# *Lab Exercise #10*

# 스레드 프로그래밍 실습

2018년도 2학기

## 컴퓨터프로그래밍2

김 영 국

충남대학교 컴퓨터공학과

# 목차

- 실습
  - 스레드의 생성, 동기화, 조정
  - 계좌이체 시뮬레이션
  - 동기화된 계좌이체 시뮬레이션
- 과제
  - 괴물 피하기 게임

# 실습10-1. 스레드의 생성, 동기화, 조정(1)

- 멀티스레드를 사용하는 것과 사용하지 않는 것에 대하여 프로그램의 속도가 얼마나 차이가 나는지 확인해보자.
  - Integer.MIN_VALUE 와 Integer.MAX_VALUE 사이의 정수들에 대하여, 홀수의 총합, 짝수의 총합을 구하는 프로그램을 멀티 스레드를 사용한 것과 그렇지 않은 것 두개를 만들어서 실행 시간을 측정한다.

1. 멀티 스레드 미사용

```
long startTime = System.currentTimeMillis(); // 시작시간

long evenSum = 0; // 짝수합
for(long l = Integer.MIN_VALUE; l <= Integer.MAX_VALUE; l++)
    if(l % 2 == 0)
        evenSum += l;
System.out.println("evenSum: " + evenSum);

long oddSum = 0; // 홀수합
for(long l = Integer.MIN_VALUE; l <= Integer.MAX_VALUE; l++)
    if(l % 2 != 0)
        oddSum += l;
System.out.println("oddSum: " + oddSum);

long endTime = System.currentTimeMillis(); // 종료시점
System.out.println("실행시간: " + (endTime - startTime)/1000.0 + "초");
```

```
evenSum: -2147483648
oddSum: 0
실행시간: 9.721초
```

# 실습10-1. 스레드의 생성, 동기화, 조정(2)

2. 멀티 스레드 사용

```
public static void main(String[] arg) throws InterruptedException {
    long start = System.currentTimeMillis();
    Runnable evenSum = new Runnable() {
        public void run() {
            long evenSum = 0; // 짝수합
            for(long l = Integer.MIN_VALUE; l <= Integer.MAX_VALUE; l++)
                if(l % 2 == 0)
                    evenSum += l;
            System.out.println("evenSum: " + evenSum);
        }
    };

    Runnable oddSum = new Runnable() {
        public void run() {
            long oddSum = 0; // 홀수합
            for(long l = Integer.MIN_VALUE; l <= Integer.MAX_VALUE; l++)
                if(l % 2 != 0)
                    oddSum += l;
            System.out.println("oddSum: " + oddSum);
        }
    };

    Thread t1 = new Thread(evenSum);
    Thread t2 = new Thread(oddSum);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    long end = System.currentTimeMillis();
    System.out.println("실행시간: " + (end - start)/1000.0 + "초");
}
```

```
oddSum: 0
evenSum: -2147483648
실행시간: 5.344초
```

# 실습10-1. 스레드의 생성, 동기화, 조정(3)

- 각 스레드에서 동일한 변수에 접근하는 경우 동기화를 했을 때와 안했을 때의 차이를 알아보자.
  - Counter 클래스를 만들고 두 스레드에서 각각 내부 변수를 증가시키는 메소드와 감소시키는 메소드를 10000번 호출해보자.
    1. Counter 클래스

```
class Counter {
    private int value = 0;
    public void increment() {
        value++;
    }
    public void decrement() {
        value--;
    }
    public int getValue() {
        return value;
    }
}
```

# 실습10-1. 스레드의 생성, 동기화, 조정(4)

2. 스레드 두개에서 각각 increment(), decrement() 10000번 호출

```
public class NonSycCounterTest {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        Runnable increment = new Runnable() {
            public void run() {
                for(int i = 0; i < 10000; i++)
                    c.increment();
            }
        };
        Runnable decrement = new Runnable() {
            public void run() {
                for(int i = 0; i < 10000; i++)
                    c.decrement();
            }
        };

        Thread t1 = new Thread(increment);
        Thread t2 = new Thread(decrement);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("value: " + c.getValue());
    }
}
```

3. 실행 마다 value 값이 다른 것을 확인할 수 있다.

```
value: 230    value: 2674    value: 290    value: -2222
```

# 실습10-1. 스레드의 생성, 동기화, 조정(5)

- Counter 클래스의 메소드들을 동기화 시켜서 코드를 다시 실행해보자.

1. 동기화 된 Counter 클래스

```
class Counter {
    private int value = 0;
    public synchronized void increment() {
        value++;
    }
    public synchronized void decrement() {
        value--;
    }
    public int getValue() {
        return value;
    }
}
```

2. 실행 마다 value 값이 0으로 나오는 것을 확인할 수 있다.

```
value: 0   value: 0   value: 0   value: 0
```

# 실습10-1. 스레드의 생성, 동기화, 조정(6)

- 생산자와 소비자 타입의 예제를 통하여 스레드 간의 조정을 실습해보자.
  - 제빵사는 테이블로 케익을 가져다 둔다. 고객은 테이블에 있는 케익을 가져간다.
  - synchronized를 사용한 코드는 강의노트에 있으므로 여기서는 ReentrantLock을 사용하여 코드를 작성해보자.

1. Table 클래스

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
class Table {
    private ReentrantLock block = new ReentrantLock(); // 동기화를 위한 ReentrantLock
    private Condition blockCondi; // 스레드 조정을 위한 Condition 클래스, 기존의 wait(), notifyAll에 해당하는 메소드제공
    private int cake; // 테이블에 있는 케이크
    private boolean empty = true; // 고객이 가져가서 테이블이 비면 true
    // 비어있으면, 즉 empty가 true이면 고객의 스레드가 쉬어야 한다.
    // 테이블이 비어있지 않으면, 즉 empty가 false면 제빵사의 스레드가 쉬어야 한다.
    public Table() {
        blockCondi = block.newCondition();
    }
```

# 실습10-1. 스레드의 생성, 동기화, 조정(7)

```
    public int get() throws InterruptedException {
        block.lock(); // 다른 스레드(제빵사)가 put에 접근하지 못하게 락을 건다.
        try {
            while(empty) // 테이블이 비어있으면
                blockCondi.await(); // 고객이 쉰다.
            empty = true; // 테이블을 비운다.
            blockCondi.signal(); // 다른 스레드(제빵사)에게 알린다.
            return cake; // 가져간 케이크를 반환
        }finally {
            block.unlock(); // 다른 스레드가 put에 접근 할 수 있도록 락을 해제.
        }
    }
    public void put(int cake) throws InterruptedException {
        block.lock(); // 다른 스레드(고객)이 get에 접근하지 못하게 락을 건다.
        try {
            while(!empty) // 테이블이 비어있지 않으면
                blockCondi.await(); // 제빵사가 쉰다.
            this.cake = cake; // 비어있으면 인자로 전달된 케이크를 테이블에 올려둔다.
            empty = false; // 테이블을 채운다.
            blockCondi.signal(); // 다른 스레드(고객)에게 알린다.
        } finally {
            block.unlock(); // 다른 스레드가 get에 접근 할 수 있도록 락을 해제
        }
    }
}
```

# 실습10-1. 스레드의 생성, 동기화, 조정(8)

2. Baker 클래스

```
class Baker implements Runnable {
    private Table table; // 제빵사와 고객은 테이블을 공유한다.

    public Baker(Table table) {
        this.table = table;
    }
    @Override
    public void run() { // 스레드에서 작업할 내용 작성
        for(int i = 0; i < 10; i++) {
            try {
                table.put(i); // 0번 부터 9번까지 케이크를 테이블에 가져다둔다.
                System.out.println("제빵사: " + i + "번 케익을 생산하였습니다.");
                Thread.sleep((int)(Math.random()*1000.0)); // 생산한 후에 난수 시간만큼 쉰다.
            }catch(InterruptedException e) {
            }
        }
    }
}
```

# 실습10-1. 스레드의 생성, 동기화, 조정(9)

3. Consumer 클래스

```
class Consumer implements Runnable {
    private Table table;

    public Consumer(Table drop) {
        this.table = drop;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                int cake = table.get(); // 테이블에서 케익을 가져옴
                System.out.println("소비자: " + cake + "번 케익을 소비하였습니다.");
                Thread.sleep((int) (Math.random() * 1000.0)); // 케익을 가져간 후에 난수 시간만큼 쉰다.
            } catch (InterruptedException e) {
            }
        }
    }
}
```

# 실습10-1. 스레드의 생성, 동기화, 조정(10)

4. Baker와 Consumer의 스레드 조정 테스트 코드와 결과

```
public class BakerConsumerTest {
    public static void main(String[] args) {
        Table table = new Table();
        (new Thread(new Baker(table))).start();
        (new Thread(new Consumer(table))).start();
    }
}
```

```
제빵사: 0번 케익을 생산하였습니다.
소비자: 0번 케익을 소비하였습니다.
제빵사: 1번 케익을 생산하였습니다.
소비자: 1번 케익을 소비하였습니다.
제빵사: 2번 케익을 생산하였습니다.
소비자: 2번 케익을 소비하였습니다.
제빵사: 3번 케익을 생산하였습니다.
소비자: 3번 케익을 소비하였습니다.
제빵사: 4번 케익을 생산하였습니다.
소비자: 4번 케익을 소비하였습니다.
제빵사: 5번 케익을 생산하였습니다.
소비자: 5번 케익을 소비하였습니다.
제빵사: 6번 케익을 생산하였습니다.
소비자: 6번 케익을 소비하였습니다.
제빵사: 7번 케익을 생산하였습니다.
소비자: 7번 케익을 소비하였습니다.
제빵사: 8번 케익을 생산하였습니다.
소비자: 8번 케익을 소비하였습니다.
제빵사: 9번 케익을 생산하였습니다.
소비자: 9번 케익을 소비하였습니다.
```

# 실습10-2. 계좌이체 시뮬레이션(1)

- 동기화를 하지 않은 멀티 스레드 기반 고객 간 계좌이체 시뮬레이션 프로그램을 작성한다. (페이스북에 올라온 Bank 라이브러리를 완성)
  - Bank에는 총 100명의 Customer가 있고, 각 Customer는 1개의 Account를 가진다.
    1. 생성자 Bank(int ncus)로 ncus 만큼의 고객 배열을 만들 수 있다.
    2. addCustomer(String f, String l)를 사용하여 내부 고객 배열 필드에 f + l의 이름을 가진 고객을 추가 할수 있다.
    3. getCustomer(int index)로 내부 고객배열 필드에서 index의 고객을 리턴 받을수 있다.
  - 각 Account의 초기 금액은 1,000원이며, 따라서 Bank가 가지는 보유잔액은 100,000원이다.
    1. Customer 클래스의 addAccount(Account acct)로 고객 객체에 계좌를 추가 할 수 있다. ( Account 클래스를 CheckingAccount 클래스와 SavingsAccount 클래스가 상속받는다. 이체 업무는 CheckingAccount만 가능하다.)
    2. CheckingAccount의 생성자 CheckingAccount(double balance)를 이용해 balance 만큼의 내부 잔액 필드의 초기화가 가능하다.

# 실습10-2. 계좌이체 시뮬레이션(2)

- 임의의 두 고객 계좌 간에 임의 액수의 계좌이체를 동시다발적으로 무한반복 수행하면서, 이체 내역과 총 보유 잔액을 화면에 출력한다.
  1. Bank클래스의 transfer(int from, int to, double amount) 메소드를 완성한다.
  2. Bank의 총 보유 잔액은 getTotalBalance() 메소드를 사용한다.

- 시뮬레이션에 필요한 클래스는 다음과 같다.
  - TransferRunnable
    - 생성자
      - Bank 생성자, Account 번호, 초기 금액을 입력 받아 변수에 삽입
    - 필드
      - Bank bank
      - int fromAccount
      - double maxAccount
      - int delay // 10을 넣어줌
    - run()
      - 임의의 고객을 생성, Math.random()을 활용(Bank의 getNumofCustomers() 활용)
      - 임의로 amount 값을 생성, Math.random()을 활용
      - bank의 transfer 메소드를 호출하여 임의의 고객에게
      - Thread를 임의의 시간 동안 sleep

# 실습10-2. 계좌이체 시뮬레이션(3)

- UnsynchBankTest
  - 필드
    - public static final int NCUSTOMERS // 초기값 100
    - public static final double INITIAL_BALANCE // 초기값 1000
    - int i, cnt;
  - main method
    - Bank 타입의 변수를 선언:
    - 0부터 NCUSTOMERS까지 반복문을 수행
    - 반복문에서 Bank 변수에 addCustomer를 호출, 첫 번째 인자값에 "CNU", 두 번째 인자 값은 Integer.toString(cnt)를 활용
    - 반복문에서 Bank 변수에 getCustomer(i)에 해당하는 고객에 addAccount를 호출 (인자 값은 new CheckingAccount(INITIAL_BALANCE))
    - 0 부터 NCUSTOMERS 까지 반복문을 수행
    - bank의 transfer 메소드 호출
    - TransferRunnable 클래스를 호출하여 스레드 수행

# 실습10-2. 계좌이체 시뮬레이션(4)

- UnsynchBankTest.java 실행 결과

```
330.02 from CNU48 to CNU46 Total Balance: 99524.12
475.88 from CNU7 to CNU41 Total Balance: 100000.00
304.63 from CNU19 to CNU85 Total Balance: 94809.27
358.13 from CNU34 to CNU90 Total Balance: 94332.17
465.29 from CNU42 to CNU10 Total Balance: 94797.46
176.12 from CNU10 to CNU68 Total Balance: 94973.58
197.27 from CNU90 to CNU16 Total Balance: 95170.85
376.72 from CNU79 to CNU24 Total Balance: 95547.56
149.42 from CNU37 to CNU91 Total Balance: 95696.98
853.63 from CNU9 to CNU49 Total Balance: 96550.61
904.77 from CNU40 to CNU15 Total Balance: 97455.39
747.06 from CNU0 to CNU83 Total Balance: 98202.45
305.57 from CNU61 to CNU51 Total Balance: 98508.02
144.42 from CNU84 to CNU17 Total Balance: 98652.44
289.28 from CNU84 to CNU91 Total Balance: 98652.44
36.52 from CNU12 to CNU76 Total Balance: 98688.96
391.66 from CNU63 to CNU1 Total Balance: 99080.62
84.15 from CNU96 to CNU86 Total Balance: 99164.77
835.23 from CNU1 to CNU27 Total Balance: 100000.00
331.96 from CNU36 to CNU4 98.23 from CNU43 to CNU60 Total Balance: 97242.89
15.08 from CNU46 to CNU57 Total Balance: 97257.97
607.87 from CNU46 to CNU44 Total Balance: 97257.97
Total Balance: 97159.74
186.32 from CNU16 to CNU93 Total Balance: 97444.29
401.13 from CNU2 to CNU41 Total Balance:
```

# 실습10-3. 동기화된 계좌이체 시뮬레이션(1)

- 실습 10-2에서 만들었던 Bank 클래스를 수정해 동기화를 해보자.
- 생성자
  - bankLock에 ReentrantLock 클래스를 삽입
  - sufficientFunds에 bankLock의 newCondition메소드를 삽입
- Transfer method
  - 시작과 끝 부분에 lock과 unlock을 호출
  - try 부분에서 기존의 if문을 제거하고 while문으로 변경한 후
  - sufficientFunds.await();을 호출
  - try의 마지막 부분에 sufficientFunds.signalAll();을 호출
- getTotalBalance method
  - 시작과 끝 부분에 lock과 unlock을 호출
- 추가되는 변수
  - private Lock bankLock;
  - private Condition sufficientFunds;

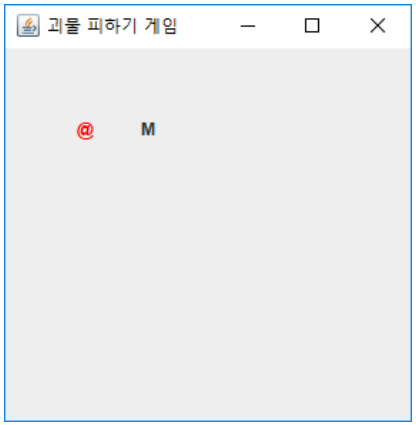
# 실습10-3. 동기화된 계좌이체 시뮬레이션(2)

- 테스트 결과

```
690.88 from CNU91 to CNU42 Total Balance:  100000.00
480.96 from CNU38 to CNU92 Total Balance:  100000.00
 72.32 from CNU38 to CNU65 Total Balance:  100000.00
555.95 from CNU79 to CNU81 Total Balance:  100000.00
188.90 from CNU73 to CNU15 Total Balance:  100000.00
752.37 from CNU42 to CNU84 Total Balance:  100000.00
953.46 from CNU95 to CNU12 Total Balance:  100000.00
 26.19 from CNU85 to CNU51 Total Balance:  100000.00
542.10 from CNU84 to CNU38 Total Balance:  100000.00
508.16 from CNU12 to CNU79 Total Balance:  100000.00
207.34 from CNU48 to CNU34 Total Balance:  100000.00
720.71 from CNU78 to CNU14 Total Balance:  100000.00
548.90 from CNU62 to CNU19 Total Balance:  100000.00
980.17 from CNU62 to CNU18 Total Balance:  100000.00
506.55 from CNU7 to CNU1 Total Balance:  100000.00
340.83 from CNU25 to CNU60 Total Balance:  100000.00
931.34 from CNU22 to CNU3 Total Balance:  100000.00
689.79 from CNU52 to CNU50 Total Balance:  100000.00
324.80 from CNU14 to CNU59 Total Balance:  100000.00
  7.31 from CNU96 to CNU3 Total Balance:  100000.00
576.43 from CNU18 to CNU28 Total Balance:  100000.00
570.21 from CNU60 to CNU39 Total Balance:  100000.00
728.98 from CNU19 to CNU52 Total Balance:  100000.00
264.05 from CNU1 to CNU80 Total Balance:  100000.00
 96.57 from CNU13 to CNU38 Total Balance:  100000.00
541.20 from CNU92 to CNU26 Total Balance:  100000.00
831.04 from CNU38 to CNU17 Total Balance:  100000.00
141.24 from CNU16 to CNU3 Total Balance:  100000.00
```

# 과제10. 괴물 피하기 게임 만들기

- 아바타와 괴물이 등장하고 괴물은 끊임 없이 아바타를 따라다니는 게임을 만들어보자.
  - 아바타는 15x15크기의 "@" 문자열 레이블로 만들고 괴물 역시 "M" 문자열 레이블로 만든다.
  - 아바타는 상,하,좌,우 키를 이용하여 패널상에서 움직이면서 도망간다.
  - 괴물은 자동으로 아바타를 추적하여 따라다닌다.
  - 괴물은 상,하,좌,우, 대각선 방향으로 움직일 수 있고 200ms마다 한 번 이동하고 그 거리는 5 픽셀이다.
  - 아바타가 괴물에게 잡히면 게임이 종료된다. (System.exit(0) 사용)



# 과제 제출 및 기한

- 제출 방법
  - 사이버캠퍼스를 통하여 제출
  - 소스코드를 제출
- 제출 기한
  - 이번 주 토요일(11/24) 자정