CHAPTER

04

가상화 지원

4.1 가상화의 목적, 장단점

4.2 가상화를 실현하기 위해

4.3 가상화를 지원하는 하드웨어 구조

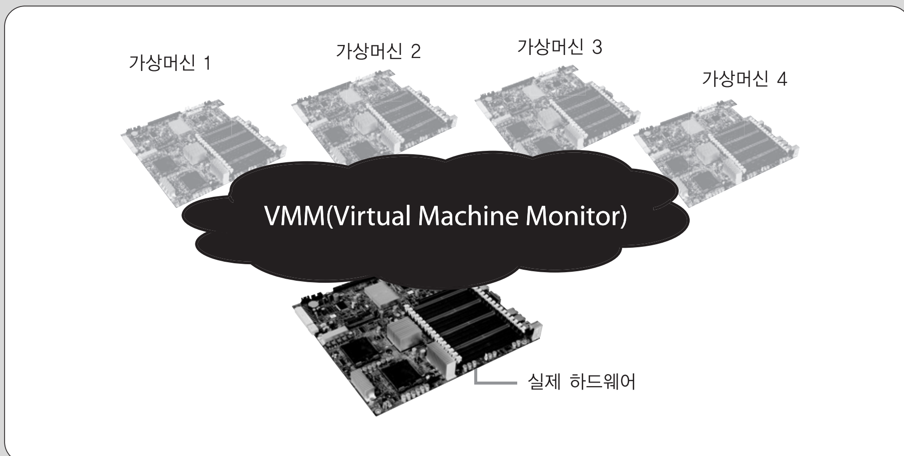
4.4 정리

제2장에서 설명한 바와 같이 멀티유저에서는 메모리 관리기구로 인한 액세스 제한으로 유저 간 분리를 수행하고 있다. 그러나 이 방법은 하나의 OS를 기반으로 한 멀티유저로, 예를 들면 윈도우와 리눅스 같이 복수의 OS를 1대의 컴퓨터로 동시에 작동시킬 수는 없다. 윈도우 XP부터 윈도우 7으로 갈아타려는 경우에도 이행기간 동안에는 양쪽 OS를 모두 작동시킬 수 있으면 편리하지만 이것도 할 수 없다. 또한 기업 등에서는 일상업무 운용은 계속해가면서 새로운 업무시스템을 개발하는 일이 자주 있지만, 개발 중인 프로그램의 버그로 운용 중인 시스템이 다운되면 곤란하다.

위와 같은 용도를 지원하는 것이 『서유기』의 손오공은 아니지만 분신술로 하나의 하드웨어를 다수인 것처럼 보이게 하는 '가상화'(Virtualization)다. 분신술을 사용하는 것은 VMM(Virtual Machine Monitor) 혹은 하이퍼바이저(Hypervisor)라고 하는 OS와 하드웨어(프로세서나 I/O) 사이에 들어가는 소프트웨어다.

가상화가 인기를 끌기 시작하게 된 것은 최근이지만, 오래 전인 1967년에 IBM 메인프레임용으로 가상화 모니터가 만들어졌으니 실제로는 40년 이상의 역사가 있는 기술이다. 가상화의 기본적인 구조에 대해서는 3.2절 후반부에서 간단히 설명했으므로 이번 장에서는 가상화 기술의 목적이나 장/단점 등을 자세히 살펴보도록 하자.

● **그림 4.A** VMM은 분신술로 다수의 가상머신을 만든다※



※ 이미지 제공: 인텔(주)
인텔 SR1560SF 보드의 일부분(Intel Server Board S5400SF)

4.1 가상화의 목적, 장/단점

오랜 역사가 있는 가상화 기술은 어떤 목적을 위해 만들어졌을까? 장/단점과 함께 살펴보도록 하자.

가상화의 기초지식(복습)

이 장의 머리글 및 3.2절에서 언급한 대로 '가상화'는 하나의 하드웨어를 여러 개인 것처럼 보이게 하는 기술로, VMM 혹은 **하이퍼바이저**라고 하는 OS와 하드웨어(프로세서나 I/O) 사이에 들어가는 소프트웨어를 통해 실현된다.

VMM은 하드웨어 자원을 관리하고 OS가 메모리 영역 할당과 같은 메모리 관리나 I/O 동작에 관계되는 레지스터 등을 조작하려고 하면, 해당 하드웨어 조작을 인터셉트(Intercept, 가로채기)해서 복수의 OS를 실행하는 경우에도 모순이 없도록 실행하도록 한다. 그래서 OS에서 보면 직접 하드웨어를 조작하고 있는 것과 동일한 결과를 얻을 수 있고, OS로의 응답도 하드웨어를 조작한 경우와 동일하다. 따라서 VMM에서 동작하는 각각의 OS로서는 마치 하드웨어를 점유하고 있는 것처럼 보이는 환경이 제공되는 것이다.

그러나 가상화에는 하드웨어에서 지원될 필요가 있는 전제가 몇 가지 있으며, 소프트웨어만으로는 가상화를 실현할 수 없다.[주1] 또한 이와 같은 가상화를 수행하기 위해서는 VMM이 여러 작업을 수행해야 할 필요가 있으므로 애플리케이션에서 사

주1 Xen이나 VMware도 인텔의 VT와 같은 기능을 이용해서 성능을 개선하는 작업을 하고 있다.

용할 수 있는 프로세서 성능이 줄어들게 된다. 이 때문에 최신 프로세서에서는 인텔의 VT[1]나 AMD의 AMD-V[2]와 같이 가상화의 오버헤드를 줄이는 하드웨어 기구를 강화해오고 있다.

최근에는 가상화 지원 기능을 갖추고 있는 프로세서가 많아져서 프로세서 지식으로서 가상화 기술의 중요도는 높아지게 되었다.

가상화의 목적

당초에 가상화의 목적은 하나의 OS에서 마치 여러 컴퓨터가 있는 것처럼 보이게 해서 여러 OS를 동작시키는 사용법을 가능하게 하는 것이었다.

가상화 이전인 종래의 시스템에서는 동일 OS에서 오너나 그룹에 따른 파일 액세스 제한이나 메모리 관리를 통해 유저 사이를 분리시켰지만 역시나 견고한 분리는 아니어서 여러 보안문제 등이 있다. 또한 특정 문제로 인해 OS가 다운되면 실행 중인 모든 프로그램이 중지되어 버린다.

멀티소켓(제5장 참조)의 대형 서버 시스템에서는 프로세서 그룹마다 액세스할 수 있는 메모리나 I/O를 하드웨어적으로 전환해서 분리하는 '하드웨어 파티션'이라는 분할방법이 있다. 하드웨어 파티션은 하드웨어 자체를 분할해서 제공함으로써 분리는 완전하지만 분할할 수 있는 단위가 프로세서 1개 단위처럼 유연성(Flexibility)이 낮은 것이 어려운 점이다.

반면, 가상화의 경우는 1개의 프로세서를 여러 개에서 수십 개의 가상 프로세서로 분할할 수 있고 또한 다이나믹하게 분할을 변경할 수도 있다. 이제부터 그 구조를 자세히 살펴보도록 하자.

유저 간 분리를 견고하게 실현 — 가상화의 장점 1

앞서 종래의 시스템에서 유저 간 분리방법에 대해 언급했는데, 그에 비해 가상화는 '분할된 하드웨어의 이미지'를 제공한다. 즉, 하드웨어 자체는 하나지만 각각의 OS는 전용 하드웨어를 가지고 있는 것처럼 생각해서 동작하는 환경을 만들어내는

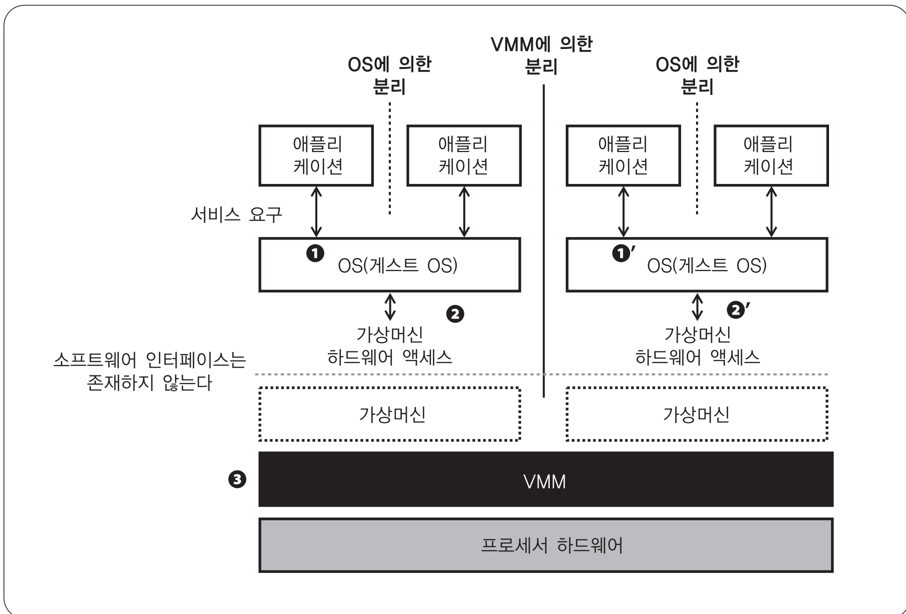
것이다. 이러한 각 OS 전용 하드웨어 이미지를 가상머신(Virtual Machine, VM)이라고 한다.

애플리케이션 프로그램은 OS에 메모리 할당이나 I/O 요청 혹은 새로운 프로세서 실행시작 등 다양한 서비스 요청을 내어놓는다. 물론 OS가 완벽하다면 다른 유저의 애플리케이션에 영향을 주지 않도록 분리할 수 있어야 하지만, 애플리케이션으로부터의 서비스 요청 인터페이스가 많으므로 빠져나갈 구멍도 쉽게 생기게 된다.

반면, VMM을 통한 가상화의 경우는 그림 4.1과 같이 각 OS(❶❶')는 가상머신의 하드웨어를 조작하는 명령을 실행(❷❷')할 뿐으로, VMM(❸)에 대해 소프트웨어에서 서비스 요청을 할 수 있는 형태의 인터페이스는 존재하지 않는다.

물론 VMM이 해킹된다면 분리가 무너질 가능성은 있지만, OS에서 VMM은 보이지 않고 VMM의 서비스를 명시적으로 호출하는 것도 아니므로 애플리케이션으로부터 각종 서비스를 요청받는 OS와 비교하면 VMM에 침입하는 것은 어렵다. 따라서 VMM을 통한 분리는 OS에 의한 분리보다도 견고하게 실현될 수 있다.

● **그림 4.1** VMM은 가상머신을 통해 견고하게 분리를 실현

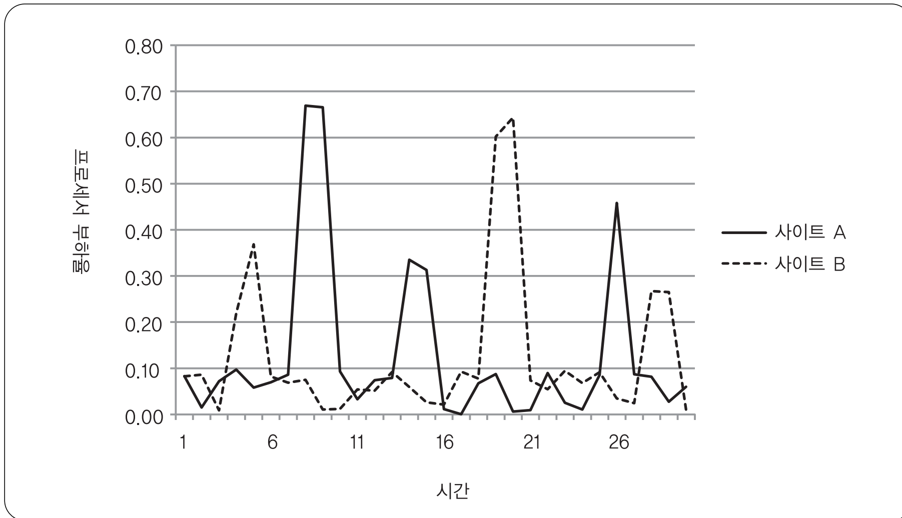


여러 서버를 모아서 가동률을 개선 — 가상화의 장점 2

또한 최근에는 가상화가 갖는 '분할의 유연성'이 좋게 평가되면서 가상화 이용이 확대되고 있다.

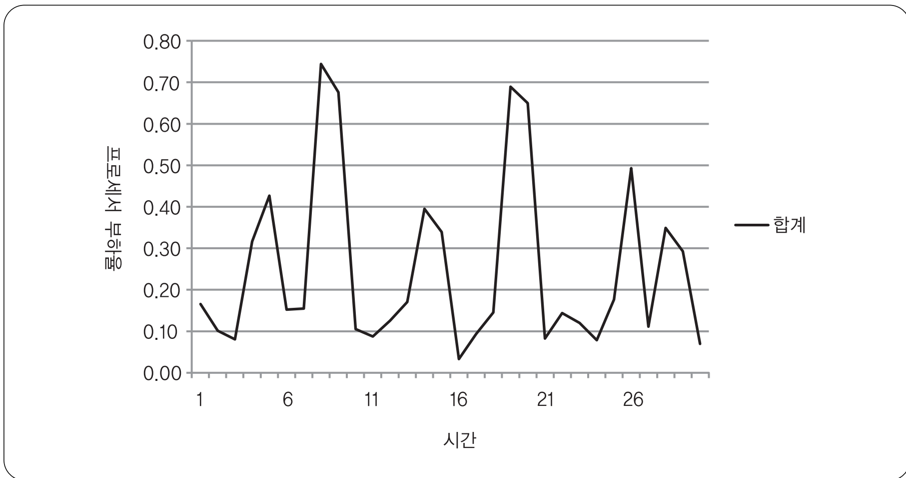
웹서버 등을 생각해보면 그림 4.2와 같이 액세스가 집중된 경우에는 프로세서 파워가 필요하지만, 대부분의 시간은 낮은 레벨의 액세스가 이루어지는 경우가 자주 있다. 이와 같은 웹사이트를 운용하는 서버로서는 피크 시 액세스 부하에 견디는 처리 능력이 필요하지만, 평상시에는 프로세서가 그다지 부하가 걸리지 않으므로 가동률은 낮아지게 된다. 그러나 여러 웹사이트의 피크 액세스가 중첩되는 경우는 확률이 낮으므로 그림 4.2에서 2개 사이트의 처리부하를 합하더라도 그림 4.3과 같이 피크 부하는 거의 증가하지 않는다. 반면 평균적인 부하는 2배가 되므로 가동률을 높일 수 있다.

● **그림 4.2** 2개의 웹사이트 A, B의 부하변동 예



몇 대의 서버를 가상화로 묶으면 적은 대수의 서버로 서비스를 제공(=가동률을 개선)할 수 있게 되므로 구입비용이 내려가고, IDC의 설치면적도 줄일 수 있고, 소비전력이 줄어들어 전기료도 낮아지는 등 경제적으로 상당히 중요한 장점을 얻을 수 있다.

● **그림 4.3** 2개의 웹사이트 A, B의 부하 합계



이와 같이 해서 서버를 묶는 것을 '서버 콘솔리데이션'(Server Consolidation)이라고 한다. 또한 처리 실행을 계속하면서 가상머신을 다른 프로세서로 이동하는 라이브 마이그레이션(뒤에서 설명) 등의 기술이 개발되고 있다. 이러한 기술을 사용하면 부하가 높은 경우는 프로세서를 추가하고, 부하가 낮은 경우에는 일부 프로세서로 작업을 모으고 그 밖의 프로세서 자원을 잘라내어 소비전력을 낮추는 것과 같은 사용방법이 가능하므로 클라우드 데이터센터 등에서는 필수 기술이다.

VMM의 실행 오버헤드 — 가상화의 단점, 주의점

앞에서와 같이 가상화에는 여러 장점이 있지만 'VMM의 실행 오버헤드'라는 문제점도 있다. 그러나 인텔이나 AMD 프로세서에서는 VMM의 오버헤드를 줄이는 하드웨어 기구가 추가되어 오고 있어서 이 오버헤드도 줄어들고 있다.

대량의 메모리가 필요

가상화를 위해 여러 서버를 모으면 메모리나 디스크에 대해서도 각각의 서버에서

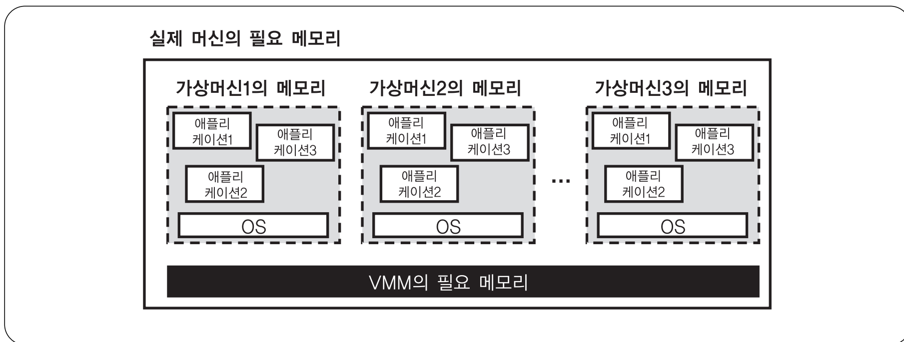
필요로 하는 여분의 합계를 여분으로서 지닐 필요가 없다. 따라서 전체적으로 필요한 용량을 줄일 수가 있다. 그러나 프로세서는 비교적 짧은 시간에 전환될 수 있지만, 메모리는 가상머신을 전환할 때마다 사용하고 있던 메모리를 디스크에 백업, 복원하는 데 시간이 오래 걸려서 프로세서를 유효하게 사용할 수가 없다.[주2]

따라서 그림 4.4에 나타냈듯이 메모리에 관해서는 모든 OS와 그 위에서 동작하는 애플리케이션이 필요로 하는 합계 용량을 확보할 필요가 있으므로 가상화하더라도 프로세서만큼 메모리를 줄일 수는 없다.

메모리가 다소 적은 정도라면 가상 메모리로 커버할 수 있을지 모르지만, 메모리가 부족해서 빈번하게 디스크와의 페이지 교체가 발생하게 되면 성능이 크게 떨어지게 되므로 사용할 수 없게 되어 버린다. 따라서 가상화를 수행하면 1대의 서버가 그 자체로 여러 대로 변하는 것 같은 달콤한 얘기만은 아니며, 구동하게 될 애플리케이션에 따라 다르겠지만 메모리는 대량으로 탑재할 필요가 있다.

그러므로 가상화 보급에 따라 필요 메모리양이 증가하며, 종래의 프로세서에서는 연결할 수 있는 메모리 DIMM 매수가 4매 정도였지만 2010년 3월에 발매된 제온 7500 프로세서에서는 12매의 DIMM을 연결할 수 있게 되어 있다.

● **그림 4.4** 가상화하더라도 필요 메모리는 프로세서만큼은 줄어들지 않는다



주2 보충하자면, 메모리 액세스와 비교해서 디스크 액세스는 훨씬 많은 시간이 걸리므로 메모리가 적으면 느린 디스크 액세스가 빈번하게 일어나서 시간이 걸리므로 성능이 나오지 않는다.

자원 할당관리가 필요 — 웹호스팅 서비스의 예

가상화를 이용해서 웹호스팅 서비스를 하려는 경우, 확률은 낮겠지만 피크가 중첩되지 않으리라고는 단언할 수 없다. 운 나쁘게 피크가 중첩되어 버리면 응답시간이 길어지는 등의 문제가 발생하지만, 통상적으로 이에 대해서는 '운용시간의 99%는 여차여차해서 아래의 응답시간을 보장하지만, 나머지 1%의 경우는 최선의 노력은 하겠지만 응답시간은 보장하지 않는다'와 같은 서비스 수준 합의서(Service Level Agreement)를 고객과 맺음으로써 대처하게 된다. 그렇더라도 A사 사이트에 대한 접속이 많아져서 처리시간이 많이 할당되면 B사 사이트의 평균 응답시간이 길어지게 되어 B사로부터 클레임이 들어오게 된다.

이와 같은 문제가 발생하지 않도록 각 가상머신에 할당되는 프로세서 시간, 메모리양, 디스크 사용량 등의 자원 할당을 관리할 필요가 있다.

4.2 가상화를 실현하기 위해

가상화를 실현하기 위해 OS와 하드웨어 사이에 위치하는 VMM은 메모리 관리나 I/O 제어를 모순 없도록 처리할 필요가 생긴다.

OS에 독립된 (가상) 하드웨어를 제공하는 VMM

가상화에서는 VMM으로 가상머신을 만들어내며 그 위에서 일반 OS가 변경 없이 그대로 동작하는 것이 원칙이다. 따라서 OS로서는 동일한 것이지만 VMM 상에서 작동한다는 것을 확실히 구분하기 위해 이 OS를 '게스트 OS'라고 한다.

반복해서 언급해왔지만, OS는 메모리 관리기구를 조작해서 유저상태로 실행하는 애플리케이션에 메모리를 할당하거나 애플리케이션의 입출력 요청에 대해 I/O를 액세스한다. 그러나 여러 OS를 작동시키는 가상화에서는 각 게스트 OS가 제멋대로 메모리 관리기구나 I/O를 조작하면 모순이 일어나게 된다.

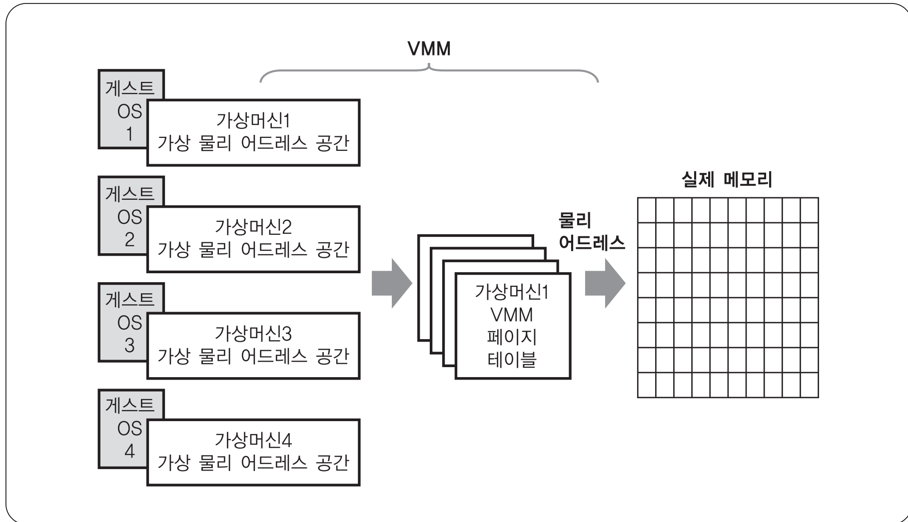
메모리의 가상화

우선은 메모리 관리기구의 가상화 관계를 살펴보면, 게스트 OS 상의 애플리케이션 프로그램이 특정 논리 어드레스인 메모리를 액세스하면 게스트 OS는 자신이 갖고 있는 페이지 테이블을 사용해서 물리 어드레스로 변환하지만, 이대로라면 각각의 게스트 OS가 동일한 물리 어드레스를 사용하게 된다.

따라서 그림 4.5와 같이 VMM은 각 게스트 OS에 대해 '별도의 메모리 공간을 제공'할 필요가 있다. 그리고 게스트 OS가 물리 어드레스라고 생각하는 것은 사실 각

게스트 OS 개별의 '가상 물리 어드레스'로, VMM은 이 가상 물리 어드레스를 자신이 가지는 게스트 OS별 페이지 테이블을 사용해서 실제 물리 어드레스로 변환해서 실제 메모리를 액세스한다.

● **그림 4.5** 각 가상머신의 가상 물리 어드레스를 VMM에서 물리 어드레스로 변환

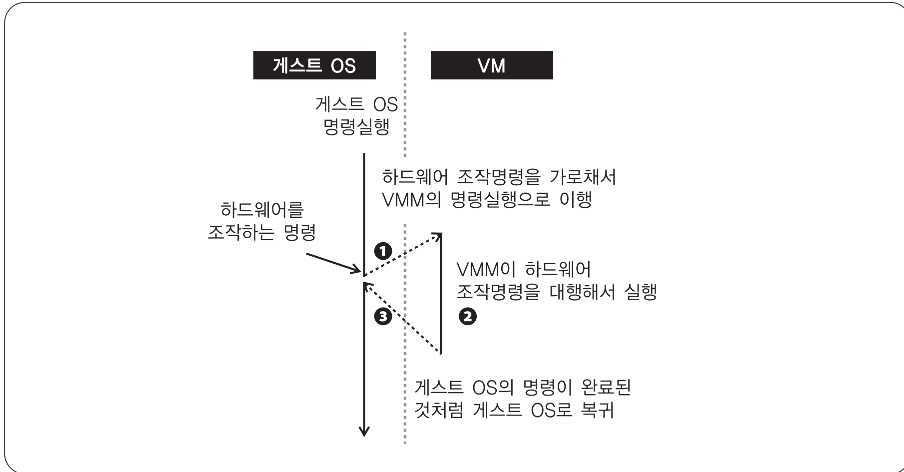


게스트 OS의 하드웨어 액세스를 인터셉트

I/O에 관해서도 여러 게스트 OS가 뒤엉켜서 I/O 컨트롤러의 제어 레지스터를 조작하게 되면 동작이 뒤죽박죽 된다.

이와 같은 추가적인 어드레스 변환을 하거나 I/O 레지스터 조작을 하기 위해서는 그림 4.6에 나타냈듯이, VMM은 각 게스트 OS가 하드웨어를 조작하는 명령을 실행하려고 하는 것을 검출하고(❶) 이를 가로채서 모순이 없도록 대행하여 실제 하드웨어를 동작시키며(❷), 마치 게스트 OS의 명령실행이 완료된 것처럼 해서 게스트 OS로 복귀한다(❸).

● **그림 4.6** 게스트 OS의 하드웨어 조작명령을 가로채서 대행한다※



※ 점선 화살표(❶, ❸)인 하드웨어를 조작하는 게스트 OS 명령은 하드웨어에서는 실행되지 않고 끊어져 있음을 의미하고 있다. 그림 4.7에서도 마찬가지임

4.3 가상화를 지원하는 하드웨어 구조

VMM의 동작은 여러 게스트 OS로부터 하드웨어 자원을 액세스하는 데 모순이 없는 형태로 실행되며, 각 게스트 OS가 전용 하드웨어를 갖고 있는 것처럼 보이게 하는 것이다. 이를 효율이 좋게 실행하기 위해서는 각종 하드웨어 기구가 필요해진다.

하드웨어 조작명령 검출

가상화를 위해서는 먼저, VMM으로서는 각 게스트 OS가 하드웨어를 조작하려고 하는 것을 검출할 필요가 있다.

특권위반 예외처리

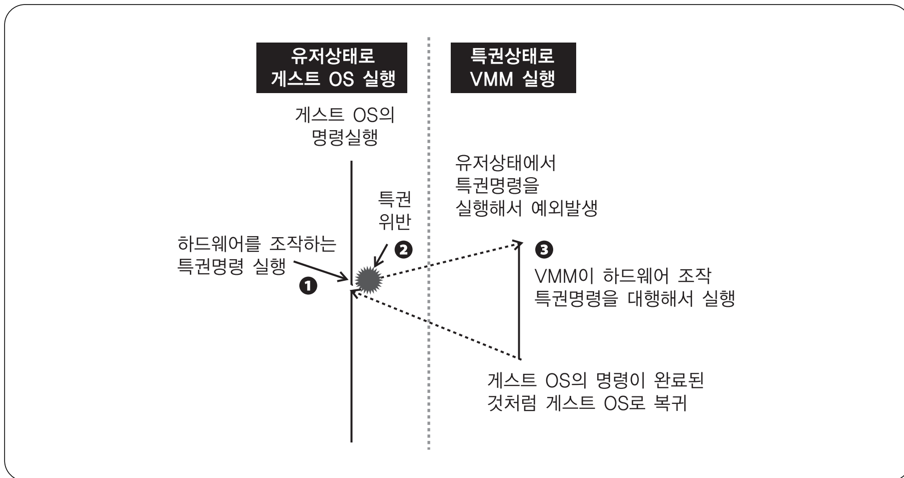
이것을 실현하는 기본적인 기구는 '특권 상태'와 '유저상태'의 구분이다. 통상 OS는 특권 상태로 동작시키지만, 가상화를 수행할 경우는 'VMM이 특권 상태'로 실행되며 '게스트 OS는 유저상태'로 실행되게 된다. 이 상태에서 게스트 OS가 메모리 관리기구 등의 하드웨어를 액세스하는 특권명령을 실행(그림 4.7❶)하려고 하면 유저상태에서의 실행이므로 그림 4.7과 같이 특권위반이 되어 예외(트랩, ❷)가 발생한다. 그러고 나서 VMM 내의 예외처리부로 명령실행이 옮겨지는(❸) 방법으로 게스트 OS의 하드웨어 자원 액세스를 검출한다.

가상화 지원을 부르짖기 이전의 프로세서도 특권과 유저상태를 구별할 수 있었으나 일부 하드웨어 자원으로의 액세스는 유저상태에서도 가능했었기 때문에, 이 메커니즘에서는 모든 하드웨어 자원으로 게스트 OS가 액세스하는 것을 검출할 수 없

는 문제가 있었다. 사실 이 시대부터 VMware와 같은 상용 VMM이 존재했고 가상화가 수행되고 있었다. VMware에서는 게스트 OS의 바이너리 코드를 실행 전에 체크해서 이와 같은 경우가 있으면 VMM에 통지되도록, 게스트 OS의 바이너리 코드를 갱신하는 방법으로 이 문제에 대처했었다.

그러나 가상화 지원이라고 내세운 프로세서에서는 이와 같은 명령도 유저상태로는 실행할 수 없게 돼 검출 누락이 없어져 특권위반 예외로 게스트 OS의 하드웨어 자원조작을 검출, 가로챌 수 있게 되어 있다.

● **그림 4.7** 게스트 OS의 특권위반 예외로 VMM에 제어를 이행한다



하드웨어 상태의 회피, 복원 — 가상머신의 전환

가상화를 수행할 경우 실제 프로세서의 수가 하나에 그치지 않고 멀티코어나 멀티소켓[주3]인 서버의 경우는 실제 여러 프로세서가 포함된다. 하지만 필요한 가상머신의 수는 실제 프로세서 수보다도 많으므로 실제 프로세서는 시분할로 여러 가상

주3 제5장에서 자세하게 설명하므로 참조하기 바란다.

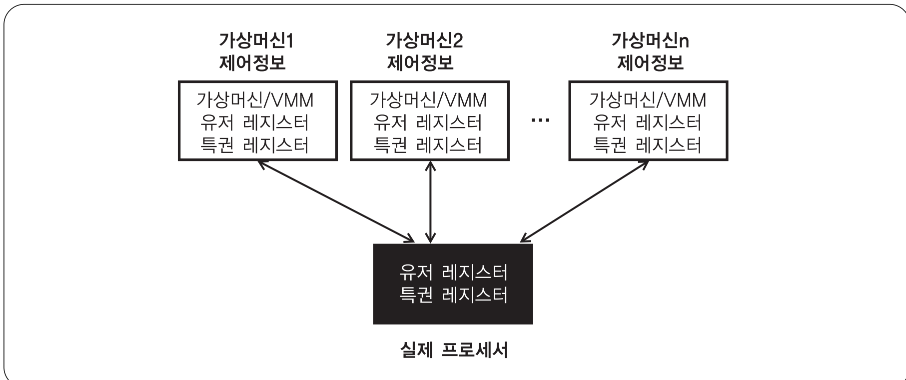
머신을 연기하게 된다. 그리고 VMM은 게스트 OS가 부여된 프로세서 시간을 다 사용하거나 입출력 등 대기시간이 긴 처리를 시작했을 때 다른 가상머신으로 전환하게 된다.

하나의 OS 내의 프로세스 전환의 경우는 유저상태로 조작할 수 있는 범용 레지스터나 부동소수점 레지스터 등을 회피, 복원하면 되지만, 가상화의 경우는 이와 함께 특권 상태에서 조작할 수 있는 각종 레지스터도 회피, 복원할 필요가 있으므로 데이터양이 많아진다. 또한 범용 레지스터의 회피, 복원은 통상의 명령으로 가능하지만, 하드웨어 자원을 조작하는 레지스터는 액세스하기 번거로운 것도 있어서 인텔이나 AMD 프로세서에서는 이와 같은 하드웨어 상태의 회피, 복원을 일괄해서 수행하는 명령[1][2]을 갖춰서 전환 오버헤드를 줄이고 있다.

가상머신 제어정보

그림 4.8에 나타냈듯이 VMM은 가상머신에 부속하는 이러한 정보를 모두 모아서 '가상머신 제어정보'로서 관리하고 가상머신으로의 전환 명령을 실행하면, 하드웨어가 VMM의 상태를 제어정보 내의 VMM 영역으로 회피하고, 가상머신의 이전 상태를 제어정보의 가상머신 영역에서 추출해서 실제 프로세서의 레지스터로 복원해 가상머신 실행을 재개한다.

● **그림 4.8** 각 가상머신의 제어정보를 메모리에 저장해서 실제 프로세서의 레지스터와 교체한다

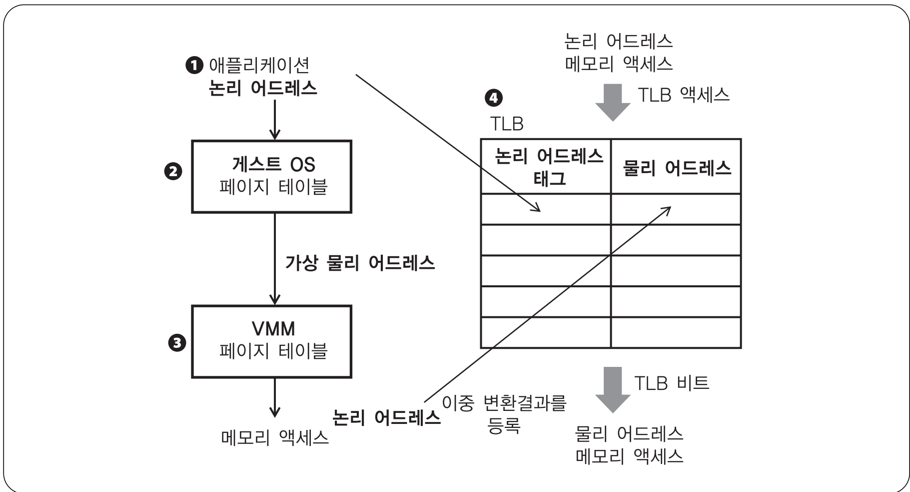


이때 복원할 레지스터 상태가 파괴되어 있으면 터무니없는 상황이 되므로 자동적으로 값의 정당성 체크가 수행된다. 또한 가상머신에서 특권위반 등의 예외가 발생하면, 가상머신의 상태를 제어정보 내의 가상머신 영역으로 회피해서 VMM 영역으로부터 이전 VMM의 상태를 실제 프로세서로 복원하는 형태로 해서 가상머신과 VMM과의 전환을 수행하고 있다.

이중 어드레스 변환, TLB

그림 4.9를 보기 바란다. 각 애플리케이션 프로그램에서의 논리 어드레스(그림 4.9 ❶) 메모리 액세스는 OS가 페이지 테이블을 사용해서 물리 어드레스로 변환한다. 그러나 가상화를 수행할 경우에 이는 물리 어드레스가 아닌 VMM이 제공한 가상 물리메모리 공간의 어드레스(❷)가 된다. 그리고 VMM이 페이지 테이블을 사용해서 가상 물리 어드레스에서 물리 어드레스(❸)로 변환을 수행해 실제 메모리를 액세스하게 된다.

● **그림 4.9** TLB를 사용하면 매번 어드레스 변환을 이중으로 할 필요가 없다



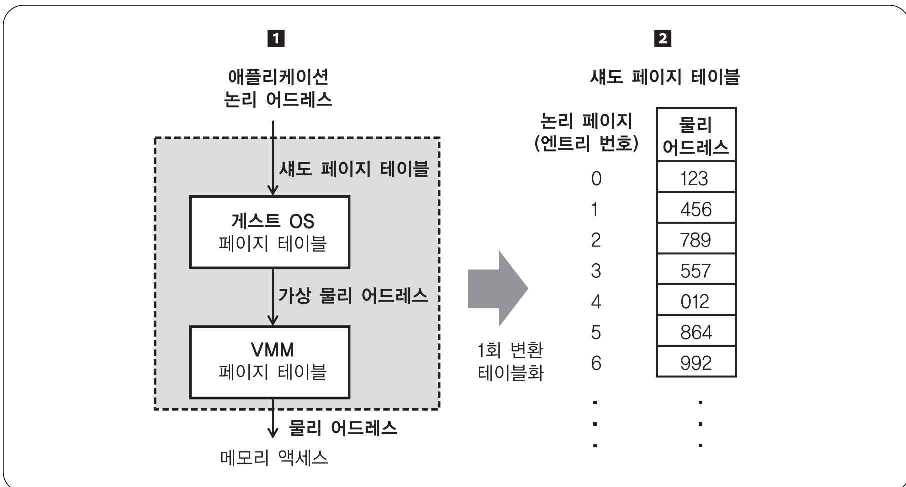
논리적으로는 이와 같이 이중 어드레스 변환이 필요하지만 메모리 액세스할 때마다 이중으로 페이지 테이블을 찾을 필요는 없다. 그림 4.9❹와 같이 애플리케이션 프로그램이 액세스하려고 하는 논리 어드레스와 이중으로 어드레스 변환을 수행한 결과인 물리 어드레스 대응관계를 TBL에 저장해두면, 그 이후의 메모리 액세스는 TLB에서 히트하고 있는 한 페이지 테이블을 찾을 필요가 없으며, 가상화가 아닌 경우와 동일한 성능으로 동작한다. 그리고 TLB 미스가 발생하면 또한 이중 어드레스 변환을 수행해서 TLB에 변환결과를 저장하게 된다.

섀도 페이지 테이블 — x86의 하드웨어 테이블 walk 구조

이렇게 이중 변환결과를 TLB에 저장하게 되지만 TLB로의 저장을 소프트웨어에서 수행하고 있는 프로세서의 경우는 간단히 할 수 있다.

그러나 x86 프로세서에서는 하드웨어가 자동적으로 페이지 테이블을 읽어서 변환결과를 TLB에 기록해주는 형태의 하드웨어 테이블 walk 기구(Hardware table walk, Hardware page table walk)를 가지고 있다.

● **그림 4.10** 하드웨어 테이블 walk를 수행할 경우는 섀도 페이지 테이블이 필요※



※ 그림 4.10 내의 작업은 모두 VMM에 의해 수행된다.

가상화를 수행하지 않는 경우에 이것은 편리한 기능이지만 가상화를 할 경우에는 쓸데없는 참견으로, VMM으로서는 그림 4.10의 왼쪽❶에 나타낸 이중의 변환결과를 모아서 그림 4.10의 우측❷와 같이 논리 어드레스에서 실제 물리 어드레스로 한 번에 변환할 수 있는 페이지 테이블을 작성할 필요가 생긴다. 이 페이지 테이블을 **섀도 페이지 테이블**(Shadow page table)이라고 한다. 그리고 게스트 OS가 실행하는 프로그램을 전환해서 페이지 테이블이 전환할 때마다 섀도 테이블을 갱신할 필요가 생기므로 가상화에서 큰 오버헤드가 되고 있다.

■ x86 프로세서의 어드레스 변환 구조의 예

x86 프로세서의 어드레스 변환기구(그림 3.78)를 예로 들면, 프로그램 전환할 때 게스트 OS가 페이지 테이블의 첫 어드레스를 가리키는 CR3 레지스터의 내용을 고쳐쓰려고 하면, 특권위반 예외가 발생해서 VMM으로 제어가 옮겨간다. VMM은 예외원인을 분석해서 CR3 레지스터로의 쓰기에서 문제가 있었음이 판명되면, 섀도 페이지 테이블을 만들어서 테이블의 첫 어드레스를 페이지 테이블의 첫 어드레스로 해서 CR3 레지스터에 쓴다.

또한 게스트 OS가 페이지 테이블의 엔트리를 고쳐쓸 경우도 그에 대응해서 섀도 페이지 테이블을 수정할 필요가 있다. 이 경우에 대응하기 위해 VMM은 게스트 OS의 페이지 테이블을 저장할 메모리 영역을 쓰기금지 속성으로 해둔다. 그렇게 하면 게스트 OS가 페이지 테이블의 엔트리를 고치려고 하면 액세스위반 예외가 발생해서 VMM에 통지된다. 그러면 VMM은 일시적으로 쓰기 가능으로 해서 OS가 수행하려고 했던 게스트 OS의 페이지 테이블 고쳐쓰기를 수행하고, 나아가 갱신내용에 대응하는 섀도 페이지 테이블을 수정해서 게스트 OS로 복귀한다.

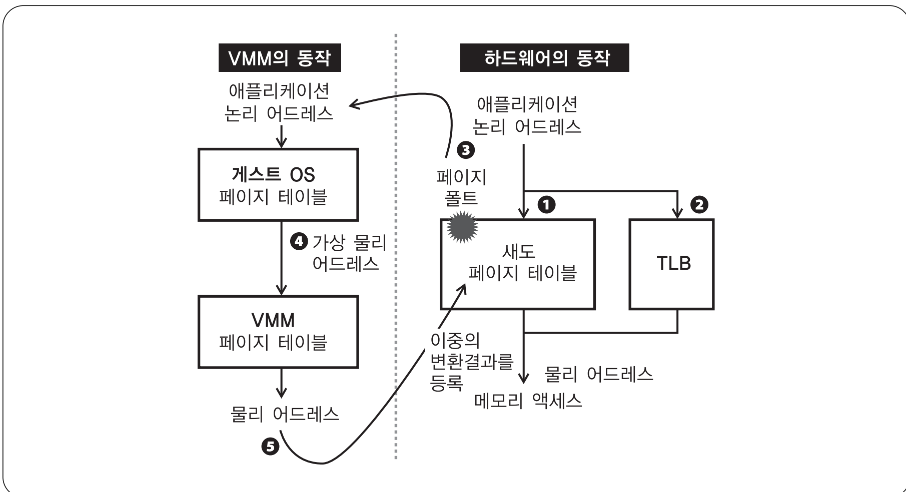
가상 TLB 방식 — 좀더 능숙하게 섀도 테이블을 만드는 법

위에서 말한 섀도 페이지 테이블 작성 과정보다 좀더 좋은 방법이 있는데, 바로 **가상 TLB 방식**이라는 방법이다. 가상 TLB 방식에서 페이지 테이블 영역은 쓰기금지가 아니고, 게스트 OS는 VMM의 개입 없이 자신의 페이지 테이블을 자유롭게 변경할 수가 있다. 그러나 하드웨어가 어드레스 변환에 사용하는 섀도 페이지 테이블은 VMM이 관리한다.

그림 4.11에 나타낸 가상 TLB 방식에서는 처음에 VMM이 관리하는 섀도 페이지 테이블(❶)이나 TLB(❷)는 비워둔다. 이렇게 하면 애플리케이션이 메모리를 액세스 할 때 해당 논리 어드레스에 대응하는 엔트리는 TLB와 섀도 페이지 테이블에 모두 존재하지 않으므로 페이지 폴트 예외(❸)가 발생해서 VMM으로 제어가 옮겨간다. 이 시점에서 VMM은 게스트 OS의 페이지 테이블을 보고 대응하는 가상 물리 어드레스(❹)를 알고 이를 자신의 페이지 테이블을 이용해서 물리 어드레스(❺)로 변환한다. 그러고 나서 해당 논리 어드레스와 이중 어드레스 변환결과인 물리 어드레스 간 대응을 하드웨어가 사용하는 섀도 페이지 테이블(❶)에 써서 액세스된 페이지에 대응하는 엔트리를 작성한다.

이와 같이 하면 하드웨어가 사용하는 섀도 페이지 테이블에는 애플리케이션이 사용한 페이지의 어드레스 변환결과가 쌓여가게 된다. 이것은 TLB와 같은 형태의 동작이므로 이 방식을 가상 TLB 방식이라고 한다. 이 방식에서는 새로운 페이지가 액세스된 시점에서 가상 TLB(=섀도 페이지 테이블)에 등록해가면 되므로 섀도 페이지 테이블을 만드는 데 필요한 노력이 상당히 감소한다.

● **그림 4.11** 가상 TLB 방식의 섀도 페이지 테이블 작성



■ 게스트 OS 페이지 테이블 변경을 가상 TLB에 반영

가상 TLB 방식에서 게스트 OS는 페이지 테이블의 내용을 마음껏 변경할 수 있으므로, 예를 들어 게스트 OS가 특정 페이지에 대응하는 가상 물리 어드레스를 고쳐 썼다고 해도 이는 VMM에 통지되지 않으므로 가상 TLB의 내용과의 사이에 모순이 나타나게 된다.

그러나 가상화를 수행하지 않을 경우라도 OS가 이와 같은 페이지 테이블 변경을 수행한 경우는 하드웨어의 TLB 내용과 변경된 페이지 테이블의 내용에 모순이 나오므로 TLB의 내용을 클리어(clear)할 필요가 생긴다. 가상화 환경에서는 이 TLB를 클리어(clear)하는 특권명령을 실행할 때 예외가 발생해서 VMM으로 제어를 이행함으로 인해 게스트 OS의 페이지 테이블 변경을 가상 TLB에 반영하는 동작을 실행할 수가 있다.

EPT, NPT — 이중 어드레스 변환을 자동화하는 하드웨어 기구

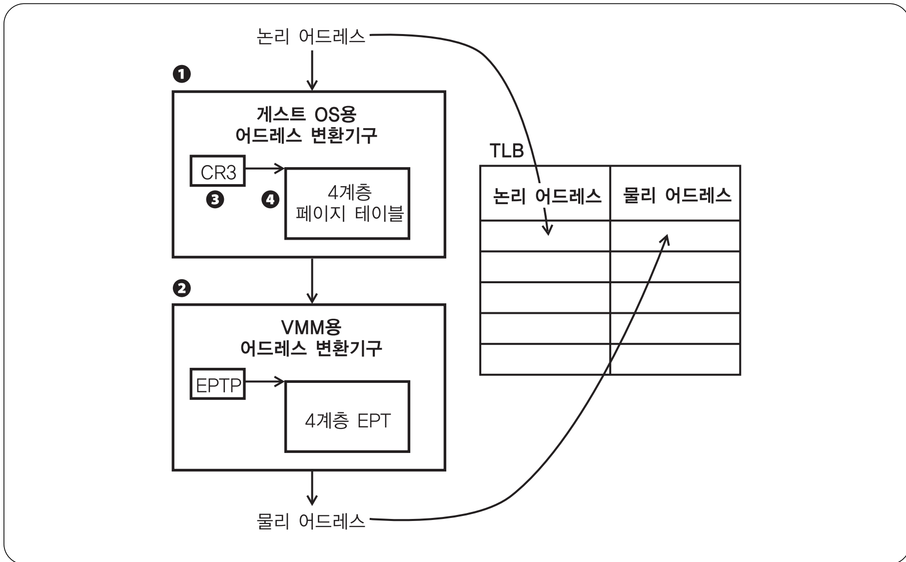
가상 TLB 방식을 사용하면 수고는 덜 수 있으나 섀도 테이블 관리는 가상화를 수행하는 데 있어서 커다란 오버헤드가 되며, 이를 해결하기 위해 가상화를 지원하는 AMD나 인텔 프로세서는 이중 어드레스 변환에 따른 오버헤드를 줄이는 기구를 갖추게 되었는데, 그것이 바로 AMD의 NPT(Nested Page Table)[3], 혹은 인텔의 EPT (Extended Page Table)라는 기구다.

그림 4.12에 나타냈듯이 이러한 기구에는 하드웨어가 이중 페이지 테이블 walk를 자동적으로 해주며 그 결과를 TLB에 저장한다.

먼저 애플리케이션 프로그램의 메모리 액세스가 TLB를 미스하면, 그림 4.12❶에서 게스트 OS의 페이지 테이블을 사용해 논리 어드레스를 가상 물리 어드레스로 변환한다. 그러고 나서 이 가상 물리 어드레스를 인텔 프로세서의 경우는 ❷VMM의 페이지 테이블 EPT를 사용해서 메모리를 액세스하는 물리 어드레스로 변환한다. 그러나 이 프로세스를 자세히 보면 CR3 레지스터(❸)가 4계층 페이지 테이블(❹)의 첫 부분을 가리키고 있는데, 이는 가상 물리공간의 어드레스로 쓰여 있으므로 먼저 CR3 레지스터와 4계층의 테이블 각각에서 EPT를 사용한 변환이 필요해진다. 또한 EPT 자체가 4계층 테이블 구조로 되어 있으므로 최종 물리 어드레스로의 액세스를

포함하면 5회의 메모리 액세스가 필요, 이들의 곱으로 어드레스를 변환하면 전체적으로는 25회의 메모리 액세스가 필요해진다.

● **그림 4.12** 이중 테이블 walk를 자동적으로 수행하는 EPT 기구(인텔)※



※ 그림 내의 조작은 모두 하드웨어가 실행한다. 'EPTP'는 EPT의 첫 어드레스를 저장하는 레지스터

그러므로 하드웨어 동작으로는 꽤 복잡하지만 NPT나 EPT를 갖춘 프로세서에서는 VMM이 이중 어드레스 변환을 위해 노력을 들일 필요가 없어지므로 가상화의 오버헤드가 크게 감소하게 된다.

I/O 가상화

원리적으로는 I/O 가상화도 프로세서의 가상화와 마찬가지로, 제어 레지스터로의 액세스를 캐치해서 게스트 OS에서 보면 실제 I/O를 제어하고 있는 것처럼 동작하면 되지만 여러 가지 사정으로 그렇게 하지 못하는 경우가 있다. 크게 나눠서 3종류의 I/O 가상화 방식이 사용되고 있으며, 차례로 살펴보도록 하자.

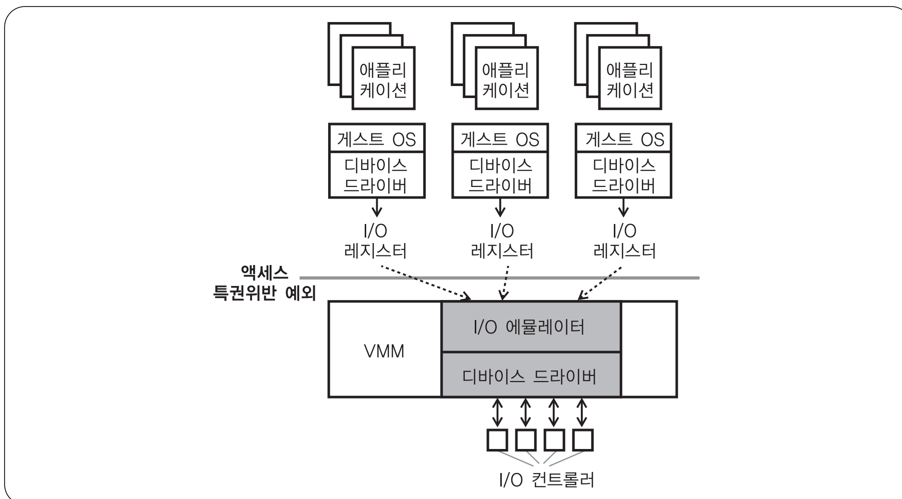
정통적인 I/O 가상화 — VMM이 I/O를 에뮬레이션

첫 번째 방식은 그림 4.13과 같이 각 게스트 OS는 통상적인 OS와 마찬가지로 입출력장치를 구동하는 디바이스 드라이버를 가지며, I/O 컨트롤러의 레지스터 등을 조작하고, 해당 하드웨어 액세스를 특권위반 예외로 검출해서 VMM이 모순이 없는 동작을 수행하도록 하는 정통적인 가상화 방식이다.

기본적으로 메모리의 가상화와 동일하지만 메모리의 경우는 쓰여진 데이터를 기억해서 해당 값을 읽어낼 뿐이지만, 입출력장치의 경우는 과거에 어떤 커맨드가 전송되었고 내부상태가 어떠했는지에 따라 동작이 달라지므로 이 방식에서 VMM은 입출력장치의 동작을 흉내 내는 에뮬레이션을 수행할 필요가 있다.

그러나 디바이스 드라이버를 속일수록 정확한 에뮬레이터를 만들기는 용이하지 않다. 또한 VMM측에는 실제 입출력장치를 구동하는 디바이스 드라이버가 필요해진다. 윈도우나 리눅스에서는 많은 종류의 디바이스 드라이버가 개발되어 있지만, VMM측에 이러한 에뮬레이터와 디바이스 드라이버를 갖추기란 어렵다. 그러므로 이 VMM에 의한 에뮬레이터 방식에서는 지원할 수 있는 입출력장치가 한정되어 있다. 또한 하드웨어 조작이나 데이터 전송에 일일이 VMM이 관여하므로 성능적으로도 한계가 있다.

● **그림 4.13** 게스트 OS의 하드웨어 액세스를 캐치하는 I/O 가상화



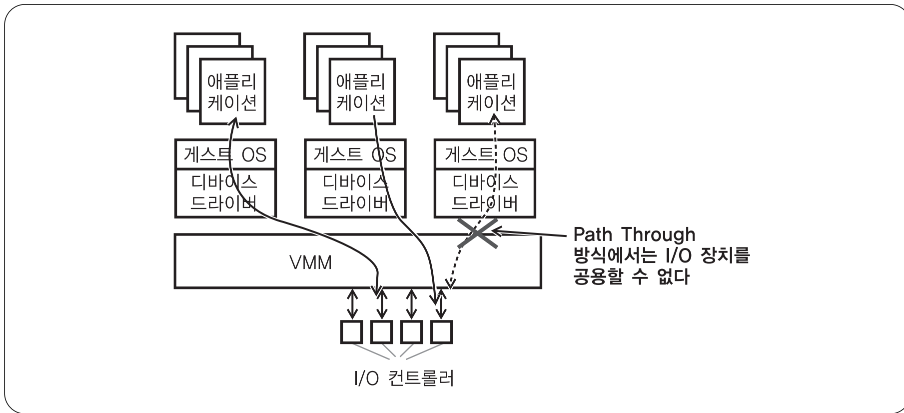
Path Through 방식 — 입출력장치를 하나의 가상머신에 할당

에뮬레이터 방식에서는 디스크나 LAN 등 고속 디바이스의 성능을 발휘할 수 없으므로 이 경우에는 고속 I/O 디바이스를 직접 하나의 가상머신에 할당해 버리는 Path Through라는 두 번째 방식이 이용되고 있다. Path Through 방식은 그림 4.14에 나타냈듯이 게스트 OS의 디바이스 드라이버가 직접 입출력장치를 구동한다. Path Through 방식은 VMM의 I/O 에뮬레이터나 디바이스 드라이버가 관여하지 않으므로 오버헤드는 적고 고속의 디바이스에서도 지원할 수 있다는 것이 특징이다.

그러나 이 방식에서는 각각의 입출력장치가 가상머신에 묶이게 되므로 공용할 수 없어서 많은 입출력장치가 필요해진다는 점이 결점이다. 또한 게스트 OS의 디바이스 드라이버는 가상 물리공간에 놓인 I/O 제어 레지스터를 액세스해서 DMA 장치에 대해서도 가상 물리 어드레스로 전송해야 할 곳의 어드레스를 지정해 버린다. 따라서 어딘가에서 가상 물리 어드레스를 실제 물리 어드레스로 변환해줄 필요가 있다.

이에 대해서 인텔의 I/O 가상화[4]를 지원하는 I/O 허브칩과 컨트롤러 칩에서는 칩 내에 어드레스 변환기구나 IOTLB(I/O용 어드레스 변환결과를 캐시하는 TLB)를 가지고, 입출력장치측은 디바이스 드라이버가 지정한 가상 물리 어드레스를 물리 어드레스로 변환해서 DMA 전송을 수행하는 기능을 가지고 있다. 단순한 Path Through 방식에서는 게스트 OS의 디바이스 드라이버에 버그가 있어서 잘못된 어드레스로 DMA해 버리면 다른 게스트 OS의 영역을 파괴할 우려가 있지만, 이와 같은 어드레스 변환기구를 사용하여 VMM이 작성하는 어드레스 변환 테이블이 올바르다면 이와 같은 문제를 피할 수 있다. 또한 이와 같은 하드웨어 기구와 아울러서 PCI 버스의 I/O 컨트롤러 부분에서 입출력장치의 가상화를 수행하는 I/O Virtualization의 규격화[6]가 진행되고 있으며, 이 규격에 준거하는 디바이스 컨트롤러나 LAN 어댑터 등에서는 Path Through 방식으로도 실제 입출력장치를 공용할 수 있게 되어 있다.

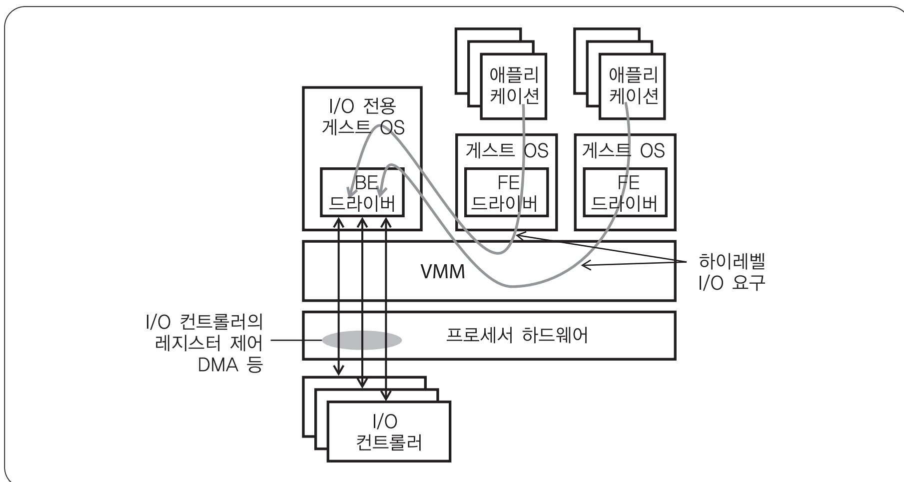
● **그림 4.14** 입출력장치를 가상머신에 할당하는 Path Through 방식



Para Virtualization(준가상화) 방식 — 하이레벨 I/O 요구 사용

세 번째 방식은 가상화의 기본개념에서 약간 벗어난 것인데, 게스트 OS를 조금 변경해서 I/O 컨트롤러의 레지스터 조작 레벨이 아닌 디스크의 데이터 블록 R/W나 LAN의 패킷 송수신과 같이 보다 추상도가 높은 I/O 요구로 변경하는 방식이다(그림 4.15).

● **그림 4.15** 하이레벨 I/O 요구를 사용하는 Para Virtualization 방식



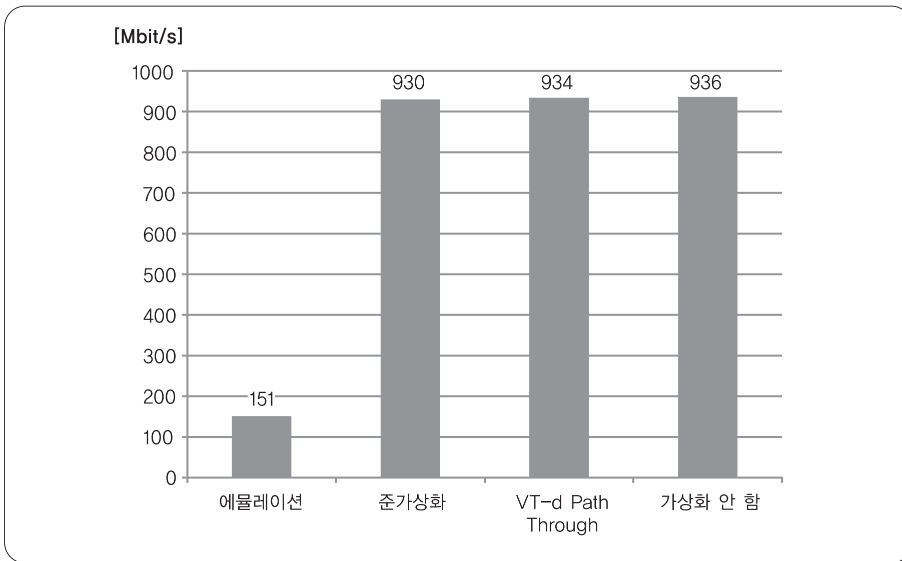
이 하이레벨 I/O 요구를 다루는 게스트 OS측의 모듈은 프론트엔드 드라이버(FE 드라이버)라고 한다. 그리고 VMM측에서는 I/O 요구를 처리하는 전용 가상머신을 만들고, 각 가상머신으로부터의 I/O 요구를 I/O 전용 가상머신에 보낸다. 그리고 이 I/O 전용 게스트 OS는 백엔드 드라이버(BE 드라이버)를 가지며, 하이레벨 요구를 I/O 컨트롤러의 레지스터 조작 레벨로 변환해서 입출력장치를 구동한다. 이렇게 하면 입출력장치를 에뮬레이션할 필요는 없고 하이레벨 I/O 요구를 처리하는 기능이 있으면 된다. 또한 이 백엔드 드라이버는 게스트 OS의 디바이스 드라이버를 유용하는 것도 가능하므로 실용성이 높은 방식이다.

이 방식은 게스트 OS에서 VMM에 대해 하이퍼바이저 호출[주4]을 사용해서 I/O 요구를 내므로, VMM의 존재를 게스트 OS에 드러내 보이지 않으므로 완전한 가상화가 아닌 Para Virtualization(준가상화)라고 한다. Para Virtualization을 사용하는 VMM으로는 Xen[5]이 유명하지만, 최근에는 대부분의 VMM이 복수의 I/O 가상화 방식을 지원하고 있어서 Para Virtualization을 지원하는 VMM도 일반적이 되었다.

이렇게 각종 가상화를 적용한 경우의 성능에 대한 예로서 그림 4.16에 인텔이 발표한 데이터[7]를 게재했다. 가상화된 입출력장치는 1Gbit 이더넷 어댑터로, 가장 오른쪽에 가상화를 하지 않고 직접 사용한 경우는 936Mbit/s의 통신 성능인 것에 비해 에뮬레이션 방식에서는 151Mbit/s로 1/6 정도의 성능으로 떨어지고 있다. 이에 비해 Para Virtualization의 경우는 930Mbit/s, I/O 어드레스 변환 등의 하드웨어 지원을 이용한 Path Through 방식에서는 934Mbit/s로 가상화를 하지 않은 경우와 비교해서 거의 손색없는 성능을 실현하고 있다.

주4 OS에 작업을 의뢰하는 인터페이스는 슈퍼바이저 호출(Supervisor call)이라고 한다. 이에 비해 하이퍼바이저(VMM)에 작업을 요구하는 인터페이스를 하이퍼바이저 호출(Hypervisor call)이라고 한다.

● **그림 4.16** 각종 가상화에 따른 1Gbit 이더넷 어댑터의 실효 전송속도※



※ 인텔의 IDF Spring 2008에서의 발표자료를 기반으로 필자 작성

라이브 마이그레이션

앞서 말한 하드웨어 기구로 인해 가상화의 오버헤드가 줄어들고 가상화 이용이 확대되어 오고 있다. 이제까지 설명해온 것은 하나의 프로세서를 여러 가상머신으로 보이게 하는 기술이지만, 이 기술을 응용함에 따라 새로운 사용법이 가능해지게 되었다.

앞서 그림 4.8과 같이 VMM은 가상머신의 상태를 제어정보로 해서 회피하고 있다. 이 가상머신의 제어정보를 다른 프로세서에 복원함으로써 이동해갈 프로세서에서 원래 프로세서와 동일한 가상머신 상태를 재현할 수가 있다.

공통 메모리 시스템에서 입출력장치도 같은 것을 사용할 수 있는 경우는 프로세서의 상태를 인계하면 그대로 프로그램 처리를 물려받을 수 있다. 공통 메모리가 아니라 다른 서버로 이동할 경우는 가상 프로세서의 상태를 이동하면서 동시에 원래의 가상 프로세서가 사용하고 있던 명령이나 데이터 메모리를 이동해갈 프로세서의

메모리로 카피해서, 애플리케이션 프로그램에서 볼 때 동일한 어드레스가 되도록 페이지 테이블을 구축할 필요가 있다. 그리고 나서 애플리케이션이 사용하고 있던 파일이나 LAN 등으로의 액세스도 넘겨줄 필요가 있다.[주5]

이와 같이 해서 1대의 서버에서 동작 중인 애플리케이션을 다른 서버로 이동해서 동작을 계속하도록 하는 것을 '라이브 마이그레이션'(Live Migration)이라고 한다. 라이브 마이그레이션이 가능하다면 부하가 높은 서버에서 부하가 낮은 서버로 처리를 옮겨서 부하를 평준화하거나, IDC 전체의 부하가 낮은 경우에는 소수의 서버로 처리를 집중시켜 남은 서버의 전원을 오프해서 에너지 사용을 줄이는 것과 같은 사용법이 가능하다.

주5 구체적으로는 파일의 액세스 버스를 새로 바꾸거나 IP 어드레스나 MAC 어드레스 인계 등을 가리킨다.

4.4 정리

이번 장에서는 최근 주목받고 있는 '가상화'에 초점을 맞춰서 '하나의 프로세서를 여러 프로세서인 것처럼 보이게 하는' 가상화가 왜 필요한지, 장점은 무엇인지에 대해 설명했다. 가상화를 수행함으로써 서버 가동률을 높여 적은 대수의 서버로 처리할 수 있게 되고 비용, 설치면적, 전력 등을 절약할 수 있으므로 최근 가상화가 주목받고 있으며, 클라우드 데이터 센터 등에서는 필수 기술이 되고 있음을 이해할 수 있었을 것이다.

그리고 프로세서가 어떤 메커니즘으로 가상화를 실현하고 있는지에 대해서 설명했다. VMM에 의한 가상화는 게스트 OS의 하드웨어 조작을 검출하는 '특권위반 예외기술'을 기반으로 해서 만들어졌다.

가상화를 수행할 때 포인트는 '메모리 가상화'와 'I/O 가상화'다. 메모리 가상화에는 이중 어드레스 변환이 필수적이며 섀도 테이블 작성이 큰 부담이 되었지만, 이중 어드레스 변환을 자동화하는 NPT나 EPT라고 하는 기구가 내장됨으로써 최신 프로세서에서는 VMM의 부담이 줄어들었다. 또한 I/O에 대해서도 어드레스 변환 기구가 갖춰져서 가상화에 따르는 VMM의 오버헤드가 줄어들고 있다.

이번 장에서는 전체적으로 하드웨어 가상화가 지향하는 견고하고 안전한 분리와 가상화 오버헤드 줄이기가 어떻게 해서 실현되고 있는지에 대해 이해할 수 있었을 것이다.

제4장: 참고문헌

[1] 제3장 참고문헌 [8] 참조

[2] 'AMD64 Architecture Programmer's Manual'(Volume 2: System Programming)
URL http://support.amd.com/us/Embedded_TechDocs/24593.pdf

[3] 'AMD-V Nested Paging'(2008)
URL http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf

[4] D.Abramson 외 공저 'Intel Virtualization Technology for Directed I/O : Architecture Specification'(Intel Technology Journal, Volume 10, Issue 3, pp.179-192, 2006)

[5] P.Burham 외 공저 'Xen and the Art of Virtualization'
(SOSP'03, pp.164-177, 2003)

[6] 'PCI-SIG SR-IOV Primer : AN INTRODUCTION TO SR-IOV TECHNOLOGY'
URL http://download.intel.com/design/network/applnots/321211.pdf

[7] B.Nan 'Intel Virtualization Technology for Directed I/O(Intel Vt-d)'
(IDF 2008 Spring)

COLUMN

가상화의 시작

이번 장 머리글에서 가상화의 역사는 40년 이상이라고 언급했었다. 제2장에서 하나의 컴퓨터를 시분할로 다수의 유저가 사용할 수 있게 하는 TSS를 개발하는 Project MAC에 대해 설명했는데, IBM은 기업의 사무처리 등 큰 처리를 자동적으로 차례로 실행시켜 가는 배치 시스템에 중점을 뒀었다. 따라서 IBM이 전력을 기울여 개발한 System 360(1957년)은 TSS를 지원하는 하드웨어 기능을 갖추고 있지 않았다. System 360은 사무처리에서는 대성공을 거두었지만 TSS를 선호하는 과학기술분야 유저에게는 판매할 수가 없었다.

그래서 IBM의 Cambridge Scientific Center[주A]는 System 360을 다수의 가상머신으로 분할하는 CP-40이라는 제어 프로그램과 그 위에서 동작하는 싱글유저의 소규모 OS인 Cambridge Monitor System(CMS)를 개발했다. 이 CP-40은 실험 시스템이었지만, 가상 메모리를 지원하는 System 360 Model 67용인 CP-67과 CMS는 유저에게 제공되었다. 이 CP/CMS는 대형 TSS OS보다 적은 오버헤드로 다수의 유저를 지원할 수 있고 동작도 안정적이었다고 한다.

주A 영국의 캠브리지가 아닌 미국의 매사추세츠주의 캠브리지다.