Chapter 14. 멀티 스레드

14.1 멀티 스레드 개념

운영체제는 실행 중인 프로그램을 **프로세스(process)** 로 관리한다. 멀티 태스킹(multi tasking)은 두 가지 이상의 작업을 동시에 처리하는 것을 말하는데, 이때 운영체제는 **멀티 프로세스**를 생성해서 처리한다. 하지만 멀티 태스킹이 꼭 멀티 프로세스를 뜻하지는 않는다.

하나의 프로세스가 두 가지 이상의 작업을 처리할 수도 있는데 이는 **멀티 스레드(multi thread)** 가 있기 때문이다. **스레드(thread)** 는 코드의 실행 흐름을 말하는데, 프로세스 내에 스레드가 두 개라면 두 개의 코드 실행 흐름이 생긴다는 의미이다.

멀티 프로세스가 프로그램 단위의 멀티 태스킹이라면 멀티 스레드는 프로그램 내부에서의 멀티 태스킹이라고 볼 수 있다.

멀티 프로세스들은 서로 독립적이므로 하나의 프로세스에서 오류가 발생해도 다른 프로세스에게 영향을 미치지 않는다. 하지만 멀티 스레드는 프로세스 내부에서 생성되기 때문에 하나의 스레드가 예외를 발생시키면 프로세스가 종료되므로 다른 스레드에게 영향을 미친다. 그렇기 때문에 멀티 스레드를 사용할 경우에는 예외 처리에 만전을 기해야 한다.

14.2 메인 스레드

모든 자바 프로그램은 **메인 스레드(main thread)** 가 **main()** 메소드를 실행하면서 시작된다. 메인 스레드는 필요에 따라 추가 작업 스레드들을 만들어서 실행시킬 수 있다.
싱글 스레드에서는 메인 스레드가 종료되면 프로세스도 종료되지만 멀티 스레드에서는 *실행 중인 스레드가 하나라도 있다면* 프로세스는 종료되지 않는다. 메인 스레드가 작업 스레드보다 먼저 종료되더라도 작업 스레드가 계속 실행 중이라면 프로세스는 종료되지 않는다.

14.3 작업 스레드 생성과 실행

자바 프로그램은 메인 스레드가 반드시 존재하기 때문에 메인 작업 이외에 *추가적인 작업 수만큼 스레드를 생성*하면 된다. 자바는 작업 스레드도 객체로 관리하므로 클래스가 필요하다. Thread 클래스로 직접 객체를 생성해도 되지만, 하위 클래스를 만들어 생성할 수도 있다.

Thread 클래스로 직접 생성

BeepPrintExample.java

```
package ch14.sec03.exam02;

import java.awt.Toolkit;

public class BeepPrintExample {
    public static void main(String[] args) {

        Thread thread = new Thread(new Runnable() { // 작업 스레드 생성
            @Override
```

```
            public void run() {     // 작업 스레드가 실행 (동시)
                Toolkit toolkit = Toolkit.getDefaultToolkit();
                for(int i=0; i<5; i++) {
                    toolkit.beep();
                    try { Thread.sleep(500); } catch(Exception e) {}
                }
            }
        });

        thread.start();
        // start() 메소드 호출해야 작업 스레드 실행됨.
        // Runnable의 run() 메소드 실행

        for(int i=0; i<5; i++) {    // 메인 스레드가 실행 (동시)
            System.out.println("띵");
            try { Thread.sleep(500); } catch(Exception e) {}
        }
    }
}
```

Thread 자식 클래스로 생성

BeepPrintExample.java

```
package ch14.sec03.exam03;

import java.awt.Toolkit;

public class BeepPrintExample {
    public static void main(String[] args) {

        Thread thread = new Thread() {  // 작업 스레드
            @Override
            public void run() {     // 작업 스레드가 실행
                Toolkit toolkit = Toolkit.getDefaultToolkit();
                for(int i=0; i<5; i++) {
                    toolkit.beep();
                    try { Thread.sleep(500); } catch(Exception e) {}
                }
            }
        };

        thread.start();

        for(int i=0; i<5; i++) {    // 메인 스레드가 실행
            System.out.println("띵");
            try { Thread.sleep(500); } catch(Exception e) {}
        }
    }
}
```

14.4 스레드 이름

스레드는 자신의 이름을 갖고 있다. 메인 스레드는 'main'이라는 이름을 가지고 있고, 작업 스레드는 자동적으로 'Thread-n'이라는 이름을 가진다. 작업 스레드의 이름을 다른 이름으로 설정하고 싶다면 `Thread` 클래스의 `setName()` 메소드를 사용하면 된다.

```
thread.setName("스레드 이름");
```

스레드 이름은 디버깅할 때 어떤 스레드가 작업을 하는지 조사할 목적으로 주로 사용된다. 현재 코드를 어떤 스레드가 실행하고 있는지 확인하려면 정적 메소드인 `currentThread()`로 스레드 객체의 참조를 얻은 다음 `getName()` 메소드로 이름을 출력해보면 된다.

```
Thread thread = Thread.currentThread();
System.out.println(thread.getName());
```

14.5 스레드 상태

스레드 객체를 **생성(NEW)** 하고, `start()` 메소드를 호출하면 곧바로 실행되는 것이 아니라 **실행 대기 상태(RUNNABLE)** 가 된다. 실행 대기하는 스레드는 CPU 스케쥴링에 따라 CPU를 점유하고 `run()` 메소드를 실행한다. 이때를 **실행(RUNNING) 상태**라고 한다. 실행 스레드는 `run()` 메소드를 모두 실행하기 전에 스케줄링에 의해 다시 실행 대기 상태로 돌아갈 수 있다. 그리고 다른 스레드가 실행 상태가 된다.

이렇게 스레드는 실행 대기 상태와 실행 상태를 번갈아 가면서 자신의 `run()` 메소드를 조금씩 실행한다. 실행 상태에서 `run()` 메소드가 종료되면 더 이상 실행할 코드가 없기 때문에 스레드의 실행은 멈추게 되는데, 이 상태를 **종료 상태(TERMINATED)** 라고 한다.

실행 상태에서 **일시 정지 상태**로 가기도 하는데, 이는 스레드가 실행할 수 없는 상태를 말한다. 스레드가 다시 실행 상태로 가기 위해서는 일시 정지 상태에서 실행 대기 상태로 가야만 한다.

- 일시 정지로 보냄
    - `sleep(long millis)`
    - `join()`
    - `wait()`
- 일시 정지에서 벗어남
    - `interrupt()`
    - `notify()`
    - `notifyAll()`
- 실행 대기로 보냄
    - `yield()`

주어진 시간 동안 일시 정지

- 실행 중인 스레드를 일정 시간 멈추게 하고 싶을 때
- Thread의 `sleep()` 메소드

SleepExample.java

```
package ch14.sec05.exam01;

import java.awt.Toolkit;

public class SleepExample {
    public static void main(String[] args) {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        for(int i=0; i<10; i++) {
            toolkit.beep();
            try {   // 일시 정지 상태에선 InterruptedException이 발생할 수 있어 예외
처리 필요
                Thread.sleep(3000);     // 3초 주기
            } catch(InterruptedException e) {
            }
        }
    }
}
```

다른 스레드의 종료를 기다림

- 다른 스레드가 종료될 때까지 기다렸다가 실행해야 하는 경우
- Thread의 `join()` 메소드

SumThread.java

```
package ch14.sec05.exam02;

public class SumThread extends Thread {
    private long sum;

    public long getSum() {
        return sum;
    }

    public void setSum(long sum) {
        this.sum = sum;
    }

    @Override
    public void run() {
        for(int i=1; i<=100; i++) {
            sum+=i;
        }
```

```
    }
}
```

JoinExample.java

```
package ch14.sec05.exam02;

public class JoinExample {
    public static void main(String[] args) {
        SumThread sumThread = new SumThread();
        sumThread.start();
        try {
            sumThread.join();
            // main() 실행 X, sumThread 끝날 때까지 일시정지 상태
        } catch (InterruptedException e) {
        }
        System.out.println("1~100 합: " + sumThread.getSum());
    }
}
```

다른 스레드에게 실행 양보

- 다른 스레드에게 실행을 양보하고 자신은 실행 대기 상태로 가는 것이 프로그램 성능에 도움이 될 때 사용.
- Thread의 `yield()` 메소드

WorkThread.java

```
package ch14.sec05.exam03;

public class WorkThread extends Thread {
    //필드
    public boolean work = true;

    //생성자
    public WorkThread(String name) {
        setName(name);
    }

    //메소드
    @Override
    public void run() {
        while(true) {
            if(work) {
                System.out.println(getName() + ": 작업처리");
            } else {
                Thread.yield();
            }
        }
```

```
    }
}
```

YieldExample.java

```
package ch14.sec05.exam03;

public class YieldExample {
    public static void main(String[] args) {
        WorkThread workThreadA = new WorkThread("workThreadA");
        WorkThread workThreadB = new WorkThread("workThreadB");
        workThreadA.start();
        workThreadB.start();

        try { Thread.sleep(5000); } catch (InterruptedException e) {}
        workThreadA.work = false;

        try { Thread.sleep(10000); } catch (InterruptedException e) {}
        workThreadA.work = true;
    }
}
```

14.6 스레드 동기화

멀티 스레드는 하나의 객체를 공유해서 작업할 수도 있는데, 이때 다른 스레드에 의해 객체 내부 데이터가 쉽게 변경될 수 있기 때문에 의도했던 것과는 다른 결과가 나올 수 있다.
스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없도록 하려면 스레드 작업이 끝날 때까지 객체에 잠금을 걸면 된다. 이를 위해 자바는 동기화(synchronized) 메소드와 블록을 제공한다. 객체 내부에 동기화 메소드와 동기화 블록이 여러 개가 있다면 스레드가 이들 중 하나를 실행할 때 다른 스레드는 해당 메소드는 물론이고 다른 동기화 메소드 및 블록도 실행할 수 없다. 하지만 일반 메소드는 실행이 가능하다.

동기화 메소드 및 블록 선언

```
public synchronized void method() {
    // 단 하나의 스레드만 실행하는 영역
}
```

스레드가 동기화 메소드를 실행하는 즉시 객체는 잠금이 일어나고, 메소드 실행이 끝나면 잠금이 풀린다. 메소드 전체가 아닌 일부 영역을 실행할 때만 객체 잠금을 걸고 싶다면 다음과 같이 동기화 블록을 만들면 된다.

```
public void method() {
    // 여러 스레드가 실행할 수 있는 영역

    synchronized(공유객체) {
        // 단 하나의 스레드만 실행하는 영역
```

```
    }

    // 여러 스레드가 실행할 수 있는 영역
}
```

wait()과 notify()를 이용한 스레드 제어

두 개의 스레드를 교대로 번갈아 가며 실행할 때 사용된다. `notify()` 메소드는 `wait()`에 의해 일시 정지된 스레드 중 한 개를 실행 대기 상태로 만들고, `notifyAll()`은 `wait()`에 의해 일시 정지된 모든 스레드를 실행 대기 상태로 만든다. 주의할 점은 이 두 메소드는 동기화 메소드 또는 동기화 블록 내에서만 사용할 수 있다는 것이다.

WorkObject.java

```
package ch14.sec06.exam02;

public class WorkObject {
    public synchronized void methodA() {    // 동기화 메소드
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + ": methodA 작업 실행");
        notify();   // 다른 스레드를 실행 대기 상태로 만듦
        try {
            wait();     // 자신의 스레드는 일시 정지 상태로 만듦
        } catch (InterruptedException e) {
        }
    }

    public synchronized void methodB() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + ": methodB 작업 실행");
        notify();
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
}
```

ThreadA.java

```
package ch14.sec06.exam02;

public class ThreadA extends Thread {
    private WorkObject workObject;

    public ThreadA(WorkObject workObject) {     // 공유 작업 객체를 받음
        setName("ThreadA");     // 스레드 이름 변경
        this.workObject = workObject;
    }
```

```
    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodA();   // 동기화 메소드 호출
        }
    }
}
```

ThreadB.java

```
package ch14.sec06.exam02;

public class ThreadB extends Thread {
    private WorkObject workObject;

    public ThreadB(WorkObject workObject) {     // 공유 작업 객체를 받음
        setName("ThreadB");     // 스레드 이름 변경
        this.workObject = workObject;
    }

    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodB();   // 동기화 메소드 호출
        }
    }
}
```

WaitNotifyExample.java

```
package ch14.sec06.exam02;

public class WaitNotifyExample {
    public static void main(String[] args) {
        WorkObject workObject = new WorkObject();   // 공유 작업 객체 생성

        // 작업 스레드 생성 및 실행
        ThreadA threadA = new ThreadA(workObject);
        ThreadB threadB = new ThreadB(workObject);

        threadA.start();
        threadB.start();
    }
}
```

결과

```
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
...
```

14.7 스레드 안전 종료

조건 이용

PrintThread.java

```
package ch14.sec07.exam01;

public class PrintThread extends Thread {
    private boolean stop;

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    @Override
    public void run() {
        while(!stop) {      // stop 필드값에 따라 반복 여부 결정
            System.out.println("실행 중");
        }
        System.out.println("리소스 정리");
        System.out.println("실행 종료");
    }
}
```

SafeStopExample.java

```
package ch14.sec07.exam01;

public class SafeStopExample {
    public static void main(String[] args) {
        PrintThread printThread = new PrintThread();
        printThread.start();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        printThread.setStop(true);
        // printThread 종료 위해 stop 필드값 변경
```

```
    }
}
```

interrupt() 메소드 이용

스레드가 일시 정지 상태에 있을 때 InterruptedException 예외를 발생시키는 역할을 한다. 스레드가 실행 대기/실행 상태일 때에는 `interrupt()` 메소드가 호출되어도 예외는 발생하지 않는다.

PrintThread.java

```
package ch14.sec07.exam02;

public class PrintThread extends Thread {
    public void run() {
        try {
            while(true) {
                System.out.println("실행 중");
                Thread.sleep(1);        // 일시 정지 만듦 (InterruptedException 발생할 수 있도록)
            }
        } catch(InterruptedException e) {
        }
        System.out.println("리소스 정리");
        System.out.println("실행 종료");
    }
}
```

InterruptExample.java

```
package ch14.sec07.exam02;

public class InterruptExample {
    public static void main(String[] args) {
        Thread thread = new PrintThread();
        thread.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }

        thread.interrupt();
    }
}
```

일시 정지를 만들지 않고도 Thread의 `interrupted()`와 `isInterrupted()` 메소드로 `interrupt()` 메소드 호출 여부를 확인할 수 있다.

14.8 데몬 스레드

데몬(daemon) 스레드는 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드다. 주 스레드가 종료되면 데몬 스레드도 따라서 자동으로 종료된다. 스레드를 데몬으로 만들기 위해서는 주 스레드가 데몬이 될 스레드의 `setDaemon(true)`를 호출하면 된다.

AutoSaveThread.java

```
package ch14.sec08;

public class AutoSaveThread extends Thread {
    public void save() {
        System.out.println("작업 내용을 저장함.");
    }

    @Override
    public void run() {
        while(true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
            save();
        }
    }
}
```

DaemonExample.java

```
package ch14.sec08;

public class DaemonExample {
    public static void main(String[] args) {
        AutoSaveThread autoSaveThread = new AutoSaveThread();
        autoSaveThread.setDaemon(true); // AutoSaveThread를 데몬 스레드로 만듦
        autoSaveThread.start();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        System.out.println("메인 스레드 종료");
    }
}
```

결과

```
작업 내용을 저장함.
작업 내용을 저장함.
메인 스레드 종료
```

14.9 스레드 풀

스레드 풀(ThreadPool)은 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐에 들어오는 작업들을 스레드가 하나씩 맡아 처리하는 방식이다. 작업 처리가 끝난 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리한다. 이렇게 하면 작업량이 증가해도 스레드의 개수가 늘어나지 않아 애플리케이션의 성능이 급격히 저하되지 않는다.