Race Condition부터 Deadlock까지

■ 작성일시	@2025년 8월 20일 오후 9:41
■ 복습	☐

1. 동시성 프로그래밍

- Thread - Safety는 여러 스레드가 동시에 사용되어도 문제 없음을 의미함
- 예를 들어 하나의 프로세스 내에서 여러 쓰레드들이 공유 자원 (객체나 변수 등) 사용하여도 의도한 대로 결과가 나오는 것

2. Thread - Safe 하지 않는 경우

Race Condition(경쟁 상태) : 두개 이상의 스레드가 동시에 공유 자원에 접근할 때

```
package RaceContion;

public class SimpleRaceCondition {
```

```
    static int count = 0;  // 공유 변수

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Race Condition 데모 - 여러 번 실행해보세요!");

        // 10번 반복 실행하여 Race Condition 확인
        for (int test = 1; test <= 10; test++) {
            count = 0;  // 초기화

            // 두 개 스레드가 각각 10000번씩 증가 (더 많이 해서 충돌 확률 높임)
            Thread t1 = new Thread(() -> {
                for (int i = 0; i < 10000; i++) {
                    count++;  // Race Condition 발생 가능!
                }
            });

            Thread t2 = new Thread(() -> {
                for (int i = 0; i < 10000; i++) {
                    count++;  // Race Condition 발생 가능!
                }
            });

            t1.start();
            t2.start();
            t1.join();
            t2.join();

            System.out.printf("테스트 %2d: 결과=%5d, 손실=%d%s%n",
                    test, count, (20000 - count),
                    (count == 20000) ? "" : " ← Race Condition!");
        }
    }
}
```

```
테스트  1: 결과=15666, 손실=4334 ← Race Condition!
테스트  2: 결과=13425, 손실=6575 ← Race Condition!
테스트  3: 결과=13934, 손실=6066 ← Race Condition!
테스트  4: 결과=12758, 손실=7242 ← Race Condition!
테스트  5: 결과=14762, 손실=5238 ← Race Condition!
테스트  6: 결과=16262, 손실=3738 ← Race Condition!
테스트  7: 결과=14808, 손실=5192 ← Race Condition!
테스트  8: 결과=20000, 손실=0
테스트  9: 결과=20000, 손실=0
테스트 10: 결과=16116, 손실=3884 ← Race Condition!
```

→ 공유 자원에 동시에 write 하려고 할 때 문제가 발생한다. → 동기화 메커니즘이 필요하지 않을까?

3. 동기화의 필요성

- 앞서서 Race Condition 문제를 해결하기 위해서 동기화 매커니즘이 필요성 언급
- 임계영역의 동시 접근을 막기 위해 뮤텍스와 세마포어를 사용함
  - 임계영역 : 공유 데이터의 일관성을 보장하기 위해 하나의 프로세스/스레드만 진입해서 실행 가능한 영역

3-1. 뮤텍스

- Mutual Exclusion의 줄임말로, 하나의 스레드만이 특정 자원에 접근할 수 있도록 하는 동기화 기법
- 공유 자원을 사용하기 전에 잠그고, 사용 후에 해제하는 간단한 방식
- 뮤텍스는 lock/ unlock 두가지 상태만 존재함

  → 뮤텍스를 lock한 스레드만이 unlock 할 수 있음(소유권 개념)

- JVM은 synchronized 를 사용해 뮤텍스 동기화를 암묵적 처리해줌
- 하나의 방만 있는 탈의실을 생각해보자 !

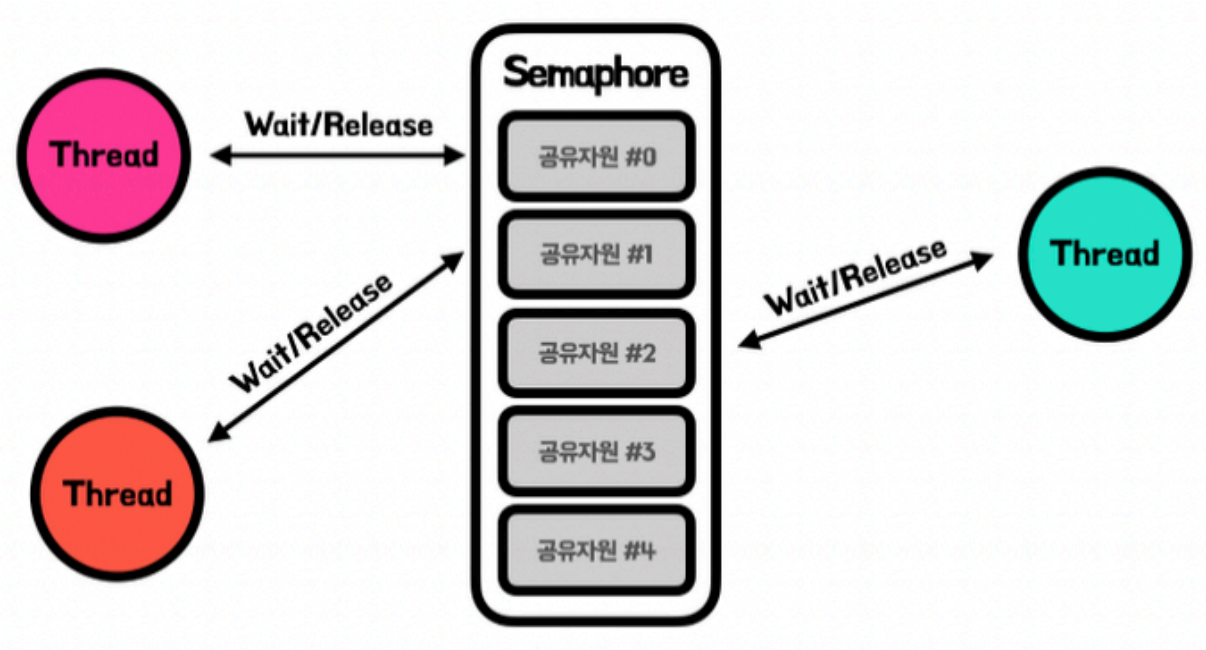
▼ 코드

```
public class ChangingRoomMutex {
    // 뮤택스 역할 - 방 1개만 있는 탈의실
    private static final Object changingRoom = new Object();

    public static void main(String[] args) {
        System.out.println("🏪 작은 탈의실 오픈! (방 1개만 이용 가능)");
        System.out.println("==========================================\n");

        // 5명의 고객이 동시에 옴
        for (int i = 1; i <= 5; i++) {
            new Thread(new Customer("고객" + i)).start();
        }
    }

    static class Customer implements Runnable {
        private String name;

        public Customer(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            try {
                System.out.println(name + ": 탈의실 도착! 방이 비어있나?");

                // 뮤택스 lock - 방 독점 사용
                synchronized (changingRoom) {
                    System.out.println("🔒 " + name + ": 방 입장 완료! (다른 사람들은 대기)");

                    // 옷 갈아입기 (1~3초 소요)
```

```
                int changingTime = (int)(Math.random() * 3 + 1) * 1000;
                System.out.println("👔 " + name + ": 옷 갈아입는 중... (" + (c
hangingTime/1000) + "초 예상)");
                Thread.sleep(changingTime);

                System.out.println("🚪 " + name + ": 옷 갈아입기 완료! 방 나
감");

            } // 뮤택스 unlock - 방 자동 해제

            System.out.println("🔓 " + name + ": 방 반납 완료! (다음 사람 입
장 가능)");

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println(name + ": 대기 중 인터럽트 발생");
        }
    }
  }
}
```

```
작은 탈의실 오픈! (방 1개만 이용 가능)
=====================================

고객2: 탈의실 도착! 방이 비어있나?
고객4: 탈의실 도착! 방이 비어있나?
고객1: 탈의실 도착! 방이 비어있나?
고객5: 탈의실 도착! 방이 비어있나?
고객3: 탈의실 도착! 방이 비어있나?
고객2: 방 입장 완료! (다른 사람들은 대기)
고객2: 옷 갈아입는 중... (3초 예상)
고객2: 옷 갈아입기 완료! 방 나감
고객3: 방 입장 완료! (다른 사람들은 대기)
고객3: 옷 갈아입는 중... (3초 예상)
고객2: 방 반납 완료! (다음 사람 입장 가능)
고객3: 옷 갈아입기 완료! 방 나감
고객3: 방 반납 완료! (다음 사람 입장 가능)
고객5: 방 입장 완료! (다른 사람들은 대기)
고객5: 옷 갈아입는 중... (3초 예상)
고객5: 옷 갈아입기 완료! 방 나감
고객5: 방 반납 완료! (다음 사람 입장 가능)
고객1: 방 입장 완료! (다른 사람들은 대기)
고객1: 옷 갈아입는 중... (1초 예상)
고객1: 옷 갈아입기 완료! 방 나감
고객1: 방 반납 완료! (다음 사람 입장 가능)
고객4: 방 입장 완료! (다른 사람들은 대기)
고객4: 옷 갈아입는 중... (3초 예상)
고객4: 옷 갈아입기 완료! 방 나감
고객4: 방 반납 완료! (다음 사람 입장 가능)
```

3-2.세마포어

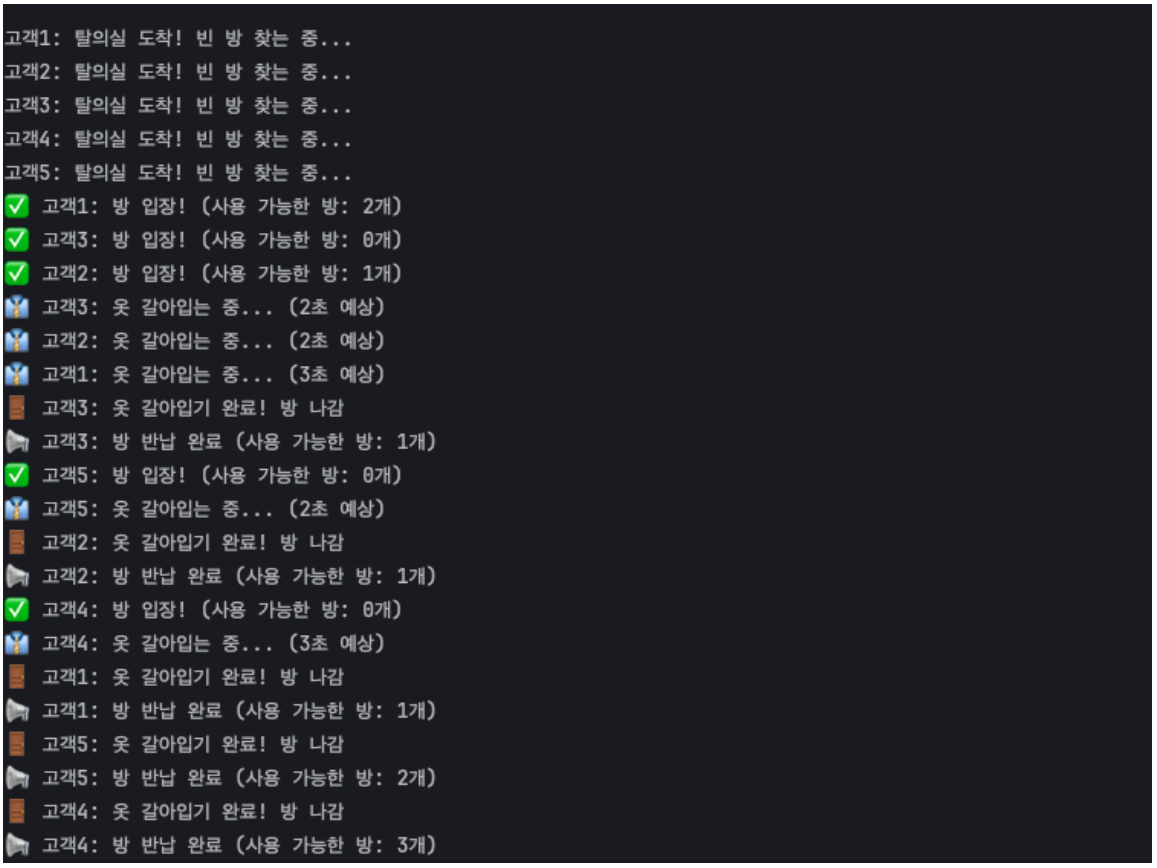


- 세마포어는 정수 값을 가지는 동기화 기법으로, 특정 자원에 접근할 수 있는 스레드의 수를 제어함
- 쓰레드가 공유 자원에 접근하면 acquire() 함수를 실행하여 정수값이 1 줄어들고, 공유 자원에서 접근을 해제하면 release() 함수를 사용하여 정수값이 1 증가함
- 여러개의 방이 있는 탈의실을 생각해보자 !

▼ 코드

```
import java.util.concurrent.Semaphore;

public class ChangingRoomSemaphore {
    // 탈의실에 3개의 방이 있음 (세마포어 초기값 = 3)
    private static final Semaphore changingRooms = new Semaphore(3);

    public static void main(String[] args) {
        System.out.println("🏪 탈의실 오픈! (총 3개 방 이용 가능)");
        System.out.println("==========================================\n");

        // 5명의 고객이 동시에 옴
        for (int i = 1; i <= 5; i++) {
            new Thread(new Customer("고객" + i)).start();
        }
    }

    static class Customer implements Runnable {
        private String name;

        public Customer(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            try {
                System.out.println(name + ": 탈의실 도착! 빈 방 찾는 중...");

                // 세마포어 acquire - 빈 방 기다리기
```

```
            changingRooms.acquire();

            System.out.println("✅ " + name + ": 방 입장! (사용 가능한 방: "
+
                    changingRooms.availablePermits() + "개)");

            // 옷 갈아입기 (1~3초 소요)
            int changingTime = (int)(Math.random() * 3 + 1) * 1000;
            System.out.println("👔 " + name + ": 옷 갈아입는 중... (" + (cha
ngingTime/1000) + "초 예상)");
            Thread.sleep(changingTime);

            System.out.println("🚪 " + name + ": 옷 갈아입기 완료! 방 나감");

            // 세마포어 release - 방 반납
            changingRooms.release();

            System.out.println("📢 " + name + ": 방 반납 완료 (사용 가능한
방: " +
                    changingRooms.availablePermits() + "개)");

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println(name + ": 대기 중 인터럽트 발생");
        }
    }
  }
}
```



3-3. 모니터 (심화)

- 뮤텍스, 세마포어만으로는 복잡한 동기화가 어려움 → 모니터 등장
- 모든 자바 객체는 모니터를 소유함 → 여러 쓰레드가 객체의 임계 영역에 진입할 때 JVM은 모니터를 사용해 스레드 간 동기화 제공
- 모니터는 상호배제, 협력을 제공함

상호배제

- `synchronized` 키워드 → JVM이 자동으로 뮤택스 처리

협력

- EntrySet(진입셋)
  - 락을 얻기 위해 대기하는 스레드들
- WaitSet(대기셋) : 조건 충족까지 기디리는 스레드들
  - `wait()` 호출 → WaitSet 이동

- `notify()` 호출 → WaitSet → EntrySet 이동

4. 동기화 주의사항

DeadLocks(교착 상태) : 두개 이상의 스레드가 서로가 점유하고 있는 자원을 얻기 위해 무한정 기다리는 상태

- 다음과 같은 네가지 조건을 동시에 모두 만족한다면 데드락 발생
  - **상호 배제(Mutual Exclusion)**: 한 번에 한 스레드만이 자원을 사용할 수 있음
  - **점유 대기(Hold and Wait)**: 자원을 가지고 있는 스레드가 다른 스레드의 자원을 기다림
  - **비선점(No Preemption)**: 어떤 스레드도 다른 스레드가 점유하고 있는 자원을 강제로 빼앗을 수 없음

- **순환 대기(Circular Wait)**: 대기하고 있는 스레드들이 자원을 서로 물고 물리는 형태로 순환을 이루게 됨

- 데드락 시나리오

1. 한번에 하나만 접근할 수 있게 함 → synchronized(resourceA)
2. A자원을 가진 채로 B 대기 → synchronized(resourceA){ synchronized(resourceB)}
3. 자원이 강제로 뺏기지 않아 강제 대기 발생

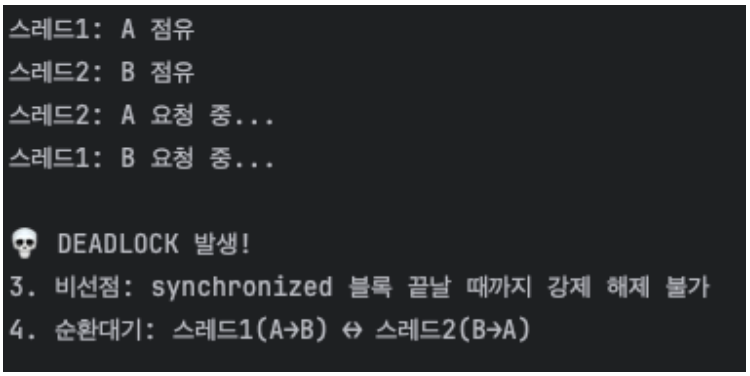
   → synchronized(resourceA){} 블록이 끝날 때 까지 resourceA 못 뺏음
4. 스레드 1 : A보유 → B요청 , 스레드 2 : B보유 → A요청 → 스레드1 → 스레드2 → 스레드1

▼ 예시 코드

```
package Deadlock;
public class SimpleDeadlock {
    static Object A = new Object();
    static Object B = new Object();

    public static void main(String[] args) throws InterruptedException {

        // 스레드1: A → B
        Thread t1 = new Thread(() → {
            synchronized (A) {          // 1. 상호배제: A 독점
                System.out.println("스레드1: A 점유");

                try {
                    Thread.sleep(100);  // 다른 스레드가 끼어들 시간 제공
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }

                System.out.println("스레드1: B 요청 중...");
                synchronized (B) {      // 2. 점유대기: A 가진 채로 B 대기
```

```
                System.out.println("완료1");
            }
        }
    });

    // 스레드2: B → A
    Thread t2 = new Thread(() → {
        synchronized (B) {          // 1. 상호배제: B 독점
            System.out.println("스레드2: B 점유");

            try {
                Thread.sleep(100);  // 다른 스레드가 끼어들 시간 제공
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }

            System.out.println("스레드2: A 요청 중...");
            synchronized (A) {      // 2. 점유대기: B 가진 채로 A 대기
                System.out.println("완료2");
            }
        }
    });

    // 동시 시작
    t1.start();
    t2.start();

    // 3초 후 데드락 확인
    Thread.sleep(3000);

    if (t1.isAlive() || t2.isAlive()) {
        System.out.println("\n💀 DEADLOCK 발생!");
        System.out.println("3. 비선점: synchronized 블록 끝날 때까지 강제
해제 불가");
        System.out.println("4. 순환대기: 스레드1(A→B) ↔ 스레드2(B→A)");
        System.out.println("\n프로그램 강제 종료");
        System.exit(0);
    } else {
```

```
            System.out.println("\n✅ 이번엔 데드락이 발생하지 않았습니다.");
            System.out.println("여러 번 실행해보세요!");
        }
    }
}
```



→ 동기화 메커니즘을 잘못 사용했을 때 데드락에 빠질 것이다 !

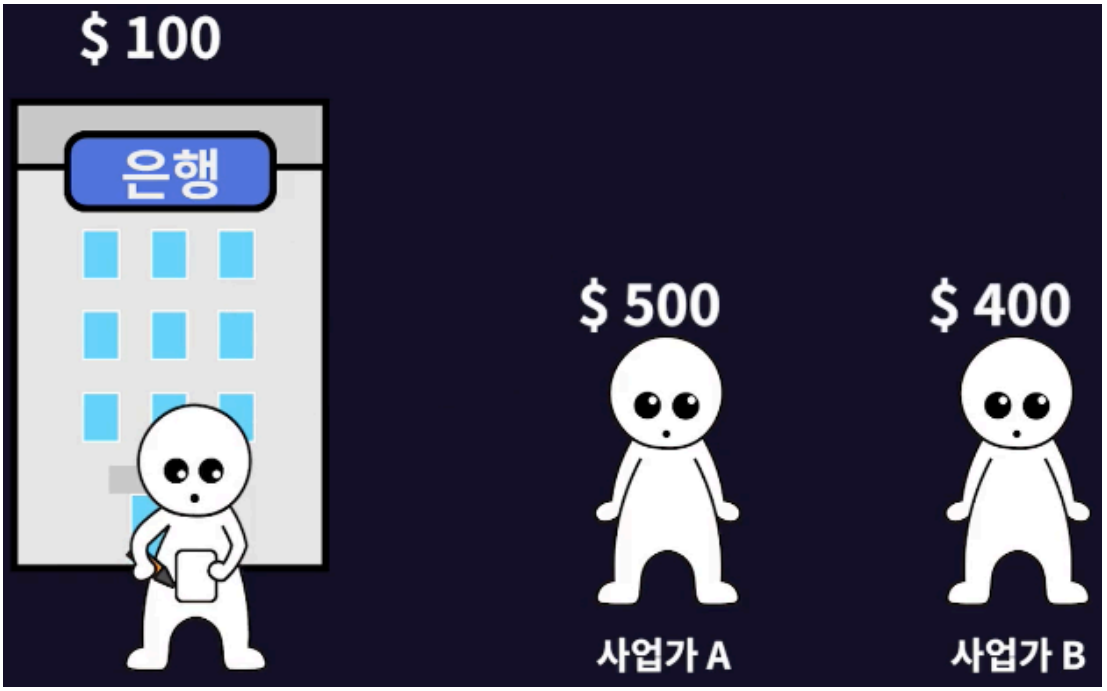
데드락 회피 방법 → 근본적인 해결 방법X

- 은행원 알고리즘 (교착상태 회피를 표현한 알고리즘)

  사업가A: 은행님 200달러만 더 빌려주면 돈 다 갚을게요 !!

  사업가B: 은행님 150달러만 더 빌려주면 돈 다 갚을게요 !!

  은행: 어라랏 .. 이미 사업가 A, 사업가 B 에게 900달러를 빌려줬는데 얘네가 무조건 100달러 이상을 빌려줘야지 돈을 갚는다고 하네... ? 난 돈을 회수할 수 있을까 ㅜㅜㅜ



- 은행은 안전하게 자금을 관리하기 위해서는 은행의 여윳돈과 사업가들에게 빌려준 돈들을 보고 대출 가능한 상황(= 안전상태)인지 확인하고 빌려줘야 함

- 운영체제에서 은행원 알고리즘을 구현하는 방법
  1. OS는 프로세스들에게 자원을 할당하기 전에 자신이 가지고 있는 전체 자원의 수를 알고 있어야 함
  2. 프로세스들은 각자 자신에게 필요한 자원의 최대 숫자를 OS에게 알려야 함
  3. 안정상태 : (OS가 가지고 있는 전체 자원의 수=시스템의 총 자원) 를 잘 관리하는 것 !

     (총 자원=14)- 9 - 6 -4 =2 → 여유 자원

프로세스	최대 요구 자원	현재 할당된 자원
P1	9	5
P2	6	4
P3	4	3

     P2의 응답만 처리 가능 → P2 응답 처리 후 총 자원 복구 → (총 자원 6)

프로세스	최대 요구 자원	현재 할당된 자원	요청이 예상되는 자원
P1	9	5	4
P2	6	4	2
P3	4	3	1

P2의 자원 반납으로 (여유자원 =2) → P1 ,P3 요청한 추가 자원 모두 할당할 수 있는 상태

프로세스	최대 요구 자원	현재 할당된 자원	요청이 예상되는 자원
P1	9	5	4
P2	6	0	6
P3	4	3	1

4. 불안정상태

프로세스	최대 요구 자원	현재 할당된 자원	요청이 예상되는 자원
P1	9	7	2
P2	6	4	2
P3	4	2	2

- 총자원 -(P1~p3=13) = 1(여유 자원) → P1~P3의 추가 요구 자원을 충족할 수 없음

5. 개발자는 동시성 문제를 어떻게 대해야 될까?

- 경쟁 상태의 원인을 정확히 파악하고, 데드락의 네가지 조건을 항상 염두하자

- 동기화를 사용한다면 적절한 기법을 선택하자
  - 뮤택스 : 단일 자원에 대한 상호 배제
  - 세마포어 : 접근 가능한 자원의 개수를 제어할 때(5개의 프린터 중 빈 프린터에 접근 허용)

    - 모니터 : 객체와 메서드를 하나의 단위로 묶어 동기화를 자동화하고, wait()/notify()를 통해 스레드 간의 협력 구현

- 데드락 예방
    - 락 획득 순서 통일 : 모든 스레드가 동일한 순서로 락을 획득하도록 설계→ 순환 대기 예방
    - 작은 단위의 락 사용: 불필요하게 큰 범위에 락 걸지 않고, 최소한의 공유 자원에만 락 적용
    - 타임아웃 설정: 락 획득 시 타임아웃 설정 → 무한 대기 상태 방지

- 테스트와 디버깅
    - 동시성 문제는 재현하기 어렵고 예측 불가능한 경우가 많음
    - 여러 차례 반복 테스트를 통해 경쟁 상태 확인하고, 디버깅 툴로 스레드의 상태를 관찰하자