31

락

병행성에 대한 소개 이후 병행 프로그램의 근본적인 문제 몇 개를 살펴보았다. 여러 개의 명령어들을 원자적으로 실행해보고 싶지만 단일 프로세서의 인터럽트로 인해서 (또는 멀티 쓰레드를 여러 프로세서에 병행성하려고 해서) 그렇게 할 수가 없었다. 이 장에서는 앞서 다룬 **락**(**lock**)을 이용하여 이 문제를 직접적으로 다루고자 한다. 프로그래머들은 소스 코드의 임계 영역을 락으로 둘러서 그 임계 영역이 마치 하나의 원자 단위 명령어인 것처럼 실행되도록 한다.

31.1 락: 기본 개념

예를 위해 다음의 임계 영역이 있다고 하자. 고전적인 공유 변수의 갱신이다.

```
balance = balance + 1;
```

연결 리스트에 노드를 삽입한다거나 공유 구조에 복잡한 갱신과 같은 다른 임계 영역을 사용할 수 있겠지만 간단한 예제를 사용하도록 하겠다. 락을 사용하기 위해서 락으로 임계 영역을 다음과 같이 감쌌다.

```
1  lock_t mutex; // 글로벌 변수로 선언된 락
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

락은 하나의 변수이므로 때문에 락을 사용하기 위해서는 락 변수를(위에 사용한 `mutex`와 같이) 먼저 선언해야 한다. 이 **락 변수**는(또는 짧게 락은) 락의 상태를 나타낸다. 이 락은 **사용 가능**(**available**) 상태이거나(**해제**(**unlocked**) 또는 **free**), 즉 어느 쓰레드도 락을 갖고 있지 않거나, **사용 중**(**acquired**), 즉 임계 영역에서 정확히 하나의 쓰레드가 락을 **획득**한 상태이다. 이 락 자료 구조에 락을 보유한 쓰레드에 대한 정보나 락을 대기하는 쓰레드들에 대한 정보를 저장할 수도 있다. 물론, 이러한 정보는 락 사용자는 알 수 없다.

`lock()`과 `unlock()` 루틴의 의미는 간단하다. `lock()` 루틴 호출을 통해 락 획득을 시도한다. 만약 어떤 쓰레드도 락을 갖고 있지 않으면 그 쓰레드는 락을 획득하여 임계 영역 내로 진입한다. 이렇게 진입한 쓰레드를 락 **소유자**(**owner**)라고 부른다.

만약 다른 쓰레드가 `lock()`을 호출한다면(이 예제에서는 `mutex`에 해당) 사용 중인 동안에는 `lock()` 함수가 리턴하지 않는다. 이와 같은 방식으로 락을 보유한 쓰레드가 임계 영역에 진입한 상태에는 다른 쓰레드들이 임계 영역 안으로 진입할 수가 없다.

락 소유자가 `unlock()`을 호출한다면 락은 이제 다시 사용 가능한 상태가 된다. 어떤 쓰레드도 이 락을 대기하고 있지 않았다면(어떤 쓰레드도 `lock()`을 호출하여 멈춰 있던 상태가 아니라면) 락의 상태는 사용 가능으로 유지된다. 만약에 대기 중이던 쓰레드가 있었다면(`lock()`으로 인해 멈춘), 락의 상태가 변경되었다는 것을 (결국엔) 인지하고 (또는 정보를 받아서) 락을 획득하여 임계 영역 내로 진입하게 된다.

락은 프로그래머에게 스케줄링에 대한 최소한의 제어권을 제공한다. 일반적으로 쓰레드는 프로그래머가 생성하고 운영체제가 제어한다. 락은 쓰레드에 대한 제어권을 일부 이양 받을 수 있도록 해준다. 락으로 코드를 감싸서 프로그래머는 그 코드 내에서는 하나의 쓰레드만 동작하도록 보장할 수 있다. 락을 통해 프로세스들의 혼란스런 실행 순서에 어느 정도를 질서를 부여할 수 있다.

31.2 Pthread 락

쓰레드 간에 **상호 배제**(**mutual exclusion**) 기능을 제공하기 때문에 POSIX 라이브러리는 락을 **mutex**라고 부른다. 상호 배제는 한 쓰레드가 임계 영역 내에 있다면 이 쓰레드의 동작이 끝날 때까지 다른 쓰레드가 임계 영역에 들어 올 수 없도록 제한한다고 해서 얻은 이름이다. 다음과 같은 POSIX 쓰레드 코드를 만나면 앞에서 언급한 것과 같은 동작을 한다고 이해하면 된다(래퍼를 사용하여 락과 언락 시에 에러를 확인하도록 하였다).

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock); // pthread_mutex_lock()을 위한 래퍼
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);
```

POSIX 방식에서는 변수 명을 지정하여 락과 언락 함수에 전달하고 있다. 다른 변수를 보호하기 위해서 *다른* 락을 사용할 수도 있기 때문이다. 이러한 방식을 통해 동시에 더 많은 일을 처리할 수 있다. 하나의 락이 임의의 임계 영역에 진입할 때마다 사용하는 것이 아니라(**거친**(**coarse-grained**) 락 사용 전략), 서로 다른 데이터와 자료 구조를 보호하기 위해 여러 락을 사용하여 한 번에 여러 쓰레드가 서로 다른 락으로 보호된 코드 내에 각자가 진입이 가능하도록 할 수 있다(**세밀한**(**fine-grained**) 락 사용 전략).

31.3 락 구현

락의 동작을 프로그래머 입장에서 대략 이해했을 것이다. 하지만 어떻게 락을 만들어야 할까? 어떤 종류의 하드웨어 지원이 필요할까? 운영체제는 무엇을 지원해야 하는가? 이러한 문제들을 이제 이번 장에서 다루어보고자 한다.

핵심 질문: 락은 어떻게 만들까

효율적인 락은 어떻게 만들어야 하는가? 효율적인 락은 낮은 비용으로 상호 배제 기법을 제공하고 다음에 다룰 몇 가지 속성들을 추가로 가져야 한다. 어떤 하드웨어 지원이 필요한가? 어떤 운영체제 지원이 필요한가?

사용 가능한 락을 만들기 위해서는 우리의 오래된 친구들인 하드웨어와 운영체제의 도움을 받아야 한다. 오랜 기간을 걸쳐 다양한 컴퓨터 구조의 명령어 집합에 여러 하드웨어 명령어들이 추가되었다. 이러한 명령어들이 어떻게 구현되었는지는 컴퓨터 구조 수업의 주제이기 때문에 다루지 않겠지만, 락과 같은 상호 배제를 위한 기법을 만드는 데 어떻게 활용하는지는 학습할 것이다. 정교한 락 라이브러리를 제작하는 데 있어 운영체제가 관여하는 것은 무엇인지 알아보겠다.

31.4 락의 평가

어떤 락이든 만들기 전에 첫째로 목표를 이해해야 하고 구현의 효율을 어떻게 평가할지 질문해야 한다. 락이 동작하는지(잘 동작하는지) 평가하기 위해 먼저 기준을 정해야 한다. 첫째는 **상호 배제**를 제대로 지원하는가이다. 가장 기본 역할이다. 기본적으로 락이 동작하여 임계 영역 내로 다수의 쓰레드가 진입을 막을 수 있는지 검사해야 한다.

둘째는 **공정성(fairness)**이다. 쓰레드들이 락 획득에 대한 공정한 기회가 주어지는가? 이것을 보는 다른 관점은 좀 더 극단적인 상황을 평가해 보는 것이다. 락을 전혀 얻지 못해 **굶주리는(starve)** 경우가 발생하는가?

마지막 기준은 **성능(performance)**이다. 특히, 락 사용 시간적 오버헤드를 평가해야 한다. 이 주제에 대해서 고려해야 할 몇 가지 사항이 있다. 하나는 경쟁이 전혀 없는 경우의 성능이다. 하나의 쓰레드가 실행 중에 락을 획득하고 해제하는 과정에서 발생하는 부하는 얼마나 되는가? 다음은 여러 쓰레드가 단일 CPU 상에서 락을 획득하려고 경쟁할 때의 성능이다. 마지막은 멀티 CPU 상황에서 락 경쟁 시의 성능이다. 이런 서로 다른 상황의 성능을 평가하여야 앞으로 다룰 다양한 락 기법들이 성능에 미치는 영향을 이해할 수가 있다.

31.5 인터럽트 제어

초창기 단일 프로세스 시스템에서는 상호 배제 지원을 위해 임계 영역 내에서는 인터럽트를 비활성화하는 방법을 사용했었다. 코드는 다음과 같다.

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

그렇게 동작하는 단일 프로세서를 사용한다고 가정해 보자. 임계 영역에 진입하기 전에 특별한 하드웨어 명령어를 사용하여 인터럽트를 막는다면 임계 영역 내의 코드에서는 인터럽트가 발생할 수 *없기* 때문에 원자적으로 실행될 수 있다. 모든 동작이 끝난 후에 다시 하드웨어 명령어를 사용하여 인터럽트를 사용 가능하도록 하여 프로그램이 이전처럼 진행할 수 있도록 한다.

이 방법의 주요 장점은 단순하다는 것이다. 이 방법이 왜 제대로 동작하는지 깊이 생각하지 않아도 된다. 인터럽트가 발생하지 않으면, 코드가 실행 중에 다른 쓰레드가 중간에 끼어들지 않는다는 것을 보장할 수 있다.

이 방법은 단점이 많다. 먼저 이 요청을 하는 쓰레드가 인터럽트를 활성/비활성화하는 **특권**(**privileged**) 연산을 실행할 수 있도록 허가해야 한다. 또, 이를 다른 목적으로 사용하지 않음을 신뢰할 수 있어야 한다. 운영체제가 잘 알지 못하는 다른 프로그램을 신뢰해야 하는 경우가 생긴다면, 대부분 곤경에 빠진 것이라고 보면 된다. 몇 가지 경우의 형태를 살펴보자. 탐욕적(Greedy) 기법을 사용한 프로그램이 시작과 동시에 `lock()`을 호출하여 프로세서를 독점하여 사용할 수도 있다. 더한 경우라면 오류가 있거나 악의적인 프로그램이 `lock()`을 호출하고 무한 반복문에 들어갈 수도 있다. 후자의 경우라면 운영체제는 시스템의 제어권을 다시 얻을 수가 없다. 동기화를 위해 인터럽트를 비활성화시키는 방법은 응용 프로그램을 너무 많이 신뢰해야 한다는 문제가 있다.

두 번째 단점은 멀티프로세서에서는 적용을 할 수가 없다는 것이다. 여러 쓰레드가 여러 CPU에서 실행 중이라면 각 쓰레드가 동일한 임계 영역을 진입하려고 시도할 수 있다. 이때에 특정 프로세서에서의 인터럽트 비활성화는 다른 프로세서에서 실행 중인 프로그램에는 전혀 영향을 주지 않는다. 결과적으로는 임계 영역에 진입할 수 있다는 말이다. 멀티프로세서가 이제는 매우 흔하게 사용되고 있기 때문에 해결법은 더 정교해야 한다.

세 번째 단점은 장시간 동안 인터럽트를 중지시키는 것은 중요한 인터럽트의 시점을 놓칠 수 있다는 것이다. 때로는 시스템에 심각한 문제를 가져 올 수 있다. 예를 들어 CPU가 저장 장치에서 읽기 요청을 마친 사실을 모르고 지나갔다고 해 보자. 운영체제가 이 읽기 결과를 기다리는 프로세스를 언제 깨울 수 있을까?

마지막으로 가장 덜 중요할 수도 있겠지만, 이 방법은 비효율적이다. 일반적인 명령어의 실행에 비해 인터럽트를 비활성화시키는 코드들은 최신의 CPU들에서는 느리게 실행되는 경향이 있다.

위의 이유로 상호 배제를 위하여 인터럽트를 비활성화하는 것은 제한된 범위에서만 사용되어야 한다. 예를 들어 운영체제가 내부 자료 구조에 대한 원자적 연산을 위해 인터럽트를 비활성화할 수 있다. 또는 적어도 복잡하게 보이는 인터럽트 발생을 방지하기 위해서 운영체제가 인터럽트를 비활성화할 수 있다. 운영체제 내부에서는 신뢰라는 문제가 사라지기 때문에 운영체제가 특혜 받은 동작을 어떤 방식으로 처리하든 인터럽트를 비활성화하더라도 용인할 수 있다.

여담: Dekker와 Peterson의 알고리즘

1960대에 Dijkstra는 병행성 문제를 자신의 친구에게 이야기 했었고 그 중에 시어도러스 요세프 데커라는 수학자가 해답을 가져왔다 [Dij68]. 여기서 다루는 하드웨어 명령어와 운영체제의 지원을 받아 사용하는 방법들과는 달리 **Dekker의 알고리즘**은 load와 store 명령어만을 사용한다(이 명령어들이 원자적으로 동작한다고 가정한다. 실제로 초창기 하드웨어에서는 이 명령어들이 원자적으로 동작을 했었다).

데커의 접근 방법은 피터슨에 의해서 개선되었다 [Pet81]. 마찬가지로 load와 store가 사용되었는데, 임계 영역에 두 개의 쓰레드가 절대로 동시에 들어가지 못하도록 보장하는 개념이다. 여기에 두 개의 쓰레드를 위한 **Peterson의 알고리즘**을 소개한다. 한 번 다음의 코드를 읽고 이해할 수 있는지 보라. flag와 turn 변수가 무엇에 사용되는 변수인가?

```
1  int flag[2];
2  int turn;
3
4  void init() {
5      flag[0] = flag[1] = 0; // 1--> 쓰레드가 락을 획득하길 원함
6      turn = 0;          // 누구 차례 인가? (쓰레드 0 또는 1?)
7  }
8  void lock() {
9      flag[self] = 1;  // self: 호출한 쓰레드의 ID
10     turn = 1 - self; // 다른 쓰레드의 차례가 되도록 만듦
11     while ((flag[1-self] == 1) && (turn == 1 - self))
12         ;            // 회전
13 }
14 void unlock() {
15     flag[self] = 0;  // 원래의 의도를 취소함
16 }
```

어떤 이유에선가 특별한 하드웨어 지원 없이 락을 구현하는 것이 한동안 유행한 적이 있었다. 이론적으로 문제를 해결하는 법을 많이 다뤘었다. 물론, 이러한 접근이 하드웨어로부터 약간의 지원을 받으면 훨씬 쉽게 해결할 수 있다는 것을 알게 된 후로 필요가 없게 되었다. 실제로 초장기 멀티프로세서 시절부터 하드웨어가 지원하기 시작했다. 더 나아가 최근의 하드웨어에서는 앞선 알고리즘들은 사용할 수가 없다(완화된 메모리 일관성 모델 때문이다). 그렇기 때문에 이와 같은 기법들은 쓸모가 없어졌다. 많은 연구들이 역사의 뒤안길로 사라졌다.

31.6 Test-And-Set (Atomic Exchange)

멀티프로세서에서는 인터럽트를 중지시키는 것이 의미가 없기 때문에 시스템 설계자들은 락 지원을 위한 하드웨어를 설계하기 시작했다. 1960년대의 버로우 B5000(Burroughs B5000)과 같은 초기의 멀티프로세서 시스템이 하드웨어 지원 기능을 갖고 있다 [May82]. 오늘날에는 모든 시스템들이 이러한 지원 기능을 갖고 있으며, 단일 CPU 시스템에서도 이런 기능이 존재한다.

하드웨어 기법 중 가장 기본은 **Test-And-Set** 명령어 또는 **원자적 교체(atomic exchange)**라고 불리는 기법이다. `Test-And-Set`의 동작을 이해하기 위해서 간단한

플래그 변수로 락을 구현해 보자.

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> 락이 사용 가능함, 1 -> 락 사용 중
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // flag 변수를 검사(TEST) 함
10     ; // spin-wait (do nothing)
11     mutex->flag = 1; // 이제 설정(SET) 한다!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

〈그림 31.1〉 **첫 번째 시도: 간단한 플래그**

그림 31.1에서 사용한 아이디어는 간단하다. 간단한 변수를 사용하여 쓰레드가 락을 획득하였는지를 나타낸다. 임계 영역에 진입하는 첫 쓰레드가 `lock()`을 호출하여 플래그 값이 1인지 **검사**(**test**)하고(첫 번째 경우는 1이 아니다), 플래그의 값을 1로 **설정**(**set**)하여 이 쓰레드가 락을 **보유**(**hold**)하고 있다고 표시한다. 임계 영역에서 나오면 쓰레드가 `unlock()`을 호출하여 플래그 값을 초기화하여 락을 더 이상 보유하고 있지 않다고 표시한다.

만약 첫 번째 쓰레드가 임계 영역 내에 있을 때 다른 쓰레드가 `lock()`을 호출하면, 그 쓰레드는 `while` 문으로 **spin-wait**을 하며 처음 쓰레드가 `unlock()`을 호출하여 플래그를 초기화하기를 기다린다. 처음 쓰레드가 플래그를 초기화하면 대기하던 쓰레드는 `while` 문에서 빠져나와 플래그를 1로 설정하고 임계 영역 내로 진입한다.

이 코드에는 두 가지 문제가 있다. 하나는 정확성이고 다른 하나는 성능이다. 정확성의 문제는 병행 프로그래밍에 익숙해지면 쉽게 알아 볼 수 있을 것이다. 그림 31.2에 나와 있는 코드가 동작 중이라 가정하자. 시작할 때 `flag = 0` 이라고 하자.

적시에 인터럽트가 발생하면 두 쓰레드 모두 플래그를 1로 설정하는 경우가 생길 수 있어서 임계 영역에 두 쓰레드 다 진입할 수 있게 된다. 이러한 현상을 전문가들은 "잘못 만들었다."고 한다. 상호 배제 제공이라는 기본 요구 조건 보장에 실패하였기 때문이다.

성능의 문제는 이후에 다시 다루겠지만, 사용 중인 락을 대기하는 방법에 문제가 있다. **spin-wait**라는 방법을 사용하여 플래그의 값을 무한히 검사하는데, 이 방법은 다른 쓰레드가 락을 해제할 때까지 시간을 낭비한다. 이 방법은 단일 프로세서에서 특히 매우 손해가 크다. 락을 소유한 쓰레드조차 실행할 수 없기(적어도 문맥 교환이 있기 전까지는) 때문이다. 하나씩 정교한 기법을 만들면서 이런 낭비를 없애는 방법을 다루게 될 것이다.

Thread 1	Thread 2
`call lock()`	
`while (flag == 1)`	
인터럽트: 쓰레드 2로 교체	
	`call lock()`
	`while (flag == 1)`
	`flag = 1;`
	인터럽트: 쓰레드 1로 교체
`flag = 1;` // flag를 1로 설정 (똑같이!)	

〈그림 31.2〉 **문제: 가까이서 친밀하게 보기**

31.7 진짜 돌아가는 스핀 락의 구현

앞에서 다룬 예제의 아이디어는 좋았지만 하드웨어 지원 없이는 동작이 불가능하다. 다행히 어떤 시스템은 이 개념에 근간한 락 구현을 위한 명령어(어셈블리 명령어)를 제공한다는 것이다. 이 강력한 명령어는 시스템마다 다른 이름을 갖고 있다. SPARC에서는 `load/store unsigned byte` 동작을 하는 `ldstub` [WG00], x86에서는 원자적 교체 명령어인 `xchg`이다. 기본적으로 동일한 일을 수행하며 일반적으로 **Test-And-Set**라고 불린다. 다음의 C 코드의 일부를 사용하여 `TestAndSet`의 동작을 정의해 보자.

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // old_ptr 의 이전 값을 가져옴
3      *old_ptr = new;     // old_ptr 에 'new' 의 값을 저장함
4      return old;         // old의 값을 반환함
5  }
```

`TestAndSet` 명령어가 하는 동작은 다음과 같다. `ptr`이 가리키고 있던 예전의 값을 반환하고 동시에 new에 새로운 값을 저장한다. 여기서의 핵심은 이 동작들이 원자적으로 수행된다는 것이다. "test and set"라고 부르는 이유가 이전 값을 "검사(test)"하는(반환되는 값이며) 동시에 메모리에 새로운 값을 "설정(set)"하기 때문이다. 약간 더 강력해진 이 명령어만으로 그림 31.3에 보인 간단한 **스핀 락**(**spin lock**)을 만들 수 있다. 아니면 더 좋은 방법을 직접 생각해서 만들어 볼 수도 있겠다.

이 락이 동작하는 이유를 재확인하자. 처음 쓰레드가 `lock()`을 호출하고 다른 어떤 쓰레드도 현재 락을 보유하고 있지 않다고 하자. 그리고 현재의 `flag`의 값은 0이다. 이 쓰레드가 `TestAndSet(flag, 1)`을 호출하면 이 루틴은 `flag`의 이전 값인 0을 반환한다. `flag` 값을 검사한 쓰레드는 `while` 문에서 회전하지 않고 락을 획득할 수 있게 된다. 이 쓰레드는 `flag` 값을 1로 설정하여 락을 보유하고 있음을 표시한다. 이 쓰레드가 임계 영역 내의 동작을 끝마치면 `unlock()`을 호출하여 `flag`를 다시 0으로 변경한다.

생각해 볼 수 있는 두 번째 경우는 처음 쓰레드가 락을 획득하여 `flag` 값이 1인

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 은 락이 획득 가능한 상태를 표시, 1 은 락을 획득했음을 표시
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12     ; // 스핀(아무 일도 하지 않음)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

〈그림 31.3〉 **TestAndSet을 이용한 간단한 스핀 락**

팁 : 병행성을 악의적인 스케줄러로 생각하자

이 예제로 병행의 실행을 어떻게 이해해야 하는지 감이 생겼을 것으로 본다. 스스로를 **악의적인 스케줄러**라고 생각하고 접근해야 한다. 이 스케줄러는 동기화 함수의 동작에서 항상 최악의 시점에 인터럽트를 발생시킨다. 참 나쁜 스케줄러가 아닐 수 없다. 인터럽트가 거의 발생하지 않더라도, 발생 가능하다는 것을 보이는 것만으로도 특정 기법이 사용할 수 없다는 것을 보이기에 충분하다. 때로는 악의적으로 생각하는 것이 도움이 될 때도 있다!(최소한 가끔은)

상태이다. 두 번째 쓰레드가 `lock()`을 호출하여 `TestAndSet(flag, 1)` 루틴을 실행한다. 이번에는 `TestAndSet()`는 이미 락을 다른 쓰레드가 보유하고 있기 때문에 예전 값으로 1을 반환하는 동시에 `flag`의 값을 다시 1로 설정한다. 락을 보유하고 있는 쓰레드가 있는 한 `TestAndSet`는 계속 1을 반환한다. 두 번째 쓰레드는 락이 해제될 때까지 계속 `while` 문을 반복한다. 락을 보유하고 있는 쓰레드에 의해 `flag` 값이 0이 되면, 대기 중인 두 번째 쓰레드가 `TestAndSet()`를 호출하여 0을 받는 동시에 원자적으로 값을 1로 변경한다. 락을 획득하였으므로 임계 영역 내로 진입한다.

락의 값을 **검사**(**test**)하고 새로운 값으로 **설정**(**set**)하는 동작을 원자적 연산으로 만듦으로써 오직 하나의 쓰레드만 락을 획득할 수 있도록 만들었다. 이제 제대로 동작하는 상호 배제 함수를 만드는 방법을 배웠다.

지금 설명한 방법이 **스핀 락**으로 불리는 이유를 이제 이해할 수 있을 것이다. 이것이 가장 기초적인 형태의 락으로서, 락을 획득할 때까지, CPU 사이클을 소모하면서 회전한다. 단일 프로세서에서 이 방식을 제대로 사용하려면 **선점형 스케줄러**(**preemptive scheduler**)를 사용하여 한다. 선점형 스케줄러는 필요에 따라 다른 쓰레드가 실행될 수 있도록 타이머를 통해 쓰레드에 인터럽트를 발생시킬 수 있다. 선점형이 아니면, 단일 CPU에서 스핀 락의 사용은 불가능하다. 왜냐하면 `while` 문을 회전하며 대기하는 쓰레드가 CPU를 영원히 독점하기 때문이다.

31.8 스핀 락 평가

기본적인 스핀 락을 이해했으니 이전 장에서 명시한 기준에 의해 스핀 락의 효율을 이제 평가해 볼 수 있다. 락에서 가장 중요한 측면은 상호 배제의 **정확성**이다. 상호 배제가 가능하다면 스핀 락은 임의의 시간에 단 하나의 쓰레드만이 임계 영역에 진입할 수 있도록 한다. 제대로 동작하는 락이다.

그 다음의 항목은 **공정성**이다. 대기 중인 쓰레드들에 있어서 스핀 락은 얼마나 공정한가? 대기 중인 쓰레드에게 임계 영역에 진입할 기회가 주어진다는 것을 보장할 수 있겠는가? 여기서의 대답은 좋지 않다. 스픽 락은 어떠한 공정성도 보장해 줄 수 없다. 실제로 `while` 문을 회전 중인 쓰레드는 경쟁에 밀려서 계속 그 상태에 남아 있을 수 있다. 스핀 락은 공정하지 않으며 쓰레드가 굶주리게 만들 수 있다.

마지막 항목은 **성능**이다. 스핀 락을 사용할 때 지불해야 하는 비용은 무엇인가? 이것을 좀 더 자세히 분석하기 위해서 몇 가지 경우를 생각해 보기를 바란다. 첫째로 단일 프로세서를 사용할 때 락을 획득하기 위해 경쟁하는 경우와 둘째로 여러 프로세서에 쓰레드가 퍼져 있는 경우를 생각해 보자.

단일 CPU의 경우에 스핀 락이 갖는 성능 오버헤드는 상당히 클 수가 있다. 임계 영역 내에서 락을 갖고 있던 쓰레드가 선점된 경우를 생각해 보자. $N-1$개의 다른 쓰레드가 있다고 가정할 때 스케줄러가 락을 획득하려고 시도하는 나머지 쓰레드들을 하나씩 깨울 수도 있다. 이런 경우, 쓰레드는 할당받은 기간 동안 CPU 사이클을 낭비하면서 락을 획득하기 위해 대기한다.

반면에 CPU가 여러 개인 경우(쓰레드의 개수가 CPU의 개수와 대충 같다면) 스핀 락은 꽤 합리적으로 동작한다. 그런 이유는 다음과 같다. 쓰레드 A는 CPU 1에 쓰레드 B는 CPU 2에서 락을 획득하기 위해 경쟁 중이라고 해 보자. 만약 CPU 1에 실행 중인 쓰레드 A가 락을 획득한 후에 쓰레드 B가 획득하려고 시도했다고 하면 B는 CPU 2에서 기다린다. 임계 영역의 구간이 매우 짧다고 하면 락은 곧 획득 가능한 상태가 될 것이고 쓰레드 B는 락을 획득하게 된다. 다른 프로세서에서 락을 획득하기 위해 `while` 문을 회전하면서 대기하는 것은 그렇게 많은 사이클을 낭비하지 않기 때문에 효율적일 수 있다.

31.9 Compare-And-Swap

또 다른 하드웨어 기법은 SPARC의 **Compare-And-Swap**, x86에서는 **Compare-And-Exchange**가 있다. 그림 31.4에 C로 작성된 이 명령어의 의사 코드가 있다.

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int actual = *ptr;
3      if (actual == expected)
4          *ptr = new;
5      return actual;
6  }
```

〈그림 31.4〉 **Compare-And-Swap**

Compare-And-Swap 기법의 기본 개념은 ptr이 가리키고 있는 주소의 값이 expected 변수와 일치하는지 검사하는가 것이다. 만약 일치한다면 ptr이 가리키는 주소의 값을 새로운 값으로 변경한다. 불일치한다면 아무 것도 하지 않는다. 원래의 메모리 값을 반환하여 CompareAndSwap를 호출한 코드가 락 획득의 성공 여부를 알 수 있도록 한다.

Compare-And-Swap 방법을 사용하면 Test-And-Set 방법을 사용했을 때와 같은 방식으로 락을 만들 수 있다. 예를 들면 앞서 사용한 lock() 루틴을 다음과 같이 변경할 수 있다.

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3      ; // 회전
4  }
```

그 이후의 코드는 앞서 사용한 Test-And-Set에서 사용한 예제와 동일하다. 이 코드는 앞선 예와 상당히 유사하게 동작한다. flag 변수가 0인지 검사하고 만약 그렇다면 자동적으로 1로 바꾸어 락을 획득한다. 다른 쓰레드가 락을 보유하고 있는 중에 락을 획득하려고 시도하는 쓰레드는 락이 획득 가능한 상태가 될 때까지 while 문에서 회전하게 된다.

만약에 x86 기반에서 C에서 호출 가능한 형태의 Compare-And-Swap 방식을 보기 원한다면 다음의 코드가 도움이 될 것이다 [Soo05].

```
1  char CompareAndSwap(int *ptr, int old, int new) {
2      unsigned char ret;
3
4      // sete 명령어는 word가 아닌 `byte' 형으로 설정함
5      __asm__ __volatile__ (
6          " lock\n"
7          " cmpxchgl %2,%1\n"
8          " sete %0\n"
9          : "=q" (ret), "=m" (*ptr)
10         : "r" (new), "m" (*ptr), "a" (old)
11         : "memory");
12         return ret;
13 }
```

깨달았을 수도 있겠지만, CompareAndSwap 명령어는 TestAndSet 명령어보다 더 강력하다. **대기없는 동기화**(**wait-free synchronization**) [Her91]를 다룰 때 이 루틴이 갖는 능력을 알게 될 것이다. 단순히 스핀 락과 같은 식으로 만들어 사용하면 앞서 분석한 스핀 락과 다를 바 없이 동작한다.

31.10 Load-Linked 그리고 Store-Conditional

어떤 플랫폼은 임계 영역 진입 제어 함수를 제작하기 위한 명령어 쌍을 제공한다. MIPS 구조에서는 [Hei93] **load-linked**와 **store-conditional** 명령어를 앞뒤로 사용하여 락이나 기타 병행 연산을 위한 자료 구조를 만들 수 있다. 그림 31.5는 이 명령어들에 대한 C 의사 코드를 나타내었다. Alpha, PowerPC, 그리고 ARM에서도 유사한 명령어를 지원한다 [W09].

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no one has updated *ptr since the LoadLinked to this address) {
7          *ptr = value;
8          return 1; // 성공!
9      } else {
10         return 0; // 갱신을 실패함
11     }
12 }
```

〈그림 31.5〉 **Load-Linked와 Store-Conditional**

`load-linked`는 일반 로드 명령어와 같이 메모리 값을 레지스터에 저장한다. 실제 차이는 `store-conditional` 명령어에서 나타난다. `store-conditional` 명령어는 동일한 주소에 다른 스토어가 없었던 경우에만 저장을 성공한다. 저장이 성공하면, `load-linked`가 탑재했던 값을 갱신한다. 성공한 경우에는 `store-conditional`은 1을 반환하고 `ptr`이 가리키는 `value`의 값을 갱신한다. 실패한 경우에는 `ptr`이 가리키는 `value`의 값이 갱신되지 않고 0을 반환한다.

`load-linked`와 `store-conditional`을 사용하여 락을 만드는 방법을 생각해 보라. 답을 구했다면 다음의 정답 코드와 비교해 보자. 정답을 보기 전에 일단 만들어 보라! 해법은 그림 31.6에 나타나 있다.

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4          ; // 0이 될 때까지 스핀
5      if (StoreConditional(&lock->flag, 1) == 1)
6          return; // 1로 변경하는 것이 성공하였다면: 완료
7                  // 아니라면: 처음부터 다시 시도
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

〈그림 31.6〉 **Load-Linked와 Store-Conditional을 사용하여 락 만들기**

흥미 있는 부분은 `lock()` 부분의 코드이다. 쓰레드가 `while` 문을 돌며 `flag`가 0이 되기를 기다린다(락이 해제된 상태). 락이 획득 가능한 상태가 되면 쓰레드는 `store-conditional` 명령어로 락 획득을 시도하고 만약 성공한다면 쓰레드는 `flag` 값을 1로 변경한다. 그리고 임계 영역 내로 진입한다.

`store-conditional` 명령어가 실패할 수도 있다. 이 점이 중요하다. 쓰레드가 `lock()`을 호출하여 `load-linked`를 실행하고 락이 사용 가능한 상태이므로 0을 반환한다. `store-conditional` 명령어를 시도하기 전에 이 쓰레드는 인터럽트에 걸렸고 다른 쓰레드가 락 코드를 호출하고 `load-linked` 명령어는 실행하여 0을 반환 받은 후에 계속 진행한다고 해 보자. 이 시점에 두 쓰레드는 모두 `load-linked` 명령어를

팁: 더 적은 코드가 더 좋은 코드다 (Lauer's Law)

때로는 프로그래머들이 어떤 작업을 하기 위해서 몇 줄의 코드를 짰는지를 자랑하곤 한다. 정말 잘못된 것이다. 오히려 얼마나 더 적은 코드로 주어진 작업을 처리했는지를 자랑해야 한다. 짧고 간결한 코드는 언제나 환영받는다. 이해하기도 더 쉬울 뿐만 아니라 버그도 더 적다. Hugh Lauer가 운영체제의 구현에 대해 논하면서 이렇게 말했다. "어떤 사람에게 시간을 두 배 더 준다면 지금의 수준의 시스템을 그 반의 코드로 만들 수 있다." [Lau81] 이것을 보고 **Lauer's Law**라고 부르는데, 충분히 기억할 만한 말이다. 다음에 혹시라도 얼마나 많은 코드를 적어서 문제를 해결했다고 자랑하고 싶다면 다시 생각하시라. 그보다 더 좋은 것은 다시 돌아가서 코드를 다시 짜라. 그리고 분명하고 간결하게 코드를 만들어라.

실행하였고 각자가 `store-conditional`을 부르려는 상황이다. 이 명령어의 주요 기능은 오직 하나의 쓰레드만 `flag` 값을 1로 설정한 후에 락을 획득할 수 있도록 하는 것이다. `store-conditional`을 두 번째로 실행하는 쓰레드는 락 획득에 실패하고 (`load-linked`와 `store-conditional` 명령어 사이에 다른 쓰레드가 `flag`의 값을 갱신하였기 때문이다) 다음 기회를 기다려야 한다.

몇 년 전의 수업 시간에 David Capel이라는 학부생이 위의 코드를 Boolean 조건문을 사용하여 좀 더 간결하게 작성하는 방법을 제시했다. 다음의 코드가 동치인 인유를 한 번 파악해 보자. 문장이 정말 짧아졌다!

```
1  void lock(lock_t *lock) {
2     while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
3      ; // 회전
4  }
```

31.11 Fetch-And-Add

마지막 하드웨어 기반의 기법은 **Fetch-And-Add** 명령어로 원자적으로 특정 주소의 예전 값을 반환하면서 값을 증가시킨다. `Fetch-And-Add` 명령어의 C 언어로 된 의사 코드는 다음과 같다.

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

이 예제에서는 `fetch-and-add` 명령어를 사용하여 Mellor-Crummey와 Scott [MS91]이 제안한 **티켓 락**이라는 재미있는 것을 만들어보자. 락과 언락 코드는 그림 31.7에 나타난 것과 같다.

하나의 변수만을 사용하는 대신 이 해법에서는 티켓(ticket)과 차례(turn) 조합을 사용하여 락을 만든다. 기본 연산은 간단하다. 하나의 쓰레드가 락 획득을 원하면, 티켓 변수에 원자적 동작인 `fetch-and-add` 명령어를 실행한다. 결과 값은 해당 쓰레드의

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14     ; // 회전
15 }
16
17 void unlock(lock_t *lock) {
18     FetchAndAdd(&lock->turn);
19 }
```

〈그림 31.7〉 **티켓 락**

"차례"(myturn)를 나타낸다. 전역 공유 변수인 `lock->turn`을 사용하여 어느 쓰레드의 차례인지 판단한다. 만약 한 쓰레드가 (`myturn == turn`) 이라는 조건에 부합하면 그 쓰레드가 임계 영역에 진입할 차례인 것이다. 언락 동작은 차례 변수의 값을 증가시켜서 대기 중인 다음 쓰레드에게(만약 있다면) 임계 영역 진입 차례를 넘겨준다.

이전까지의 접근 방법과 이번 해법의 중요한 차이 중 하나는 모든 쓰레드들이 각자의 순서에 따라 진행한다는 것이다. 쓰레드가 티켓 값을 할당받았다면 미래의 어느 때에 실행되기 위해 스케줄되어 있다는 것이다(앞선 쓰레드들이 임계 영역을 지나가고 락을 해제한 후에 실행되도록 스케줄되어 있다). 이전까지의 해법들에서는 이러한 보장이 없었다. 예를 들어 `Test-And-Set`의 경우 다른 쓰레드들은 락을 획득/해제하더라도 어떤 쓰레드는 계속 회전만하고 있을 수 있다.

31.12 요약: 과도한 스핀

이제까지 소개한 하드웨어 기반의 락은 간단하고, 그리고 제대로 동작한다. 이것은 큰 장점이다. 하지만 때로는 이러한 해법이 효율적이지 않은 경우도 있다. 두 개의 쓰레드를 프로세서가 하나인 시스템에서 실행한다고 해 보자. 쓰레드 0이 임계 영역 내에 있어서 락을 보유한 상태에서 인터럽트에 걸렸다고 해 보자. 운도 참 없다. 두 번째 쓰레드인 쓰레드 1이 락을 획득하려고 시도하는데, 누군가 이미 갖고 있는 것을 알았다. 그래서 락을 대기하며 스핀하기 시작한다. 스핀하고, 조금 더 스핀한다. 마침내 타이머 인터럽트에 의해서 쓰레드 0이 다시 실행하게 되고 락을 해제한다. 마지막으로 쓰레드 1이 다시 실행하게 되면 그렇게 많이 스핀하지 않아도 락을 획득할 수 있게 된다. 쓰레드가 스핀 구문을 실행하면서 변경되지 않을 값을 변경되기 기다리며 시간만 낭비한다. N개의 쓰레드가 하나의 락을 획득하기 위해 경쟁하게 되면 상황은 더욱 심각해진다. $N-1$개의 쓰레드에 할당된 CPU 시간 동안, 비슷한 이유로 낭비하게 된다. 스핀하면서 락 해제를 기다린다. 이것이 우리가 다룰 다음 질문이다.

핵심 질문: 회전을 피하는 방법
어떻게 하면 스핀에 CPU 시간을 낭비하지 않는 락을 만들 수 있을까?

하드웨어의 지원으로만 이 문제를 해결할 수가 없다. 운영체제로부터의 지원이 추가로 필요하다. 어떻게 하면 이 문제를 해결할 수 있을지 생각해 보라.

31.13 간단한 접근법: 무조건 양보!

하드웨어의 지원을 받아 많은 것을 해결하였다. 동작이 검증된 락과 락 획득의 공정성(티켓 락을 사용한 경우)까지도 해결할 수 있었다. 하지만 아직도 문제가 남아 있다. 문맥 교환이 되어 쓰레드가 실행이 되었지만 이전 쓰레드가 인터럽트에 걸리기 전에 락을 이미 획득한 상태라서 그 쓰레드가 락을 해제하기를 기다리며 스핀만 무한히 하는 경우에 어떻게 해야 할 것인가?

첫 번째 시도는 단순하고 친근한 방법이다. 락이 해제되기를 기다리며 스핀해야 하는 경우 자신에게 할당된 CPU를 다른 쓰레드에게 양보하는 것이다. 또는 Al Davis가 말했듯이 "무조건 양보!"[Dic91]. 그림 31.8에 이 방법을 소개한다.

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // CPU를 양보함
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

〈그림 31.8〉 **Test-And-Set와 양보를 이용한 락**

이 방법에서는 운영체제에 자신이 할당받은 CPU 시간을 포기하고 다른 쓰레드가 실행될 수 있도록 하는 `yield()` 기법이 있다고 가정한다. 하나의 쓰레드는 실행 중(running), 준비(ready), 막힘(blocked)이라는 세 가지 상태가 있다. 양보(yield)라는 시스템 콜은 호출 쓰레드 상태를 실행 중(running) 상태에서 **준비**(ready) 상태로 변환하여 다른 쓰레드가 **실행 중** 상태로 전이하도록 한다. 결과적으로 양보 동작은 스케줄 대상에서 자신을 빼는 것(**deschedule**)이나 마찬가지이다.

단일 CPU 시스템에서 두 개의 쓰레드를 실행하는 예를 생각해 보자. 이 경우 양보를 기반으로 한 기법은 잘 동작한다. 만약 쓰레드가 `lock()`을 호출하였지만 다른 쓰레드가 락을 보유한 상황이었다면 이 쓰레드가 갖고 있던 CPU 시간을 단순히 양보하여 먼저의 쓰레드가 동작하여 임계 영역 밖으로 나올 수 있도록 하는 것이다. 간단한 예이지만 양보 기법이 제대로 동작한다는 것을 알 수 있다.

이번에는 100개 정도의 쓰레드들이 락을 획득하기 위해 경쟁하는 경우를 살펴보자. 이 경우 한 쓰레드가 락을 획득하고 선점 되었다. 이때에 나머지 99개의 쓰레드가 각자 `lock()`을 호출하고 다른 쓰레드가 락을 보유하고 있기에 CPU를 양보한다. 라운드 로빈(round-robin) 스케줄러를 사용하는 경우라면 락을 갖고 있는 쓰레드가 다시 실행되기까지 99개의 쓰레드가 실행하고 양보하는(run-and-yield) 패턴으로 동작하게 된다. 99개의 시간 간격을 낭비하게 되는 회전 방식보다는 좀 더 좋기는 하지만 아직도 이 기법을 사용하는 비용이 만만치 않다. 문맥 교환 비용이 상당하며 낭비가 많다.

더 안 좋은 것은 굶주림 문제는 전혀 손도 대지 못한 상황이라는 것이다. 나머지 쓰레드들은 임계 영역에 반복적인 출입을 하고 있는데, 어떤 쓰레드는 무한히 양보만 하고 있는 경우가 있을 수 있다. 이 문제를 직접적으로 다루는 해법이 필요하다.

31.14 큐의 사용: 스핀 대신 잠자기

이전 방법들의 근본 문제는 너무 많은 부분을 운에 맡긴다는 것이다. 스케줄러가 다음으로 실행될 쓰레드를 선정하는데, 만약 스케줄러가 안 좋은 선택을 한다면 실행되는 스케줄러는 회전을 하면서 대기를 하거나(처음에 제시한 방법), 또는 CPU를 즉시 양보해야 한다(두 번째로 제시한 방법). 어느 것이 되었든 낭비의 여지가 있고 쓰레드가 굶주리게 되는 상황을 막지 못한다.

어떤 쓰레드가 다음으로 락을 획득할지를 명시적으로 제어할 수 있어야 한다. 이를 위해서는 운영체제로부터 적절한 지원과 큐를 이용한 대기 쓰레드들의 관리가 필요하다.

간단한 설명을 위해 Solaris의 방식을 사용하겠다. 두 개의 호출 문이 있는데, `park()`는 호출하는 쓰레드를 잠재우는 함수이고 `unpark(threadID)`는 threadID로 명시된 특정 쓰레드를 깨우는 함수이다. 이 두 루틴은 이미 사용 중인 락을 요청하는 프로세스를 재우고 해당 락이 해제되면 깨우도록 하는 락을 제작하는 데 앞뒤로 사용할 수 있다. 그림 31.9에 나타난 코드를 사용하여 이 기법의 사용 예를 이해해 보자.

이 예제에서 두 가지 흥미로운 것을 해볼 것이다. 첫째, 앞서 배운 `Test-And-Set` 개념을 락 대기자 전용 큐와 함께 사용하여 좀 더 효율적인 락을 만들 것이다. 두 번째, 큐를 사용하여 다음으로 락을 획득할 대상을 제어하여 기아 현상을 피할 수 있도록 해보겠다.

그림 31.9에서 `guard` 변수를 사용하여 `flag`와 큐의 삽입과 삭제 동작을 스핀 락으로 보호하는 데 사용한다. 이 방법은 회전 대기를 완전히 배제하지는 못했다. 이 쓰레드는 락을 획득하거나 해제하는 과정에서 인터럽트에 걸릴 수 있다. 다른 쓰레드는 락의 해제를 기다리며 회전 대기하게 된다. 하지만, 회전 대기 시간은 꽤 짧다. 사용자가 지정한 임계 영역에 진입하는 것이 아니라 락과 언락 코드 내의 몇 개의 명령어만 수행하면 되기 때문이다. 그렇기 때문에 이 접근법은 합리적이다.

두 번째로 `lock()` 내에 추가된 동작이 있다. `lock()`을 호출하였는데, 다른 쓰레드가 이미 락을 보유했기 때문에 자신은 락을 획득할 수 없다면, `gettid()`를 호출하여 현재 실행 중인 쓰레드 ID를 얻고, 락 소유자의 큐에 자기 자신을 추가하고 `guard` 변수를 0으로 변경한 후에 CPU를 양보한다. 한 가지 질문을 하겠다. `park()` 호출

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15     ; // 회전하면서 guard 락을 획득
16     if (m->flag == 0) {
17         m->flag = 1; // 락을 획득함
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; // 회전하면서 guard 락을 획득
29     if (queue_empty(m->q))
30         m->flag = 0; // 락을 포기함: 누구도 락을 원치 않음
31     else
32         unpark(queue_remove(m->q)); // 락을 획득함 (다음 쓰레드를 위해!)
33     m->guard = 0;
34 }
```

〈그림 31.9〉 **큐, Test-And-Set, 양보와 깨우기를 사용한 락**

전이 아니라 호출 후에 `guard` 락을 해제하면 어떤 일이 생길까? 힌트: 안 좋은 일이 발생한다.

다른 쓰레드가 깨어났을 때 `flag` 변수가 0으로 설정되지 않는다는 사실을 깨달았을 것이다. 왜 그럴까? 이것은 오류가 아니고 필요에 의한 것이다! 쓰레드가 깨어날 때는 마치 쓰레드가 `park()`에서 리턴하는 것과 같이 보인다. 하지만 그 시점에서는 `guard` 락을 획득한 상태가 아니기 때문에 `flag` 변수의 값을 1로 변경하는 것을 시도조차 할 수 없다. 그렇기 때문에 락을 획득하려는 다음 쓰레드로 직접 전달한다. 그 사이에 `flag`는 0으로 바뀌지 않는다.

마지막으로 이 해법에서 `park()` 직전에 경쟁 조건이 발생한다는 것을 인지 했을 것이다. 이것을 인지하는 것은 사실 쉽지 않다. 한 쓰레드는 락이 사용 중이라 `park()` 문을 수행하려고 한다. 그 직전에 락 소유자한테 CPU가 할당된다면 문제가 발생할 수 있다. 예를 들어 락을 보유한 쓰레드가 그 락을 해제했다고 해 보자. 쓰레드가 자기 차례에 `park()`를 수행하면 (잠재적으로) 깨어날 방법이 없다. 이 문제는 **깨우기/대기 경쟁**(**wakeup/waiting race**)이라고도 불린다. 이 문제를 피하기 위해서는 뭔가 더 해야 한다.

Solaris는 이 문제를 세 번째 시스템 콜인 `setpark()`를 추가하여 해결하였다. 이 루틴은 `park()`를 호출하기 직전이라는 것을 표시한다. 만약 그때 인터럽트가 수행되고

`park()`가 실제로 호출되기 전에 다른 쓰레드가 `unpark()`를 먼저 호출한다면, 추후 `park()` 문은 잠을 자는 대신 바로 리턴이 된다. `lock()` 내에서 변경되는 부분은 그리 많지 않다.

```
1  queue_add(m->q, gettid());
2  setpark(); // 새로운 코드
3  m->guard = 0;
```

`guard` 변수의 역할을 커널에서 담당한는 것도 하나의 방법이다. 그런 경우 커널은 락 해제와 실행 중인 쓰레드를 큐에서 제거하는 동작을 원자적으로 처리하기 위해 주의를 해야 한다.

31.15 다른 운영체제, 다른 지원

좀 더 효율적인 락을 만들기 위한 운영체제 지원 내용을 살펴보았다. 다른 운영체제들도 유사한 기능을 지원하지만 세부적인 내용은 다르다.

Linux의 경우 **futex**라는 것을 지원한다. 이것은 Solaris의 인터페이스와 유사하지만 커널 내부와 관련이 깊다. 구체적으로, 각 `futex`는 특정 물리 메모리 주소와 연결이 되어 있으며 `futex`마다 커널 내부의 큐를 갖고 있다. 호출자는 futex를 호출하여 필요에 따라 잠을 자거나 깨어날 수 있다.

`futex`에는 두 개의 명령어가 제공된다. `futex_wait(address, expected)` 명령어는 address의 값과 expected의 값이 동일한 경우 쓰레드를 잠재운다. 같지 않다면 즉시 리턴한다. `futex_wake(address)` 명령어를 호출하면 큐에서 대기하고 있는 쓰레드 하나를 깨운다. Linux에서의 사용법은 그림 31.10에 나타내었다.

`nptl` 라이브러리의 `lowlevellock.h`(`gnu libc` 라이브러리의 일부)에서 발췌한 이 코드 [L09]는 매우 흥미롭다. 이 코드에서는 하나의 정수를 이용하여 락의 사용 중 여부와(최상위 비트를 사용하여), 대기자 수를 표현한다(나머지 비트를 사용하여). 만약 락이 음수라면 락이 사용 중(최상위 비트는 정수의 부호를 나타내기 때문)인 것을 나타낸다. 이 코드가 흥미로운 또 다른 이유는 경쟁이 없는 일반적인 경우에 대한 최적화 방법을 제시하기 때문이다. 단 하나의 쓰레드가 락을 획득하고 해제하는 경우라면 아주 약간만 일을 하도록 하였다(원자적으로 비트 단위의 `TestAndSet`로 락을 획득하고 원자적 덧셈을 하여 락을 해제한다). "실세계"에서 사용하는 락의 동작 메커니즘을 스스로 이해할 수 있는지 시도해 보라.

31.16 2단계 락

마지막으로 하나 더 살펴보자. Linux의 기법에서는 1960년 초에 있었던 Dahm 락 [May82]을 시작으로 지금까지 가끔씩 사용되던 오래된 기법의 정취를 느낄 수가 있다. 요즘은 **2단계 락**(**two-phase lock**)이라고 불린다. 2단계 락 기법은 락이 곧 해제될 것 같은 경우라면 회전 대기가 유용할 수 있다는 것에서 착안하였다. 첫 번째 단계에서는 곧 락을 획득할 수 있을 것이라는 기대로 회전하며 기다린다. 만약 첫 회전 단계에서 락을 획득하지

```
1  void mutex_lock (int *mutex) {
2      int v;
3      /* 31번째 비트가 이미 초기화되어 있다.  mutex를 이미획득했다. 바로 리턴한다. (이것이 빠르게 실행하는
           방법이다) */
4
5      if (atomic_bit_test_set (mutex, 31) == 0)
6          return;
7      atomic_increment (mutex);
8      while (1) {
9          if (atomic_bit_test_set (mutex, 31) == 0) {
10             atomic_decrement (mutex);
11             return;
12         }
13         /* 이제 대기해야 한다. 먼저, 우리가 관찰 중인 futex 값이
14            실제로 음수 인지 확인해야 한다(잠겨있는 상태인지). */
15         v = *mutex;
16         if (v >= 0)
17             continue;
18         futex_wait (mutex, v);
19     }
20 }
21
22 void mutex_unlock (int *mutex) {
23     /*  필요충분 조건으로 관심 대상의 다른 쓰레드가 없는 경우에 한해서
24         0x80000000를 카운터에 더하면 0을 얻는다. */
25     if (atomic_add_zero (mutex, 0x80000000))
26         return;
27
28     /* 이 mutex를 기다리는 다른 쓰레드가 있다면
29        그 쓰레드들을 깨운다. */
30     futex_wake (mutex);
```

〈그림 31.10〉 **Linux 기반의 futex 락**

못했다면 두 번째 단계로 진입하여 호출자는 잠에 빠지고 락이 해제된 후에 깨어나도록 한다. 앞서 본 Linux의 락은 이러한 형태를 갖는 락이지만 한 번만 회전한다. 일반화된 방법은 **futex**가 잠재우기 전에 일정 시간 동안 반복문 내에서 회전하도록 하는 것이다.

2 단계 락은 두 개의 좋은 개념을 사용하여 개선된 하나를 만들어 내는 하이브리드 방식의 일종이다. 물론, 이 기법이 좋은지는 하드웨어 환경이나, 쓰레드의 개수, 그리고 세부 작업량 등과 같은 많은 것들에 의존하기 때문에 확실하지 않다. 항상 그렇듯이, 모든 경우에 다 잘 동작하는 하나의 범용적 락을 만든다는 것은 상당히 어려운 일이다.

31.17 요약

방금 보인 기법은 락이 실제 어떻게 구현되었는지를 보여준다. 일부 하드웨어 지원(강력한 명령어의 형태)과 일부 운영체제의 지원(예, `park()`와 `unpark()`와 같은 형태)을 받아 락을 구현하였다. 물론, 세부적인 내용은 다르고 락을 실행하는 정확한 코드는 대체적으로 최적화가 되어 있다. 좀 더 구체적인 내용을 알아보고 싶다면 Solaris나 Linux의 코드를 살펴보기를 바란다 [L09; S09]. 현대의 멀티프로세서에서 락 사용 전략의 비교를 한 데이비드 외가 진행한 훌륭한 연구도 보기를 바란다 [DGT13].

참고 문헌

[DGT13] **"Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask"**
Tudor David, Rachid Guerraoui, and Vasileios Trigonakis
SOSP '13, Nemacolin Woodlands Resort, Pennsylvania, November 2013
하드웨어 기법을 사용하는 여러 방법들을 비교한 최근에 발표한 훌륭한 논문이다. 몇 해에 걸쳐 개발된 많은 개념들이 현대의 하드웨어에서 동작하는 것을 볼 수 있는 좋은 글이다.

[Dic91] **"Just Win, Baby: Al Davis and His Raiders"**
Glenn Dickey
Harcourt 1991
실제로 알 데이비스(Al Davis)와 그의 유명한 "이겨버려(just win)"라는 표현을 다룬 안 좋은 책도 나와 있다. 또는 어쩌면 이 책이 인용문에 대한 것이 아니라 알 데이비스와 침입자들에 대한 이야기일 수도 있다. 어떤 내용인지 읽어보는 것은 어떨까?

[Dij68] **"Cooperating sequential processes"**
Edsger W. Dijkstra
URL: `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`
이 분야의 초기 토론 강의용 논문 중 하나이다. Dijkstra가 처음 제시한 병행 연산 문제와 데커의 해법을 다룬다.

[Hei93] **"MIPS R4000 Microprocessor User's Manual"**
Joe Heinrich
Prentice-Hall, June 1993
URL: `http://cag.csail.mit.edu/raw/documents/R4400_Uman_book_Ed2.pdf`

[Her91] **"Wait-free Synchronization"**
Maurice Herlihy
ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 13, Issue 1, January 1991
병행 용 자료 구조를 만들기 위한 기법들을 소개할 때 기준이 되는 논문이다. 하지만 너무 복잡하기 때문에, 소개된 많은 개념들은 시스템 분야에서 인정받기까지 상당히 오랜 시간이 흘렀다.

[L09] **"glibc 2.9 (include Linux pthreads implementation)"**
URL: `http://ftp.gnu.org/gnu/glibc/`
nptl의 하위 디렉터리를 주요하게 살펴보기 바란다. 그곳에 현재 Linux가 pthread를 지원하기 위해 사용하는 대부분의 기법을 볼 수가 있다.

[Lau81] **"Observations on the Development of an Operating System"**
Hugh Lauer
SOSP '81, Pacific Grove, California, December 1981
초창기 PC용 운영체제인 파일럿 운영체제 개발에 대한 회고적 기록으로 꼭 읽어야 할 글이다. 재미있고 통찰력이 대단한 글이다.

[May82] **"The Architecture of the Burroughs B5000 20 Years Later and Still Ahead of the Times?"**
Alastair J.W. Mayer
URL: `www.ajwm.net/amayer/papers/B5000.html`

다음은 인용문이다. "특히 유용한 명령어는 RDLK (읽기 락, read-lock)이다. 분할하거나 나누어질 수 없는 연산으로 메모리 위치의 값을 읽고 값을 쓴다." RDLK는 최초라고 할 수 있을 만큼 초창기에 사용된 TestAndSet 기법이다. Dave Dahm이라는 공학자에게 공로를 돌려야 한다. 그는 Burroughs systems를 위해 이러한 것을 다수 개발하였고, 그 중에는 "Buzz lock"라고 부르는 스핀 락의 일종을 포함하여 "Dahm Locks"라고 불렸던 2단계 락도 있다.

[MS91] **"Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors"**
John M. Mellor-Crummey and M.L. Scott
ACM TOCS, Volume 9, Issue 1, February 1991
여러 락 알고리즘에 대한 조사 논문으로 탁월하고 빈틈이 없다. 하지만, 운영체제 지원에 대한 것은 없고 오직 현란한 하드웨어 명령어들만 나와 있다.

[Pet81] **"Myths About the Mutual Exclusion Problem"**
G.L. Peterson
Information Processing Letters, 12(3), pages 115.116, 1981
Peterson의 알고리즘을 여기서 만날 수 있다.

[S09] **"OpenSolaris Thread Library"**
URL: `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c`
Sun을 이제는 Oracle이 소유하고 있으니 무슨 일이 생길지는 모르겠지만 그래도 이 자료도 매우 흥미롭다. 이 자료의 위치를 가르쳐준 Mike Swift에게 감사를 표한다.

[Soo05] **"Guide to porting from Solaris to Linux on x86"**
Ajay Sood
April 29, 2005
URL: `http://www.ibm.com/developerworks/linux/library/l-solar/`

[W09] **"Load-Link, Store-Conditional"**
Wikipedia entry on said topic, as of October 22, 2009
URL: `http://en.wikipedia.org/wiki/Load-Link/Store-Conditional`
위키페디아의 글을 인용했다는 것이 믿겨지는가? 게으른 것 같은가? 여기서 최초로 이 정보를 얻었고 그렇기 때문에 참고 문헌으로 넣지 않는다는 것이 오히려 더 이상했다. 더 나아가 서로 다른 아키텍처의 명령어들을 정리해 놓았다. Alpha의 `ldl_l/stl_c`와 `ldq_l/stq_c`, PowerPC의 `lwarx/stwcx`, MIPS의 `ll/sc`, 그리고 ARM 버전 6과 그 이상의 `ldrex/strex`를 정리해 놓았다. 사실, 위키페디아는 상당히 놀라운 것이니 너무 뭐라고 하지 말자, ok?

[WG00] **"The SPARC Architecture Manual: Version 9"**
David L. Weaver and Tom Germond
SPARC International, San Jose, California, September 2000
URL: `http://www.sparc.org/standards/SPARCV9.pdf`
참고: Sparc에서 원자적 연산에 대해 더 자세히 알고 싶다면 다음의 링크를 참고하라. `http://developers.sun.com/solaris/articles/atomic_sparc/`

숙제

`x86.py` 는 경쟁 조건을 유발하거나 그런 상황을 회피하는 쓰레드 간의 상호 동작을 볼 수 있는 프로그램이다. README 파일에 이 프로그램의 동작 방법과 기본 입력 값에 대한 설명이 나와 있다. 다음의 질문들에 답해 보라.

문제

1. `x86.py`에 `-p flag.s`라는 플래그를 주고 실행할 준비를 하자. 단일 메모리 플래그를 사용하여 락을 구현하였다. 어셈블리 코드가 무엇을 하려고 하는지 알겠는가?
2. 기본 값을 사용하여 실행하면 `flag.s` 가 의도한 것처럼 동작하는가? 올바른 결과를 만들어 내는가? `-M`과 `-R` 플래그를 사용하여 변수와 레지스터의 값의 흐름을 살펴보라 (`-c`를 플래그를 사용하여 값을 확인할 수 있다). 코드가 실행되면 최종적으로 flag 변수가 갖는 값을 예측할 수 있겠는가?
3. `%bx` 레지스터의 값을 `-a` 플래그를 사용하여 변경해 보라. 두 개의 쓰레드를 사용하고 있다면 `-a bx=2,bx=2`를 사용하면 된다. 이 코드는 어떤 동작을 하는가? 위의 질문을 여기도 적용해 보면 답은 무엇인가?
4. `bx`의 값을 큰 수로 변경한 후 `-i` 플래그를 사용하여 인터럽트 발생 빈도를 조정해 보라. 어떤 값이 이상한 결과를 만들어 내는가? 정상적인 결과를 얻을 수 있는 값은 무엇인가?
5. 이제 `Test-And-Set.s` 프로그램을 살펴보자. 먼저, `xchg` 명령어를 사용하여 간단한 락 기법을 구현한 코드를 이해해 보도록 하자. 락을 획득하기 위해 작성된 코드는 어떻게 작성되었는가? 락을 해제하는 부분의 코드는 어떻게 작성되었는가?
6. 이제 인터럽트 간격(`-i`)을 변경해가면서 코드를 실행해 보자. 반복문이 여러 차례 수행될 수 있도록 해야 하는 것을 유의하자. 코드가 항상 의도한 대로 동작하는가? CPU를 비효율적으로 사용하는 경우가 가끔 발생하는가? 수치화할 수 있겠는가?
7. `-P` 플래그를 사용하여 락 코드를 테스트해 보자. 예를 들어, 첫 번째 쓰레드가 락을 획득하자마자 두 번째 쓰레드가 락 획득을 시도하게 스케줄을 짜보자. 올바르게 동작하는가? 무엇을 테스트해 보아야 하겠는가?
8. 이제 본문의 여담에서 다뤘던 피터슨의 알고리즘을 구현한 `peterson.s`의 코드를 살펴보자. 이해할 수 있는지 코드를 분석해 보라.
9. 이제 `-i`에 여러 값을 넣어서 코드를 실행해 보자. 어떤 차이점이 보이는가?
10. 코드가 동작하는 것을 "증명"하기 위해 스케줄(`-P` 플래그를 사용)을 제어 할 수 있겠는가?
11. 이제 `ticket.s`의 티켓 락 코드를 살펴보자. 본문에 나와 있는 코드와 동일한가?

12. 코드를 실행할 때 `-a bx=1000,bx=1000`라는 플래그를 사용해 보자. 이 플래그는 각 쓰레드가 임계 영역을 1000번 반복하도록 한다. 시간이 지남에 따라 어떻게 바뀌는지 살펴보자. 락을 기다리기 위해 회전하는 시간이 길어지는가?

13. 쓰레드를 추가해가면 코드의 동작이 어떻게 바뀌는가?

14. 이제 `yield` 명령어가 CPU의 사용권을 다른 쓰레드에 양보한다고 가정하고 `yield.s`를 살펴보자(실제로는 이 명령어는 운영체제의 기법이지만, 시뮬레이션을 간단하게 하기 위해서 이 동작을 위한 명령어가 있다고 하자). `test-and-set.s` 가 회전을 하면서 사이클을 낭비하지만 `yield.s`는 낭비하지 않는 시나리오를 찾아보자. 몇 개의 명령어가 절약되는가? 어떤 시나리오에서 이런 절약이 가능한가?

15. 마지막으로, `test-and-test-and-set.s`를 살펴보자. 이 락은 어떤 동작을 하는가? `test-and-set.s`과 비교하여 무엇을 절약할 수 있도록 해주는가?