverUnload.

  // Normally there would be code here to attach to some other device
  // or devices, but in VTrace we do this elsewhere.

  return STATUS_SUCCESS;
}
```

Figure 1 shows the driver initialization routine that VTrace uses to initialize the file system driver. This routine sets the MajorFunction entries in the driver object so the appropriate dispatch routine gets called for each request type.

The heart of VTrace is the logger, the piece of code that collects trace records and writes them to the trace file. It accepts and serializes requests to add events to the in-memory log, and periodically writes the log to disk.

Figure 2 shows the dispatch routine that VTrace uses to handle read and write requests. This routine logs the request initiation, sets a completion routine to be called when the request completes, then calls the lower-level driver to complete the request.

Figure 3 shows the completion routine that VTrace uses to log the results of a file system request that just completed.

Initialization Driver

Many of VTrace's drivers have to start at boot time in order to work. If they cause problems, there are several things that can remove them: a recent emergency repair disk, the recovery console

feature of Windows 2000, the NTFSDOS product sold at the SysInternals Web site, and, as a last resort, reinstalling the operating system. Unfortunately, these can be time-consuming. Therefore, the ability to disable the drivers at startup is quite useful. To achieve this, VTrace includes an initialization driver that starts before all of its other components. This driver decides whether VTrace should be disabled for this boot. When they start up, each of the other drivers first sends the initialization driver's device a special I/O control request. If VTrace is disabled, the device returns null; otherwise, it returns a pointer to a structure it allocated from non-paged memory for all the VTrace drivers to share.

To decide if it should disable VTrace, the initialization driver needs user-provided information to be available early in the boot process. About the only state information the user can provide at that time is which boot configuration to use. For instance, the user can choose to use the "last known good" configuration. So, the initialization driver opens the registry key HKEY_LOCAL_MACHINE\System\Select and reads the Current and LastKnownGood values. Each of these is an index into the list of registry configurations. If they are the same, it means that the user chose the last known good configuration, so the driver disables VTrace.

Figure 2 Dispatch Routine

```
NTSTATUS VTrcFSDispatchReadWrite (PDEVICE_OBJECT FilterDevice,
                                  IN PIRP Irp)
{
  PIO_STACK_LOCATION  currentIrpStack =
      IoGetCurrentIrpStackLocation(Irp);
  PIO_STACK_LOCATION  nextIrpStack    = IoGetNextIrpStackLocation(Irp);
  PFILE_OBJECT        fileObject      = currentIrpStack->FileObject;
  PFS_HOOK_EXTENSION  filterExtension = FilterDevice->DeviceExtension;
  PDEVICE_OBJECT      nextDevice      = filterExtension->attachedDevice;
  PCHAR               eventPosInLog   = NULL;
  ULONG               seq;
  KIRQL               oldirql;

  // If the file has a name, log the read or write request.

  if (fileObject->FileName.Buffer) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      (currentIrpStack->MajorFunction == IRP_MJ_READ ?
       ENTRY_TYPE_FILE_READ : ENTRY_TYPE_FILE_WRITE), 24);
    if (eventPosInLog) {
      seq = InterlockedIncrement(&globalSequenceNumber);
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &fileObject, 4);
      RtlCopyMemory(&eventPosInLog[9],
                    &currentIrpStack->Parameters.Read.ByteOffset, 5);
      RtlCopyMemory(&eventPosInLog[14],
                    &currentIrpStack->Parameters.Read.Length, 4);
      RtlCopyMemory(&eventPosInLog[18], &Irp->Flags, 4);
      eventPosInLog[22] = currentIrpStack->MinorFunction;
      eventPosInLog[23] = currentIrpStack->Flags;
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  if (!eventPosInLog) {
    // If we didn't enter the request into the log, we just pass this
    // IRP on normally to the next lower device. We do this by backing
    // up the IRP stack location to reuse the current location for the
    // next lower device. In Windows 2000, use the
    // IoSkipCurrentIrpStackLocation macro for the following.

    Irp->CurrentLocation++;
    Irp->Tail.Overlay.CurrentStackLocation++;
  }
  else {
    // Copy parameters to the next position in the stack for the next
    // lower device. In Windows 2000, use the
    // IoCopyCurrentIrpStackLocationToNext macro for the following.

    *nextIrpStack = *currentIrpStack;

    // Set a completion routine, passing the sequence number as the
    // "context" parameter so that it can be used in the completion
    // log entry.

    IoSetCompletionRoutine(Irp, VTrcFSCompletionRoutine, (PVOID) seq,
                           TRUE, TRUE, TRUE);
  }

  // Pass the IRP on to the lower-level device.

  return IoCallDriver(nextDevice, Irp);
}
```

Figure 3 Completion Routine

```
NTSTATUS VTrcFSCompletionRoutine (PDEVICE_OBJECT DeviceObject, PIRP Irp,
                                  PVOID Context)
{
  ULONG seq = (ULONG) Context;
  KIRQL oldirql;
  PCHAR eventPosInLog;

  // Log the return values.

  KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
  eventPosInLog = (sharedState->logEventFunctionPointer)(
    ENTRY_TYPE_FILE_COMPLETE_OPERATION, 13);
  if (eventPosInLog) {
    RtlCopyMemory(&eventPosInLog[1], &seq, 4);
    RtlCopyMemory(&eventPosInLog[5], &Irp->IoStatus.Status, 4);
    RtlCopyMemory(&eventPosInLog[9], &Irp->IoStatus.Information, 4);
  }
  KeReleaseSpinLock(&sharedState->mainMutex, oldirql);

  // Always do the following in a completion routine. (By the way, the
  // braces are necessary since IoMarkIrpPending is a macro that
  // expands to multiple statements.)

  if (Irp->PendingReturned) {
    IoMarkIrpPending(Irp);
  }

  return Irp->IoStatus.Status;
}
```

Figure 4 Calling a Driver from User Mode

```
void LogEvent (char *eventDescription, DWORD descriptionLength)
{
  DWORD returnSize;

  HANDLE hDevice = CreateFile("\\\\.\\VTrcLog",
                              GENERIC_READ | GENERIC_WRITE, 0,
                              NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                              NULL);

  if (hDevice != INVALID_HANDLE_VALUE) {
    DeviceIoControl(hDevice,          // Handle to device
                    VTRACE_LOG_EVENT, // Device I/O control code
                    eventDescription, // Outbound communication buffer
                    descriptionLength, // Length of outbound buffer
                    NULL,             // Inbound communication buffer
                    0,              // Length of inbound buffer
                    &returnSize,    // Bytes returned in inbound buffer
                    NULL);          // Unused (for async communication)
    CloseHandle(hDevice);
  }
}
```

Logger Driver

The heart of VTrace is the logger, the piece of code that collects trace records and writes them to the trace file. It accepts and serializes requests to add events to the in-memory log, and periodically writes the log to disk. We implemented the logger as a device so its code could run in kernel mode. This lets other kernel-mode code (most of the tracer) call it efficiently. It also lets user-mode code call it without a context switch. (A kernel trap is still needed, though.) The logger driver implements several specialized device control I/O request types, including some to start logging, stop logging, add an event to the log, and flush the log to disk. User-mode code makes these requests using code like that shown in **Figure 4**. This function, which logs an event, illustrates how user-mode code communicates with a kernel-mode driver. Kernel-mode code communicates with the logger driver more efficiently by calling functions in the driver directly. The logger driver puts pointers to these functions in the memory region shared by VTrace's drivers.

Filters

To log key presses, we made simple modifications to the keyboard filter driver, Ctrl2cap, whose code is available from the SysInternals Web site. Its original purpose was to make the Caps Lock key function as an extra Control key. We just made it encrypt and log the key presses instead of modifying them.

To log file system activity, we made a few modifications to Filemon, another filter driver whose source code can be found on the SysInternals Web site. This filter driver logs and displays file system activity.

Our most important modification changed the way in which the code selects devices to filter. Filemon allows the user to choose specific file systems, but we wanted to filter all file systems. We couldn't simply find all the file systems at system startup and filter those, because some—like floppy disks and CDs—may be added dynamically. Instead, we hook the Windows NT or Windows 2000 system call that opens files (discussed in more detail later), check in that hook whether we've already filtered the file system containing that file, and start filtering it if we haven't. To determine whether we're already filtering the file system containing that file, the filter increments a counter whenever it filters a file open request. So, if the counter remains unchanged throughout an open system call, we know we haven't filtered that file system. Keep in mind, however, that some file open calls go directly to the disk device and skip

Drivers for Windows NT and Windows 2000

In Windows NT and Windows 2000, a device is a software object that can receive I/O requests, representing things like a disk drive, a file system, or a keyboard. Devices can be layered, meaning that the top-level device processes its I/O requests by sending I/O requests to the lower-level device. For example, a device for a file system will typically be layered over a physical disk device so that file requests can be translated into disk requests. Some devices create file objects, which are pieces of data with state carried over between I/O requests. Despite the name, file objects can represent more than just open files—they can be network connection endpoints, open directories, and so on.

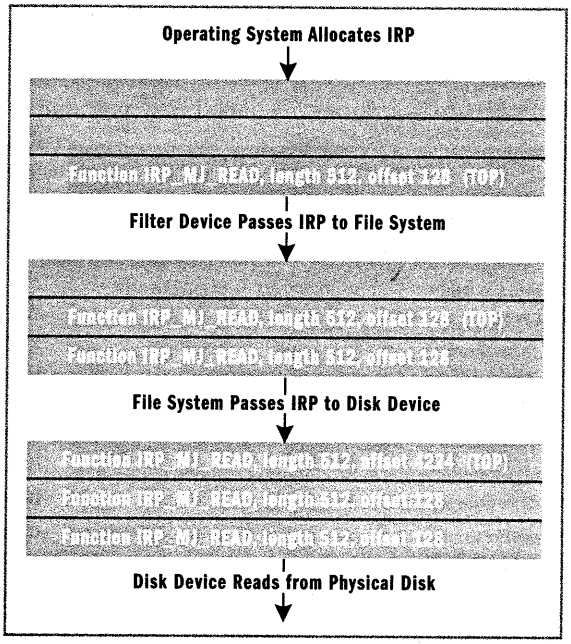

Figure A How Layered Devices Use an IRP Stack

The way in which I/O requests are handled allows device layering to be accomplished. A device receives a structure called an I/O request packet (IRP), which represents an I/O request and contains a stack of I/O stack locations. (See **Figure A** for an example of how layered devices use an IRP stack.) This stack's top location contains a description of the request in a form that the device understands. Before the device passes the I/O request to a lower-level device, it pushes onto the stack a new stack location describing the I/O request in a form that the lower-level device understands. When that device finishes processing the IRP, the extra stack location is popped from the stack. Thus, when the top-level device gets the IRP back to complete its own processing, the top location of the stack is still the one relevant to that device. The most important fields of an I/O stack location are the major function code, describing the general request type; the minor function code, describing the request type more specifically; and the 16-byte parameters field, whose meaning depends on the function codes.

A driver is the code implementing a class of devices. For instance, all NTFS file system devices use the NTFS driver code. This code includes a driver entry routine and several dispatch routines. The driver entry routine performs per-driver initialization, including entering the dispatch routines into the dispatch routine table. The operating system uses the dispatch routine table, indexed by major function code, to determine which routine handles a given IRP.

Figure 5 Fast I/O Routine

```
BOOLEAN MyFastIoRead (PFILE_OBJECT FileObject, PLARGE_INTEGER FileOffset,
                      ULONG Length, BOOLEAN Wait, ULONG LockKey,
                      PVOID Buffer, PIO_STATUS_BLOCK IoStatus,
                      PDEVICE_OBJECT DeviceObject)

{
  PFS_HOOK_EXTENSION filterExtension = DeviceObject->DeviceExtension;
  PDEVICE_OBJECT nextDevice          = filterExtension->attachedDevice;
  PFAST_IO_DISPATCH nextFastIoTable  = nextDevice->DriverObject->
                                       FastIoDispatch;
                                     = NULL;

  PCHAR eventPosInLog
  BOOLEAN retval;
  ULONG seq;
  KIRQL oldirql;

  // If the next lower driver has no fast I/O routine, return FALSE.

  if ((ULONG) &nextFastIoTable->FastIoRead - (ULONG) &nextFastIoTable >=
      nextFastIoTable->SizeOfFastIoDispatch ||
      nextFastIoTable->FastIoRead == NULL)
    return FALSE;

  // If there is a file name, record this call.

  if (FileObject->FileName.Buffer != NULL) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      ENTRY_TYPE_FILE_READ, 24);
    if (eventPosInLog) {
      seq = InterlockedIncrement(&globalSequenceNumber);
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &FileObject, 4);
      RtlCopyMemory(&eventPosInLog[9], FileOffset, 5);
      RtlCopyMemory(&eventPosInLog[14], &Length, 4);
      RtlZeroMemory(&eventPosInLog[18], 4);      // no IRP flags
      eventPosInLog[22] = IRP_MN_NORMAL;
      eventPosInLog[23] = '\0';                  // no stack flags
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  // Call the real fast I/O routine, recording the return value

  retval = nextFastIoTable->FastIoRead(FileObject, FileOffset, Length,
                                       Wait, LockKey, Buffer, IoStatus,
                                       nextDevice);

  // If the call completed successfully, and we logged the call, log
  // the return.

  if (retval && eventPosInLog) {
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventPosInLog = (sharedState->logEventFunctionPointer)(
      ENTRY_TYPE_FILE_COMPLETE_OPERATION, 13);
    if (eventPosInLog) {
      RtlCopyMemory(&eventPosInLog[1], &seq, 4);
      RtlCopyMemory(&eventPosInLog[5], &IoStatus->Status, 4);
      RtlCopyMemory(&eventPosInLog[9], &IoStatus->Information, 4);
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  // Return the real routine's return value.

  return retval;
}
```

the file system altogether, so the counter will be unchanged during that call, even if the associated file system is already filtered. Fortunately, all such calls reference a file with no name, so by ignoring calls that reference a file without a name we found that we could avoid this case.

Another way to find all file systems is to call IoRegisterFsRegistrationChange, part of the Microsoft Installable File System (IFS) Kit. This call registers a function to be called whenever a file system becomes active or inactive.

Our method has two advantages over the IFS method. First, it allows notification that network file systems have become active. Second, since you open and create network objects using the same system call used for file objects, you can easily extend our method to allow notification when a network device becomes active. VTrace uses this method to ensure its network filter (described shortly) hooks all the network transport devices in the system.

The fast I/O path optimization in Windows NT and Windows 2000 can make filtering file system devices a bit tricky. If a file system device can handle a request without involving a lower-level device (such as during a cache hit) it's a waste to create an IRP. A file system driver can specify a table of fast dispatch routines (one for each I/O request type) that can handle requests not packaged in IRPs. If the fast

Figure 6 TDI Internal Device Control Requests

Function	Description
TDI_ASSOCIATE_ADDRESS	Associates a connection endpoint with a network address
TDI_DISASSOCIATE_ADDRESS	Disassociates a connection endpoint from the network address it was previously associated with
TDI_CONNECT	Establishes a connection between a local connection endpoint and a specified remote address
TDI_LISTEN	Listens for requests from any of a set of remote addresses to a local connection endpoint
TDI_ACCEPT	Accepts a connection request made by a remote address to a local connection endpoint
TDI_DISCONNECT	Terminates the connection in which a connection endpoint is participating
TDI_SEND	Sends an ordered packet over a connection
TDI_RECEIVE	Receives an ordered packet over a connection
TDI_SEND_DATAGRAM	Sends a datagram over a connection
TDI_RECEIVE_DATAGRAM	Receives a datagram over a connection
TDI_SET_EVENT_HANDLER	Establishes a routine for handling a certain type of event, such as the arrival of a datagram
TDI_QUERY_INFORMATION	Gets information about some network object, such as its network address
TDI_SET_INFORMATION	Sets information about some network object
TDI_ACTION	Performs some transport-specific action

dispatch routine can't handle the request, such as when it misses the cache, it returns an error value, forcing the operating system to send an IRP to the regular dispatch routine. To filter accesses that use fast I/O, we had to make a set of fast dispatch routines for our filter driver. These routines log each time they get called, and pass calls on to the fast dispatch routines of the lower-level driver. **Figure 5** shows the fast I/O routine that VTrace uses to handle fast-path read requests. Also, the sample driver initialization code shown in **Figure 1** includes a command to set up the fast I/O dispatch table.

To log activity at the physical disk level, we modified a physical disk filter driver, DiskPerf, whose source code is in the DDK. This driver collects and reports statistics about raw disk access, so it was easy to retool it for our purposes.

Network Filter

Unlike the other filter drivers we needed, we found no source code for a network transport layer filter driver. This meant we had to write one essentially from scratch.

In Windows NT and Windows 2000, all transport protocols must use the same programming interface: the transport driver interface (TDI). **Figure 6** shows the minor function codes of some useful TDI internal device control requests (from the Windows NT DDK help). I/O requests passed to the transport layer all conform to the single format described in the DDK help and the DDK files TDI.H and TDIKRNL.H. It seems this would make our job easy, but building a filter for these requests still presented us with a challenge.

One problem we encountered is that some IRPs have the major function code "device control," but we couldn't find anything describing their parameter format. However, we learned from the DDK help that the first thing a device does when it receives such a request is call the function TdiMapUserRequest to convert it to one with a major function code of "internal device control" (which we know how to interpret). So, in our filter driver dispatch routine for device control requests, we first call TdiMapUserRequest.

Another problem is an apparent bug in the way Windows NT handles network filter devices. When Windows NT constructs an IRP, it must allocate enough stack space in it to account for the maximum depth of the device stack that the IRP will pass through. To allow this, each device object has a stack count field indicating how large the stack must be in IRPs it receives. Unfortunately, Windows NT sometimes brazenly ignores the stack count field in our filter device objects and sends it an IRP with insufficient stack space. If we push a new location onto this stack and pass it on, eventually the stack overflows and we see the blue screen. We solve the stack problem by checking for insufficient stack space, and creating a new IRP to pass to the lower-level driver when needed.

Yet another problem is that not all network I/O uses IRPs. If I/O must happen in response to some event, such as a datagram arrival, the system invokes an event handler function rather than a dispatch routine. This is unfortunate, since while Windows NT and Windows 2000 provide the elegant, well-supported filter driver approach for intercepting IRPs sent to dispatch routines, it gives no such help in intercepting calls to event handlers.

Figure 7 Hooking the Context Switch Routine

```
// Before the context swap hook is in place, SwapContext looks like:
//     SwapContext:
//       mov byte ptr es:[esi+2Dh],2
//       or cl,cl
//       mov ecx,dword ptr [ebx]
//       pushfd
//       ... etc. ...
// After the context swap hook is in place, SwapContext looks like:
//     SwapContext:
//       jmp <the address 6 bytes into NewSwapContext>
//       or cl,cl
//       mov ecx,dword ptr [ebx]
//       pushfd
//       ... etc. ...
// (The 6-byte offset is to skip over the compiler-inserted stack set-up
// stuff at the beginning of NewSwapContext.)
// NewSwapContext logs the context switch, executes the overwritten
// instruction from the old context swap routine, then jumps to the point
// in the old swap routine past that instruction.

void FASTCALL NewSwapContext (void)
{
  __asm {
    // Save registers we may overwrite.
    push eax
    push ecx
    // Save interrupt mask and stop all interrupts. Since context-swaps
    // can't occur on a uniprocessor while a spinlock is held, we know no
    // one else has the spinlock.
    pushfd
    cli
    // eventPosInLog = LogEvent(ENTRY_TYPE_THREAD_SWITCH [0xE], 5);
    push 5
    push 0Eh
    call LogEvent
    // if (eventPosInLog)
    cmp eax, 0
    je DoneLogging
    // Save eventPosInLog on stack for later use.
    push eax
    //   * (DWORD *) &eventPosInLog[1] = PsGetCurrentThreadId();
    call PsGetCurrentThreadId
    pop ecx    // Pop eventPosInLog from stack to use now.
    mov dword ptr [ecx+1], eax
  DoneLogging:
    // Restore the interrupt mask.
    popfd
    // Restore saved registers.
    pop ecx
    pop eax
    // Execute overwritten instruction from original swap routine.
    mov byte ptr es:[esi+2dh],2
    // Jump to point in original swap routine past overwritten part.
    jmp dword ptr globals.nonOverwrittenPartOfOrigSwapRoutine
  }
}
```

We overcame this with our own technique for intercepting calls to event handlers. The key is our ability—thanks to filter devices—to intercept and change any request that sets a new event handler for a file object. (These are the requests with minor function code TDI_SET_HANDLER.) Each of these requests contains the location of the event-handling function, the type of event it handles, and a four-byte context value to pass to that function. All the driver must do, then, is allocate a structure to store this information, then modify the request so that instead of containing the location of the real event-handling function and the real four-byte context value, it contains the location of a special logging event-handling function and the four-byte address of the allocated structure. So whenever an event of the given type happens, our special

logging event-handling function is called and passed the address of the structure we allocated. This function logs the event, and then inspects the structure in order to call the appropriate event-handling function with the appropriate context value. When that function returns, our special logging function can trace its return value. (We later refined this approach so the driver allocates a single structure per file object, not per event handler, so it can quickly free all the memory allocated for a file object when it closes.)

Logging Context Switches

Logging context switches should be easy, since kernel-mode software can use KeSetSwapContextNotifyRoutine to make the system call a given function whenever it switches contexts. Unfortunately, this call only works on the multiprocessor and checked-build versions of Windows NT and Windows 2000. Few people use these versions, and because we wanted our tracer to run on any machine, we had to design a method for doing this on the uniprocessor free build.

For performance reasons, the user-level DLL tracing code shares a region of memory with VTrace's drivers. It obtains a pointer to this region by sending a request to the initialization device.

We started by finding the assembler code for SwapContext in Windows NT using WinDbg. We found that the first five bytes of this function are a single instruction, so we can overwrite these bytes in memory with a jump to our own NewSwapContext function, shown in

Figure 7. NewSwapContext logs the context switch and the thread being switched to, executes the first five bytes of the original pre-overwrite version of SwapContext, then jumps to the sixth byte of SwapContext. Unfortunately, finding the location of SwapContext in memory is not straightforward. We know it's always in the in-memory image of the kernel executable, NTOSKRNL.EXE, which is loaded at address 0x80100000. However, its position within NTOSKRNL.EXE changes from version to version of Windows NT 4.0. For instance, in the original Windows NT 4.0 it's at 0x8013F4F0, but after applying Service Pack 3 it's at 0x80140CA0, and after applying Service Pack 6a it's at 0x80142420. The good news is that in all of these versions—and all other versions of Windows NT 4.0 that we've seen—the instructions of this function are unchanged. So to find SwapContext, we just searched for the known first 28 bytes of the routine in the memory section where we expected it. Of course, we check a few common locations first, since the routine is most likely to be in one of them. Doing this check is dangerous because if kernel-mode software accesses an invalid (paged out, say) memory location, the system will crash. So before checking any location, we first call MmIsAddressValid to make sure we can read it.

Logging Win32 System Calls

Windows NT and Windows 2000 support multiple user-level subsystems, such as Win32, POSIX, and OS/2. So, the term "system call" is vague; it could mean a call to the Win32 subsystem, to some other subsystem such as OS/2, or to the Windows NT or Windows 2000 kernel itself. Now we'll look at how to log system calls to the Win32 subsystem.

Our technique for logging Win32 system calls borrows heavily from the technique Matt Pietrek used for APISPY32 in his article "Learn System-Level Win32 Coding Techniques by Writing an API Spy Program" (*MSJ*, December 1994). This technique basically works as follows. An application makes a Win32 system call by making an indirect call through one of an array of function pointers. (**Figure 8** illustrates the way Win32 system calls are performed.) We just need to find that array of function pointers (which is easy to do once the image and file format is understood), and replace the pointers to functions we want to log with pointers to our own logging functions. These logging functions, which reside in a special DLL that's part of the tracer software, will call the original functions and log those calls.

This method requires that each application load this special DLL into its address space. In his article, Matt Pietrek provides several ways to ensure this. We chose the simplest of them: putting the name of the DLL in the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs. This doesn't take effect until you reboot, but our raw disk filter driver needs a reboot anyway. Also, this technique only loads the DLL into applications that load USER32.DLL; this was fine for us because we only needed to log functions in USER32.DLL.

Matt designed APISPY32 for Windows NT 3.5. Some Usenet messages report it can't be used with Windows NT 4.0 because virtual memory protections prevent the replacement of some of the function pointers. To fix this problem, we call VirtualProtect to temporarily change those protections. Another problem with APISPY32 is that it only hooks system calls made directly by the application. If an ap-

Figure 8 How Win32 System Calls are Performed

Code	Description
`call dword ptr __imp__GetMessageA@16`	Application code. The application calls GetMessageA, which is compiled as an indirect call through __imp__GetMessageA@16.

`__imp__PostThreadMessageA@16: 0x10001E50`
`__imp__GetMessageA@16: 0x10001410`
`__imp__PeekMessageA@20: 0x10001550`
	Array of function pointers. The location __imp__GetMessageA@16 is in an array of imported function pointers located in the import data section.

`0x10001410:`
`sub esp, 18h`
`push ebx`
	Function body. The actual body of the function GetMessageA is at the specified memory location 0x10001410. This location is part of the memory image of the USER32 DLL.

plication calls a DLL function, which in turn makes a system call, APISPY32 doesn't notice that system call. This is because APISPY32 performs its function interception on the application executable image, but not on the image of any DLL.

Our solution to this has two parts. First, when an application loads the logging DLL, the logging DLL calls EnumerateLoadedModules to get the memory locations of the application and all its loaded DLLs. It then intercepts functions in all those modules. Second, it intercepts the LoadLibrary functions (even though we don't need to log them), so that when one completes we can call EnumerateLoadedModules to intercept functions in all the newly loaded library images. Note that it isn't sufficient to consider only the library mentioned in the LoadLibrary call, since that library may have implicitly or explicitly caused other libraries to be loaded. Also, be aware that this implementation will miss logging some calls that libraries make when they initialize.

An important part of parsing a PE format file is translating virtual addresses into file positions.

Debugging a logging DLL can be difficult, since any bug can make the system unusable. For instance, it could make the logon screen fail. If this happens, the only recourse may be to restore the registry to a previous state without the DLL in the AppInit_DLLs list, or to delete the offending DLL file. Each of these approaches is annoying and time-consuming. A good solution, suggested in a Usenet post, is to put the DLL on a floppy disk and tell AppInit_DLLs to get it from there. That way, if there's a bug, you can just remove the floppy so no app can load the DLL. For VTrace, we have the DLL check with the initialization device to see if VTrace is disabled for this boot.

For performance reasons, the user-level DLL tracing code shares a region of memory with VTrace's drivers. It obtains a pointer to this region by sending a request to the initialization device. To satisfy this request, the initialization device must construct a pointer that is valid in the user-mode process calling it, using code such as that shown in **Figure 9**. One important issue is that the device must unmap this address before the process exits, or the system will crash. Fortunately, in order to access the device to request the mapping, the user process must create a "file" representing a link to the driver. When that process is about to terminate, it automatically closes this file. VTrace stores the user-level address in the corresponding file object, and unmaps the address when it receives a close request for the file object.

Logging Kernel System Calls

We also wanted to log kernel system calls; that is, system calls made via a transition from user mode to kernel mode. Our approach comes from the Regmon application, available from the SysInternals Web site. The idea is to find the service table list (an in-memory array of system call function pointers indexed by system call number), and replace the function pointers with pointers to special logging functions. The trickiest part is determining which system call number corresponds to each system call.

Figure 9 Mapping Kernel Memory to User Level

```
PVOID GetUserLevelAddress (PVOID kernelLevelAddress, ULONG length,
                           FILE_OBJECT *fileObject)

{
  PVOID address;
  PMDL mdl;

  mdl = IoAllocateMdl(kernelLevelAddress, length, FALSE, FALSE, NULL);
  if (mdl == NULL)
    return NULL;

  // Build the MDL for the kernel-level address, assumed to lie in
  // non-paged memory. Then, map it into a user-level address.

  MmBuildMdlForNonPagedPool(mdl);
  address = MmMapLockedPages(mdl, UserMode);
  if (address == NULL) {
    IoFreeMdl(mdl);
    return NULL;
  }

  // Save the address and MDL pointer so they can be unmapped and
  // freed, respectively, when this file object is closed.

  fileObject->FsContext = address;
  fileObject->FsContext2 = mdl;

  // In Windows NT 4.0 SP3 and earlier, 'address' refers to the base
  // virtual address of the page instead of the actual virtual address of
  // the MDL. So, we use the following code, which will work whether or
  // not it's one of those versions.

  return (PVOID) ((ULONG)PAGE_ALIGN(address) + MmGetMdlByteOffset(mdl));
}
```

User-level code makes a kernel system call by putting the system call number in EAX, putting parameters in other registers, then executing the INT 2E instruction. An application usually does this indirectly. For example, a call to WaitForSingleObject in KERNEL32.DLL will eventually call NtWaitForSingleObject in NTDLL.DLL, which performs the register manipulation and the call to INT 2E. Kernel-mode code usually does it a little differently, by calling routines with the prefix Zw exported by NTOSKRNL.EXE. Each Zw routine, such as ZwWaitForSingleObject, handles the register manipulation and the call to INT 2E.

Regmon's authors noted that the first thing these Zw functions do is load the system call number into EAX. Thus kernel-mode code can find the system call number in bytes 2-5 of each Zw function. One caveat is that in a debug version of the driver, each Zw function is only a wrapper that calls the "real" Zw function. It turned out that we didn't have to worry about this because, for reasons we'll now explain, we wound up not reading the Zw functions from the memory image of our running driver.

We found that, unfortunately, NTOSKRNL.EXE doesn't export all the system calls we wanted to log. Some, such as ZwSignalAndWaitForSingleObject, are only exported by NTDLL.DLL, but we couldn't link our driver with NTDLL.DLL. (Regmon doesn't have this problem since it only hooks calls exported by NTOSKRNL.EXE.) So we had our tracer find the Zw function bodies by reading and parsing the NTDLL.DLL disk file. Our technique is based on an understanding of the well-documented portable executable (PE) file format.

An important part of parsing a PE format file is translating

virtual addresses into file positions. Many structures in the file refer to other structures in the file using their relative virtual addresses (RVAs). A structure's RVA describes where it will be in memory when the file is mapped. Its virtual address will be its RVA plus the base address where the file is mapped in memory. The problem is that we need to know where in the file those structures are. To translate from RVAs to file positions, we need the section header information. This is an array of IMAGE_SECTION_HEADER structures, each of which has the absolute file position, length, and RVA of a section. Using this information, we can figure out which section contains a given RVA, and from that we can determine the file position for that address. **Figure 10** shows how to find these section header structures in the file.

Once we can translate from RVAs to file positions, we can find the names and bodies of all the exported functions using the information in **Figure 10**. This lets us find where the Zw function bodies are and the contents of their first few bytes.

As mentioned earlier, we hook the system calls for opening files so our file system filter driver can attach a device to each file system. Unfortunately, the DDK doesn't document one of these system calls, ZwOpenFile. The book *Windows NT File System Internals*, by Rajeev Nagar (O'Reilly & Associates, 1997) does document this function, so we were able to use its parameters to determine which file system to filter.

Parsing File System Metadata

We also wanted to take periodic snapshots of each local NTFS partition, so that for every file we would know its name, size, attributes, and its physical location on the disk. For this we found valuable NTFS driver documentation and source code at http://www.informatik.hu-berlin.de/~loewis/ntfs.

We found that just about all the data we need is in a special file in each partition called the Master File Table (MFT). This file, named $MFT, contains fixed-length records describing the attributes of each file (and directory, since directories are basically just special files). However, we can't just dump this file, for at least three reasons. First, the file is sparse: many files' attributes don't use an entire record and many records are unused because they correspond to deleted files. Second, an attribute can be nonresident, meaning that it's somewhere else on disk and only a pointer to it is in the MFT record. Third, the contents of a file are considered an attribute of the file, so recording the MFT would record file contents and violate the confidentiality of our users' data.

So, instead, we do a depth-first search of the directory structure of each NTFS partition and, for each file, find and record certain non-data attributes of that file. Finding the metadata for a file requires knowing its file number, which is the index of the MFT record containing that file's attributes. The partition root always has file number 5.

We still haven't explained how you read directly from a disk, or how you find specific MFT records. To read a raw disk on Windows NT and Windows 2000, a user-mode program can open a file called \\.*X*:, where *X* is the appropriate drive letter. The first file block contains useful information: the size of a block (the 2-byte

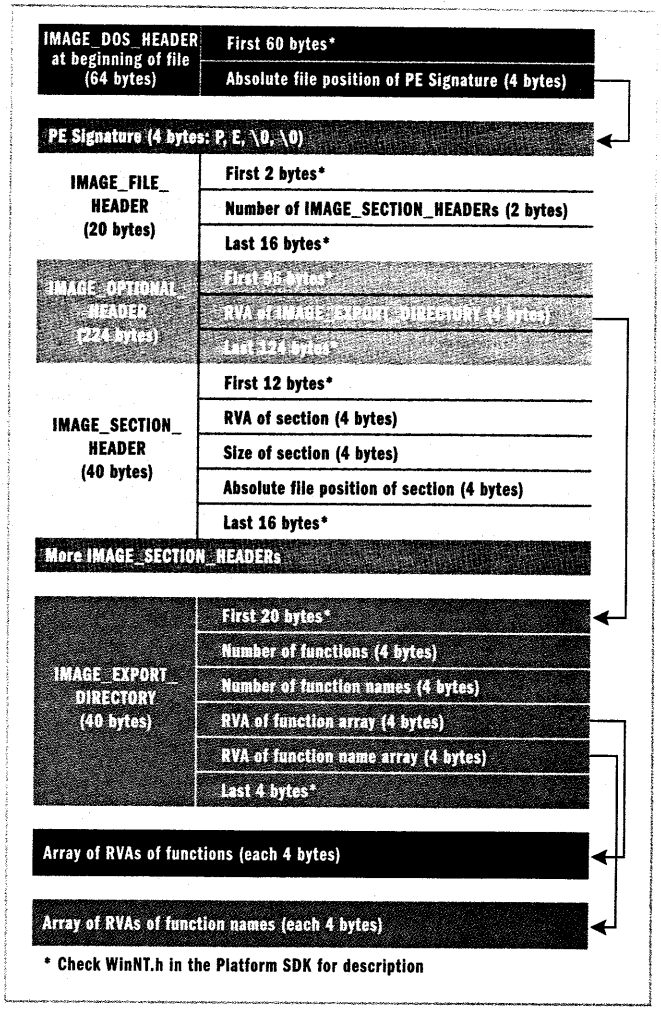

Figure 10 PE File Format for Executable Files

value at offset 0xB), the number of blocks in a cluster (the 1-byte value at offset 0xD), the number of clusters in an MFT record (the 1-byte value at offset 0x40), and the cluster number of the first MFT record (the 8-byte value at offset 0x30). The first MFT record is useful to find, since it contains the file attributes for $MFT itself. By parsing its data attribute information you can locate any MFT record. Then, parsing a file's MFT record reveals the entire file's attributes. (This is easy if you read the NTFS documentation described earlier and judiciously inspect sections of the Linux NTFS driver code.) If the file is actually a directory, you can parse its index allocation attribute to find the file numbers of its contents.

Listing Processes and Threads

To log when processes and threads start and stop, we use the barely documented functions PsSetCreateProcessNotifyRoutine and PsSetCreateThreadNotifyRoutine. With them, we can have the system call a given logging function when a process (or thread) is created or destroyed.

We also need to record a list of the existing processes and threads when the tracer starts logging. Unfortunately, there's no documented way to do this from kernel mode. Fortunately, we found a

Usenet message describing how do this with the undocumented function ZwQuerySystemInformation. The function has the prototype shown here:

```
unsigned long ZwQuerySystemInformation
 (ULONG tag, VOID *buffer, ULONG bufferSize, ULONG *returnedSize);
```

The tag parameter in this prototype indicates what kind of information is to be returned; the value 5, for instance, indicates process and thread information. **Figure 11** shows how we use this function to get a sequence of process information structures, one for each process. **Figure 12** illustrates what you can find in each of these structures (at least, in the current uniprocessor build of Windows NT). We also use this routine to obtain and log the name of a process when it starts, since the notification only tells us the process ID.

Figure 11 Getting Process and Thread Information

```
#define TAG_GET_PROC_THREAD_INFO              5
#define FIRST_GUESS_AT_PROC_THREAD_INFO_SIZE  8192
#define INCREMENT_FOR_PROC_THREAD_INFO_SIZE   1024

// GetProcessAndThreadInfo() returns a pointer to an allocated buffer
// containing process and thread information. *bytesReturnedPtr will
// hold the useful length of this information. If an error occurs,
// this function returns NULL. Otherwise, the caller is expected to
// eventually call ExFreePool() to deallocate the returned buffer.

char *GetProcessAndThreadInfo (ULONG *bytesReturnedPtr)
{
  char *buf;          // buffer to hold the process and thread information
  ULONG bufSize;      // size of the buffer
  NTSTATUS status;    // status code returned by ZwQuerySystemInformation

  bufSize = FIRST_GUESS_AT_PROC_THREAD_INFO_SIZE;
  while ((buf = ExAllocatePool(NonPagedPool, bufSize)) != NULL) {
    *bytesReturnedPtr = 0;
    status = ZwQuerySystemInformation(TAG_GET_PROC_THREAD_INFO,
                                      buf, bufSize, bytesReturnedPtr);
    if (status == STATUS_SUCCESS) return buf;

    // If the buffer was the wrong size, make the buffer bigger; use the
    // value returned in bytesReturnedPtr as a hint about the needed size.

    ExFreePool(buf);
    if (status == STATUS_BUFFER_OVERFLOW ||
        status == STATUS_INFO_LENGTH_MISMATCH)
      bufSize = MAX(*bytesReturnedPtr,
                    bufSize + INCREMENT_FOR_PROC_THREAD_INFO_SIZE);
    else
      return NULL;
  }
  return NULL;
}
```

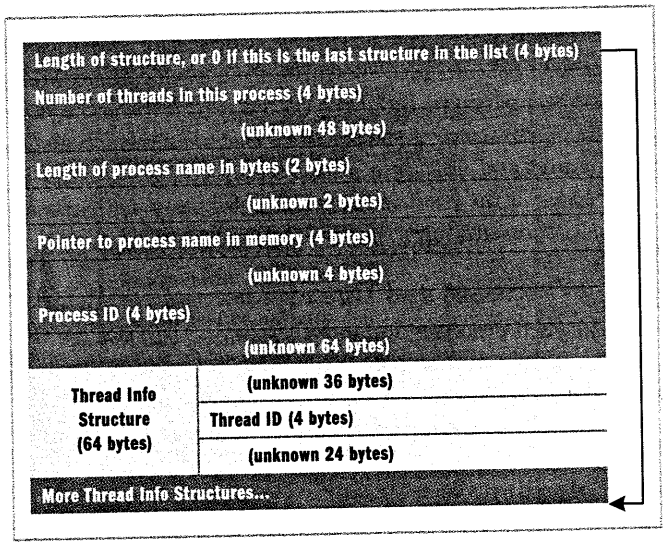

Figure 12 Process Information Structure

User-level Service

Some of VTrace's general operations are easier and safer to perform at user level than at kernel level. So VTrace includes a user-level service, VTrcSrvc, which the system launches at startup. This service runs continuously. When the user has been idle for two hours, it takes a metadata snapshot, compresses all the trace and metadata files collected, uploads those files to our Web site, and deletes them from the local hard drive. It doesn't do this again for 24 hours unless the trace files take up more than 50MB.

The user-level service also checks for changes in the current user. Whenever a new user logs on or the logger signals that a new log file has started, it generates a log entry describing the current user's name. It uses RegNotifyChangeValue on the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon to ensure it's notified when the current user name changes.

Pentium Cycle Counter

Whenever the logger driver receives a description of an event, it writes a time stamp to the log just before it writes the event description to the log. To get accurate time stamps for our trace events, we use the Pentium cycle counter. This counter gives the number of cycles that have passed since the computer started up. We access it with the RDTSC instruction, which can be coded in C by invoking assembler, as shown in these lines of code:

```
__asm {
  push eax          ; Save registers we will overwrite (eax, ebx, edx).
  push ebx
  push edx
  _emit 0x0F        ; The RDTSC instruction consists of these two bytes.
  _emit 0x31
  mov ebx, bufPtr   ; Put the address where the timestamp goes in ebx.
  mov [ebx], eax    ; Save low 4 bytes of timestamp there.
  mov [ebx+4], edx  ; Save high 4 bytes of timestamp next.
  pop edx           ; Restore overwritten registers.
  pop ebx
  pop eax
}
```

Implementing VTrace in Windows 2000

Windows 2000 is similar to Windows NT, but there are enough differences that porting VTrace to it required some effort. In this section we'll describe some of the changes we made so that VTrace would work on Windows 2000.

The biggest difference between Windows 2000 and Windows NT is that Windows 2000 has kernel-mode write protection. This means that kernel-mode code cannot write read-only memory (such as the kernel image) without causing a system crash. Since our method of hooking context switches requires that we overwrite the first instruction of the context-swap code in memory, the kernel mode write protection causes problems for VTrace. The

Figure 13 Writing Read-only Memory

```
// This function should only be called on Windows 2000. It's not
// necessary on Windows NT, and, besides, on Windows NT 4.0 SP3 and
// earlier, MmMapLockedPages() does not give the result we need.

NTSTATUS WriteReadOnlyMemory (char *dest, char *source, int length)
{
  KSPIN_LOCK tempSpinLock;
  KIRQL oldirql;
  PMDL mdl;
  PVOID writableAddress;

  mdl = IoAllocateMdl((PVOID) dest, length, FALSE, FALSE, NULL);
  if (mdl == NULL)
    return STATUS_UNSUCCESSFUL;
  MmBuildMdlForNonPagedPool(mdl);
  MmProbeAndLockPages(mdl, KernelMode, IoWriteAccess);
  writableAddress = MmMapLockedPages(mdl, KernelMode);
  if (writableAddress == NULL) {
    MmUnlockPages(mdl);
    IoFreeMdl(mdl);
    return STATUS_UNSUCCESSFUL;
  }

  // It is imperative that no context switch happens during the
  // copying, so we protect the write with a spin lock. (Context
  // switches are disabled while a spin lock is held.)

  KeInitializeSpinLock(&tempSpinLock);
  KeAcquireSpinLock(&tempSpinLock, &oldirql);
  RtlCopyMemory(writableAddress, source, length);
  KeReleaseSpinLock(&tempSpinLock, oldirql);

  MmUnmapLockedPages(writableAddress, mdl);
  MmUnlockPages(mdl);
  IoFreeMdl(mdl);
  return STATUS_SUCCESS;
}
```

solution is to map the memory to a writable address (see **Figure 13**). Also, the context-swap routine is different in Windows 2000, so the tracer must look for this new routine in memory.

Windows 2000 expects filter drivers to provide two additional dispatch routines to deal with power management and plug-and-play requests. To pass on a power-management IRP, a dispatch routine must first call PoStartNextPowerIrp, and must use PoCallDriver instead of IoCallDriver. When a filter device receives a plug-and-play request, it must check whether the minor function number is IRP_MN_REMOVE_DEVICE. If this type of request completes successfully, the device to which the filter device is attached has removed itself, so the filter device should detach and delete itself.

A particularly complicated plug-and-play request type to handle is IRP_MN_DEVICE_USAGE_NOTIFICATION. Such a request can indicate that a page file on the underlying device either started or stopped being used. A file system filter or disk filter must keep track of how many in-use page files the underlying device has, and update this count whenever it receives one of these notifications. Updating this count is complicated by the fact that the device must, in some cases, update the count when this request arrives, then undo it if the request fails. The DDK provides samples showing how to do this.

In the metadata section we discussed how to determine the disk and partition numbers of a drive. Unfortunately, this particular method does not work in Windows 2000, and the method that does work will not work in Windows NT. In Windows 2000, you must use a new device I/O control code, IOCTL_STORAGE_GET_DEVICE_NUMBER. Passing this code to an open file representing the raw disk yields a STORAGE_DEVICE_NUMBER structure containing the disk and partition numbers.

In Windows 2000, the process information structure is slightly different from that shown in **Figure 12**. The unknown 64 bytes in the header are actually 112 bytes long in Windows 2000.

The rest of the changes in Windows 2000 concern its filter plug-and-play feature, which makes it easier to attach a filter to every device of a certain type. To use this, our installer adds the name of the filter driver to the UpperFilters value of type REG_MULTI_SZ in the registry key corresponding to the class of devices we want it to filter (such as HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-08002BE10318} for keyboards). The filter driver's DriverEntry routine makes DriverObject->DriverExtension->AddDevice a pointer to a function that (like the example in **Figure 14**), creates a filter device, and has it filter a given device. Also, if a filter driver doesn't have to start at boot, the installer gives it a Start parameter of SERVICE_DEMAND_START (instead of SERVICE_AUTO_START) when it sets this parameter with CreateService or ChangeServiceConfig.

If you use a filter driver in this way, be extremely careful that your installer and uninstaller never leave the system in a state where your driver does not exist but is listed as an upper filter. This is important because the operating system will refuse to create any devices requiring a nonexistent filter. The user would get pretty

Figure 14 AddDevice Routine

```
NTSTATUS VTrcKbdAddDevice (IN PDRIVER_OBJECT driverObject,
                           IN PDEVICE_OBJECT physicalDeviceObject)
{
  PDEVICE_OBJECT       filterDevice;
  PKBD_HOOK_EXTENSION  filterExtension;
  NTSTATUS             status;

  // Create the filter device.

  status = IoCreateDevice(driverObject, sizeof(KBD_HOOK_EXTENSION), NULL,
                          FILE_DEVICE_KEYBOARD, 0, FALSE, &filterDevice);
  if (!NT_SUCCESS(status))
    return status;

  // Set flags for the filter device.

  filterDevice->Flags |= (DO_BUFFERED_IO | DO_POWER_PAGABLE);

  // Attach the filter device to the existing keyboard device.

  filterExtension = filterDevice->DeviceExtension;
  filterExtension->attachedDevice =
    IoAttachDeviceToDeviceStack(filterDevice, physicalDeviceObject);
  if (filterExtension->attachedDevice == NULL) {
    IoDeleteDevice(filterDevice);
    return STATUS_UNSUCCESSFUL;
  }

  // Indicate that the filter device is finished initializing.

  filterDevice->Flags &= ~DO_DEVICE_INITIALIZING;

  return STATUS_SUCCESS;
}
```

Figure 15 VTrace Benchmark Results

Operation	Time Without VTrace	Time With VTrace	Slowdown
Read an uncached 32KB file	9.16 ms	9.17 ms	0.1%
Write 1KB file (write-through)	25.05 ms	25.05 ms	0%
Read 32KB direct from disk	9.17 ms	9.17 ms	0%
Copy a 32KB file locally	6.29 ms	6.57 ms	4.5%
Copy a 32KB file remotely	27.73 ms	35.07 ms	26.4%
ZwFlushInstructionCache	2.78 µs	3.72 µs	33.8%
WaitMessage	8.98 µs	64.84 µs	722%
TranslateMessage	0.11 µs	42.19 µs	40178%
Compile logger with DDK	10.23 s	11.60 s	13.4%
Format article with LaTeX	1.69 s	1.79 s	5.3%

annoyed if the keyboard didn't work, or if the system blue-screened on boot because it couldn't start a disk device necessary to mount the boot partition. For the same reason, make sure your DriverEntry routine never returns an error, because in this case as well, the operating system will refuse to create any devices that appear to require your failing filter. We think that these behaviors constitute a bug in the operating system, but it hasn't been treated as one.

As useful as the filter plug-and-play feature is, we only use it for our keyboard and raw disk filters. It appears that Windows 2000 will not support it on virtual device classes such as file systems and network protocols. We could filter these classes on a device-by-device basis using SetupDiSetDeviceRegistryProperty, but then if any additional devices were added to the system later on, they would not get filtered.

Benchmarks

With all the tracing that VTrace does, you may be wondering how it affects the system performance. We wanted the overhead to be unnoticeable so users would let us install it on their systems. By this measure, we succeeded, since no user has ever complained about a performance hit.

This may not mean much, though, since it's hard for users to detect subtle differences, especially on today's fast machines. So we designed a few benchmarks to show the effects of running VTrace. We ran each of these benchmarks on our PC, which has a 450MHz Pentium III, is connected to a 100Mbps switched Ethernet, has 128MB of memory, and has 10GB divided among three SCSI disks. We ran each benchmark (other than the compilation and document format benchmarks, which take too long) often enough that we could be 95 percent sure the real mean was within 0.1 percent of the estimated mean. We also instrumented VTrace to find out how much overhead there is just to write a single short log entry; on average, this takes 20.24 µs from user level, but only 0.95 µs from kernel level.

Figure 15 lists the mean benchmark results, showing how much VTrace slows down various operations. You can see that VTrace has almost no effect on simple reads and writes, since there isn't much to log and all the logging is at kernel level. Copying files incurs more tracing overhead, especially when VTrace is also tracing network operations. Calling various traced functions like ZwFlushInstructionCache, WaitMessage, and TranslateMessage incurs overhead essentially due to the overhead of writing a log entry. As you can see, this is substantial for the latter two functions since they don't do much, but they're at user level. Finally, you can see the big picture from the two application benchmarks, which show that VTrace makes a 10-second compilation take 13.4 percent longer and a two-second document-format take 5.3 percent longer.

These benchmarks suggest that the biggest area for improvement is the overhead of tracing user-level events. They seem to indicate that we could substantially improve VTrace's performance by having it trace user-level events entirely at user level. To test this, we wrote a version of VTrace that did separate kernel-level and user-level logging. This approach reduced the overhead for user-level logging tremendously, from about 20 µs to only about 0.25µs. However, the extra processing required to perform separate user-level and kernel-level tracing dominated these improvements, causing this separation approach to actually do slightly worse in the macrobenchmarks than our original approach. So, in the final version of VTrace, we perform all logging at kernel level.

Conclusion

If you're interested in participating in our experiments, please download our tracer from http://www.cs.berkeley.edu/~lorch/vtrace and install it on your machine. Of course, you must be running Windows NT or Windows 2000.

Building VTrace for Windows NT and Windows 2000 was challenging because of the difficulty inherent in system-level programming and the lack of official documentation. Nevertheless, with the help of many sources of information, including developer tools, magazines, books, Web sites, and Usenet, we achieved our goal. We believe the techniques we've described, as well as the references we've provided, will be helpful to system-level programmers using Windows NT and Windows 2000. Keep in mind that even if an operating system isn't designed to let you perform tasks like we've described here, that doesn't mean it can't be done.

Jacob R. Lorch and **Alan Jay Smith** are affiliated with the Computer Science Division at the University of California at Berkeley. Jacob (http://www.cs.berkeley.edu/~lorch) is a graduate student researching ways to reduce the energy consumption of portable computers. Alan (smith@cs.berkeley.edu) is a professor whose research interests include the analysis and modeling of computer systems and devices, computer architecture, and operating systems.