9

제한적 직접 실행 원리

CPU를 가상화하기 위해서 운영체제는 여러 작업들이 동시에 실행되는 것처럼 보이도록 물리적인 CPU를 공유한다. 기본적인 아이디어는 간단하다. 한 프로세스를 잠시 동안 실행하고 다른 프로세스를 또 잠깐 실행하고, 이런 식으로 계속해서 잠깐씩 실행시키면 된다. 이러한 방식처럼 **CPU 시간을 나누어 씀**으로써 가상화를 구현할 수 있다.

그러나 이러한 가상화 기법을 구현하기 위해서는 몇 가지 문제를 해결해야 한다. 첫 번째는 *성능 저하*이다. 시스템에 과중한 오버헤드를 주지 않으면서 가상화를 구현할 수 있을까? 두 번째는 *제어 문제*이다. CPU에 대한 통제를 유지하면서 프로세스를 효율적으로 실행시킬 수 있는 방법은 무엇인가? 운영체제의 입장에서는 자원 관리의 책임자로서 특히 제어 문제가 중요하다. 제어권을 상실하면 한 프로세스가 영원히 실행을 계속할 수 있고 컴퓨터를 장악하거나 접근해서는 안 되는 정보에 접근하게 된다. 제어권을 유지하면서 성능 저하가 없도록 하는 것이 운영체제를 구축하는 데 핵심적인 도전 과제이다.

핵심 질문: 제어를 유지하면서 효과적으로 CPU를 가상화하는 방법

운영체제는 효율적인 방식으로 CPU를 가상화하지만 시스템에 대한 제어를 잃지는 않는다. 제어를 잃지 않기 위해서 하드웨어와 운영체제의 지원이 필수적이다. 종종 운영체제는 작업을 효과적으로 수행하기 위하여 하드웨어가 제공하는 기능을 신중하게 사용한다.

9.1 기본 원리: 제한적 직접 실행

운영체제 개발자들은 프로그램을 빠르게 실행하기 위하여 **제한적 직접 실행**(**Limited Direct Execution**) 이라는 기법을 개발하였다. 이 아이디어의 "직접 실행"에 해당하는 부분은 간단하다. 프로그램을 CPU 상에서 그냥 직접 실행시키는 것이다. 따라서 운영체제가 프로그램을 실행하기 시작할 때 프로세스 목록에 해당 프로세스 항목을 만들고 메모리를 할당하며 프로그램 코드를 디스크에서 탑재하고 진입점(예, `main()` 루틴 혹은 유사한 무엇)을 찾아 그 지점으로 분기하여 사용자 코드를 실행하기 시작한다. 그림

운영체제	프로그램
프로세스 목록의 항목을 생성	
프로그램 메모리 할당	
메모리에 프로그램 탑재	
`argc/argv`를 위한 스택 셋업	
레지스터 내용 삭제	
`call main()` 실행	
	`main()` 실행
	`main`에서 `return` 명령어 실행
프로세스 메모리 반환	
프로세스 목록에서 항목 제거	

〈그림 9.1〉 **직접 실행 프로토콜(제한 없음)**

9.1은 이 기본적인 직접 실행 프로토콜(아무 제한이 없는)을 보이고 있다. 그림 9.1은 프로그램의 `main()`으로 분기하고 커널로 되돌아 가기 위해 일반적인 호출과 리턴을 사용하였다.

간단하다, 그렇지? 그러나 이 접근법은 CPU를 가상화함에 있어 몇 가지 문제를 일으킨다. 첫 번째는 간단한 문제이다. 프로그램을 직접 실행시킨다면 프로그램이, 운영체제가 원치않는 일을 하지 않는다는 것을 어떻게 보장할 수 있는가? 두 번째 문제는 프로세스 실행 시, 운영체제는 어떻게 프로그램의 실행을 중단하고 다른 프로세스로 전환시킬 수 있는가, 즉, CPU를 가상화하는 데 필요한 **시분할**(**time sharing**) 기법을 어떻게 구현할 수 있느냐는 것이다.

아래에서 이와 같은 질문에 답하면서 CPU를 가상화하는 데 필요한 사항을 더 잘 이해하게 될 것이다. 이러한 기법을 발전시키는 과정 중에 "제한적"이라는 이름이 어디서 비롯되었는지 보게 될 것이다. 프로그램 실행에 제한을 두지 않으면 운영체제는 어떠한 것도 제어할 수 없으며 따라서 단순한 라이브러리일 뿐이다. 장차 운영체제로 발전하기 위해서는 매우 슬픈 상황이다.

9.2 문제점 1: 제한된 연산

직접 실행의 장점은 빠르게 실행된다는 것이다. 기본적으로 프로그램이 하드웨어 CPU에서 실행되기 때문이다. 그러나 CPU에서 직접 실행시키면 새로운 문제가 발생한다. 만일 프로세스가 특수한 종류의 연산을 수행하길 원한다면 어떻게 될 것인가? 이러한 연산에는 디스크 입출력 요청이나 CPU 또는 메모리와 같은 시스템 자원에 대한 추가할당 요청 등이 포함된다.

프로세스가 원하는 대로 할 수 있게 방치하는 방안이 있다. 그러나 이 방안은 바람직한 시스템을 구축하는 데에는 방해 요인이다. 파일에 대한 접근을 허용하기 전에 접근 권한을 검사하는 파일 시스템을 구현하는 것을 예를 들어 보자. 프로세스가 디스크에

핵심 질문: 제한 연산을 수행하는 방법

프로세스는 입출력 연산을 비롯한 다른 제한된 연산을 수행해야 한다. 그러나, 프로세스는 시스템에 대한 권한이 없기 때문에 제한된 연산을 수행할 수 없다. 이 일을 위해 운영체제와 하드웨어가 할 일은 무엇인가?

팁: 보호된 제어 양도

하드웨어는 두 가지 실행 모드를 제공하여 운영체제를 돕는다. **사용자 모드(user mode)**에서 응용 프로그램은 하드웨어 자원에 대한 접근 권한이 일부 제한되어 있다. 운영체제는 컴퓨터의 모든 자원에 대한 접근 권한을 **커널 모드(kernel mode)**에서 가진다. 이를 위하여 커널 모드로 진입하기 위한 trap명령어와 사용자 모드로 돌아가기 위한 `return-from-trap` 명령어가 제공된다. 또한, 운영체제가 하드웨어에게 트랩 테이블(trap table)의 메모리 주소를 알려주기 위한 명령어도 함께 제공된다.

대하여 입출력하는 것을 제한하지 않으면 프로세스는 전체 디스크를 읽고 쓸 수 있기 때문에 접근 권한을 검사하는 기능이 아무런 의미가 없다.

이 때문에 **사용자 모드(user mode)**라고 알려진 새로운 모드가 도입되었다. 사용자 모드에서 실행되는 코드는 할 수 있는 일이 제한된다. 예를 들어, 프로세스가 사용자 모드에서 실행 중이면 입출력 요청을 할 수 없도록 설정한다. 이때 입출력 요청을 하면 프로세서가 예외를 발생시키고, 운영체제는 해당 프로세스를 제거한다.

커널 모드(kernel mode)는 사용자 모드와 대비되는 모드로서 운영체제의 중요한 코드들이 실행된다. 이 모드에서 실행되는 코드는 모든 특수한 명령어를 포함하여 원하는 모든 작업을 수행할 수 있다.

그러나 우리에겐 아직 해결해야 할 한 가지 문제가 남아 있다. 사용자 프로세스가 디스크를 읽기와 같은 특권 명령어를 실행해야 할 때는 어떻게 해야 하는가? 이런 제한 작업의 실행을 허용하기 위하여 거의 모든 현대 하드웨어는 사용자 프로세스에게 **시스템 콜**을 제공한다. Atlas [Kil+62; Lav78]와 같은 커널은 시스템 콜을 통하여 자신의 주요 기능을 사용자 프로그램에게 제공한다. 이러한 기능에는 파일 시스템 접근, 프로세스 생성 및 제거, 다른 프로세스와의 통신 및 메모리 할당 등이 포함된다. 대부분의 운영체제는 수백 개의 시스템 콜을 제공한다(자세한 사항에 관해서는 POSIX 표준을 참조하시오 [P10]). 초기 Unix 시스템은 약 20개 정도의 시스템 콜을 제공하였다.

시스템 콜을 실행하기 위해 프로그램은 **trap** 특수 명령어를 실행해야 한다. 이 명령어는 커널 안으로 분기하는 동시에 특권 수준을 커널 모드로 상향 조정한다. 커널 모드로 진입하면 운영체제는 모든 명령어를 실행할 수 있고 이를 통하여 프로세스가 요청한 작업을 처리할 수 있다. 완료되면 운영체제는 **return-from-trap** 특수 명령어을 호출한다. 예상하는 것처럼 이 명령어는 특권 수준을 사용자 모드로 다시 하향 조정하면서 호출한 사용자 프로그램으로 리턴한다.

하드웨어는 `trap` 명령어를 수행할 때 주의가 필요하다. 호출한 프로세스의 필요한

여담: 왜 시스템 콜은 프로시저 콜과 비슷하게 보일까

`open()` 또는 `read()`와 같은 시스템 콜을 호출하는 형태는 C 프로그램의 전형적인 프로시저 호출의 형태와 똑같다. 즉, 만약 프로시저 호출과 똑같다면 운영체제는 시스템 콜을 어떻게 구분할 수 있으며 그에 맞는 작업을 할 수 있는가? 간단한 이유: 진짜 프로시저 호출이지만 유명한 trap 명령어가 그 안에 감추어져 있다. 더 정확하게 설명하면, 예를 들어, `open()`을 호출할 때, C 라이브러리의 프로시저 호출을 수행하고 있는 것이다. `open()`(다른 시스템 콜들도 마찬가지이다)이 호출되면, 라이브러리는 커널과 약속된 호출 규약을 사용하여 `open`에게 전달할 인자와 시스템 콜 번호를 지정된 장소(스택 또는 특정 레지스터)에 저장한다. 그런 다음 `trap` 명령어를 실행한다. 그 다음에 반환 값을 정리하고 시스템 콜을 호출한 프로그램에게 제어를 넘긴다. 이를 위해서 C 라이브러리에서 시스템 콜을 호출하는 부분의 코드는 어셈블리어로 작성된다. 인자의 반환 값을 올바르게 처리하고, 하드웨어마다 다른 `trap` 명령어를 실행하기 위해 꼭 필요한 일이다. 자 이제 운영체제로 진입하기 위해서 어셈블리 코드 작업을 직접할 필요가 없다는 것을 알았다. 어셈블리 코드는 이미 만들어져 있다.

레지스터들을 저장해야 한다. 운영체제가 `return-from-trap` 명령어 실행 시 사용자 프로세스로 제대로 리턴할 수 있도록 하기 위함이다. 예를 들면, x86에서는 프로그램 카운터, 플래그와 다른 몇 개의 레지스터를 각 프로세스의 **커널 스택**(**kernel stack**)에 저장한다. `return-from-trap` 명령어가 이 값들을 스택에서 팝(pop)하여 사용자 모드 프로그램의 실행을 다시 시작한다(Intel systems manual [Int11]을 참조하시오). 다른 하드웨어 시스템은 다른 규약을 사용하긴 하지만 기본적인 개념은 모두 비슷하다.

현재까지의 논의에서 다루지 않은 중요한 사항이 있다. 그것은 `trap`이 운영체제 코드의 어디를 실행할지 어떻게 아느냐는 것이다. 호출한 프로세서는 분기할 주소를 명시할 수 없다. 주소를 명시한다는 것은 커널 내부의 원하는 지점을 접근할 수 있다는 것이기 때문에 위험하다. 커널이 임의의 코드를 실행하기 위해서는 접근 권한 검사가 끝난 후 분기해야 한다 [Sha07]. 이러한 문제 때문에 커널은 trap 발생 시 어떤 코드를 실행할지 신중하게 통제해야 한다.

커널은 부팅 시에 **트랩 테이블**(**trap table**)을 만들고 이를 이용하여 시스템을 통제한다. 컴퓨터가 부트될 때는 커널 모드에서 동작하기 때문에 하드웨어를 원하는 대로 제어할 수 있다. 운영체제가 하는 초기 작업 중 하나는 하드웨어에게 예외 사건이 일어났을 때 어떤 코드를 실행해야 하는지 알려주는 것이다. 예를 들어, 하드 디스크 인터럽트가 발생하면, 키보드 인터럽트가 발생하면, 또는 프로그램이 시스템 콜을 호출하면 무슨 코드를 실행해야 하는가? 운영체제는 특정 명령어를 사용하여 하드웨어에게 **트랩 핸들러**(**trap handler**)의 위치를 알려준다. 하드웨어는 이 정보를 전달받으면 해당 위치를 기억하고 있다. 따라서 시스템 콜과 같은 예외적인 사건이 발생했을 때 하드웨어는 무엇을 해야 할지(즉, 어느 코드로 분기하여 실행할지) 알 수 있다.

마지막 하드웨어에게 트랩 테이블의 위치를 알려주는 것은 매우 **강력한** 기능이다. 당연히 이 역시 특권 명령어이다. 사용자 모드에서 이 명령어를 실행하려고 하면 실행할

운영체제 @부트 **(커널 모드)**	**하드웨어**	
트랩 테이블을 초기화한다		
	syscall 핸들러의 주소를 기억한다	
운영체제 @실행 **(커널 모드)**	**하드웨어**	**프로그램** **(사용자 모드)**
프로세스 목록에 항목을 추가한다		
프로그램을 위한 메모리를 할당한다		
프로그램을 메모리에 탑재한다		
argv를 사용자 스택에 저장한다		
레지스터와 PC를 커널 스택에 저장한다		
return-from-trap		
	커널 스택으로부터 레지스터를 복원한다	
	사용자 모드로 이동한다	
	main으로 분기한다	
		main()을 실행한다
		...
		시스템 콜을 호출한다
		운영체제로 **트랩**한다
	레지스터를 커널 스택에 저장한다	
	커널 모드로 이동한다	
	트랩 핸들러로 분기한다	
트랩을 처리한다		
syscall의 임무를 수행한다		
return-from-trap		
	커널 스택으로부터 레지스터를 복원한다	
	사용자 모드로 이동한다	
	트랩 이후의 PC로 분기한다	
		main에서 리턴한다
		trap(**exit()**를 통하여)
프로세스의 메모리를 반환한다		
프로세스 목록에서 제거한다		

〈그림 9.2〉 **제한적 직접 실행 프로토콜**

수 없고, 어떤 결과가 올지 알고 있을 것이다(힌트, 잘 가거라. 프로그램). 숙고해 볼 주제 : 만약 당신이 자신의 트랩 테이블을 설치할 수 있다면, 시스템에 어떤 황당한 일이 일어날까? 컴퓨터를 장악할 수 있는가?

그림 9.2는 이 메커니즘을 요약해서 나타내고 있다. 연대표는 시간의 흐름에 따라 아래 방향으로 진행된다. 프로세스는 커널 스택을 각자 가지고 있다. 커널 모드로 진입하거나 진출할 때 하드웨어에 의해 프로그램 카운터와 범용 레지스터 등의 레지스터가 저장되고 복원되는 용도로 사용된다.

LDE 프로토콜은 두 단계로 진행된다. 전반부에서(부팅 시) 커널은 트랩 테이블을 초기화하고 CPU는 나중에 사용하기 위하여 테이블의 위치를 기억한다. 커널은 이러한 작업을 커널 모드에서만 사용할 수 있는 명령어를 이용하여 수행한다(이러한 명령어는 굵게 강조하였다).

후반부에서(프로세스를 실행할 때) return-from-trap을 이용하여 사용자 프로세스를 시작할 때 몇 가지 작업을 수행한다. 새로운 프로세스를 위한 노드를 할당하여 프로세스 리스트에 삽입하고, 메모리를 할당하는 등의 작업이 포함된다. `return-from-trap` 명령어는 CPU를 사용자 모드로 전환하고 프로세스 실행을 시작한다. 프로세스가 시스템 콜을 호출하면 운영체제로 다시 트랩된다. 운영체제는 시스템 콜을 처리하고 `return-from-trap` 명령어를 사용하여 다시 제어를 프로세스에게 넘긴다. 프로세스는 이후 자신의 할 일을 다하면 `main()`에서 리턴한다. 이때 일반적으로 스텁 코드로 리턴하고 스텁 코드가 프로그램을 종료시킨다. 종료시킬 때 `exit()` 시스템을 호출하고 다시 운영체제로 트랩된다. 이 시점에 운영체제는 정리 작업을 하게 되어 모든 일이 완료된다.

9.3 문제점 2: 프로세스 간 전환

직접 실행의 두 번째 문제점은 프로세스 간 전환을 할 수 있어야 한다는 점이다. 프로세스 간 전환은 당연히 간단해야 한다. 맞지? 운영체제는 실행 중인 프로세스를 계속 실행할 것인지, 멈추고 다른 프로세스를 실행할 것인지를 결정해야 한다. 이게 뭐가 문제지? 실제로는 까다로운 문제이다. CPU에서 프로세스가 실행 중이라는 것은 운영체제는 실행 중이지 않다는 것을 의미한다. 운영체제가 실행하고 있지 않다면 어떻게 이런 일들을 할 수 있을까? (힌트: 할 수 없다). 거의 철학적으로 들릴지 모르겠지만 이건 진짜 심각한 문제다. CPU에서 실행하고 있지 않다면 운영체제는 어떠한 조치도 취할 수 없다. 이제 핵심 문제에 도달했다.

핵심 질문: CPU를 어떻게 다시 획들할 수 있는가
운영체제는 어떻게 CPU를 **다시 획득하여** 프로세스를 전환할 수 있는가?

협조 방식: 시스템 콜 기다리기

협조(cooperative) 방식으로 알려진 방법은 과거의 몇몇 시스템에서 채택되었던 방식이다. 예를 들어, Macintosh 운영체제의 초기 버전 [Mac11], 또는 오래 전 Xerox Alto 시스템 [Alt79]이 채택했던 방식이다. 이 방식에서는 운영체제가 프로세스들이 합리적으로 행동할 것이라고 신뢰한다. 너무 오랫동안 실행할 가능성이 있는 프로세스는 운영체제가 다른 작업을 실행할 결정을 할 수 있도록 주기적으로 CPU를 포기할 것이라고 가정한다.

그러면 이렇게 질문할 수 있을 것이다: 이 이상적인 환경에서 우호적인 프로세스는 어떻게 CPU를 포기할 수 있을까? 알려진 것처럼 대부분의 프로세스는 자주 **시스템 콜**을 호출하여 CPU의 제어권을 운영체제에게 넘겨준다. 예를 들어, 파일을 열고 읽는 작업을 한다거나 다른 컴퓨터에게 메시지를 송신하거나 또는 새 프로세스를 생성하는 등의 시스템 콜을 호출한다. 이런 유형의 운영체제는 **yield** 시스템 콜을 제공하는데,

이 시스템 콜은 운영체제에게 제어를 넘겨 운영체제가 다른 프로세스를 실행할 수 있게 한다.

팁: 응용 프로그램의 오작동 처리하기

운영체제는 종종 오작동 프로세스를 처리해야 한다. 오작동하는 프로세스란 악의적으로 설계되었거나 실수로 인한 버그 때문에 해서는 안 될 무언가를 하려는 프로세스를 말한다. 현대 시스템에서 운영체제가 그러한 부정 행위를 처리하는 방법은 단순히 행위자를 종료시키는 것이다. 원 스트라이크에 아웃! 조금 가혹하다고 할지는 모르겠지만, 만약 메모리에 불법적으로 접근하거나 불법적인 명령어를 실행할 때 운영체제가 취할 수 있는 일은 무엇이 있을까?

응용 프로그램이 비정상적인 행위를 하게 되면 운영체제에게 제어가 넘어간다. 예를 들어 응용 프로그램이 어떤 수를 0으로 나누는 연산을 실행하거나 접근할 수 없는 메모리에 접근하려고 하면 운영체제로의 **트랩**이 일어난다. 그러면 운영체제는 다시 CPU를 획득하여 해당 행위를 하는 프로세스를 종료할 수 있다.

협조 방식의 스케줄링 시스템에서 운영체제는 시스템 콜이 호출되기를 기다리거나 불법적인 연산이 일어나기를 기다려서 CPU의 제어권을 다시 획득한다. 이렇게 생각할 수도 있다: 이상적이라기보다 수동적인 방법 아닐까? 예를 들어 악의적이든 버그로 인한 것이든 프로세스가 무한 루프에 빠져 시스템 콜을 호출할 수 없을 때에는 어떤 일이 벌어지는가? 운영체제가 할 수 있는 일은 무엇인가?

비협조 방식: 운영체제가 전권을 행사

프로세스가 시스템 콜을 호출하기를 거부하거나 실수로 호출하지 않아 운영체제에게 제어를 넘기지 않을 경우 하드웨어의 추가적인 도움없이는 운영체제가 할 수 있는 일은 거의 없다. 사실, 협조적 방법에서 프로세스가 무한 루프에 빠졌을 경우 할 수 있는 일은 컴퓨터 시스템의 모든 문제를 해결해 왔던 아주 오래된 방법을 동원하는 수밖에 없다. 바로 **컴퓨터를 다시 부팅**하는 것이다. CPU의 제어권을 획득하기 위한 과정에서 해결해야 할 사소한 문제를 만나게 된다.

핵심 질문: 협조 없이 제어를 얻는 방법

프로세스가 비협조적인 상황에서도 CPU의 제어를 획득하는 방법은 무엇인가? 악의적인 프로세스가 컴퓨터를 장악하지 않도록 보장하기 위하여 운영체제는 무엇을 할 수 있을까?

그 해결책은 간단한 것으로 수십 년 전에 컴퓨터 시스템을 만들었던 많은 사람들에 의해 발명되었다. 해결책은 **타이머 인터럽트**(**timer interrupt**)를 이용하는 것이다 [McC+63]. 타이머 장치는 수 밀리 초마다 인터럽트를 발생시키도록 프로그램

가능하다. 인터럽트가 발생하면 현재 수행 중인 프로세스는 중단되고 미리 구성된 운영체제의 **인터럽트 핸들러(interrupt handler)**가 실행된다. 이 시점에 운영체제는 CPU 제어권을 다시 얻게 되고 자신이 원하는 일을 할 수 있다. 현재의 프로세스를 중단하고 다른 프로세스를 실행시키는 작업 등이 해당된다.

팁: 제어를 다시 획득하기 위해 타이머 인터럽트 사용하기

타이머 인터럽트 기능을 추가하면 운영체제는 프로세스가 비협조적으로 행동하는 상황에서도 CPU 상에서 실행될 수 있는 능력을 가지게 된다. 타이머 인터럽트 하드웨어 기능은 운영체제가 컴퓨터의 제어를 유지하는 핵심적인 기능이다.

운영체제는 하드웨어에게 타이머 인터럽트가 발생했을때 실행해야 할 코드[1]를 알려주어야 한다. 부팅될 때 운영체제가 이런 준비를 한다. 부팅 과정 진행 중에 운영체제는 타이머를 시작한다. 타이머가 시작되면 운영체제는 자신에게 제어가 돌아올 것이라는 것을 알고 부담 없이 사용자 프로그램을 실행할 수 있다. 또한, 타이머는 특정 명령어를 수행하여 끌 수도 있다. 이에 대한 자세한 사항은 병행성을 설명할 때 논의할 것이다.

인터럽트 발생 시 하드웨어에게도 약간의 역할이 있다. 인터럽트가 발생했을 때 실행중이던 프로그램의 상태를 저장하여 나중에 `return-from-trap` 명령어가 프로그램을 다시 시작할 수 있도록 해야 한다. 이러한 일련의 동작은 시스템 콜이 호출되었을 때 하드웨어가 하는 동작과 매우 유사하다. 다양한 레지스터가 커널 스택에 저장되고, `return-from-trap` 명령어를 통하여 복원된다.

문맥의 저장과 복원

시스템 콜을 통하여 협조적으로 하던, 또는 타이머 인터럽트를 통하여 약간은 강제적으로 하던, 운영체제가 제어권을 다시 획득하면 중요한 결정을 내려야 한다. 즉, 현재 실행중인 프로세스를 계속 실행할 것인지 아니면 다른 프로세스로 전환할 것인지를 결정해야 한다. 이 결정은 운영체제의 **스케줄러(scheduler)**라는 부분에 의해 내려진다. 우리는 이후의 장에서 스케줄링 정책에 관해 상세하게 논의할 것이다.

다른 프로세스로 전환하기로 결정되면 운영체제는 **문맥 교환(context switch)**이라고 알려진 코드를 실행한다. 문맥 교환은 개념적으로는 간단하다. 운영체제가 해야 하는 작업은 현재 실행 중인 프로세스의 레지스터 값을 커널 스택 같은 곳에 저장하고 곧 실행될 프로세스의 커널 스택으로부터 레지스터 값을 복원하는 것이 전부다. 그렇게 함으로써 운영체제는 `return-from-trap` 명령어가 마지막으로 실행될 때 현재 실행중이던 프로세스로 리턴하는 것이 아니라 다른 프로세스로 리턴하여 실행을 다시 시작할 수 있다.

프로세스 전환을 위하여 운영체제는 저수준 어셈블리 코드를 사용하여 현재 실행중인 프로세스의 범용 레지스터, PC뿐 아니라 현재 커널 스택 포인터를 저장한다.

1) 특정 작업을 대처하는 코드

그리고 곧 실행될 프로세스의 범용 레지스터, PC를 복원하고 커널 스택을 이 프로세스의 커널 스택으로 전환한다. 이로써 운영체제는 인터럽트된 프로세스 문맥에서 전환 코드를 호출하고, 실행될 프로세스 문맥으로 리턴할 수 있다. 운영체제가 마지막으로 `return-from-trap` 명령어를 실행하면 곧 실행될 프로세스가 현재 실행 중인 프로세스가 된다. 그래서 문맥 교환이 마무리 된다.

이 모든 과정의 연대표가 그림 9.3에 나와 있다. 이 예에서 프로세스 A가 실행 중이고 타이머 인터럽트에 의해 중단된다. 하드웨어는 A의 레지스터를 커널 스택에 저장하고 커널 모드로 진입한다. 타이머 인터럽트 핸들러에서 운영체제는 프로세스 B로 전환하기로 결정한다. 이 시점에 `switch()` 루틴을 호출한다. 이 루틴이 A의 레지스터의 현재 값을 A의 프로세스 구조체에 저장하고 B의 프로세스 구조체에서 B의 레지스터를 복원한다. 그런 후에 A의 커널 스택이 아니라 B의 커널 스택을 사용하도록 스택 포인터를 바꾸어서 **문맥 교환**을 수행한다. 마지막으로 운영체제는 `return-from-trap`을 수행하여 B의

운영체제 @부트 (커널 모드)	**하드웨어**	
트랩 테이블을 초기화 한다		
	syscall 핸들러, 타이머 핸들러의 주소를 기억한다	
인터럽트 타이머를 시작시킨다		
	타이머를 시작시킨다 X msec 지난 후 CPU를 인터럽트한다	
운영체제 @실행 (커널 모드)	**하드웨어**	**프로그램 (사용자 모드)**
		프로세스 A …
	타이머 인터럽트 A의 레지스터를 A의 커널 스택에 저장 커널 모드로 이동 트랩 핸들러로 분기	
트랩을 처리한다 **switch()** 루틴 호출 A의 레지스터를 A의 proc 구조에 저장 B의 proc 구조로부터 B의 레지스터를 복원 B의 커널 스택으로 전환 **return-from-trap (B 프로세스로)**		
	B의 커널 스택을 B의 레지스터로 저장 사용자 모드로 이동 B의 PC로 분기	
		프로세스 B …

〈그림 9.3〉 **제한적 직접 실행 프로토콜(타이머 인터럽트)**

```
1  # void switch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl switch
6  switch:
7      # Save old registers
8      movl 4(%esp), %eax      # old포인터를 eax에 넣는다
9      popl 0(%eax)            # old IP를 저장한다
10     movl %esp, 4(%eax)      # 그리고 스택
11     movl %ebx, 8(%eax)      # 그리고 다른 레지스터
12     movl %ecx, 12(%eax)
13     movl %edx, 16(%eax)
14     movl %esi, 20(%eax)
15     movl %edi, 24(%eax)
16     movl %ebp, 28(%eax)
17
18     # Load new registers
19     movl 4(%esp), %eax      # new포인터를 eax에 넣는다
20     movl 28(%eax), %ebp     # 다른 레지스터를 복원한다
21     movl 24(%eax), %edi
22     movl 20(%eax), %esi
23     movl 16(%eax), %edx
24     movl 12(%eax), %ecx
25     movl 8(%eax), %ebx
26     movl 4(%eax), %esp      # 스택은 이 지점에서 전환된다
27     pushl 0(%eax)           # 리턴 주소를 지정된 장소에 넣는다
28     ret                     # 마지막으로 new문맥으로 리턴한다
```

〈그림 9.4〉 **xv6 문맥 교환 코드**

레지스터를 복원하고 실행을 시작한다.

이 과정이 실행되는 동안 두 번의 레지스터의 저장/복원이 일어난다는 것에 주의하자. 첫 번째는 타이머 인터럽트가 발생했을 때 일어난다. 이 경우 실행 중인 프로세스의 *사용자 레지스터가 하드웨어에* 의해 암묵적으로 저장되고 저장 장소로 해당 프로세스의 커널 스택이 사용된다. 두 번째는 운영체제가 A에서 B로 전환하기로 결정했을 때 일어난다. 이 경우 *커널 레지스터는 운영체제에* 의하여 해당 프로세스의 프로세스 구조체에 저장된다. 이것은 운영체제가 A가 아닌 B로부터 커널로 트랩된 것처럼 만든다.

이러한 문맥 교환이 어떻게 일어나는지 잘 이해할 수 있도록 xv6의 문맥 교환 코드를 그림 9.4에 보이고 있다. 우선 이해할 수 있는지 살펴보기 바란다. 물론 xv6와 x86에 관한 얼마간의 지식을 가지고 있어야 한다. `context` 구조체 `old`와 `new`는 구 프로세스와 새 프로세스의 프로세스 구조체 안에 위치해 있다.

9.4 병행성이 걱정

집중력과 사고력이 좋은 몇몇 친구는 지금 이런 생각을 하고 있을지 모른다. "음, 만약 시스템 콜을 처리하는 도중에 타이머 인터럽트가 발생하면 어떤 일이 발생하는가?" 또는 "하나의 인터럽트를 처리하고 있을 때 다른 인터럽트가 발생하면 어떤 일이 생기는가? 커널에서 처리하기 더 어려워지는 것은 아닌가?" 좋은 질문. 우리에겐 아직 희망이 있다!

대답은 "그렇다" 이다. 운영체제는 인터럽트 또는 트랩을 처리하는 도중에 다른 인터럽트가 발생할 때 어떤 일이 생기는가에 대해 신중하게 고려할 필요가 있다. 사실

여담: 문맥 교환하는 데 걸리는 시간은 얼마인가

떠올릴 수 있는 자연스러운 질문은 문맥 교환과 같은 작업의 처리 소요 시간이다. 혹은 시스템 콜의 처리 소요 시간은? 호기심 많은 당신을 위해 lmbench [MS96]라는 도구가 있다. 이 도구는 언급한 그러한 작업의 시간을 정확하게 측정할 뿐 아니라 연관된 다른 성능 수치도 측정한다.

결과는 시간이 지나면서 대략 프로세서의 성능개선 추이와 비슷하게 점점 좋아졌다. 예를 들어, 1996년에 200 MHz P6 CPU에서 실행되는 Linux 1.3.37의 경우 시스템 콜은 4마이크로초 그리고 문맥 교환에는 대략 6마이크로초가 소요되었다 [MS96]. 현대 시스템에서는 자릿수가 달라질 정도로 성능이 좋아져서 2 또는 3 GHz 프로세서의 경우 1마이크로초 미만이 소요된다.

운영체제의 모든 동작이 CPU 성능에 따라 좋아지는 것이 아니라는 것에 주의하자. Ousterhout가 발견한 것처럼 운영체제의 많은 연산은 주로 메모리를 접근하는 연산이며 메모리의 대역폭은 프로세서 속도가 발전하는 것만큼 극적으로 향상되지 않았기 때문이다 [Ous90]. 사용한 워크로드의 특성에 따라 최신의 좋은 프로세서를 구매하더라도 운영체제의 속도가 기대만큼 증가하지 않을 수 있다.

이 내용이 바로 이 책의 두 번째 부분에서 논의할 주제인 **병행성**에 관한 것으로서 자세한 논의는 그때까지 미룰 것이다.

흥미를 돋우기 위하여 운영체제가 이 다루기 힘든 상황을 어떻게 처리하는지 간략히 알아보자. 운영체제가 할 수 있는 간단한 해법은 인터럽트를 처리하는 동안 **인터럽트를 불능화**시키는 것이다. 이럴 경우 하나의 인터럽트가 처리되고 있는 동안 다른 어떤 인터럽트도 CPU에게 전달되지 않는다. 물론, 운영체제는 이 작업을 신중하게 해야 한다. 인터럽트를 너무 오랫동안 불능화시키면 인터럽트를 놓치게 되고 기술적으로도 좋지 않다.

운영체제는, 또한 내부 자료 구조에 동시에 접근하는 것을 방지하기 위해 많은 정교한 **락(lock)** 기법을 개발해 왔다. 이 잠금 기법은 커널 안에서 동시에 다수의 활동이 진행될 수 있게 허용한다. 이 책의 병행성에 관한 부분에서 살펴보겠지만 이런 잠금 기법은 복잡해질 수 있으며 그로 인해 흥미로우면서도 발견하기 힘든 버그를 만들어 낸다.

9.5 요약

우리는 CPU 가상화를 구현하기 위한 핵심적인 저수준 기법에 관해 설명하였다. 이런 기법들을 묶어서 **제한적 직접 실행**이라고 부른다. 기본적인 아이디어는 간단하다. CPU에서 실행하고 싶은 프로그램을 실행시킨다. 그러나 운영체제가 CPU를 사용하지 못하더라도 프로세스의 작업을 제한할 수 있도록 하드웨어를 셋업해야 한다.

이러한 일반적인 방법은 실생활에서도 사용된다. 예를 들어, 아이가 있거나 적어도 아이들에 관해 들어 보았다면 방에 **아기 보호 장치**를 설치하는 일에 친숙할 것이다. 위험한 물건이 들어 있는 캐비닛을 잠그고 전기 소켓의 두껑을 덮는다거나 하는 일이

팁: 리부팅은 유용하다

협조적 선점 모드에서 무한루프와 같은 행동의 유일한 해결책은 컴퓨터를 **리부팅**하는 것이다. 단순 무식한 방법이지만, 연구가들은 리부팅이나 소프트웨어를 다시 시작하는 방법이 강건한 시스템 구축에 있어 매우 유용하다는 것을 보였다 [Can+04].

구체적으로 리부트는 소프트웨어를 이미 알려진 상태 그리고 검증된 상태로 되돌리기 때문에 유용하다. 리부트는 오래된 자원 또는 제어를 벗어난 자원을 시스템에 반환한다. 이러한 자원은 반환하지 않으면 처리하기 곤란하다. 마지막으로 리부트는 자동화하기 쉽다. 이러한 이유들 때문에 시스템 관리 소프트웨어를 위한 대규모 클러스터 인터넷 서비스에서는 컴퓨터 일부를 리셋시키기 위하여 주기적으로 리부팅하는 것이 흔한 일이다. 이렇게 함으로써 앞서 언급한 모든 이점을 얻을 수 있다.

따라서 리부팅한다고 해서 그것이 단순무식한 방법은 아니다. 오히려 컴퓨터 시스템의 동작을 향상시키기 위하여 충분히 검증된 방법을 사용하고 있는 것이다. 잘했어!

포함된다. 이렇게 방이 준비되면 대부분의 위험한 요소들을 막아 놓았기 때문에 아기가 자유롭고 안전하게 돌아다니게 할 수 있다.

비슷한 방식으로 운영체제는 CPU에 안전 장치를 준비해 놓을 수 있다. 우선 부팅할 때 트랩 핸들러 함수를 셋업하고 인터럽트 타이머를 시작시키고 그런 후에 제한된 모드에서만 프로세스가 실행되도록 한다. 이로써 운영체제는 프로세스를 효율적으로 실행할 수 있고, 특별한 연산을 수행할 때, 즉 프로세스가 CPU를 독점하거나, 다른 프로세스로 전환해야 할 때만 개입한다.

이리하여 우리는 CPU 가상화를 위한 기본 개념을 학습하였다. 중요한 질문이 그대로 남았다. 특정 시점에 어떤 프로세스를 실행시켜야 하는가? 바로 스케줄러가 답해야 하는 질문이고 우리가 공부할 다음 주제이다.

참고 문헌

[Alt79] **"Alto User's Handbook"**
Xerox Palo Alto Research Center, September 1979
URL: `http://history-computer.com/Library/AltoUsersHandbook.pdf`
시대를 앞선 환상적인 시스템. Steve Jobs가 방문하고 기록하고 Lisa, 궁극적으로 Mac을 구현하였기 때문에 유명해졌다.

[Can+04] **"Microreboot---A Technique for Cheap Recovery"**
George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox
OSDI '04, San Francisco, CA, December 2004
강인한 시스템을 구축하는 데 리부트를 어디까지 활용할 수 있는지를 지적한 훌륭한 논문.

[Int11] **"Intel 64 and IA-32 Architectures Software Developer's Manual"**
Volume 3A and 3B: System Programming Guide, Intel Corporation, January 2011

[Kil+62] **"One-Level Storage System"**
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner
IRE Transactions on Electronic Computers, April 1962
Atlas는 현대 시스템에서 우리가 볼 수 있는 많은 것들을 개척하였다. 그러나 이 논문이 읽을 만한 가장 좋은 논문은 아니다. 한편의 논문을 읽어야 한다면 아래에 있는 역사적 관점의 논문을 읽는 것이 좋을 것이다 [Lav78].

[Lav78] **"The Manchester Mark I and Atlas: A Historical Perspective"**
S.H. Lavington
Communications of the ACM, 21:1, January 1978
초창기 컴퓨터 개발과 Atlas의 개척적인 노력에 관한 역사.

[Mac11] **"Mac OS 9"**
January 2011
URL: `http://en.wikipedia.org/wiki/Mac_OS_9`

[McC+63] **"A Time-Sharing Debugging System for a Small Computer"**
J. McCarthy, S. Boilen, E. Fredkin, and J.C.R. Licklider
AFIPS '63 (Spring), May, 1963, New York, USA
타이머 인터럽트 사용에 관해 언급한 시분할에 관한 초기 논문. 그 논의를 인용하면 "채널 17 클록 루틴의 기본 임무는 코어에서 현재 사용자를 제거할지 결정하는 것이고 제거하기로 결정한 경우 대신에 어느 사용자 프로그램이 스왑-인 될지 결정하는 것이다."

[MS96] **"lmbench: Portable tools for performance analysis"**
Larry McVoy and Carl Staelin
USENIX Annual Technical Conference, January 1996
운영체제와 그의 성능에 관해 여러 측면을 어떻게 측정하느냐에 관한 흥미로운 논문. lmbench를 다운로드하여 한 번 측정해 보라.

[Ous90] **"Why Aren't Operating Systems Getting Faster as Fast as Hardware?"**
J. Ousterhout
USENIX Summer Conference, June 1990
운영체제 성능의 본질에 관한 고전적인 논문.

[P10] **"The Single Unix Specification, Version 3"**
The Open Group, May 2010
URL: `http://www.unix.org/version3/`
읽기에 어렵고 힘든 문서. 가능하다면 읽지 말도록.

[Sha07] **"The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"**
Hovav Shacham
CCS '07, October 2007
연구 중에 가끔 만나게 되는 경이로우면서 놀라운 아이디어 중 하나. 저자는 임의의 위치로 분기할 수 있다면 주어진 커다란 코드 베이스에서 원하는 코드 시퀀스를 만들어 낼 수 있다는 것을 보인다. 자세한 사항은 논문을 읽도록. 안타깝게도 이 기법은 악의적인 공격에 대한 방어를 훨씬 어렵게 만든다.

숙제 (측정)

여담 : 측정 숙제

측정 숙제는 운영체제 또는 하드웨어 성능의 특정 측면을 측정하기 위해 실제 컴퓨터에서 실행시킬 코드를 작성하는 작은 연습이다. 이 숙제의 의도에는 운영체제를 직접 경험해 보자는 의미가 있다.

이 숙제에서 시스템 콜과 문맥 교환의 비용을 측정할 것이다. 시스템 콜의 비용을 측정하는 것은 상대적으로 쉽다. 예를 들어, 간단한 시스템 콜을(0 바이트 읽기 등) 반복적으로 호출하여 통틀어 걸린 시간을 측정한다. 이 시간을 반복 횟수로 나누면 시스템 콜의 비용을 추정할 수 있다.

한 가지 고려해야 할 사항은 타이머의 정확도와 정밀도이다. 사용할 수 있는 전형적인 타이머는 `gettimeofday()`이다. 자세한 사항에 대해서는 `man` 페이지를 참조하도록. `man` 페이지에서 볼 수 있는 정보는 `gettimeofday()`이 1970년 이후 경과한 시간을 마이크로초 단위로 반환한다는 것이다. 그러나 이 사실이 타이머가 마이크로초 수준의 정밀도를 가진다는 것을 의미하지는 않는다. 타이머가 얼마나 정밀한지를 알고 싶다면 `gettimeofday()`를 연속해서 호출하여 측정하라. 이 결과로부터 만족스러운 측정 결과를 얻기 위해서는 널 시스템 콜 테스트를 몇 번 반복해야 하는지 알 수 있을 것이다. `gettimeofday()`가 원하는 만큼 정밀도를 보이지 않는다면 x86 CPU에서 제공되는 `rdtsc` 명령어를 사용하는 것을 검토해 보아야 한다.

문맥 교환의 비용을 측정하는 것은 더 곤란하다. lmbench 벤치마크는 두 개의 프로세스를 하나의 CPU에서 실행시키고 둘 사이에 Unix 파이프를 설정하여 문맥 교환 비용을 측정한다. 파이프는 Unix 시스템에서 프로세스끼리 통신할 수 있는 여러 방법 중의 하나이다. 첫 번째 프로세스는 첫 번째 파이프에 데이터를 쓰고, 두 번째 파이프로부터 데이터가 전달되기를 기다린다. 첫 번째 프로세스가 두 번째 파이프로부터 읽을 무언가를 기다린다는 사실을 알게 되면 운영체제는 첫 번째 프로세스를 봉쇄 상태로 만들고 다른 프로세스로 전환한다. 이 프로세스는 첫 번째 파이프로부터 데이터를 읽고 두 번째 파이프에 데이터를 쓴다. 두 번째 프로세스가 첫 번째 파이프로부터 다시 데이터를 읽으려고 하면 봉쇄 상태로 들어간다. 이런 식으로 주고 받는 통신 사이클이 계속된다. 이러한 통신 비용을 반복적으로 측정하여 lmbench는 문맥 교환의 비용에 대해 훌륭한 추정치를 구할 수 있다. 이 숙제에서 파이프를 사용하거나 Unix 소켓과 같은 다른 통신 기법을 사용하여 유사한 상황을 만들어 볼 수 있다.

문맥 교환의 비용을 측정하는 데 있어 어려움은 하나 이상의 CPU를 가진 시스템에서 발생한다. 그런 시스템에서 해야만 하는 일은 문맥 교환되는 프로세스들이 같은 프로세서 상에 위치하는 것을 보장하는 것이다. 다행히도 대부분의 운영체제는 프로세스를 특정 프로세서에 묶어두는 시스템 콜을 가진다. 예를 들어, Linux에서는 `sched_setaffinity()` 시스템 콜이 기대하는 일을 한다. 두 프로세스를 같은 프로세

서에 존재하도록 보장하면 같은 프로세서에 존재하는 한 프로세스를 중단시키고 다른 프로세스를 복원하는 운영체제의 비용을 확실하게 측정할 수 있다.