Chapter 11

문제 11-1의 해답

×… (1) ThreadLocal의 인스턴스는 3개 생성된다.

➜ ThreadLocal의 인스턴스는 한 개만 생성됩니다. 그 한 개 인스턴스는 Log의 클래스 필드 tsLogCollection에 대입됩니다.

○… (2) PrintWriter의 인스턴스는 3개 생성된다.

➜ PrintWriter의 인스턴스는 TSLog 클래스의 인스턴스 필드 writer에 대입됩니다. TSLog의 인스턴스는 ClientThread 인스턴스의 수(3개)와 같으므로 PrintWriter의 인스턴스도 3개가 됩니다.

○… (3) ThreadLocal의 set 메소드는 3번 호출된다.

➜ ThreadLocal의 set 메소드는 ClientThread의 쓰레드가 처음 getTSLog 메소드를 호출했을 때 호출됩니다. 따라서 ClientThread 인스턴스의 수(3개)와 같은 회수만큼 호출됩니다.

×… (4) ThreadLocal의 get 메소드는 3번 호출된다.

➜ ThreadLocal의 set 메소드는 쓰레드가 println 메소드나 close 메소드를 호출할 때마다 호출됩니다. println은 쓰레드마다 10번 호출되고 close는 쓰레드마다 한 번 호출됩니다. 따라서 3번이 아니라 11×3(쓰레드의 개수) = 33번 호출됩니다.

○… (5) TSLog의 인스턴스는 3개 생성된다.

➜ ClientThread 인스턴스의 수(3개)와 같은 수만큼 생성됩니다.

×… (6) Log의 인스턴스는 3개 생성된다.

➜ Log의 인스턴스는 한 개도 생성되지 않습니다.

문제 11-2의 해답

(1)의 해답

TSLog 클래스의 println 메소드와 close 메소드는 여러 쓰레드로부터 호출되는 일이 없

기 때문입니다. TSLog의 인스턴스는 ThreadLocal 클래스에 의해 관리되며 쓰레드 고유의 정보로서 취급됩니다. 이것은 어떤 쓰레드가 이용하는 TSLog의 인스턴스는 정해져 있음을 의미합니다. 즉, 한 개의 쓰레드가 다른 쓰레드에 대응한 TSLog의 인스턴스를 이용하는 것은 불가능합니다.

(2)의 해답

Log 클래스의 println 메소드와 close 메소드는 복수의 쓰레드로부터 호출됩니다. 그러나 Log 클래스는 복수의 쓰레드에서 액세스하면 곤란한 필드를 가지고 있지 않습니다. 즉, 보호해야 할 상태를 가지고 있지 않으므로 synchronized로 할 필요가 없습니다.

TsLogCollection 필드에 복수의 쓰레드가 액세스하고 있지만 java.lang.ThreadLocal 클래스가 쓰레드 세이프한 클래스이므로 안전성은 지켜집니다.

잠깐! 한 마디 : java.lang.ThreadLocal 클래스의 구현

ThreadLocal 클래스는 쓰레드 세이프이지만 항상 synchronized로 배타제어를 하고 있는 것은 아닙니다. synchronized로 배타제어를 하고 있는 지 아닌지는 클래스 라이브러리의 구현에 의존합니다.
synchronized를 사용하지 않고 쓰레드 세이프한 ThreadLocal을 구현할 수 있습니다. 예를 들어 java.lang.Thread 클래스 내부에 Thread-Specific Storage를 위한 정보를 주고 ThreadLocal로부터 그 정보에 액세스하게 합니다. 자세한 내용은 다음 웹 페이지를 참조하여 주세요.

:: Exploiting ThreadLocal to enhance scalability
http://www-128.ibm.com/developerworks/java/library/j-threads3.html

문제 11-3의 해답

다음에 제시하는 해답에서는 쓰레드의 종료를 감시하는 또 다른 쓰레드를 준비하는 방법을 사용했습니다.

수정한 ClientThread 클래스와 Log 클래스는 각각 〈리스트 11-1〉과 〈리스트 11-2〉입니다. Log 클래스 안에서 TSLog의 인스턴스를 새로 만들 때에 새로운 쓰레드(watcher)를 기동합니다. watcher는 TSLog에 로그를 출력하는 쓰레드(target)의 종료를 join을 사용하여 감시합니다. target이 종료하면 watcher의 쓰레드는 join으로부터 돌아오므로 거기에서 close 합니다.

이렇게 하면 TSLog의 인스턴스가 복수의 쓰레드(target과 watcher)로부터 액세스되는

것처럼 보입니다. 하지만 watcher가 close 메소드를 호출하는 시점은 이미 target 쓰레드가 종료한 상태이므로 복수의 쓰레드가 액세스하여 간섭 받을 위험은 없습니다. 실행 예는 〈그림 11-1〉과 같습니다.

리스트 11-1 수정 후 ClientThread 클래스 (ClientThread.java)

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName() + " BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        // Log.close()는 이제 불필요
        System.out.println(getName() + " END");
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadSpecificStorage/A11-3*

리스트 11-2 수정 후 Log 클래스 (Log.java)

```
public class Log {
    private static final ThreadLocal<TSLog> tsLogCollection = new
ThreadLocal<TSLog>();

    // 로그를 적는다
    public static void println(String s) {
        getTSLog().println(s);
    }

    // 로그를 닫는다
    public static void close() {
        getTSLog().close();
    }

    // 쓰레드 고유의 로그를 취한다
    private static TSLog getTSLog() {
        TSLog tsLog = tsLogCollection.get();

        // 그 쓰레드로부터의 호출이 시작되면 새로 작성하여 등록한다
        if (tsLog == null) {
```

```
            tsLog = new TSLog(Thread.currentThread().getName() + "-log.txt");
            tsLogCollection.set(tsLog);
            startWatcher(tsLog);
        }

        return tsLog;
    }

    // 쓰레드의 종료를 기다리는 쓰레드를 기동한다
    private static void startWatcher(final TSLog tsLog) {
        // 종료를 감시 당하는 쪽 쓰레드
        final Thread target = Thread.currentThread();
        // target을 감시하는 쓰레드
        final Thread watcher = new Thread() {
            public void run() {
                System.out.println("startWatcher for " +
                        target.getName() + " BEGIN");
                try {
                    target.join();
                } catch (InterruptedException e) {
                }
                tsLog.close();
                System.out.println("startWatcher for"+target.getName()+"END");
            }
        };
        // 감시 개시
        watcher.start();
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadSpecificStorage/A11-3*

그림 11-1 실행 예

```
Alice BEGIN
Bobby BEGIN
Chris BEGIN
startWatcher for Alice   BEGIN   ← Alice용 watcher 쓰레드 개시
startWatcher for Bobby   BEGIN   ← Bobby용 watcher 쓰레드 개시
startWatcher for Chris   BEGIN   ← Chris용 watcher 쓰레드 개시
Bobby END
startWatcher for Bobby   END     ← Bobby용 watcher 쓰레드 종료
Alice END
startWatcher for Alice   END     ← Alice용 watcher 쓰레드 종료
Chris END
startWatcher for Chris   END     ← Chris용 watcher 쓰레드 종료
```

문제 11-4의 해답

이 프로그램에서는 생성자를 실행하고 있는 쓰레드가 메인 쓰레드이기 때문입니다. ClientThread의 run 메소드를 실행하고 있는 메소드는 메인 쓰레드로부터 기동된 새로운 쓰레드이며 메인 쓰레드와는 다릅니다.

이 문제는 「ClientThread의 생성자를 실행하고 있는 쓰레드와 run 메소드를 실행하고 있는 쓰레드가 다르다」는 사실을 이해하고 있는지 확인하는 문제입니다.

주의 … 문제문의 프로그램을 실행하면 main-log-txt라고 하는 파일이 만들어집니다. 파일명에 main이 붙는 것은 메인 쓰레드의 이름이 main이기 때문입니다. 그러나 메인 쓰레드에서는 close를 호출하지 않으므로 "constructor is called."라는 문자열이 올바로 보관되어 있지 않을 수도 있습니다.

문제 11-5의 해답

가장 자주 사용하는 것은 「쓰레드 이름」입니다. 쓰레드 이름은 java.lang.Thread의 인스턴스 안에 위치하며 getName 메소드로부터 구할 수 있습니다.

현재 쓰레드의 이름은 Thread.currentThread().getName 식(式)으로 구할 수 있습니다. 쓰레드 이름 외에도 다음과 같은 내용이 쓰레드 고유의 정보가 됩니다. 괄호 안은 그 정보를 얻기 위한 메소드입니다.

:: 쓰레드의 우선도(getPriority)

:: 쓰레드 그룹(getThreadGroup)

:: 인터럽트 상태(isInterrupted)

:: 데몬 쓰레드인지 아닌지(isDaemon)

:: 생존 여부(isAlive)

:: 쓰레드를 식별하는 long 값(getId)

:: 쓰레드의 상태(getState)

:: 캐치되지 않은 예외의 핸들러(getUncaughtExceptionHandler)

문제 11-6의 해답

태스크가 10개라도 실제 그것을 실행하는 쓰레드가 3개뿐이기 때문입니다.

문제 11-6의 프로그램(리스트 11-8)에서는 태스크를 실행하는 ExecutorService를 Executors.newFixedThreadPool(3)을 이용해 구하고 있습니다. 이 때는 3개라고 하는 일정 수의 쓰레드를 돌려가면서 사용합니다. Thread-Specific Storage 패턴에서는 실행하고 있는 쓰레드마다 고유 영역을 확보하기 때문에 쓰레드가 3개이면 고유 영역도 3개밖에 확보되지 않습니다. 그 결과 로그파일도 3개가 되는 것입니다.

Executors.newFixedThreadPool 메소드 대신 Executors.newCachedThreadPool 메소드를 사용하여 ExecutorsService를 확보하면 예제 프로그램의 범위에서는 로그파일이 가능해 보입니다. 하지만 Executors.newCachedThreadPool로 구할 수 있는 ExecutorsService를 이용한 경우에도 쓰레드가 재사용되면 로그파일의 개수는 줄어들고 맙니다. 〈리스트 11-3〉처럼 모든 태스크가 확실하게 다른 쓰레드에 의해 실행됨을 보증하지 않는 한 Thread-Specific Storage 패턴은 제대로 작동하지 않습니다. 이는 java.lang. ThreadLocal을 이용할 때 기억해야 할 중요한 제어 사항입니다. 결국 java.lang.ThreadLocal을 사용할 때에는 어떠한 태스크가 어떠한 쓰레드에서 실행되는가라고 하는 쓰레드 설계에 주의를 기울여야 할 것입니다.

리스트 11-3 모든 태스크가 서로 다른 쓰레드에서 실행되도록 한 Main 클래스 (Main.java)

```
public class Main {
    private static final int TASKS = 10;
    public static void main(String[] args) {
        for (int t = 0; t < TASKS; t++) {
            // 로그를 입력하는 태스크
            Runnable printTask = new Runnable() {
                public void run() {
                    Log.println("Hello!");
                    Log.close();
                }
            };
            // 태스크의 실행
            new Thread(printTask).start();
        }
    }
}
```

⇒ *예제파일 경로 : 부록CD/src/ThreadSpecificStorage/A11-6*