Porting the Arla file system to Windows NT

Magnus Ahltorp
Royal Institute of Technology
ahltorp@nada.kth.se
Love Hörnquist-Åstrand
Royal Institute of Technology
lha@nada.kth.se
Assar Westerlund
Swedish Institute of Computer Science
assar@sics.se

Abstract

This paper describes how we ported the Arla filesystem to Windows NT/2000. Windows is very different from the platforms (different flavors of Unix) that Arla was written for before. Arla consists of a complex userland daemon (arlad) and a rather simple kernel-module (xfs).

Arlad needed very little work to be able to work on Windows because we used Cygwin. The Windows kernel-module was written from scratch under influence from the earlier Unix kernel-modules.

Since everthing was different it took very long time to complete the work with xfs. A problem is that there is little example code available to be used as reference and inspiration.

1 Introduction

Writing a kernel driver for a new operating system is quite an adventure. The basic principle of how a computer operates is the same, but almost everything else has changed. A new operating system is a new paradigm and there are new common constructs that one needs to learn. Often this feeling that a construct would be much easier to create with the previous operating system bubble to the surface of consciousness.

But there is also this new (sometimes scary) thought entering your mind: "Wow, this might be useful, almost beautiful.".

2 AFS and Coda

AFS[8] was developed at Carnegie Mellon University (CMU). AFS was created as part of a prototype computing enviroment for Universities in a collaborative effort between IBM and Carnegie Mellon.

Scalability is one of the main properties of AFS. It was designed to handle more then 5000 concurrent workstations. Other important features of AFS is location transparency, compatibility with existing operating system interfaces, and user mobility.

Security is an important design consideration, and the mechanisms for it do not assume that the workstations or the network are secure.

Coda[7] is designed with the same principles as AFS, but with emphasis of very high availability. This is done by using optimistic writing regarding consistency.

3 The Architecture of Arla

The Arla client is divided into two components, a kernel module (xfs) and a user-level daemon (arlad), see figure 1.

The kernel module is quite small and does not know very much about any particular file system. It is mainly designed to hook in a file system into the operating system. A simple way of doing this would be to forward all the requests that the particular file system receives to the daemon and let it handle them. This would, however, be detrimental to performance. Having to context switch at least two times for every file access (reading of attributes, reading of data, ...) would greatly slow down a file system. For this reason, the kernel module keeps a cache of accessed information and only relies on the user level part to service the cache misses. In this way, performance is quite close to and comparable to a "monolithic" file system, implemented entirely in the kernel [9, 10].

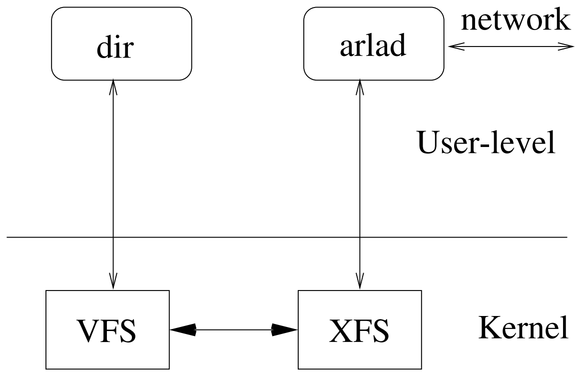

Figure 1: The architecture of arla

Apart from the two already mentioned interfaces (to the kernel and to arlad), there is also one interface for user programs, via the system call *pioctl*. This allows ordinary programs to control the behaviour of the client and do AFS-specific operations.

The arlad daemon keeps track of the on-disk cache, performs all the network communication, and protocol processing. It serves requests from the kernel module for data and also invalidates cached data in the kernel when necessary.

This separation has proved itself to be quite productive as it has enabled porting the kernel module without having to change much if any of the user-level parts. Development has also been simplified, since the user-level parts can benefit from the ordinary development tools usually not available for kernel development.

4 Porting arlad to Windows NT

Porting arlad to a new Unix dialect is quite simple. The only part that really needs porting is the context switch function for the *LWP* threads. There is also an implementation of LWP on top of pthreads, which simplifies that part of the porting. When this is done, arlad can be run in a command-line based test mode, which allows you to navigate the AFS tree and read files.

Windows is however quite different from any Unix dialect. All code in arlad that call Unix-specific functions or assume that functions work as in Unix would have to be modified. Instead of doing this, we opted for trying to use Cygwin[2, 11], an Unix-layer on top of Windows. This also had the benefit of making the development environment similar to Unix.

4.1 Cygwin

Cygwin is a Unix-emulation layer for Windows 9x and NT/2000. It basically implements all of the Unix interfaces that people and programs associate with Unix. This is quite a daunting task, since Windows has a mostly different architecture. Take the Unix system call `fork`, for example. There is no such service on Windows, the creation of a new process is done from a program on disk rather than based on a running process. Cygwin takes care of doing all this hard work. It also contains all the usual programs (bash, sed, make, gcc, ...).

Using Cygwin was not always a dance on roses. We have been affected by some bugs in rather fundamental functions. Somtimes these have been caused by us using functions in a way that the Cygwin developers had not forseen. But these bugs have been fixed in more recent versions, sometimes with our contributed bugfixes.

Without Cygwin, porting arlad to Windows would be a major (and mind-numbing) task.

4.2 Threading Issues

Arlad is a threaded program with a number of threads for doing background tasks and a pool of threads for handling requests from xfs. It uses the LWP thread package. LWP implements non-preemptive threads with a small assembly routine that does context switching (this routine has to be ported to every new architecture). The rest of LWP is portable C.

There is also a LWP-on-pthread implementation that can be used (useful when linking to other programs using threads). Since Cygwin at that time did not contain any pthread package itself, we added Windows Thread support to the LWP library.

Now there is almost pthread support in Cygwin, unfortunately it does not include support for pthread_cond, and since LWP-on-pthreads uses that, it is currently unusable. We expect that when pthread_cond appears in Cygwin, arlad will work with that too.

5 Introduction to the NT kernel

The Windows NT Executive (kernel) consists of several discrete subsystems. Each subsystem has quite strict boundaries due to the object-oriented nature of the executive. A simplified overview of the kernel-mode components is shown in figure 2.

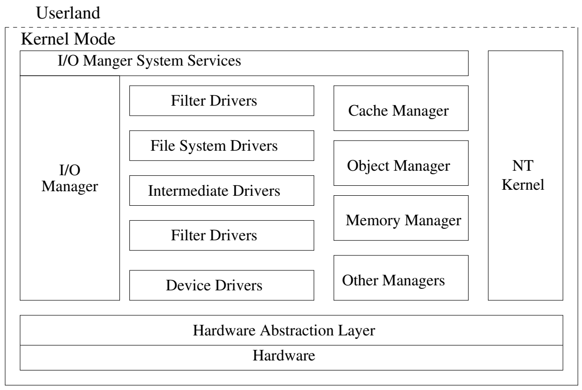

Figure 2: Simplified overview of NT kernel

5.1 The I/O Subsystem

The I/O Manager is responsible for the I/O Subsystem. The parts of the I/O Subsystem are filter drivers, file system drivers intermediate drivers, and device drivers. Filter drivers can insert themselves anywhere in the driver stack to add or remove functionality.

A typical example of a filter driver is an anti-virus program. Before the user is allowed to open a file, it is checked for viruses. An example of an intermediate driver is the SCSI subsystem.

The I/O manager's responsibilities are, dynamically load kernel-modules, provide an interface to kernel-modules, interact with the Cache Manager, and the Virtual Memory Manager.

Characteristic of the I/O subsystem is that is uses packets for I/O (a packet is called an I/O Request Packet, IRP). Each of the drivers is an object that has a predefined set of methods. The modules can only be accessed through this interface. This enables Windows to have layered drivers, which enable developers to add functionality in the middle of a driver stack.

5.2 Virtual Memory Manager

The Virtual Memory Manager (VMM) provides a demand paged virtual memory system. Each process has its own address space. Each virtual page is backed by a physical page. Each physical page can be paged out to a local file system.

The VMM also supports memory mapped files. The files can be arbitrarily large, but files larger then 2GB have to be mapped using partial views. There is also support for shared memory between processes.

5.3 Cache Manager

The Cache Manager adds features like read-ahead and write-behind for higher performance. It has, as the name suggests, the responsibility of caching objects in the kernel. To provide this service the Cache Manager has to make use of file system drivers and the VMM. The Cache Manager is a very integral part of Windows NT. Every time data from a file is accessed, the services of the Cache Manager are used.

5.4 Filesystem Literature and Documenation

To develop file systems on Windows NT/2000 you will need IFS[3] and NTDDK[4]. The NTDDK is available for free from Microsoft's website. The IFS-Kit, which is an extention to the NTDDK, is available for a $995. It consists of a headerfile (`ntifs.h`) and two sample drivers (FASTFAT and cdfs). There is no documentation included.

There is also an IFS-Kit written by Bo Branten called "GNU IFS-Kit" (it is released under the GNU GPL-license) that can be used as a replacement. Then the only thing missing is a sample file system. This is a major problem since all information is not contained in the `ntifs.h` header file.

Another good source of information is the File System Internals[6] book by Rajeev Nagar. This book is a must for those that wish to develop file systems for Windows. It explains how file systems drivers work in Windows NT, how they interact with the rest of the system, and how to write them. There are some errors in the book, but they are well published.

There is a slight problem in that there is not any documentation for Windows 2000. The only information available is a page on Microsoft's web site that lists the things that have changed between NT4 and 2000.

5.5 Anatomy of a Windows FSD

A Windows file system driver (FSD) has five important structures that it needs to handle. First is the File Control Block (FCB) (the Unix/Linux equivalent is struct vnode) that is needed to keep track of all individual files.

To keep current offset and other state there is a Context Control Block (CCB) (the Unix/Linux equivalent is a file system maintained part of struct file). The CCB is maintained by the FSD and is used for example to keep track of the last accessed entry in a directory. The third structure is an almost opaque structure called FILE_OBJECT, this structure in maintained by Windows for each file that is opened (by a user or the kernel). It is almost opaque since some of its fields are well defined and exported to the FSD's.

Another almost opaque structure is the Volume Parameter Block (VPB). The VPB associates the Physical Object to the Mounted volume. The VPB is never changed by the file system driver. Note that the FILE_OBJECT also contains a pointer to the Physical Device.

The last structure is the Volume Control Block (VCB) that is used for storing private data. This corresponds to the private part of struct mount (mnt_data) in Unix/Linux.

The structures are tied together as shown in figure 3.

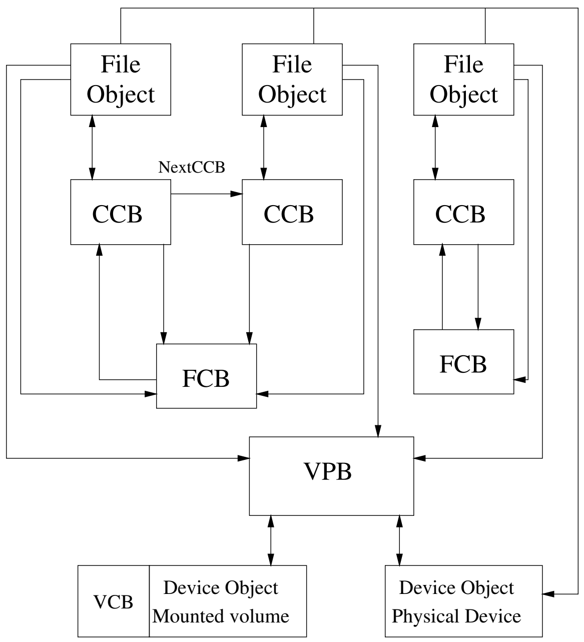

Figure 3: Windows Filesystem overview

5.6 I/O Requests

Requests to the FSD are passed as IRP's, an IRP is composed of the IRP header and several I/O stack locations. The FSD can either handle the IRP itself or pass it down to some other driver. If it can finish the request by itself, it completes the request with the IoCompleteRequest() function. In case the file system needs to take help of the lower level driver it can pass on the IRP with IoCallDriver() after setting up the next IRP stack location.

This is how file system filter drivers work, they hook into the driver stack and pass or modify the requests going down/up the stack. This way several filter drivers can be stacked on top of each other and implement different parts of the functionality.

6 Porting xfs

Porting xfs to Windows was quite straight forward, the only problem was that it was very time consuming. The starting point was a good understanding of the Unix code. This we had since we had been working a great deal with modifying and porting the Unix code. The question then was how this should fit in the Windows model of a kernel driver.

After that, pieces of code got implemented, and then bugs and bad interactions with other parts of the driver had to be fixed. In some rare cases we realized that we had misunderstood the driver and had to rewrite some other part of the code to make it fit.

One difference between Windows NT/2000 and Unix is that the filesystem has to do a lot of more work itself. For example there is no DNLC (Directory Name Lookup Cache) that will help you with looking up a name, you have to implement that yourself.

Another problem is that the major IRP's are often very overloaded and lots of code is required to figure out what the kernel wants returned. As an example, there are three different formats a directory can be returned in.

The Windows Device Driver Kit's documentation was quite useful and easy to use when writing code. The bad things about it was that it was often very verbose, and did not list all error codes that some function might return.

7 Issues Encountered

LWP under Unix relies on using select[1] to know what threads waiting for data should be awaken. Since under Windows you cannot call select on files, we had to solve the problem of communication between arlad and xfs in some other way.

To pass down messages to the kernel we send them with an IoDeviceControl to the xfs-device. To receive a message we created a special IoDeviceControl call that waited until there was something to report up to the userland. And since this operation blocks this thread's execution we had to implement this using a Windows thread.

Doing it this way created problems with the LWP package. The problem was that there was no easy way to wake up the LWP-thread that was sleeping in Cygwin's select. Instead of making LWP work with the new scheme we created a helper-program that received all requests from arlad via TCP and forwarded them with IoDeviceControl messages to the kernel.

[1] Wait on I/O file-handle

7.1 Interactions with AFS

Lots of Windows programs, particular file managers and such are prone to fetching the attributes of all objects. This interacts poorly with AFS where objects (mount points) can reside in other (possibly remote) cells. The typical example is the `/afs` top level directory that consists of all cells known to the local machine. Fetching the attributes of all of these is obviously a bad idea, but the program trying to show lots of information to the user does not realize this. This is also a problem with some non-Windows programs like the CDE file manager.

The way of handling this that we foresee is faking the attributes for the mount points. Just by saying that they are directories and that they have been modified a long time ago should be fine enough. And when the real information is fetched, it will replace the fake one.

8 Debugging Tools

There are two major debuggers available. They work almost in the same way, but there are some major differences. WinDBG is a graphical debugger. The second debugger, kd is textbased.

They are always quite well documented, and are almost easy to set up and use when you figure out how to get them to do what you want. Unfortunately they are not always that stable and they seem to have problems with keeping in sync with the target computer[2] when debugging over a serial-line.

9 Related Work

Coda[1] has been ported to Windows 9x and Windows NT. Unfortunately the client kernel module for Windows NT has never been released to the public. The Coda servers were ported to Windows NT with Cygwin.

Transarc has created a Windows client. It creates a SMB server and lets the local machine mount that. In other words, they did not implement it as a proper file system.

10 Future Work

Work needs to be done in the area of stability. Another important area is developing easy to use user-land tools, integrating these into the native environment of Windows so that regular Windows users feel at home.

[2]The computer on which you load your kernel module and later crash

Today we think that a client for Windows is just enough, servers will wait until we get stable servers for Unix. The framework for creating a fast fileserver on Windows exists, we just currently believe it is not worth the effort.

Since there is now a framework to enhance the application we anticipate that developing has much lower resistance. And since there are instructions on how to set up a development enviroment, it is much easier for third parties to help develop xfs.

To be able to improve performance and make everything work perfectly does require a deeper understanding of the Windows NT kernel than we currently have.

Making it really stable so that it can be unleashed on unsuspecting users needs to be done. Our experience tells us that this is hard but gives you lots of useful feedback. It's quite non-glamorous, boring, and very challenging work.

11 Summary

Porting Unix userland programs to Windows is very easy when you are allowed to use Cygwin. There are some minor bugs to be found in Cygwin, but those are slowly getting fixed.

Writing file system drivers for Windows it not an especially hard task, it just a very time consuming. Documentation will be an issue, but reading Rajeev Nagar's File System Interals will get you very far. There will be some missing pieces, but looking at an existing file system driver should fill those in.

12 More Information

See `http://www.stacken.kth.se/projekt/arla/`. We plan to release the Windows code once it get it get more stable.

13 Acknowledgements

Björn Grönvall started everything by writing xfs for SunOS. Robert Burgess has helped and encouraged writing Windows support. We must also thank Cygnus Solutions (a RedHat Company) for creating Cygwin and continually developing it.

References

[1] Peter J Braam, Michael J. Callahan, M. Satyanarayana, and Marc Schnieder, *Porting Coda Filesytem to Windows*, Usenix 1999 Freenix track, Monterey, California, USA, June, 1999.

[2] Cygwin, Cygnus Solutions, `http://sourceware.cygnus.com/cygwin/`.

[3] Installable FileSystem Kit, IFSKit , Microsoft Corporation, `http://www.microsoft.com/ddk/IFSkit/`.

[4] Windows NT/2000 Device Driver Kit, Microsoft Corporation, `http://www.microsoft.com/ddk/`.

[5] GNU IFSKit, Bo Branten, `http://www.acc.umu.se/~bosse/ntifs.h/`.

[6] Rajeev Nagar, *File System Interals - A developer's guide*, ISBN 1-56592-249-2, O'Reilly, Sebastopol, California, USA, 1997.

[7] Coda: A Resilient Distributed File System, Satyanarayanan, M., Kistler, J.J., Siegel, E.H. IEEE Workshop on Workstation Operating Systems, Cambridge, MA, USA, Nov 1987.

[8] Satyanarayanan, M., Howard, J.H., Nichols, D.A., Spector, A.Z., West, M.J. *The ITC Distributed File System: Principles and Design*, Proceedings of the Tenth ACM Symposium on Operating System Principles, Vol. 19, No. 5, Orcas Island, WA, USA, Dec. 1985.

[9] D. C. Steere, J. J. Kistler, M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun Vnode Interface*, In Proceedings of the USENIX Summer Technical Conference, 1990.

[10] Johan Danielsson and Assar Westerlund, *Arla—a free AFS client*, Proceedings USENIX Technical Conference, New Orleans (1998)

[11] Geoffrey J. Noer, *Cygwin: A Free Win32 Porting Layer for UNIX Applications*, USENIX Windows NT Workshop Proceedings, Seattle, Washington, USA, Aug 1998.