2019학년도 2학기

JAVA 프로그래밍 II

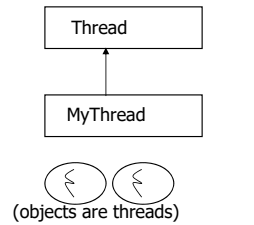
514770
2019년 가을학기
11/20/2019
박경신

Lab #7 (Multi-Thread)

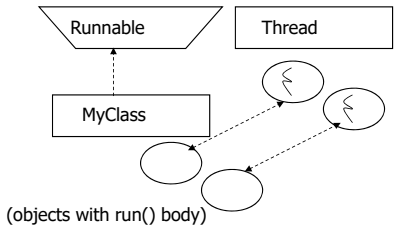
- 기존 요구사항 분석
  - **Lab #7는 Thread, Runnable를 이용한 다양한 멀티스레드 기능을 사용**
  - **Mutex, Semaphore, Monitor 개념 이해**
  - **Object 클래스 wait(), notify(), notifyAll()**
- Thread 클래스, Runnable 인터페이스, Mutex, Semaphore, Monitor, Worker thread (a.k.a. background thread)

Lab #7_1 extends Thread vs implements Runnable

1. Create a class that extends the Thread class
2. Create a class that implements the Runnable interface



[MyThread extends Thread]



[MyClass implements Runnable]

Lab #7_1 extends Thread vs implements Runnable

- **Lab#7_1에서는 Thread 클래스를 상속 받는 Multithread 클래스를 구현한다.**
  - ***public class Multithread extends Thread {***

```
    private static int count = 0; // global counter
// 중간생략..
    @Override
    public void run() {
        while(true) {
            System.out.println("global counter=" + count);
            count++;
        }
    }
}
```

Lab #7_1 extends Thread vs implements Runnable

- Lab#7_1에서는 Runnable 인터페이스를 상속 받는 MultithreadRunnable 클래스를 구현한다.

```
public class MultithreadRunnable implements Runnable {
    private static int count = 0; // global counter
// 중간생략..
    @Override
    public void run() {
        while(true) {
            System.out.println("global counter=" + count);
            count++;
        }
    }
}
```

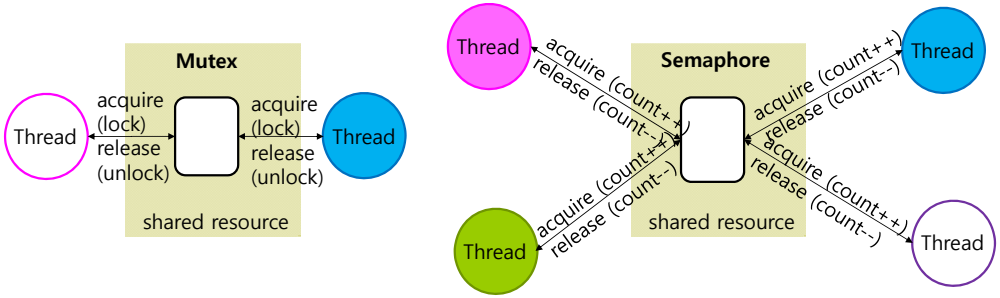
Lab #7_1 extends Thread vs implements Runnable

- Lab#7_1에서는 Thread 클래스 또는 Runnable 인터페이스를 상속 받는 간단한 멀티스레드 프로그램을 구현한다.

```
public class MultithreadTest {
    public static void main(String[] args) {
        new Multithread("Thread1 :").start();
        new Multithread("Thread2 :").start();
        new Multithread("Thread3 :").start();
        new Thread(new MultithreadRunnable(), "Thread1 :").start();
        new Thread(new MultithreadRunnable(), "Thread2 :").start();
        new Thread(new MultithreadRunnable(), "Thread3 :").start();
    }
}
```

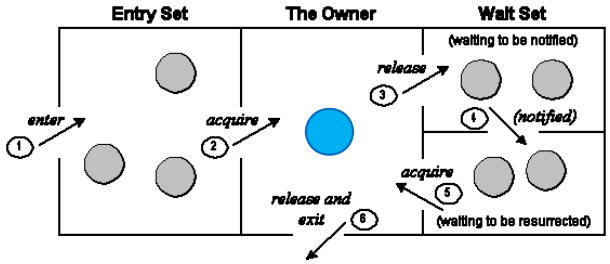
Mutex vs Semaphore vs Monitor

- **Mutex** (mutual execlusion) semaphore controls **only one thread at a time** executing on the shared resource.
- **Semaphore** controls **the number of threads** executing on the shared resources.



Mutex vs Semaphore vs Monitor

- **Monitor** controls **only one thread at a time**, and can execute in the **monitor (shared object)**



Lab #7_2 Monitor

- Lab#7_2에서는 Monitor를 사용하여 멀티 스레드 프로그램을 구현한다. Lab7_2 실행 결과를 Lab7_1과 비교한다.

```
public class SynchronizedMultithread extends Thread {
    SharedCounter counter; // global counter (using monitor)
// 중간생략..
    @Override
    public void run() {
        int i = 0;
        while(true) {
            counter.nextCount(i);
            i++;
        }
    }
}
```

Lab #7_2 Monitor

- 모든 클래스는 Monitor로 구현되어 있다. synchronized 키워드를 사용하여 공유 자원 (shared object)을 동기화 (synchronization) 한다.

```
public class SharedCounter {
    private int count = 0;
    public SharedCounter() { }
    public synchronized void nextCount(int i) {
        System.out.println(Thread.currentThread().getName() + " cycle=" + i + "
global count=" + count);
        count++;
    }
}
```

Lab #7_2 Monitor

- 공유객체 counter는 Monitor로 동기화 (synchronization) 된다.

```
public class MonitorTest {
    public static void main(String[] args) {
        SharedCounter counter = new SharedCounter();
        new SynchronizedMultithread("SynThread1 :", counter).start();
        new SynchronizedMultithread("SynThread2 :", counter).start();
        new SynchronizedMultithread("SynThread3 :", counter).start();
    }
}
```

Lab #7_2 Mutex (Mutual Exclusion Semaphore)

- Lab#7_2에서는 Mutex를 사용하여 멀티 스레드가 공유 자원(shared resource)을 하나씩 access하도록 한다. Lab7_2 실행 결과를 Lab7_1과 비교한다.

```
private static Semaphore mutex = new Semaphore(1); // binary semaphore
public void run() {
    while (true) {
        try {
            mutex.acquire(); // mutex lock
            System.out.println(getName() + " global count=" + count);
            count++;
            mutex.release(); // mutex unlock
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
            return;
        }
    }
}
```

Lab #7_2 Mutex (Mutual Exclusion Semaphore)

- Lab#7_2에서는 Mutex를 사용하여 멀티 스레드가 공유 자원(shared resource)을 하나씩 access하도록 한다. Lab7_2 실행 결과를 Lab7_1과 비교한다.

```
public class MultithreadMutexTest {
    public static void main(String[] args) {
        new MultithreadMutex("Thread1 :").start();
        new MultithreadMutex("Thread2 :").start();
        new MultithreadMutex("Thread3 :").start();
        new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
        new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
        new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
    }
}
```

Lab #7_2 Semaphore (Counting Semaphore)

- Lab#7_2에서는 Semaphore를 사용하여 멀티 스레드가 공유 자원(shared resource)을 동시 access하도록 한다. Lab7_2 실행 결과를 Lab7_1과 비교한다.

```
private static Semaphore sem = new Semaphore(3); // counting semaphore
public void run() {
        while (true) {
                try {
                        sem.acquire(); // semaphore lock
                        System.out.println(getName() + " global count=" + count);
                        count++;
                        sem.release(); // semaphore unlock
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                        return;
                }
        }
`
```

Lab #7_2 Semaphore (Counting Semaphore)

- Lab#7_2에서는 Semaphore를 사용하여 멀티 스레드가 공유 자원(shared resource)을 동시 access하도록 한다. Lab7_2 실행 결과를 Lab7_1과 비교한다.

```
public class MultithreadSemaphoreTest {
    public static void main(String[] args) {
        new MultithreadSemaphore("Thread1 :").start();
        new MultithreadSemaphore("Thread2 :").start();
        new MultithreadSemaphore("Thread3 :").start();
        new Thread(new MultithreadRunnableSemaphore("Thread1 :")).start();
        new Thread(new MultithreadRunnableSemaphore("Thread1 :")).start();
        new Thread(new MultithreadRunnableSemaphore("Thread1 :")).start();
    }
}
```

Lab #7_3 Monitor (TicketSeller)

- Lab#7_3에서는 Monitor를 사용하여 seller(즉, 멀티스레드)에서 ticket(즉, 공유 자원)을 동시 판매 가능한 TicketSeller를 구현한다.

```
private SharedCounter numTicket; // global counter (using monitor)
public void run() {
        while (!done) {
                if (numTicket.getCount() == 0) done= true;
                else {
                        sold++;
                        numTicket.decreaseCount(sold);
                }
        }
}
```

Lab #7_3 Mutex (TicketSeller)

- Lab#7_3에서는 Mutex를 사용하여 seller(즉, 멀티스레드)에서 공유 자원(즉, ticket)을 동시 판매 가능한 ticketSeller를 구현한다.

```
private static Semaphore mutex = new Semaphore(1); // semaphore with max access count = 1
private static int numTicket = 50;
public void run() {
        while (!done) {
                mutex.acquire(); // mutex aquire
                if (numTicket == 0) done= true;
                else {
                        sold++;
                        numTicket--;
                        System.out.println(getName() + " sold=" + sold + " ticket=" + numTicket);
                }
                mutex.release(); // mutex release
        }
}
```

Lab #7_3 Semaphore (TicketSeller)

- Lab#7_3에서는 Semaphore를 사용하여 seller(즉, 멀티스레드)에서 공유 자원(즉, ticket)을 동시 판매 가능한 ticketSeller를 구현한다.

```
private Semaphore semaphore; // semaphore with max access count = numTicket
private static int numTicket = 50;
public void run() {
        while (!done) {
                if (numTicket == 0) done= true;
                else {
                        semaphore.acquire(); // semaphore aquire
                        sold++;
                        numTicket--;
                        System.out.println(getName() + " sold=" + sold + " ticket=" + numTicket);
                }
        }
        semaphore.release(); // semaphore release
}
```

Lab #7_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)
  - public class SharedBuffer {

```
    private volatile String[] buffer;
    private volatile int tail, head, count = 0;
    public synchronized void put(String data) { // put data into the buffer
            while (count >= buffer.length) { wait(); } // wait (buffer is full)
            System.out.println("produce=" + data);
            buffer[tail] = data;
            tail = (tail + 1) % buffer.length;
            count++;
            notifyAll(); // signal
    }
```

Lab #7_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
    public synchronized String take() { // take data from the buffer
            while (count <= 0) { wait(); } // wait (buffer is empty)
            String data = buffer[head];
            head = (head + 1) % buffer.length;
            count--;
            System.out.println("produce=" + data);
            notifyAll(); // signal
            return data;
    }
}
```

Lab #7_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
public abstract class Worker implements Runnable {
    protected final SharedBuffer data; // sharedbuffer
    protected String currentItem;
    protected boolean running, shutdown;
    public Worker(SharedBuffer data) {    this.data = data;        }
    public synchronized void setCurrentItem(String item) { this.currentItem = item; }
    public synchronized String getCurrentItem() { return this.currentItem; }
    public synchronized boolean isRunning() {
            if (this.shutdown) this.running = false;
            return this.running;
    }
    public synchronized void stop() { this.shutdown = true; }
}
```

Lab #7_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
public class Producer extends Worker {
    private static volatile int count = 0; // global counter among producer threads
    public Producer(String name, SharedBuffer data) {
            super(data);
    }
    public void run() {
            while (isRunning()) {
                    String str = "gcount=" + count++);
                    data.put(str); // put data into the buffer
                    setCurrentItem(str);
            }
    }
}
```

Lab #7_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
public class Consumer extends Worker {
    public Consumer(String name, SharedBuffer data) {
            super(data);
    }
    public void run() {
            while (isRunning()) {
                    String str = data.take(); // take data from the buffer
                    setCurrentItem(str);
                    System.out.println(Thread.currentTread().getName() + str);
            }
    }
}
```

Lab 7_4 Monitor (Producer-Consumer Problem)

- 공유객체 counter는 Monitor로 동기화 (synchronization) 된다.

```
public class MonitorTest {
    public static void main(String[] args) {
            SharedBuffer data = new SharedBuffer(5); // buffer size=5
            new Thread(new Producer("Producer1 :", data)).start();
            new Thread(new Producer("Producer2 :", data)).start();
            new Thread(new Producer("Producer3 :", data)).start();
            new Thread(new Consumer("Consumer1 :", data)).start();
            new Thread(new Consumer("Consumer2 :", data)).start();
            new Thread(new Consumer("Consumer3 :", data)).start();
            new Thread(new Consumer("Consumer4 :", data)).start();
    }
}
```

Lab #7_4 Semaphore (Producer-Consumer Problem)

- Producer() {
  while (1) { <<< Produce item >>>
    P(empty); // for **wait**, Get an empty buffer count, block if unavailable
    P(mutex); // acquire critical section: shared buffer
    <<< critical section: Put item into shared buffer >>>
    V(mutex); // release critical section
    V(full); // for **signal**, increase number of full buffer count
  }
- }
- Consumer() {
  while (1) {
    P(full); // for **wait**, Get a full buffer count, block if unavailable
    P(mutex);
    <<< critical section: Take item from shared buffer >>>
    V(mutex);
    V(empty); // for **signal**, increase number of empty buffer count
  }
}

Lab #7_4 Semaphore (Producer-Consumer Problem)

- **Producer-Consumer Problem (Semaphore)**
  - *public class BoundedBuffer {*

```
private String[] buffer;
private Semaphore mutex;
private Semaphore empty;
private Semaphore full;
private int tail = 0, head = 0;

public BoundedBuffer(int size) {
    buffer = new String[size];
    mutex = new Semaphore(1);
    empty = new Semaphore(size);
    full = new Semaphore(0);
}
```

Lab #7_4 Semaphore (Producer-Consumer Problem)

- **Producer-Consumer Problem (Semaphore)**

```
public void put(String data) { // put data into the buffer
    if (full.availablePermits() >= buffer.length) System.out.println("buffer is full");
    empty.acquire(); // decrease empty count
    System.out.println("produce=" + data);
    mutex.acquire(); // mutex lock for shared data
    buffer[tail] = data;
    tail = (tail + 1) % buffer.length;
    mutex.release(); // mutex unlock
    full.release(); // increase full count
}
```

Lab #7_4 Semaphore (Producer-Consumer Problem)

- **Producer-Consumer Problem (Semaphore)**

```
public String take() { // take data from the buffer
    if (full.availablePermits() <= 0) System.out.println("buffer is empty");
    full.acquire(); // decrease full count
    mutex.acquire(); // mutex lock for shared data
    String data = buffer[head];
    head = (head + 1) % buffer.length;
    mutex.release(); // mutex unlock
    empty.release(); // increase empty count
    return data;
  }
}
```

Lab #7_5 WorkerThread

- **Lab#7_5_ThreadedImageGrayscale에서는 # of thread에 따른 20 images를 grayscale 변환하는데 걸리는 시간(elapsedTime) performance를 출력한다.**

# of thread	elapsedTime (ms)
1	2144 ms
2	1262 ms
4	1060 ms
5	1010 ms
10	816 ms
20	746 ms

과제 Lab7_WOEID

- woeid 패키지 안에 WOEID를 작성한다.
  - private long id;
  - private String city;
  - private String country;
  - private double latitude;
  - private double longitude;
- woeid 패키지 안에 WoeidImporter, WoeidManager을 작성한다.
- woeid 패키지 안에 WoeidTest을 작성하여, 모든 데이터 파일을 로딩해서 각종 테스트를 한다.
  - WOEIDLIST1.csv
  - WOEIDLIST2.csv
  - WOEIDLIST3.csv
  - WOEIDLIST4.csv

과제 Lab7_WOEID

- woeid.threaded 패키지 안에 ThreadSafeWoeidImporter, ThreadSafeWoeidManager을 작성한다.
  - List<Woeid> list = Collections.*synchronizedList*(new ArrayList<Woeid>()); 사용
  - ConcurrentHashMap<String, Woeid> hashmap 사용
- woeid.threaded 패키지 안에 ThreadedWoeidTest을 작성하여, 멀티스레드로 각각 데이터를 로딩하여 프린트 테스트 한다.
  - ThreadedWoeidTest는 Runnable을 상속받는다
  - Thread0 - WOEIDLIST1.csv 로딩하여 manager에 추가
  - Thread1 - WOEIDLIST2.csv 로딩하여 manager에 추가
  - Thread2 – WOEIDLIST3.csv 로딩하여 manager에 추가
  - Thread3 - WOEIDLIST4.csv 로딩하여 manager에 추가
  - 모든 데이터가 다 추가된 후 manager로 출력(print)

과제 제출

- 모든 Lab 코드와 보고서를 전체적으로 묶어서 e-learning에 과제 제출
- 각 Lab마다 **본인이 추가로 작성한 코드**와 설명을 중점적으로 보고할 것!