Thread & Multitasking

514770-1
2017년 봄학기
5/10/2017
박경신

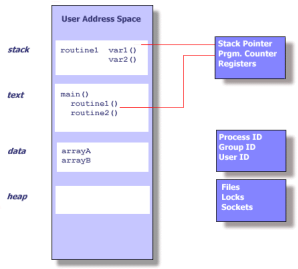
멀티태스킹과 스레드 개념

- 멀티태스킹(Multi-tasking)
  - 하나의 응용프로그램이 여러 개의 작업(태스크)을 동시에 처리
- 스레드 (Thread)
  - 마치 바늘이 하나의 실(thread)을 가지고 바느질하는 것과 자바의 스레드는 일맥 상통함

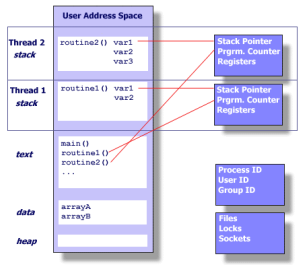
다림질하면서 이어폰으로 전화하는 주부
운전하면서 화장하는 운전자
제품의 판독과 포장 작업의 두 기능을 갖춘 기계

Process vs Thread

- 프로세스(process)
  - 운영체제로부터 process를 할당받고, 운영되기 위해 필요한 주소공간, 메모리 등 자원을 할당받는다.
- 스레드(thread)
  - 한 프로세스 내에서 동작되는 여러 실행의 흐름으로, 같은 process 내의 주소공간이나 자원들을 thread끼리 공유하면서 실행된다.



UNIX PROCESS

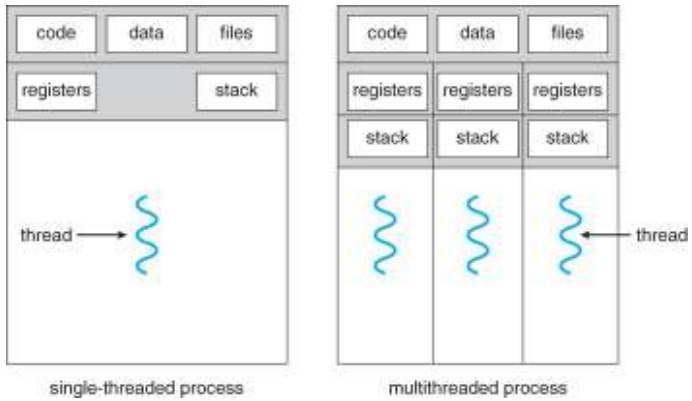


THREADS WITHIN A UNIX PROCESS

https://computing.llnl.gov/tutorials/pthreads/

Single vs Multithreaded Processes

Threads are light-weight processes within a process

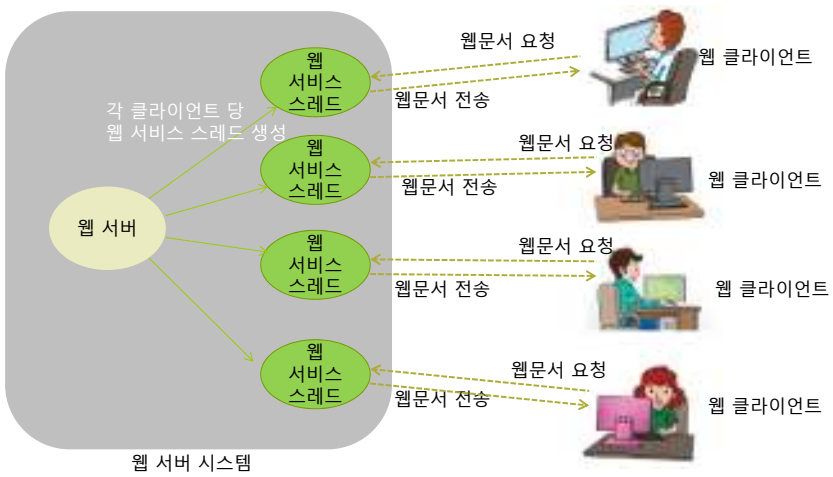


https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

스레드와 멀티스레딩

- 스레드
  - 프로그램 코드를 이동하면서 실행하는 하나의 제어
- 자바의 멀티태스킹
  - 멀티스레딩(Multi-threading)만 가능
    - 자바에 프로세스 개념은 존재하지 않고, 스레드 개념만 존재
    - 스레드는 실행 단위
    - 스레드는 스케쥴링 단위
  - 하나의 응용프로그램은 여러 개의 스레드로 구성 가능
    - 스레드 사이의 통신 오버헤드가 크지 않음

웹 서버의 멀티스레딩 사례



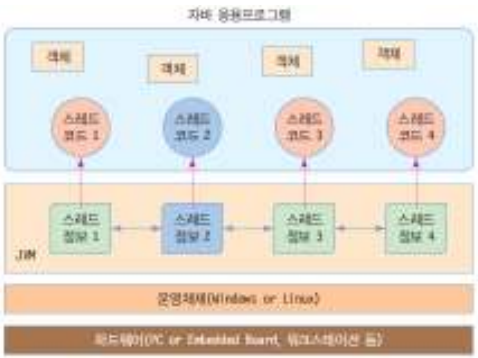
자바 스레드(Thread)란?

- 자바 스레드
  - 자바 가상 기계(JVM)에 의해 스케쥴되는 실행 단위의 코드 블럭
  - 스레드의 생명 주기는 JVM에 의해 관리됨
    - JVM은 스레드 단위로 스케쥴링
- JVM과 멀티스레드의 관계
  - 하나의 JVM은 하나의 자바 응용프로그램만 실행
    - 자바 응용프로그램이 시작될 때 JVM이 함께 실행됨
    - 자바 응용프로그램이 종료하면 JVM도 함께 종료함
  - 하나의 응용프로그램은 하나 이상의 스레드로 구성 가능

JVM과 자바 응용프로그램, 스레드의 관계

두 개의 자바 응용프로그램이 동시에 실행시키고자 하면
두 개의 JVM을 이용하고 응용프로그램은 서로 소켓 등을 이용하여 통신

자바 스레드와 JVM



각 스레드의 스레드 코드는
응용프로그램 내에 존재함

JVM이 스레드를 관리함
- 스레드가 몇 개인지?
- 스레드 코드의 위치가 어디인지?
- 스레드의 우선순위는 얼마인지?
- 등

현재 하나의 JVM에 의해 4 개의 스레드가 실행 중이며
그 중 스레드 2가 JVM에 의해 스케쥴링되어 실행되고 있음

스레드 만들기

- 스레드 실행을 위해 개발자가 하는 작업
  - 스레드 코드 작성
  - 스레드를 생성하고 스레드 코드를 실행하도록 JVM에게 요청
- 스레드 만드는 2 가지 방법
  - **java.lang.Thread 클래스를 상속하는 경우**
  - **java.lang.Runnable 인터페이스를 구현하는 경우**

Thread 클래스의 메소드

- 생성자
  - Thread()
  - Thread(Runnable target)
  - Thread(String name)
  - Thread(Runnable target, String name)
- 스레드 시작시키기
  - void start()
- 스레드 코드
  - void run()
- 스레드 잠자기
  - static void sleep(long mills)
- 다른 스레드 죽이기
  - void interrupt()

Thread 클래스의 메소드

- 다른 스레드에게 양보
  - static void yield()
  - 현재 스레드 실행을 중단하고 다른 스레드가 실행될수 있게 양보
- 다른 스레드가 죽을 때까지 기다리기
  - void join()
- 현재 스레드 객체 알아내기
  - static Thread currentThread()
- 스레드 ID 알아내기
  - long getId()
- 스레드 이름 알아내기
  - String getName()
- 스레드 우선순위값 알아내기
  - int getPriority()
- 스레드의 상태 알아내기
  - Thread.State getState()

Thread 클래스를 이용한 스레드 생성

- 스레드 클래스 작성
  - Thread 클래스 상속 새 클래스 작성
- 스레드 코드 작성
  - run() 메소드 오버라이딩
    - run() 메소드를 스레드 코드라고 부름
    - run() 메소드에서 스레드 실행 시작
- 스레드 객체 생성
- 스레드 시작
  - start() 메소드 호출
    - 스레드로 작동 시작
    - JVM에 의해 스케줄되기 시작함

```
class TimerThread extends Thread {
    .....
    public void run() { // run() 오버라이딩
        .....
    }
}
```

```
TimerThread th = new TimerThread();
```

```
th.start();
```

Thread를 상속받아 1초 단위로 출력하는 TimerThread 클래스

스레드 클래스 정의

스레드 코드 작성

1초에 한 번씩 n을 증가시켜 콘솔에 출력한다.

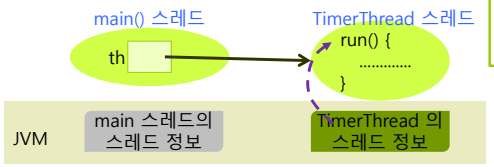
스레드 객체 생성

스레드 시작

```
class TimerThread extends Thread {
    int n = 0;
    public void run() {
        while(true) { // 무한루프를 실행한다.
            System.out.println(n);
            n++;
            try {
                sleep(1000); //1초 동안 잠을 잔 후 깨어난다.
            }
            catch(InterruptedException e){return;}
        }
    }
}
```

```
public class TestThread {
    public static void main(String [] args) {
        TimerThread th = new TimerThread();
        th.start();
    }
}
```

```
0
1
2
3
4
```



Thread를 상속받아 1초 단위로 출력하는 타이머 레이블 만들기

```
import java.awt.*;
import javax.swing.*;
class TimerThread extends Thread {
    JLabel timerLabel;
    public TimerThread(JLabel timerLabel) {
        this.timerLabel = timerLabel;
    }
    public void run() {
        int n=0;
        while(true) {
            timerLabel.setText(Integer.toString(n));
            n++;
            try {
                Thread.sleep(1000);
            }
            catch(InterruptedException e) {
                return;
            }
        }
    }
}
```

```
public class ThreadTimerEx extends JFrame {
    public ThreadTimerEx() {
        setTitle("ThreadTimerEx 예제");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        JLabel timerLabel = new JLabel();
        timerLabel.setFont(new Font("Gothic", Font.ITALIC, 80));
        TimerThread th = new TimerThread(timerLabel);
        c.add(timerLabel);
        setSize(300,150);
        setVisible(true);
        th.start();
    }
    public static void main(String[] args) {
        new ThreadTimerEx();
    }
}
```

스레드 주의 사항

- run() 메소드가 종료하면 스레드는 종료한다.
  - 스레드가 계속 살아있도록 하려면 run() 메소드 내 무한 루프 구성
- 한번 종료한 스레드는 다시 시작시킬 수 없다.
  - 다시 스레드 객체를 생성하고 스레드로 등록하여야 한다.
- 한 스레드에서 다른 스레드를 강제 종료할 수 있다.

Runnable 인터페이스로 스레드 만들기

- 스레드 클래스 작성
  - Runnable 인터페이스 구현하는 새 클래스 작성
- 스레드 코드 작성
  - run() 메소드 구현
    - run() 메소드를 스레드 코드라고 부름
    - run() 메소드에서 스레드 실행 시작
- 스레드 객체 생성
- 스레드 시작
  - start() 메소드 호출

```
class TimerRunnable implements Runnable {
    .....
    public void run() { // run() 메소드 구현
        .....
    }
}
```

```
Thread th = new Thread(new TimerRunnable());
```

```
th.start();
```

Runnable인터페이스를 구현하여 1초 단위 출력 TimerRunnable 클래스

Runnable 클래스로 구현

스레드 코드 작성

1초에 한 번씩 n을 증가시켜 콘솔에 출력한다.

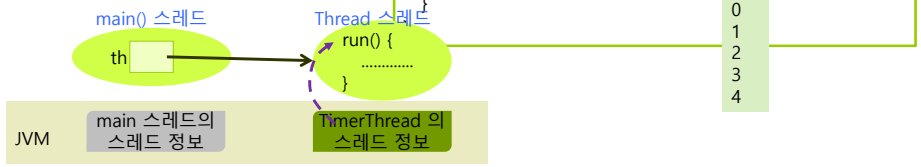
스레드 객체 생성

스레드 시작

```
class TimerRunnable implements Runnable {
    int n = 0;
    public void run() {
        while(true) { // 무한루프를 실행한다.
            System.out.println(n);
            n++;
            try {
                Thread.sleep(1000); // 1초 동안 잠을 잔 후 깨어난다.
            }
            catch(InterruptedException e) { return; }
        }
    }
}

public class TestRunnable {
    public static void main(String [] args) {
        Thread th = new Thread(new TimerRunnable());
        th.start();
    }
}
```

```
0
1
2
3
4
```



Runnable인터페이스를 구현하여 1초 단위 출력 레이블 만들기

```
import java.awt.*;
import javax.swing.*;
class TimerRunnable implements Runnable {
    JLabel timerLabel;
    public TimerRunnable(JLabel timerLabel) {
        this.timerLabel = timerLabel;
    }
    public void run() {
        int n=0;
        while(true) {
            timerLabel.setText(Integer.toString(n));
            n++;
            try {
                Thread.sleep(1000);
            }
            catch(InterruptedException e) {
                return;
            }
        }
    }
}
```

```
public class RunnableTimerEx extends JFrame {
    public RunnableTimerEx() {
        setTitle("RunnableTimerEx 예제");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        JLabel timerLabel = new JLabel();
        timerLabel.setFont(new Font("Gothic", Font.ITALIC, 80));
        TimerRunnable runnable = new
TimerRunnable(timerLabel);
        Thread th = new Thread(runnable);
        c.add(timerLabel);
        setSize(300,150);
        setVisible(true);
        th.start();
    }
    public static void main(String[] args) {
        new RunnableTimerEx();
    }
}
```

스레드 정보

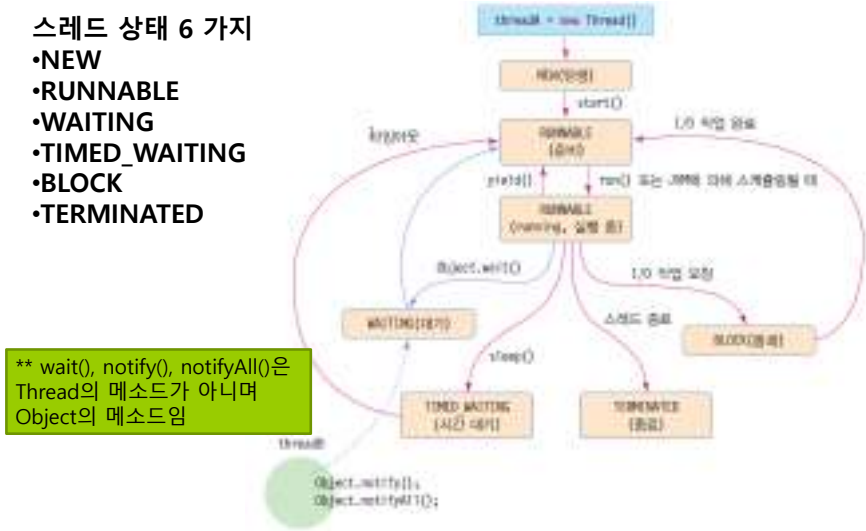
필드	타입	내용
스레드 이름	스트링	스레드의 이름으로서 사용자가 지정
스레드 ID	정수	스레드 고유의 식별자 번호
스레드의 PC(Program Count)	정수	현재 실행 중인 스레드 코드의 주소
스레드 상태	정수	NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCK, TERMINATED 등 6개 상태 중 하나
스레드 우선순위	정수	스레드 스케줄링 시 사용되는 우선순위 값으로서 1~10 사이의 값이며 10이 최상위 우선순위
스레드 그룹	정수	여러 개의 자바 스레드가 하나의 그룹을 형성할 수 있으며 이 경우 스레드가 속한 그룹
스레드 레지스터 스택	메모리 블록	스레드가 실행되는 동안 레지스터들의 값

스레드 상태

- 스레드 상태 6 가지
  - NEW
    - 스레드가 생성되었지만 스레드가 아직 실행할 준비가 되지 않았음
  - RUNNABLE
    - 스레드가 JVM에 의해 실행되고 있거나 실행 준비되어 스케쥴링을 기다리는 상태
  - WAITING
    - 다른 스레드가 notify(), notifyAll()을 불러주기를 기다리고 있는 상태
    - 스레드 동기화를 위해 사용
  - TIMED_WAITING
    - 스레드가 sleep(n)을 호출하여 n 밀리초 동안 잠을 자고 있는 상태
  - BLOCK
    - 스레드가 I/O 작업을 요청하면 JVM이 자동으로 이 스레드를 BLOCK 상태로 만듦
  - TERMINATED
    - 스레드가 종료한 상태
- 스레드 상태는 JVM에 의해 기록 관리됨

스레드 상태와 생명 주기

스레드 상태 6 가지
•NEW
•RUNNABLE
•WAITING
•TIMED_WAITING
•BLOCK
•TERMINATED



** wait(), notify(), notifyAll()은 Thread의 메소드가 아니며 Object의 메소드임

스레드 우선순위와 스케쥴링

- 스레드의 우선순위(Priority)
  - 최대값 = 10(MAX_PRIORITY)
  - 최소값 = 1(MIN_PRIORITY)
  - 보통값 = 5(NORMAL_PRIORITY)
- 스레드 우선순위는 응용프로그램에서 변경 가능
  - void setPriority(int priority)
  - int getPriority()
- main() 스레드의 우선순위 값은 초기에 5
- 스레드는 부모 스레드와 동일한 우선순위 값을 가지고 탄생
- JVM의 스케쥴링 정책
  - 철저한 우선순위 기반
    - 가장 높은 우선순위의 스레드가 우선적으로 스케쥴링
    - 동일한 우선순위의 스레드는 돌아가면서 스케쥴링(라운드 로빈)

main()은 자바의 main 스레드

- main 스레드와 main() 메소드
  - JVM은 응용프로그램이 실행될 때 스레드 생성 : main 스레드
  - main 스레드에 의해 main() 메소드 실행
    - main() 메소드가 종료하면 main 스레드 종료

```
public class ThreadMainEx {
   public static void main(String [] args) {
      long id = Thread.currentThread().getId();
      String name = Thread.currentThread().getName();
      int priority = Thread.currentThread().getPriority();
      Thread.State s = Thread.currentThread().getState();
      System.out.println("현재 스레드 이름 = " + name);
      System.out.println("현재 스레드 ID = " + id);
      System.out.println("현재 스레드 우선순위 값 = " + priority);
      System.out.println("현재 스레드 상태 = " + s);
   }
}
```

```
현재 스레드 이름 = main
현재 스레드 ID = 1
현재 스레드 우선순위 값 = 5
현재 스레드 상태 = RUNNABLE
```

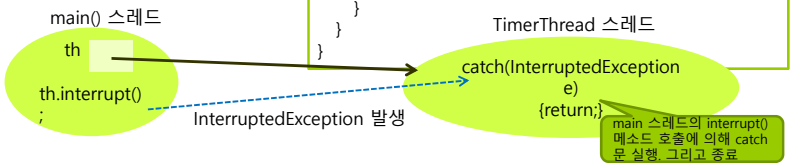
스레드 종료와 타 스레드 강제 종료

- 스스로 종료
  - run() 메소드 리턴
- 타 스레드에서 강제 종료 : interrupt() 메소드 사용

```
public static void main(String [] args) {
   TimerThread th = new  TimerThread();
   th.start();
   th.interrupt(); // 강제 종료
}
```

```
class TimerThread extends Thread {
   int n = 0;
   public void run() {
      while(true) {
         System.out.println(n); // 화면에 카운트 값 출력
         n++;
          try {
            sleep(1000);
         }
         catch(Interrupt
            return; // 예외를 받고 스스로 리턴하여 종료
         }
      }
   }
}
```

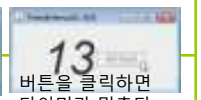
만일 return 하지 않으면 스레드는 종료하지 않음



1초씩 작동하는 타이머 스레드 강제 종료

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class TimerRunnable implements Runnable
{
   JLabel timerLabel;
   public TimerRunnable(JLabel timerLabel) {
      this.timerLabel = timerLabel;
   }
   public void run() {
      int n=0;
      while(true) {
         timerLabel.setText(Integer.toString(n));
         n++;
         try {
            Thread.sleep(1000); // 1초 sleep
         }
         catch(InterruptedException e) {
            return; // 예외발생시 스레드 종료
         }
      }
   }
}
```

정상 작동



버튼을 클릭하면 타이머가 멈춘다. 버튼은 비활성화된다.

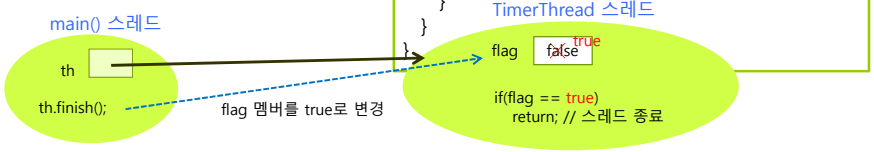
```
public class ThreadInterruptEx extends JFrame {
   Thread th;
   public ThreadInterruptEx() {
      setTitle(" hreadInterruptEx 예제");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      JLabel timerLabel = new JLabel();
      timerLabel.setFont(new Font("Gothic", Font.ITALIC, 80));
      TimerRunnable runnable = new TimerRunnable(timerLabel);
      th = new Thread(runnable); // 스레드 생성
      c.add(timerLabel);
      // 버튼을 생성하고 Action 리스너 등록
      JButton btn =new JButton("kill Timer");
      btn.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            th.interrupt(); // 타이머 스레드 강제 종료
            JButton btn = (JButton)e.getSource();
            btn.setEnabled(false); // 버튼 비활성화
         }
      });
      c.add(btn);
      setSize(300,150);
      setVisible(true);
      th.start(); // 스레드 동작시킴
   }
   public static void main(String[] args) {
      new ThreadInterruptEx();
   }
}
```

flag를 이용한 종료

- 스레드 A가 스레드 B의 flag를 true로 만들면, 스레드 B가 스스로 종료하는 방식

```
public static void main(String [] args) {
   TimerThread th = new  TimerThread();
   th.start();
   th.finish(); // TimerThread 강제 종료
}
```

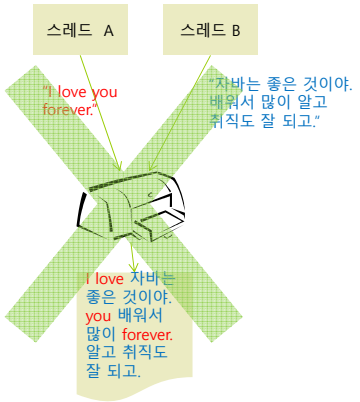
```
class TimerThread extends Thread {
   int n = 0;
   bool flag = false; // false로 초기화
   public void finish() { flag = true; }
   public void run() {
      while(true) {
         System.out.println(n); // 화면출력
         n++;
         try {
            sleep(1000);
            if(flag == true)
               return; // 스레드 종료
         }
         catch(InterruptedException e){
            return;
         }
      }
   }
}
```



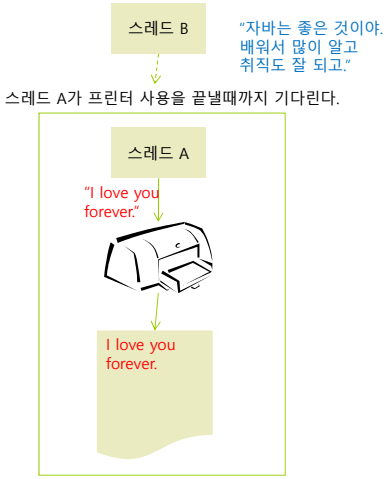
스레드 동기화(Thread Synchronization)

- 멀티스레드 프로그램 작성시 주의점
  - 다수의 스레드가 공유 데이터에 동시에 접근하는 경우
    - 공유 데이터의 값에 예상치 못한 결과 발생 가능
- 스레드 동기화
  - 멀티스레드의 공유 데이터의 동시 접근 문제 해결책
    - 공유 데이터를 접근하는 모든 스레드의 한 줄 세우기
    - 한 스레드가 공유 데이터에 대한 작업을 끝낼 때까지 다른 스레드가 대기 하도록 함

두 스레드의 프린터 동시 쓰기로 충돌하는 경우

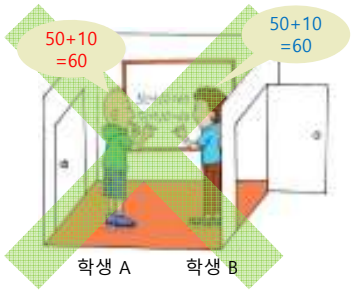


두 스레드가 동시에 프린터에 쓰는 경우
문제 발생



한 스레드의 출력이 끝날 때까지
대기함으로써 **정상 출력**

공유 집계판에 동시 접근하는 경우



두 학생이 동시에 방에 들어와서
집계판을 수정하는 경우
집계판의 **결과가 잘못됨**

방에 먼저 들어간 학생이
집계를 끝내기를 기다리면
정상 처리

synchronized 블록 지정

- **synchronized** 키워드
  - 한 스레드가 독점적으로 실행해야 하는 부분(동기화 코드)을 표시하는 키워드 - 임계 영역(critical section) 표기 키워드
- synchronized 사용
  - 메소드 전체 혹은 코드 블록
- synchronized 블록이 실행될 때,
  - 먼저 실행한 스레드가 모니터 소유
    - 모니터란 해당 객체를 독점적으로 사용할 수 있는 권한
  - 모니터를 소유한 스레드가 모니터를 내놓을 때까지 다른 스레드 대기

```
synchronized void add() {
   int n = getCurrentSum();
   n+=10;
   setCurrentSum(n);
}
```

synchronized 메소드

```
void execute() {
   // 다른 코드들
   synchronized(this) {
      int n = getCurrentSum();
      n+=10;
      setCurrentSum(n);
   }
   // 다른 코드들
}
```

synchronized 사용 예 : 집계판

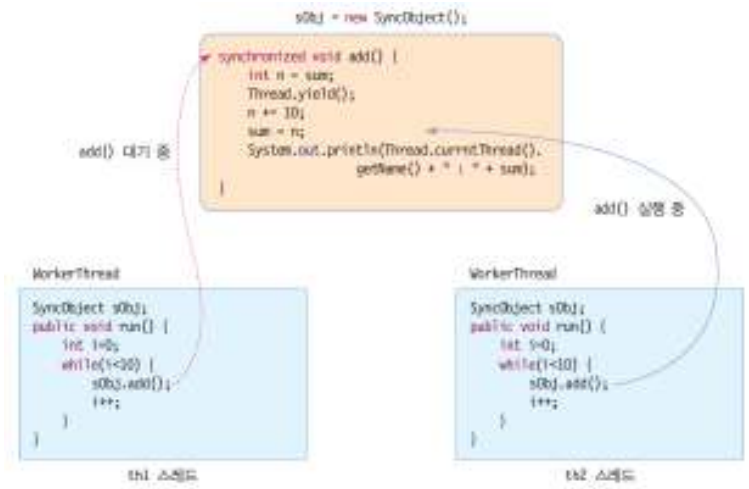
```
public class SynchronizedEx {
   public static void main(String [] args) {
      SyncObject obj = new SyncObject();
      Thread th1 = new WorkerThread("kitae", obj);
      Thread th2 = new WorkerThread("hyosoo", obj);
      th1.start();
      th2.start();
   }
}
class SyncObject {
   int sum = 0;
    synchronized void add() {
      int n = sum;
      Thread.yield(); // 현재 실행하고 있는 스레드가 다른 스레드에게 양보
      n += 10;
      sum = n;
      System.out.println(Thread.currentThread().getName() + " : " + sum);
   }
   int getSum() {return sum;}
}
class WorkerThread extends Thread {
   SyncObject sObj;
   WorkerThread(String name, SyncObject sObj) {
      super(name);
      this.sObj = sObj;
   }
   public void run() {
      int i=0;
      while(i<10) {
         sObj.add();
         i++;
      }
   }
}
```

- 집계판 : class SyncObject
- 각 학생 : class WorkerThread

```
kitae : 10
hyosoo : 20
kitae : 30
hyosoo : 40
kitae : 50
hyosoo : 60
kitae : 70
hyosoo : 80
hyosoo : 90
hyosoo : 100
hyosoo : 110
hyosoo : 120
hyosoo : 130
hyosoo : 140
kitae : 150
kitae : 160
kitae : 170
kitae : 180
kitae : 190
kitae : 200
```

kitae와 hyosoo가 각각 10번씩 add()를 호출하였으며 동기화가 잘 이루어져서 최종 누적 점수 sum이 200이 됨

SyncObject에 대한 두 스레드의 동시 접근 과정



집계판에서 synchronized 사용하지 않을 경우

```
public class SynchronizedEx {
    public static void main(String [] args) {
        SyncObject obj = new SyncObject();
        Thread th1 = new WorkerThread("kitae", obj);
        Thread th2 = new WorkerThread("hyosoo", obj);
        th1.start();
        th2.start();
    }
}
class SyncObject {
    int sum = 0;
    void add() { // synchronized가 없을 경우
        int n = sum;
        Thread.yield();
        n += 10;
        sum = n;
        System.out.println(Thread.currntThread().getName() + " : " + sum);
    }
    int getSum() {return sum;}
}
class WorkerThread extends Thread {
    SyncObject sObj;
    WorkerThread(String name, SyncObject sObj) {
        super(name);
        this.sObj = sObj;
    }
    public void run() {
        int i=0;
        while(i<10) {
            sObj.add();
            i++;
        }
    }
}
```

```
kitae : 10
hyosoo : 20
kitae : 30
hyosoo : 30
kitae : 40
hyosoo : 50
kitae : 50
kitae : 60
hyosoo : 60
kitae : 70
hyosoo : 70
kitae : 80
hyosoo : 90
kitae : 100
hyosoo : 100
kitae : 110
hyosoo : 120
kitae : 130
hyosoo : 140
hyosoo : 150
```

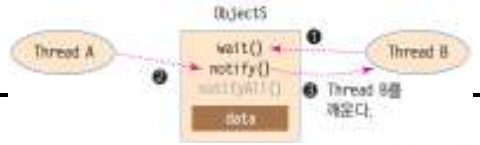
add() 충돌 (kitae : 30 / hyosoo : 50 / kitae : 60 / hyosoo : 70 / kitae : 100)

kitae와 hyosoo가 각각 10번씩 add()를 호출하였지만 **동기화가 이루어지지 않아** 공유 변수 sum에 대한 접근에 충돌이 있었고, 수를 많이 잃어버리게 되어 누적 점수가 150 밖에 되지 못함

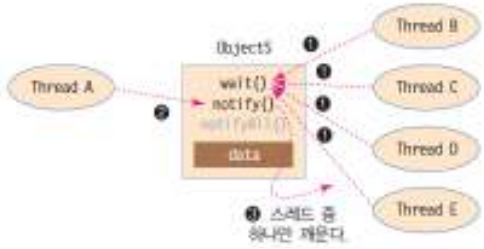
wait(), notify(), notifyAll()

- 동기화 객체
  - 두 개 이상의 스레드 사이에 동기화 작업에 사용되는 객체
- 동기화 메소드
  - synchronized 블록 내에서만 사용되어야 함
  - wait()
    - 다른 스레드가 notify()를 불러줄 때까지 기다린다.
  - notify()
    - wait()를 호출하여 대기중인 스레드를 깨우고 RUNNABLE 상태로 만든다.
    - 2개 이상의 스레드가 대기중이라도 오직 한 스레드만 깨운다.
  - notifyAll()
    - wait()를 호출하여 대기중인 모든 스레드를 깨우고 모두 RUNNABLE 상태로 만든다.
- wait(), notify(), notifyAll()은 Object의 메소드
  - 모든 객체가 동기화 객체가 될 수 있다.
  - Thread 객체도 동기화 객체로 사용될 수 있다.

Thread A가 ObjectS.wait()를 호출하여 무한 대기하고, Thread B가 ObjectS.notify()를 호출하여 ObjectS에 대기하고 있는 Thread A를 깨운다.



4 개의 스레드가 모두 ObjectS.wait()를 호출하여 대기하고, ThreadA는 ObjectS.notify()를 호출하여 대기중인 스레드 중 하나만 깨우는 경우



4 개의 스레드가 모두 ObjectS.wait()를 호출하여 대기하고, ThreadA는 ObjectS.notifyAll()를 호출하여 대기중인 4개의 스레드를 모두 깨우는 경우