File System Filter Driver 작성시 주의할 점

드라이버 개발자가 Windows NT의 다양한 운영 체제 환경에서 동작하는 File System Filter Driver를 개발할 때 한번쯤 주의 깊게 살펴 봐야 할 문제들에 대해서 간단히 알아보자.

1. Device Attachment

Device Attachment 관리를 어떻게 처리하는 가는 Windows NT에서 항상 문제가 된다. 예를 들어 두 개의 File System Filter Driver가 File System Driver에 동시에 attach 될 때, 두 드라이버의 로딩 순서는 끊임없이 논의되고 있는 사항이기 때문이다. NT 2000 운영 체제에서는 CIFS/SMB와 같은 Network Redirector가 File System Driver로 등록되어 있기 때문에, 두 개 이상의 File System Filter Driver를 등록할 때 발생하는 내부 문제들이 덜 한 편이지만, NT 4.0 운영 체제에서 둘 이상의 File System Filter Driver를 등록할 때 심각한 문제가 발생할 수 있다. 이러한 문제점들을 해결하기 위해서 File System Filter Driver를 개발할 때 유의해야 할 일은 볼륨을 attach하기 전 이 볼륨이 이미 attach되었나 살펴봐야 한다.
일반적으로 디바이스의 attach 여부를 판별하기 위하여 다음과 같은 코드를 사용한다.

```
// CheckForFiltering
//     이 함수는 이미 필터링 하고 있는 볼륨이 있는지 확인하기 위하여
//     이미 존재하는 Device Object를 조사한다
// 입력 :
//    DeviceObject - 드라이버에서 조사하고자 하는 Device Object.
//                   Device Object는 File System의 Object를 기본으로 한다
// 출력 :
//    없음.
// 반환값 :
//    TRUE - Device Object가 이미 필터링 되고 있을 경우
//    FALSE - Device Object가 필터링 되고 있지 않을 경우
// 주의할 점 :
//    이 코드는 드라이버에서 동일한 Device Object를 생성하지 않도록
//    보장하기 위해서 IoAttach*** 호출하기 전에 호출되어야만 한다.
//
static BOOLEAN CheckForFiltering(PDEVICE_OBJECT DeviceObject)
{
    PDEVICE_OBJECT    topDeviceObject = DeviceObject;
```

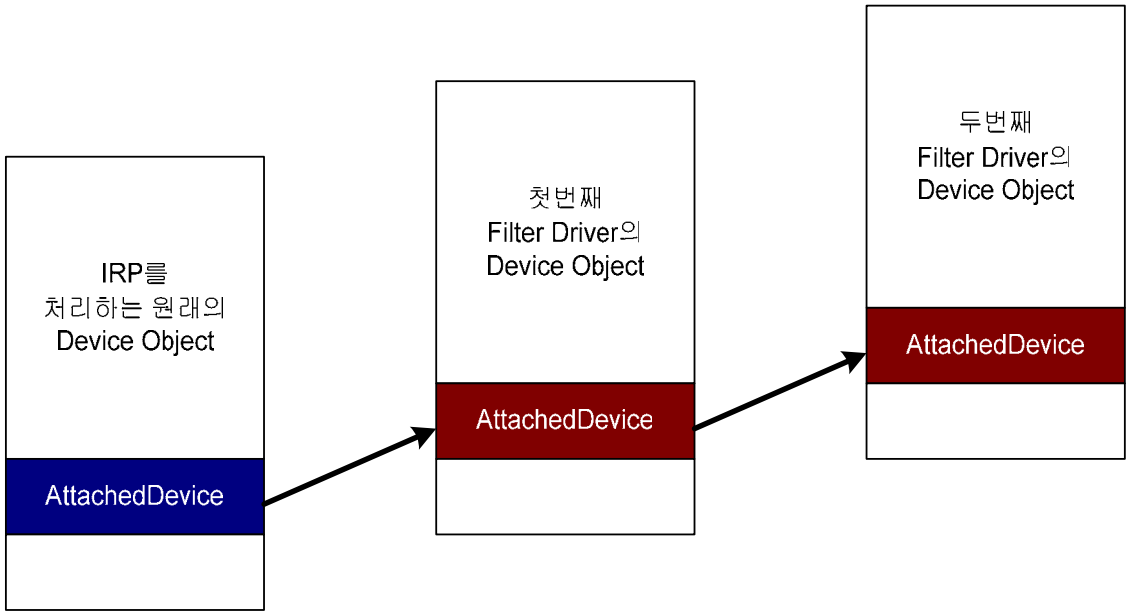
```
    // 드라이버에서 attach된 장치가 있는지 확인하기 위해 attach된 장치의
    // 연결고리를 계속 조사한다.
    while( topDeviceObject ) {
        POUR_DEVICE_EXT   ourExt;

        OurExt = (POUR_DEV_EXT) topDeviceObject->DeviceExtension;
        If ((NULL != ourExt) &&
            (OUR_EXTENSION_MAGIC_NUMBER == ourExt->ExtensionMagicNumber))
        {
            // 이것은 우리가 만든 Device Object이기 때문에,
            // 이미 필터링을 하고 있다.
            return   TRUE;
        }

        // 이것은 우리가 만든 장치가 아니다.
        // 그래서 attach된 리스트의 다음 Device Object를 조사한다
        topDeviceObject = topDeviceObject->AttachedDevice;
    }

    // 이곳까지 왔다면, 드라이버를 위한 Device Object는
    // 필터링 되지 않은 것을 알고 있다.
    return   FALSE;
}
```

필터링 하고자 하는 디바이스에 attach 되는 필터 드라이버는 <그림 1>과 같이 IRP가 전달되는 드라이버의 Device Object 구조체의 AttachedDevice 필드에 저장된다.



<그림 1> File System Filter Driver의 Attach 과정

한가지 더 유의할 점은 볼륨을 attach 할 때 **IoAttachDeviceByPointer()** 함수 대신에 **IoAttachDeviceToDeviceStack()** 함수를 사용하라는 것이다. 왜냐하면 두 개의 File System Filter Driver가 하위에 위치한 동일한 장치(File System Driver)에 attach될 때 "narrow race condition"[1]이 발생할 가능성이 있기 때문이다.

2. Filter Driver간 상호 동작

Windows 2000에서 File System Filter Driver간 통신은 심각한 문제가 될 수 있다. 특히 Windows 2000은 많은 third-party 개발자들이 File System Filter Driver를 사용하고, Microsoft에서 좀더 많은 File System Filter Driver의 사용을 권장하는 새로운 기능들을 소개한 것 때문에, File System Filter Driver간 통신은 어려운 문제 중 하나이다.

---

1) 두개의 File System Filter Driver가 하나의 File System Driver에 attach 된다고 생각해 보자. Windows NT의 드라이버 스택 구조상 하나의 File System Filter Driver는 다른 하나의 File System Filter Driver 위에 놓이게 된다. 그러나 동시에 두개의 File System Filter Driver가 하나의 File System Driver에 attach 될 때, File System Driver는 어떤 File System Filter Driver를 먼저 attach 시켜야 될지 모른다. 따라서 하나의 File System Filter Driver는 다른 File System Filter Driver가 attach 될 때까지 기다려야 되는데, 두개의 File System Filter Driver는 다른 File System Filter Driver가 attach 될 때까지 대기하게 되지만, 어떤 File System Filter Driver도 attach 되지 않아 기약 없이 대기 상태에 놓이게 된다. 이러한 현상을 narrow race condition이라고 한다.

하나의 File System Filter Driver가 다른 File System Filter Driver와 통신을 할 때 나타나는 일반적인 문제는 다음과 같다.

1) Functional Interference

하나의 File System Filter Driver에서 구현한 기본적인 동작이 다른 File System Filter Driver의 동작에 영향을 미치는 경우이다. 예를 들어, 암호화 필터와 바이러스 스캐너가 상호 동작할 경우 잠재적인 문제가 발생할 가능성이 있다.

2) Reentrant Interference

두 종류의 File System Filter Driver가 재 진입에 의한 호출 여부를 알기 위해 서로 다른 방법을 사용한다면 충돌이 발생할 수 있다. 이러한 두 가지 방식이 사용되면, 두 개의 File System Filter Driver가 재 진입을 사용할 때, 안전한 방식을 선택하는 것은 쉬운 일이 아니다.

3) Incorrect or Partial Implementation

File System Filter Driver에서 사용하는 Fast I/O 함수 사용에서 하나의 File System Filter Driver가 Fast I/O를 구현하고, 다른 File System Filter Driver가 Fast I/O를 구현하지 않았다면 심각한 문제가 발생할 수 있다.
Device Stack의 맨 위에 위치한 File System Filter Driver가 Fast I/O를 구현하지 않았다면, I/O Manager가 이 File System Filter Driver에게 I/O를 전달하지 않거나 처리 못하도록 하여 직접 File System Driver와 통신 할 경우 Fast I/O를 구현한 File System Filter Driver까지 그냥 통과하는 상황이 발생할 수 있다.

4) Locking behavior

하나의 File System Filter Driver에서 데이터 보호를 사용하고, 두 번째 File System Filter Driver를 호출할 경우 발생할 수 있는 문제이다. 예를 들어, 하나의 File System Filter Driver가 “fast mutex”를 사용한 후 Zwxxx함수를 호출했을 때, 두번째 File System Filter Driver가 IRP를 Pending 상태로 표시하고 STATUS_PENDING을 반환하는 것으로 되어 있다면, I/O 동작을 완료하지 못하는 경우가 발생할 수 있다. 두개의 File System Filter Driver 중 어느 하나만 로딩되어 있다면 문제가 발생하지 않지만, 두 드라이버가 로딩되어 있다면 시스템이 정지된다.

5) Timing behavior

새로운 데이터 보호 방법이나 IRP 관리를 위한 큐를 가지고 있는 새로운 File System Filter Driver의 추가는 시스템에서 I/O 흐름에 대한 변경이나 I/O를 처리하는 타이밍에 영향을 미칠 수 있다. 여러분이 어떤 시점에서 타이밍 행동을 변경하고자 한다면, 항상

발견되지 않는 버그를 주의해야 한다.

6) Stack Overflow

다수의 File System Filter Driver를 사용할 때는 종종 스택 공간의 부족을 야기할 수도 있다. 커널 스택은 최대 12KB로 제한되어 있기 때문에, File System과 File System Filter Driver의 재 진입과 같은 특성은 스택 공간의 부족을 야기할 수 있다.

불행하게도 이러한 문제들을 모두 해결할 수 있는 방법은 없다. 더구나 하나의 문제를 해결하는 방법은 다른 문제를 야기할 수 있다. 예를 들어, Stack Overflow를 해결하기 위한 한가지 방법은 IRP를 Worker thread의 "Work Item"에 보내는 것이다. 이 방법은 데이터 보호를 위한 방법으로는 잘 동작한다. 그러나 하나의 File System Filter Driver에서 잘 동작되는 기술이 다른 File System Filter Driver에서는 동작하지 않을 수 있다는 것이다.

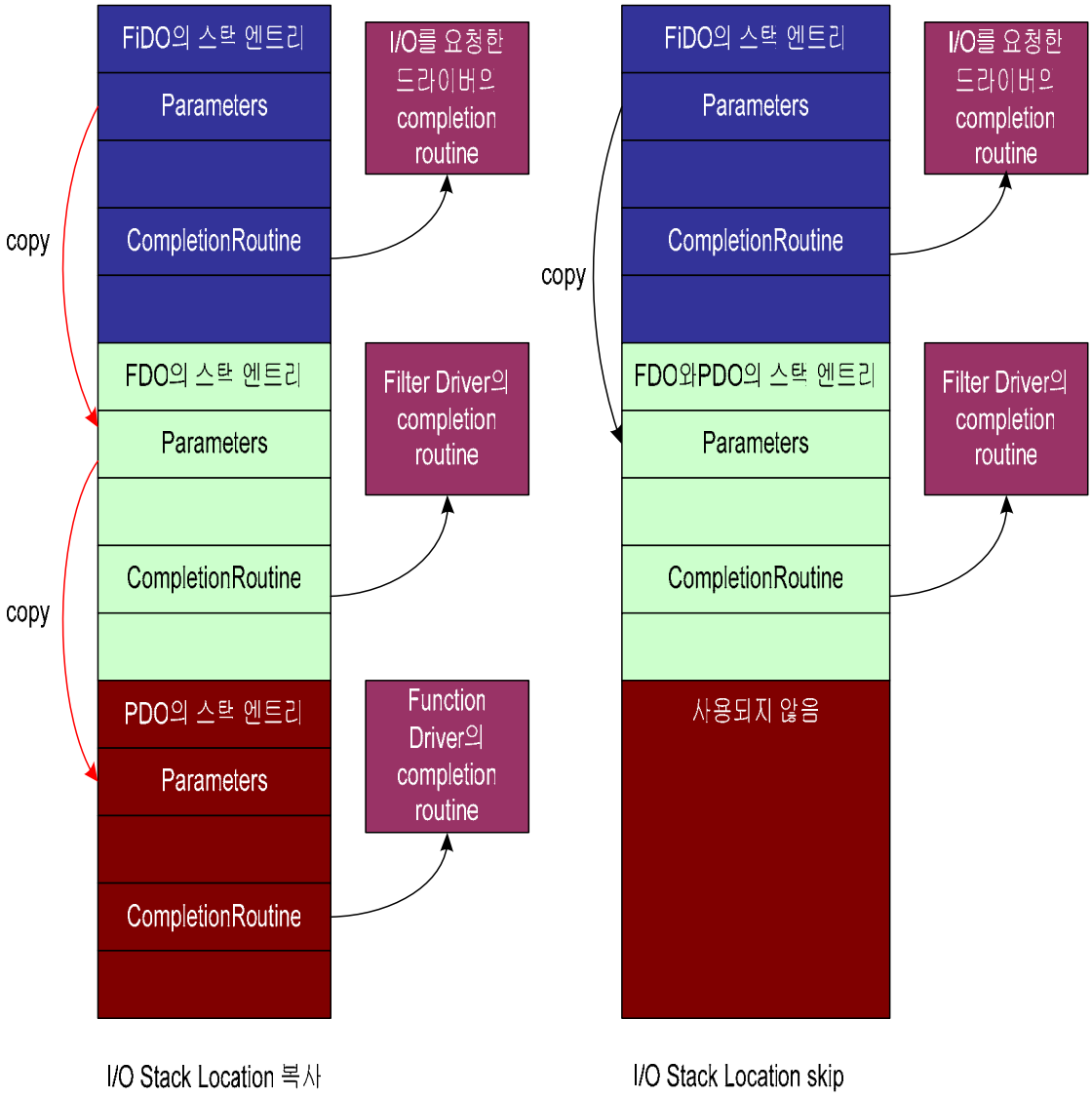
3. IRP Management

Windows 2000에서는 File System Filter Driver가 I/O Request Packet(IRP)을 좀더 쉽게 처리하기 위한 함수들이 소개되었는데, 이중 *IoCopyCurrentIrpStackLocationToNext()* 함수와 *IoSkipCurrentIrpStackLocation()* 함수에 대해서 알아보자.

IoCopyCurrentIrpStackLocationToNext() 함수는 Current Stack Location의 파라미터들을 Next Stack Location으로 복사한다. 이 함수는 호출자의 I/O Completion Routine에서 부적절한 기능을 야기할 가능성이 있는 다른 정보의 복사는 수행하지 않는다.

IoSkipCurrentIrpStackLocation() 함수는 하위 드라이버를 호출할 경우 Current Stack Location를 다시 사용하기 위해 I/O 스택을 수정한다. 이 함수는 데이터 복사와 드라이버와 관련된 완료 과정을 처리 하지 않기 때문에 현재 드라이버가 Completion Routine이 필요하지 않을 경우 유리한다.

<그림 2>는 IoCopyCurrentIrpStackLocationToNext() 함수와 IoSkipCurrentIrpStackLocation() 함수가 처리하는 과정에 대한 내용을 도식화한 것이다.



<그림 2> I/O Stack Location복사와 Skip의 차이점

<그림 2>는 스택에 세 개의 드라이버가 있는 상황을 나타낸 것이다. FDO는 여러분이 개발한 드라이버를 말하며, FiDO는 Fileter Driver를, PDO는 물리 장치를 위한 드라이버를 나타낸다. 왼쪽 그림은 **IoCopyCurrentIrpStackLocationToNext()** 함수를 사용 할 때의 관계를 보여주며, 오른쪽 그림은 **IoSkipCurrentIrpStackLocation()** 함수를 사용할 때의 관계를 보여주고 있다. **IoSkipCurrentIrpStackLocation()** 함수 사용시 세 번째 Stack Location은 사용되지 않는다.

이 함수의 사용 예는 다음과 같다.

```
// 만약 디버깅이 가능하다면 패킷이 완료될 때까지 요구된 처리를 수행하라.
// 그렇지 않으면, 이 요청은 더 이상 처리하지 않는다.

if (SfDebug) {
    PIO_STACK_LOCATION  nextIrpSp;

    // 간단하게 현재 Driver Stack Location의 내용을
    // 하위 드라이버의 스택으로 복사한다.
    IoCopyCurrentIrpStackLocationToNext (Irp);

    IoSetCompletionroutine (
            Irp,
            SfCreateCompletion,
            NULL,
            TRUE,
            FALSE,
            FALSE,
            );
} else {
    IoSkipCurrentIrpStackLocation (Irp);
}
```

위의 두 함수는 Windows 2000에서 새롭게 지원되는 함수이기 때문에 NT 4.0에서는 동작하지 않는다. NT 4.0에서도 동작하도록 하기 위해서는 다음과 같이 매크로를 선언해야 한다.

```
#if !defined(IoCopyCurrentIrpStackLocationToNext)

#define IoCopyCurrentIrpStackLocationToNext( Irp ) { ₩
    PIO_STACK_LOCATION irpSp; ₩
    PIO_STACK_LOCATION nextIrpSp; ₩
    irpSp = IoGetCurrentIrpStackLocation( (Irp) ); ₩
    nextIrpSp = IoGetNextIrpStackLocation( (Irp) ); ₩
    RtlCopyMemory( nextIrpSp, irpSp, FIELD_OFFSET(IO_STACK_LOCATION,
                                        CompletionRoutine)); ₩
    nextIrpSp->Control = 0; }
```

```
#endif

#if !defined(IoSkipCurrentIrpStackLocation)

#define IoSkipCurrentIrpStackLocation( Irp ) ₩
    (Irp)->CurrentLocation++; ₩
    (Irp)->Tail.Overlay.CurrentStackLocation++;

#endif
```

I/O Manager는 미리 생성된 IRP를 위해 두 개의 "Pool"을 관리하며, Windows 2000도 예외는 아니다. I/O Manager가 관리하는 Pool 중 하나는 하나의 I/O Stack Location을 가지고 있고, 다른 하나의 Pool은 Windows 2000에서는 7개의 IRP Stack Location을 가지고 있으며, Windows NT에서는 4개의 IRP Stack Location을 가지고 있다. 이러한 사실은 Windows 2000에서 드라이버 스택이 좀더 커졌다는 것을 의미하며, 이것은 Plug & Play를 지원하는데 사용되는 Bus Driver와 Function Driver 계층이 소개된 탓이다.

물론 운영 체제는 특정 드라이버 스택이 7개의 IRP Stack Location 보다 많은 수의 Stack Location을 요청할 때 역시 잘 동작하지만, 이 경우 IRP는 I/O Manager의 Free List 보다는 Non-Paged Pool에서 할당된다.

File System Filter Driver 개발자들에게 또 하나 유용한 함수는 **IoCancelFileOpen()** 함수이다. 이 함수는 File System Driver에서 성공적으로 처리한 IRP_MJ_CREATE를 취소할 수 있도록 추가되었다. 즉, **IoCancelFileOpen()** 함수는 File System Driver에게 IRP_MJ_CLEANUP과 IRP_MJ_CLOSE IRP를 보내고 새로 생성된 File Object의 상태를 제거하는 역할을 담당한다. Windows 4.0에서는 이러한 함수가 지원되지 않기 때문에 직접 IRP에 대한 Cleanup 처리를 함으로써 이와 같은 동작을 처리해야 한다.

4. Removable Media

몇 가지 사항은 Removable Media에서 심각한 문제를 발생할 수 있다. 이러한 문제 중 한 가지는 File System Filter Driver가 자신의 Device Object를 제거할 때 발생하는 문제이다. 이러한 상황은 File System Filter Driver가 Completion Handler를 사용해야만 확인할 수 있다. File System Filter Driver가 Device Object의 삭제 명령을 받고, 이 명령을 File System Driver에 전달하면, File System Driver는 자신의 Device Object를 삭제하며, 이것은 결국 File System Filter Driver의 Fast I/O Detach 함수를 호출하는 결과를 초래한다. 그러면 File System Filter Driver는 자신의 Device Object를 분리(Detach)하고 삭제한다.

다음으로 File System Driver가 IRP_MJ_CLOSE를 사용하여 현재 처리중인 I/O를 완료시키면, I/O Manager가 File System Filter Driver의 Completion Routine을 호출한다. 만약

Completion Routine에서 Device Object를 액세스할 경우 에러가 발생한다. 왜냐하면 Device Object는 이미 제거된 상태이기 때문이다.
이러한 현상은 IFS Kit에 있는 FileSpy라는 File System Filter Driver의 SpyThroughCompletion 코드를 보면 다음과 같이 자신의 Device Object를 액세스 하는 것을 볼 수 있을 것이다.

```
if (SHOULD_LOG(DeviceObject))
```

또한 SpyFastIoDetachDevice에서도 이와 유사한 코드를 볼 수 있다.

```
IoDetachDevice( TargetDevice );
IoDeleteDevice( SourceDevice );
```

만약 여러분이 IFS Kit을 가지고 있다면, IFS Kit에 포함된 FastFat 관련 소스를 참조해 보면 볼륨상에 더 이상 Open된 파일이 없을 때 다음과 FatCheckForDismount() 함수를 호출하는 것을 볼 수 있다. 이 코드는 FastFat 소스에서 Close.c 파일의 FatCommonClose의 마지막 부분에서 확인할 수 있다.

```
if ( (Vcb->OpenFileCount == 0) &&
                        ((Vcb->VcbCondition == VcbNotMounted) ||
                         (Vcb->VcbCondition == VcbBad)) &&
                        FatCheckForDismount( &IrpContext, Vcb, FALSE ) ) {

                    //
                    //   If this is not the Vpb "attached" to the device, free it.
                    //
```

FatCheckForDismount의 코드의 내부에서 사용되는 코드는 다음과 같다. 이 코드는 strucsup.c 파일에 있다.

```
FatDeleteVcb( IrpContext, Vcb );

        Vpb->DeviceObject = NULL;

        IoDeleteDevice( (PDEVICE_OBJECT)
                        CONTAINING_RECORD( Vcb,
```

```
                                        VOLUME_DEVICE_OBJECT,
                                        Vcb ) );

        VcbDeleted = TRUE;
```

볼륨상에 Open된 파일이 존재하는지 검사한 후, FatCommonClose에서 나머지 처리를 완료한 후 File System Driver는 FatFsdClose에서 FatCompleteRequest를 호출한다. 이것은 결국 File System Filter Driver(여기서는 FileSpy 드라이버)의 Completion Routine을 호출하고, 삭제된 Device Object의 Device Extension을 참조하게 된다.

이러한 문제를 해결하기 위한 가장 간단한 방법은 Reference Count를 관리하는 것이다. 즉, IRP를 File System Driver에 전달하기 전 Reference Count를 증가시키고, Completion Routine에서 이 값을 감소시킨다. 이 Reference Count와 attachment를 위한 변수를 Device Extension 구조체에 저장하여 관리하며, 처리해야 할 IRP가 존재하지 않고 장치가 Detach 되었을 경우에만 Device Object를 삭제하는 방법을 사용하는 것이 가장 간단한 방법이다.

5. Managing File Objects

File System Filter Driver 개발자에게 계속해서 혼동을 주는 것 중 하나는 File System Filter Driver에서 File Object를 처리하는 것이다. 특히 Windows 2000에서 소개된 새로운 함수인 IoCreateStreamFileObjectLife()는 File Object의 처리를 더 힘들게 만든다.

여기서 File System Filter Driver 개발자가 명심해야 할 사항은, File Object는 파일을 나타내는 것이 아니라 실제 파일을 참조한다는 것이다. 파일이 Open될 때마다 새로운 File Object가 생성된다. 파일은 어플리케이션이나 File System Filter Driver에서 Open할 수 있으며, 심지어 File System Driver도 파일을 Open할 수 있다. 파일 Open은 대부분 IoCreateFile() 함수를 사용하여 Open되지만(IoCreateFile은 결국 Native 함수인 NtCreateFile()을 호출하며, File System Filter Driver 역시 파일을 Open하기 위하여 ZwCreateFile()을 사용하지만, 이 함수는 결국 NtCreateFile() Win32 Native 함수를 호출한다), 몇몇 파일은 IoCreateStreamFileObject() 함수나 IoCreateStreamFileObjectLite() 함수를 사용하여 Open된다.

일반적으로 File System Filter Driver는 FsContext 필드를 사용하여 File Object를 관리하는데, Network File System의 경우 이 필드 값은 파일이 실행되는 동안에도 업데이트 된다. 왜냐하면 Network File System은 파일에 대해 어플리케이션이 I/O 동작을 구현하기 전까지는 실제 Open되지 않기 때문이다.

아마 File System Filter Driver 개발자들이 직면하게 되는 가장 큰 문제는 File System이 자체적으로 생성한 File Object이다. 왜냐하면 일반적으로 새로운 File Object의 생성은

IRP_MJ_CRETEA를 통해 발생하기 때문에 File System Filter Driver에서 모니터링 할 수 있지만, File System이 직접 생성한 File Object를 File System Driver에게 직접 지시하기 때문이다.

Windows NT 4.0에서 IoCreateStreamFileObject()에서 생성된 File Object를 위해 IRP_MJ_CLEANUP IRP를 발생시키기 때문에 File System Filter Driver에서 관찰할 수 있지만, Windows 2000에서 IoCreateStreamFileObjectLite()는 IRP_MJ_CLEANUP IRP를 발생시키지 않는다.

이러한 환경에서 동작하는 File System Filter Driver를 개발하기 한가지 방법은 각 파일에 대한 유일한 값을 나타내는 FsContext 값을 관리하기 위한 데이터 구조체를 생성하고, 관리하고자 하는 파일과 관련된 File Object의 목록을 유지한다. 그 다음에 IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_QUERY_INFORMATION, IRP_MJ_SET_INFORMATION, IRP_MJ_CREATE, IRP_MJ_CLEANUP IRP를 처리하는 동안, 처리중인 File Object가 주어진 파일과 연관된 File Object의 하나로써 리스트에 있는지 조사한다. 만약 관리되고 있는 리스트에 없다면, 이 File Object를 리스트에 저장하여 관리한다.

6. Plug & Play

Windows 2000에서 Plug & Play 기능이 소개되었으며, File System Filter Driver 개발자들이 관심을 가져볼 만한 IRP들에 대해서 살펴보자.

1) IRP_MJ_QUERY_REMOVE_DEVICE

이 IRP는 Plug & Play Manager가 스택 하부에 존재하는 저장 장치의 제거 요청을 받았다는 것을 나타낸다. File System Filter Driver에서 IRP를 처리해야 할 경우는 장치 제거 전 처리해야 할 작업이 있거나, 어떤 동작이 처리 중일 경우 제거 요청을 거부하기 위해서 사용할 수 있다.

2) IRP_MJ_CANCEL_REMOVE_DEVICE

장치를 제거하고자 하는 요청이 취소되었다는 것을 나타낸다.

3) IRP_MJ_REMOVE_DEVICE

장치가 제거되었다는 것을 나타낸다.

4) IRP_MJ_SURPRISE_REMOVAL

어떤 장치를 제거한다는 메시지 없이 장치의 제거가 발생했음을 나타낸다.

7. Virtual Memory

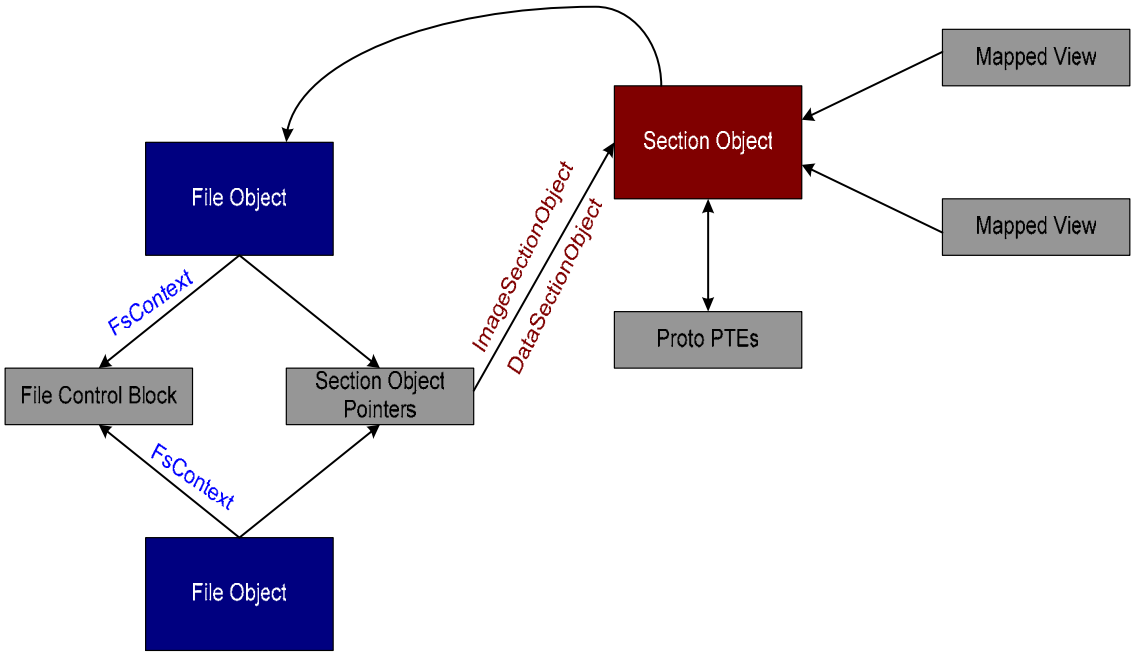
File System Filter Driver 개발자에게 혼동을 주는 것 중 한가지 사실은 File System Driver와

Virtual Memory Manager간의 간섭에 관한 사실이다. NT 4.0에 비교해서 Windows 2000이 달라진 점은 메모리 매핑 파일의 사용이 증가했다는 것이다. 예를 들어 "Notepad"는 Windows 2000에서 파일을 수정하기 위해 메로리 맵핑을 사용할 수 있도록 수정되었다는 사실이다. Windows NT에서 대부분 메모리 맵핑 파일을 아무도 사용하지 않았기 때문에 File System Filter Driver에서 이를 무시했지만, Windows 2000에서 File System Filter Driver 개발자는 더 이상 메모리 맵핑 파일을 무시하면 안 된다.

Windows NT 4.0에서 파일은 메모리 맵핑된 파일 인터페이스를 사용하여 Cache에서 직접 수정되었다. Virtual Memory 인터페이스를 사용한 수정은 어플리케이션에서 발생하는 일반적인 IRP_MJ_WRITE에서 볼 수 없다. 왜냐하면 파일의 수정은 직접 메모리 참조를 통해 Cache에 있는 메모리 복사본에 대해서 이루어졌기 때문이다. 결국 File System은 이러한 파일에 대한 변경 동작을 전혀 포함하지 않는다. 그러나 변경된 파일은 영구적인 저장 장치에 전달하기 위하여 결국 하위에 위치한 File System Driver에게 전달된다.

File System Filter Driver는 이러한 변경을 Paging I/O 동작에서 발견할 수 있으며, Paging I/O 동작은 File System에 전달되는 IRP_MJ_WRITE IRP에 IRP_PAGING_IO 비트가 설정되어 있는지 확인함으로써 알 수 있다.

물론 Paging I/O를 추적하는데 어려운 것 중 하나는 File System이 더미(dummy) File Object의 사용과 연관되어, 내부에서 생성된 File Object 중 하나에 대해서 Paging I/O가 발생하는 것이다. <그림 3>은 이러한 현상을 나타낸 것이다.



<그림 3> File Object의 관계

그림에서 보는 것처럼 동일한 데이터 구조를 참조하는 여러 개의 File Object가 있다. 여기

서 한가지 흥미로운 점은 Section Object가 거꾸로 특정한 File Object를 참조한다는 것이다. Virtual Memory 시스템은 이 Section Object에 대한 동작 구현이 필요할 때마다 특정 File Object에 대해 페이징 I/O 동작을 발생 시킨다.
이러한 행동은 NTFS에서는 일반적인 현상은 아니다. 왜냐하면 NTFS는 일반적으로 자신이 소유하기 위해 내부적으로 File Object를 생성하기 때문이다. NTFS가 내부에서 생성한 File Object들이 Section Object를 추적하기 위해 사용된다면, 이어서 발생하는 Paging I/O 동작은 Section Object가 참조하는 File Object에 대해서만 발생한다. 이런 File Object는 IRP_MJ_CREATE IRP로 볼 수 없기 때문에 File System Filter Driver가 NTFS가 생성한 File Object를 추적하지 않는다면, 이 File Object에 대한 I/O는 처리할 수 없을 것이다.

출처 : NT Insider 2000년 판

- Filtering the Riff-Raff : Observation on File System Filter Drivers
(http://www.osronline.com/article.cfm?id=34)