Portable Systems Group

NT OS/2 File System Design Note

Author: *Gary D. Kimura*

Revision 1.0, Sep 11, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction....1
1.1 Function of File System....1
1.2 NT OS/2 Environment for the File System....1

2. File System Operations....2
2.1 FSD/FSP Dispatch and Communication....2
2.2 File System Initialization....3
2.3 Before the First Operation on a Volume....3
2.4 Open and Create File Operations....5
2.5 Read, Write, Set, and Query File Operations....6
2.6 Close File Operation....7

3. Loose Ends....7

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

1. Introduction

This design note describes the implementation of an NT OS/2 file system. All NT OS/2 file systems share similar properties in their communication with the NT OS/2 I/O system, and their basic internal flow of control. The information presented here is based on knowledge gained while implementing the FAT file system for NT OS/2, but also contains information applicable to all NT OS/2 file systems.

Before reading this design note, the reader should be familiar with the "NT OS/2 I/O System Specification" document and the section within the "NT OS/2 Memory Management Design Note" regarding I/O support. The I/O system specification presents a high level view of the NT OS/2 user API calls and the support routines it provides for use by the various NT OS/2 file systems and device drivers. This design augments the I/O system document by describing in detail how a file system actually ties into the I/O system, in terms of the I/O system's major data structures and flow of control.

This design note presents a tour of the communication that occurs between the NT OS/2 I/O system and the file system. It is intended as a guide to file system programmers in understanding how the file system interacts with the I/O system, and avoids discussing file system internals, such as its internal data structures or its resource locking mechanism and granularity.

1.1 Function of File System

The task of the file system is to handle user (and system) generated requests to read and write files to or from a disk volume. The user passes all I/O requests to the I/O system (via calls such as NtReadFile) which in turn passes the request to the appropriate file system or device driver in a structure called an I/O request packet (IRP). The IRP contains a function code and parameters appropriate for the function. For example, to open a file the IRP contains a code indicating an open file operation and a file name parameter[1].

In the case of a file system the IRP also contains a pointer to the device object denoting the target device that the file system should direct all sector read and write requests towards. That is, to actually read a sector on a device the file

[1]The IRP actually contains additional open file parameters that are not discussed in this example.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

system calls back to the I/O system with information from the IRP indicating which device to read.

In this sense, the file system is not tied to any physical device. From the standpoint of the I/O system, a file system acts more like a filter. In NT OS/2, disk drivers only deal with reading and writing physical device sectors. The primary job of the file system is to add structure to this device. So a user or system request to manipulate a file goes from the I/O system to the file system where it is passed back to the I/O system as actual reads and writes of sectors on a disk.

1.2 NT OS/2 Environment for the File System

An NT OS/2 file system exports three entry points and these entry points are only called by the I/O system (i.e., a user never directly calls the file system). There is an initialization routine, a dispatch routine, and an unload routine.

The initialization routine is called at system start up during I/O initialization. This routine is responsible for creating the file system's device object and for initializing any global data structures and processes/threads used by the file system. The structure of a file system's device object is covered later in this design note.

The dispatch routine is called by the I/O system to process all IRPs targeted for the file system. These include requests to mount a volume, open or create a file, read and write to a file, and close a file. Logically, the dispatch routine either completes the I/O request and then returns control to the I/O system, or it returns to the I/O system immediately and indicates that the I/O will be completed at a later time (i.e., its return status is STATUS_PENDING).

The unload routine is called by the I/O system when the file system is being removed from a running NT OS/2 system. The unload routine is responsible for cleaning up all of its global data structures, processes, and threads. After calling unload, the file system is essentially unavailable for use until its initialization routine is called again.

2. File System Operations

This section describes the implementation details of an NT OS/2 file system. It describes the basic flow of control through the file system and its communication with the I/O system.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.1 FSD/FSP Dispatch and Communication

The term *File System Driver* (FSD) refers to a set of routines called by the I/O system (via the file system's dispatch routine). These are kernel-mode routines that execute in the same context in which they are called by the I/O system. That is, there is not a special process/thread or context switch associated with executing the FSD. Execution paths through the FSD should be fairly short and not require long waits (e.g., for a disk I/O to complete).

The workhorse of the file system is the *File System Process* (FSP). The term FSP refers to the set of threads executing within a dedicated file system process. Unlike the FSD, the FSP has the entire user address space available for use, and can dedicate threads to systematically attend to the tasks associated with an I/O request. An FSP thread can wait for I/O to complete and can easily maintain context unavailable to the FSD.

Figure 1 illustrates the relationship between the FSD and the FSP. The FSD is called by the I/O system with an IRP containing the requested operation. Then, based on the amount of work and context required by the request, the FSD either sends the necessary read and write requests to the target device object (as associated IRPs) or it enqueues the IRP to the work queue used by the FSP. If the request is queued to the FSP, the FSP dequeues the IRP and performs the work required to complete the request.

Figure 1 - FSD/FSP Layout

As a rule of thumb, file read and write requests can usually go through the FSD directly to the target device object. All other requests, such as file create, that require file system buffering or data structure editing go through the FSP.

Within a single file system, there are multiple work queues and an FSP thread dedicated to each queue. The FSP thread is a normal thread (i.e., it is merely the name of a thread that is handling work queue activity). There is a queue to

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

handle mount requests and a queue for each mounted volume. The construction and use of each work queue is described in the following subsections.

2.2 File System Initialization

The file system initialization procedure is called at system start up during I/O initialization. This procedure is responsible for creating the device object for the file system, creating the FSP process, and starting the FSP mount thread. It also initializes all of the global data structures used by the file system. It is called with a pointer to the driver object for the file system created by the I/O system.

The device object created at initialization is called the *file system device object* and contains the name of the file system (e.g., "\Fat") and a work queue. All mount requests are sent by the FSD dispatcher to this work queue and are processed by a dedicated FSP mount thread. Figure 2 illustrates the relationship between the file system device object and the FSP mount thread.

Figure 2 - File System Initialization

In summary, the initialization phase creates a file system device object and sets a pointer to it in the Driver Object. It also creates the FSP process and FSP mount thread. The FSP mount thread only processes requests that are placed in the file system device object's work queue, and the only type of requests valid in that work queue are mount volume requests.

2.3 Before the First Operation on a Volume

When a user calls NtCreateFile, one of the first things the I/O system must decide is whether the volume targeted by the request is mounted. If this is the first operation on the volume, then the volume is not mounted and the I/O system must send an IRP to the file system requesting that the volume be mounted. After the volume is mounted, the I/O system can then forward the create file request.

On a mount request, the I/O system calls the file system FSD dispatch routine passing as input parameters a pointer to the file system device object and an IRP

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

(See Figure 3a). The IRP contains two parameters of interest[2]. There is a pointer to a volume parameter block (VPB) and to a targeted device object. The VPB is a structure used to denote a disk volume. It will contain, after the volume is mounted, the volume label and its serial number. Within the system, there is one VPB for every mounted volume.

The targeted device object is the device object that is to be used by the file system when it issues a read or write sector request. The VPB also contains a pointer to a device object (this field is set by the I/O system) called the real device (not shown in Figure 3). In most cases, the targeted device object and real device are identical and simply denote the disk driver containing the volume (e.g., a floppy or hard disk). However, to accommodate multi-volume disk sets and disk strippers, the targeted device object and the real device can be different device objects. The file system only sends request to the targeted device object and never to the real device.

Figure 3a shows the parameters and data structures sent to the file system FSD dispatch routine on a mount request. When it is called, the FSD dispatcher will enqueue the mount request to the FSP mount thread's work queue. Note that the work queue is located by the FSD dispatcher via the file system device object which is passed in as an input parameter.

Figure 3 - Processing a Mount Request

To process the mount request, the FSP mount thread issues the necessary read sector requests to the targeted device object to decide if the volume belongs to this file system (e.g., whether it is a Fat file system or a Pinball file system). If the mount is successful, the file system creates a new device object for the volume. This device object replaces the device object originally referenced by the VPB. It contains a work queue for the volume, and a volume control block (VCB). Upon completion of a successful mount operation, the file system also inserts in the

[22]An IRP actually contains more than two parameters, but only these two are germane to this discussion.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

VPB the volume label and serial number. Figure 3b shows the results after a successful mount request.

The FSP mount thread also creates a new FSP volume thread to process subsequent I/O requests targeted for the volume. That is, when the I/O system calls the FSD dispatcher to do I/O to a mounted volume, it will pass as input a pointer to the volume device object and not the file system device object. The FSD dispatcher will then send the request to the work queue for the appropriate FSP volume thread.

A slightly different chain of events takes place if the volume being mounted has previously been mounted by the file system. That is, the file system already possesses a VCB for the volume. This situation can occur with removable media where the I/O system is attempting an operation on what it thinks is a new volume, when in reality the volume was mounted in a different drive at an earlier time.

To handle the remount case, the file system will keep track of every device object that is mounted and search this list whenever a new mount request is processed. If a match with a previous volume is found, the file system will then simply replace the new VPB it is given as input with the old VPB. This simply requires changing the real device pointer found in the VPB and the target device object pointer found in the old VCB. The remount process is illustrated in Figure 4.

Figure 4 - Volume Remount

In summary, the file system's mount procedure is used to establish that the target device object contains a proper file system structure. If the file system recognizes the on disk structure, it creates a new device object for the volume in place of the file system device object, sets the volume label and serial number in the VPB, and starts up an FSP volume thread.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.4 Open and Create File Operations

The open and create file[3] operation takes as input a file object denoting the file being opened and the device object (created by the file system) which is targeted for the operation. The FSD dispatcher queues the open and create file requests to the work queue for the appropriate FSP volume thread. Figure 5a illustrates the input given to the FSD dispatcher.

Figure 5 - Opening and Creating a File

The file object passed to the file system contains the file name being opened relative to the root of the volume and it contains the requested access mode. It is the job of the file system to open the file on the targeted device object. The information available to the file system is the device object of the volume (passed as input from the I/O system), the VCB and VPB associated with the volume, and a pointer to the targeted device object. The FSP volume thread is free to issue as many read and write sector commands to the targeted device object as necessary to open (create) the file.

When the file is successfully opened, the file system creates a file control block (FCB)[4] and a context control block (CCB) which are both used to store information about the opened file. Both of these records are referenced by the file object through its two FsContext pointers. The file object also contains a section object pointer which points to a reserved longword in the FCB. This longword is used by the memory management system. See Figure 5b.

The FCB and CCB are internal file system control structures. An FCB is created for every opened file on a volume. It contains the file name, mapping information, and a pointer to the VCB containing the file[5]. Only one FCB is created per opened file. The FCB is shared by multiple file objects if the file objects denote the same

---

[3]This discussion also applies to opening and creating directories.

[4]If the operation opens a directory, then the structure is called a directory control block (DCB).

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

file. A CCB is created for every file object[6] and contains information specific to the file object, such as current file position information.

In summary, when opening a file, the file system is passed as input a file object denoting the file and a device object denoting the volume where the file is to exist. If the open operation is successful, the file system creates an FCB and a CCB record, both referenced by the file object.

2.5 Read, Write, Set, and Query File Operations

The basic operations on a file, after it is opened, are read, write, set, and query. These operations take as input a file object and a pointer to the device object for the volume. See Figure 6.

Figure 6 - Read, Write, Set, and Query File Input

When processing an IRP of this type, the FSD dispatcher can either send the IRP to the FSP work queue and let the FSP volume thread handle the request, or it can complete the request itself. The FSP can actually handle all types I/O requests; however, for efficiency, it is better to have the FSD handle the requests when possible. The criterion used to determine if the FSD can handle a request are the amount of time required to process the request and the type of resource locks needed to read or alter the file system's internal data structures. If a lot of extra processing is required or if too many time consuming locks are needed to perform the operation, then the FSP should handle the request.

These operations do not alter the infrastructure connecting the file object, device object, FCB, or CCB. However, they still may need to acquire an exclusive resource lock on the structures in order to alter their other fields (such as setting a delete flag or expanding file allocation).

---

[5]This is a simplification of the actual FCB structure.

[6]Actually as an optimization it is only created when absolutely necessary.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

2.6 Close File Operation

The last operation on a file is the close operation. This operation takes the same input as the preceding read, write, set, and query operations, but unlike those operations, the close operation modifies the infrastructure.

To do a close operation, the file system first performs any necessary I/O to the target device object to either close or delete the file. It then removes the FCB[7] and CCB from its internal data structure. After the close operation the file object is no longer valid and must be removed (or recycled) by the I/O system.

3. Loose Ends

[7]The FCB is removed only it if is no longer reference by any file object.

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License

Revision History

Original Draft 1.0, September 7, 1989

Copyright (c) Microsoft Corporation - Use subject to the Windows Research Kernel License