Chapter 06

문제 6-1의 해답

○… (1) doWrite 메소드를 복수의 쓰레드가 동시에 실행하는 경우는 없다.

×… (2) doRead 메소드를 복수의 쓰레드가 동시에 실행하는 경우는 없다.

➜ doRead 메소드는 여러 ReaderThread 쓰레드에서 동시에 실행되는 경우가 있습니다.

○… (3) doWrite 메소드를 어떠한 쓰레드가 실행하고 있을 때 readingReaders 필드의 값은 반드시 0이다.

➜ 쓰기 처리를 하고 있는 중에는 읽기 처리를 하는 쓰레드가 없습니다.

○… (4) doRead 메소드를 어떠한 쓰레드가 실행하고 있을 때 writingWriters 필드의 값은 반드시 0이다.

➜ 읽기 처리를 하는 중에는 쓰기 처리를 하는 쓰레드가 없습니다.

문제 6-2의 해답

ReaderThread의 쓰레드나 WriterThread의 쓰레드에 대해서 전혀 배타제어가 일어나지 않습니다. 따라서 〈그림 6-1〉처럼 문자를 써넣고 있는 상태를 읽을 수 있게 되거나, 복수의 WriterThread가 입력하고 있는 상태를 읽을 수 있게 됩니다. 전자는 read-write conflict이며, 후자는 write-write conflict와 read-write conflict 둘 다 입니다.

Data 역할의 안전성은 잃게 됩니다.

그림 6-1 실행 예(Good! 과 Nice! 가 섞여서 표현된다)

```
(전략)
Thread-1 reads   aaaa ***** ← a를 적고 있는 상태가 읽혀진다(read-writer conflic)
Thread-2 reads   aaaa *****
Thread-0 reads   aaaa *****
Thread-3 reads   aaaa *****
Thread-4 reads   aaaa *****
Thread-5 reads   aaaa *****
Thread-0 reads   aaaaa *****
Thread-1 reads   aaaaa *****
(중략)
Thread-0 reads   hhhhFFgggg ← F를 적고 있는 중간에 h를 적고 있는 상태가 읽혀진다
Thread-3 reads   hhhhhFFggg     (write-write conflict와 read-write
conflict)
Thread-4 reads   hhhhhFFggg
Thread-5 reads   hhhhhFFggg
Thread-2 reads   hhhhhFFggg
(이하 생략. CTRL+C로 종료)
```

문제 6-3의 해답

〈리스트 6-2〉와 〈리스트 6-8〉에서 걸리는 시간을 비교하기 위해 ReaderThread 클래스를 〈리스트 6-1〉처럼 수정했습니다. ReaderThread의 쓰레드가 20회 실시하는 read 호출 앞뒤에서 현재 시각을 구하고, 걸린 시간을 산출하고 있습니다.

실행 예는 각각 〈그림 6-2〉와 〈그림 6-3〉과 같습니다. ReadWriter 클래스를 사용한 쪽은 약 3초만에 실행이 끝나는데 반해 synchronized를 사용한 쪽은 약 10~11초나 시간이 걸리는 것을 알 수 있습니다. 단, 이들 값은 doRead나 doWrite에 걸리는 시간, ReaderThread와 WriterThread의 개수 등에 영향을 받습니다. 또한 Java 실행 처리계의 구현에도 영향을 받으므로 주의해야 합니다.

리스트 6-1 시간을 계측하도록 수정한 ReaderThread 클래스 (ReaderThread.java)

```
public class ReaderThread extends Thread {
    private final Data data;
    public ReaderThread(Data data) {
        this.data = data;
    }
    public void run() {
        try {
```

```
            long begin = System.currentTimeMillis();
            for (int i = 0; i < 20; i++) {
                char[] readbuf = data.read();
                System.out.println(Thread.currentThread().getName() +
                        " reads " + String.valueOf(readbuf));
            }
            long time = System.currentTimeMillis() - begin;
            System.out.println(Thread.currentThread().getName() + ":
                    time = " + time);
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ReadWriteLock/A6-3a, A6-3b

그림 6-2 실행 예 (ReadWriteLock 클래스를 사용한 경우)

```
Thread-4 reads   bbbbbbbbbb
Thread-3 reads   bbbbbbbbbb
Thread-1 reads   bbbbbbbbbb
Thread-2 reads   bbbbbbbbbb
Thread-0 reads   bbbbbbbbbb
Thread-0: time = 3054
Thread-5 reads   bbbbbbbbbb
Thread-4 reads   bbbbbbbbbb
Thread-4: time = 3054
Thread-2 reads   bbbbbbbbbb
Thread-1 reads   bbbbbbbbbb
Thread-3 reads   bbbbbbbbbb
Thread-1: time = 3054
Thread-3: time = 3054
Thread-2: time = 3054
Thread-5: time = 3054 ← 도중에 정지
(CTRL+C로 종료)
```

그림 6-3 실행 예 (ReadWriteLock 클래스를 사용하지 않고 synchronized를 사용한 경우)

```
Thread-2 reads   EEEEEEEEEE
Thread-3 reads   EEEEEEEEEE
Thread-4 reads   EEEEEEEEEE
Thread-5 reads   EEEEEEEEEE
Thread-0 reads   EEEEEEEEEE
Thread-1 reads   EEEEEEEEEE
Thread-2 reads   EEEEEEEEEE
Thread-3 reads   eeeeeeeeee
Thread-4 reads   eeeeeeeeee
Thread-5 reads   eeeeeeeeee
Thread-0 reads   eeeeeeeeee
Thread-0: time = 10835
Thread-1 reads   eeeeeeeeee
Thread-1: time = 10885
Thread-2 reads   eeeeeeeeee
Thread-2: time = 10935
Thread-3 reads   FFFFFFFFFF
Thread-3: time = 11486
Thread-4 reads   FFFFFFFFFF
Thread-4: time = 11536
Thread-5 reads   FFFFFFFFFF
Thread-5: time = 11586   ← 도중에 정지
(CTRL+C로 종료)
```

문제 6-4의 해답

Read-Writer Loc 패턴을 적용한 Database 클래스는 〈리스트 6-2〉와 같습니다.

java.util.concurrent.locks 패키지의 lock, ReadWriteLock, 그리고 ReentrantRead WriteLock을 사용하여 만들었습니다.

Database 클래스의 3개 public한 메소드 가운데 clear와 assign은 「쓰기」 메소드이며, retrieve는 「읽기」 메소드입니다. 그래서 clear와 assign은 writeLock으로 보호하며 retrieve는 readLock으로 보호합니다. 사용된 synchronized는 모두 삭제합니다.

리스트 6-2 Database 클래스 (Database.java)

```
import java.util.Map;
import java.util.HashMap;

import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Database<K,V> {
    private final Map<K,V> map = new HashMap<K,V>();

    private final ReadWriteLock lock = new ReentrantReadWriteLock(true
                                        /* fair */);
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    // 모두 삭제한다
    public void clear() {
        writeLock.lock();
        try {
            verySlowly();
            map.clear();
        } finally {
            writeLock.unlock();
        }
    }

    // key에 value를 할당한다
    public void assign(K key, V value) {
        writeLock.lock();
        try {
            verySlowly();
            map.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }

    // key에 할당된 값을 취득한다
    public V retrieve(K key) {
        readLock.lock();
        try {
            slowly();
            return map.get(key);
        } finally {
            readLock.unlock();
        }
    }

    // 처리에 시간이 걸리는 것을 시뮬레이트한다
    private void slowly() {
        try {
```

```
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }

    // 처리에 많은 시간이 걸리는 것을 시뮬레이트한다
    private void verySlowly() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ReadWriteLock/A6-4a*

힘들여 만들었으니 Read-Write Lock 패턴을 사용한 Database(리스트 6-2)와 사용하지 않는 Database(리스트 6-9, 본문 263p)의 성능을 비교해 봅시다. 그러기 위해 RetrieveThread, AssignThread, Main 등 3개 클래스를 준비합니다.

RetrieveThread 클래스(리트스 A6-3)는 Database의 retrieve 메소드를 계속 호출하는 클래스입니다.

여기에서는 「retrieve 메소드를 호출한 전체 회수」를 세는 atomicCounter라고 하는 클래스 필드를 준비했습니다. 이 필드는 java.util.concurrent.atomic.AtomicInterger 클래스의 인스턴스이며, incrementAndGet 메소드를 사용하여 최소 단위에 +1 증가시킬 수 있습니다. atomicCounter는 클래스 필드이므로 모든 쓰레드의 retrieve 호출 횟수를 알 수 있습니다.

리스트 6-3 Database의 retrieve 메소드를 계속 호출하는 RetrieveThread, 클래스 (RetrieveThread,.java)

```
import java.util.concurrent.atomic.AtomicInteger;

public class RetrieveThread extends Thread {
    private final Database<String,String> database;
    private final String key;
    private static final AtomicInteger atomicCounter=new AtomicInteger(0);

    public RetrieveThread(Database<String,String> database, String key) {
        this.database = database;
        this.key = key;
    }
```

```
    public void run() {
        while (true) {
            int counter = atomicCounter.incrementAndGet();
            String value = database.retrieve(key);
            System.out.println(counter + ":" + key + " => " + value);
        }
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ReadWriteLock/A6-4a

AssignThread 클래스(리스트 6-4)는 Database의 assign 메소드를 계속해서 호출하는 클래스입니다.

「쓰기」 처리의 빈도가 낮음을 나타내기 위해 불특정 시간 동안 sleep하고 있습니다.

리스트 6-4 Database의 assign 메소드를 계속 호출하는 AssignThread 클래스 (AssignThread.java)

```
import java.util.*;

public class AssignThread extends Thread {
    private static Random random = new Random(314159);
    private final Database<String,String> database;
    private final String key;
    private final String value;

    public AssignThread(Database<String,String> database, String key,
String value) {
        this.database = database;
        this.key = key;
        this.value = value;
    }

    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName() +
                    ":assign(" + key + ", " + value + ")");
            database.assign(key, value);
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ReadWriteLock/A6-4a

Main 클래스(리스트 6-5)는 AssignThread와 RetrieveThread를 약 10초 동안만 동작시키는 클래스입니다.

AssignThread 쓰레드를 4개, RetrieveThread 쓰레드를 200개 기동하고 약 10초간 정지한 다음 강제 종료하고 있습니다.

리스트 6-5 AssignThread와 RetrieveThread를 약 10초간 동작시키는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Database<String,String> database  = new
                Database<String,String>();

        // AssignThread 쓰레드 기동
        new AssignThread(database, "Alice", "Alaska").start();
        new AssignThread(database, "Alice", "Australia").start();
        new AssignThread(database, "Bobby", "Brazil").start();
        new AssignThread(database, "Bobby", "Bulgaria").start();

        // RetrieveThread 쓰레드 기동
        for (int i = 0; i < 100; i++) {
            new RetrieveThread(database, "Alice").start();
            new RetrieveThread(database, "Bobby").start();
        }

        // 약 10초간 정지
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
        }

        // 강제 종료
        System.exit(0);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ReadWriteLock/A6-4a*

실행 예는 〈그림 6-4〉와 〈그림 6-5〉와 같습니다. Read-Write Lock 패턴을 사용한 쪽(그림 6-4)에서는 RetrieveThread가 retrieve를 1197번 호출하고 있습니다. 그런데 Read-Write Lock 패턴을 사용하지 않은 쪽(그림 6-5)에서는 160번 정도 밖에 호출하지 않음을 알 수 있습니다.

그림 6-4 Read-Write Lock 패턴을 사용한 Database (리스트 6-2)의 실행 예

```
Thread-0:assign (Alice, Alaska)
Thread-1:assign (Alice, Australia)
Thread-2:assign (Bobby, Brazil)
Thread-3:assign (Bobby, Bulgaria)
Thread-1:assign (Alice, Australia)
Thread-0:assign (Alice, Alaska)
Thread-3:assign (Bobby, Bulgaria)
1:Alice => Australia
2:Bobby => Bulgaria
3:Alice => Australia
4:Bobby => Bulgaria
(중략)
1200:Bobby => Brazil
1192:Bobby => Brazil
1193:Bobby => Brazil
1194:Alice => Australia
1195:Alice => Australia
1196:Alice => Australia
1197:Alice => Australia
```

그림 6-5 Read-Write Lock 패턴을 사용하지 않은 Database(리스트 6-9)의 실행 예

```
Thread-0:assign (Alice, Alaska)
Thread-1:assign (Alice, Australia)
Thread-2:assign (Bobby, Brazil)
Thread-3:assign (Bobby, Bulgaria)
Thread-1:assign (Alice, Australia)
Thread-0:assign (Alice, Alaska)
Thread-3:assign (Bobby, Bulgaria)
1:Alice => Australia
2:Bobby => Bulgaria
3:Alice => Australia
4:Bobby => Bulgaria
(중략)
153:Alice => Australia
154:Bobby => Bulgaria
155:Alice => Australia
156:Bobby => Bulgaria
157:Alice => Australia
158:Bobby => Bulgaria
159:Alice => Australia
160:Bobby => Bulgaria
```

문제 6-5의 해답

예제 프로그램의 범위에서는 올바로 동작합니다. 하지만 쓰레드가 interrupt된 경우 readUnlock 혹은 writeUnlock을 필요 이상으로 호출할 위험이 있습니다.

지금 쓰레드가 lock.readLock() 안에서 wait하고 있다고 합시다. 이 쓰레드가 interrupt되면 예외 InterruptedException이 통보되면서 readLock으로부터 탈출합니다. 이 때 readingReaders 필드는 증가하지 않습니다.

readLock으로부터 탈출한 쓰레드는 finally절로 들어가 lock.readUnlock()을 실행합니다. 이 안에서 방금 전 인클리먼트하지 않았던 readingReaders가 디클리먼트되어 readingReaders 필드의 값이 필요 이상으로 작아져 버립니다.

InterruptedException은 프로그램의 응답성을 높이는데 필요하지만 자칫하면 프로그램의 안전성을 잃을 수 있으므로 수정할 때에는 주의가 필요합니다. 문제문에서 제시한 Before/After 패턴의 구조를 다시 한 번 살펴봅시다.

```
before();
try {
    execute();
} finally {
    after();
}
```

여기에서 before가 try의 **바깥**에 있는 것은 「만일 before 실행 중 예외가 발생한 경우, execute은 물론이거니와 after도 실행하지 않는다」는 것을 의미합니다. 예를 들어 before 안에서 예외 InterruptedException이 통보되었을 때에는 「before의 실행을 중단했다」고 이해할 수 있습니다. 만일 before를 try절 안에 놓게 되면 beofre의 실행을 중단한 경우에도 after가 호출되고 맙니다.

문제 6-6의 해답

이는 ReaderThread 쓰레드가 WriterThread 쓰레드보다 수가 많아서 일어나는 현상입니다. ReaderThread가 한 개라도 doRead를 실행하는 중이라면 WriterThread는 dowrite를 실행할 수 없습니다. 그런데 ReaderThread는 배타제어되지 않기 때문에 차례 차례 doRead를 실행합니다. 결국 WriterThread는 doWrite을 좀처럼 실행할 수가 없게

됩니다. 이는 자동차가 연이어서 달려오는 도로를 횡단하는 것이 어려운 것과 비슷합니다.

예제 프로그램 ReadWriteLock 클래스(리스트 6-5)의 waitingWriters 필드는 wait하고 있는 WriterThread의 수를 보관합니다. waitingWriters 〉 0이 성립될 때에는 ReaderThread의 쓰레드를 wait 시킴으로써 WriterThread가 실행을 개시하지 못하는 현상을 막을 수 있습니다. 이는 자동차 신호가 자동적으로 빨간색으로 바뀌면 달리던 차가 멈추기 때문에 길을 무사히 건널 수 있는 것과 마찬가지입니다.

그런데 waitingWriters에만 신경쓰다 이번에는 ReaderThread가 doRead의 실행을 개시하지 못하게 될 수 있습니다. 한 개의 WriterThread가 doWrite의 실행을 마치기도 전에 다른 WriterTharead가 write에 도착하는 경우가 그렇습니다. WriterThread가 한 개라도 wait하고 있으면 ReaderThread는 doRead 실행을 시작할 수 없습니다. 이는 많은 사람들이 횡단보도를 끊임없이 건널 때에는 자동차가 꼼짝없이 서 있어야 하는 경우와 같습니다.

예제 프로그램의 ReadWriteLock 클래스(리스트 6-5)의 **preferWriter 필드는 ReaderThread와 WriterThread 중 어느 쪽을 우선할지 결정하는 플래그입니다.** preferWriter가 true일 때에 한하여 가드 조건인 waitingWriters를 고려하도록 만들어져 있습니다. 그리고 readUnlock 안에서(즉, doRead가 끝났을 때)는 preferWriter를 false로 합니다. 결국 read 처리가 끝나면 write 처리를 우선하고, write 처리가 끝나면 read 처리를 우선하게 됩니다. preferWriter 필드는 ReaderThread와 WriterThread를 교대로 우선하기 위해 사용하고 있는 것입니다. 이는 자동차의 빨간 신호와 보행자의 빨간 신호가 교대로 켜지는 것과 같습니다.

문제 6-7의 해답

(1) 아니오, 그렇지 않습니다.

ReaderThread와 WriterThread 양쪽 쓰레드가 this 상에서 wait하는 경우가 있습니다.

(2) 아니오, 그렇지 않습니다.

ReaderThread와 WriterThread 양쪽 쓰레드가 this 상에서 wait하는 경우가 있습니다. 예를 들어 preferWriter && waitingWriters 〉 0이 true일 때 뒤에 오는 ReaderThread 쓰레드는 기다리게 됩니다.