Chapter 03

문제 3-1의 해답

○… (1) getRequest와 putRequest는 서로 다른 쓰레드에서 호출한다.

➜ getRequest는 ServerThread로부터, putRequest는 ClientThread로부터 각각 호출됩니다.

×… (2) RequesQueue의 인스턴스는 2개 만들어진다.

➜ 만들어지는 인스턴스는 Main 클래스가 만드는 것 한 개뿐입니다. 이 한 개의 인스턴스를 ClientThread와 ServerThread에서 공유합니다.

○… (3) getReques 안에서 remove 메소드를 호출할 때 queue.peek() != null의 값은 반드시 참(true)이 된다.

➜ queue.peek() != null은 가드 조건입니다. 목적하는 처리가 실행될 때 가드 조건이 충족되어 있음을 보증하는 것이 Guarded Suspension 패턴입니다.

○… (4) getReques 안에서 wait 메소드를 호출할 때 queue.peek() != null의 값은 반드시 거짓(false)이 된다.

➜ wait 메소드를 호출하러 가는 것은 가드 조건을 만족하지 않았을 때입니다.

×… (5) ClientThread 쓰레드가 putRequest를 실행하는 동안에는 ServerThread의 쓰레드가 동작하지 않는다.

➜ ServerThread의 쓰레드에는 동작하는 것과 동작하지 않는 것이 있습니다. getRequest를 실행하려 했다면 락을 취하지 못해 블록합니다. wait을 실행했다면 wait 셋 안에서 멈추어 있습니다. sleep를 실행했다면 지정한 시간만큼 정지합니다. 그러나 그 외의 경우 SeverThread의 쓰레드는 자유롭게 동작합니다.

×… (6) getReques 안에서 wait 메소드를 불러낸 쓰레드는 락을 해제하고 queue의 wait 셋으로 들어간다.

➜ queue(LinkedList 클래스의 인스턴스)의 wait 셋이 아니라 this(RequestQueue 클래스의 인스턴스)의 wait 셋으로 들어갑니다.

×… (7) putRequest 메소드 안의 notifyAll();이라고 하는 문은 queue.notifyAll();이라고 적어도 의미가 같다.

➜ queue.notifyAll();이라고 적으면 의미가 달라집니다. this.notifyAll();이라고 적으면 notifyALL();와 의미가 같습니다.

문제 3-2의 해답

notifyAll을 먼저 실행해도 RequestQueue 클래스는 안전하게 동작합니다.

notifyAll을 실행한 시점에서 인수 request는 아직 queue에 추가되지 않습니다. 그러나 notifyAll을 실행한 쓰레드는 this의 락을 가지고 있기 때문에 notifyAll로 인해 wait 셋 밖으로 나온 다른 쓰레드들은 락을 취하려고 전원 블록합니다. 따라서 다른 쓰레드의 처리는 실질적으로 진행되지 않습니다(가드 조건도 테스트하지 않습니다).

한편 notifyAll을 실행한 쓰레드는 offer를 실행한 후에 putRequest에서 돌아옵니다. 이 때서야 this의 락이 해제됩니다. 그 후 블록되어 있던 다른 쓰레드(중의 한 개)가 this의 락을 취하고 처리를 진행하게 됩니다(우선 가드 조건을 테스트하러 간다).

결국, putRequest 안에 있는 2개 문의 순서가 어떠하든지 안전하게 동작합니다.

단, notifyAll을 가장 마지막에 적는 편이 이해하기 쉬우므로 그렇게 프로그램을 짜는 것이 좋겠지요.

문제 3-3의 해답

우선 「기대한 동작」이란 어떤 동작일까, 구체적으로 생각해 봅시다.

:: getRequest를 호출했을 때 queue.peek() == null 이라면 쓰레드 Bobby는 wait한다

:: putRequest 했을 때 쓰레드 Alice는 notifyAll한다.

:: notifyAll 되면 wait하고 있던 쓰레드 Bobby는 wait을 마친다

그래서 wait()의 전후 그리고 notifyAll() 실행부분에 디버그 프린트를 넣어보았습니다(리스트 3-1).

리스트 3-3 디버그 프린트를 넣은 RequestQueue 클래스 (RequestQueue.java)

```
import java.util.Queue;
import java.util.LinkedList;

public class RequestQueue {
    private final Queue<Request> queue = new LinkedList<Request>();
    public synchronized Request getRequest() {
        while (queue.peek() == null) {
            try {
                System.out.println(Thread.currentThread().getName() +
                        ": wait() begins, queue = " + queue);
                wait();
                System.out.println(Thread.currentThread().getName() +
                        ": wait() ends,   queue = " + queue);
            } catch (InterruptedException e) {
            }
        }
        return queue.remove();
    }
    public synchronized void putRequest(Request request) {
        queue.offer(request);
        System.out.println(Thread.currentThread().getName() + ":
                notifyAll() begins, queue = " + queue);
        notifyAll();
        System.out.println(Thread.currentThread().getName() + ":
                notifyAll() ends, queue = " + queue);
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/GuardedSuspension/A3-3*

그림 3-1 실행 예

```
Alice requests [ Request No.0 ]       ←Alice가 No.0을 리퀘스트 한다
Alice : notifyALL() begins, queue = [[ Request No.0 ]] ←Alice가
notifyALL 한다(아무도 기다리고 있지 않지만)
Alice : notifyALL() ends, queue = [[ Request No.0 ]]
Bobby handles [ Request No.0 ]        ←Bobby가 No.0을 처리한다(기다리지 않는다)
Bobby : wait() begins, queue = []     ←Bobby가 wait 한다(대기 상태에 들어갔다)
Alice requests [ Request No.1 ]       ←Alice가 No.1을 리퀘스트 한다
Alice : notifyALL() begins, queue = [[ Request No.1 ]] ←Alice가
notifyALL(Bobby가 기다리고 있다) 한다
Alice : notifyALL() ends, queue = [[ Request No.1 ]]
Bobby : wait() ends, queue = [[ Request No.1 ]] ←Bobby가 wait을 마친다
                                                  (queue에는 No.1이 들어가 있
                                                  다)
Bobby handles [ Request No.1 ]        ←Bobby가 No.1을 처리한다(대기한 후)
Alice requests [ Request No.2 ]       ←Alice가 No.2를 리퀘스트 한다
Alice : notifyALL() begins, queue = [[ Request No.2 ]] ←Alice가
notifyALL 한다(아무도 기다리고 있지 않지만)
Alice : notifyALL() ends, queue = [[ Request No.2 ]]
Bobby handles [ Request No.2 ]        ←Bobby가 No.2를 처리한다(기다리지 않는다)
Alice requests [ Request No.3 ]       ←Alice가 No.3을 리퀘스트 한다
Alice : notifyALL() begins, queue = [[ Request No.3 ]] ←Alice가
notifyALL 한다(아무도 기다리고 있지 않지만)
Alice : notifyALL() ends, queue = [[ Request No.3 ]]
Alice requests [ Request No.4 ]       ←Alice가 No.4를 리퀘스트 한다
Alice : notifyALL() begins, queue = [[ Request No.3 ], [Request No.4] ]
←Alice가 notifyALL 한다(아무도 기다리고 있지 않지만)
Alice : notifyALL() ends, queue = [[ Request No.3 ], [Request No.4]]
Bobby handles [ Request No.3 ]        ←Bobby가 No.3을 처리한다(기다리지 않는다)
Bobby handles [ Request No.4 ]        ←Bobby가 No.4를 처리한다(기다리지 않는다)
```

문제 3-4의 해답

(1)~(4)에는 다음과 같은 문제점이 있습니다.

(1) while을 if로 한 경우

예제 프로그램의 범위에서는 문제가 없지만 일반적으로는 문제가 됩니다.

복수의 쓰레드가 wait하고 있을 때 RequestQueue의 인스턴스가 notifyAll되었다고 합시다. 그러면 여러 개의 쓰레드가 모두 움직이기 시작합니다. 만일 이 때 queue 안에 존재하는 요소가 한 개뿐이라면 움직이기 시작한 첫 번째 쓰레드가 queue.remove()를 호출하

고 그 결과 queue는 텅 비게 됩니다. queue가 비어있으면 queue.peek()의 값은 null이 됩니다. 그러나 움직이기 시작한 2개째 이후의 쓰레드는 이미 가드 조건의 체크가 끝난 상태이기 때문에 queue.peek() == null 임에도 불구하고 queue.remove()를 호출해 버립니다. 따라서 이 클래스는 안전성에 문제가 있습니다.

wait하고 있던 쓰레드는 움직이기 전에 가드 조건을 다시 체크해야 합니다. 그러기 위해서는 if가 아니라 while을 사용해야 합니다.

notifyAll 대신에 notify를 사용한다면 while이 아니라 if라도 괜찮을까요? 이 프로그램의 범위에서라면 괜찮습니다. 그러나 훨씬 큰 프로그램 안에서 사용하면 문제가 발생합니다. RequestQueue의 인스턴스가 누군가(쓰레드)에 의해서 notify/notifyAll되어 버릴 수 있기 때문입니다. notifyAll 대신 notify를 사용하고 while 대신 if를 사용하여 Guarded Suspension 패턴을 구현한 클래스는 재사용성이 낮습니다.

거듭 말하지만 **Guarded Suspension 패턴은 while을 사용하여 「조건으로 가드」하는 것이 중요합니다. notify/notifyAll은 조건을 체크하게 만드는 계기에 지나지 않습니다.**

예를 들어 다음과 같은 장면을 상상해 보십시오. 자동차를 운전하던 당신은 빨간 신호등 앞에서 차를 멈추고 멍하니 있습니다. 조수석에서 「이봐!」하며 출발을 재촉한다고 해서 당황한 나머지 엑셀레이터를 밟아서는 안 됩니다. 신호가 파란 불인지, 전진해도 안전한지 스스로 확인한 다음에 엑셀레이터를 밟아야 하는 것입니다. Guarded Suspension도 이와 마찬가지입니다. notify/notifyAll은 「이봐!」하는 소리에 불과합니다. 처리를 진행시키기 전에는 가드 조건을 다시 한번 확실히 체크해야 합니다.

(2) synchronized의 범위를 wait만으로 한 경우(리스트 3-9)

예제 프로그램의 범위에서는 문제가 없습니다만, 일반적으로는 문제가 됩니다. 이 경우 다음과 같은 처리가 synchronized 블록 밖으로 나와 버립니다.

:: 조건의 테스트

:: remove의 호출

queue 안에 요소가 한 개 밖에 존재하지 않을 때 〈그림 3-2〉와 같이 2개의 쓰레드가 실행해 버리면 쓰레드 1에서 예외 NoSuchElementException이 통보됩니다.

게다가 원래 queue 필드의 LinkedList 클래스는 쓰레드 세이프가 아닙니다.

이 클래스를 사용하면 안전성이 떨어지게 됩니다.

그림 3-2 2개 쓰레드의 처리가 교차

쓰레드 1	쓰레드 2
조건 테스트	
	조건 테스트
	remove 호출
remove 호출	

(3) try…catch를 while 밖으로 꺼낸 경우 (리스트 -10)

예제 프로그램의 범위에서는 문제가 없습니다만, 일반적으로는 문제가 됩니다.

쓰레드가 wait하고 있는 중에 다른 쓰레드에서 interrupt 메소드를 호출했다고 합시다. 그러면 아직 가드 조건이 충족되지 않은 상태라 하더라도 이 쓰레드는 while문을 탈출하여 catch절로 뛰어 들어가 remove를 호출해 버립니다. 즉, 가드 조건이 충족될 때까지 기다린다고 하는 기능을 수행하지 못합니다.

이 클래스를 사용하면 안전성에 문제가 생깁니다. 연습문제 3-6에서 예외 InterruptedException을 올바로 다루는 프로그램을 만들어보겠습니다.

(4) wait 대신 Thread.sleep를 사용한 경우 (리스트 3-11)

예제 프로그램의 범위에서도 문제가 있습니다.

약 100밀리 초 단위로 가드 조건을 테스트하기 때문에 수행 능력이 떨어집니다…라고 하는 답은 틀렸습니다. 수행 능력의 문제가 아니라 생존성의 문제입니다. wait과 Thread.sleep는 다릅니다. **wait을 실행했던 쓰레드는 대상 인스턴스의 락을 해제합니다.** 한편 **Thread.sleep는 인스턴스의 락을 해제하지 않습니다.** 그러니까 getRequest라고 하는 synchronized 메소드 안에서 Thread.sleep를 실행해 버리면 그 어떤 쓰레드도 putRequest 쓰레드나 getRequest 쓰레드로 들어갈 수 없습니다(블록해 버립니다). putRequest에 들어갈 수 없기 때문에 queue.peek()의 값은 언제까지나 null 상태이며 가드 조건 역시 충족되지 못한 상태로 남습니다. sleep하고 있던 쓰레드는 약 100밀리 초 간격으로 깨어나 가드 조건을 테스트합니다. 하지만 가드 조건은 여전히 거짓 상태이기 때문에 쓰레드는 다시 잠이 듭니다. 「기상 → 테스트 → 다시 취침」동작을 영원히 반복하게 되는 것이죠. 그 동안 putRequest나 getRequest를 실행하려 한 다른 쓰레드는 모두 블록해 버립니다.

결국 이 클래스를 사용하면 생존성을 잃게 됩니다. 이 예에서 약 100밀리 초마다 가드 조건을 테스트하고 있는 쓰레드는 처리가 멈춘 것이 아닙니다. 하지만 정기적으로 가드 조건을 계속 테스트하고 있을 뿐 영원히 다음 단계로 나아가지 않습니다. 이렇게 **움직이고는 있으나 실질적으로 전혀 진전이 없는 상황**을 일반적으로 **라이브 락**(livelock)이라고 합니다. 라이브 락은 데드락과 마찬가지로 생존성을 잃은 상태입니다.

이 예는 연습문제 I1-2에 등장했던 「synchronized 메소드 안에서 무한 루프」하고 있는 상황의 하나입니다.

문제 3-5의 해답

작동을 멈춘 이유는 Alice와 Bobby 2개 쓰레드가 **데드락**을 일으키고 있기 때문입니다.

TalkThread 클래스의 run 메소드를 잘 살펴보면 제일 처음 getRequest 메소드에서 시작되고 있습니다.

```
:: Alice는 getRequest 안에서 Bobby로부터의 리퀘스트를 기다리며 wait합니다.
:: Bobby는 getRequest 안에서 Alice로부터의 리퀘스트를 기다리며 wait합니다.
```

이로써 Alice와 Bobby 양쪽이 모두 서로 마주선채로 움직일 수 없게 되고 말았습니다(데드락). 이는 상대를 흉내 내는 앵무새 2마리가 입을 꾹 다문 채 서로 쳐다보고 있는 것과 같은 상태입니다.

해결 방법의 하나를 〈리스트 3-2〉에 소개합니다. 〈리스트 3-2〉에서는 처음에 requestQueue1 쪽에 "Hello"라는 이름의 리퀘스트를 putRequest합니다. 이것은 이른바 「재료」입니다. 이 리퀘스트는 Alice가 제일 처음 getRequest로 구하게 됩니다.

그러면 〈그림 3-3〉처럼 Hello 뒤에 느낌표가 잔뜩 붙은 대화가 2개 쓰레드 사이에서 이뤄지게 됩니다(다른 해결 방법도 있습니다).

참고로 Alice와 Bobby가 데드락에 빠졌을 때에는 2사람 모두 requestQueue1이나 requestQueue2의 락을 가지고 있지 않습니다. wait 하면서 락은 해제했기 때문입니다. 연습문제 1-6에서 소개했던 데드락과 비교해 보세요.

리스트 3-2 제일 처음에 「재료」을 넣어서 해결 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        RequestQueue requestQueue1 = new RequestQueue();
        RequestQueue requestQueue2 = new RequestQueue();
        requestQueue1.putRequest(new Request("Hello"));
        new TalkThread(requestQueue1, requestQueue2, "Alice").start();
        new TalkThread(requestQueue2, requestQueue1, "Bobby").start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/GuardedSuspension/A3-5*

그림 3-3 실행 결과

```
Alice:BEGIN
Alice gets [ Request Hello ]      ← 제일 처음 Alice가 「재료」를 취한다
Alice puts [ Request Hello! ]     ← 느낌표를 한 개 붙여서 반환한다
Bobby:BEGIN
Bobby gets [ Request Hello! ]     ← Bobby가 그것을 취한다
Bobby puts [ Request Hello!! ]    ← 느낌표 한 개를 다시 추가하여 반환한다 (총 2개)
Alice gets [ Request Hello!! ]    ← Alice가 그것을 취한다
Alice puts [ Request Hello!!! ]   ← 느낌표 한 개를 다시 추가하여 반환한다 (총3개)
(중략)
Alice gets [ Request Hello!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! ]
Alice puts [ Request Hello!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! ]
Alice:END
Bobby gets [ Request Hello! ]
Bobby puts [ Request Hello!! ]
Bobby:END
```

문제 3-6의 해답

interrupt 메소드를 호출해도 쓰레드가 정지하지 않는 것은 sleep 메소드와 wait 메소드의 호출 부분에서 예외 InterruptedException을 무시하고 있기 때문입니다.

RequestQueue 클래스를 수정하여 getRequest 메소드가 InterruptedException을 통보하도록 합니다.

또한 ClientThread 클래스와 ServerThread 클래스를 InterruptedException이 통보되었을 때 for 문 밖으로 탈출하도록 수정합니다. 이렇게 하면 올바로 종료됩니다.

리스트 3-3 예외 InterruptedException를 무시하지 않는 RequestQueue 클래스(RequestQueue.java)

```
import java.util.Queue;
import java.util.LinkedList;

public class RequestQueue {
    private final Queue<Request> queue = new LinkedList<Request>();
    public synchronized Request getRequest() throws InterruptedException {
        while (queue.peek() == null) {
            wait();
        }
        return queue.remove();
    }
    public synchronized void putRequest(Request request) {
        queue.offer(request);
        notifyAll();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/GuardedSuspension/A3-6*

리스트 3-4 예외 InterruptedException를 무시하지 않는 ClientThread 클래스(ClientThread.java)

```
import java.util.Random;

public class ClientThread extends Thread {
    private final Random random;
    private final RequestQueue requestQueue;
    public ClientThread(RequestQueue requestQueue, String name, long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                Request request = new Request("No." + i);
                System.out.println(Thread.currentThread().getName() +
                        " requests " + request);
                requestQueue.putRequest(request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/GuardedSuspension/A3-6*

리스트 3-5 InterruptedException를 무시하지 않는 ServerThread 클래스(ServerThread.java)

```
import java.util.Random;

public class ServerThread extends Thread {
    private final Random random;
    private final RequestQueue requestQueue;
    public ServerThread(RequestQueue requestQueue, String name, long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                Request request = requestQueue.getRequest();
                System.out.println(Thread.currentThread().getName() +
                        " handles " + request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/GuardedSuspension/A3-6*

그림 3-4 실행 예

```
(전략)
Alice requests [ Request No.14 ]
Bobby handles [ Request No.14 ]
Alice requests [ Request No.15 ]
Bobby handles [ Request No.15 ]
Alice requests [ Request No.16 ]
Bobby handles [ Request No.16 ]
Alice requests [ Request No.17 ]
Bobby handles [ Request No.17 ]
***** calling interrupt ***** ← 여기에서 프로그램을 종료한다
```

잠깐! 한 마디 : RequestQueue 클래스를 수정하는 것을 잊으면 어떻게 되나?

RequestQueue 클래스를 수정하는 것을 잊으면 어떻게 될 지 생각해 볼까요. RequestQueue 클래스를 수정하지 않았다고 합시다. interrupt 메소드가 호출되었을 때 쓰레드가 **sleep**하고 있었다면 프로그램은 올바로 종료됩니다. 하지만 interrupt 메소드가 호출되었을 때 쓰레드가 **wait** 하고 있었다면 프로그램은 종료되지 않습니다. 통보된 예외 InterruptedException이 무시되기 때문입니다. 즉, 몇 번에 한 번(혹은 몇 십 번, 몇 백 번에 한 번)은 기대한 대로 종료하지 않는 프로그램이 만들어 집니다. 쓰레드를 확실하면서도 우아하게 종료시키는 방법에 대해서는 Chapter 10 Two-Phase Termination 패턴에서 다루겠습니다.