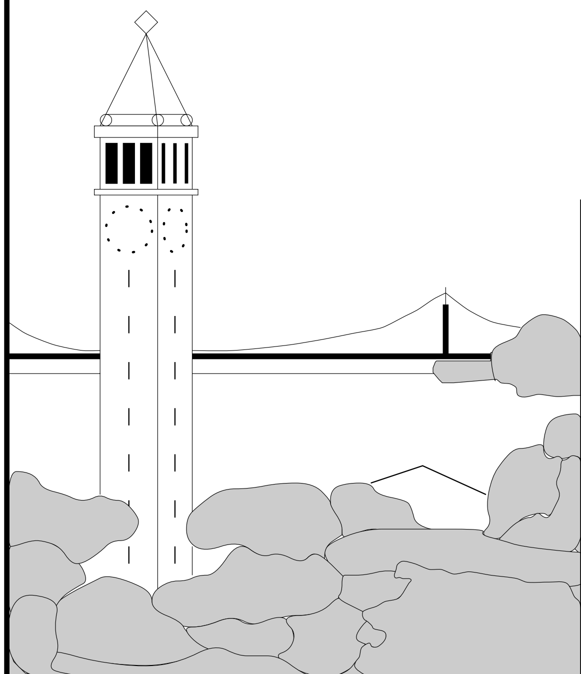
Building VTrace, a Tracer for Windows NT and Windows 2000

Jacob R. Lorch
Alan Jay Smith



Report No. UCB/CSD-00-1093

February 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Building VTrace, a Tracer for Windows NT and Windows 2000*

Jacob R. Lorch† Alan Jay Smith†

February 2000

Abstract

In order to conduct accurate simulations of new approaches to energy management, we needed to collect detailed, time-stamped traces of several diverse types of activity on Windows NT and Windows 2000. For this purpose, we wrote VTrace, which collects data about processes, threads, messages, disk operations, network operations, the keyboard, the mouse, and the cursor. Building this tool required a large number of special techniques, which we describe in this paper. These techniques included using a DLL loaded into the address space of every process to intercept Win32 system calls; establishing hook functions for Windows NT kernel system calls; modifying the context switch code in memory to log context switches despite inadequate operating system support; and using device filters to log accesses to devices such as file systems, disk partitions, network transport layers, and the keyboard. We also describe related issues, such as where we found the necessary information, and how to debug a tracing tool that is intimately connected to the operating system kernel. Finally, since VTrace was originally written for Windows NT but later modified and extended to run with Windows 2000, we briefly discuss some of the changes required for Windows 2000.

1 Introduction

Writing a tracer for an operating system is difficult for many reasons. An operating system is a large, complex piece of software to analyze. Debugging code that runs before the system has fully started up is tricky. Many runs require a reboot of the computer, and failed runs can necessitate reinstalling the entire operating system or even reformatting the hard drive. However, writing a tracer for Windows NT/2000 is *especially* difficult, because source code is unavailable, descriptions of its internal operations are largely unavailable, and documentation of its interface is incomplete.

We found ourselves facing this difficulty when we decided we needed detailed, time-stamped traces of certain Windows NT/2000 activities for a study of new energy management techniques for laptop computers. This required that we deal with all of these problems. Our studies include the effect of varying the CPU voltage and clock speed, and powering down various system components. We needed to know when various power-consuming components (such as the CPU, the disk, and the network interface card) are active and what they're doing at each instant. Thus, we needed traces of several different types of system objects: processes, threads, messages, waitable objects, key presses, file systems, disks, and the network. Furthermore, we wanted the tracer to be non-intrusive and to respect the confidentiality of users' data so that users would agree to let us trace their systems. Ultimately, we succeeded in developing such a tracer: VTrace, containing over 30,000 lines of code in C, C++, and some assembler. In this paper, we describe what we did to accomplish this so that the reader can repeat and extend these techniques.

The paper is organized as follows. In Section 2, we describe the information resources we found helpful. We report the difficulties we had in setting up a debugging environment and how we overcame them in Section 3. Section 4 describes how we adapted standard techniques to create drivers that perform logging and that filter accesses to the keyboard, file systems, network, and hard disks. In Section 5, we describe our unique approach to logging context switches in Windows NT. Sections 6 and 7 describe how we logged Win32 system calls and NT kernel system calls. In Section 8, we discuss how we parse the master file table of NTFS partitions to deduce file system metadata information. In Section 9, we briefly consider some of the more interesting miscellaneous interesting features of the tracer. In Section 10, we discuss how we modified the tracer to work on Windows 2000. We present results from benchmarks that show how much overhead VTrace places on the system in Section 11. In Section 12, we describe software that uses techniques similar to those used in VTrace. Finally, we summarize in Section 13.

2 Information Resources

Writing software that traces system activity requires a great deal of information, both about how the operating sys-

*Funding for this research has been provided by the State of California under the MICRO program, and also by Cirrus Corporation, Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Microsoft Corporation, Quantum Corporation, Sun Microsystems, and Toshiba Corporation.

†Computer Science Division, EECS Department, University of California, Berkeley, CA 94720-1776, {lorch, smith}@cs.berkeley.edu

tem works and about how to write tracing tools. Microsoft provides a lot of this information through their developer tools, magazine, and web sites. Unfortunately, this information is not sufficient, since (1) Microsoft does not document many aspects of the internal operations and interfaces of Windows NT/2000, and (2) when Microsoft *does* describe something, this description may be difficult to understand or to generalize, e.g. when the documentation is simply one source code example. To be successful, therefore, we needed information from many sources: developer tools, USENET, magazines, books, and web sites.

Useful information came with a few of our development tools. Microsoft Developer Studio, naturally, has help that describes the interfaces to many well-documented Win32 system calls. It also has sufficient help on Microsoft Foundation Classes (MFC) that we were able to learn MFC programming without a book. The Windows NT and Windows 2000 Driver Development Kits (DDK's) have extensive help systems that describe basic and advanced driver development; they also include some useful sample source code. Finally, Microsoft's kernel-mode debugger, WinDbg, comes with some information about how to use it.

USENET was a good source for discussions of real-world problems and solutions in Windows NT/2000 programming. The useful articles are in the comp.os.ms-windows.programmer.* hierarchy. For our purposes, the most useful newsgroup was comp.os.ms-windows.programmer.nt.kernel-mode, which covers kernel-mode programming. To get the most out of USENET, we recommend using a site that can search USENET archives, such as Deja News (http://www.dejanews.com).

Magazines were another source of useful information. Open Systems Resources, Inc. (http://www.osr.com, ntinsider@osr.com) provides free subscriptions to *NT Insider*, which discusses many useful aspects of driver development. *Microsoft Systems Journal* (http://www.microsoft.com/msj) also contains many helpful articles, especially the Under the Hood columns by Matt Pietrek. Windows Developer's Journal is another good source. Finally, we recommend *Dr. Dobb's Journal* (http://www.ddj.com), especially for its articles by Mark Russinovich and Bryce Cogswell.

We also used books. "Inside Windows NT" and "Inside the Windows NT File System," both by Helen Custer, are useful for a general overview of the operating system and file system, although they do not offer aid in actual programming [2, 3]. In contrast, "Windows NT File System Internals," by Rajeev Nagar, both describes how the file system works and provides practical advice for interfacing with it [8]. For basic Windows NT programming strategies, we used "Windows NT 4 Programming from the Ground Up," by Herbert Schildt [13].

However, by far the most useful source of information, without which the tool might never have been developed, was the World Wide Web. Often, we found the solution to a problem simply by using a web search engine on key words or phrases. The web also has sites containing large, structured bodies of information on Windows NT/2000. For example, the Systems Internals web site (http://www.sysinternals.com) has a great deal of useful information, utilities, and even source code for Windows NT systems programming. Also, the Microsoft Developer Network Library (http://premium.microsoft.com/msdn/library) has helpful articles and documentation.

We thus observe that although Microsoft does not completely document Windows NT/2000, so many developers have used it and are willing to share information that it has become extensively, if unofficially, documented.

3 Creating a Debugging Environment

A tracer contains and interacts with a lot of code that runs in kernel mode, so we needed a kernel-mode debugger. Furthermore, since much of the code in a tracer gets run before the system has completely started up and thus before a debugger program can be launched, we did well to use a two-system debugging environment. In such a configuration, the debugger runs on the *host* machine (also the development machine), and the software under test runs on the *target* machine. The debugger monitors and controls the target machine via a serial cable connecting the two machines. Another advantage to this approach is that a reboot or reinstallation of the operating system on the test machine does not affect the development environment.

Unfortunately, setting up kernel debugging with our debugger, WinDbg, is notoriously difficult. Some of the hardest things are configuring the debugger program settings correctly and making the target machine communicate with a remote debugger. For this, documentation included with the debugger is helpful, as are stories on the web about user experiences. However, even with all this, we still had difficulty. We finally succeeded once we discovered we had an old, buggy version of WinDbg, and downloaded a fixed version from an obscure location at Microsoft described in an equally obscure USENET article posted by Paul Sanders (paulsan@microsoft.com) to the kernel-mode programming newsgroup on October 17, 1997. (The latest version of WinDbg is now easier to find and lacks these bugs.) We had other problems until we realized we needed to upgrade the debug symbol files to match the service pack installed on the target machine. We searched the web, and found we could download these files from Microsoft's FTP site.

Once we had the debugger 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 (albeit in uncommented assembler).

When the target machine being debugged is running Windows 2000, sometimes it will inexplicably hang while starting up. To get past this, use the Break command in the Debug menu of WinDbg.

4 Drivers

4.1 Background

Driver programming for Windows NT/2000 is a subject of tremendous breadth, so our treatment here will necessarily be incomplete. [2], [8], back issues of NT Insider, and the Windows NT and Windows 2000 DDK's discuss it more thoroughly.

In Windows NT/2000, a *device* is an object that can receive I/O requests, such as one that represents a disk drive, a file system, or a keyboard. Devices can be *layered* above each other, in 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 state carried over between I/O requests. Despite the name, file objects can represent more than just open files; they can also represent things like network connection endpoints and open directories.

Device layering is accomplished naturally by the way I/O requests are handled. 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*. This stack's top location contains a description of the request in a form the device understands. Before that 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 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 in order 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. See Figure 1 for an illustration of this.

A *driver* is the code implementing a class of devices. For instance, all NTFS file system devices use the NTFS driver code for handling requests. This code includes a driver entry routine and several dispatch routines. The driver entry routine does 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.

A filter driver implements a filter device, a special kind of device extremely helpful in tracing system events in Windows NT/2000. A filter device can *attach* to an existing device, causing it to intercept any requests destined for that existing device. Typically, it modifies the request in some way, then passes it on to the existing device. This allows it to add functionality to that device, e.g., to turn a traditional file system into an encrypted file system. However, a filter can be

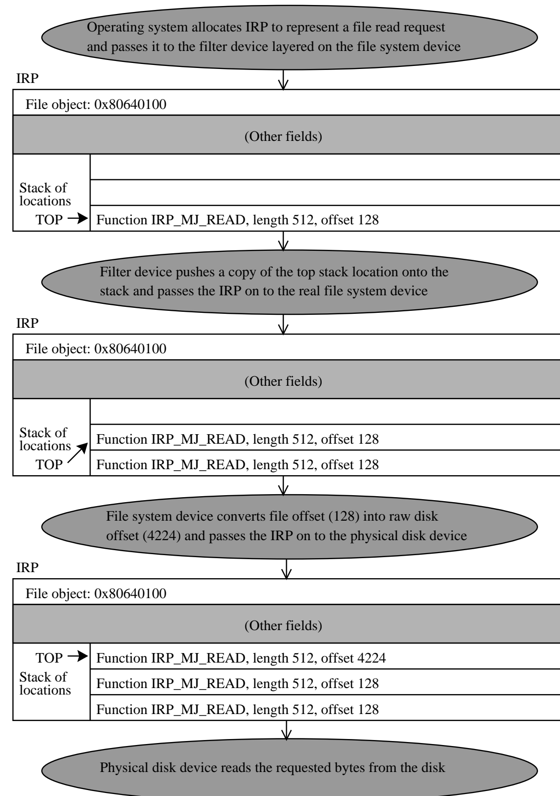


Figure 1: Example of how layered devices use an IRP stack

used simply to record information about requests, pass those requests on unchanged to the device they were meant for, then record information about the results of those requests.

Figure 3 gives an example of an initialization routine for a filter driver. This routine, DriverEntry(), sets the MajorFunction entries in the driver object so the appropriate dispatch routine gets called for each request type. Those dispatch routines, such as the one shown in Figure 2, log the request initiation, set a completion routine to be called when the request completes, then call the lower-level driver to complete the request. Figure 4 shows an example of a completion routine that logs the request completion.

4.2 Logger

Kernel-mode code can also use device control requests like this to communicate with the logger driver. But, it's more efficient for kernel-mode code to just call the logger driver's functions directly. To get pointers to these functions, a driver makes a single device control request to the logger driver requesting a structure containing all such function pointers.

A major component of VTrace is the *logger*, which 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 that its code could run in kernel mode. This enables other kernel-mode code, such as that

```
NTSTATUS VTrcFSDispatchReadWrite (PDEVICE_OBJECT HookDevice, IN PIRP Irp)
{
  PIO_STACK_LOCATION   currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
  PIO_STACK_LOCATION   nextIrpStack    = IoGetNextIrpStackLocation(Irp);
  PFILE_OBJECT         fileObject      = currentIrpStack->FileObject;
  PSTD_HOOK_EXTENSION  hookExt;
  PDEVICE_OBJECT       deviceObject;
  ULONG                seq;
  KIRQL                oldirql;
  PCHAR                eventBuffer;

  // If the file has no name, just pass this IRP down to the next driver normally.

  if (fileObject->FileName.Buffer == NULL)
    return VTrcFSPassOnNormally(HookDevice, Irp);

  // Log the read or write request.

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

  // Get a pointer to the lower-level device object from the extension, then
  // copy parameters down to next level in the stack for the driver below us.

  hookExt = HookDevice->DeviceExtension;
  deviceObject = hookExt->attachedDevice;
  *nextIrpStack = *currentIrpStack;
  nextIrpStack->DeviceObject = deviceObject;

  // Set the completion routine, passing the sequence number as the
  // "context" parameter so that it is used in the completion log entry.
  // Then, call the lower-level driver.

  IoSetCompletionRoutine(Irp, VTrcFSCompletionRoutine, (PVOID) seq,
                         TRUE, TRUE, TRUE);
  return IoCallDriver(deviceObject, Irp);
}
```

Figure 2: VTrace uses a dispatch routine like this one to handle read and write calls that its file system filter intercepts.

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

  // Read shared state from main driver.
  // [Code for GetSharedState() not shown.]

  status = GetSharedState(&sharedState);
  if (status != STATUS_SUCCESS)
    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 = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    DriverObject->MajorFunction[i] =
      VTrcFSPassOnNormally;

  DriverObject->MajorFunction[IRP_MJ_CREATE] =
    VTrcFSDispatchCreate;
  DriverObject->MajorFunction[IRP_MJ_READ] =
  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 put an unload routine
  // address in 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 3: VTrace uses a DriverEntry() routine like this one to initialize the file system filter driver.

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

  // Log the return values.

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

  // Always do the following in a completion
  // routine.

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

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

Figure 4: VTrace uses a completion routine like this one to log the results of a file system request that just completed.

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

  // Get a handle to the device.

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

  if (hDevice != INVALID_HANDLE_VALUE) {
    DeviceIoControl(
      hDevice, VTRACE_LOG_EVENT,
      eventDescription, descriptionLength,
      NULL, 0, // No inbound buffer
      &returnSize, NULL);
    CloseHandle(hDevice);
  }
}
```

Figure 5: This function, which logs an event, illustrates how user-mode code communicates with a kernel-mode driver.

comprising most of the tracer, to call it efficiently. It also allows user-mode code to call it without incurring a context switch, although with the overhead of a kernel trap. The logger driver implements several specialized device control I/O request types, including ones to start logging, stop logging, add an event to the log, flush the log to disk, and change the event mask (the set of event types the logger leaves out of the log). User-mode code makes such requests using code like that in Figure 5.

The logger periodically changes the event mask (the set of event types the logger leaves out of the log) according to a fixed schedule. This lets us collect the full set of event types only part of the time, to reduce the space taken up by trace files on the user's disk. For example, we collect file-open events (which take up little space) all the time, but thread switches (which are very frequent) only part of the time.

4.3 Keyboard filter

To log key presses, we made straightforward modifications to the keyboard filter driver Ctrl2cap, whose code is freely available from the Systems Internals web site. Its original purpose was to exchange the functions of the control and caps lock keys. We simply made it encrypt and log the key presses instead of modifying them.

4.4 File system filter

To log file system activity, we made a few modifications to Filemon, another filter driver whose source code is available from the Systems Internals web site. This filter driver, described in [12], logs and displays file system activity. The most important modification we made was in how to choose the file system devices to filter. Filemon requires the user to specify the file systems desired, but we wanted to filter all file systems. Furthermore, we could not simply find all the file systems at system start-up and filter those, since some file systems, such as floppy disks and CD's, may be added dynamically. So, instead, we arranged to hook the NT system call that opens files, check in that hook whether we have yet filtered the file system containing that file, and start filtering any unfiltered file system we find. Section 7 will discuss how we hooked the file-open and other NT system calls.

One complication in filtering file system devices is an optimization called the *fast I/O path*. If a file system device can handle a request without involving a lower-level device, e.g., during a cache hit, the overhead of creating an IRP is unnecessary. 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 IRP's. If the fast dispatch routine cannot handle the request, e.g. if it needs to use a lower-level device, it returns an error value, making the operating system send an IRP to the regular dispatch routine. To filter accesses that use fast I/O, we must specify a set of fast dispatch routines in our filter driver. These fast dispatch routines will log getting called and pass on calls to the fast dispatch routines of the lower-level driver. Figure 6 shows a sample logging fast I/O routine. Also, the sample driver initialization code in Figure 3 has a line to set up the fast I/O dispatch table.

4.5 Raw disk partition filter

To log activity at the physical disk level, we modified a physical disk filter driver, DiskPerf, whose source code is included in the DDK. This driver collects and reports statistics about raw disk accesses, so it was straightforward to retool it so it instead logged information about each disk access.

4.6 Network filter

In contrast to the other filter drivers we needed, we found no source code for a network transport layer filter driver. We thus had to write one from scratch. More precisely, we had to write one mostly from scratch, since many things are the same from one filter driver to another, such as how to initialize the dispatch table, how to attach to lower-level drivers, etc.

Windows NT/2000 provides a single programming interface, called the transport driver interface (TDI), to the transport layers of all network protocols. (See Table 1.) I/O requests passed to the transport layer all conform to the same format, described in the Windows NT DDK help and in the DDK files TDI.H and TDIKRNL.H. There were still challenges, however, in building a filter for these requests.

One problem we encountered is that some IRP's have the major function code "device control," and we found no description of the parameter format used by these IRP's. However, we learned from the DDK help that the first thing a device does upon receiving 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. In our filter driver dispatch routine for device control requests, we therefore first call TdiMapUserRequest().

Another problem is caused by an apparent bug in how Windows NT handles network filter devices. When it constructs an IRP, it must allocate enough stack space in it to account for the maximum depth of the device stack the IRP will pass through. To ensure this, each device object has a stack count field indicating how large the stack must be in IRP's it receives. Unfortunately, Windows NT sometimes 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 the system crashes.

We solve the stack space problem in different ways, depending on whether we need to post-process the request. When we need to post-process, we create a new IRP with the appropriate stack space to pass on to the lower-level

```
BOOLEAN MyFastIoRead (PFILE_OBJECT FileObject, PLARGE_INTEGER FileOffset,
                      ULONG Length, BOOLEAN Wait, ULONG LockKey, PVOID Buffer,
                      PIO_STATUS_BLOCK IoStatus, PDEVICE_OBJECT DeviceObject)
{
  PSTD_HOOK_EXTENSION hookExt = DeviceObject->DeviceExtension;
  PDEVICE_OBJECT origDevice = hookExt->attachedDevice;
  PFAST_IO_DISPATCH origFastIoTable = origDevice->DriverObject->FastIoDispatch;
  BOOLEAN retval;
  ULONG seq;
  KIRQL oldirql;
  PCHAR eventBuffer;

  // If there is no fast I/O routine in the original driver, return FALSE.
  if ((ULONG) &origFastIoTable->FastIoRead - (ULONG) &origFastIoTable >=
      origFastIoTable->SizeOfFastIoDispatch || !origFastIoTable->FastIoRead)
    return FALSE;

  // If there is a file name, record this call.
  if (FileObject->FileName.Buffer != NULL) {
    // Get a new sequence number to keep track of this request.
    seq = InterlockedIncrement(&sequenceNumber);

    // Log this call, describing the input parameters.
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventBuffer = (*sharedState->logEventFunc)((char) ENTRY_TYPE_FILE_READ, 24);
    if (eventBuffer) {
      RtlCopyMemory(&eventBuffer[1], &seq, 4);
      RtlCopyMemory(&eventBuffer[5], &FileObject, 4);
      RtlCopyMemory(&eventBuffer[9], FileOffset, 5);
      RtlCopyMemory(&eventBuffer[14], &Length, 4);
      RtlZeroMemory(&eventBuffer[18], 4); // no IRP flags
      eventBuffer[22] = IRP_MN_NORMAL;
      eventBuffer[23] = '\0';             // no stack flags
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  // Call the real fast I/O routine, recording the return value.
  retval = origFastIoTable->FastIoRead(FileObject, FileOffset, Length, Wait,
                                       LockKey, Buffer, IoStatus, origDevice);

  if (retval && FileObject->FileName.Buffer != NULL) {
    // Log the return, including return values.
    KeAcquireSpinLock(&sharedState->mainMutex, &oldirql);
    eventBuffer = (*sharedState->logEventFunc)
      ((char) ENTRY_TYPE_FILE_COMPLETE_OPERATION, 13);
    if (eventBuffer) {
      RtlCopyMemory(&eventBuffer[1], &seq, 4);
      RtlCopyMemory(&eventBuffer[5], &IoStatus->Status, 4);
      RtlCopyMemory(&eventBuffer[9], &IoStatus->Information, 4);
    }
    KeReleaseSpinLock(&sharedState->mainMutex, oldirql);
  }

  // Return the original routine's return value.
  return retval;
}
```

Figure 6: VTrace uses a fast I/O routine like this one to handle fast-path read requests.

Minor function code	Meaning
TDI_ASSOCIATE_ADDRESS	associate a connection endpoint with a network address
TDI_DISASSOCIATE_ADDRESS	disassociate a connection endpoint with the network address it was previously associated with
TDI_CONNECT	establish a connection between a local connection endpoint and a specified remote address
TDI_LISTEN	listen for requests from any of a set of remote addresses to a local connection endpoint
TDI_ACCEPT	accept a connection request made by a remote address to a local connection endpoint
TDI_DISCONNECT	terminate the connection in which a connection endpoint is participating
TDI_SEND	send an ordered packet over a connection
TDI_RECEIVE	receive an ordered packet over a connection
TDI_SEND_DATAGRAM	send a datagram over a connection
TDI_RECEIVE_DATAGRAM	receive a datagram over a connection
TDI_SET_EVENT_HANDLER	establish a routine for handling a certain type of event, such as the arrival of a datagram
TDI_QUERY_INFORMATION	get information about some network object, such as its network address
TDI_SET_INFORMATION	set information about some network object
TDI_ACTION	perform some transport-specific action

Table 1: Minor function codes of some useful TDI internal device control requests, taken from the Windows NT DDK help

driver. When we do not need to post-process, we use a trick borrowed from Filemon: we do not push anything onto the stack, allowing the next device down to use the same stack location it used. We can only do this when we do not need to post-process, since if the request were passed back after doing this the stack would be empty.

Yet another difficulty stems from a unique aspect of network devices, namely that not all network I/O uses IRP's. Specifically, I/O that happens in response to some event, such as a datagram arrival, is performed entirely by functions called *event handlers* and does not involve the dispatch routines. This is unfortunate, since while Windows NT provides filter devices as an elegant, well-supported approach to intercepting IRP's sent to dispatch routines, it provides no special support for intercepting calls to event handlers.

We overcame this by developing our own technique for intercepting calls to event handlers. The key to this technique is our ability, thanks to filter devices, to intercept and change any request that specifies 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 be passed 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 structure we allocated. In this way, whenever an event of the given type happens, our special logging event-handling function gets called and passed the address of the structure we allocated. This function logs the event, 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. In actuality, we used a slightly different approach to memory allocation than described above: the driver allocates a single structure per file object, not per event handler; this permits it to quickly free all the memory allocated for a file object when it is closed.

5 Logging Context Switches

Logging context switches should be easy, since kernel-mode software can use the function KeSetSwapContextNotifyRoutine() to set a function that gets called whenever context switches occur. Unfortunately, this call only works on the *checked build* of Windows NT/2000, a special version that contains extra hooks and symbols for use in driver development. Few people use this version, and we wanted our tracer to run on anyone's machine, so we designed a method that will work on the standard version.

We devised the method as follows. WinDbg comes with debug symbols for Windows NT files, so we used it to find the assembler code for the Windows NT function SwapContext(). The function is more than five bytes long, and contains no jumps, branches, or calls in its first five bytes, enabling us to do the following. We overwrite, in memory, its first five bytes with a jump to our own function, NewSwapContext(), as shown in Figures 7 and 8. NewSwapContext() logs the context switch, including the thread being switched to; then executes the first five bytes of the original, pre-overwrite version of SwapContext(); then jumps to the sixth byte of SwapContext(). This method was inspired by an article in Microsoft Systems Journal unrelated to context switches[10].

Finding the location of SwapContext() in memory is not straightforward, however. It is always in the in-memory image of the kernel executable, NTOSKRNL.EXE, which is loaded at address 0x80100000. However, its position within NTOSKRNL.EXE varies 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 5 it's at

Before:

```
SwapContext:
  mov byte ptr es:[esi+2Dh],2
  or cl,cl
  mov ecx,dword ptr [ebx]
  pushfd
  push
  [etc.]
```

After:

```
SwapContext:
  jmp NewSwapContext
  or cl,cl
  mov ecx,dword ptr [ebx]
  pushfd
  push
  [etc.]
```

Figure 7: How we hook the context switch routine. (NewSwapContext is shown in Figure 8.)

0x80141E70. However, in all these versions, the instructions of the function are unchanged, and we expect this to remain the case in future versions of Windows NT 4.0. Thus, to find SwapContext(), we simply search for the known first 28 bytes of the routine in the memory section where we expect it. (We check a few common locations first, since the routine is most likely to be in one of them.) Doing this check is dangerous, since an access to an invalid (e.g., paged out) memory location by kernel-mode software will crash the system. Thus, before checking any location, we first call MmIsAddressValid() to make sure we can read it.

When Service Pack 6 became available, we tested this technique on it, and it worked perfectly.

One final complication in logging context switches involves locking. Normally, we require a thread to hold a spin lock when writing to the log. However, we found that the system sometimes crashed when the context-swap hook tried to acquire this spin lock. This may be because acquring the spin lock can itself cause a context-swap, creating an infinite loop. Thus, we instead disable interrupts while the context-swap routine accesses the log. We can be sure that no other thread holds the spin lock, since context-swaps cannot occur on a uniprocessor while a thread holds a spin lock.

6 Logging Win32 System Calls

The Windows NT/2000 kernel supports multiple user-level subsystems, such as Win32, POSIX, and OS/2. Thus, the term “system call” is vague; it could refer to a call to the Win32 subsystem, to some other subsystem such as OS/2, or even to the Windows NT/2000 kernel itself. In this section, we will discuss logging system calls to the Win32 subsystem. Section 7 will discuss logging system calls to the kernel.

Our technique for logging Win32 system calls borrows

```
void FASTCALL NewSwapContext (void)
{
    // Save parameters we will overwrite.
    push eax
    push ecx

    // Save interrupt mask and stop all
    // interrupts. Since context-swaps can't
    // occur while a spinlock is held, we
    // know no one else has the spinlock.
    pushfd
    cli

    // Put space on stack for local variable
    // eventBuffer, which will be [esp].
    push ecx

    // eventBuffer = LogEvent(
    //   (char) ENTRY_TYPE_THREAD_SWITCH, 5);
    push 5
    push 0Eh
    call LogEvent

    // if (eventBuffer)
    cmp eax, 0
    je DoneLogging
    mov dword ptr [esp], eax

    //   eventBuffer[1] = PsGetCurrentThreadId();
    call PsGetCurrentThreadId
    mov ecx, dword ptr [esp]
    mov dword ptr [ecx+1], eax

  DoneLogging:
    // Pop space on stack for local variable.
    pop ecx

    // Restore interrupt mask.
    popfd

    // Pop saved parameters.
    pop ecx
    pop eax

    // Execute the overwritten instruction from
    // the original swap routine.
    mov byte ptr es:[esi+2dh],2

    // Jump to the point in the original swap
    // routine past the overwritten part.
    jmp dword ptr globals.nonOverwrittenPart;
  }
}
```

Figure 8: VTrace uses this routine for logging context switches.

Code	Explanation
⋮ call _GetMessageA@16 ⋮	Application code. The application calls GetMessageA(), which is compiled as a call to _GetMessageA@16.
_GetMessageA@16: jmp dword ptr [__imp__GetMessageA@16] _PeekMessageA@20: jmp dword ptr [__imp__PeekMessageA@20] ⋮	Stub functions. _GetMessageA@16 is one of several stub functions; it simply performs an indirect jump to the function pointer at location __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 part of 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.

Figure 9: An example of how Win32 system calls are performed

heavily from the technique developed by Matt Pietrek for his APISPY32 program [10]. In this section, we describe Pietrek's technique only briefly; the reader is referred to [10] for a more complete description. We then describe the major ways in which our technique differs from it.

Pietrek's technique relies on a key observation about how applications make Win32 system calls. A Win32 system call is effected by calling a stub function which does an indirect jump to one of an array of function pointers. (See Figure 9.) We must therefore merely find that array of function pointers (which is easy to do once the image and file format is understood [6, 9]), 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 is part of the tracer software, will call the original functions and log those calls. The tracer must load this special DLL into every application's address space.

Pietrek describes several techniques for loading the logging DLL into every application's address space. We chose the simplest of those, putting the name of the DLL in the registry key HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs. This does not take effect until reboot, but our tracer requires a reboot anyway to install the raw disk filter driver.

APISPY32 was designed for Windows NT 3.5, and some USENET articles report it cannot be used with Windows NT 4.0. The reason is that virtual memory protections prevent the replacement of some of the function pointers. To fix this problem, we call VirtualProtect() to temporarily change these protections.

Another problem with APISPY32 is that it only hooks system calls made directly by the application. If an application calls a DLL function, which in turn makes a system call, it does not notice that system call. This is because APISPY32 performs its function interception on the application executable image but not on the image of any loaded DLL. Our solution to this is twofold. First, when the logging DLL is loaded, it calls EnumerateLoadedModules() to obtain the memory locations of the application and all its loaded DLL's. Then, it does the function interception in each of those modules. Second, we intercept the LoadLibrary. . .() functions (even though we do not need to log them), so that when a new library is loaded we can perform function interception on its image.

We found it useful for the logging DLL to be able to determine whether the current thread had performed any recent activity. In this way, we could perform online compression of the log entries logging calls to functions like PeekMessage and SendMessage when the thread was not doing anything else. However, since most thread activity is recorded at kernel level, this required either expensive, frequent communication between kernel and user level, or a region of memory that could be shared between these levels. We opted for the latter approach, to improve the performance of our tracer. Figure 10 shows how a driver can map a region of non-paged pool to a user-level address. One important issue is that the driver must undo this mapping before the process exits, or the system will crash. Fortunately, in order to access the driver to request the mapping, a user process must create a "file" representing a link to the driver. When that process is about to terminate, it automatically closes this file. Therefore, 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.

Debugging a logging DLL can be difficult, since any bug

```
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 user level.

  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;

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

Figure 10: VTrace uses a function like this one to map a region of kernel-level non-paged memory to user level.

in it can make the system unusable, e.g., by making a fundamental application like the login screen fail. If this happens, the only recourse may be to restore the registry to a previous state in which the DLL is not in the AppInit_DLLs list, or to delete the offending DLL file. Each of these approaches is annoying and time-consuming if the system cannot be run normally. One solution is to avoid putting the DLL into AppInit_DLLs and write test applications that explicitly load the DLL. However, this will not test how the DLL works with general applications. The best approach, suggested by a USENET post, is to put the DLL on a floppy disk and tell AppInit_DLLs to get it from there. In this way, if there is a bug, one can simply remove the floppy disk and the DLL will not get loaded into any application.

7 Logging NT System Calls

Our approach for logging system calls to the kernel is derived from the Regmon application, available from the Systems Internals web site and described in [11]. 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 those function pointers with pointers to special logging functions. The trickiest part is figuring out what system call number corresponds to each system call.

Regmon accomplishes this by observing the following about how kernel-mode software makes these system calls. NTOSKRNL.EXE provides the interface to the system calls by exporting functions whose names have the prefix “Zw.” Inspecting these functions, one can see that the first thing they do is load the system call number into register EAX. Thus, the system call number can be extracted from bytes 2–5 of the Zw function.

As mentioned in Subsection 4.4, we hooked the system calls for opening files so we could make sure our file system filter driver attached a device to each active file system. Unfortunately, one of these system calls, ZwOpenFile(), is undocumented in the DDK, so we did not know how to use its parameters to determine what file system to filter. Fortunately, we found this function documented in Nagar’s book [8].

Another problem we encountered is that NTOSKRNL.EXE does not export all the system calls we were interested in logging. Some, such as ZwSignalAndWaitForSingleObject(), are only exported by NTDLL.DLL, with which we were unable to link our driver. (Regmon does not have this problem, since it only hooks system calls exported by NTOSKRNL.EXE.) So, our tracer reads and parses the file NTDLL.DLL to find the Zw function bodies. The file format it uses, called the portable executable (PE) file format, is well documented [6, 9], so parsing it is not difficult.

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 the virtual addresses they will have when loaded into memory, but we need to know where in the *file* those structures are.

To translate from virtual addresses to file positions, we need the section header information. This is an array of IMAGE_SECTION_HEADER structures, each of which describes the absolute file position of a section, the length of that section, and the virtual address where that section will be loaded. Using this information, we can figure out which section contains a given virtual address, and from that the file position for that address. Figure 11 shows where to find these section header structures in the file.

Once we can translate from virtual addresses to file positions, we can find the names and bodies of all the exported functions, using the outline shown in Figure 11. This lets us find where the Zw function bodies are and what their first few bytes are.

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 fails to document one of these system calls, ZwOpenFile(), so we could not at first determine how to use its parameters to determine what file system to filter. Fortunately, Nagar’s book documents this function [8].

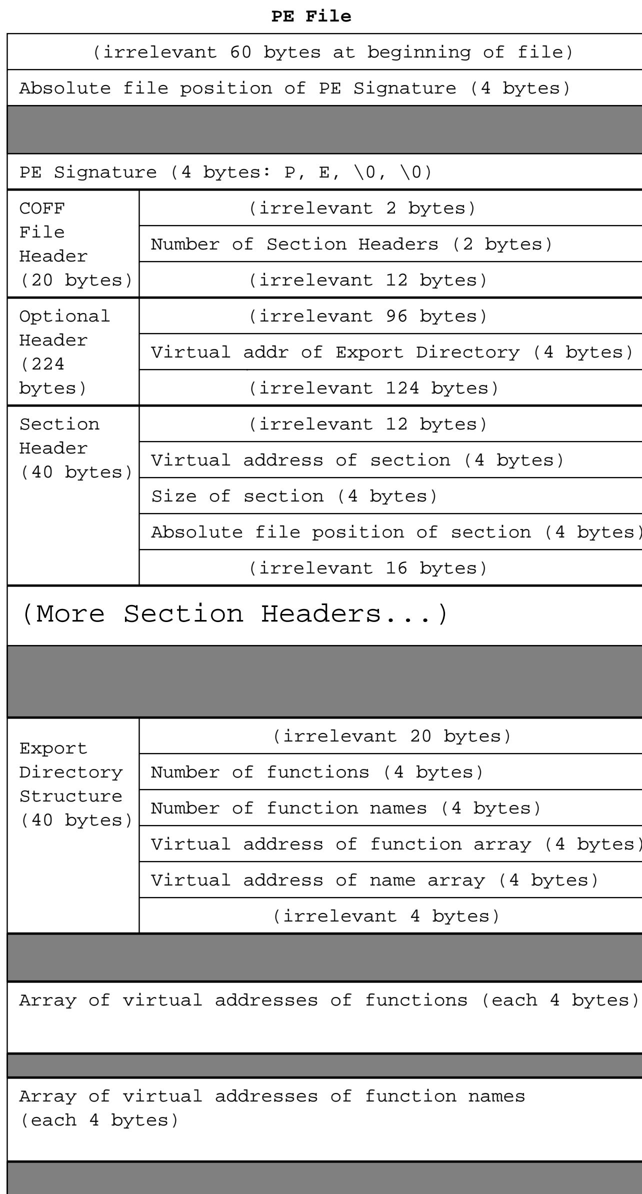


Figure 11: This outline of the structure of PE image files shows how to get what we need from NTDLL.DLL. It shows how to find the section headers you need to translate virtual addresses into absolute file positions. It also shows how to find the virtual addresses of the names and bodies of the exported functions.

8 Parsing File System Metadata

Our design goals required us to take periodic snapshots of the file system metadata of each local NTFS partition. Helen Custer's book about NTFS [3] discusses NTFS at a high level, but we needed detailed information about its layout. For this, we used the documentation and source code for the Linux NTFS driver.This documentation is at http://www.via.ecp.fr/∼regis/ntfs/new/, and the source code is at http://www.informatik.hu-berlin.de/∼loewis/ntfs/.

We learned that essentially all the data we needed 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, there are at least three problems with recording the metadata by simply dumping this file. First, the file is sparse, because (1) many files' attributes do not use an entire record, and (2) many records are unused, having been allocated to files that have since been deleted. Second, an attribute can be *non-resident*, meaning that it is 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 might also record file contents; this would violate the confidentiality of our users' data.

This led us to the following approach. We do a depth-first search of the directory structure of each NTFS partition and, for each file found, we find and record certain non-data attributes of that file. Finding the metadata for a file requires knowing its *file number*, the number of the MFT record containing that file's attributes. The partition root always has file number 5.

We still have not explained how one reads directly from a disk, or how one finds specific MFT records. To read a raw disk on Windows NT, a user-mode program opens a file called "\\.\X:", replacing "X" with the appropriate drive letter. The first file block contains useful information: the size of a block (the 2-byte 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 one can locate any MFT record. Parsing a file's MFT record reveals all the file's attributes. (This is not difficult, especially if one judiciously copies sections of the Linux NTFS driver code and reads the Linux NTFS documentation described above.) If the file is actually a directory, one can parse its index allocation attribute to find its subfiles' file numbers.

To determine what raw disk device a DOS disk name like "X:" corresponds to, we call QueryDosDevice(). It takes a DOS disk name and returns the corresponding raw disk name, such as "\Device\Harddisk1\Partition3."

```
#define TAG_GET_PROC_THREAD_INFO     5
#define FIRST_GUESS_AT_PT_INFO_SIZE  8192
#define INCREMENT_FOR_PT_INFO_SIZE   1024

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_PT_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_PT_INFO_SIZE);
    else
      return NULL;
  }
  return NULL;
}
```

Figure 12: This function returns a buffer containing a sequence of process information structures, one for each process. It puts the length of the returned buffer in bytesReturnedPtr. The caller of this function is responsible for freeing the returned buffer if it is not NULL.

9 Miscellaneous Features of VTrace

9.1 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 set a logging function to be called 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 is no documented way to do this from kernel mode. Fortunately, there was a USENET article on comp.os.ms-windows.programmer.nt.kernel-mode by Fizal Khan describing how do this with the undocumented function ZwQuerySystemInformation(). This function has the following prototype:

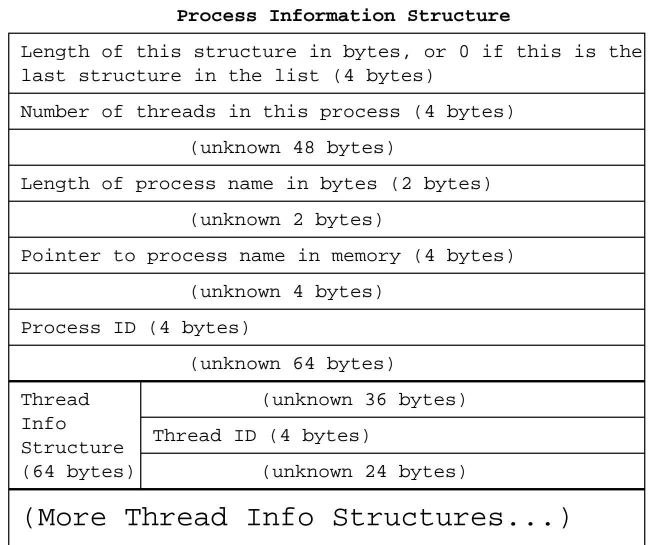


Figure 13: NtQuerySystemInformation() returns a list of process information structures, each of which looks like this.

```
unsigned long ZwQuerySystemInformation
 (ULONG tag, VOID *buffer, ULONG bufSize,
  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 12 shows how we use this function to get a sequence of process information structures, one for each process. Figure 13 illustrates the contents of each of these structures.

We also use this undocumented feature to obtain and log the name of a process when we are notified of it starting, since the notification only tells us the process ID.

9.2 Idle timer

To keep trace file sizes down, the tracer should automatically stop logging when the user is idle for 10 minutes. Because we were logging keyboard and mouse messages, we could determine when the user was active; this permitted the following approach.

When the system starts up, we initialize a timer to go off in ten minutes, using code like that in Figure 14. In this code, the constant is -6 billion, meaning 6 billion 100-ns units (10 minutes) from now (negativeness indicates relative time). The call to KeInitializeDpc() initializes a deferred procedure call object by binding it to the function UserGoesIdleRoutine(). KeInitializeTimer() associates the timer with that deferred procedure call, so that its associated function is executed when the timer goes off. Finally, KeSetTimer() initializes the timer to go off ten minutes later. When we detect user activity, we repeat the call to KeSetTimer(), delaying when the timer will go off until ten minutes after *then*.

```
void InitializeUserInactivityWatch (void)
{
  LARGE_INTEGER tenMinsFromNow =
    { 0x9A5F4400, -2 };

  KeInitializeDpc(&globals.userGoesIdleDpc,
                  &UserGoesIdleRoutine, NULL);
  KeInitializeTimer(&globals.userGoesIdleTimer);
  KeSetTimer(&globals.userGoesIdleTimer,
             tenMinsFromNow,
             &globals.userGoesIdleDpc);
}
```

Figure 14: This code initializes a timer. If this timer is not canceled, it will call UserGoesIdleRoutine 10 minutes from now.

9.3 Disabling drivers at startup

Many of VTrace's drivers must be started automatically at startup time for them to work. This means that if they cause problems, it may be impossible to remove them by any means short of reinstalling the operating system. Thus, it is useful to be able to disable the drivers at startup.

In order to do this, we need access in the very early stages of startup to some state that the user can control. About the only thing the user can indicate to the system at this stage is what boot configuration to use. For instance, the user can choose to boot with the "last known good" configuration, meaning the most recent registry configuration that led to a successful boot. This is a natural signal we can use to decide to turn off VTrace.

To determine which configuration is in use, we open the registry key HKEY_LOCAL_MACHINE\System\Select and read the Current and LastKnownGood values. Each of these is an index into the list of registry configurations. If the current configuration in use is the same as the last known good configuration, we know that the user has chosen the last known good configuration and we turn off all components of VTrace.

9.4 User-level service

Some of VTrace's general operations are easier and safer to implement at user level than at kernel level. For this reason, VTrace includes a user-level service, VTrcSrvc, that is launched at startup and runs in the background to perform the following two operations: (1) After the user has been idle for 2 hours, it turns off tracing, takes a metadata snapshot, compresses all the trace and metadata files collected, uploads those files to our web site, deletes them from the local hard drive, and turns tracing on again. It waits 24 hours before doing any of this again. (2) Whenever a new user logs in or the logger signals that a new log file has started, it generates a log entry describing the current user's name.

Paula Tomlinson's article [15] describes in detail how to write and install a user-level service, so we describe it only

```
void main (void)
{
  SERVICE_TABLE_ENTRY ServiceTable[] =
    { { "VTrcSrvc",
        (LPSERVICE_MAIN_FUNCTION) &ServiceMain },
      { NULL, NULL } };
  StartServiceCtrlDispatcher(ServiceTable);
}
```

Figure 15: This is the main routine of VTrace's service, VTrcSrvc.

```
__asm {
  _emit 0x0F      ; Byte 1 of the RDTSC instruction
  _emit 0x31      ; Byte 2 of the RDTSC instruction
  mov ebx, bufPtr ; Put the addr where we want the
                  ; timestamp in register ebx.
  mov [ebx], eax  ; Save low 4 bytes of timestamp.
}
```

Figure 16: This code reads the Pentium cycle counter by invoking assembler in C.

briefly here. The main() routine initializes a table of service table entries and dispatches them. Ours looks like Figure 15. The service main function, in our case called ServiceMain(), registers a handler for service control messages (such as pause and stop), initializes other global state, launches threads to perform the service's tasks, then waits on an event set when a stop message is received. Throughout the initialization process, it calls SetServiceStatus() to send messages to the service control manager indicating how far along it is.

One of the service control messages VTrcSrvc is programmed to respond to is one we made up called SERVICE_CONTROL_RELOAD_REGISTRY. When VTrcSrvc receives this message, it rechecks the VTrace parameters in the registry and starts using the new values if they have changed. This allows the utility that changes VTrace parameters to make those changes go into effect immediately without waiting for the next reboot.

9.5 Pentium cycle counter

To get accurate time stamps on each of our trace events, we use the Pentium cycle counter. This counter contains the number of cycles that have passed since the computer was started up. It is accessed via the RDTSC instruction, which can be coded in C by invoking assembler as in Figure 16.

VTrace only needs the low four bytes of the counter for the following reason. Given the low four bytes of two consecutive event times, one can determine, modulo 2^{32}, how many cycles separated the events. The logger automatically places a null event in the log every $0.75 \cdot 2^{32}$ cycles, ensuring that no two consecutive events are separated by 2^{32} cycles or more. Thus, the time between any two consecutive events is unambiguous. (If the high four bytes were needed as well, they could be found in register EDX.)

10 Windows 2000

Windows 2000 is substantially similar to Windows NT, but different enough that porting VTrace to it required some effort. In this section, we briefly describe some of the changes we made so 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 any attempt to overwrite kernel code in memory causes a system crash. Since our method of hooking context switches requires that we overwrite the first instruction of the context-swap code in memory, this causes problems for VTrace. The solution is to turn off kernel-mode write protection by opening the registry key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Memory Management, and creating in it the value EnforceWriteProtection=0. A USENET poster described this feature when discussing how to get a certain debugger to work with Windows 2000.

The context swap routine is different in Windows 2000, so the tracer must look for this new routine in memory.

Windows 2000 expects 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 examples showing how to do this.

At the end of Section 8, we discussed how to determine the disk and partition numbers of a given disk, such as “X:”. Unfortunately, this method does not work in Windows 2000, and the Windows 2000 method does not work in Windows NT. In Windows 2000, you must use a new device I/O control code called 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 changed from Figure 13. The “unknown 64 bytes” in the header are actually 112 bytes long in Windows 2000.

Operation	Time without VTrace	Time with VTrace	Slow-down
Read an uncached 32KB file	9.16 ms	9.17 ms	0.1%
Write 1KB file (write-thru)	25.05 ms	25.05 ms	0%
Read 32 KB direct from disk	9.17 ms	9.17 ms	0%
Copy a 32 KB file locally	6.29 ms	6.57 ms	4.5%
Copy a 32 KB file remotely	27.73 ms	35.07 ms	26.4%
ZwFlushInstructionCache()	2.78 μs	3.72μs	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%

Figure 17: These benchmark result means show how much VTrace slows down various operations.

11 Benchmarks

Considering all the tracing that VTrace does, it is important to determine how much it slows down the system. We wanted to make the overhead unnoticeable, so users would let us install it on their systems. By this measure, we succeeded, since none of our users has ever complained about performance suffering.

That said, it can be hard for users to detect subtle differences, especially on today’s fast machines. So we designed various benchmarks to show the effect of running VTrace. We ran each of these benchmarks on our PC, which has a 450 MHz Pentium III, is connected to a 100 Mbps switched Ethernet, has 128 MB of memory, has 10 GB divided among three SCSI disks, and is running Windows NT 4.0 with Service Pack 6a. We ran each benchmark (other than the compilation and document format benchmarks, which take too long) enough times that the 95% confidence interval about the sample mean included no values more than 0.1% away from the sample 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 17 shows the results. We see that VTrace has almost no effect on simple reads and writes, since there is little 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 log entries. As you can see, this is substantial for the latter two functions since they do little but are at user level, and furthermore because each call requires two log entries: one for the initiation of the function and one for its completion. Finally, one can see the “big picture” from the two application benchmarks, which show that VTrace makes a 10-second compilation take 13.4% longer and a 2-second document formatting take 5.3% longer.

These benchmarks suggest that the biggest area for improvement is the overhead of tracing user-level events. It would thus seem 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. Thus, in the final version of VTrace, we perform all logging at kernel level.

12 Similar Software

VTrace is not the first piece of software to modify the operating system (without recompiling it) on machines used for normal operation. WMonitor [16] uses the Windows message hook facility and installable file system support to trace application messages and file activity in Windows 95. SLIC [4] uses interposition to intercept system calls and signals in Solaris 2.5. KernInst [14] performs a structural analysis on the Solaris 2.5.1 kernel running on an UltraSPARC so that it can insert code at almost any point in the kernel without rebooting and without disturbing any live registers. COLA [7], like our Win32 system call hooking method, looks into each library loaded into a UNIX application to find all the points at which system calls are made, in order to intercept those system calls at those points. Instrumented Connectors [1] is a general system you can use to wrap your own function around any function exported by a Windows dynamically linked library. Michael Jones created an interposition agents toolkit [5] that expresses the objects in the 4.3BSD operating system as C++ classes, so that you can extend their functionality by writing derived classes. And there are many other examples, including various commercial virus checkers and disk compression utilities.

13 Summary

Writing the tracer VTrace for Windows NT/2000 was a difficult task, because of both the inherent difficulty of 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. This paper has described the major techniques we used in doing so.

Writing the tracer involved a lot of driver programming to implement special device types. The heart of our tracer is the logger device, which processes requests to add events to the log. The other devices we implemented were filter devices, which layer themselves above existing devices in order to intercept and log I/O requests destined for them. We filter file systems, the keyboard, disk partitions, and network transport layers.

We also did a substantial amount of hooking system functions. We developed a technique for intercepting calls to the context switch function. We also adapted to our purposes some published hacks for intercepting NT kernel and Win32 system calls.

We had to do many other things for our tracer, several of which we describe. We designed a file system metadata parser, used a technique to get a list of existing processes and threads, implemented an idle timer, and included a user-level service, among other things.

Although VTrace incurs overhead on the system, this overhead is relatively low considering how much it traces. No user has complained about the load VTrace places on the system. Logging common file operations incurs very little overhead. Benchmarks measuring the effect on realistic batch workloads show only a 5–13% increase in execution time.

It is our hope that the techniques we describe, as well as the references we give to more detailed descriptions of related techniques, will be helpful to future Windows NT/2000 system-level programmers. These techniques can be used for tracing many things VTrace does not trace, and for many things besides tracing.

References

[1] Balzer, R. and Goldman, N. Mediating Connectors. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, Austin, TX, 73–77, May/June 1999.

[2] Custer, H. *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[3] Custer, H. *Inside the Windows NT File System*, Microsoft Press, Redmond, WA, 1994.

[4] Ghormley, D., Petrou, D., Rodrigues, S., and Anderson, T. SLIC: an Extensibility System for Commodity Operating Systems. *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 39–52, June 1998.

[5] Jones, M. Interposition Agents: Transparently Interposing User Code at the System Interface. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Ashville, NC, 80–93, December 1993.

[6] Kath, R. *The Portable Executable File Format, from Top to Bottom*, Microsoft Developer Network Technology Group Technical Report, June, 1993. Available from http://premium.microsoft.com/msdn/library/techartmsdn_pefile.htm.

[7] Krell, E. and Krishnamurthy, B. COLA: Customized Overlaying. *Proceedings of the USENIX Winter 1992 Technical Conference*, San Francisco, CA, 3–7, January 1992.

[8] Nagar, R. *Windows NT File System Internals*, O'Reilly and Associates, Inc., Sebastopol, CA, 1997.

[9] Pietrek, M. Peering inside the PE: a tour of the Win32 portable executable file format. *Microsoft Systems Journal*, 9(3):15–32, March 1994.

[10] Pietrek, M. Learn system-level Win32 coding techniques by writing an API spy program. *Microsoft Systems Journal*, 9(12):17–38, December 1994.

[11] Russinovich, M. and Cogswell, B. Windows NT system-call hooking. *Dr. Dobb's Journal*, 22(1):42–46, January 1997.

[12] Russinovich, M. and Cogswell, B. Examining the Windows NT filesystem. *Dr. Dobb's Journal*, 22(2):42–50, February 1997.

[13] Schildt, H. *Windows NT Programming from the Ground Up*, Osborne/McGraw-Hill, Berkeley, CA, 1997.

[14] Tamches, A. and Miller, B. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 117–130, February 1999.

[15] Tomlinson, P. How to Write an NT Service. *Windows Developer's Journal*, 7(2):6–18, February 1996.

[16] Zhou, M. and Smith, A. Tracing Windows 95. *Technical Report UCB/CSD-99-1037*, Computer Science Division, EECS, University of California at Berkeley, January 1999. Available from http://www.ncstrl.org.