39

I/O 장치

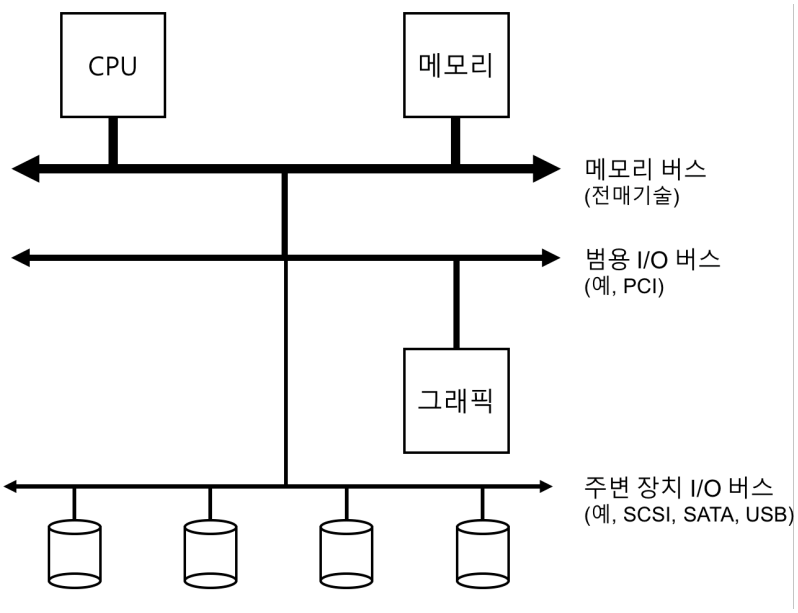
본론(영속성)에 들어가기에 앞서 먼저 **입력/출력 장치**의 개념을 소개하고 운영체제가 이 장치들과 상호 작용하는 방법을 알아보자. 당연하겠지만, I/O는 컴퓨터 시스템에서 상당히 중요한 부분이다. 입력이 전혀 없는 프로그램이나(늘 같은 결과를 출력하는), 출력이 없는 프로그램을 생각해 보라(실행이 어떤 의미가 있겠는가?). 컴퓨터 시스템을 유용하게 쓰려면 입력과 출력이 모두 필요하다는 것은 분명하다. 그러므로 우리가 해결해야 할 문제는 다음과 같다.

핵심 질문: 어떻게 I/O를 시스템에 통합할까
시스템에 I/O를 어떻게 통합해야 하는가? 일반적인 방법은 무엇인가? 어떻게 효율적으로 통합할 수 있을까?

39.1 시스템 구조

논의를 시작하기 위해, 그림 39.1의 일반적인 시스템 구조를 살펴보자. 이 그림에서는 CPU와 주메모리가 **메모리 버스**로 연결되어 있다. 몇 가지 장치들이 범용 **I/O 버스**에 연결이 되어 있는데, 많은 현대의 시스템에서는 **PCI**(또는 많은 파생 버스)를 사용하고 있다. 그래픽이나 다른 고성능 I/O 장치들이 여기에 연결될 수 있다. 마지막으로, 그 아래에는 **SCSI**나 **SATA** 또는 **USB**와 같은 **주변장치용 버스**가 있다. 이 버스들을 통해 **디스크**, **마우스**와 같은 가장 느린 장치들이 연결된다.

이런 계층적인 구조가 필요한 이유를 질문할 수 있는데, 간단하게 답하자면, 물리학적인 이유와 비용 때문이다. 버스가 고속화되려면 더 짧아져야 하는데, 그런 고속의 메모리 버스는 여러 장치들을 수용할 공간이 없다. 또한, 고속의 성능을 내는 버스를 만드는 기술은 꽤나 비싸다. 그렇기 때문에 시스템 설계자들은 계층적 구조를 택하여 (그래픽 카드와 같이) 고성능 장치들을 CPU에 가깝게 배치하였고 느린 성능의 장치들을 그보다 멀리 배치하였다. 디스크처럼 느린 장치들을 주변 장치 I/O 버스에 연결하여 얻을 수 있는 이득은 많다. 특히, 많은 장치들을 연결할 수 있다.

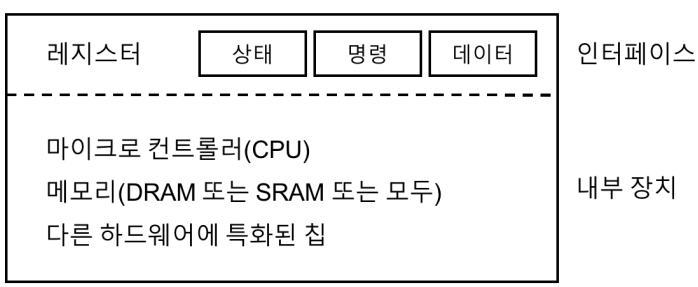


〈그림 39.1〉 **시스템 구조 모형**

39.2 표준 장치

이번에는 가상의 표준 장치를 살펴보고 이 장치를 효율적으로 활용하기 위해 필요한 것은 무엇인지를 알아보자. 그림 39.2에서 나타낸 것 같이 두 개의 중요한 요소가 있다. 첫 번째는 시스템의 다른 구성 요소에게 제공하는 하드웨어 **인터페이스**이다. 소프트웨어가 인터페이스를 제공하듯이 하드웨어도 인터페이스를 제공하여 시스템 소프트웨어가 동작을 제어할 수 있도록 해야 한다. 그렇기 때문에 모든 하드웨어 장치들은 특정한 상호 동작을 위한 방식과 명시적인 인터페이스를 갖고 있다.

장치가 갖고 있어야 하는 두 번째 요소는 **내부 구조**이다. 구현 방법에 따라 다르지만 시스템에게 제공하는 장치에 대한 추상화를 정의하는 책임을 갖고 있다. 매우 단순한 장치들은 하나 또는 몇 개의 하드웨어 칩을 사용하여 기능을 구현할 것이고 좀 더 복잡한 장치는 단순한 CPU와 범용 메모리 그리고 장치에 특화된 칩들을 사용하여 목적에



〈그림 39.2〉 **표준 장치**

맞는 동작을 한다. 최신 RAID 컨트롤러는 수십만 줄에 달하는 **펌웨어**(**firmware**)라는 소프트웨어가 하드웨어 내부의 동작을 정의하고 있다.

39.3 표준 방식

위에서 보인 그림에서 (단순화된) 장치의 인터페이스는 세 개의 레지스터로 구성되어 있는 것을 보았다. **상태**(**status**) 레지스터는 장치의 현재 상태를 읽을 수 있으며 **명령어**(**command**) 레지스터는 장치가 특정 동작을 수행하도록 요청할 때, 그리고 **데이터**(**data**) 레지스터는 장치에 데이터를 보내거나 받거나 할 때 사용한다. 이 레지스터들을 읽거나 쓰는 것을 통해 운영체제는 장치의 동작을 제어할 수 있다.

이번에는 장치가 운영체제를 대신하여 특정 동작을 할 때에 운영체제와 장치 간에 일어날 수 있는 상호 동작의 과정을 살펴보자. 이 경우 다음과 같은 방식을 따른다.

```
While (STATUS == BUSY)
    ; // 장치가 바쁜 상태가 아닐 때까지 대기
데이터를 DATA 레지스터에 쓰기
명령어를 COMMAND 레지스터에 쓰기
    (그러면 장치가 명령어를 실행한다)
While (STATUS == BUSY)
    ; // 요청을 처리하여 완료할 때까지 대기
```

방식은 네 단계로 이루어져 있다. 먼저 반복적으로 장치의 상태 레지스터를 읽어서 명령의 수신 가능 여부를 확인한다. 이 동작을 장치에 대해 **폴링**(**polling**)한다고 표현한다(기본적으로 어떤 작업을 하고 있는지 묻는 것과 같다). 두 번째는 운영체제가 데이터 레지스터에 어떤 데이터를 전달한다. 예를 들어 장치가 디스크였다면 디스크 블럭(4 KB로 가정)에 해당하는 데이터를 전달할 것이고 여러 번의 쓰기를 수행할 것이다. 데이터 전송에 메인 CPU가 관여하는 경우를 (이 예제의 방식처럼) **programmed I/O**라고 부른다. 세 번째로 운영체제가 명령 레지스터에 명령어를 기록한다. 이 레지스터에 명령어가 기록되면 데이터는 이미 준비되었다고 판단하고 명령어를 처리한다. 마지막으로 운영체제는 디바이스가 처리를 완료했는지를 확인하는 폴링 반복문을 돌면서 기다린다 (성공과 실패를 알리는 에러 코드를 받게 된다).

기본 방식은 방식이 간단하며 제대로 작동한다. 하지만, 매우 비효율적이다. 이 방식이 갖고 있는 첫 번째 문제는 매우 비효율적인 폴링을 사용하고 있다는 점이다. 다른 프로세스에게 CPU를 양도하지 않고, 장치가 동작으로 완료하는 동안 계속 루프를 돌면서 장치상태를 체크하고 있다. 이를 폴링이라 한다. 입출력 장치는 무척 느리다. 그리고 대기하는 중에 특별히 따로 하는 일이 있는 것도 아니다. CPU 시간을 많이 소모하게 된다.

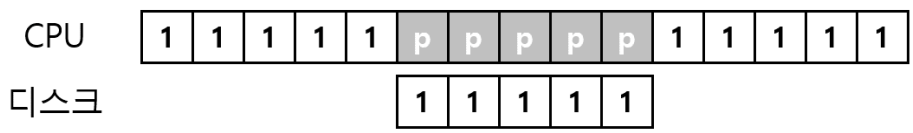
핵심 질문: 폴링 사용 비용을 어떻게 피하는가

어떻게 하면 자주 폴링을 하지 않으면서 운영체제가 장치의 상태를 확인할 수 있고, 장치를 관리하는 CPU의 오버헤드를 줄일 수 있을까?

39.4 인터럽트를 이용한 CPU 오버헤드 개선

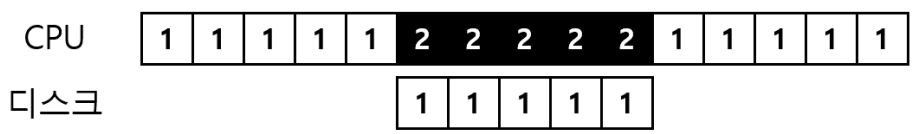
이미 수년 전에 엔지니어들이 장치와의 상호작용을 개선하기 위해 **인터럽트**라는 것을 개발하였다. 인터럽트는 이미 다루었다. 디바이스를 폴링하는 대신 운영체제는 입출력 작업을 요청한 프로세스를 블록 시키고 CPU를 다른 프로세스에게 양도한다. 장치가 작업을 끝마치고 나면 하드웨어 인터럽트를 발생시키고 CPU는 운영체제가 미리 정의 해 놓은 **인터럽트 서비스 루틴(interrupt service routine(ISR))** 또는 간단하게 **인터럽트 핸들러(interrupt handler)**를 실행한다. 이 핸들러는 운영체제 코드의 일부이다. 인터럽트 핸들러는 입출력 요청의 완료(예를 들어, 데이터와 장치가 전달하는 에러 코드를 읽는 것 등), I/O를 대기 중인 프로세스 깨우기 등을 담당한다. 깨어난 프로세스가 작업을 계속할 수 있도록 한다.

사용률을 높이기 위한 핵심 방법 중 하나는 인터럽트를 활용하여 CPU 연산과 I/O를 **중첩**시키는 것이다. 다음의 시간 흐름표가 이 방법을 나타낸다.



이 도표에서는 CPU에서 프로세스 1이 일정 시간 동안 실행되다가(CPU 줄에 반복된 1로 표현) 디스크의 데이터를 읽기 위해 I/O 요청을 발생시킨다. 인터럽트가 없다면 시스템은 I/O가 완료될 때까지 반복적으로 장치의 상태를 폴링한다(p로 표현). 디스크가 요청의 처리를 완료하면 다시 프로세스 1이 동작할 수 있다.

그와 다르게 인터럽트를 사용하여 연산과 I/O 작업을 중첩시킬 수 있다면 운영체제는 디스크의 응답을 기다리면서 다른 일을 할 수 있다.



이 예제에서는 프로세스 1의 요청이 디스크에서 처리되는 동안에 운영체제는 프로세스 2를 CPU에서 실행시킨다. 디스크 요청이 완료되어 인터럽트가 발생하면 운영체제가 프로세스 1을 깨워 다시 실행시킨다. 그러므로 *CPU*와 *디스크*가 시간 흐름표의 중간 부분에서 적절히 활용되었다.

인터럽트가 *항상* 최적의 해법은 아니란 것에 유의해야 한다. 예를 들어 대부분 작업이 한 번의 폴링만으로 끝날 정도로 매우 빠른 장치라고 해 보자. 이 경우에 인터럽트를 사용하면 시스템이 *느려지게* 된다. 다른 프로세스로 문맥을 교환하고 인터럽트를 처리한 후 다시 I/O를 요청한 프로세스로 문맥 교환하는 것은 매우 비싼 작업이다. 그러므로 빠른 장치라면 폴링이 최선이고 느리다면 인터럽트를 사용하여 중첩시키는 것이 최선이다.

팁: 인터럽트가 PIO보다 항상 좋은 것은 아니다

인터럽트을 사용하면 연산과 I/O 작업을 중첩시킬 수 있지만 느린 장치에 대해서만 타당한 접근법이다. 그 외의 경우에는 인터럽트 처리와 문맥 교환 비용이 인터럽트가 제공하는 장점을 넘어서게 된다. 너무 많은 인터럽트로 인해서 시스템이 멈춘 것처럼 보이게 되는 경우가 있다 [MR96]. 그러한 경우에 운영체제가 원하는 대로 스케줄링 할 수 있게 하는 폴링이 역시 더 유용하다.

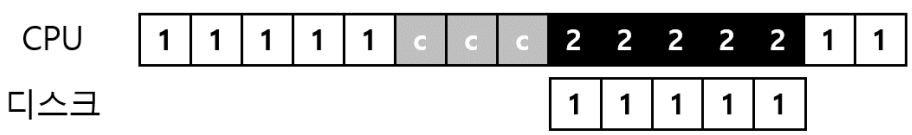
만약 장치의 속도를 모른다거나 빠를 때도 있고 느릴 때도 있는 장치라고 한다면, 짧은 시간 동안만 폴링을 하다가 처리가 완료되지 않으면 인터럽트를 사용하는 **하이브리드** 방식을 채용하는 것이 최선일 것이다. 이와 같은 **두 단계** 접근법으로 양쪽의 장점만 취할 수 있다.

인터럽트를 사용하지 않는 다른 이유는 네트워크 환경에서 찾아볼 수 있다 [MR96]. 대량으로 도착하는 패킷이 있을 때에 각 패킷마다 인터럽트가 발생된다. 이 경우 인터럽트만 처리하다가 운영체제가 사용자 프로세스의 요청을 처리할 수 없도록 만드는 **무한반복(livelock)**에 빠질 가능성이 있다. 예를 들어 "슬래시닷(slashdot) 현상"으로 인해 일시적으로 웹 서버에 사용자가 많이 몰리는 경우가 발생했다고 가정하자. 이 경우, 폴링을 사용하면 시스템의 상황을 보다 효율적으로 제어할 수 있다. 웹 서버가 패킷 도착을 검사하기 전에 사용자 요청들을 좀 더 처리할 수 있기 때문이다.

또 다른 인터럽트 기반의 최적화 기법은 **병합(coalescing)**이다. 이 환경에서는 CPU에 인터럽트를 전달하기 전에 잠시 기다렸다가 인터럽트를 발생시킨다. 기다리는 동안에 다른 요청들도 끝나게 되기 때문에 여러 번 인터럽트를 발생시키는 대신 인터럽트를 한 번만 CPU에 전달하게 된다. 이 방법으로 인터럽트 처리의 오버헤드를 줄일 수 있다. 물론, 너무 오래 기다리면 요청에 대한 지연 시간이 늘어나기 때문에 시스템의 절충 시간을 찾아야 한다. 이 문제에 대해서는 Ahmad 등 [AGM11]이 훌륭하게 정리한 글을 참고 바란다.

39.5 DMA를 이용한 효율적인 데이터 이동

앞서 설명한 표준 방식에서 고려해야 하는 부분이 하나 더 있다. 많은 양의 데이터를 디스크로 전달하기 위해 programmed I/O(PIO)를 사용하면 CPU가 또 다시 단순 작업 처리에 소모된다. 다른 프로세스를 처리하기 위해 사용될 수 있는 많은 시간을 허비하게 된다. 다음의 시간 흐름표가 그 문제를 나타낸다.



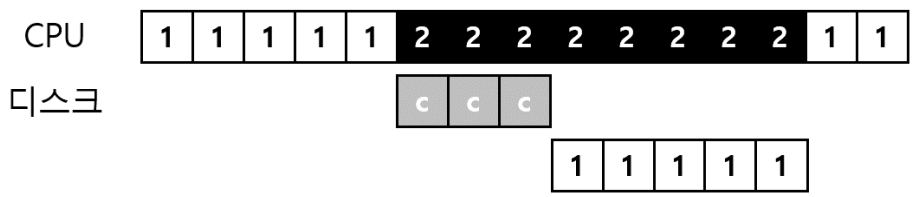
이 시간 흐름표에서는 프로세스 1이 실행 도중에 디스크에 어떤 데이터를 기록하려고 한다. 그래서 I/O를 발생시켜서 명시적으로 데이터를 메모리에서 디스크로 한 워드씩 복사한다(그림에서 c로 표기). 복사가 완료되면 디스크에서 I/O의 처리를 시작하고, 마침내 CPU를 다른 작업 처리에 사용할 수 있게 된다.

핵심 질문: 어떻게 PIO의 오버헤드를 줄이는가

PIO를 사용하면 CPU는 너무 많은 시간을 데이터를 디스크에서 또는 디스크로 이동하는 데 사용한다. 이 작업을 어떻게 줄일 수 있으며 CPU를 더 효율적으로 활용하는 방법은 무엇인가?

이 문제에 대한 해법을 **직접 메모리 접근 방식**(**Direct Memory Access**, **DMA**)이라고 부른다. DMA 엔진은 시스템 내에 있는 특수 장치로서 CPU의 간섭없이 메모리와 장치 간에 전송을 담당한다.

DMA는 다음과 같이 동작한다. 데이터를 장치로 전송한다고 했을 때 운영체제는 DMA 엔진에 메모리 상의 데이터 위치와 전송할 데이터의 크기와 대상 장치를 프로그램한다. 그 시점에 전송하기 위해 할 일은 끝나기 때문에 운영체제는 다른 일을 진행할 수 있다. DMA 동작이 끝나면 DMA 컨트롤러가 인터럽트를 발생시켜 운영체제가 전송이 완료되었음을 알 수 있도록 한다. 수정된 시간 흐름표는 다음과 같다.



이 시간 흐름표에서 데이터의 복사는 DMA 컨트롤러가 처리하고 있는 것을 알 수 있다. CPU가 그 시간 동안 자유롭기 때문에 운영체제는 다른 작업을 처리할 수 있고, 여기서는 프로세스 2를 선택하여 실행하였다. 프로세스 1이 다시 실행하기 전에 프로세스 2가 더 많은 CPU를 사용할 수 있다.

39.6 디바이스와 상호작용하는 방법

I/O 요청의 서비스에서 효율성의 중요함을 어느 정도 파악했을 것이다. 효율성외에 고려해야 할 다른 문제들이 있다. 그 중 하나는 이미 예상했을 수도 있다. 장치와 운영체제가 실제로 어떻게 정보를 서로 교환하는지를 아직 다루지 않았다! 그것이 문제이다.

이제까지 장치와 통신하는 두 가지 기본적인 방법이 개발되었다. 첫 번째는 가장 오래된 방법으로(수년 동안 IBM 메인프레임에서 사용되었다) **I/O 명령**을 명시적으로 사용하는 것이다. 이 명령어들은 운영체제가 특정 장치 레지스터에 데이터를 전송할 수 있는 방법을 제공하며, 앞서 언급한 방식을 구현할 수 있게 한다.

핵심 질문 : 어떻게 장치와 통신하는가
하드웨어가 장치와 통신하는 방법은 무엇인가? 명시적인 명령어들이 있는가? 아니면 다른 방법이 있는가?

예를 들어 x86의 경우 `in`과 `out` 명령어를 사용하여 장치들과 통신할 수 있다. 데이터를 장치에 보내야 하는 경우에는 호출자가 데이터가 저장된 레지스터를 지정하고 장치를 지칭하는 포트를 지정한다. 명령어를 실행하면 원하는 동작이 실행된다.

이 명령어들은 대부분 **특권(privileged)** 명령어(previledge instruction)들이다. 운영체제가 장치를 제어하는 역할을 한다. 때문에 운영체제만이 장치들과 직접 통신할 수 있다. 만약 어떤 프로그램이 디스크를 읽고 쓸 수 있다고 가정해 보자. 어떤 사용자 프로그램이 그 허점을 이용하여 기계의 제어권을 갖게 된다면 일대혼란(늘 그렇듯이)이 발생하게 될 것이다.

장치와 상호작용하는 두 번째 방법은 **memory mapped I/O(memory mapped I/O)**를 사용하는 것이다. 이 접근법에서 하드웨어는 장치의 레지스터들이 마치 메모리 상에 존재하는 것처럼 만든다. 특정 레지스터를 접근하기 위해서 운영체제는 해당 주소에 load(읽기) 또는 store(쓰기)를 하면 된다. 하드웨어는 load/store 명령어가 주 메모리를 향하는 대신 장치로 연결되도록 한다.

두 방법 모두 대단히 큰 장점이 있는 것은 아니다. 메모리 맵 방식의 경우 새로운 명령어가 필요 없기 때문에 좋기는 하지만, 두 방법 모두 현재도 쓰이고 있다.

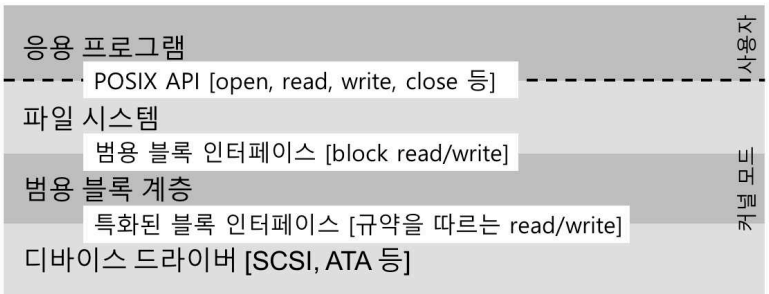
39.7 운영체제에 연결하기 : 디바이스 드라이버

최종적으로 다룰 문제는 서로 다른 인터페이스를 갖는 장치들과 운영체제를 연결시키는 가능한 일반적인 방법을 찾는 것이다. 파일 시스템을 예로 들어보자. SCSI 디스크와 IDE 디스크, USB 이동식 드라이브 등과 같은 기기 위에서 동작하는 파일 시스템을 만들고자 한다. 하지만 파일 시스템이 각 장치들의 구체적인 입출력 명령어 형식에 종속되지 않도록 만들고 싶다. 우리의 문제는 다음과 같다.

핵심 질문 : 어떻게 장치 중립적인 운영체제를 만드는가
어떻게 하면 운영체제를 장치 중립적으로 만들고, 장치와의 상호작용을 위한 상세 내용을 운영체제로부터 숨길 수 있을까?

추상화(abstraction)라는 고전적 방법을 사용하여 이 문제를 해결할 수 있다. 운영체제 최하위 계층의 일부 소프트웨어는 장치의 동작 방식을 반드시 알고 있어야 한다. 이 소프트웨어를 우리는 **디바이스 드라이버(device driver)**라고 부르며 장치와의 상세한 상호작용은 그 안에 캡슐화되어 있다.

Linux의 파일 시스템 소프트웨어 계층을 살펴보면서 이 추상화가 어떻게 운영체제의 디자인과 구현을 지원하는지 알아보자. 그림 39.3은 대략적인 Linux 소프트웨어 구조에 대한 그림이다. 그림에서 볼 수 있듯이 파일 시스템(그리고 그 위의 응용 프로그램을 포함해서)은 어떤 디스크 종류를 사용하는지 전혀 알지 못한다. 파일 시스템은 범용 블럭 계층(generic block layer)에 블럭 `read/write` 요청할 뿐이다. 범용 블럭 계층은 적절한 디바이스 드라이버로 받은 요청을 전달하며, 디바이스 드라이버는 특정 요청을 장치에 내리기 위해 필요한 일들을 처리한다. 이 그림은 단순하기는 하지만 장치에 대한 구체적인 동작이 운영체제의 대부분에게 숨겨질 수 있는 이유를 설명한다.



〈그림 39.3〉 **파일 시스템 스택**

이러한 캡슐화는 단점도 있다는 것에 유의해야 한다. 예를 들어 특수 기능을 많이 갖고 있는 어떤 장치가 있다고 했을 때, 커널이 범용적인 인터페이스만을 제공할 수밖에 없다면 그 많은 특수 기능들은 사용할 수 없게 된다. 예를 들어, 이와 같은 상황은 에러 보고 체계가 아주 잘되어 있는 Linux의 SCSI 장치들에서 볼 수 있다. 다른 블럭 장치들(예, ATA/IDE)은 단순한 에러 처리만을 지원하기 때문에 상위의 소프트웨어 계층에서 받을 수 있는 정보로 범용 EIO(generic IO error) 에러 코드 이상은 받을 수가 없다. SCSI가 제공하는 추가정보는 파일 시스템에 전달하지 못하고 사라지게 된다 [Gun+08].

흥미로운 것은 어떤 장치를 시스템에 연결하든 디바이스 드라이버가 필요하기 때문에 시간이 흐름에 따라 디바이스 드라이버 코드가 커널 코드의 대부분을 차지하게 되었다. Linux 커널에 대한 연구에 따르면 70% 이상의 운영체제 코드가 디바이스 드라이버를 위한 코드라고 한다 [Cho+01]. Windows 기반의 시스템의 경우도 역시 상당히 많은 부분을 차지할 것이다. 그러므로 누군가 운영체제가 수백만 줄의 코드로 이루어졌다고 하면 그들이 실제로 이야기하는 것은 운영체제가 수백만 줄의 디바이스 드라이버 코드를 포함하고 있다는 것이다. 물론, 특정 설치 환경에서 대부분의 코드는 동작을 하지 않을 것이다(즉, 몇 개의 장치들만이 시스템에 연결되어 있다는 말이다). 좀 더 우울하게 만드는 것은 드라이버들이 대부분 "아마추어"들에 의해서 만들어지다 보니(전업 커널 개발자가 아니라) 상당한 버그를 포함하고 있으며 커널 크래시의 주범이다 [SBL03].

39.8 사례 연구: 간단한 IDE 디스크 드라이버

좀 더 깊게 이해하기 위해서 실제 장치인 IDE 디스크 드라이브를 살펴보자 [Law94]. 참고 문헌에서 설명하고 있는 방식을 정리한다 [W10]. 실제 동작하는 IDE 드라이버인 xv6의 간단한 예를 위한 소스 코드도 살펴볼 것이다 [Cox+08].

IDE 디스크는 시스템에 다음과 같은 4개의 레지스터로 이루어진 단순한 인터페이스를 제공한다. 그 레지스터들은 Control, Command block, Status, 그리고 Error 이다. 이 레지스터들은 (x86의) `in`과 `out` I/O 명령어를 사용하여 특정 "I/O 주소들"(그림 39.4의 `0x3F6`와 같은)을 읽거나 씀으로써 접근 가능하다.

```
Control Register:
   Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
   Address 0x1F0 = Data Port
   Address 0x1F1 = Error
   Address 0x1F2 = Sector Count
   Address 0x1F3 = LBA low byte
   Address 0x1F4 = LBA mid byte
   Address 0x1F5 = LBA hi byte
   Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
   Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7       6       5     4      3      2       1        0
   BUSY   READY   FAULT  SEEK   DRQ    CORR   IDDEX    ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6      5       4      3      2      1        0
   BBK    UNC    MC      IDNF   MCR    ABRT   T0NF     AMNF

    BBK   =  Bad Block
    UNC   =  Uncorrectable data error
    MC    =  Media Changed
    IDNF  =  ID mark Not Found
    MCR   =  Media Change Requested
    ABRT  =  Command aborted
    T0NF  =  Track 0 Not Found
    AMNF  =  Address Mark Not Found
```

〈그림 39.4〉 **IDE 인터페이스**

장치와 상호작용하기 위한 기본 방식은 다음과 같으며 장치는 이미 초기화되었다고 가정한다.

- **장치가 준비될 때까지 대기.** 드라이브가 사용 중이지 않고 READY 상태가 될 때까지 Status 레지스터(0x1F7)를 읽는다.
- **Command 레지스터에 인자 값 쓰기.** 섹터의 수와 접근해야 할 섹터들의 논리 블럭 주소(LBA), 그리고 드라이브 번호(IDE는 두 개의 드라이브만 지원하기 때문에 마스터=0x00 또는 슬레이브=0x10)를 Command 레지스터(0x1F2-0x1F6)에 기록한다.
- **I/O 시작.** Command 레지스터에 읽기/쓰기를 전달한다. READ-WRITE 명령어를 Command 레지스터에 기록한다(0x1F7).

- **(쓰기의 경우) 데이터 전송.** 드라이브의 상태가 READY이고 DRQ(drive request for data, 데이터를 위한 드라이브 요청)일 때까지 기다린다. 데이터 포트에 데이터를 기록한다.
- **인터럽트 처리.** 가장 간단하게는 각 섹터가 전송되었을 때마다 인터럽트를 처리하게 하고 좀 더 복잡한 방법은 일괄처리가 가능하도록 만들어서 모든 전송이 완료되었을 때 최종적으로 한 번만 인터럽트를 발생시키도록 한다.
- **에러 처리.** 각 동작 이후에 Status 레지스터를 읽는다. 만약 ERROR 비트가 설정되어 있다면 Error 레지스터를 읽어서 상세 정보를 확인한다.

대부분의 프로토콜은 xv6 IDE 드라이버에 나타나 있으며(그림 39.5), (초기화 이후에) 네 개의 기본 함수를 통하여 동작한다. 첫 번째는 `ide_rw()`로서 (대기 중인 다른 요청들이 있다면)요청을 큐에 삽입하거나 (`ide_start_request()`를 통해) 디스크에 직접 명령한다. 어느 경우건 요청이 처리 완료되기를 기다리며 호출한 프로세스는 재운다. 두 번째는 `ide_start_request()`로서, 요청(쓰기의 경우에는 데이터도 함께)을 디스크로 내려 보낸다. x86의 in과 out 명령어가 장치 레지스터를 읽거나 쓰는 데 각각 사용된다. 시작 요청 루틴은 세 번째 함수인 `ide_wait_ready()`를 사용하여 요청을 명령하기 전에 드라이브가 준비가 되었는지 확인한다. 마지막으로 `ide_intr()`는 인터럽트가 발생하였을 때 호출된다. 장치에서 데이터를 읽고(쓰기가 아닌 읽기 요청인 경우) I/O가 종료되기를 기다리는 프로세스를 깨운다. 그리고 (더 많은 요청이 I/O 큐에 있다면) `ide_start_request()`를 이용하여 다음 요청 처리를 시작한다.

39.9 역사상의 기록

이 장을 마치기 전에 이제까지 논의한 근본 개념의 역사적인 유래에 대해 잠깐 살펴보자. 만약 더 자세하게 알고 싶다면 Smotherman이 훌륭하게 정리한 글을 읽어보자 [S08].

인터럽트는 아주 오래된 개념으로 초창기의 기기들에서도 존재하였다. 예를 들어 1950년대의 UNIVAC은 정확히 몇 년도에 채용했는지 분명하지는 않지만 인터럽트 벡터 형태를 갖고 있었다 [S08]. 슬프게도, 컴퓨터 발전이 아직 초창기임에도 불구하고 역사에 대한 많은 기원을 잃어버리고 있다.

DMA를 어느 기계가 가장 먼저 도입했는지에 대해 논쟁이 있었다. Knuth가 일부는 DYSEAC("이동식" 기계로 그 당시에는 트레일러에 실어 겨우 움직일 수 있다는 의미로 사용됨)라고 하고 또 다른 일부에서는 IBM SAGE가 시초라고 하기도 한다 [S08]. 여하간에 1950년대 중반까지는 메모리와 직접 통신하고, 전송 작업이 완료되었을 때 CPU에게 인터럽트를 전달하는 입출력 장치를 가진 시스템이 존재하였다.

역사를 따라 추적하기 어려운 이유는 여러 발명들이 실존했지만 잘 알려지지 않은 기계에 적용된 것이기 때문이다. 예를 들어 Lincoln Lab의 TX-2가 벡터 기반의 인터럽트를 처음 소개했다 [S08]지만 분명한 것은 아니다.

개념들이 대체적으로 당연한 것들이지만—느린 I/O를 기다리면서 CPU가 다른 일을 하도록 하는 것이 아인슈타인의 상대성 이론 같은 도약이 필요한 것이 아니지만—

```
1  static int ide_wait_ready() {
2      while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
3          ; // 드라이브가 바쁘지 않을 때까지 반복문 수행
4  }
5
6  static void ide_start_request(struct buf *b) {
7      ide_wait_ready();
8      outb(0x3f6, 0); // 인터럽트 발생
9      outb(0x1f2, 1); // 섹터는 몇 개?
10     outb(0x1f3, b->sector & 0xff);          // 여기에 LBA기록 ...
11     outb(0x1f4, (b->sector >> 8) & 0xff);   // ... 여기도
12     outb(0x1f5, (b->sector >> 16) & 0xff); // ... 여기도!
13     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
14     if(b->flags & B_DIRTY){
15         outb(0x1f7, IDE_CMD_WRITE); // 이것이 WRITE 명령어
16         outsl(0x1f0, b->data, 512/4); // 데이터도 전송!
17     } else {
18         outb(0x1f7, IDE_CMD_READ); // 이것이 READ (데이터 없음)
19     }
20 }
21
22 void ide_rw(struct buf *b) {
23     acquire(&ide_lock);
24     for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
25         ; // 큐를 순회
26     *pp = b; // 요청을 맨 뒤에 추가
27     if (ide_queue == b)          // q 가 비었다면
28         ide_start_request(b); // 디스크에 req를 보냄
29     while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
30         sleep(b, &ide_lock);  // 완료를 대기
31    release(&ide_lock);
32 }
33
34 void ide_intr() {
35     struct buf *b;
36     acquire(&ide_lock);
37     if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
38         insl(0x1f0, b->data, 512/4); // READ 라면: 데이터를 가져오기
39     b->flags |= B_VALID;
40     b->flags &= ~B_DIRTY;
41     wakeup(b); // 대기 중인 프로세스를 깨우기
42     if ((ide_queue = b->qnext) != 0)  // 다음의 요청을 시작
43         ide_start_request(ide_queue); // (존재한다면)
44     release(&ide_lock);
45 }
```

〈그림 39.5〉 **(단순화 한) xv6 IDE 디스크 드라이버**

여기서의 쟁점이 된 "누가 먼저?"는 잘못된 주제로 보인다. 분명한 것은 사람들이 이 초기 기계를 만들 때부터 I/O 지원이 필요해졌다는 것이다. 인터럽트와 DMA 그리고 관련된 개념들은 빠른 CPU와 느린 장치들의 특성을 활용한 결과물이다. 만약 당신도 그 시대에 살았다면 그러한 비슷한 개념을 생각했을 것이다.

39.10 요약

이제는 운영체제가 장치들과 어떻게 상호작용하는지에 대한 아주 기본적인 내용은 이해했을 것이다. 두 가지 기술인 인터럽트와 DMA는 장치의 효율을 높이기 위해 도입되었으며 명시적 I/O 명령어와 메모리 맵 I/O를 사용하여 장치의 레지스터에 접근할 수 있다는 것을 설명하였다. 마지막으로 디바이스 드라이버의 개념을 소개하면서 하위 계층의 세부적인 내용을 운영체제가 캡슐화 할 수 있으며, 이를 활용하여 운영체제의 나머지를 장치 중립적으로 구현할 수 있다는 것을 보였다.

참고 문헌

[AGM11] **"vIC: Interrupt Coalescing for Virtual Machine Storage Device IO"**
Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh
USENIX '11
전통적인 그리고 가상화 환경에서 인터럽트 병합에 대한 아주 멋진 조사.

[Cho+01] **"An Empirical Study of Operating System Errors"**
Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler
SOSP '01
최신의 운영체제에 버그가 얼마나 있는지를 체계적으로 탐색한 최초의 논문 중 하나이다. 근사한 발견 중 하나는 디바이스 드라이버 코드가 메인 라인 커널 코드보다 약 7배 더 많은 버그를 갖고 있다는 것이다.

[Cox+08] **"The xv6 Operating System"**
Russ Cox, Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
URL: `http://pdos.csail.mit.edu/6.828/2008/index.html`
IDE 디바이스 드라이버를 살펴보려면 `ide.c`를 보자. 좀 더 자세한 내용이 있다.

[Dre07] **"What Every Programmer Should Know About Memory"**
Ulrich Drepper
November, 2007
URL: `http://www.akkadia.org/drepper/cpumemory.pdf`
DRAM으로부터 시작하여 가상화와 캐시에 최적화된 알고리즘까지 다룬 현대의 메모리 시스템에 대한 환상적인 글이다.

[Gun+08] **"EIO: Error-handling is Occasionally Correct"**
Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Ben Liblit
FAST '08, San Jose, CA, February 2008
에러 리턴을 제대로 처리하지 않는 Linux 파일 시스템의 코드를 찾는 도구에 관한 우리의 연구이다. 우리는 수만 개의 버그를 발견하였고 그 중 대부분은 수정되었다.

[Law94] **"AT Attachment Interface for Disk Drives"**
Lawrence J. Lamers and X3T10 Technical Editor
Reference number: ANSI X3.221-1994
URL: `ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf`
디바이스 인터페이스에 대한 상당히 단조로운 문서이다. 위험을 각오하고 읽어보라.

[MR96] **"Eliminating Receive Livelock in an Interrupt-driven Kernel"**
Jeffrey Mogul and K.K. Ramakrishnan
USENIX '96, San Diego, CA, January 1996
웹 서버의 네트워크 성능 분야에 상당히 많은 선구적인 일을 Mogul과 그의 동료들이 하였다. 이 논문은 그 많은 노력의 결과 중에 하나이다.

[S08] **"Interrupts"**
Mark Smotherman, *as of July '08*
URL: `http://people.cs.clemson.edu/~mark/interrupts.html`
인터럽트와 DMA와 그리고 그에 연관된 컴퓨터의 초기 개념들에 대한 역사에 대한 정보로 숨겨진 보물과 같다.

[SBL03] **"Improving the Reliability of Commodity Operating Systems"**
Michael M. Swift, Brian N. Bershad, and Henry M. Levy
SOSP '03
Swift의 연구는 마이크로커널 류의 운영체제 접근법에 대한 관심을 불러 일으켰다. 최소한, 주소 공간 기반의 보호가 현대의 운영체제에서 유용한 이유에 대한 타당한 이유를 제시하였다.

[W10] **"Hard Disk Driver"**
Washington State Course Homepage
URL: `http://eecs.wsu.edu/.cs460/cs560/HDdriver.html`
간단한 IDE 디스크 드라이브 인터페이스와 IDE를 위한 디바이스 드라이버를 만드는 법을 잘 정리하였다.