18

주소 변환의 원리

CPU 가상화 부분에서, **제한적 직접 실행**(**LDE**)이라는 기법을 집중적으로 다루었다. LDE의 아이디어는 간단하다. 대부분의 경우 프로그램은 하드웨어에서 직접 실행된다. 그러나, 프로세스가 시스템 콜을 호출하거나 타이머 인터럽트가 발생할 때 등의 특정 순간에는 운영체제가 개입하여 문제가 발생하지 않도록 한다. 운영체제는 약간의 하드웨어 지원을 받아 *효율적인* 가상화를 제공하기 위해 실행 프로그램에게 방해가 되지 않도록 최선을 다한다. 중요한 순간에 운영체제가 관여하여 하드웨어를 직접 제어한다. 효율성과 제어가 현대 운영체제의 목표이다.

메모리 가상화에서 비슷한 전략을 추구할 것이다. 가상화를 제공하는 동시에 효율성(efficiency)과 제어(control) 모두를 추구한다. 효율성을 높이려면 하드웨어 지원을 활용할 수밖에 없다. 처음에는 몇 개의 레지스터만 사용하는 정도부터 TLB, 페이지 테이블 등으로 점차 복잡한 하드웨어를 사용하게 될 것이다. 제어는 응용 프로그램이 자기자신의 메모리 이외에는 다른 메모리에 접근하지 못한다는 것을 운영체제가 보장하는 것을 의미한다. 프로그램을 다른 프로그램으로부터 보호하고 운영체제를 프로그램으로부터 보호하기 위하여 하드웨어 도움이 필요하다. 마지막으로, *유연성*(flexibility) 측면에서 VM 시스템에서 필요한 사항이 있다. 프로그래머가 원하는 대로 주소 공간을 사용하고, 프로그래밍하기 쉬운 시스템을 만들기 원한다. 핵심 문제는 다음과 같다.

핵심 질문: 어떻게 효율적이고 유연하게 메모리를 가상화하는가

어떻게 효율적인 메모리 가상화를 구축할 수 있을까? 프로그램이 필요로 하는 유연성을 어떻게 제공하는가? 프로그램이 접근할 수 있는 메모리의 위치에 대한 제어를 어떻게 유지하는가? 메모리 접근을 어떻게 적절히 제한할 수 있는가? 어떻게 이 모든 것을 효율적으로 할 수 있는가?

우리가 다룰 기법은 **하드웨어-기반 주소 변환**(**hardware-based address translation**) 또는 그냥 짧게 **주소 변환**(**address translation**)이다. 이 기술은 제한적 직접 실행 방식에 부가적으로 사용되는 기능이라고 생각할 수 있다. 주소 변환을 통해 하드웨어는 명령어 반입, 탑재, 저장 등의 **가상** 주소를 정보가 실제 존재하는 **물리** 주소로

변환한다. 프로그램의 모든 메모리 참조를 실제 메모리 위치로 재지정하기 위하여 하드웨어가 주소를 변환한다.

물론, 하드웨어에 의해 제공되는 저수준(low level) 기능들은 변환을 가속화 시키는 도움을 주지만, 하드웨어만으로 메모리 가상화를 구현할 수는 없다. 정확한 변환이 일어날 수 있도록 하드웨어를 셋업하기 위해 운영체제가 관여해야 한다. 운영체제는 메모리의 빈 공간과 사용 중인 공간을 항상 알고 있어야 하고, 메모리 사용을 제어하고 관리한다.

이 모든 작업의 목표는 다음과 같다. 프로그램이 자신의 전용 메모리를 소유하고 그 안에 자신의 코드와 데이터가 있다는 **환상**을 만드는 것이다. 이 가상 현실의 이면에는 불편한 진실이 존재한다. CPU가 실행 중인 한 프로그램에서 다음 프로그램으로 전환하는 것처럼, 많은 프로그램이 메모리를 공유한다. 운영체제는 하드웨어의 도움을 받아 쓸모없는 기계를 유용하고, 강력하며 사용하기 쉬운 개념으로 변환시킨다.

18.1 가정

메모리 가상화를 위한 첫 번째 시도는 매우 간단하다, 거의 터무니없을 정도다. 오케이, 실컷 웃어라. 잠시 후에 TLB, 멀티 레벨 페이지 테이블 및 기타 기법을 이해해야 할 때에는 운영체제가 당신을 비웃을 것이다. 운영체제가 비웃는다는 것이 별로 기분이 좋지 않겠지? 글쎄, 운이 없게도 그게 바로 운영체제의 구동방식이기 때문에 어쩔 수 없다.

당분간 사용자 주소 공간은 물리 메모리에 연속적으로 배치되어야 한다고 가정한다. 논의를 단순화하기 위해 주소 공간의 크기가 너무 크지 않다고 가정한다. 주소 공간은 *물리 메모리 크기보다 작다*. 우리는 또한 각 주소 공간의 *크기는 같다고* 가정한다. 이러한 가정이 비현실적이라고 걱정하지 말기 바란다. 논의를 진행하면서 가정들을 완화하여 실제적인 메모리의 가상화를 이끌어 낼 것이다.

18.2 사례

주소 변환 구현을 위해 어떤 것이 필요한지, 왜 그런 기법이 필요한지 알기 해 간단한 예를 살펴보자. 주소 공간이 그림 18.1과 같은 프로세스가 있다고 가정하자. 여기서 우리가 검토하려고 하는 코드는 메모리에서 값을 탑재하고, 3을 증가시키고, 다시 메모리에 저장하는 짧은 코드이다. 이 코드의 C 언어 표현은 다음과 같다고 예상할 수 있다.

```
void func() {
     int x = 3000;
     x = x + 3; // 우리가 관심있는 코드
```

컴파일러는 이 코드를 어셈블리 코드로 변환하고 그 결과는 x86 어셈블리로 다음과 같을 것이다. Linux 상에서는 `objdump` 도구 또는 Mac OS X 상에서는 `otool`을 사용하여 디스어셈블한다.

```
128: movl 0x0(\%ebx), \%eax ; 0+ebx를 eax에 저장
```

팁: 개입은 강력하다

개입은 컴퓨터 시스템에 큰 효과를 주기 위해 종종 사용되는 일반적이고 강력한 기술이다. 메모리를 가상화할 때 하드웨어는 모든 메모리 접근에 개입하여 프로세스에 의해 발생하는 가상 주소를 원하는 정보가 실제로 저장되어 있는 물리 주소로 변환할 것이다. 개입이라는 일반적인 기술은 훨씬 더 광범위하게 적용 가능하다. 사실 거의 모든 잘 정의된 인터페이스는 새 기능을 추가하기 위해 또는 시스템의 다른 측면을 개선하기 위해 개입을 사용할 수 있다. 이러한 접근 방식의 장점 중 하나는 **투명성**이다. 개입은 종종 인터페이스 사용자인 클라이언트를 변경하지 않고 이루어지고 따라서 클라이언트는 수정할 필요가 없다.

```
132: addl \$0x03, \%eax     ; eax레지스터에 3을 더한다
135: movl \%eax, 0x0(\%ebx) ; eax를 메모리에 다시 저장
```

이 코드는 비교적 이해하기 쉽다. **x**의 주소는 레지스터 **ebx**에 저장되어 있다고 가정하고 이 주소에 저장되어 있는 값을 `movl` 명령어(`long word`를 이동하는 명령어)를 사용하여 범용 레지스터 **eax**에 넣는다. 다음 명령은 **eax**에 3을 더하고, 마지막 명령은 **eax**의 값을 같은 위치의 메모리에 저장한다.

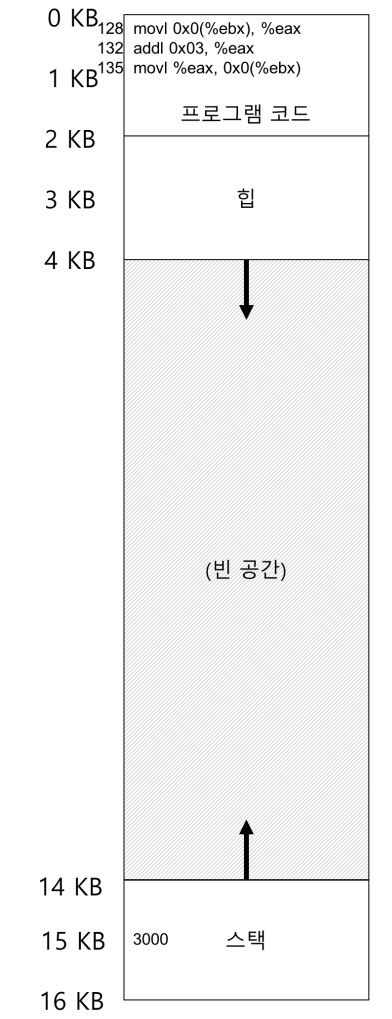
그림 18.1에서 코드와 데이터가 프로세스 주소 공간에 어떻게 배치되어 있는지 볼 수 있다. 세 개 명령어의 코드는 주소 128에 위치하고(상단 코드 섹션에), 변수 **x**의 값은 주소 15 KB(아래 쪽 스택에)에 위치한다. 그림에서 **x**의 초기 값은 3000이다(스택 영역 참조).

이 명령어가 실행되면 프로세스의 관점에서 다음과 같은 메모리 접근이 일어난다.

- 주소 128의 명령어를 반입
- 이 명령어 실행(주소 15 KB에서 탑재)
- 주소 132의 명령어를 반입
- 이 명령어 실행(메모리 참조 없음)
- 주소 135의 명령어를 반입
- 이 명령어 실행(15 KB에 저장)

프로그램 관점에서 **주소 공간**은 주소 0부터 시작하여 최대 16 KB까지이다. 프로그램이 생성하는 모든 메모리 참조는 이 범위 내에 있어야 한다. 메모리 가상화를 위해 운영체제는 프로세스를 물리 메모리 주소 0이 아닌 다른 곳에 위치시키고 싶다. 어떻게 하면 프로세스 모르게 메모리를 다른 위치에 **재배치** 하느냐가 우리가 해결해야 할 문제이다. 프로세스 주소 공간이 실제로는 다른 물리 주소에 배치되어 있을 때, 주소 0번지부터 시작하는 가상 주소 공간의 환상을 어떻게 제공할 수 있을까?

이 프로세스의 주소 공간이 메모리에 배치되었을 때 가능한 물리 메모리 배치의 한 예가 그림 18.2에 나와 있다. 그림에서 물리 메모리의 첫 번째 슬롯은 운영체제 자신이



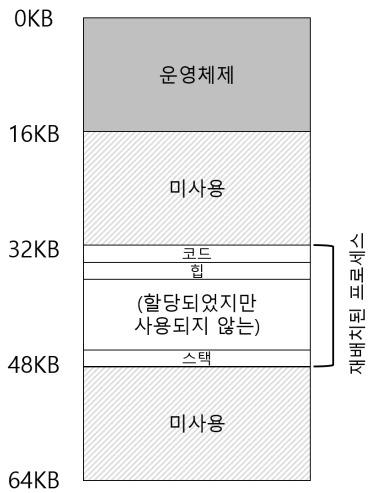
〈그림 18.1〉 **프로세스와 그 주소 공간**

사용하는 것을 볼 수 있다. 위 예의 프로세스는 물리 주소 32 KB에서 시작하는 슬롯에 재배치된 것을 볼 수 있다. 다른 두 슬롯(16 KB-32 KB와 48 KB-64 KB)은 비어 있다.

18.3 동적(하드웨어-기반) 재배치

하드웨어 기반 주소 변환을 이해하기 위하여 먼저 첫 번째 실현 사례를 설명한다. 1950년대 후반의 첫 번째 시분할 컴퓨터에서 **베이스와 바운드**(**base and bound**)라는 간단한 아이디어가 채택되었다. 이 기술은 또한 **동적 재배치**(**dynamic relocation**)라고 한다. 우리는 두 용어를 혼용한다 [SS74].

각 CPU마다 2개의 하드웨어 레지스터가 필요하다. 하나는 **베이스**(**base**) 레지스터라고 불리고, 다른 하나는 **바운드**(**bound**) 레지스터 혹은 **한계**(**limit**) 레지스터라고



〈그림 18.2〉 **하나의 재배치된 프로세스를 가진 물리 메모리**

불린다. 이 베이스와 바운드 쌍은 우리가 원하는 위치에 주소 공간을 배치할 수 있게 한다. 배치와 동시에 프로세스가 오직 자신의 주소 공간에만 접근한다는 것을 보장한다.

이 설정에서 각 프로그램은 주소 0에 탑재되는 것처럼 작성되고 컴파일된다. 프로그램 시작 시, 운영체제가 프로그램이 탑재될 물리 메모리 위치를 결정하고 베이스 레지스터를 그 주소로 지정한다. 위의 예에서 운영체제는 프로세스를 물리 주소 32 KB에 저장하기로 결정하고 베이스 레지스터를 이 값으로 설정한다.

프로세스가 실행되면서 재미있는 일이 발생한다. 프로세스에 의해 생성되는 모든 주소가 다음과 같은 방법으로 프로세서에 의해 **변환**된다.

```
physical address = virtual address + base
```

프로세스가 생성하는 메모리 참조는 **가상 주소**이다. 하드웨어는 베이스 레지스터의 내용을 이 주소에 더하여 **물리 주소**를 생성한다.

이를 이해하기 위해 명령어가 실행될 때 무슨 일이 벌어지는지 추적해 보자. 우리의 예전 코드 중 하나를 살펴보자.

```
128: movl 0x0(%EBX), % eax
```

프로그램 카운터(PC)는 128로 설정된다. 하드웨어가 이 명령어를 반입할 때, 먼저 PC 값을 베이스 레지스터의 값 32 KB(32768)에 더해 32896의 물리 주소를 얻는다. 그런 후 하드웨어는 해당 물리 주소에서 명령어를 가져온다. 그리고 프로세서는 명령어의 실행을 시작한다. 얼마 후 프로세스는 가상 주소 15 KB의 값을 탑재하라는 명령어를 내린다. 이 주소를 프로세서가 받아 다시 베이스 레지스터(32 KB)를 더하고 물리 주소 47 KB에서 원하는 내용을 탑재한다.

가상 주소에서 물리 주소로의 변환이 **주소 변환**이라고 부르는 바로 그 기술이다. 하드웨어는 프로세스가 참조하는 가상 주소를 받아들여 데이터가 실제로 존재하는 물리 주소로

여담: 소프트웨어-기반 재배치

초창기, 하드웨어 지원이 제공되기 전, 일부 시스템은 소프트웨어만으로 엉성하게 재배치를 수행하였다. 이 기술은 **정적 재배치(static relocation)**라고 불리고, **로더(loader)**라 불리는 소프트웨어가 실행하고자 하는 실행 파일의 모든 주소를 원하는 물리 메모리 오프셋으로 변경한다.

예를 들어, 명령어가 주소 1000번지로부터 레지스터로 탑재하는 경우(예를 들어, `movl 1000, %eax`) 그리고 프로그램 주소 공간이 주소 3000부터 탑재되었다면(프로그램이 생각하는 주소 0이 아닌), 로더는 명령어의 모든 주소를 3000씩 이동한 주소로 다시 작성한다(예를 들어, `movl 4000, %eax`). 이런 식으로 간단하게 프로세스 주소 공간의 정적 재배치가 이루어진다.

정적 재배치는 문제가 많다. 첫 번째 가장 중요한 문제는 보호 기능이 없다는 것이다. 잘못된 주소를 생성하여 다른 프로세스나 운영체제의 메모리를 불법적으로 접근할 수 있다. 제대로 보호하기 위해서는 하드웨어 지원이 필요하다 [Wah+93]. 또 다른 단점은 한 번 배치되면 추후 주소 공간을 재배치하는 것이 어렵다는 것이다 [McG65].

팁: 하드웨어 기반의 동적 재배치

동적 재배치를 구현하기 위해서는 작은 하드웨어면 큰 성공을 거둔다. 즉, 베이스 레지스터는 프로그램에 의해 생성된 가상 주소를 물리 주소로 변환하는 데 사용된다. **베이스**(또는 **한계**) 레지스터는 이 가상 주소가 주소 공간의 범위 내에 있다는 것을 보장한다. 두 레지스터는 함께 간단하고 효율적인 메모리의 가상화를 제공한다.

변환한다. 이 주소의 재배치는 실행 시에 일어나고, 프로세스가 실행을 시작한 이후에도 주소 공간을 이동할 수 있기 때문에, **동적 재배치(dynamic relocation)** [McG65]라고도 불린다.

이제 다음과 같은 의문을 가질 수 있다. 바운드(한계) 레지스터는 어디에 쓰는 거지? 베이스와 바운드 방식이라고 하지 않았나? 사실 맞다. 예측했을지 모르지만, 바운드 레지스터는 보호를 지원하기 위해 존재한다. 프로세서는 먼저 메모리 참조가 합법적인가를 확인하기 위해 가상 주소가 바운드 안에 있는지 확인한다. 앞에서 다룬 간단한 예에서, 바운드 레지스터는 항상 16 KB로 설정된다. 프로세스가 바운드보다 큰 가상 주소 또는 음수인 가상 주소를 참조하면 CPU는 예외를 발생시키고 프로세스는 종료될 것이다. 바운드의 요점은 프로세스가 생성한 모든 주소가 합법적이고 프로세스의 "범위"에 있다는 것을 확인하는 것이다.

베이스와 바운드 레지스터는 CPU 칩 상에 존재하는 하드웨어 구조임을 주의하라 (CPU당 1쌍). 주소 변환에 도움을 주는 프로세서의 일부를 **메모리 관리 장치(memory management unit, MMU)**라고 부르기도 한다. 더 정교한 메모리 관리 기법을 개발할수록 MMU에 더 많은 회로를 추가하게 될 것이다.

바운드 레지스터는 두 가지 방식 중 하나로 정의될 수 있다. 한 가지 방법은 앞에서처

럼 주소 공간의 *크기*를 저장하는 방식으로 하드웨어는 가상 주소를 베이스 레지스터에 더하기 전에 먼저 바운드 레지스터와 비교한다. 두 번째 방식은 주소 공간의 마지막 *물리 주소*를 저장하는 방식으로 하드웨어는 먼저 베이스 레지스터를 더하고 그 결과가 바운드 안에 있는지 검사한다. 두 방법 모두 논리적으로는 동일하다. 논의를 간단하게 하기 위해 전자의 방식을 사용한다고 가정한다.

예제

베이스와 바운드를 이용한 주소 변환을 상세하게 이해하기 위해 하나의 예를 살펴보자. 주소 공간의 크기가 4 KB(비현실적으로 작기는 하다)인 프로세스가 물리 주소 16 KB에 탑재되어 있다고 가정하자. 주소 변환의 결과는 다음과 같다.

가상 주소		물리 주소
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	폴트(바운드 벗어남)

예에서 볼 수있는 것처럼 물리 주소를 얻기 위해서는 간단히 가상 주소에 베이스 주소를 더하기만 하면 된다 (가상 주소를 주소 공간 내에서의 오프셋을 나타낸다고 생각할 수 있음). 가상 주소가 너무 크거나 음수일 경우에 폴트를 일으키고 예외가 발생하게 된다.

18.4 하드웨어 지원: 요약

필요한 하드웨어 지원을 요약해 보자(그리고 그림 18.3을 보자). 먼저 CPU 가상화 장에서 설명했듯이 두 가지 CPU 모드가 필요하다. 운영체제는 **특권 모드**(또는 **커널 모드**)에서 실행하며 컴퓨터 전체에 대한 접근 권한을 가진다. 응용 프로그램은 **사용자 모드**에서 실행되며, 할 수 있는 일에 제한이 있다. **프로세서 상태 워드**(**processor status word**) 레지스터의 한 비트가 CPU의 현재 실행 모드를 나타낸다. 특정한 순간에 (예를 들면, 시스템 콜 또는 다른 종류의 예외나 인터럽트 발생 시) CPU는 모드를 전환한다.

하드웨어는 **베이스와 바운드 레지스터**를 자체적으로 제공한다. CPU는 **메모리 관리 장치**(**MMU**)의 일부인 추가의 레지스터 쌍을 가진다. 프로그램이 실행 중인 경우, 하드웨어는 프로그램이 생성한 가상 주소에 베이스 값을 더하여 주소를 변환한다. 하드웨어는 주소가 유효한지 검사할 수 있어야 한다. 이 검사는 바운드 레지스터와 CPU 내의 일부 회로를 사용하여 이루어진다.

하드웨어는 베이스와 바운드 레지스터 값을 변경하는 명령어를 제공해야 한다. 다른 프로세스를 실행시킬 때 운영체제가 이 명령어를 사용하여 베이스와 바운드 레지스터 값을 변경할 수 있다. 이 명령어들은 특권 명령어이다. 커널 모드(또는 **특권** 모드)에서만

여담: 자료 구조—빈 공간 리스트

운영체제는 프로세스에게 메모리를 할당할 수 있도록 사용되지 않는 메모리 공간의 리스트를 유지한다. 다양한 자료 구조가 사용될 수 있다. 가장 간단한 (여기서 가정하고 있는) 자료 구조는 **빈 공간 리스트(free list)**이다. 이 리스트는 현재 사용 중이지 않은 물리 메모리 영역의 리스트이다.

하드웨어 요구사항	노트
특권 모드	사용자 모드 프로세스가 특권 연산을 실행하는 것을 방지하기 위해 필요
베이스/바운드 레지스터	주소 변환과 범위 검사를 지원하기 위하여 CPU 당 한 쌍의 레지스터가 필요
가상 주소를 변환하고 범위 안에 있는지 검사하는 능력	주소 변환과 범위 검사를 위한 회로. 매우 간단함
베이스/바운드를 갱신하기 위한 특권 명령어	프로그램 시작 전에 운영체제가 베이스와 바운드 레지스터 값을 지정할 수 있어야 함
예외 핸들러 등록을 위한 특권 명령어	운영체제가 예외 처리 코드를 하드웨어에게 알려줄 수 있어야 함
예외 발생 기능	프로세스가 특권 명령어 실행을 시도하거나 범위를 벗어난 메모리의 접근을 시도할 때 예외를 발생시킬수 있어야 함

〈그림 18.3〉 **동적 재배치: 하드웨어 요구사항**

레지스터를 변경할 수 있다. 사용자 프로세스가 실행 중에 베이스 레지스터를 임의대로 변경하여 야기할 혼란[1]을 상상해 보라. 악몽과 같다. 즉시 이러한 어두운 생각을 머릿속에서 지워 버려라.

마지막으로, CPU는 사용자 프로그램이 바운드를 벗어난 주소로 불법적인 메모리 접근을 시도하려는 상황에서 **예외**를 발생시킬 수 있어야 한다. 이 경우 CPU는 사용자 프로그램의 실행을 중지하고 운영체제의 "바운드 벗어남" **예외 핸들러(exception handler)**가 실행되도록 조치해야 한다. 운영체제 핸들러는 어떻게 대처할지 결정할 수 있다. 이 경우에 프로세스를 종료시킬 확률이 매우 높다. 마찬가지로 사용자 프로그램이 특권이 필요한 베이스와 바운드 레지스터 값의 변경을 시도하면 CPU는 예외를 발생시키고 "사용자 모드에서의 특권 연산 발생" 핸들러를 실행시켜야 한다. CPU는 이 핸들러들의 주소를 파악하기 위한 방법을 제공해야 한다. 몇 개의 특권 명령어가 더 필요하다.

1) 피해를 입힐 게 "대규모 피해" 밖에 없나?

18.5 운영체제 이슈

동적 재배치 지원을 위해 하드웨어가 새로운 기능을 제공하는 것과 마찬가지로, 운영체제에도 새로운 이슈가 등장한다. 하드웨어 지원과 운영체제 관리가 결합되면 간단한 가상 메모리를 구현할 수 있다. 베이스와 바운드 방식의 가상 메모리 구현을 위해서 운영체제가 반드시 개입되어야 하는 중요한 세 개의 시점이 존재한다.

첫째, 프로세스가 생성될 때 운영체제는 주소 공간이 저장될 메모리 공간을 찾아 조치를 취해야 한다. 다행히도 각 주소 공간은 (a) 물리 메모리의 크기보다 작고 (b) 크기가 일정하다라는 가정하에서는 운영체제가 쉽게 처리할 수 있다. 운영체제는 물리 메모리를 슬롯의 배열로 보고 각 슬롯의 사용여부를 관리한다. 새로운 프로세스가 생성되면 운영체제는 새로운 주소 공간 할당에 필요한 영역을 찾기 위해 (흔히 **빈 공간 리스트**(**free list**)라고 불리는) 자료 구조를 검색해야 한다. 검색을 통해 선택된 공간을 사용 중이라고 표시한다. 가변 크기 주소 공간의 경우, 더 복잡해질 테지만 그에 대한 걱정은 미래의 장으로 넘기자.

예를 들어 살펴보자. 앞서 살펴 본 그림 18.2에서 물리 메모리의 첫 번째 슬롯을 운영체제 자신이 사용하는 것을 보았다. 운영체제는 예의 프로세스의 위치를 물리 주소 32 KB에서 시작하는 슬롯으로 재배치한 것을 볼 수 있다. 다른 두 슬롯(16 KB-32 KB와 48 KB-64 KB)은 비어 있다. 따라서 **빈 공간 리스트**는 이 두 항목으로 구성된다.

둘째, 프로세스가 종료할 때, 즉 정상적으로 종료될 때 또는 잘못된 행동을 하여 강제적으로 죽게될 때 프로세스가 사용하던 메모리를 회수하여 다른 프로세스나 운영체제가 사용할 수 있게 해야 한다. 프로세스가 종료하면, 운영체제는 종료한 프로세스의 메모리를 다시 빈 공간 리스트에 넣고 연관된 자료 구조를 모두 정리한다.

셋째, 운영체제는 문맥 교환이 일어날 때에도 몇 가지 추가 조치를 취해야 한다. CPU마다 한 쌍의 베이스-바운드 레지스터만 존재하고 각 프로그램은 다른 물리 주소에 탑재되어야 하기 때문에 실행 중인 프로그램마다 다른 값을 가진다. 운영체제는 프로세스 전환 시 베이스와 바운드 쌍을 저장하고 복원해야 한다. 운영체제가 실행 중인 프로세스를 중단시키기로 결정하면 운영체제는 메모리에 존재하는 프로세스 별 자료 구조 안에 베이스와 바운드 레지스터의 값을 저장해야 한다. 이 자료 구조는 **프로세스 구조체**(**process structure**) 또는 **프로세스 제어 블럭**(**process control block**, **PCB**)이라고 불린다. 마찬가지로 운영체제는 실행 중인 프로세스를 다시 시작할 때 또는 처음 실행시킬 때, 이 프로세스에 맞는 값으로 CPU의 베이스와 바운드 값을 설정해야 한다.

프로세스가 중단되면(즉, 실행 상태가 아닐 때에는), 운영체제가 메모리의 현 위치에서 다른 위치로 주소 공간을 비교적 쉽게 옮길 수 있다는 사실에 주목해야 한다. 프로세스의 주소 공간을 이동시키려면 운영체제는 먼저 프로세스의 실행을 중지시킨다. 그런 후 운영체제는 현재 위치에서 새 위치로 주소 공간을 복사한다. 마지막으로, 운영체제는 프로세스 구조체에 저장된 베이스 레지스터를 갱신하여 새 위치를 가리키도록 한다. 프로세스가 실행을 재개하면 새로운 베이스 레지스터가 복원되고 다시 실행을 시작하고, 명령어와 데이터가 전혀 다른 새 위치에 존재한다는 사실을 인식하지 못한다.

넷째, 위에서 언급한 것처럼 운영체제는 예외 핸들러 또는 호출될 함수를 제공해

운영체제 요구사항	노트
메모리 관리	새 프로세스의 메모리 할당에 필요;
	종료 프로세스로부터 메모리 회수;
	빈 공간 리스트를 통한 일반적인 메모리 관리
베이스/바운드 관리	문맥 교환 시 올바르게 베이스/바운드 설정
예외 처리	예외가 발생할 때 실행할 코드;
	범법 프로세스를 종료하는 것이 가능성이 높은 처리

〈그림 18.4〉 **동적 재배치 : 운영체제 책임**

야 한다. 운영체제는 부팅할 때 특권 명령어를 사용하여 이 핸들러를 설치한다. 예를 들어, 프로세스가 바운드 밖의 메모리에 접근하려는 경우 CPU는 예외를 발생시킨다. 운영체제는 이러한 예외가 발생할 때 조치를 취할 준비가 되어 있어야 한다. 운영체제의 일반적인 대응은 적대적인 양상을 보인다. 운영체제는 불법 행위를 한 프로세스를 종료시킨다. 운영체제는 자신이 실행되는 컴퓨터를 보호해야 한다. 허가 안 된 메모리에 접근하거나 또는 명령어를 실행하려는 프로세스를 좋아하지 않는다. 잘가라~, 잘못 행동한 프로세스여. 너를 만나서 반가웠어.

그림 18.5는 하드웨어/OS의 상호작용을 타임라인으로 보여준다. 그림은 부팅할 때 컴퓨터를 사용 가능한 상태로 만들기 위하여 운영체제가 무엇을 하는지 보여 준다. 그리고 프로세스(Process A)가 실행을 시작할 때 무슨 일이 일어나는지 보여 준다. 메모리 변환은 운영체제의 개입 없이 하드웨어에 처리된다는 사실에 주목하라. 시간이 흐른 후에 타이머 인터럽트가 발생하고 운영체제는 Process B로 전환한다. Process B는 불법적인 주소에 잘못된 탑재를 실행한다. 그 시점에 운영체제가 개입하여 프로세스를 끝낸 후 B의 메모리를 해제하고 프로세스 테이블에서 해당 항목을 제거하여 정리한다. 그림에서 볼 수 있듯이, 우리는 아직 **제한적 직접 실행**의 기본적인 접근 방식을 따르고 있다. 대부분의 경우, 운영체제는 다만 하드웨어를 적절하게 설정하고 프로세스가 CPU에서 직접 실행할 수 있게 한다. 프로세스가 잘못된 행동을 했을 때에만 운영체제가 개입하여야 한다.

18.6 요약

이 장에서는 주소 변환이라고 알려진 가상 메모리 기법을 통해 제한적 직접 실행의 개념을 확장하였다. **주소 변환**을 사용하면 운영체제는 프로세스의 모든 메모리 접근을 제어할 수 있고, 접근이 항상 주소 공간의 범위 내에서 이루어지도록 보장할 수 있다. 이 기술의 효율성의 열쇠는 하드웨어 지원이다. 하드웨어 지원은 프로세스가 이해하는 메모리인 가상 주소를 실제 메모리 모습인 물리 주소로 변환하며 이 변환을 빠르게 수행한다. 이 모든 것은 재배치된 프로세스에게 투명한 방식으로 이루어진다.

운영체제 @ 부트 **(커널 모드)**	**하드웨어**	
트랩 테이블을 초기화		
	다음의 주소를 기억 시스템 콜 핸들러 타이머 핸들러 불법적인 메모리 접근 핸들러 불법적인 명령어 핸들러	
인터럽트 타이머 시작		
	타이머 시작. X msec 후에 인터럽트	
프로세스 테이블 초기화		
빈 공간 리스트 초기화		

운영체제 @ 실행 **(커널 모드)**	**하드웨어**	**프로그램** **(사용자 모드)**
프로세스 A를 시작하기 위해: 프로세스 테이블의 항목 할당 프로세스 메모리 할당 베이스/바운드 레지스터 설정 return-from-trap (A로)		
	A의 레지스터 복원 **사용자 모드**로 이동 A의 (시작) PC로 분기	
		프로세스 A 실행 명령어 반입
	가상 주소 변환 및 반입 실행	
		명령어 실행
	명시적 탑재/저장이라면 주소가 범위 안 인지 확인 가상 주소 변환 및 탑재/저장 실행	
		…
	타이머 인터럽트 **커널 모드**로 이동 인터럽트 핸들러로 분기	
트랩 처리 **switch()** 루틴 호출 A의 레지스터를 A의 **proc-struct**에 저장(베이스/바운드 포함) B의 레지스터를 B의 **proc-struct**에서 복원(베이스/바운드 포함) **return-from-trap** (B로)		
	B의 레지스터 복원 **사용자 모드**로 이동 B의 PC로 분기	
		프로세스 B 실행 불법적인 탑재 실행
	바운드-밖 탑재 **커널 모드**로 이동 트랩 핸들러로 분기	
트랩 처리 프로세스 B 종료시키기로 결정 B의 메모리 회수 프로세스 테이블의 B의 항목 해제		

〈그림 18.5〉 **제한된 실행 프로토콜(동적 재배치)**

베이스와 바운드(base-and-bound) 또는 동적 재배치로 알려진 가상화의 한 형태를 살펴보았다. 베이스 레지스터를 가상 주소에 더하고 생성된 주소가 바운드를 벗어나는지 검사하기 위한 간단한 하드웨어 회로만 추가하면 되기 때문에 base-and-bound 가상화는 매우 *효율적이다*. base-and-bound 가상화는 또한 *보호* 기능도 제공한다. 운영체제와 하드웨어는 협력하여 프로세스가 자신의 주소 공간 이외의 밖의 메모리는 참조를 할 수 없도록 한다. 보호는 운영체제의 가장 중요한 목표 중의 하나이다. 보호 없이는 운영체제가 컴퓨터를 통제할 수 없다. 프로세스가 자유롭게 메모리를 덮어쓸 수 있다면, 그들은 트랩 테이블을 덮어써 시스템을 장악하는 등의 위험한 일을 쉽게 할 수 있다.

동적 재배치는 비효율적이다. 예를 들어, 그림 18.2에서 볼 수 있듯이, 재배치된 프로세스는 32 KB에서 48 KB까지의 물리 메모리를 사용한다. 그러나 프로세스 스택과 힙이 아주 크지 않기 때문에, 둘 사이의 공간이 단순히 낭비되고 있다. 할당된 영역의 내부 공간이 사용되지 않기 때문에, 즉 단편화가 발생되어 *낭비된다*. 이런 유형의 낭비를 **내부 단편화**(**internal fragmentation**)라고 한다. 현재 접근 방식에서 비록 더 많은 프로세스를 탑재할 수 있는 충분한 물리 메모리가 있더라도, 고정 크기의 슬롯에 주소 공간을 배치해야 하기 때문에 내부 단편화가 발생한다[2]. 물리 메모리의 이용률을 높이고 내부 단편화를 방지하기 위해 더 정교한 기법이 필요하다. 첫 번째 시도는 base-and-bound를 일반화하는 것이다. 이러한 일반화된 베이스-바운드 기법을 **세그멘테이션**(**segmentation**)이라고 부르고, 다음 장에서 논의할 것이다.

2) 다른 해결책은 고정 크기의 스택을 코드 바로 아래에 배치하고 힙을 그 아래에 배치하여 힙이 아래 쪽으로 확장하게 하는 것이다. 그러나 이것은 재귀 호출과 함수 호출의 중첩이 긴 경우 그 구현을 어렵게 만들기 때문에 융통성에 제약을 주고 따라서 이는 피하고 싶다.

참고 문헌

[McG65] **"On Dynamic Program Relocation"**
W.C. McGee
IBM Systems Journal Volume 4, Number 3, 1965, pages 184-199
동적 재배치에 관한 초기 연구뿐 아니라 일부 정적 재배치의 기본에 대해 잘 요약한 논문이다.

[Pil90] **"Relocating loader for MS-DOS .EXE executable files"**
Kenneth D.A. Pillay
Microprocessors & Microsystems archive Volume 14, Issue 7 (September 1990)
MS-DOS를 위한 재배치 로더의 예. 최초는 아니지만 그러한 시스템의 동작에 관한 상대적으로 현대적인 예를 설명한다.

[SS74] **"The Protection of Information in Computer Systems"**
J. Saltzer and M. Schroeder
CACM, July 1974
*논문에서 "베이스와 바운드 레지스터의 개념과 하드웨어-해석 설명자는 분명히 독립적으로 1957년과 1959년에 다양한 목표를 가진 세 개의 프로젝트에서 선보였다. MIT의 McCarthy가 시분할 시스템을 구현하기 위한 메모리 보호 시스템의 일부로 베이스-바운드를 제안하였다. IBM은 Stretch (7030) 컴퓨터 시스템의 신뢰성 있는 다중프로그래밍을 허용하기 위한 기법으로서 독립적으로 베이스-바운드 레지스터를 개발하였다. Burroughs에서는 R. Barton이 하드웨어-해석 설명자가 B5000 컴퓨터 시스템의 고수준 언어의 명칭 범위 규칙을 직접적으로 지원한다고 제안하였다." 우리는 이 인용을 Mark Smotherman의 멋진 역사 페이지 [**S04**]에서 발견하였다. 더 자세한 사항은 그 페이지를 참조하도록.*

[Wah+93] **"Efficient Software-based Fault Isolation"**
Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham
SOSP '93
프로그램에서 메모리 참조를 범위 내에 두기 위하여 하드웨어의 도움 없이 컴파일러의 지원을 사용할 수 있는 방법에 관한 굉장한 논문. 이 논문은 메모리 참조의 고립을 위한 소프트웨어 기술에 관한 새로운 관심을 불러 일으켰다.

숙제

프로그램 `relocation.py`는 베이스와 바운드 레지스터를 가진 시스템에서 주소 변환이 어떻게 일어나는지 볼 수 있게 한다. 자세한 내용은 README를 참조하시오.

문제

1. 시드 1, 2 및 3을 가지고 실행하고 프로세스에 의해 생성된 각 가상 주소가 범위 내에 있는지 계산하라. 바운드 안에 있다면 주소 변환을 수행하라.
2. 다음과 같은 플래그를 주고 실행하라: `-s 0 -n 10`. 생성된 모든 가상 주소가 범위 안에 있는 것을 보장하기 위해서는 -l을 어떤 값으로 설정해야 하는가?
3. 다음과 같은 플래그를 주고 실행하라: `-s 1 -n 10 -l 100`. 주소 공간 전체가 여전히 물리 메모리에 들어가려면 설정할 수 있는 바운드 레지스터의 최댓값은 얼마인가?
4. 더 큰 주소 공간(`-a`)과 물리 메모리(`-p`)를 설정하여 위 문제와 동일하게 실행시켜 보아라.
5. 바운드 레지스터의 값이 변함에 따라 임의로 생성된 가상 주소 중 유효한 주소의 비율은 얼마인가? 다른 랜덤 시드를 가지고 실행한 결과를 그래프로 나타내시오. 값의 범위는 0부터 주소 공간의 최대 크기로 한다.