36

이벤트 기반의 병행성(고급)

이제까지는 쓰레드를 병행 프로그램을 제작하는 유일한 도구인 것처럼 언급했다. 인생의 많은 일들이 대부분 그렇듯, 운영체제에서도 방법이 하나만 있는 경우는 거의 없다. 특히, GUI 기반 프로그램이나 [Ous96] 인터넷 서버 [PDZ99]에서는 다른 스타일의 병행 프로그래밍이 사용된다. 이런 스타일을 **이벤트 기반의 병행성**(**event-based concurrency**)이라 한다. **node.js** [N1313]와 같은 서버 프레임워크에서 사용되지만, 그 시작점은 지금부터 다룰 C와 유닉스 시스템이다.

이벤트 기반의 병행성은 두 개의 문제를 갖고 있다. 먼저 멀티 쓰레드 프로그램에서 이벤트 기반 병행성을 올바르게 사용하는 것이 매우 어렵다. 이미 다루었듯이 락을 누락시키거나, 교착 상태 또는 다른 골치 아픈 문제들이 발생할 수 있기 때문이다. 또 다른 문제는 멀티 쓰레드 프로그램에서는 개발자가 쓰래드 스케줄링에 대한 제어권을 전혀 갖고 있지 않다는 것이다. 개발자는 운영체제가 생성된 쓰레드를 CPU들 간에 합리적으로 스케줄링하기만을 기대할 수밖에 없다. 모든 종류의 워크로드에 잘 동작하는 범용 스케줄러를 만드는 것은 어려운 일이기 때문에 운영체제가 때로는 원치않는 순서로 스케줄링하는 경우가 발생할 수 있다. 핵심 질문은 다음과 같다.

핵심 질문: 어떻게 쓰레드 없이 병행 서버를 개발할까
쓰레드 없이 병행 서버를 구현할 때, 어떻게 병행성을 유지하면서 각종 문제들을 피할 수 있을까?

36.1 기본 개념: 이벤트 루프

우리가 다룰 방법은 앞서 언급한 **이벤트 기반의 병행성**이다. 이 접근 방법은 단순하다. 특정 사건(즉, "이벤트")의 발생을 대기한다. 사건이 발생하면, 사건(즉, 이벤트)의 종류를 파악한 후, I/O을 요청하거나 추후 처리를 위하여 다른 이벤트를 발생시키거나 하는 등의 작업을 한다. 그게 전부다!

자세한 설명 전에 고전적인 이벤트 기반의 서버가 어떻게 생겼는지 살펴보자. 이 응용 프로그램은 **이벤트 루프**(**event loop**)라는 단순한 구조를 기반으로 짜여 있다. 이벤트 루프에 대한 코드는 다음과 같다.

```
1 while (1) {
2     events = getEvents();
3     for (e in events)
4        processEvent(e);
5 }
```

매우 간단한다. 루프내에서 사건 발생을(코드의 `getEvents()`를 실행한 결과) 대기한다. 이벤트가 발생하면 하나씩 처리한다. 이때 각 이벤트를 처리하는 코드를 **이벤트 핸들러**(**event handler**)라 부른다. 중요한 것은 이벤트의 처리가 시스템의 유일한 작업이기 때문에, 다음에 처리할 이벤트를 결정하는 것이 스케줄링과 동일한 효과를 갖는다. 스케줄링을 제어할 수 있는 기능이 이벤트 기반 방법의 큰 장점 중 하나이다.

하지만 큰 질문 하나가 생긴다. 발생한 이벤트가 무슨 이벤트인지 어떻게 판단할까? 네트워크나 디스크 I/O의 경우 특히 쉽지 않다. 즉, 디스크 I/O가 완료되었다는 이벤트가 도착했을 때 어떤 디스크 요청이 완료되었느냐 하는 것이다. 좀 더 구체적으로 도착한 메시지가 자신을 위한 것인지 어떻게 판단할까?

36.2 중요 API: `select()` (또는 `poll()`)

기본질문: 이벤트를 어떻게 받을까? 대부분의 시스템은 `select()` 또는 `poll()` 시스템 콜을 기본 API로서 제공한다.

인터페이스의 기능은 간단하다. 도착한 I/O들 중 주목할 만한 것이 있는지를 검사하는 것이다. 예를 들면, 웹 서버 같은 네트워크 응용 프로그램이 자신이 처리할 패킷의 도착 여부를 검사하는 것이다. 이 시스템 콜들이 정확히 해당 역할을 한다.

`select()`를 예로 살펴보자. Mac OS X가 제공하는 매뉴얼은 이 API를 다음과 같이 설명한다.

```
1 int select(int nfds,
2            fd_set *restrict readfds,
3            fd_set *restrict writefds,
4            fd_set *restrict errorfds,
5            struct timeval *restrict timeout);
```

매뉴얼의 내용은 다음과 같다. *`select()`는 `readfds`, `writefds`, 그리고 `errorfds`를 통해 전달된 I/O 디스크립터(descriptor) 집합들을 검사해서, 각 디스크립터들에 해당하는 입출력 디바이스가 읽을 준비가 되었는지, 쓸 준비가 되었는지, 처리해야 할 예외 조건이 발생했는지 등을 파악한다. 각 집합의 첫 번째 `nfds` 개의 디스크립터들, 즉 0 부터 `nfds-1`까지의 디스크립터를 검사한다. `select`는 집합을 가리키는 각 포인터들을 준비된 디스크립터들의 집합으로 교체한다. `select()`는 전체집합에서 준비된 디스크립터들의 총 개수를 반환한다.*

`select()`에 대한 두 가지 알아두어야 할 사항이 있다. 첫 번째, `select()`를 이용하면 디스크립터에 대한 읽기 가능여부, 쓰기 가능여부를 검사할 수 있다. 전자는

여담: 차단(blocking)과 비차단(non-blocking) 인터페이스

차단(또는 **동기(synchronous)**) 인터페이스는 호출자에게 리턴하기 전에 자신의 작업을 모두 처리하는 반면 비차단(또는 **비동기(asynchronous)**) 인터페이스는 작업을 시작하기는 하지만, 즉시 반환하기 때문에 처리되어야 하는 일이 백그라운드에서 완료가 된다.

차단 호출은 주로 I/O 때문에 발생한다. 예를 들어, 작업 완료를 위해서 디스크에서 읽어야 하는 자료가 있다면, 디스크에 요청한 I/O 요청을 대기해야 한다.

비차단 인터페이스(non-blocking interface)는 모든 프로그래밍(예, 멀티 쓰레드 프로그래밍) 스타일에서 사용될 수 있다. 하지만, 이벤트 기반의 프로그래밍 방식에서는 필수적이다. 차단 방식의 시스템 콜(blocking call)이 전체 시스템을 멈출 수 있기 때문이다.

처리해야 할 패킷의 도착 여부를 파악할 수 있도록 한다. 후자는 서비스가 응답전송이 가능한 시점(예를 들어 아웃바운드(outbound) 큐가 가득 차지 않았다)을 파악토록 해준다.

두 번째는 `timeout` 인자의 존재이다. 일반적으로는 `NULL`로 설정한다. 그러면 `select()`는 디스크립터가 준비가 될 때까지 무한정 대기한다. 하지만, 오류에 대비토록 설계된 서버들의 경우 `timeout` 값을 설정해 두기도 한다. 널리 사용되는 방법으로는 `timeout` 값을 0으로 설정하여 `select()`가 즉시 리턴하도록 하는 것이다.

`poll()` 시스템 콜도 유사하다. 좀 더 자세한 내용은 매뉴얼이나 Stevens와 Rago [SR05a]를 참고하기 바란다.

이런 기본 함수로 non-blocking event loop를 만들어, 패킷 도착을 확인하고, 소켓에서 메시지를 읽고 필요에 응답할 수 있도록 해준다.

36.3 select()의 사용

확실한 이해를 위해, `select()`를 이용해 어떤 네트워크 디스크립터에 메시지가 도착했는지를 파악하는 경우를 살펴보자. 그림 36.1에 그 예를 나타내었다.

이 코드는 이해하기 쉽다. 초기화 후에 서버는 무한 루프에 들어간다. 그 루프 내에서 `FD_ZERO()` 매크로를 사용하여 파일 디스크립터들을 초기화한 후, `FD_SET()`를 사용하여 `minFD`에서 `maxFD`까지의 파일 디스크립터 집합에 포함시킨다. 이 집합은 서버가 보고 있는 모든 네트워크 소켓 같은 것들을 나타낼 수 있다. 마지막으로 서버는 `select()`를 호출하여 데이터가 도착한 소켓이 있는지를 검사한다. 반복문 내의 `FD_ISSET()`를 사용하여 이벤트 서버는 어떤 디스크립터들이 준비된 데이터를 갖고 있는지를 알 수 있으며 도착하는 데이터를 처리할 수 있게 된다.

물론, 실제 서버는 이보다 더 복잡하다. 디스크 작업이나 메시지를 보내는 시점, 그 외 세부 사항들을 결정하는 로직이 필요하다. API 정보는 Stevens와 Rago [SR05b]의 연구를 참고하고, 이벤트 기반 서버의 일반적인 동작 흐름을 위해서는 Pai 등의 연구나

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main(void) {
8      // 여러 개의 소켓을 열고 설정(여기엔 나타나 있지 않음)
9      // 주 반복문
10     while (1) {
11         // fd_set를 모두 0으로 초기화함
12         fd_set readFDs;
13         FD_ZERO(&readFDs);
14
15         // 이제 이 서버가 관심 있어 하는
16         // 디스크립터들의 bit를 설정
17         // (단순함을 위해서, min부터 max까지)
18         int fd;
19         for (fd = minFD; fd < maxFD; fd++)
20             FD_SET(fd, &readFDs);
21
22         // 선택을 함
23         int rc = select(maxFD+, &readFDs, NULL, NULL, NULL);
24
25         // FD_ISSET()를 사용하여 실제 데이터 사용 여부 검사
26         int fd;
27         for (fd = minFD; fd < maxFD; fd++)
28             if (FD_ISSET(fd, &readFDs))
29                 processFD(fd);
30     }
31 }
```

〈그림 36.1〉 **select()를 사용하는 간단한 코드**

팁: 이벤트 기반의 서버 내에서는 블럭을 하지 말자

이벤트 기반 서버는 작업의 스케줄링을 정밀하게 제어할 수 있다. 하지만, 정밀한 제어를 위해서는 호출자가 실행한 것을 차단할 수 있는 어떠한 호출도 있어서는 안 된다. 이 디자인 팁을 지키지 않는다면 이벤트 기반 서버가 멈추게 될 것이고 사용자는 불만을 가질 것이다. 무엇보다 이 책을 제대로 읽었는지 의문이 들 것이다.

Welsh 등의 연구를 참고하자 [PDZ99; WCB01].

36.4 왜 간단한가? 락이 필요 없음

단일 CPU를 사용하는 이벤트 기반의 응용 프로그램에서는 병행 프로그램을 다룰 때 나타났던 문제들은 더 이상 보이지 않는다. 그 이유는 매순간에 단 하나의 이벤트만 다루기 때문에 락을 획득하거나 해제해야 할 필요가 없다. 이벤트 기반의 서버는 단 하나의 쓰레드만 갖고 있기 때문에 다른 쓰레드에 의해서 인터럽트에 걸릴 수가 없다. 그렇기 때문에 쓰레드 프로그램에서 흔한 병행성 버그는 기본적인 이벤트 기반 접근법에서는 나타나지 않는다.

36.5 문제: 블로킹 시스템 콜(Blocking System Call)

이제까지는 이벤트 기반 프로그래밍이 괜찮아 보이지 않았는가? 간단한 루프가 있어서 이벤트가 발생하면 그때마다 처리하도록 개발하면 된다. 락을 고려할 필요조차 없다! 하지만 문제가 있다. 차단될 수도 있는 시스템 콜을 불러야 하는 이벤트가 있다면 어떻게 할까?

예를 들어 디스크에서 데이터를 읽어서 그 내용을 사용자에게 전달하는 요청을 생각해 보자(간단한 HTTP 요청과 비슷한 경우이다). 이러한 요청을 처리하려면 이벤트 핸들러가 `open()` 시스템 콜을 사용하여 파일을 열어서 `read()` 명령어를 사용하여 파일을 읽어야 한다. 파일을 읽어서 메모리에 탑재한 후에야 서버는 그 결과를 사용자에게 전달할 수 있게 된다.

`open()`과 `read()` 모두 저장 장치에 I/O 요청을 보내야 한다면 (메타데이터가 필요하거나 데이터가 메모리에 없기 때문에), 이 요청을 처리하기 위해서 오랜 시간이 필요하다. 쓰레드 기반 서버는 이런 것이 문제 되지 않는다. 한 쓰레드가 I/O를 대기하면 다른 쓰레드가 실행이 되며 서버는 계속 동작할 수 있다. I/O 처리와 다른 연산이 자연스럽게 **겹쳐지는 현상(overlap)**이 쓰레드 기반 프로그래밍의 장점이다.

반면에 이벤트 기반의 접근법에서는 쓰레드가 없고 단순히 이벤트 루프만 존재한다. 즉, 이벤트 핸들러가 블로킹 콜을 호출을 하면 서버 전체가 오직 그 일을 처리하기 위해, 명령어가 끝날 때까지 다른 것들을 차단한다. 이벤트 루프가 블록되면 시스템은 유휴 상태가 된다. 심각한 자원 낭비가 발생된다. 이벤트 기반 시스템의 기본 원칙은 블로킹 호출을 허용하면 안 된다는 것이다.

36.6 해법: 비동기 I/O

언급된 한계를 극복하기 위해 여러 현대의 운영체제들이 I/O 요청을 디스크로 내려보낼 수 있는 일반적으로 **비동기 I/O(asynchronous I/O)**라고 부르는 새로운 방법을 개발하였다. 이 인터페이스는 프로그램이 I/O 요청을 하면 I/O 요청이 끝나기 전에 제어권을 즉시 다시 호출자에게 돌려주는 것을 가능하게 했으며 추가적으로 여러 종류의 I/O들이 완료가 되었는지도 판단할 수 있도록 하였다.

예를 들어 Mac OS X가 제공하는 인터페이스를 살펴보도록 하자(다른 운영체제도 비슷한 API를 제공한다). 이 API는 `struct aiocb` 또는 전문 용어로 **AIO 제어 블럭(AIO control block)**이라고 불리는 기본적인 구조를 사용하고 있다. 간단화한 구조는 다음과 같다(더 자세한 내용은 매뉴얼을 보기 바란다).

```
struct aiocb {
    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
};
```

파일에 대한 비동기 읽기 요청을 하려면 응용 프로그램은 먼저 이 자료 구조에 읽고자 하는 파일의 파일 디스크립터(`aio_fildes`), 파일 내에서 위치(`aio_offset`)와 더불어

요청의 길이(`aio_nbytes`), 그리고 마지막으로 읽기 결과로 얻은 데이터를 저장할 대상 메모리의 위치(`aio_buf`)와 같은 관련 정보를 채워 넣어야 한다.

이 자료 구조에 정보가 다 채워지면 응용 프로그램은 읽으려는 파일에 비동기 호출을 보낸다. Mac OS X에서는 간단한 **비동기 읽기**(**asynchronous read**) API를 사용한다.

```
int aio_read(struct aiocb *aiocbp);
```

이 명령어를 통해 I/O 호출을 성공하면, 즉시 리턴을 하며 응용 프로그램(이벤트 기반의 서버 류)은 하던 일을 계속 진행할 수 있다.

풀어야 하는 나머지 퍼즐 한 조각이 있다. I/O가 종료되었다는 것을 어떻게 알 수 있을까 그리고 `aio_buf`가 가리키는 버퍼에 요청했던 데이터가 있다는 것을 어떻게 알 수 있을까?

마지막으로 API 하나가 필요하다. Mac OS X에서는 이 API를 `aio_error()`(혼란스럽기는 하지만)라고 한다. API는 다음과 같다.

```
int aio_error(const struct aiocb *aiocbp);
```

이 시스템 콜은 `aiocbp`에 의해 참조된 요청이 완료되었는지를 검사한다. 완료되었다면 성공했다고 리턴을 하고(0으로 표시) 실패했다면 `EINPROGRESS`을 반환한다. 모든 대기 중인 비동기 I/O는 주기적으로 `aio_error()` 시스템 콜로 시스템에 **폴링**(**poll**)하여 해당 I/O가 완료되었는지 확인할 수 있다.

어떤 I/O가 완료되었는지 확인하는 것이 귀찮게 느껴질 수도 있다. 만약 어떤 시점에 수십 또는 수백 개의 I/O를 요청하는 프로그램이 있다면, 그 많은 요청들을 일일이 다 검사해야 할 것인가 아니면 먼저 일정 시간 동안을 대기해야 할까, 그것도 아니라면?

이 문제의 해결을 위해서 어떤 시스템들은 **인터럽트** 기반의 접근법을 제공한다. 유닉스의 **시그널**(**signal**)을 사용하여 비동기 I/O가 완료되었다는 것을 응용 프로그램에게 알려주기 때문에 시스템에 반복적으로 완료 여부를 확인할 필요가 없다. 폴링 대 인터럽트 문제는 I/O 장치들을 다룰 때에도 나타난다.

비동기 I/O가 없는 시스템에서는 제대로 된 이벤트 기반의 접근법을 구현할 수 없다. 하지만, 현명한 연구자들이 그 대신 꽤 괜찮게 동작할 수 있는 방법을 고안했다. 예를 들면 Pai 등 [PDZ99]은 네트워크 패킷을 처리하기 위해 이벤트를 사용하고 대기 중인 I/O들을 처리하기 위해 쓰레드 풀을 사용하는 하이브리드 기법을 제안하였다. 더 자세한 내용은 논문을 살펴보자.

36.7 또 다른 문제점: 상태 관리

이벤트 기반 접근법의 또 다른 문제점은 전통적인 쓰레드 기반 코드보다 일반적으로 더 작성하기 복잡하다는 것이다. 그 이유는 다음과 같다. 이벤트 핸들러가 비동기 I/O를 발생시킬 때, I/O 완료 시 사용할 프로그램 상태를 정리해 놓아야 한다. 이 작업은 쓰레드 기반 프로그램에서는 불필요하다. 왜냐하면 쓰레드 스택에 그 정보들이 이미 들어 있기 때문이다. Adya 등은 이것을 **수동 스택 관리**(**manual stack management**)라고 부르며 이벤트 기반 프로그래밍에서는 기본이다 [Ady+02].

여담: 유닉스 시그널

시그널(**signal**)은 거의 모든 현대 UNIX에서 찾아 볼 수 있는 매우 흥미롭고, 유용하고, 포괄적인 개념이다 (역자 주: 시그널의 아름다움을 이해한다면 당신은 전문가이다). 간단하게 시그널은 프로세스 간의 통신 방법이다. 응용 프로그램에게 시그널이 전달되면 해당 응용 프로그램은 하던 작업을 중지하고 **시그널 핸들러**(**signal handler**)를 실행한다. 시그널 핸들러는 프로그램이 정의한 시그널 처리 코드이다. 시그널 처리가 완료되면 프로세스는 이전 작업을 재개한다.

시그널마다 이름이 있다. **HUP**(hang up의 약자, 끊다), **INT**(interrupt의 약자, 인터럽트), **SEGV**(segmentation violation의 약자, 세그멘트 위반)등이 그것이다. 각각에 대한 자세한 설명은 매뉴얼을 참고하기 바란다. 커널 자체도 시그널 전송의 주체가 되기도 한다. 예를 들어 당신이 만든 프로그램이 세그멘트 위반(segment violation)을 하면 운영체제는 **SIGSEGV**(**SIG**는 시그널의 이름 앞에 붙는 흔한 접두어이다)를 보낸다. 프로그램이 시그널을 접수할 수 있도록 설정되어 있다면 (역자 주: 시스템 콜을 사용하여 프로세스가 특정 시그널을 받지 못하도록 설정할 수 있다. 이를 시그널을 masking한다고 한다.), 프로그램이 오류를 발생시켰을 때, 특정 코드를 실행하도록 만들 수 있다(디버깅 할 때 유용하다). 대부분의 프로세스들은 각 시그널들에 대한 처리코드가 정의되어 있다. 정의되지 않은 시그널을 받으면 프로세스는 기본 동작을 수행한다. SEGV의 경우에 프로세스를 강제 종료한다.

다음의 간단한 프로그램은 무한 루프를 수행한다. 무한 루프에 들어가기 전에 SIGHUP 시그널에 대한 핸들러를 먼저 설정하고 있다.

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}
int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // 시그널을 기다리는 것 외에는 아무 일도 안함
    return 0;
}
```

이 프로그램에 **kill** 명령어(초기부터 사용된 이름이다. 좀 어색하고 지나치게 과격한 면이 있는 명령어 이름이다.)를 사용하여 시그널을 보낼 수 있다. 이 시그널이 도착하면 프로그램은 **while** 루프를 중단하고 **handle()**함수를 실행한다.

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
  stop wakin' me up...
prompt> kill -HUP 36705
  stop wakin' me up...
prompt> kill -HUP 36705
  stop wakin' me up...
```

시그널에 대한 설명은 한 페이지로는 부족하며 한 챕터를 할애해도 충분하지 않다. 늘 그렇듯이 잘 설명된 자료가 있다. Stevens와 Rago [SR05b] 이다. 관심있는 사람들은 읽어보기 바란다.

구체적으로 기술을 해보겠다. 쓰레드 기반 서버를 예로 들자. 이 서버는 파일 디스크립터 (`fd`)로 명시된 파일에서 데이터를 읽어들여, 해당 데이터들을 네트워크 소켓 디스크립터(`sd`)로 전송한다. (오류 처리는 무시한) 코드는 다음과 같다.

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

이러한 작업은 멀티 쓰레드 프로그램에서는 매우 간단한 일이다. `read()`가 리턴되면 전송할 네트워크 소켓에 관한 정보가 같은 스택에 존재하기 때문이다.

이벤트 기반의 시스템에서는 상황이 그렇게 간단하지만은 않다. 같은 일을 하려면 앞서 명시한 AIO 호출들을 사용하여 `read()`를 비동기로 요청해야 한다. `aio_error()`를 사용하여 주기적으로 읽기가 종료되었는지를 확인한다고 가정해 보자. 이 호출이 읽기가 종료되었다고 알려주면 이벤트 기반 서버는 다음으로 무슨 일을 해야 할 지 어떻게 알 것인가?

그 해법은 Adya 등 [Ady+02]이 기술하듯이 프로그램 언어 분야에서 오랫동안 사용되어온 개념인 **continuation**을 사용하는 것이다 [FHK84]. 복잡하게 들릴 수도 있지만 개념은 단순하다. 이벤트를 종료하는 데에 필요한 자료들을 한곳에 저장해 둔다. 이벤트가 발생하면(디스크 I/O가 완료되면), 저장해 놓은 정보들을 활용하여 이벤트를 처리한다.

앞서 사용한 예의 해법은 소켓 디스크립터(`sd`)를 파일 디스크립터(`fd`)가 사용하는 자료 구조(예: 해시 테이블)에 저장해 놓는 것이다. 디스크 I/O가 완료되면 이벤트 핸들러가 파일 디스크립터에서 다음 할 일을 파악하여 호출자에게 소켓 디스크립터의 값을 반환하도록 한다. 이 시점에서 (최종적으로), 서버는 소켓에 데이터를 기록하는 마지막 동작을 할 수 있게 된다.

36.8 이벤트 사용의 어려움

이벤트 기반 접근법에는 다른 어려운 점이 몇 개 존재한다. 예를 들어 단일 CPU에서 멀티 CPU로 변경되면, 이벤트 기반 접근법의 단순함이 없어진다. 구체적으로 말하자면, 하나 이상의 CPU를 활용하기 위해서는 다수의 이벤트 핸들러를 병렬적으로 실행해야. 그렇게 되면 동기화 문제(예, 임계 영역)가 발생하게 되며 이의 해결에 필요한 기능(예, 락)을 사용할 수밖에 없다. 때문에 근례의 멀티코어 시스템은 락이 없는 이벤트 처리 방식은 더 이상 사용할 수 없게 된다.

또 다른 문제는 이벤트 기반의 접근법은 **페이징**(**paging**)과 같은 특정 종류의 시스템과 잘 맞지 않는다. 예를 들면 이벤트 핸들러에서 페이지 폴트가 발생하면 동작이 중단되기 때문에 서버는 페이지 폴트가 처리 완료되기 전까지는 진행을 할 수 없게 된다. 서버가 비차단(non-blocking) 방식으로 설계되었다 할지라도, 페이지 폴트와 같은 내재적 원인으로 인한 차단은 피하기가 어렵다. 이런 상황이 자주 발생하는 경우에 심각한 성능 하락을 가져 올 수 있다.

세 번째 문제는 루틴의 작동 방식이 계속 변화하기 때문에, 이벤트 기반에서는 이들의 관리가 어려워진다 [Ady+02]. 예를 들어 루틴 동작이 비차단 방식에서 차단방식으로

변경된다면 그 루틴을 호출하는 이벤트 핸들러 역시 새로운 성질에 적응하도록 변경해야 한다. 이에 적합하게 루틴을 두 버전으로 나눠야 한다. 이벤트 기반 서버에서 차단(block)이라는 것은 치명적이기 때문에 개발자는 각 이벤트가 사용하는 API의 문법이 변경이 되었는지를 늘 주의 깊게 살펴야 한다.

마지막으로 비동기 디스크 I/O가 대부분의 플랫폼에서 사용 가능하지만, 그렇게 되기까지 상당히 오랜 시간이 걸렸다 [PDZ99]. 더구나, 아직까지도 비동기 네트워크 I/O는 생각하는 것만큼 간단하고 일관성 있게 적용되어 있지 않다. 예를 들면, 모든 입출력 처리에 `select()`를 사용하여 일관성을 유지하는 것이 이상적이지만, 일반적으로 네트워크 요청의 처리에는 `select()`가, 디스크 I/O 에는 AIO가 사용되고 있다.

36.9 요약

이벤트 스타일에 따른 다른 종류의 병행성에 대해 간략히 살펴보았다. 이벤트 기반 서버는 프로그램 자체에 스케줄링에 대한 제어권을 부여하지만 복잡도가 높고 현대 시스템의 다른 부분들(예, 페이징)로 인해 적용이 어렵다는 문제가 있다. 이러한 도전 과제들 때문에 어떤 한 가지 기법도 만족스럽지 않다. 그렇기 때문에 쓰레드와 이벤트는 병행성 문제에 대한 두 개의 다른 접근법으로 오랜 기간 남을 것이다. 연구 논문들을 읽어보도록 하자(예, [Ady+02; PDZ99; vBeh+03; WCB01]. 더 좋은 방법을 배우기 위해서 이벤트 기반 코드를 작성하여 보자.

참고 문헌

[Ady+02] **"Cooperative Task Management Without Manual Stack Management"**
Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur
USENIX ATC '02, Monterey, CA, June 2002
이 논문의 핵심은 이벤트 기반의 병행성 문제들을 먼저 분명하게 소개한 것과 간단한 해결법들을 제시하였고 그와 더불어 하나의 응용 프로그램에 두 개의 서로 다른 병행성 관리 방법을 병합하는 말도 안 될 법한 방법들을 고찰한 것이다.

[FHK84] **"Programming With Continuations"**
Daniel P. Friedman, Christopher T. Haynes, and Eugene E. Kohlbecker
In Program Transformation and Programming Environments, Springer Verlag, 1984
고전적인 참고 문헌으로 프로그래밍 언어의 세계에서 사용되는 오래된 개념을 설명하였다. 이제는 현대의 언어들에서도 상당한 인기를 끌고 있다.

[N1313] **"Node.js Documentation"**
By the folks who build node.js
URL: `http://nodejs.org/api/`
웹 서비스와 응용 프로그램을 손쉽게 만들어 줄 수 있는 새로운 프레임워크 중에 하나이다. 현대의 시스템 해커는 이와 같은 프레임워크에 능통해야 한다(하나 이상에 능통해야겠지만). 이와 같은 프레임워크 중에 하나를 습득하는 데 시간을 들여서 전문가가 되어 보자.

[Ous96] **"Why Threads Are A Bad Idea (for most purposes)"**
John Ousterhout
Invited Talk at USENIX '96, San Diego, CA, January 1996
GUI-기반의 응용 프로그램은 쓰레드가 상대할 대상이 아니라는 것을 잘 보여준 강연이다 (대체적으로 일반론이기는 하다). Ousterhout는 Tcl/Tk를 구현할 때 이러한 주장을 설파했다. Tcl/Tk는 스크립트 기반의 언어이자 도구로서 그 당시의 최고 수준의 기법들 보다 GUI-기반의 응용 프로그램을 100배 더 쉽게 만들 수 있도록 하였다. Tk GUI 도구는 아직도 사용되고 있지만 (Python이 그 중 하나이다), (불행하게도) Tcl은 서서히 사라지고 있는 중이다.

[PDZ99] **"Flash: An Efficient and Portable Web Server"**
Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel
USENIX '99, Monterey, CA, June 1999
그 당시에는 급증하고 있던 인터넷 시대에 웹 서버를 구성하는 방법을 다룬 선구자적인 논문이다. 기초적인 것을 이해하기 위해서 뿐만 아니라 비동기 I/O의 지원이 부족하다고 여길 때 적용 가능한 하이브리드 기법에 대한 저자의 발상을 읽어볼 수 있다.

[SR05a] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
다시 한 번, 책장에 반드시 꼽혀 있어야 할 유닉스 시스템 프로그래밍의 고전을 소개 한다. 어떤 것에 설명이 필요로 하다면 여기에 나와 있다.

[SR05b] **"Advanced Programming in the Unix Environment"**
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
UNIX *API 사용의 미묘한 차이와 절묘함을 발견할 수 있는 책. 책을 사서 읽어라. 그리고 더 중요한 건 항상 곁에 두어야 한다.*

[vBeh+03] **"Capriccio: Scalable Threads for Internet Services"**

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer
SOSP '03, Lake George, New York, October 2003
모든 이벤트 기반의 작업에 대한 카운터를 동시에 처리하는 것과 같이 극단의 범위까지 확장하였을 때 쓰레드를 사용하는 방법을 설명한 논문이다.

[WCB01] **"SEDA: An Architecture for Well-Conditioned, Scalable Internet Services"**
Matt Welsh, David Culler, and Eric Brewer
SOSP '01, Banff, Canada, October 2001
쓰레드와 큐 그리고 이벤트 기반 핸들러를 하나로 통합하여 이벤트 기반 서버를 구현하였다. 여기서 사용된 몇몇 개념들은 구글과 아마존과 같은 기업들의 기반 시설에 응용되기도 하였다.