35

병행성 관련 오류

수년 동안 병행성 관련 오류[1] 해결을 위해 연구자들이 엄청난 시간과 노력을 들였다. 대부분의 초기 연구는 **교착 상태**(**deadlock**)에 초점이 맞추어져 있었다. 이번 장에서는 좀 더 심도 있게 살펴보기로 한다 [CES71]. 최근의 연구들은 다른 종류의 병행성 버그들을 다루고 있다 (비 교착 상태 버그). 이번 장에서는 실제 코드들을 예로 사용하여 병행성 문제들을 살펴보고 어떤 문제들을 조심해야 하는지 보도록 하겠다. 우리가 이번 장에서 다룰 핵심 문제는 다음과 같다.

핵심 질문 : 일반적인 병행성 관련 오류들을 어떻게 처리하는가
병행성 버그는 몇 개의 전형적인 패턴을 갖고 있다. 튼튼하고 올바른 병행 코드를 작성하기 위한 가장 첫 단계는 어떤 경우들을 피해야 할지 파악하는 것이다.

35.1 오류의 종류

첫 번째 질문은 이것이다 : 복잡한 병행 프로그램에서 발생하는 병행성 오류들은 어떤 것들이 있는가? 이 질문에 대한 답을 구하는 것은 어렵지만, 다행히 다른 사람들이 이미 비슷한 작업을 해놓았다. 여기서는 Lu 등의 연구 내용을 기반으로 설명하겠다 [Lu+08]. 실제 상황에서 어떤 종류의 오류들이 발생하는지 이해하기 위해 그들은 많이 사용되는 병행 프로그램들을 자세히 분석하였다.

이 연구는 대표적인 오픈소스 프로그램 4개에 집중하였다. 유명한 DBMS인 MySQL, 웹 서버로 잘 알려진 Apache, 유명한 웹 브라우저인 Mozilla, 그리고 실제 사람들이 사용하고 있는 무료 버전의 MS 오피스 모음인 OpenOffice를 분석하였다. 저자들은 연구에서 이들 코드에서 발견된 병행성 관련 오류에 대해 정량적 분석을 수행하였다. 이 연구를 통해 완성도 높은 코드에서 발생하는 오류의 특성을 쉽게 이해할 수 있다.

그림 35.1은 Lu와 그의 동료들이 정리한 오류의 통계이다. 그림에서 보는 바와 같이 총 105개의 오류가 있고 대부분(74개)의 오류는 교착 상태와 무관한 오류였다. 나머지 31

1) 역자 주 : 여기서의 "오류"는 fault나 error가 아니고 프로그램 bug를 일컫는다.

개의 오류들이 교착 상태 관련 오류였다. 각 응용 프로그램의 오류의 수에 대한 결과에서 알 수 있는 것은 OpenOffice는 총 8개의 병행성 오류가 있었고 Mozilla는 거의 60개의 관련 오류가 있었다.

응용 프로그램	하는 일	비 교착 상태	교착 상태
MySQL	데이터베이스 서버	14	9
Apache	웹 서버	13	4
Mozilla	웹 브라우저	41	16
OpenOffice	오피스 모음	6	2
총 계		74	31

〈그림 35.1〉 **현대 응용 프로그램들의 오류들**

이제 이런 종류의 오류(비 교착 상태와 교착 상태)들에 대해서 좀 더 자세히 알아보도록 하겠다. 먼저 비 교착 상태 오류를 살펴보기 위해서 이 연구에서 나온 예제들을 사용할 것이다. 교착 상태 오류들을 다룰 때에는 이 분야에서 오랫동안 다루어 온 예방(prevention), 회피(avoidence), 또는 교착 상태를 해결/관리하는 연구들을 논의하겠다.

35.2 비 교착 상태 오류

Lu의 연구 결과를 따르면 비 교착 상태 오류가 병행성 관련 오류의 과반수를 차지한다. 그것들은 어떤 종류일까? 어떻게 발생되는가? 어떻게 해결할 수 있을까? 이제 Lu 등이 발견한 비 교착 상태 오류의 분류 중 대표적인 두 종류의 오류인 **원자성 위반**(**atomicity violation**) 오류와 **순서 위반**(**order violation**) 오류를 살펴보겠다.

원자성 위반 오류

첫 번째로 만나게 되는 문제는 **원자성 위반**이라고 알려져 있다. 여기 MySQL에서 발견한 간단한 예제가 있다. 계속 읽기 전에 오류를 한 번 찾아보자. 시작!

```
1  Thread 1::
2  if (thd->proc_info) {
3      ...
4      fputs(thd->proc_info, ...);
5      ...
6  }
7
8  Thread 2::
9  thd->proc_info = NULL;
```

이 예제에서 `thd` 자료 구조의 `proc_info` 필드를 두 개의 다른 쓰레드가 접근한다. 첫 번째 쓰레드는 그 값이 `NULL`인지 검사하고 값을 출력한다. 두 번째 쓰레드는 값을 `NULL`로 설정한다. 첫 번째 쓰레드가 검사를 완료한 후 `fputs`를 호출하기 전에 인터럽

트로 인해서 두 번째 쓰레드가 그 사이에 실행될 수가 있다. 두 번째 쓰레드가 실행되면 필드의 값을 NULL로 설정하기 때문에 fputs 함수는 NULL 포인터 역참조를 하게 되어 프로그램은 크래시될 것이다.

Lu 등이 기술한 원자성 위반에 대한 정의는 이렇다. "다수의 메모리 참조 연산들 간에 있어 예상했던 직렬성(serializability)이 보장되지 않았다!(즉, 코드의 일부에 원자성이 요구되었으나, 실행 시에 그 원자성이 위반되었다)" 어느 부분일까? 지금 예제 코드는 proc_info 필드의 NULL 값 검사와 fputs() 호출 시 proc_info 를 인자로 사용하는 동작이 원자적으로 실행되는 것(*atomicity assumption*)을 가정했다(Lu의 말을 인용하였다). 이 가정이 깨지면, 코드는 의도한 대로 동작하지 않는다. 가정은 당연히 성립하지 않는다.

이러한 문제의 수정은 대부분(항상은 아니지만)의 경우 아주 단순한다. 코드를 어떻게 수정하면 될까?

락을 추가하여 어느 쓰레드든 proc_info 필드 접근 시, proc_info_lock 이라는 락 변수를 획득토록 한다. 물론, 이 자료 구조를 사용하는 다른 모든 코드들도 락으로 보호해야 한다.

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6      ...
7      fputs(thd->proc_info, ...);
8      ...
9  }
10 pthread_mutex_unlock(&proc_info_lock);
11
12 Thread 2::
13 pthread_mutex_lock(&proc_info_lock);
14 thd->proc_info = NULL;
15 pthread_mutex_unlock(&proc_info_lock);
```

순서 위반 오류

Lu 등이 발견한 또 다른 비 교착 상태 오류는 **순서 위반**이다. 간단한 예제가 있다. 마찬가지로 다음 코드에서 어디에 오류가 있는지 찾아보자.

```
1  Thread 1::
2  void init() {
3      ...
4      mThread = PR_CreateThread(mMain, ...);
5      ...
6  }
7
8  Thread 2::
9  void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13 }
```

이 코드에서 쓰레드 2의 코드는 mThread 변수가 이미 초기화(그리고 NULL이 아닌

것으로)가 된 것을 가정하고 있다. 하지만, 만약 쓰레드 1이 먼저 실행되지 않았다면 쓰레드 2는 NULL 포인터를 사용하기 때문에 크래시될 것이다(mThread의 값이 초기에 NULL이었다고 가정하였다. 그게 아니라면 쓰레드 2가 임의의 메모리 주소를 접근하게 되어 더 이상한 일이 발생한다. 차라리 크래시가 낫다).

순서 위반의 정의는 다음과 같다. "두 개의(그룹의) 메모리 참조 간의 순서가 바뀌었다(즉, A가 항상 B보다 먼저 실행되어야 하지만 실행 중에 그 순서가 지켜지지 않았다)." [Lu+08]

이러한 오류를 수정하는 방법은 순서를 강제하는 것이다. 앞에서 자세히 논의했던 것처럼, 이러한 종류의 동기화에는 **컨디션 변수**가 잘 맞는다. 이미 말하기 전에 알았겠지만 말이다. 예제 코드는 다음과 같이 재작성 될 수 있다.

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit = 0;
4
5  Thread 1::
6  void init() {
7      ...
8      mThread = PR_CreateThread(mMain, ...);
9
10     // 쓰레드가 생성되었다는 것을 알리는 시그널 전달...
11     pthread_mutex_lock(&mtLock);
12     mtInit = 1;
13     pthread_cond_signal(&mtCond);
14     pthread_mutex_unlock(&mtLock);
15     ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20     ...
21     // 쓰레드가 초기화되기를 대기...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

수정된 코드에서는 mtLock이라는 락, 그에 대한 컨디션 변수 mtCond, 그리고 상태 변수 mtInit을 추가하였다. 초기화 코드가 실행되면 mtInit의 상태를 1로 설정하고 초기화를 완료했다고 시그널을 발생시킨다. 만약 쓰레드 2가 이 시점 전에 실행된다면 상태가 변경되기를 대기한다. 이후에 다시 쓰레드 2가 실행되면 상태 값 초기화 여부를 검사한 후(즉, mtInit 이 1로 설정되었는지를 검사하고), 올바르게 계속 진행한다. mThread 자체를 상태 변수로 사용할 수도 있겠지만 간단함을 위해서 그렇게 하지 않았다. 쓰레드 간의 순서가 문제가 된다면 컨디션 변수(또는 세마포어)를 사용하여 해결할 수 있다.

비 교착 상태 오류: 정리

Lu 등이 연구한 비 교착 상태 오류의 대부분(97%)은 원자성 또는 순서 위반에 대한 것이었다. 이러한 오류 패턴들을 유의하면 관련 오류들을 좀 더 줄일 수 있다. 더 나아가, 자동화된 코드 검사 도구들이 개발될수록 비 교착 상태 오류들에 초점을 맞추게 될 것이다. 비 교착 상태 오류가 전체 오류 분포에서 많은 비중을 차지하기 때문이다.

하지만, 모든 오류가 예제에서 봤던 것처럼 쉽게 수정될 수 있는 것은 아니다. 어떤 것들은 프로그램 동작에 대한 심도 있는 이해, 방대한 코드에 대한 이해와 자료 구조 재수정이 필요하다. 더 자세한 내용은 Lu 등이 쓴 훌륭한(그리고 읽을 만한) 논문을 읽어보라.

35.3 교착 상태 오류

앞서 다룬 병행성 관련 오류 외에 복잡한 락 프로토콜을 사용하는 다수의 병행 시스템에서 **교착 상태**(**deadlock**)라는 고전적 문제가 발생한다. 예를 들어 락 L1을 갖고 있는 쓰레드 1 이 또 다른 락 L2를 기다리는 상황에서 불행하게도 락 L2를 갖고 있는 쓰레드 2가 락 L1이 해제되기를 기다리고 있을 때 교착 상태가 발생한다. 교착 상태가 발생할 가능성이 있는 코드를 다음에 나타내었다.

```
Thread 1:          Thread 2:
lock(L1);          lock(L2);
lock(L2);          lock(L1);
```

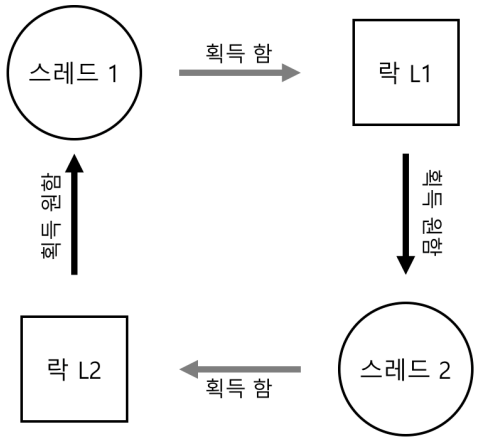
이 코드에서는 교착 상태가 발생할 수 있다. 발생하는 경우를 살펴보자. 쓰레드 1이 락 L1을 획득하고 난 후에 문맥 교환이 발생하여 쓰레드 2가 실행한다. 그때, 쓰레드 2가 락 L2를 획득하고 락 L1을 획득하려고 시도한다. 그러면 교착 상태가 발생한다. 각 쓰레드가 상대방이 소유하고 있는 락을 대기하고 있기 때문에 누구도 실행할 수 없게 된다. 그림 35.2에 도표로 나타내었다. 그래프에서 **사이클**(**cycle**)의 존재는 교착 상태 발생 가능성을 의미한다.

이 그림이 문제를 명확히 해줄 것이다. 교착 상태를 방지하기 위해서는 어떻게 코드를 작성해야 할까?

교착 상태는 왜 발생하는가

앞서 본 간단한 교착 상태의 상황은 손쉽게 막을 수 있겠다고 생각할 수 있다. 예를 들어 쓰레드 1과 2가 락을 같은 순서로 획득한다면 교착 상태는 절대 발생하지 않는다. 그러면 교착 상태는 왜 발생하는가?

한 가지 이유는 코드가 많아지면서 구성 요소 간에 복잡한 의존성이 발생하기 때문이다. 운영체제를 생각해 보자. 가상 메모리 시스템이 디스크 블럭을 가져오기 위해 파일 시스템을 접근하는 경우가 있다. 파일 시스템은 디스크 블럭을 메모리에 탑재하기 위해 메모리 페이지를 확보해야 하고, 이를 위해 가상 메모리 시스템에 접근한다. 코드 상에서



〈그림 35.2〉 **교착 상태 의존성 그래프**

핵심 질문: 교착 상태를 어떻게 다룰 것인가
시스템을 어떻게 개발해야 교착 상태를 예방하여 회피하거나 최소한 감지하고 회복할 수 있을까? 현대의 시스템에도 있는 실제의 문제인가?

자연스럽게 존재하는 순환 의존성이 교착 상태를 야기시키는 것을 방지하기 위해서 대형 시스템의 락 사용 전략의 설계는 매우 신중해야 한다.

또 다른 이유는 **캡슐화**(**encapsulation**)의 성질 때문이다. 소프트웨어 모듈화가 개발을 쉽게 하기 때문에 소프트웨어 개발자들은 상세한 구현 내용은 감추라고 교육받았다. 하지만, 모듈화와 락은 잘 조화되지 않는다. Jula 등이 지적했듯이 전혀 문제없어 보이는 인터페이스도 교착 상태를 발생시킨다 [Jul+08]. 예를 들어 자바의 Vector 클래스에서 `AddAll()` 메소드를 생각해 보자. 이 루틴은 다음과 같은 형식으로 호출될 수 있다.

```
Vector v1, v2;
v1.AddAll(v2);
```

이 메소드는 멀티 쓰레드에 안전해야 하기 때문에 내부적으로는 v1에 더해지는 벡터뿐만 아니라 인자로 전달되는 v2에 대한 락도 같이 획득해야 한다. 이 루틴은 v2의 내용을 v1에 더하기 위해서 임의의 순서로 말한 락들을 획득하는데, 여기서는 v1을 먼저 획득하고 v2를 획득한다고 하자. 어떤 쓰레드가 `v2.AddAll(v1)`을 거의 동시에 호출하면 교착 상태 발생 가능성이 있다. 이 모든 상황은 호출한 응용 프로그램 모르게 진행된다.

교착 상태 발생 조건

교착 상태가 발생하기 위해서는 네 가지 조건이 충족되어야 한다 [CES71].

- **상호 배제(Mutual Exclusion):** 쓰레드가 자신이 필요로 하는 자원에 대한 독자적인 제어권을 주장한다(예, 쓰레드가 락을 획득함).
- **점유 및 대기(Hold-and-wait):** 쓰레드가 자신에게 할당된 자원(예: 이미 획득한 락)을 점유한 채로 다른 자원(예: 획득하고자 하는 락)을 대기한다.
- **비 선점(No preemption):** 자원(락)을 점유하고 있는 쓰레드로부터 자원을 강제적으로 빼앗을 수 없다.
- **환형 대기(Circular wait):** 각 쓰레드는 다음 쓰레드가 요청한 하나 또는 그 이상의 자원(락)을 갖고 있는 쓰레드들의 순환 고리가 있다.

이 네 조건 중에 하나라도 만족시키지 않는다면 교착 상태는 발생하지 않는다. 먼저 교착 상태를 *예방할* 수 있는 기술들을 먼저 살펴보자. 각 전략들은 위의 조건들이 발생하는 것을 막는다. 그리고 그 전략이 교착 상태를 다루는 방법 중에 하나이다.

교착 상태의 예방

순환 대기(Circular Wait)

아마도 가장 실용적인 교착 상태 예방 기법은 (그리고 자주 사용되는 방법이기도 함) 순환 대기가 절대 발생하지 않도록 락 코드를 작성하는 것이다. 가장 간단한 방법은 락 획득을 하는 **전체 순서(total ordering)**를 정하는 것이다. 예를 들어 L1과 L2라는 두 개의 락만이 시스템에 존재하면 L1을 무조건 L2 전에 획득하도록 하면 교착 상태를 피할 수 있다. 이 순서를 따르면 순환 대기는 발생하지 않고 따라서 교착 상태도 발생하지 않는다.

물론, 좀 더 복잡한 시스템의 경우, 두 개 이상의 락이 존재할 것이고 전체 락의 요청 순서를 정의하는 것이 어려울 수 있다(또는 불필요할 수 있다). 교착 상태를 피하기 위해 **부분 순서(partial ordering)**를 제공하는 것이 락 획득 구조를 만드는 데 유용할 것이다. Linux의 메모리 매핑 코드가 부분 순서를 제공받아 락을 획득하는 방식에 대한 좋은 예다 [Tmo94]. 소스 코드의 상단의 주석을 보면 열 개의 서로 다른 그룹으로 묶여 있는 락과 그에 대한 획득 순서를 볼 수 있다. 그 순서에는 "`i_mmap_mutex` 전에 `i_mutex`"를 획득해야 한다는 간단한 것부터 좀 더 복잡한 "`mapping->tree_lock` 전에 `swap_lock`, 그 전에 `private_lock`, 그리고 그 전에 `i_mmap_mutex`"과 같은 순서도 있다.

전체 또는 부분 순서를 제공하기 위해서는 세심하게 락 획득 전략을 설계해야 한다. 더 나아가 순서라는 것은 단순히 관례이기 때문에 숙련되지 않은 개발자들이 이 관례를 무시하고 코드를 개발할 경우, 교착 상태가 발생할 수 있다. 마지막으로 락의 순서를 정의하기 위해서는 코드와 다양한 루틴 간의 상호 호출 관계를 이해해야 한다. 작은 실수라 할지라도 "D"로 시작하는 문제[2]를 만날 수 있게 된다.

2) 힌트: "D"는 Deadlock(교착 상태)의 앞글자이다.

팁: 락 주소를 사용하여 락 요청 순서 강제하기

어떤 경우엔 함수가 두 개 또는 그 이상의 락을 획득해야 하기 때문에 주의하지 않으면 교착 상태가 발생할 수 있다. `do_something(mutex_t *m1, mutex_t *m2)`과 같이 호출되는 함수가 있다고 하자. 이 코드가 m1을 m2 전에(또는 그 반대 순서로) 항상 획득한다면 교착 상태가 될 수 있다. 왜냐하면 한 쓰레드가 `do_something(L1, L2)`이라고 호출하고 다른 쓰레드가 `do_something(L2, L1)`을 호출할 수도 있기 때문이다.

이러한 경우를 피하기 위해서 현명한 개발자라면 주소의 값을 사용하여 락 획득의 순서를 정하기도 한다. 오름차순이나 내림차순으로 주소를 정렬하여 락의 획득 순서를 정하면 `do_something()` 문장에 인자를 어떤 순서로 넣어 호출하든 락의 획득 순서는 변하지 않게 된다. 코드는 다음과 비슷한 형태를 띤다.

```
if (m1 > m2) { // 락을 주소의 내림차순으로 획득하기
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
// 코드는 m1 != m2를 가정함 (서로 같은 락이 아님)
```

이 간단한 기법을 사용하면 개발자는 멀티 락 획득 상황에서 간단하고 효율적으로 교착 상태를 방지할 수 있다.

점유 및 대기(Hold-and-Wait)

교착 상태가 발생하는 조건인 점유 및 대기는 원자적으로 모든 락을 단번에 획득하도록 하면 예방할 수 있다. 실제로는 다음과 같은 방법을 사용할 수 있다.

```
1  lock(prevention);
2  lock(L1);
3  lock(L2);
4  ...
5  unlock(prevention);
```

이 코드에서는 제일 먼저 prevention 락을 획득하여, 락을 획득하는 과정 중에 쓰레드의 문맥 교환이 발생하는 것을 방지하고, 결과적으로 교착 상태의 발생 가능성을 차단한다. 쓰레드가 락을 획득하려면 전역 prevention 락을 먼저 획득해야 한다. 다른 쓰레드가 L1과 L2를 다른 순서로 획득하려고 한다하더라도 괜찮다. 왜냐하면 그 쓰레드가 prevention 락을 이미 획득한 후에 나머지 락을 요청하기 때문이다.

이 해법은 문제점이 많다. 먼저와 같이 캡슐화와 관련된 사항이다. 필요한 락들을 정확히 파악해야 하고 그 락들을 미리 획득해야 하기 때문이다. 락이 실제 필요할 때 요청하는 것이 아니라 미리 모든 락을 (단번에) 획득하기 때문에 병행성이 저하되는 문제도 있다.

비선점(No Preemption)

일반적으로 락을 해제하기 전까지는 락을 보유하고 있는 것으로 보기 때문에 여러 락을 획득하는 것에는 문제의 소지가 있다. 왜냐하면 락을 이미 보유하고 있는 채로 다른 락을 대기하기 때문이다. 많은 쓰레드 라이브러리들은 이러한 상황을 피할 수 있도록 유연한 인터페이스 집합을 제공한다. `trylock()` 루틴의 경우 (획득 가능하다면) 락을 획득하거나 현재 락이 점유된 상태이니 락을 획득하기 원하면 나중에 다시 시도하라는 것을 알리는 -1 값을 리턴한다.

이 인터페이스 (`trylock()`)를 이용하면 교착 상태 가능성이 없고 획득 순서에 영향을 받지 않는 락 획득 방법을 만들 수 있다.

```
1 top:
2     lock(L1);
3     if (trylock(L2) == -1) {
4         unlock(L1);
5         goto top;
6     }
```

다른 쓰레드가 같은 프로토콜을 사용하면서 락을 다른 순서(L2 먼저 L1 그 다음)로 획득하려고 해도 역시 교착 상태는 발생하지 않는다. 그렇지만 **무한반복(livelock)**이라는 새로운 문제가 생긴다. 두 개의 쓰레드가 이 순서대로 시도하기를 반복하면서 락 획득에 실패하는 것도 (대체적으로 그럴 리는 없겠지만) 가능하다. 두 쓰레드 모두 이 코드를 반복 실행하겠지만 (그래서 교착 상태는 아니지만), 실제 진척이 있는 것은 아니기 때문에 이름대로 무한반복의 상황이다. 무한반복의 문제에 대한 해법도 역시 존재한다. 예를 들면 반복문에 지연 시간을 무작위로 조절하는 것이다. 그러면 경쟁하는 쓰레드 간의 반복 간섭 확률을 줄일 수 있다.

이 해법에 대해 마지막으로 짚고 넘어가야 할 것이 있다. 이 해법은 `trylock()` 방식의 어려운 부분은 다루지 않고 있다. 첫 번째 문제는 캡슐화이다. 만약 사용하려는 락이 호출되는 루틴 깊숙한 곳에 존재한다면 처음 부분으로 되돌아가도록 구현하는 것이 쉽지 않다. 만약에 코드가 실행 과정에서 (L1이 아닌 다른) 자원을 획득하였다면, 그 자원 역시 반납해야 한다. 예를 들어 L1 획득 후 코드에서 메모리 영역을 할당하였다면, L2 획득 실패 시에 처음으로 돌아가서 전체 순서를 다시 시작하기 전에 할당받았던 메모리도 같이 반납을 해야 한다. 하지만, 제한된 경우에만 (예: 앞서 언급했던 자바 벡터 메소드) 이러한 접근이 제대로 동작할 것이다.

상호 배제(Mutual Exclusion)

마지막 예방 기법은 상호 배제 자체를 없애는 방법이다. 일반적 코드는 모두 임계 영역을 포함하고 있기 때문에 어려운 일이다. 그러면 어떻게 해야 할까?

Herlihy는 **대기없는(wait-free)** 자료 구조를 고안했다 [Her91]. 그의 생각은 간단하다. 명시적 락이 필요 없는 강력한 하드웨어 명령어를 사용하여 자료 구조를 만들면 된다는 내용이다.

간단한 예제로 다음과 같이 동작하는 Compare-And-Swap 명령어를 가정해 보자. 이 명령은 하드웨어가 지원하는 원자적 명령어를 다룰 때 살펴보았다.

```
1  int CompareAndSwap(int *address, int expected, int new) {
2      if (*address == expected) {
3          *address = new;
4          return 1; // 성공
5      }
6      return 0;      // 실패
7  }
```

어떤 한 값을 원자적으로 임의의 크기만큼 증가하는 경우를 생각해 보자.

```
1  void AtomicIncrement(int *value, int amount) {
2      do {
3        int old = *value;
4      } while (CompareAndSwap(value, old, old + amount) == 0);
5  }
```

락을 획득하여 값을 갱신한 후에 락을 해제하는 대신, 이 코드에서는 Compare-And-Swap 명령어를 사용하여 값에 새로운 값을 갱신하도록 반복적으로 시도한다. 이와 같은 방식을 사용하면 락을 획득할 필요가 없으며 교착 상태가 발생할 수도 없다(무한반복은 여전히 발생 가능성이 있기는 하다).

좀 더 복잡한 리스트에 삽입 예제를 살펴보자. 리스트 헤드에 개체를 삽입 하는 코드이다.

```
1  void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
4      n->value = value;
5      n->next = head;
6      head = n;
7  }
```

간단한 삽입문을 실행하는 이 코드가 만약 여러 쓰레드에 의해 "동시에" 호출이 되면 경쟁 조건(race condition)이 발생된다(왜 그런지 이유를 찾아보자). 삽입문 앞뒤에 락의 획득과 해제 코드를 두어 해결하는 방법이 있기는 하다.

```
1  void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
4      n->value = value;
5      lock(listlock);    // 임계 영역의 시작
6      n->next = head;
7      head = n;
8      unlock(listlock); // 임계 영역의 끝
9  }
```

이 해법에서는 전형적인 방식으로 락을 사용하고 있다[3]. 단순히 Compare-And-Swap 명령어를 사용하여 대기 없이 삽입 명령어를 처리해 보자. 다음과 같은 방법이 있다.

```
1  void insert(int value) {
2      node_t *n = malloc(sizeof(node_t));
3      assert(n != NULL);
```

3) 현명한 독자라면 락을 `insert()`문에 진입할 때 락을 획득했으면 되는데 왜 락을 늦게 획득하였는지 질문할 것이다. 그런데 그 방법이 대체적으로 맞는 이유를 현명한 독자가 알아 낼 수 있겠는가?

```
4       n->value = value;
5       do {
6           n->next = head;
7       } while (CompareAndSwap(&head, n->next, n) == 0);
8   }
```

이 코드는 `next` 포인터가 현재의 헤드를 가리키도록 갱신하고 새로 생성된 노드는 리스트의 헤드가 되도록 동작하고 있다. 이 코드를 처리하는 도중 만약 어떤 쓰레드가 새로운 헤드를 성공적으로 추가하였다면, 이 Compare-And-Swap는 실패한다. 쓰레드는 삽입과정을 재시도한다.

유용한 리스트를 만드는 것에는 삽입 동작 이외에 여러 가지가 필요하다. 대기없는 방식으로 삽입과 삭제 그리고 검색을 할 수 있도록 리스트를 만드는 것은 더욱 쉽지 않다. 만약 이 분야가 흥미로워 보인다면, 대기없는 동기화와 관련된 문서들이 많으니 읽어보기 바란다.

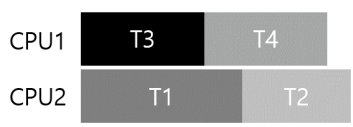
스케줄링으로 교착 상태 회피하기

어떤 시나리오에서는 교착 상태를 예방하는 대신 **회피**하는 것이 더 유용할 때가 있다. 회피하기 위해서는 실행 중인 여러 쓰레드가 어떤 락을 획득하게 될 것인지에 대해 전반적으로 파악하고 있어야 하며 그것을 바탕으로 쓰레드들을 스케줄링하여 교착 상태가 발생하지 않도록 그때그때 보장한다.

예를 들어 쓰레드 네 개가 프로세서 두 개에서 스케줄링된다고 해 보자. 그리고 추가로 쓰레드 1(T1)이 `L1`과 `L2` 락을, (실행 중일 때 임의의 순서로 획득), T2도 `L1`과 `L2` 락을, T3은 `L2`를 필요로 하고, T4는 락을 필요로 하지 않는다고 가정하자. 쓰레드들의 락 요청에 대한 정보를 다음과 같이 표로 정리 해 볼 수 있다.

	T1	T2	T3	T4
L1	예	예	아니오	아니오
L2	예	예	예	아니오

똑똑한 스케줄러라면 T1과 T2가 동시에 실행만 하지 않는다면 교착 상태가 절대로 발생하지 않도록 할 수 있다. 그와 같이 스케줄링된 예는 다음과 같다.



(T3과 T1) 또는 (T3과 T2) 끼리는 겹쳐서 실행이 되도 괜찮다. T3이 절대로 교착 상태를 유발하지 않는 이유는 단 하나의 락만 필요하기 때문이다.

또 다른 예를 살펴보자. 이번에는 다음의 표에서 나타난 것과 같이 동일한 자원(마찬가지로 L1과 L2락을 사용)에 대해 경쟁이 심해졌다고 해 보자.

	T1	T2	T3	T4
L1	예	예	예	아니오
L2	예	예	예	아니오

쓰레드 T1, T2, 그리고 T3이 실행 중 어느 시점에 모두 `L1`과 `L2` 락을 획득하는 경우를 예로 들어보자. 그런 경우에 교착 상태가 절대로 발생하지 않도록 하는 가능한 스케줄링은 다음과 같다.

보는 바와 같이 정적 스케줄링은 T1, T2, 그리고 T3이 모두 한 프로세서에서 실행되도록 보수적인 방법을 택하기 때문에 전체 작업이 끝나기까지 상당히 오랜 시간이 소요된다. 병행이 가능할 수도 있겠지만, 교착 상태가 발생할 수 있기 때문에 그렇게 할 수 없으며, 어쩔 수 없이 성능 하락을 수반한다.

유명한 예로 Dijkstra의 은행원 알고리즘(Banker's Algorithm) [D64]이 있으며 그와 비슷한 방법들도 다수 소개가 되었다. 상당히 제한적인 환경에서만 유용한 방법들이다. 예를 들어 전체 작업에 대한 모든 지식을 알고 있는 임베디드 시스템에서 작업을 실행하면서 필요한 락을 획득하는 경우이다. 더 나아가 이러한 방법들은 두 번째 예에서 본 것과 같이 병행성에 제약을 가져 올 수도 있다. 때문에 스케줄링으로 교착 상태를 회피하는 것은 보편적으로 사용되는 방법은 아니다.

발견 및 복구

마지막 전략은 교착 상태 발생을 허용하고, 교착 상태를 발견하면 복구토록 하는 방법이다. 예를 들어 운영체제가 일 년에 한 번 멈춘다고 했을 때 재부팅을 하고 기분 좋게 (또는 우울하게) 다시 작업을 처리하는 식이다. 교착 상태가 아주 가끔 발생한다면 이런 방법도 꽤 유용하다.

많은 데이터베이스 시스템들이 교착 상태를 발견하고 회복하는 기술을 사용한다. 교착 상태 발견은 주기적으로 실행되며 자원 할당 그래프를 그려서 사이클이 생겼는지를 검사한다. 사이클이 발생하는 경우(교착 상태인 경우) 시스템은 재부팅되어야 한다. 자료구조에 대한 복잡한 복구가 필요할 경우, 사람이 직접 복구 작업을 수행할 수도 있다.

데이터베이스의 병행성, 교착 상태, 그리고 관련 문제들에 대한 상세한 내용은 다른 문헌들에서 찾아 볼 수 있다 [Phi87; Kna87]. 그 자료들을 읽어보자. 더 좋은 방법은 데이터베이스 수업을 들어서 이 흥미로운 주제에 대해서 더 자세히 알아보자.

팁: 항상 완벽을 추구하지는 말자(Tom West's Law)

컴퓨터 산업에 고전이 된 *새로운 기계의 영혼(Soul of a New Machine)* [K81]이라는 책의 제목만큼 유명한 Tom West는 "해야하는 한 모든 일이 모두 다 잘해야 하는 일은 아니다."라는 공학적으로 아주 훌륭한 명언을 남겼다. 만약 안 좋은 일이 아주 가끔 일어난다면 그 일을 방지하기 위해서 아주 많은 시간을 들일 필요가 전혀 없다. 특히, 그 안 좋은 일에 대한 대가가 작다면 더 그렇다. 반면에, 우주 비행선을 만든다고 했을 때, 잘못되었을 때 그 비행기가 폭발할 수 있다고 하면, 그때는 이 조언을 무시해야 할 것이다.

35.4 요약

이 장에서는 병행 프로그램에서 발생할 수 있는 오류의 종류들에 대해 학습했다. 첫 번째 부류는 비 교착 상태 오류로서 상당히 흔하지만 대체적으로 고치기 쉬운 오류들이다. 이 부류는 함께 실행해야 하는 명령어들이 함께 실행이 안되어 발생하는 원자성 위반 오류와 두 쓰레드 간의 실행 순서가 지켜지지 않아 발생하는 순서 위반 오류를 포함한다.

교착 상태에 대해서도 구체적으로 다루었다. 교착 상태의 발생 원인과 대응방법에 대해 살펴보았다. 이 문제는 병행성 문제 자체만큼이나 오래된 것이라 관련 주제에 대해 수백 편의 논문들이 소개되었다. 가장 좋은 해법은 조심하는 것과 락 획득 순서를 정해서 애초에 교착 상태가 발생하지 않도록 예방하는 것이다. Linux를 포함해서 보편적으로 사용되는 라이브러리나 중요한 시스템에서 대기없는 자료 구조가 사용되는 것을 보아 대기없는 기법들도 유망해 보인다. 하지만, 보편성이 부족하고 새로운 대기 없는 자료 구조를 만드는 것이 복잡하기 때문에 제한적으로 사용될 것으로 보인다. 최선의 해법은 아마도 새로운 병행 프로그래밍 방법론을 만드는 것이라고 보인다. 구글의 맵리듀스(MapReduce)와 같은 시스템 [GD04]에서는 락을 전혀 사용하지 않고도 개발자가 특정한 병렬 연산을 처리할 수 있도록 해준다. 락은 원천적으로 문제점을 수반하기 때문에 반드시 필요한 경우가 아니면 사용을 피하도록 노력해야 한다.

참고 문헌

[CES71] **"System Deadlocks"**
E.G. Coffman, M.J. Elphick, and A. Shoshani
ACM Computing Surveys, 3:2, June 1971
교착 상태가 발생하는 조건의 정리와 해결 방안을 제시한 고전 논문이다. 그 전에 이 주제를 다뤘던 논문들에 대해서는 이 논문의 참고 문헌을 살펴보기 바란다.

[D64] **"Een algorithme ter voorkoming van de dodelijke omarming"**
Circulated privately, around 1964
URL: `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF`
Dijkstra가 교착 상태 문제에 대한 여러 해법들을 제시했을 뿐만 아니라, 이 문제가 있다는 것을, 최소한 문서화하여, 세상에 알린 첫 번째 사람이다. 하지만 그는 "죽음의 포옹(deadly embrace)"이라고 불렀지만 (다행스럽게도) 보편적으로 사용되지 않았다.

[GD04] **"MapReduce: Simplified Data Processing on Large Clusters"**
Sanjay Ghemawhat and Jeff Dean
OSDI '04, San Francisco, CA, October 2004
맵리듀스 논문은 대용량 데이터 처리의 시대를 안내하면서 일반적으로 신뢰성이 낮은 기기들로 클러스터를 사용하여 연산하는 프레임워크를 제안하였다.

[Her91] **"Wait-free Synchronization"**
Maurice Herlihy
ACM TOPLAS, 13(1), pages 124-149, January 1991
헐리히는 대기없는 기법을 사용하여 병행 프로그램을 구현하는 개념을 개척하였다. 이 기법들은 복잡하고 어렵고 때로는 락을 올바르게 사용하는 것보다 어려운 경향이 있기 때문에 실제 세계에서의 성공은 어려워 보인다.

[Jul+08] **"Deadlock Immunity: Enabling Systems To Defend Against Deadlocks"**
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea
OSDI '08, San Diego, CA, December 2008
최근에 발표된 교착 상태에 대한 훌륭한 논문으로 특정 시스템에서 같은 문제에 반복적으로 빠지는 경우를 피하는 방법에 대해 다루고 있다.

[K81] **"Soul of a New Machine"**
Tracy Kidder, 1981
시스템 개발자나 공학자라면 꼭 읽어야 할 책으로서 데이터 제너럴(DG) 회사 내에서 탐 웨스트가 이끈 팀이 "새로운 기계"를 만들었던 초창기 시절을 설명한다. 산들 너머의 산들(Mountains beyond the Mountains)을 포함해서 키더의 다른 책들도 역시 훌륭하다. 우리와 동의 안 할 수도 있겠지요?

[Kna87] **"Deadlock Detection in Distributed Databases"**
Edgar Knapp
ACM Computing Surveys, Volume 19, Number 4, December 1987
분산 데이터베이스 시스템에서 교착 상태 탐색에 대한 훌륭한 개론이다. 그와 더불어 다른 관련 여러 주제들에 대해서도 다루고 있기 때문에 입문을 위해 시작하기 좋은 책이다.

[Lu+08] **"Learning from Mistakes. A Comprehensive Study on Real World Concurrency Bug Characteristics"**
Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou
ASPLOS '08, March 2008, Seattle, Washington

실제 소프트웨어에서의 병행성 오류를 심도 있게 다룬 첫 연구이자, 이번 장의 바탕이 되었다. Y. Y. Zhou나 Shan Lu의 홈페이지에 들어가서 오류들을 다룬 다른 많은 흥미로운 연구들을 살펴보자.

[Phi87] **"Concurrency Control and Recovery in Database Systems"**
Nathan Goodman Philip A. Bernstein Vassos Hadzilacos
Addison-Wesley, 1987
데이터베이스 관리 시스템에서 병행성을 다룬 고전. 병행성, 교착 상태, 그리고 그 외의 데이터베이스의 다른 주제들은 그 자체로 또 다른 세상이다. 공부해보고 왜 그런지 스스로 찾아보자.

[Tmo94] **"Linux File Memory Map Code"**
Linus Torvalds and many others
URL: `http://lxr.free-electrons.com/source/mm/filemap.c`
이 소중한 예제를 지적해준 NYU의 Michael Walfish에게 감사를 표한다. 이 파일에서 볼 수 있듯이 실제 세계에서는 교과서의 단순한 예들보다 좀 더 복잡해질 수 있다.