Chapter 05

문제 5-1 해답

○… (1) MakerThread 클래스의 생성자에 적혀 있는 식 super(name)는 Thread 클래스의 생성자를 호출하고 있다.

×… (2) MakerThread 클래스의 nextId 메소드가 synchronized로 되어 있는 것은 EaterThread 클래스로부터도 호출 받기 때문이다.

➜ MakerThread 클래스의 nextId 메소드가 synchronized로 되어 있는 것은 MakerThread로부터 호출되기 때문입니다. EaterThread에서는 nextId를 호출하지 않습니다.

×… (3) 테이블에 케이크가 한 개도 없는 상황에서 tale 메소드를 호출하면 쓰레드는 Table 인스턴스의 락을 취하려고 블록한다.

➜ 쓰레드는 「락을 취하려고 블록한다」가 아니라 가드 조건이 충족되지 않았기 때문에 wait하고 있는 것입니다.

○… (4) 테이블에 케이크가 한 개도 놓여 있지 않을 때 count 필드의 값은 0이다.

×… (5) 테이블이 케이크로 가득 차 더 이상 놓을 수 없을 때 count 필드의 값은 buffer.length -1과 같다.

➜ 테이블이 케이크로 가득 찼을 때 count의 값은 buffer.length -1이 아닙니다.

○… (6) head 필드의 값은 buffer.length 이상이 될 수 없다.

문제 5-2 해답

MakerThread의 Alice와 EaterThread의 Bobby에게 Table 클래스의 서로 다른 인스턴스가 전달되었습니다. 즉, 이 쓰레드에서는 테이블을 공유하고 있지 않는 것입니다.

Alice는 케이크를 3개(No.0, No.1, No.2) 만들어 테이블에 놓고 4개째(No.3)를 놓으려고 wait하고 있습니다. 그러나 Alice가 사용하고 있는 테이블에서 케이크를 가져가 줄 쓰레드는 어디에도 없습니다. 때문에 Alice는 영원히 wait 합니다. 한편 Bobby는 Alice가 케이

크를 올려놓은 테이블과는 다른 테이블에서 케이크를 먹으려 합니다. 그 때 테이블 위에는 아무것도 없기 때문에 Bobby는 wait 합니다. 그런데 Bobby가 사용하는 테이블에 케이크를 놓아 줄 쓰레드가 어디에도 없기 때문에 Bobby 역시 영원히 wait 하게 됩니다.

그래서 〈그림 5-8〉과 같은 실행 결과가 만들어 진 것입니다.

문제 5-3 해답

멀티 쓰레드가 아닌 프로그램에서는, 예컨대 take 메소드의 「케이크가 있으면 먹는다」고 하는 처리가 계속해서 이뤄지는데에 아무런 문제가 없습니다. 하지만 멀티 쓰레드 프로그램에서는 복수의 쓰레드가 take 메소드를 호출할 가능성이 있기 때문에 문제가 그렇게 간단하지 않습니다.

take 메소드가 실시하고 있는 「케이크가 있으면 먹는다」는 처리는 다음 2단계로 이루어져 있습니다.

(1) 「케이크가 있다」고 하는 판단

(2) 「먹는다」고 하는 처리

take 메소드를 synchronized로 하지 않았다고 합시다. 그러면 한 개의 쓰레드가 (1)을 실행한 후 (2)를 실행하기 전에 다른 쓰레드 B가 끼어 들어 (1)과 (2)를 실행해버릴 가능성이 있습니다. 즉,「케이크가 있다」고 판단하고 나서 실제로 「먹는다」는 처리를 실행하려는 사이에 다른 쓰레드가 「케이크가 있으면 먹는다」는 처리를 실행해 버릴 위험이 있다는 것입니다. 만일 테이블 위에 놓인 케이크가 마지막 한 개였다면 쓰레드 A는 먹을 수 있으리라 생각했던 케이크를 먹을 수 없게 됩니다.

take 메소드를 synchronized로 하면 take를 한 번에 실행할 수 있는 쓰레드가 한 개뿐이므로 앞에서 설명한 것 같은 쓰레드의 간섭은 사라집니다.

때문에 take 메소드를 synchronized로 하는 것입니다. put 메소드를 synchronized로 하는 것도 같은 이유입니다.

문제 5-4의 해답

〈리스트 5-1〉처럼 됩니다.

리스트 5-1 디버그 표시를 추가한 Table 클래스 (Table.java)

```
public class Table {
    private final String[] buffer;
    private int tail;  // 다음에 put할 장소
    private int head;  // 다음에 take할 장소
    private int count; // buffer 안의 케이크 수
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    public synchronized void put(String cake) throws InterruptedException {
        System.out.println(Thread.currentThread().getName()+"puts"+cake);
        while (count >= buffer.length) {
            System.out.println(Thread.currentThread().getName()+" wait BEGIN");
            wait();
            System.out.println(Thread.currentThread().getName() + " wait END");
        }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    public synchronized String take() throws InterruptedException {
        while (count <= 0) {
            System.out.println(Thread.currentThread().getName()+"wait BEGIN");
            wait();
            System.out.println(Thread.currentThread().getName()+"wait END");
        }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName()+"takes"+cake);
        return cake;
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ProducerConsumer/A5-4*

실행 예는 〈그림 5-1〉과 같습니다.

그림 5-1 리스트 5-1의 실행 예

```
(전략)
MakerThread-1    puts[ Cake No.8 by MakerThread -1 ]
MakerThread-2    puts[ Cake No.8 by MakerThread -2 ]
EaterThread-3    takes    [ Cake No.8 by MakerThread -1 ]
EaterThread-2    takes    [ Cake No.8 by MakerThread -2 ]
EaterThread-3    wait BEGIN    ← 테이블이 비어있어 EaterThread-3이 기다린다
EaterThread-1    wait BEGIN    ← 테이블이 비어있어 EaterThread-1도 기다린다
MakerThread-3    puts[ Cake No.10 by MakerThread -3 ]
                               ← 거기에 MakeThread-3이 케이크를 놓는다(notifyAll 한다)
EaterThread-3    wait END      ← EaterThread-3이 락을 취하고 다음 단계로 나아가
EaterThread-3    takes    [ Cake No.10 by MakerThread -3 ] ← 케이크를 먹는다
EaterThread-1    wait END      ← EaterThread-1이 락을 취하고 다음 단계로 나아가지만
EaterThread-1    wait BEGIN    ← 테이블이 비어있어 다시 기다린다
MakerThread-1    puts[ Cake No.11 by MakerThread -1 ]
                               ← 거기에 MakerThread-1이 케이크를 놓는다(notifyAll한다)
EaterThread-1    wait END      ← EaterThread-1이 락을 취하고 다음 단계로 나아가
EaterThread-1    takes    [ Cake No.11 by MakerThread -1 ] ← 케이크를 먹는다
EaterThread-2    wait BEGIN    ← 테이블이 비어있어 EaterThread-2가  기다린다
EaterThread-1    wait BEGIN    ← 테이블이 비어있어 EaterThread-1이 기다린다
MakerThread-2    puts[ Cake No.12 by MakerThread -2 ]
                               ← 거기에 MakerThread-2가 케이크를 놓는다(notifyAll한다)
EaterThread-2    wait END      ← EaterThread-2가 락을 취하고 다음 단계로 나아가
EaterThread-2    takes    [ Cake No.12 by MakerThread -2 ] ← 케이크를 먹는다
(이하 생략. CTRL+C로 종료)
```

문제 5-5의 해답

예를 들어 〈리스트 5-2〉와 같이 됩니다. clear 메소드에서는 head, tail, count 각 필드의 값을 0으로 하여 notifyAll합니다. notifyAll을 잊으면 put 메소드 안에서 「테이블이 비워질 것」을 기다리던 쓰레드가 있을 경우 그 쓰레드는 wait 상태에서 벗어나지 못합니다.

〈리스트 5-2〉에서는 buffer의 내용을 그대로 두었지만 다음과 같이 하여 null을 대입해도 상관없습니다.

```
for (int i = 0; i < buffer.length; i++) {
   buffer[i] = null;
}
```

clear 메소드를 사용한 예제로서 ClearThread 클래스(리스트 5-3)를 만들어 보았습니다. ClearThread는 약 1초 간격으로 clear을 호출하는 청소부입니다.

리스트 5-2 clear 메소드를 추가한 Table 클래스 (Table.java)

```
public class Table {
    private final String[] buffer;
    private int tail;  // 다음에 put할 장소
    private int head;  // 다음에 take할 장소
    private int count; // buffer 안의 케이크 수
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    public synchronized void clear()  {
        // 이 while문은 치워진 케이크를 표시하기 위한 것이며 없어도 상관없다
        while (count > 0) {
            String cake = buffer[head];
            System.out.println(Thread.currentThread().getName() +
                               " clears " + cake);
            head = (head + 1) % buffer.length;
            count--;
        }
        head = 0;
        tail = 0;
        count = 0;
        notifyAll();
    }
    public synchronized void put(String cake) throws InterruptedException {
        System.out.println(Thread.currentThread().getName()+"puts"+cake);
        while (count >= buffer.length) {
            wait();
        }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    public synchronized String take() throws InterruptedException {
        while (count <= 0) {
            wait();
        }
        String cake = buffer[head];
```

```
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName()+"takes"+cake);
        return cake;
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ProducerConsumer/A5-5

리스트 5-3 약 1초 간격으로 clear를 호출하는 청소부, ClearThread 클래스 (ClearThread.java)

```
public class ClearThread extends Thread {
    private final Table table;
    public ClearThread(String name, Table table) {
        super(name);
        this.table = table;
    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(1000);
                System.out.println("===== "+getName()+"clears=====");
                table.clear();
            }
        } catch (InterruptedException e) {
        }
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ProducerConsumer/A5-5

리스트 5-4 요리사와 손님과 청소부를 움직이는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);  // 케이크 3개까지 놓을 수 있는 테이블
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();
        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
        new ClearThread("ClearThread-0", table).start();
    }
}
```

⇒ 예제파일 경로 : 부록CD/src/ProducerConsumer/A5-5

문제 5-6의 해답

〈리스트 5-5〉와 같습니다. sleep를 사용하여 약 10초 쉰 다음 각 쓰레드에 대하여 interrupt 메소드를 호출하도록 만들었습니다. 이로써 프로그램이 종료합니다.

리스트 5-5 약 1초 후에 쓰레드를 종료시키는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);  // 케이크를 3개까지 놓을 수 있는 테이블을 만든다
        Thread[] threads = {
            new MakerThread("MakerThread-1", table, 31415),
            new MakerThread("MakerThread-2", table, 92653),
            new MakerThread("MakerThread-3", table, 58979),
            new EaterThread("EaterThread-1", table, 32384),
            new EaterThread("EaterThread-2", table, 62643),
            new EaterThread("EaterThread-3", table, 38327),
        };

        // 쓰레드의 기동
        for (int i = 0; i < threads.length; i++) {
            threads[i].start();
        }

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

        System.out.println("***** interrupt *****");

        // 인터럽트
        for (int i = 0; i < threads.length; i++) {
            threads[i].interrupt();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Pr●ducerC●nsumer/A5-6*

그림 5-2 실행 예

```
EaterThread-1 takes  [ Cake No.95 by MakerThread -3 ]
MakerThread-2 puts   [ Cake No.97 by MakerThread -2 ]
EaterThread-1 takes  [ Cake No.96 by MakerThread -1 ]
MakerThread-3 puts   [ Cake No.98 by MakerThread -3 ]
EaterThread-2 takes  [ Cake No.97 by MakerThread -2 ]
====ClearThread-0 clears====  ← 여기에서 테이블을 청소했다(케이크 No.98이 놓여 있었다)
ClearThread-0 clears [ Cake No.98 by MakerThread -3 ]
MakerThread-3 puts   [ Cake No.99 by MakerThread -3 ]
EaterThread-3 wait   [ Cake No.99 by MakerThread -3 ]
MakerThread-2 puts   [ Cake No.100 by MakerThread -2 ]
MakerThread-1 puts   [ Cake No.101 by MakerThread -1 ]
EaterThread-1 wait   [ Cake No.100 by MakerThread -2 ]
MakerThread-2 puts   [ Cake No.102 by MakerThread -2 ]
MakerThread-3 puts   [ Cake No.103 by MakerThread -3 ]
EaterThread-3 takes  [ Cake No.101 by MakerThread -1 ]
EaterThread-2 wait   [ Cake No.102 by MakerThread -2 ]
EaterThread-2 wait   [ Cake No.103 by MakerThread -3 ]
====ClearThread-0 clears==== ← 여기에서 테이블을 청소하려 했으나 케이크가 놓여있지 않았다
MakerThread-3 puts   [ Cake No.104 by MakerThread -3 ]
EaterThread-1 takes  [ Cake No.104 by MakerThread -3 ]
MakerThread-1 puts   [ Cake No.105 by MakerThread -1 ]
MakerThread-3 puts   [ Cake No.106 by MakerThread -3 ]
EaterThread-1 takes  [ Cake No.105 by MakerThread -1 ]
MakerThread-3 puts   [ Cake No.107 by MakerThread -3 ]
EaterThread-2 takes  [ Cake No.106 by MakerThread -3 ]
MakerThread-1 puts   [ Cake No.108 by MakerThread -1 ]
EaterThread-3 takes  [ Cake No.107 by MakerThread -3 ]
MakerThread-2 puts   [ Cake No.109 by MakerThread -2 ]
MakerThread-2 puts   [ Cake No.110 by MakerThread -2 ]
EaterThread-2 takes  [ Cake No.108 by MakerThread -1 ]
MakerThread-2 puts   [ Cake No.111 by MakerThread -2 ]
====ClearThread-0  clears==== ← 여기에서 테이블을 청소했다(케이크 No.109, 110, 111이 놓
                                여 있었다)
ClearThread-0 clears [ Cake No.109 by MakerThread -2 ]
ClearThread-0 clears [ Cake No.110 by MakerThread -2 ]
ClearThread-0 clears [ Cake No.111 by MakerThread -2 ]
MakerThread-3 puts   [ Cake No.112 by MakerThread -3 ]
MakerThread-3 puts   [ Cake No.113 by MakerThread -3 ]
EaterThread-1 takes  [ Cake No.112 by MakerThread -3 ]
EaterThread-3 takes  [ Cake No.113 by MakerThread -3 ]
MakerThread-1 puts   [ Cake No.114 by MakerThread -1 ]
EaterThread-1 takes  [ Cake No.114 by MakerThread -1 ]
(CTRL+C로 종료)
```

문제 5-7의 해답

execute 메소드가 예외 InterruptedException을 통보하도록 throws절을 붙이고 취소해도 되는 곳에 다음 문을 넣습니다.

```
if (Thread.interrupted() ) {
    throw new InterruptedException();
}
```

이렇게 하면 현재 움직이고 있는 쓰레드가 인터럽트 상태였다면 InterruptedException이 통보됩니다. 그리고 Thread.interrupted를 실행하고 있기 때문에 쓰레드는 인터럽트가 아닌 상태가 됩니다.

〈리스트 5-6〉은 무거운 처리를 실행하기 시작하지만 약 15초 후에 취소하는 프로그램이며, 그 실행 예가 〈그림 5-3〉입니다. Thread.interrupted로 인터럽트 상태를 체크하는 타이밍에서만 InterruptedException이 통보됩니다. ***** interrupt *****라고 하는 표시가 나온 뒤 바로 InterruptedException이 통보되는 것이 아닙니다.

리스트 5-6 메인 처리 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        // Host의 무거운 처리를 실행하는 쓰레드
        Thread executor = new Thread() {
            public void run() {
                System.out.println("Host.execute BEGIN");
                try {
                    Host.execute(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Host.execute END");
            }
        };

        // 기동한다
        executor.start();

        // 약 15초 휴식
        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {
```

```
        }

        // 취소 실행
        System.out.println("***** interrupt *****");
        executor.interrupt();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Pr●ducerC●nsumer/A5-7*

리스트 5-7 무거운 처리 (H●st.java)

```
public class Host {
    public static void execute(int count) throws InterruptedException {
        for (int i = 0; i < count; i++) {
            if (Thread.interrupted()) {
                throw new InterruptedException();
            }
            doHeavyJob();
        }
    }
    private static void doHeavyJob() {
        // 이하는
        // 「취소 불가능한 무거운 처리」의 대용
        // 약 10초간 도는 루프
        System.out.println("doHeavyJob BEGIN");
        long start = System.currentTimeMillis();
        while (start + 10000 > System.currentTimeMillis()) {
            // busy loop
        }
        System.out.println("doHeavyJob END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Pr●ducerC●nsumer/A5-7*

그림 5-3 실행 예

```
Host.execute BEGIN                    ← execute 실행 개시
doHeavyJob BEGIN                      ← 무거운 처리 doHeavyJob 1번째 실행 개시
doHeavyJob END                        ← 1번째 실행 개시 약 10초 후 실행 종료
doHeavyJob BEGIN                      ← 무거운 처리 doHeavyJob 2번째 실행 개시
***** interrupt *****                 ← 여기에서 메인 쓰레드가 interrupt 실행
doHeavyJob END                        ← 2번째 실행 개시 약 10초 후 실행 종료
java.lang.InterruptedException        ← catche절 안에서 콜 스택 표시
   at Host.execute(Host.java:5)
   at Main$1.run(Main.jaba:8)
Host.execute END                      ← execute 실행 종료
```

문제 5-8의 해답

notify 메소드는 wait 셋 안에서 대기하고 있는 쓰레드들 중에서 한 개만 깨웁니다. 따라서 **관계가 없는 쓰레드가 wait 셋에 들어가 있을 때에 종종 notify로 인해 그 쓰레드가 깨어나면 notify의 통보는 잃어버리게 됩니다.** 이것을 실증하는 프로그램을 제시합니다.

〈리스트 5-8〉의 LazyThread 클래스는 Table의 인스턴스 상에서 wait하지만 실제로 아무 처리도 하지 않는 쓰레드입니다. 〈리스트 5-9〉의 Main 클래스가 LazyThread의 쓰레드를 섞으면 〈그림 5-4〉 실행 예에서처럼 프로그램이 중간에 멈춰버립니다.

이것은 notify 메소드의 호출이 LazyThreaddml 쓰레드를 깨우는데 사용되어 버렸기 때문입니다. Table 클래스에서 notify 메소드가 아닌 notifyAll 메소드를 사용하면 설령 LazyThread가 섞여 있다 하더라도 프로그램이 서지 않고 계속 동작합니다.

리스트 5-8 아무것도 하지 않는 쓰레드 (LazyThread.java)

```
public class LazyThread extends Thread {
    private final Table table;
    public LazyThread(String name, Table table) {
        super(name);
        this.table = table;
    }
    public void run() {
        while (true) {
            try {
                synchronized (table) {
                    table.wait();
                }
                System.out.println(getName() + " is notified!");
            } catch (InterruptedException e) {
            }
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ProducerConsumer/A5-8*

잠깐! 한 마디 : 락을 취하는 객체를 감춘다

문제 5-8에서 알 수 있듯이 락을 취하는 객체의 wait 셋 안에 아무 관계도 없는 쓰레드가 섞이는 것은 바람직하지 않습니다. wait 셋에 관계 없는 쓰레드가 섞이지 않도록 하기 위해서는 락을 취하는 객체를 감춰두는 것이 좋습니다. 즉, 락을 취하는 객체를 새로 만들어 로컬 변수나 private 필드에 보관해 두는 것입니다.

리스트 5-9 아무것도 하지 않는 쓰레드까지 뒤섞인 상태에서 요리사와 손님을 움직이는 Main 클래스 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);  // 케이크를 3개까지 놓을 수 있는 테이블을 만든다
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();
        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
        new LazyThread("LazyThread-1", table).start();
        new LazyThread("LazyThread-2", table).start();
        new LazyThread("LazyThread-3", table).start();
        new LazyThread("LazyThread-4", table).start();
        new LazyThread("LazyThread-5", table).start();
        new LazyThread("LazyThread-6", table).start();
        new LazyThread("LazyThread-7", table).start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Pr●ducerC●nsumer/A5-8*

그림 5-4 실행 예

```
LazyThread-5 is notified!
EaterThread-3 takes [ Cake No.25 by MakerThread-2 ]
LazyThread-6 is notified!
MakerThread-2 puts [ Cake No.26 by MakerThread-2 ]
LazyThread-7 is notified!
EaterThread-3 takes [ Cake No.26 by MakerThread-2 ]
LazyThread-1 is notified!
MakerThread-2 puts [ Cake No.27 by MakerThread-2 ]
LazyThread-2 is notified!
MakerThread-2 puts [ Cake No.28 by MakerThread-2 ]
MakerThread-2 puts [ Cake No.29 by MakerThread-2 ]
MakerThread-3    puts [ Cake No.30 by MakerThread-3 ] ← 도중에 정지해 버린다
(CTRL+C로 종료)
```

문제 5-9의 해답

Something.method(long)은 **Thread.sleep(long)와 똑같은 기능을 합니다.** 즉, 인수로 지정한 시간(밀리초)만큼 일시 정지합니다.

sleep를 인수 0으로 호출하면 0밀리 초 동안 실행을 정지하지만 wait을 인수 0으로 호출하면 무한정 타임아웃되므로 if문에서 0을 제외하였습니다. 또한 wait이 notify나 notifyAll로 인해 중단되지 않도록 락을 취하는 인스턴스를 메소드 안에서 생성하고 있습니다. 외부에서 이 인스턴스를 구할 방법이 없기 때문에 notify나 notifyAll하는 것은 불가능합니다.

〈리스트 5-10〉은 Something.method를 사용하여 약 3초간 「휴식」하는 프로그램입니다. 실행 결과는 〈그림 5-5〉와 같습니다.

리스트 5-10 시험 삼아 동작시켜 봅시다 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("BEGIN");
        try {
            Something.method(3000);
        } catch (InterruptedException e) {
        }
        System.out.println("END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/Pr●ducerC●nsumer/A5-9*

그림 5-5 실행 결과

```
BEGIN    ← 바로 표시
END      ← 약 3초 후에 표시
```