6장. 가상 머신 관리자
(The Virtual Machine Manager)

VMM(Virtual Machine Manager)는 윈도우즈 95 운영체제에서 아주 중요한 부분이다. VMM은 VM(virtual machine)을 관리하는데 대한 기본적인 골격을 만들고 이를 유지한다. VxD(virtual device driver)는 하드웨어 장치들을 "가상화(virtualize)"하고, VMM과 함께 동작하면서 각 응용 프로그램에게 시스템 서비스를 제공한다. 쉽게 생각한다면 VMM은 컴퓨터 전체에 대한 가상 드라이버라고 생각할 수 있다. 왜냐하면 실제 기계에는 한 개밖에 없음에도 불구하고 여러 개의 프로그램이 이를 계속적으로 동작시킬 수 있기 때문이다. 그러나 VMM은 보통의 VxD보다 더 책임이 막중한데, 그것은 VMM이 VxD에서 호출하는 유틸리티 함수를 제공하며 모든 VxD를 관리하기 때문이다.

이 장에서는 VMM 동작의 3가지 기능에 대해서 설명한다. 이 동작은 실제적으로 윈도우즈 95에서 발생하는 모든 것이며, 여기에는 메모리 관리, 인터럽트 핸들링, 쓰레드 스케줄링이 있다. 메모리 관리에 대해 기본적으로 알아두어야 할 것은 VMM이 어드레스 공간을 어떻게 나누느냐 하는 것이다. VxD에서 호출하는 메모리 할당 및 콘트롤 서비스에 대한 것은 나중에 다른 장에서 설명할 것이다. VMM이 인터럽트를 어떻게 처리하는가에 대해서도 이해해야 한다. 왜냐하면 이에 대한 지식이 없으면 몇 종류의 인터럽트는 어떻게 후크 하는가를 알기 어렵기 때문이다. 마지막으로 VMM이 실행할 쓰레드를 어떤 방법에 의해 선택하는지에 대한 이해도 필요하다. 왜냐하면 이것은 응용 프로그램이 동작을 허용 받기 위한 기본 메커니즘이기 때문이다.

6.1 메모리 관리 (Managing Memory)

컴퓨터에 있는 모든 자원은 여러 가지 방법으로 우리를 위협한다. 프로세서 사이클은 모두에게 돌아갈 만큼 충분하지 않고, 화면은 우리가 오픈 하려고 하는 윈도우를 다 나타내줄 만큼 크지 않고, 비디오 카드는 충분히 빠르지 않거나 요즘 유행하는 이미지를 지원할 정도로 색상을 지원하지 못하고, 모뎀은 초고속 정보 통신망에서 요구하는 밴드 폭에 만족하지 못한다. 그러나 컴퓨터의 모든 자원 중에서 계속 우리를 위협해온 것은 메모리다. 윈도우즈 95는 향상된 메모리 관리로 인해 이 걱정거리를 많이 덜어주기는 한다.

6.1.1 시스템 메모리 맵 (The System Memory Map)

VMM은 32비트 가상 어드레스 공간을 사용하기 위해 인텔의 80386 이상 프로세서의 페이지 기능을 사용한다. 메모리 관리를 간단하게 하기 위해, VMM은 그림 6-1과 같이 사용 가능한 어드레스 공간을 5구간으로 나눈다. 4개의 주요 구간은 다음과 같다.

- V86 구간(The V86 region) : V86구간은 선형 어드레스의 0h ~ 10FFEFh로서, 현재 실행중인 가상 머신을 가지고 있다. (10FFF0h ~ 003FFFFFh의 어드레스는 사용하지 않는다.) 이 구간은 0000h:0000h ~ FFFFh:FFFFh의 16:16 포인터로 주소지정 가능한 모든 메모리가 여기에 포함된다.
- 응용 프로그램 전용 구간(The private application region) : 응용 프로그램 전용 구간은 00400000h ~ 7FFFFFFFh이다. 이 가상 메모리의 위치는 특정한 ***메모리 컨텍스트(memory context)***에 전용으로 사용된다. 이것은 기본적으로 Win32 프로세스에 해당된다. 각 가상 머신과 각 응용 프로그램은 서로 다르게 구별되는 프로세스에 속해 있기 때문에, 이들은 자체의 컨텍스트를 가지고 있다.
- 응용 프로그램 공유 구간(The shared application region) : 응용 프로그램 공유 구간은 80000000h ~ BFFFFFFFh이다. 윈도우즈는 어드레스 공간 중 KERNEL32.DLL과 USER32.DLL 같은 시스템 자체 3순위 권한 DLL과 모든 Win16 응용 프로그램을 이 구간에 로드 한다. 또한 윈도우즈는 시스템 VM에서 실행되는 Win32 프로그램으로부터의 파일 맵핑 요구(***CreateFileMapping*** 호출을 통해서 생성)에 대해서나 가상

머신에서 DPMI 클라이언트로부터의 메모리 할당 요구에 이 구간을 사용한다.

- 시스템 공유 구간(The shared system region) : 시스템 공유 구간은 C0000000h부터 가상 메모리의 끝까지이며, 이 구간에는 모든 프로세서와 VM이 공유하는 시스템 데이터와 프로그램에 있다. 각 VM의 V86 구간에 있는 페이지도 역시 어드레스 공간중 이 구간에 맵핑 된다. 이것은 ***하이리니어(high linear)*** 맵핑이라 부르며, 이 하이리니어 맵핑은 VxD가 현재의 VM이 아니더라도 특정한 VM의 V86 메모리를 엑세스하도록 허용한다. (이런 경우 다른 VM의 메모리는 주소 공간의 하위 1MB를 점유하는 것이다.)

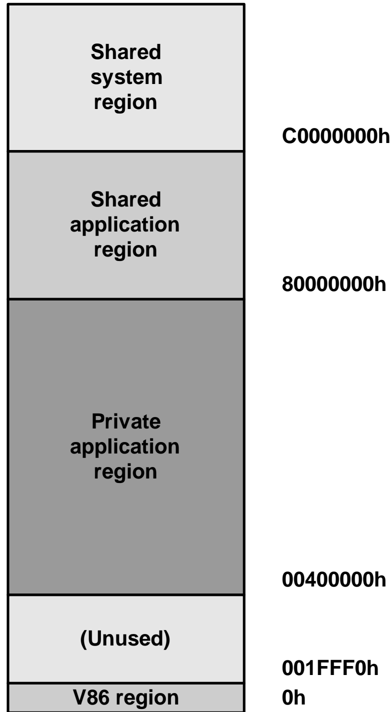

그림 6-1. 윈도우즈 95의 가상 어드레스 공간 분포도

6.1.2 메모리 공유 (Memory Sharing)

전용 메모리(context-specific memory)와 공유 메모리(shared memory)를 구별하는 것은 시스템 프로그래머에게 중요한 것이 관련되어 있다. 모든 Win16 응용 프로그램이 응용 프로그램 공유 구간에 위치한다는 것은 보기에는 좋지 못하지만, 윈도우즈 95는 응용 프로그램이 서로 다른 메모리를 자유롭게 엑세스 할 수 있었던 이전 버전과 호환성을 유지하기 위한 필요악의 선택이었다. 또한, Win16 프로그램과 도스 확장 프로그램이 DPMI 호출에 의해 할당하는 메모리는 이 구간에 위치하게 되며 계속적으로 선형 어드레스를 공유할 수 있다.

그러나, 원칙적으로 공유가 불가능한 Win32는 호환성에 또 다른 문제가 있다. 두개의 Win32 응용 프로그램이 정당하게 메모리를 공유할 수 있는 유일한 방법은 파일 맵핑이라 불리는 것을 사용하는 것이다. (*GlobaAlloc*는 옵션인 GMEM_SHARE을 지정하더라도 공유 메모리를 생성하지 않고 현재 프로세스의 전용 구간에 메모리를 할당한다. 이것은 윈도우즈 NT 경험 있는 프로그래머에게는 놀라운 일이 아니다.) 그리고 Win32 응용 프로그램에 의해 메모리가 공유 가능하다면, Win16과 확장 도스 프로그램에게도 자동적으로 역시 가능해진다는 말이다.

호환성에 대한 설명 (Compatibility Note)

현재 운영되는 윈도우즈의 버전을 체크하지 않은 채 Win16 프로그램이나 도스 프로그램과 Win32 응용 프로그램간의 메모리 공유는 사용하지 말아야 한다. 마이크로소프트는 추후 이 어드레스 배치를 바꿀지도 모른다.

윈도우즈 95에서는 현재 공유 메모리 맵 파일(shared memory-mapped file, 정식 명칭)과 비공유 메모리 맵 파일(unshared memory-mapped file, 비공식 명칭)을 응용 프로그램 공유 구간(shred application region)에 할당하고 있지만, 마이크로소프트는 이를 버그로 간주하고 있다. 결국, 비공유 메모리 맵 파일은 이것이 공유되는 것을 방지하기 위하여 응용 프로그램 전용 구간(private application region) 에 할당되게 될 것이다.

(역자 주 : 페이지 테이블 디렉토리와 페이지 테이블은 서로 다른 테이블이다. 원문에서는 이렇게 사용하고 있지만, 두 용어의 혼란을 피하기 위해 페이지 테이블 디렉토리는 페이지 디렉토리로 기재 할 것이며, 페이지 테이블은 그대로 페이지 테이블을 사용할 것이다)

하위 4MB (The First Four Megabytes)

V86 구간의 끝에서부터 응용 프로그램 전용구간의 시작(0010FFF0h ~ 003FFFFFh)까지의 대략 3MB의 구간을 윈도우즈 95는 무엇에 사용하는지 궁금해 할 지 모르겠다. 대답은 아무 것도 아니다 이다. 윈도우즈 95가 이 영역을 포기한 이유는 인텔칩이 페이지 테이블을 구성하는 방법과 관계가 있다. 시스템 콘트롤 레지스터의 하나인 CR3는 4096 바이트 페이지 디렉토리를 가리킨다. 페이지 디렉토리의 각 4바이트 엔트리는 4096 바이트 페이지 테이블의 물리 어드레스를 가리킨다. 페이지 테이블의 각 4바이트 엔트리는 4096 바이트 페이지의 물리 어드레스나 VMM이 이 페이지를 초기화하거나 스왑 파일로부터 불러들여야 할지를 나타낸다.

한 개의 페이지 테이블 디렉토리의 한계 엔트리는 얼마의 메모리를 제어하는가. 1024 [페이지 테이블 엔트리] × 4096 [페이지당 바이트 수] = 4MB. 페이지 디렉토리의 한 개 DWORD는 00400000h 바이트의 가상 메모리를 나타냄을 볼 수 있다. 이것은 VMM이 가상 머신의 V86 구간을 위해 예약한 크기이다. 윈도우즈 3.0과 3.1 (여기서는 프로세스마다 메모리 구간은 없음)에서 VMM은 페이지 디렉토리의 페이지 테이블 포인터를 바꾸어서 메모리 컨텍스트를 한 VM에서 다른 VM으로 간단히 전환할 수 있었다. V86 메모리는 하위 1MB 만을 점유하고 있기 때문에 이 페이지 디렉토리의 엔트리에 의해 등록된 3MB는 간단하게 잃어버리게 되는 것이다. (윈도우즈 95는 아직도 V86 메모리를 참조하는데 한 개의 페이지 디렉토리 엔트리를 사용하고 있지만, 각 컨텍스트의 전환에 있어서 윈도우즈 95는 *많은* 페이지 디렉토리 포인터를 스왑 해야 한다.)

VxD에서 물리 메모리에 없는 메모리 참조(out-of-memory reference)에 주의하지 않으면 매우 심각한 사태를 발생할 수 있다. 예를 들어 하드웨어 인터럽트 처리 중에는 되도록 프로세스가 메모리를 참조하지 않도록 해야 한다. 이러한 규칙을 어기면, VMM이 페이지 폴트를 처리할 수 없을 때 페이지 폴트가 발생하게 되어 시스템이 정지할 할 수 있다. ***_LinPageLock*** 서비스는 VxD 프로그래머에게 전용 페이지를 물리 메모리에 묶어 두거나 현재의 컨텍스트가 무엇이든지 관계없이 항상 유효한 공유 구간의 앨리어스를 얻도록 해준다.

6.1.3 메모리 관리 API (Memory Management APIs)

VxD는 가상 메모리를 할당하거나 해제하기 위해 세 가지 종류의VMM 서비스를 사용할 수 있다. 페이징 시스템은 몇 개의 서비스를 제공하는데, 여기에는 시스템 페이지 크기인 4096 바이트의 배수로 가상 메모리의 큰 영

역을 할당하기 위해 *_PageAllocate*와 *_PageFree*가 포함되어 있다. 힙 관리자는 몇 개 힙 중의 하나에서 작은 블럭의 메모리를 할당받기 위한 *_HeapAllocate*와 *_HeapFree* 서비스를 제공한다. 힙은 3개로 분류할 수 있는데, 페이징 가능한 메모리, 페이지 록 된 메모리, 디바이스 초기화의 마지막(다른 말로, Init_Complete 시스템 제어 메시지가 처리된 후)에 제거되는 메모리가 그것이다. 링크드 리스트 관리자는 VxD가 *List_Allocate* 호출을 통해 리스트를 만들 고정 크기의 메모리 블록을 생성하는 것을 제공한다.

VxD는 0순위 권한 코드에 의해서만 사용할 메모리를 위해 힙과 링크드 리스트 서비스를 사용하며, 이와는 대조적으로 응용 프로그램이 사용할 메모리 영역을 할당하기 위해 *_PageAllocate*를 우선적으로 사용한다. 예를 들어, 큰 V86 메모리 블록은 *_PageAllocate*를 사용하여 할당받으며, 이것은 단일 VM의 내부용으로 사용되고 MS-DOS의 끝과 패러그래프 A000h사이에서 할당되지 않는 공간을 할당한다. 이것을 VM의 비디오 램 어드레스 뒤에 놓여있는 페이지를 끌어 모으는 일을 한다. 응용 프로그램의 *VirtualAlloc*나 DPMI 메모리 요청 함수는 다시 *_PageAllocate* 호출을 한다. 또한 힙과 링크드리스트 관리자는 *_PageAllocate*를 사용하여 호출된 페이지를 하위로 관리한다.

6.2 인터럽트 핸들링(Handling Interrupts)

컴퓨터는 그 결과가 사용자(end-user)에게 이익이 되는 연산을 수행하고 있을때 보통(normal)의 상태에 있다고 생각할 수 있다. 인터럽트는 운영체제가 예외적인 상태를 처리하고 운영체제 서비스를 공급하기 위하여, 이러한 이익이 되는 연산을 유보시키는 기본 메커니즘이다. 따라서 그림 6-2와 같이, VMM이 무엇을 하는가에 대해 실제적으로 정리해 보면 다음과 같다. 인터럽트가 발생했을 때 VMM은 이를 받는다. VMM은 인터럽트를 처리하고 인터럽트가 발생하기 전에 수행하던 것을 계속하기 위해 IRETD 명령을 실행한다.

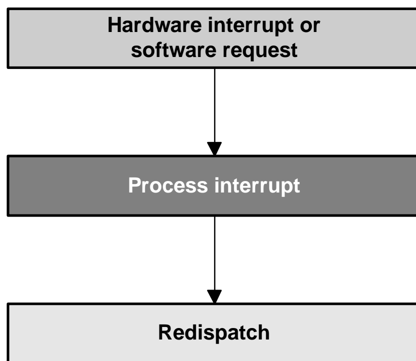

그림 6-2. 가상 머신 관리자의 주요 경로

6.2.1 1수준과 2수준 인터럽트 핸들러 (First-Level and Second Level Interrupt Handlers)

VMM은 각자 인터럽트를 처리할 수 있도록 1수준 인터럽트 핸들러의 큰 집합을 가지고 있다. 1수준 핸들러의 작업은 단순히 인터럽트 된 프로그램의 상태를 저장하는 것이며, 실제 인터럽트를 처리하는 2수준 핸들러로 제어권을 넘기는 일을 한다. 상태 저장이란 스택에 범용 레지스터와 세그먼트 레지스터를 넣기 위해 프로세서의 PUSHAD와 MOV 명령을 사용하는 것이다. 2수준 핸들러는 각각의 인터럽트에 적절한 서비스를 수행하고 후처리루틴(redispatch routine)으로 제어권을 넘긴다. 후처리기는 저장된 상태를 복구하고 인터럽트 된 지점으로 실행권을 넘겨주기 위해 IRETD 명령을 실행한다. (역자 주 : redispatch는 말 그대로는 재처리지만 실제 동작은 후처리가 맞는것 같아 여기서는 후처리로 쓰기로 했다.)

코프로세서 상태의 저장(Saving the Coprocessor Context)

VMM이 현재 프로그램의 상태를 저장하기 위해 PUSHAD와 MOV 명령을 사용한다는 것을 읽었을 때, 산술 코프로세서는 어떻게 되는지 궁금할 것이다. FSAVE와 FRSTOR 명령어는 코프로세서의 모든 상태를 저장하고 복구한다. 그러나 이 명령들은 다른 명령어에 엄청나게 비싼 대가를 지불해야 한다. 따라서 윈도우즈는 인터럽트 동안 코프로세서의 상태를 자동으로 저장하지 않는다. 사실은, 한 쓰레드에서 다른 쓰레드로 전환할 때 이 상태를 저장하지 않는 대신, 눈에 잘 띄지는 않지만 CR0 레지스터에 있는 태스크 전환 플래그(task switched flag)를 사용한다. 쓰레드를 전환하는 동안 이 플래그를 설정하면 다음 쓰레드가 코프로세서 명령을 사용할 때 프로세서는 폴트를 발생시킨다. 윈도우즈는 그때 코프로세서의 상태를 전환한다. 이것이 바로 철저한 "just in time" 프로그래밍이다.

인터럽트의 후처리는 실제적으로 더 복잡하다. 후처리 이전에서 수행되었던 2수준 핸들러는 인터럽트 핸들링 동안 처리하지 못한 작업을 수행하기 위해 이벤트를 스케줄 해 놓았을 것이다. 만약 그렇다면 후처리기는 현재 인터럽트를 종료하기 전에 이 이벤트와 연관된 콜백 함수를 호출해야 한다. 더구나 후처리기는 인터럽트를 끝내고 되돌아가는 곳이 인터럽트 된 프로그램이 아닌 다른 쓰레드인 경우도 있다. 예를 들어, 인터럽트 된 쓰레드의 실행시간(time slice)이 만료되었거나 2수준 핸들러가 그 쓰레드를 막아 버린 경우이다. 이 경우 스케줄러는 다음에 누가 실행되어야 하는지 결정하게 되는 것이다.

6.2.2 인터럽트 디스크립터 테이블(Interrupt Descriptor Table)

모든 VM은 각자의 IDT(interrupt descriptor table)을 가지고 있다. 왜냐하면 윈도우즈 95는 항상 보호모드에서 실행하기 때문에(단일 MS-DOS 응용 프로그램 모드 같은 특별한 경우는 "진짜(real)" 리얼모드다.), VM의 초기 벡터를 통한 모든 인터럽트는 그 VM의 IDT를 통해 VMM의 1수준 핸들러로 전달된다.

VMM은 VM이 생성된 최초에 VM의 IDT를 초기화한다. IDT의 초기값은 시스템이 초기화되는 동안 VMM과 VxD가 만든 디폴트 테이블이다. VxD는 VMM의 디폴트값과는 다른 IDT 엔트리의 초기값을 지정하기 위해 *Set_PM_Int_Vector*와 *Set_PM_Int_Type* 서비스를 사용한다.

얼마나 많은 IDT가 있는가? (How Many IDTs?)

Set_PM_Int_Vector 서비스는 보호모드 응용 프로그램으로부터 오는 인터럽트에 대한 1수준 핸들러를 지정한다. 그러나 이 1수준 핸들러는 V86 모드 응용 프로그램에게는 적당하지 않다. 그래서 각 VM은 실제적으로 두개의 IDT를 가지고 있다. 하나는 V86 프로그램을 위한 것이며, 다른 하나는 보호모드 프로그램을 위한 것이다. V86 인터럽트에 대한 1수준 핸들러의 디폴트 동작은 0000h:0000h에 있는 인터럽트 벡터를 통해 인터럽트를 넘겨주는 것이다.

6.2.3 인터럽트 맛보기(Flavors of Interrupts)

윈도우즈 95는 하드웨어 인터럽트, 소프트웨어 인터럽트, 프로세서 예외를 다르게 취급한다. 하드웨어 인터럽트는 서비스에서 I/O 디바이스의 사용이 필요하다. 프로그램은 INT *n* 명령을 사용해서 소프트웨어 인터럽트를 발생시킨다. 프로세서는 스스로가 어떤 예외적인 상태를 운영체제가 처리해야 한다고 판단될 때 예외 인터럽트를 발생시킨다.

인터럽트는 VxD와 보호모드 응용 프로그램이 인터럽트를 후크하는 방법과 VMM이 인터럽트를 지원하는 방법에 따라 분류할 수 있다. 응용 프로그램은 DPMI 서비스를 이용하여 인터럽트와 예외를 후크하며, VxD는 DPMI 호출과 추가적으로 VMM 서비스를 이용하여 하드웨어 인터럽트, 소프트웨어 인터럽트, 프로세서 예외를 후크 할 수 있다.

■ ***Set_PM_Int_Vector***와 ***Set_PM_Int_Type***는 IDT의 게이트 엔트리를 변경함으로써 보호모드 응용 프로그램으

로부터 오는 인터럽트의 1수준 핸들러를 바꾸는 것이다. VxD는 보통 3순위 권한 핸들러를 설치하기 위해서만 이 두 함수를 사용한다.(왜냐하면 0순위 권한 1수준 핸들러는 상태 저장 및 스택 전환 등의 매우 많은 작업이 필요하기 때문이다.) 그러나, 3순위 권한 핸들러는 종종 실제적으로는 ***Allocate_PM_Call_Back*** 호출에 의해 생성되는 INT 30h을 실행하는 것이다. 인터럽트가 발생했을 때, 이에 연관된 0순위 권한 콜백 루틴은 VMM이 보통의 VxD 실행을 위한 환경을 설치한 후 제어권을 얻게 된다.

- ***Set_V86_Int_Vector***는 0000h:0000h의 인터럽트 벡터 테이블의 원거리 함수(far function)의 주소를 변경한다. 이것은 가상 인터럽트를 받기 위해 V86모드 서비스 루틴의 어드레스를 바꾼다.
- ***Hook_PM_Fault***는 프로세서 예외나 보호모드 응용 프로그램으로부터 발생하는 소프트웨어 인터럽트를 처리하기 위한 2수준 핸들러를 설치한다. 디폴트 핸들러는 0000h:0000h 인터럽트 벡터 테이블로 인터럽트를 전달한다.
- ***Hook_VMM_Fault***는 0순위 권한 코드가 실행 중 발생한 프로세서 예외를 처리하기 위한 2수준 핸들러를 설치한다. VxD는 이 서비스가 거의 필요하지 않다. 왜냐하면 VMM은 벌써 0순위 권한 폴트에 대한 적절한 핸들러를 가지고 있기 때문이다.

VxD에서는 잘못된 포인터 참조를 잡아내기 위해 ***Install_Exception_Handler***를 사용할 수 있다. (그리고, 우선적으로 잘못된 포인터를 생성하는 VxD를 방지하기 위해 VMM 서비스인 ***_Assert_Rang***를 사용할 수도 있다.)

6.2.4 디폴트 핸들링 (Default Handling)

VMM의 원래 목적은 윈도우즈와 다중 MS-DOS 세션이 동시에 동작하기 위해 컴퓨터를 가상화하기 위함이었다. MS-DOS는 벌써 인터럽트 핸들러의 집합을 가지고 있다는 것을 알고 있을 것이다(역자 주 : 인터럽트 벡터 테이블). 윈도우즈의 KERNEL, USER, GDI 모듈 및 이 모듈을 사용하는 3순위 권한 디바이스 드라이버들도 역시 인터럽트를 처리한다. 따라서 컴퓨터를 가상화 하기 위한 VMM의 목적은 맞아떨어진 것이다. 거의 모든 인터럽트에 대한 VMM의 디폴트 핸들링은 그 인터럽트와 관계된 3순위 권한 핸들러를 가진 VM으로 인터럽트를 전달한다. 인터럽트를 전달하는 것은 그 VM을 처리한 후 다시 제어권을 얻기 위해 해당 VM에 저장된 레지스터 이미지를 변경하게 된다.

그러나 항상 인터럽트를 즉시 VM에게 전달해야 하는 것만은 아니다. 하드웨어 인터럽트가 발생했을 때, 이에 대한 인터럽트 핸들러는 인터럽트가 발생한 시점에서 실행 중이던 VM과는 다른 VM에 있을 수 있다. 많은 프로세서 예외(특히 페이지 폴트)의 핸들링은 VM의 핸들링보다는 VMM에 의한 핸들링이 필요하다. VMM과 VxD는 그 인터럽트와 관련된 자원을 가상화 하기 위해 소프트웨어 인터럽트를 검사하는 것이 최소화되어야 한다.

(1) 하드웨어 인터럽트 (Hardware Interrupts)

시스템 요소 중의 하나인 VPICD(Virtual Programmable Interrupt Controller Device)는 인터럽트 50h ~ 5Fh의 1수준 핸들러를 가지고 있다. VPICD는 IRQ 0h ~ Fh를 리얼모드의 BIOS 디폴트 인터럽트(08h ~ 0Fh와 70h ~ 77h)를 50h ~ 5Fh로 바꾸기 위해 인터럽트 콘트롤러를 프로그램 한다. 이 인터럽트에 대한 IDT 게이트는 DPL이 0을 가리키고 있어서, 3순위 권한 프로그램이 소프트웨어 인터럽트처럼 사용하려고 하면 GP 폴트가 발생한다. 물론 0순위 권한 코드는 이것을 소프트웨어 인터럽트처럼 사용하지는 않을 것이다.

이 인터럽트에 대한 VPICD의 디폴트 동작은 가장 적절하다고 판단되는 VM의 3순위 권한 핸들러에 하드웨어 인터럽트를 전달한다. 인터럽트를 받는 VM은 몇 가지에 의해 결정되는데 여기에는 윈도우즈 95가 시작할 때 IRQ가 마스크 되었는지 마스크 되지 않았는지에 따라 달라지는 것도 포함되어 있다. 13장에서 가상 인터럽트의 경로에 대하여 좀더 자세히 알아본다.

VxD는 ***VPICD_Virtualize_IRQ*** 호출을 통해 IRQ의 디폴트 핸들러를 변경할 수 있다.(하드웨어 인터럽트를 위한 IDT 엔트리를 직접 변경하는 것은 매우 좋지 못한 생각이다.) 어떤 때 VxD는 그 인터럽트의 서비스를 VPICD가 선택한 것과는 다른 VM으로 고르고 싶을 때가 있다. IRQ를 가상화 하기 위한 좋은 이유는 0순위 권한에서 하드웨어 디바이스를 위한 안전한 핸들링을 제공한다는 것이다. 이런 경우 VxD는 어떤 VM에게도 전혀 인터럽트를 전달하지 않도록 조정하는 것이다.

(2) 프로세서 예외 (Processor Exceptions)

인텔은 인터럽트 0h ~ 1Fh를 프로세서 예외를 사용하기 위하여 인텔에서 내부적으로 사용하도록 남겨 두기를 바랬다. 이 이야기는 IBM 시절로 돌아간다. IBM은 인텔의 문서에서 "예약되었음(reserved)"이란 단어를 읽고 "IBM을 위해 예약 되었음(reserved for IBM)"으로 해석하고, 8088에서 사용하면 안 되는 부분을 하드웨어 인터럽트와 BIOS 소프트웨어 인터럽트용으로 사용해 버렸다. 그래서 리얼모드 운영체제나 단독으로 실행되는 도스 확장자는 프로세서 예외를 즉시 구별하지 못한다. 예를 들어 페이지 폴트 INT 0Eh와 플로피 디스크 콘트롤러 IRQ 6에 의해 발생되는 인터럽트가 그것이다. 윈도우즈 95는 이러한 모호성을 제거하기 위해 하드웨어 인터럽트를 50h ~ 5Fh로 옮겼으며 INT *n*명령을 막기 위해 기특하게도 DPL=0 게이트 엔트리를 사용하였다.

VxD는 ***Hook_PM_Fault*** 호출을 통해서 보호모드 예외를 위한 디폴트 핸들러를 변경할 수 있다. 이것은 윈도우즈 95 커널 같은 코드가 예외를 후크하기 위해 DPMI 함수 0203h를 사용했을 때와 같다. ***Set_PM_Int_Vector***의 호출을 통해 IDT 게이트를 변경하는 것은 디폴트 핸들러의 정상적인 연결을 파괴할 수 있기 때문에 사용하지 않도록 한다. VxD는 ***Hook_V86_Fault*** 호출을 통해 V86모드 예외의 디폴트 핸들러도 변경할 수 있다. 표준 VxD는 예외 중의 일부를 훅킹하기 때문에 자작한 VxD를 시스템에 넣기 전에 조정해야 할 것들에 대한 좋은 예제가 될 것이다.

■ 예외 00h, 0으로 나눔 (Exception 00h, Devid Error) : 디폴트 동작은 인터럽트를 VM으로 전달한다. 그래서 응용 프로그램의 런타임 환경에 설치된 핸들러는 응용 프로그램을 강제로 종료시킨다.

■ 예외 01h, 디버거 호출 (Exception 01h, Debugger Call) : 디폴트 동작은 인터럽트를 VM으로 전달하는데, 통상 VM은 이 인터럽트를 무시할 것이다. 응용 프로그램 수준의 디버거는 프로그램을 싱글 스텝으로 실행시키거나 디버깅 레지스터에 있는 브레이크 포인트 주소를 조정하기 위해 이 인터럽트를 후크 한다.

■ 예외 02h, NMI 인터럽트 (Exception, Nonmaskable Interrupt) : 디폴트 동작은 이 인터럽트를 무시한다.

■ 예외 03h, 브레이크 포인트 (Exception 02h, Break point) : 디폴트 동작은 VM으로 인터럽트를 전달하며, VM은 주로 이 인터럽트를 무시한다. 응용 프로그램 수준 디버거는 단일 바이트 브레이크 포인트(single byte break point)를 설정하기 위해 이 인터럽트를 후크 한다.

■ 예외 04h, INTO 명령으로 검출한 오버플로 (Exception 04h, INTO-Detected Overflow) : 디폴트 동작은 VM으로 인터럽트를 전달한다.

■ 예외 05h, 경계영역 초과 (Exception 05h, BOUND Range Exceeded) : 디폴트 동작은 VM으로 인터럽트를 전달한다.

■ 예외 06h, 잘못된 명령어 (Exception 06h, Invalid Opcode) : 이것은 보통 윈도우즈 95가 브레이크 포인트 설정에 사용하는 ARPL 명령을 V86 코드가 실행했을 때 발생한다. 이런 경우의 디폴트 동작은 각 VxD에 있는 브레이크 포인트 콜백 함수를 호출하며, 이 콜백 함수는 적당한 방법으로 VM 레지스터를 조정하게 된다. 다른 경우의 디폴트 동작은 VM을 끝내게 한다. 윈도우즈 95 커널은 시스템 VM이 파괴는 되는 것을 막기 위해 이 인터럽트를 항상 후크 한다.

■ 예외 07h, 코프로세서 없음(Exception 07h, Coprocessor Not Present) : 이 예외에 대한 디폴트 동작은 VM으로 전달되어야 하겠지만, VCPD(Virtual Coprocessor Device)는 이 예외를 후크한다. 코프로세서가 하나라면 VCPD는 실제의 코프로세서를 가상화 하기 위해 CR0 레지스터의 태스크 선택 플래그(task switch flag)를 사용하며, 코프로세서가 없는 경우 Win32 응용 프로그램에게 이를 에뮬레이트 해주기 위해 VWIN32(KERNEL의 저수준 함수를 호출하기 위한 VxD)를 조정한다.

■ 예외 08h, 중첩 폴트 (Exception 08h, Double Fault) : 이 예외는 시스템 콘트롤 테이블에 심각한 문제가 발생했음을 가리키며, 시스템의 안전 보장이나 모순되지 않은 레지스터의 사용을 위해 발생되는 태스크 게이트의 벡터를 통한 인터럽트이다. 그러나 이 인터럽트에 대한 핸들러는 아무 것도 처리하지 않는데 이것은 이 폴트의 원인이 무엇이든 사용자가 지칠 때까지 재 시도하거나 기계를 끄기 전까지는 계속 발생할 것이라는 뜻이다.

■ 예외 0Ah, 무효한 태스크 상태 세그먼트(Exception 0Ah, Invalid Task State Segment) :이 폴트는 발생하지 않을 것이다. 그러나 만약 발생했다면, 이것은 윈도우즈 95가 파괴되었다는 것이다. 어쨌든 실제로 이

인터럽트는 VM으로 전달된다.

■ 예외 0Bh, 세그먼트가 존재하지 않음(Exception 0Bh, Segment Not Present) : 이 예외는 V86 모드에서는 발생할 수 없다. 보호모드에서의 디폴트 동작은 VM을 파괴하는 것이다. 윈도우즈 95 커널은 이 폴트를 항상 후크하는데, 이것은 세그먼트가 존재하지 않는 예외에서 시스템 VM은 16비트 프로그램의 LOADONCALL 세그먼트를 로드 하도록 되어 있기 때문이다.

■ 예외 0Ch, 스택 예외 (Exception 0Ch, Stack Exception) : 보호모드 스택 폴트에 대한 디폴트 동작은 해당 VM을 파괴하는 것이다. 그러나 윈도우즈 95 커널은 시스템 VM의 파괴를 방지하기 위해 이 폴트를 항상 후크 한다. V86 모드에서 이 폴트를 발생시키면, 이 폴트는 무한 루프를 돌게 된다. 왜냐하면 디폴트 핸들러는 폴트 된 명령어를 재실행하기 때문이다.

■ 예외 0Dh, 일반 보호(Exception 0Dh, General Protection) : 이 폴트는 한가지의 디폴트 동작이 있는 것이 아니며 GP 폴트는 매우 서로 다른 것이 발생할 수 있고 정상적인 경우에도 많이 발생한다. VMM 뿐만 아니라 어떠한 VxD도 이 폴트를 예상할 수 없고 이 폴트 발생 시 VM을 파괴한다. 윈도우즈 95의 커널은 시스템 VM이 파괴되는 것을 막기 위해 이 인터럽트를 항상 후크 한다.

■ 예외 0Eh, 페이지 폴트 (Exception 0Eh, Page Fault) : 이 인터럽트에 대한 디폴트 동작은 CR2 레지스터에 있는 폴트가 발생한 가상 어드레스를 읽어서 물리 메모리에 로드 하는 것이다.

■ 예외 10h, 부동 소숫점 에러 (Exception 10h, Floating-Point Error) : 윈도우즈 95에서는 이 예외가 발생하지 않는다. 왜냐하면 윈도우즈 95는 CR0 레지스터의 인에블 비트를 변경하지 않기 때문이다.

■ 예외 11h, 정렬 체크 (Exception 11h, Alignment Check) : 윈도우즈 95에서는 이 예외가 발생하지 않는다. 왜냐하면 윈도우즈 95는 CR0 레지스터의 인에블 비트를 변경하지 않기 때문이다.

■ 예외 12h, 머신 체크 (Exception 12h, Machine Check) : VMM은 이 펜티엄 예외에 대해 해당 VM을 파괴하는 것이 아니라 현재의 VM으로 전달한다.

(3) V86 모드의 소프트웨어 인터럽트 (V86-Mode Software Interrupts)

V86 모드에서 소프트웨어 인터럽트의 디폴트 동작은 0000h:0000h의 인터럽트 벡터 테이블을 통해 VM으로 전달하는 것이다. VxD는 ***Hook_V86_Int_Chain***을 사용해 인터럽트를 중재할 수 있다. 예를 들어, VMM과 표준 VxD들은 함수 16xxh를 걸러내기 위해 INT 2Fh를 후크 한다. 또한 SHELL 가상 디바이스는 클립보드 엑세스를 위해 함수 17xx를 걸러내는데 INT 2Fh를 후크 한다. 또 어떤 VxD는 제작자가 지정한 API 진입점(Vendor-Specific API Entry Point)을 얻는데 사용하는 함수 168Ah를 보기 위해 INT 2Fh을 후크 한다.

(4) 보호모드 소프트웨어 인터럽트 (Protected-Mode Software Interrupts)

보호모드에서 소프트웨어 인터럽트에 대한 디폴트 동작은 V86 모드에 있는 VM에게 인터럽트를 전달하는 것이다. 윈도우즈 3.0과 3.1에서 디폴트로 사용되는 이것은 MS-DOS와 BIOS가 응용 프로그램에서 사용하는 소프트웨어 인터럽트의 대부분을 핸들링 한다. 엄밀히 말해서 이것은 여전히 윈도우즈 95에서도 디폴트이지만, VxD들은 가능한 한 보호모드에서 인터럽트를 처리하기 위해 이들의 대부분을 후크 한다. 이것의 중요한 예 중의 하나가 MS-DOS나 Win16 프로그램에서 사용하는 시스템 서비스인 INT 21h이다. 이전 버전의 윈도우즈에서 VMM은, 특히 파일 엑세스에 관계된 것, 포인터 인자를 V86 메모리에 맞도록 적당히 변경한 후 V86 모드에 있는 MS-DOS에 넘긴다. 윈도우즈 95에서 IFSM(Installable File System Manager)는 파일 시스템 요구를 가로채서 0순위 권한에서 처리한다.

6.2.5 이벤트 (Events)

대부분의 운영체제에서처럼 VMM은 이벤트 큐(queue of event)를 생성하고 서비스하기 위한 내부 메커니즘이 필요하다. 여기서 ***이벤트(event)***라 함은 그것이 발생했을 때 VMM이 안전하고 적절하게 수행할 수 없는 일의 단위이다. 간단한 예로, 한 VM이 어떤 동작의 완료를 기다리고 있는데 이 동작의 완료를 가리키는 하드웨어 인터럽트가 다른 VM이 운영되는 동안 발생할 수 있다. 이러한 경우 인터럽트 핸들러는 이것을 실행하기 위해 다른

VM의 컨텍스트에서 이벤트 콜백을 스케줄 해 놓는다.

그러나 VxD가 이벤트를 큐에 넣는 주된 이유는 페이징을 처리해야 하기 때문이다. 하드웨어 인터럽트는 인터럽트가 인에블 되었을 때는 언제든지 발생한다. 대부분 그렇다. 인터럽트를 놓치지 않기 위해, VMM은 페이징과 관계된 데이터 구조체를 처리하는 동안 인터럽트를 인에이블 된 상태로 유지한다. (사실상 VMM과 VxD는 매우 짧은 임계구역의 코드를 제외하고는 인터럽트를 인에이블 상태로 해 놓은 채 인터럽트를 빠져 나온다. 그러나, 이 단락에서는 페이징과 관계된 것에 대해서만 논의한다.) 비록 MS-DOS는 달리 0순위 권한 프로그램이 페이징 I/O를 처리하는 윈도우즈 95라고 하더라도 페이징 시스템은 재진입 할 수 없기 때문에 하드웨어 인터럽트는 재귀적으로 발생하지 않는다

하드웨어 인터럽트 핸들러는 특별히 비동기적으로 설계된 이러한 VxD 서비스만을 사용할 수 있다. 비동기 서비스는 어떠한 페이징 동작도 유발하지 않음을 보장한다. 이것은 록 된 메모리 페이지를 점유하고 록 된 페이지 구조체만을 참조한다는 것을 의미한다. 더욱이 비동기 서비스는 비동기 VxD 서비스만을 이용한다.

대부분의 하드웨어 인터럽트 핸들러는 비동기가 아닌 동작을 수행해야 한다. 예를 들어 많은 핸들러는 동작 완료를 3순위 권한 코드에게 통보해야 한다. 그러나 그렇게 사용되는 서비스들을 하드웨어 인터럽트 핸들러에 사용하는 것은 절대적으로 불안하다. 이벤트 큐 메커니즘은 하드웨어 인터럽트 핸들러가 미뤄 놓은 작업 목록을 안전할 때 스케줄 하도록 하는 탈출구를 제공한다.

이벤트 큐 후면의 기본적인 아이디어는, 특히 하드웨어 인터럽트를 서비스하고 있는 동안, VxD가 언제든지 이벤트 큐를 생성할 수 있다는 것이다. VMM이 페이지 폴트를 충분히 소화해 낼 수 있는 안정점에 도달했을 때 VMM은 이벤트 큐를 조사하고 큐에 있는 이벤트와 연결된 콜백 루틴을 호출한다. 콜백 루틴은 이 시점에서 어떠한 VxD 서비스도 호출할 수 있다. 그래서 모든 서비스를 발생시킬 수 있는 것이다. 모든 이벤트가 다 서비스되었을 때 VMM은 최종적으로 어떤 응용 프로그램을 최종적으로 후처리하기 위한 IRETD 명령을 생성한다. 제어권을 다시 얻은 응용 프로그램은 최초로 인터럽트 된 것일 수도 있고 스케줄러가 동작하기로 결정한 새것일 수도 있다. VMM은 실행되는 컨텍스트에 따라 이벤트를 분류한다.

- ***글로벌 이벤트(global events)***는 부분적인 VM, 프로세스, 쓰레드보다는 시스템 전체에 적용된다. VMM은 큐에 있는 글로벌 이벤트를 우선적으로 실행한다.
- ***로컬 이벤트(local events)***는 부분적인 VM, 프로세스, 쓰레드에 적용되며 그 컨텍스트에서 운용되어야 한다. VMM은 글로벌 이벤트를 처리한 후 현재의 VM, 프로세스, 쓰레드에 대한 큐에 있는 어떠한 로컬 이벤트라도 실행한다. 이벤트 콜백 중의 하나는 VM이나 쓰레드 전환을 발생시킬지도 모른다. 이렇게 되면 VMM은 새로운 로컬 컨텍스트에 대한 새로운 이벤트 큐를 처리하게 될 것이다.

6.2.6 겹실행(Nested Execution)

이벤트는 VMM의 실행 중심 경로를 간단한 "서비스와 IREDT" 패러다임의 전형적인 예와는 전혀 다른 방법이다. 겹실행(Nest Execution)은 또 다른 방법으로써, 인터럽트 된 코드와는 다른 코드를 VMM이 후처리 할때 발생한다. 이 상황에서 "네스팅(nesting)"은 특별히 처리된 코드가 VMM으로 리턴하기 때문에 발생하는 것으로(또 다른 인터럽트를 발생 시켜서), 원래의 중심 경로를 되찾고, 가장 바깥의 인터럽트로부터 리턴 하게 된다. 기술적인 이유로 인하여 겹실행 블록은 이벤트 콜백 루틴의 한계가 발생한다.

이 겹실행의 개념은 예제가 없이는 혼란스러울 것이다. 그림 6-3과 다음의 설명은 이 개념을 명확히 하는데 도움을 줄 것이다. 윈도우즈 응용 프로그램은 NetBIOS 요구를 수행하기 위해 INT 5Ch를 발생시킬 것이다. 그러나 NetBIOS 제공자는 리얼모드에 있는 진부하고 낡은 네트워크 드라이버일 수 있다. 이런 경우 DPL이 맞지 않기 때문에 INT 5Ch 명령어는 GP 폴트를 발생시킨다. VNETBIOS 드라이버는 INT 5Ch를 후크해 있기 때문에 결국 제어권을 얻게 된다. 그래서 리얼모드 INT 5Ch 핸들러를 실행하기 위해 겹실행 블록으로 진입할 것이다. 결국 리얼모드 핸들러는 NetBIOS를 호출한 곳으로 되돌아가기 위해 IRET 명령을 수행했을 때, 실제는 잘못된 명령 예외(Invalid Opcode exception)를 발생시키는 ARPL 명령이라는 것을 알고 있으므로, 겹실행 블록으로부터 빠져 나와서 VMM으로 제어권을 되돌리도록 한다. 얼마 후 VMM은 INT 5Ch를 호출한 원래의 보호모드 응용 프로그램을 재처리하게 된다.

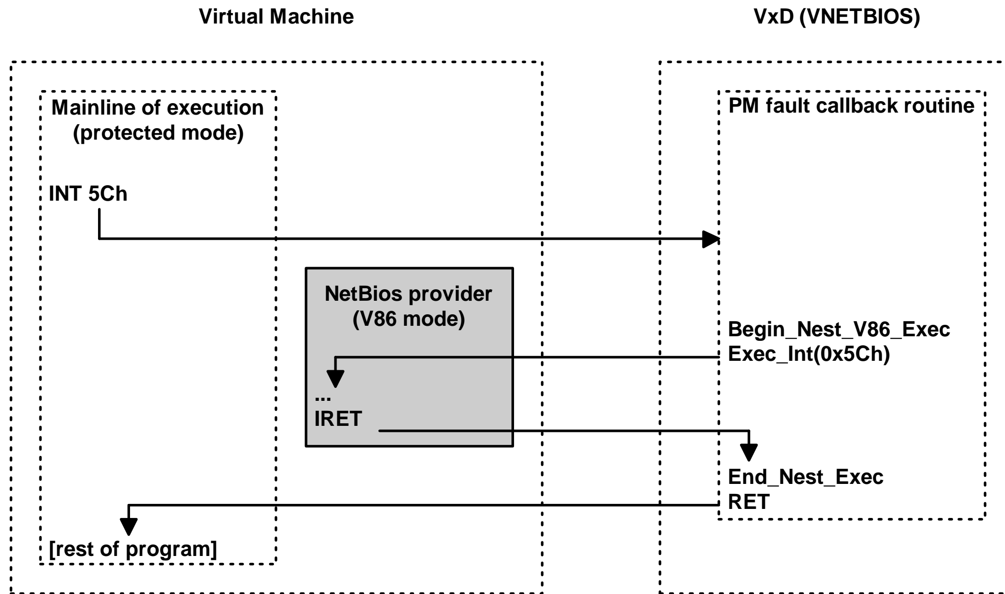

그림 6-3. VNETBIOS가 겹실행을 사용하는 방법
(VMM이 제어권을 가지는 중간 단계는 그림에 나타내지 않았음)

6.3 쓰레스 스케줄 잡기 (Scheduling Threads)

VMM은 다중 쓰레드와 선점적 멀티태스킹을 수행하기 위한 요소 중의 하나로 2개의 스케줄러를 사용한다. ***1차 스케줄러(primary scheduler)***라 하는 것은, 다음으로 실행할 쓰레드를 선택한다. 기본적으로 1차 스케줄러는 적절한 쓰레드를 골라 놓은 리스트 중 제일 높은 우선권을 가진 쓰레드를 선택한다. 이 선택은 VMM이 제어권을 가진 동안(인터럽트를 처리하는 동안) 이루어진다. 그리고 이 결과는 나중에 사용자 코드를 다시 처리하게 되었을 때 VMM이 저장할 레지스터를 결정한다. (역자 주 : 리턴 될 VM을 선택한다는 뜻이다.) VxD는 이 선택을 좌우하는 쓰레드의 실행 우선권을 조정하기 위해 1차 스케줄러를 사용한다. 또한 VxD는 스케줄링에 쓰레드를 적당하거나 부적당하게 만들기 위해 다양한 동기화 프리미티브를 사용한다.

1차 스케줄러의 클라이언트 중의 하나가 바로 ***2차 스케줄러(secondary scheduler)***인 ***시분할 스케줄러(time-slicing scheduler)***이다. 시분할 스케줄러는 복잡한 우선권 체제를 바탕으로 하여 쓰레드에게 원탁형(round-robin)방식으로 프로세서를 할당하는데 1차 스케줄러를 사용한다. 또한 이것은 KERNEL32.DLL과 VWIN32 VxD 사이의 내부 연결에 유용한 ***SetThreadPriority***와 ***SetPriorityClass***와 같은 응용 프로그램 수준의 API도 서비스해 준다.

6.3.1 실행 우선권과 Win32 우선권 (Execution and Win32 Priorities)

1차 스케줄러와 시분할 스케줄러는 서로 다른 우선권 체제를 사용한다. 쓰레드의 ***실행 우선권(execution priority)***은 1차 스케줄러가 다음에 제어권을 줄 것이지 말 것인지를 결정한다. 이 결정은 전부냐 하나도 없느냐(all-or-nothing)이다. (역자 주 : 1차 스케줄러는 결정된 쓰레드에 대해서 시간은 생각하지 않고 무조건 제어권을 준다는 뜻이다.) 가장 높은 우선권의 쓰레드가 실행하게 된다. 윈도우즈 95는 단일 프로세서 운영체제이기 때문에 한 쓰레드가 실행하고 있는 동안에 다른 쓰레드가 실행할 수 없다. 즉, 그 쓰레드가 제거될 때까지 한 쓰레드만 실행하는 운영체제인 것이다. 이것은 매우 좋은 멀티태스킹 시스템은 아니다. 그러나, 실행 우선권은 VxD들이 *우선권 부양(priority boosts)*을 적용하고 제거함에 의해 계속적으로 변한다. 예를 들어, 8h의 우선권을 가진 쓰레드가 임계구역(critical section)으로 진입했다. 이 쓰레드의 이 동작을 중지하고 임계구역으로부터 해제하기 위해 VMM은

CRITICAL_SECTION_BOOST값(00100000h)을 더해서 쓰레드의 우선권을 부양한다. 새로운 실행 우선권 00100008h을 가진 쓰레드는 이전보다 더 실행할 가능성이 높다. 시분할 스케줄러도 또한 실행할 쓰레드를 선발하는데 우선권 부양을 적용한다.

표 6-1은 윈도우즈 95에서 사용되는 실행 우선권 부양을 나타내었다. 표 6-1에 나타낸 것보다 더 많은 양의 실행 우선권 부양을 할 수 있지만, 보통은 그렇게 하지 않는다. 1차 스케줄러는 RESERVED_LOW_BOOST와 RESERVED_HIGH_BOOST를 내부적으로 사용하기 때문에 쓰레드 부양이 이 값들을 사용할 수 없을 것이다. 시분할 스케줄러는 원탁형 체제(round-robin scheme)에 의해 다음에 실행될 것이라고 선택된 쓰레드에는 임시적으로 CUR_RUN_VM_BOOST 값이 사용된다. ***Begin_Critical_Section*** 서비스는 자동으로 쓰레드를 CRITICAL_SECTION_BOOST 값으로 부양하며, ***End_Critical_Section***은 자동으로 부양을 제거한다. 이와 비슷하게, VPICD는 하드웨어 인터럽트 핸들러가 실행되는 동안 인터럽트를 가능한 한 빨리 지우기 위해 TIME_CRITCAL_BOOST를 적용한다. 시간 임계 부양(time-critical boost)은 숫자적으로 임계 구역부양(critical-section boost)보다 크다. 이것은 하드웨어 처리가 임계구역에 의해 보호되는 동작의 완료보다 우위를 점하고 있다는 것을 의미한다.

심볼 이름	*숫자 값*
RESERVED_LOW_BOOST	00000001h
CUR_RUN_VM_BOOST	00000004h
LOW_PRI_DEVICE_BOOST	00000010h
HIGH_PRI_DEVICE_BOOST	00001000h
CRITICAL_SECTION_BOOST	00100000h
TIME_CRITICAL_BOOST	00400000h
RESERVED_HIGH_BOOST	40000000h

표 6-1. 실행 우선권 부양 값들

다른 두 개의 부양(LOW_PRI_DEVICE_BOOST와 HIGH_PRI_DEVICE_BOOST)의 뜻을 이해하기 위해서는 시분할 스케줄러의 우선권 모델에 대한 이해를 해야한다. 시분할 스케줄러는 윈도우즈 NT에서 만든 Win32 우선권 모델(Win32 priority model)을 사용한다. 이 모델에서는, 표 6-2와 같이 한 쓰레드는 5개의 우선권 클래스(priority class)중의 하나를 가지고 있으며, 우선권 범위는 부분적으로 겹치는데, 예를 들어 IDLE_PRIORITY_CLASS의 높은 우선권 쓰레드는 NORMAL_PRIORITY_CLASS의 낮은 우선권 쓰레드보다 절대적으로 높은 우선권을 가지고 있다. 응용 프로그램(KERNEL32 같은 3순위 권한 코드를 포함)은 디폴트값으로부터 쓰레드의 클래스를 바꾸기 위해 Win32 API 함수인 ***SetPriorityClass***를 사용하며, 그 클래스에서 유효한 범위의 쓰레드 우선권을 변경하기 위해 ***SetThreadPriority***를 사용한다.

쓰레드 우선권 (THREAD_-PRIORITY_*xxx*)	우선권 클래스 (*xxx*_PRIORITY_CLASS)				
	IDLE	NORMAL	NORMAL	HIGH	REALTIME
IDLE	01h	01h	01h	01h	10h
LOWEST	02h	05h	07h	0Bh	11h~16h
BELOW_NORMAL	03h	06h	08h	0Ch	17h
NORMAL	04h	07h	09h	0Dh	18h
ABOVE_NOMAL	05h	08h	0Ah	0Eh	19h
HIGHEST	06h	09h	0Bh	0Fh	1Ah
TIME_CRITICAL	0Fh	0Fh	0Fh	0Fh	1Bh~1Fh

표 6-2. Win32 우선권 클래스들

예를 들어 다음의 Win32 호출은 쓰레드의 Win32 우선권을 10h로 설정하는 것이다.

```
SetPriorityClass(hThread, REALTIME_PRIORITY_CLASS);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
```

시분할 스케줄러는 가장 높은 우선권을 공유하는 모든 쓰레드들을 차례로 시간을 나눠줌으로써 CPU를 분할한다. 이에 대한 기본 알고리즘은 제일 높은 우선권 쓰레드의 리스트를 만드는 것과 첫 쓰레드에게 우선권 부양을 주는 것이다. 이와 동시에 시분할 스케줄러5는 시간이 만료된 쓰레드를 선점하기 위해 타이머 스케줄러를 스케줄한다. 그로 인하여 스케줄러는 잠시동안 다음의 쓰레드의 우선권을 끌어올릴 것이다. 시분할 스케줄러는 또한 시스템을 매끄럽게 유지하기 위해 추가적인 알고리즘을 적용한다. 예를 들어

- 만약 적당한 길이의 시간동안 실행하지 않는 쓰레드가 있다면 스케줄러는 그 쓰레드가 굶어 죽지 않도록(to prevent starvation) 우선권을 부양한다. 적당한 시간을 만든다는 것은 평균적인 시간조각의 길이, 얼마나 많은 쓰레드가 실행에 적당한가, 내부적인 알고리즘 상수 등에 따라 달려있다.
- 만약 어떤 쓰레드가 자원(세마포어 같은)을 가지고 있어서 제일 높은 우선권을 가진 쓰레드를 막고 있다면, 스케줄러는 자원을 소유하고 있는 쓰레드의 우선권을 막힌 쓰레드와 같은 수준으로 끌어올린다. 이러한 우선권 이양(priority inheritance)은 자원의 낮은 소유자가 자원을 필요로 하는 높은 우선권 쓰레드로부터 벗어나도록 해준다.
- 만약 어떤 쓰레드가 다른 쓰레드를 막고 있다면, 스케줄러는 나중에 이 쓰레드가 다른 쓰레드를 막지 않을 때 우선권을 부양한다. 이것은 자원에 대해 투쟁하는 쓰레드에게 프로세스 시간의 동등한 평균량을 제공함으로써 자원을 가지고 경쟁하지 않도록 하는 효과가 있다.
- 스케줄러는 때때로 시간이 만료되었을 때 우선권을 부양된 값에서 베이스 값으로 낮춘다. 이와 다른 경우에 스케줄러는 처음으로 부양한 태스크가 완료된 후에 간단히 쓰레드를 부양 값을 낮춘다.

정리하면 시분할 스케줄러는 0h에서 1Fh의 범위의 절대적인 우선권을 생성하기 위해 자신의 알고리즘 규칙뿐만 아니라 Win32 응용 프로그램의 명령에도 따른다. 이것은 1차 스케줄러의 ***Adjust_Thread_Priority*** 서비스 호출을 통해 쓰레드의 실행 우선권을 만들며, 이것은 다음으로 실행할 가장 접합한 쓰레드를 선택하는 1차 스케줄러를 설득하는데 실행 우선권 부양을 적용한다.

그것은 효과적으로 Win32 우선권 체제를 오버라이드하기 위해 하드웨어 인터럽트와 임계구역 부양은 실행 우선권을 수정한다. (비록 둘 다 임계 시간 부양을 가진 같은 Win32 우선권의 쓰레드를 가지고 있다고 하더라도 그것들은 시분할에 의해 프로세스를 공유할 것이다.)

VxD는 쓰레드가 주어진 시간에 동작을 수행할 수 있는 가능성을 증가시키기 위해 ***LOW_PRI_DEVICE_BOOST***를 사용한다. 그래서 ***NORMAL_PRIORITY_CLASS***의 우선권 클래스에 있는 ***THREAD_PRIORITY_IDLE*** 쓰레드에 이 부양을 적용하면 쓰레드의 절대 우선권은 1에서 11h로 증가된다. 이렇게 되면 ***REALTIME_PRIORITY_CLASS***는 아니지만 모든 쓰레드에 우선하게 된다. 그러나 ***HIGH_PRI_DEVICE_BOOST***를 적용하면 같은 쓰레드의 우선권이 1h에서 1001h로 증가된다. 이것은 역시 ***REALTIME_PRIORITY_CLASS*** 쓰레드에 우선한다. 어떤 경우든 더 높은 부양을 가진 쓰레드 뒤로 밀려난다.