The following paper was originally published in the
Proceedings of the USENIX Windows NT Workshop
Seattle, Washington, August 1997

Creating User-Mode Device Drivers with a Proxy

Galen C. Hunt
Department of Computer Science
University of Rochester
Rochester, NY

Creating User-Mode Device Drivers with a Proxy

Galen C. Hunt
gchunt@cs.rochester.edu

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

Abstract

Writing Windows NT device drivers can be a daunting task. Device drivers must be fully re-entrant, must use only limited resources and must be created with special development environments. Executing device drivers in user-mode offers significant coding advantages. User-mode device drivers have access to all user-mode libraries and applications. They can be developed using standard development tools and debugged on a single machine. Using the Proxy Driver to retrieve I/O requests from the kernel, user-mode drivers can export full device services to the kernel and applications. User-mode device drivers offer enormous flexibility for emulating devices and experimenting with new file systems. Experimental results show that in many cases, the overhead of moving to user-mode for processing I/O can be masked by the inherent costs of accessing physical devices.

1. Introduction

The creation of device drivers is one of the most difficult challenges facing Windows NT developers. Device drivers are generally written with development environments and debuggers that differ from those used to create other NT programs. Perhaps most challenging to many developers, NT device drivers must be fully re-entrant and must not block.

NT file systems, a special class of NT device drivers, are particularly complex because the developer must anticipate interconnections between the NT Cache Manager, the NT Memory Manager and the file system.

Writing device drivers is complicated because drivers operate as kernel-mode programs. Device drivers have limited access to other OS services and must be conscious of kernel paging demands. Debugging kernel-mode device drivers normally requires two machines: one for the driver and another for the debugger.

Device drivers run in kernel mode to optimize system performance. Kernel-mode device drivers have full access to hardware resources and the data of user-mode programs. From kernel mode, device drivers can exploit operations such as DMA and page sharing to transfer data directly into application address spaces. Device drivers are placed in the kernel to minimize the number of times the CPU must cross the user/kernel boundary.

User-mode device drivers offer significant development advantages over kernel-mode drivers with some loss in performance and direct access to hardware. User-mode drivers need not be re-entrant. They have full access to user-mode libraries and applications. User-mode drivers can be created with standard development environments, including high-level languages such as Java or Visual Basic. User-mode drivers can be tested and debugged on a single machine without special tools.

User-mode device drivers are especially useful for emulating non-existent devices or implementing experimental systems. They offer significant advantages in cases where their additional overhead can be masked by I/O latency. Particularly for file system development, user-mode device drivers give the developer great flexibility for developing and designing new systems.

The next section describes the Proxy Driver system for creating user-mode device drivers. Section 3 contains descriptions of several sample user-mode device drivers. Section 4 shows that for typical scenarios user-mode device drivers offer performance similar to kernel-mode drivers. Section 5 lists related work, and the last section concludes with a summary and suggestions for future work.

2. The Proxy Driver

Windows NT I/O is packet driven. Upon entering the NT executive, individual I/O requests are encoded in an I/O Request Packet (IRP). IRPs often pass through multiple drivers; winding, for example, from the

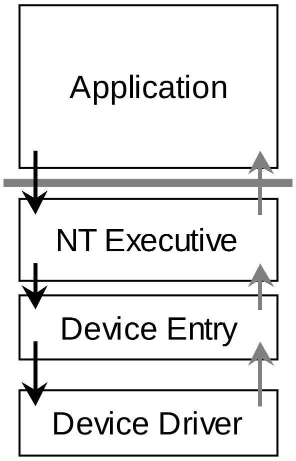

Figure 1, Composition of a Kernel-Mode Driver.

executive to a file system driver to a hard disk driver and back. IRPs are processed asynchronously.

Execution enters a driver through a device entry that represents a logical device. Figure 1 shows the composition of a kernel-mode device driver. I/O requests from the application are converted to IRPs in the NT Executive and passed to the corresponding driver through the device entry.

We implement user-mode drivers using a kernel-mode *Proxy Driver*, see Figure 2. The Proxy Driver acts as a kernel agent for user-mode device drivers. User-mode device drivers connect to the Proxy Driver using a device `open` request (through the File System API) to a special device entry called the *host entry*. The host entry is the doorway to the Proxy Driver's API. The user-mode driver registers with Proxy Driver, informing it of the I/O requests it would like to process. In response, the Proxy Driver creates a new entry in the device table for the user-mode driver called a *stub entry*. The kernel and other device drivers access the user-mode driver through its stub entry. Hidden behind the stub entry, the user-mode device appears as a kernel-mode device. The user-mode driver communicates with the Proxy Driver via `read` and `write` operations on the host device entry.

When an application or the kernel makes an I/O request relevant to the user-mode driver, the IRP is routed through the kernel to the Proxy Driver using the stub entry. The Proxy Driver marshals the IRP and forwards it to the user-mode driver through the host entry. The user-mode driver first processes the request then returns the response to the Proxy Driver through the host entry. The Proxy Driver returns the completed IRP to the kernel through the stub entry.

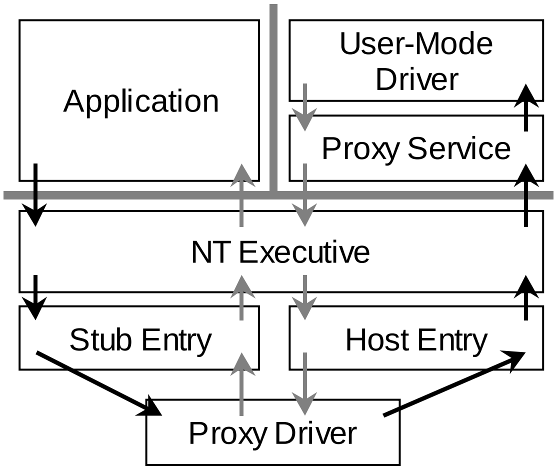

Figure 2, Composition of a User-Mode Driver. The kernel-mode proxy driver passes IRPs to the User-Mode driver through a host device entry and a COM service.

To simplify user-mode driver development even further, drivers are implemented as Component Object Model (COM) components. As shown in Figure 2, the Proxy Service sits between the NT Executive and the user-mode device driver. All user-mode device drivers share a single instantiation of the Proxy Driver and Proxy Service. The Proxy Service converts incoming IRPs to calls on the COM interfaces of user-mode drivers. User-mode drivers export COM interfaces for I/O requests they support. Drivers can even inherit functionality from other drivers through COM aggregation. Appendix A lists the IDL definition for `IDeviceFileSink`, the interface used to deliver the most common file IRPs.

The Proxy Service operates under control of the Windows NT Service Manager [4]. It can be stopped or started through the NT Service control panel. At startup, the Proxy Service dynamically loads the Proxy Driver into the kernel. It then looks for user-mode device driver components in the system registry under the key `HKEY_LOCAL_MACHINE\SOFTWARE-\URCS\ProxyDevices`. For each registered component, the service opens a handle on the Proxy Driver and creates a stub entry in the NT Executive. Information stored in the system registry determines where the user-mode device will appear in the Windows NT I/O namespace. System administrators control access to the Proxy Driver by restricting access to the `ProxyDevices` registry key.

The Proxy Driver does not support the NT Fast I/O path or filter drivers. The Fast I/O path is a mechanism by which the NT Executive passes non-

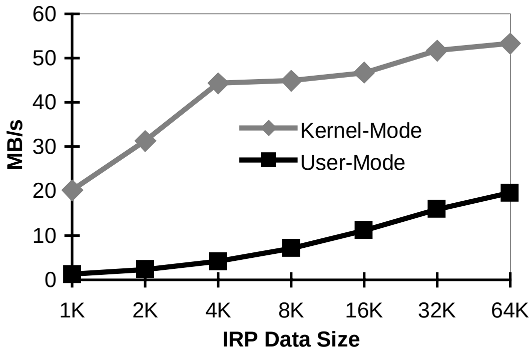

Figure 3, Raw Driver Throughput. IRPs are completed with no side effects.

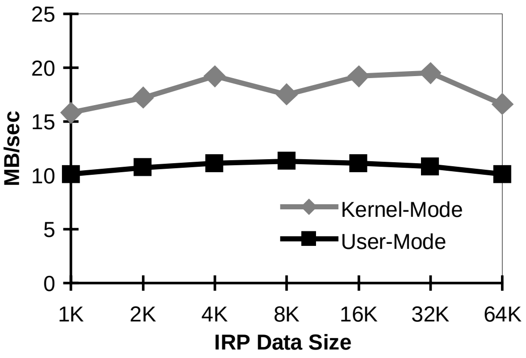

Figure 4, Writing data into either a kernel-mode RAM disk or a user-mode virtual memory disk.

blocking I/O requests directly to device drivers without creating an IRP. Filter drivers are drivers that sit between the NT executive and another driver.

The user-mode driver lives in an ideal world. It receives only IRPs destined for its device entry, the stub entry. Because the user-mode driver runs in user-mode, it has access to all traditional user-mode APIs and resources. Finally, because it receives IRPs from the Proxy Driver only after a `read`, the user-mode driver can execute sequentially as a single-threaded program; it need not be re-entrant. Sophisticated user-mode drivers can handle multiple concurrent requests either through asynchronous I/O calls on the host entry or via multiple threads of execution. User-mode driver programmers can choose a level of sophistication appropriate for their domain.

3. Example Drivers

We have created a number of example user-mode drivers. Each is a COM component.

- `rawdev`: A null device for testing user-mode driver performance, `rawdev` completes all IRPs successfully with no side effects.
- `vmdisk`: Similar to a RAM disk, the Virtual-Memory Disk uses a region of virtual memory to emulate a physical disk.
- `efs`: The Echo File System acts as a proxy for another file system. It converts incoming IRPs to Win32 File API calls on the "echoed" file system.
- `ftpfs`: The FTP File System mounts a remote FTP server as a local file system. Incoming IRPs are converted to outgoing FTP requests using the WinInet APIs.

4. Performance

User-mode device drivers trade performance for code simplicity. Whereas a request on a kernel-mode driver would cross the user/kernel boundary twice, each request on a user-mode driver must cross the user/kernel boundary at least four times. In this section we present performance results gathered from the Proxy Driver implementation on Windows NT 4.0 Workstation using a 133MHz Pentium processor.

In Figure 3, raw driver throughput is shown for two functionally equivalent device drivers: one a kernel-mode driver and the other a user-mode driver using the Proxy Driver. Each driver reads, but doesn't copy, the data in an incoming IRP. The great disparity between performance for the two drivers is a result of two factors. First, requests for the user-mode driver must cross the user/kernel boundary twice as often as requests for the equivalent kernel-mode driver. Second, although kernel-mode drivers can directly access IRP data, data bound for user-mode drivers must be copied from the stub IRP to the host IRP. As would be expected, total throughput increases with larger packets as the cost of crossing the user/kernel boundary is amortized. Kernel-mode throughput decreases below 4K as sub-page requests require additional alignment processing.

Figure 4 compares the difference in performance for an application writing to a kernel-mode RAM disk device versus a user-mode virtual memory disk device. Performance for the user-mode driver is better than that in Figure 3 for small requests due to aggregation by the NT Cache Manager. Overall performance for the kernel-mode driver decreases from Figure 3 because the driver must now copy data from the IRP to

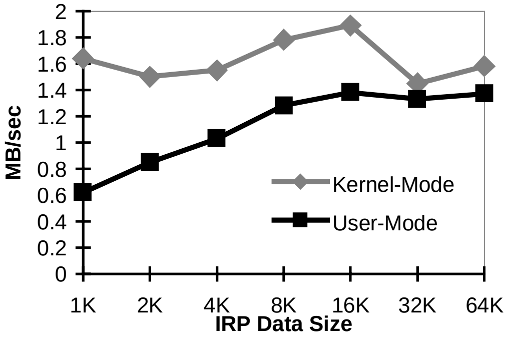

Figure 5, File system throughput.

the RAM disk. Note that the Proxy Driver was already making a copy from the kernel's address space into that of the driver.

The relative performance of file system drivers is shown in Figure 5. The kernel-mode file system is Microsoft's NTFS on a physical disk. The user-mode file system, `efs`, forwards I/O requests to NTFS using the Win32 file APIs. Total system throughput is reduced primarily as a result of accessing the physical disk. The difference in performance between the two drivers is essentially the cost of using a user-mode device driver.

Figure 5 demonstrates that particularly when using an external device, such as a disk or network, the impact of using a user-mode driver is minimal. By exploiting full access to user-mode resources, user-mode device drivers can achieve better performance than kernel-mode drivers. The `ftpfs` driver, for example, shares the WinInet cache with user-mode applications, such as Internet Explorer, to reduce network traffic and optimize performance.

5. Related Work

Other researchers have noted the benefits of supporting user-mode device drivers.

Frigate [8] supports user-mode file servers using a dispatch layer in the UCLA Stackable Layers file system [6]. Applications issue file I/O requests either through kernel calls to user-mode servers or directly through a server backdoor. A contemporary of the Proxy Driver, Frigate requires a kernel supporting UCLA Stackable Layers. Frigate has been used to implement an Enigma encryption layer above SunOS file systems.

Bershad and Pinkerton's *watchdogs* [1] are user-mode file-system components. A watchdog is attached to a particular point in the file-system namespace. File I/O requests at the point of the watchdog are intercepted. The watchdog may refuse the I/O operation, perform it on behalf of the system, or return the request to the underlying file system. Watchdogs are intended to allow unsophisticated users to modify the behavior of isolated file system functions. Bershad and Pinkerton use a special communication channel to pass I/O requests to watchdogs.

The HURD system [2] supports the concept of user-mode file systems via a mechanism called *translators*. All files requests are sent through Mach ports. User file systems are just providers of "file" ports. Sample systems proposed include a file system level FTP client and a `/proc` file system similar to [7].

Unlike watchdogs and the HURD system, the Proxy Driver makes no modifications to the NT kernel. It exploits the NT I/O architecture, particularly IRPs, to provide simple, efficient user-mode device drivers for either file system or device operations.

The Semantic File System (SFS) [5] is a user-mode file system that creates dynamic directories based on indexed search criteria. For example, the SFS `/subject:report` directory contains one file for each email message with the word "report" in the subject line. Client machines connect to an SFS server using the NFS protocol.

Similar in principle to our `ftpfs`, the Alex file system [3] provides access to FTP archives using the NFS protocol. NFS requests on the Alex server are converted to FTP requests and forwarded to the appropriate server.

SNFS [9] is a generalized NFS server that supports an internal Scheme interpreter. File requests on the NFS server are processed through user-loaded modules written in a version of Scheme. Proposed modules include union file systems, copy-on-write file systems, and FTP and HTTP file systems.

The Proxy Driver is more efficient than NFS-based user-mode file systems. By exporting native IRPs, the Proxy Driver avoids costly re-marshaling to a foreign I/O model such as NFS.

6. Conclusions

We have described our Proxy Driver system for creating user-mode device drivers. The Proxy Driver resides in the kernel and passes I/O requests to user-mode drivers through a host device entry. User-mode drivers are much easier to write and debug than kernel-mode drivers. Although limited in scope to drivers needing no kernel-mode hardware access, user-mode drivers offer the programmer great flexibility. User-mode device drivers are particularly effective in cases where physical I/O dominates driver computation.

We are currently developing a toolkit for creating user-mode file system drivers. The toolkit will provide COM components for volatile and persistent cache management, name-space manipulation, and file system layering. With the toolkit, developers will be able to create simple file watchdogs or fully functional file systems in as little as a few hundred lines of code.

Acknowledgments

During the development of the Proxy Driver, the author was supported by a research fellowship from Microsoft Corporation.

Bibliography

[1] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *Computing Systems*, Spring, 1988.

[2] M. Bushnell. Towards a New Strategy of OS Design. In *GNU's Bulletin*, January 1994.

[3] V. Cate. Alex- a Global Filesystem. In *Proceedings of the USENIX File System Workshop*, pp. 1-11. Ann Arbor, MI, May 1992.

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

[5] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic File Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 16-25. Pacific Grove, CA, October 1991.

[6] J. S. Heidemann and G. J. Popek. File-System Development with Stackable Layers. In *ACM Transactions on Computer Systems*, vol. 12(1), pp. 58-89, 1994.

[7] T. J. Killian. Processes as Files. In *Proceedings of the USENIX Software Tools Users Group Summer Conference*, pp. 203-207, 12-15 June 1984.

[8] T. H. Kim and G. J. Popek. Frigate: An Object-Oriented File System for Ordinary Users. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pp. 115-129. Portland, OR, June 1997.

[9] T. Lord. Extensible Linux NFS server soon to be available. In *comp.os.linux.announce*, September 1996.

Appendix A

The `IDeviceFileSink` interface is used to deliver common file IRPs to user-mode drivers.

```
interface IDeviceFileSink : IUnknown
{
   HRESULT Create(
       [in] IDevIrp *pIrp,
       [in] IDevSecurityContext
*pCtxt,
       ULONG Disposition,
       ULONG Options,
       ULONG FileAttributes,
       ULONG ShareAccess,
       ULONG EaLength,
       LARGE_INTEGER AllocationSize);

   HRESULT Cleanup(
       [in] IDevIrp *pIrp);

   HRESULT Close(
       [in] IDevIrp *pIrp);

   HRESULT Shutdown(
       [in] IDevIrp *pIrp);

   HRESULT Read(
       [in] IDevIrp *pIrp,
       LARGE_INTEGER ByteOffset,
       ULONG Length,
       ULONG Key);

   HRESULT Write(
       [in] IDevIrp *pIrp,
       LARGE_INTEGER ByteOffset,
       ULONG Length,
       ULONG Key);

   HRESULT DeviceControl(
       [in] IDevIrp *pIrp,
       ULONG IoControlCode,
       ULONG InputBufferLength,
       ULONG OutputBufferLength);

   HRESULT QueryInformation(
       [in] IDevIrp *pIrp,
       ULONG Length,
       FILE_INFORMATION_CLASS
FIClass);

   HRESULT SetInformation(
       [in] IDevIrp *pIrp,
       ULONG Length,
       FILE_INFORMATION_CLASS FIClass,
       [in] IDevFileObject *pFileObj,
       BOOL ReplaceIfExists,
       BOOL AdvanceOnly,
       ULONG ClusterCount,
       ULONG DeleteHandle);

   HRESULT FlushBuffers(
       [in] IDevIrp *pIrp);
};
```