# 고급 동기화 - concurrent.Lock

| 소유자 | 종수 김 |
| --- | --- |
| 태그 | |

## LockSupport1

synchronized는 자바 1.0부터 제공되는 매우 편리한 기능이나, 한계가 있음.

1. 무한 대기 : BLOCKED 상태의 쓰레드는 락이 풀릴 때 까지 무한 대기
    a. 타임아웃 X
    b. 인터럽트 X
2. 공정성 : 락이 돌아왔을 때 BLOCKED 상태의 여러 쓰레드 중에 어떤 쓰레드가 락을 획득할 지 알 수 없음.

이런 문제를 해결하기 위해 자바 1.5부터 java.util.concurrent라는 동시성 문제 해결을 위한 라이브러리 패키지 추가.

### LockSupport 기능

LockSupport를 사용하면, synchronized의 가장 큰 단점인 무한 대기 문제를 해결할 수 있음.

LockSupport는 쓰레드를 WAITING 상태로 변경.

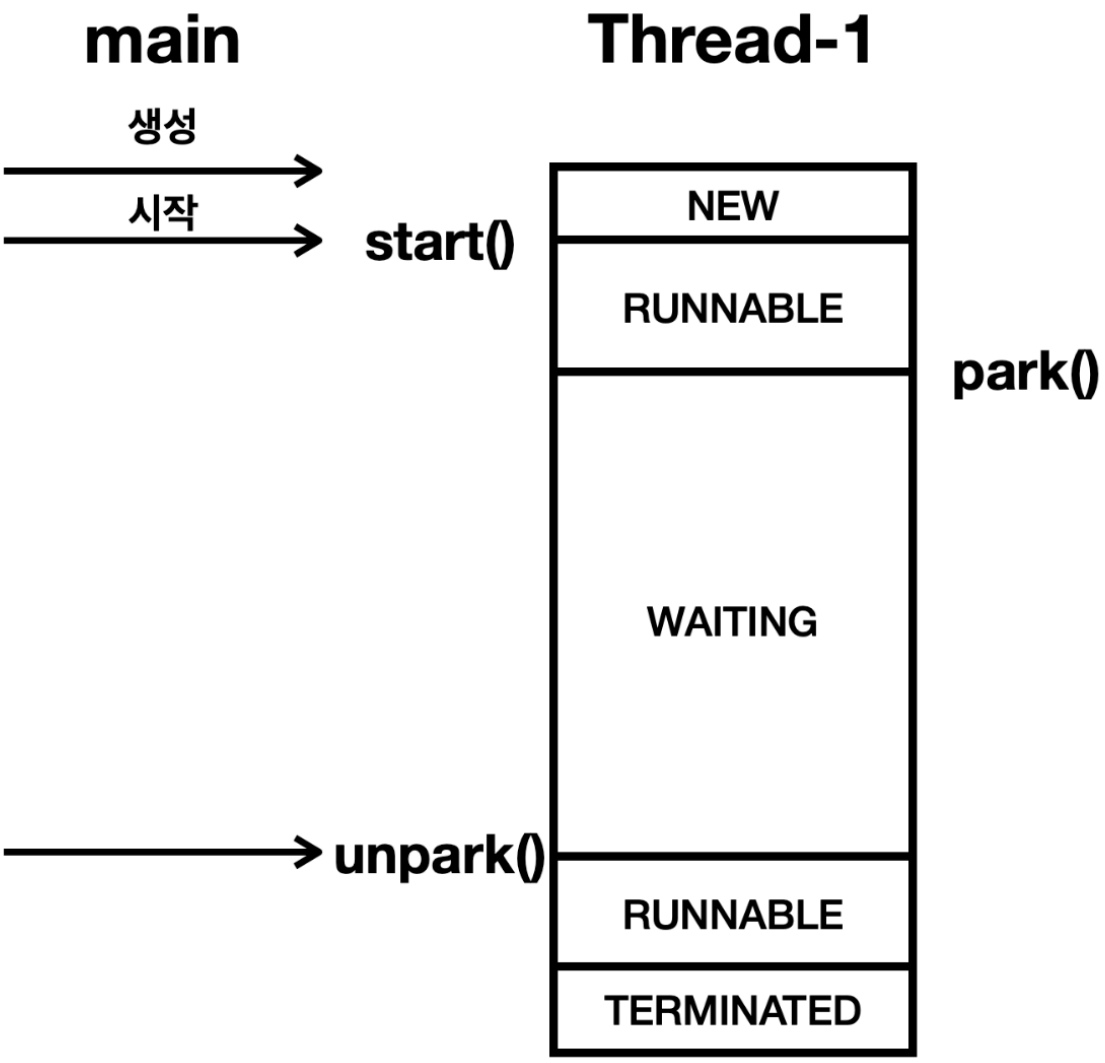
WAITING 상태는 누가 깨워주기 전 까지 계속 대기하며, CPU 실행 스케줄링에 들어가지 않음.

- park() : 쓰레드를 WAITING 상태로 변경
    - 쓰레드를 대기 상태로 둠
- parkNanos(nanos): 쓰레드를 나노초 동안만 TIMED_WAITING 상태로 변경
    - 지정된 나초고가 지나면 TIMED_WAITING 상태에서 빠져나오고 RUNNABLE상태로 둠
- unpark(thread) : WAITING 상태의 대상 쓰레드를 RUNNABLE로 바꿈

```
package lock;

import java.util.concurrent.locks.LockSupport;
```

```
import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class LockSupportMainV1 {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new ParkTest(), "Thread-1");
        thread1.start();

        // 잠시 대기하여 Thread-1이 park 상태에 빠질 시간을 줌.
        // main 쓰레드가 Thread-1이 run이 실행되기도 전에 thread-1의 상태를 찍어
버리므로 RUNNABLE이 나오기때문.
        sleep(100);
        log("Thread-1 state: " + thread1.getState());

        log("main → unpark(Thread-1)");

        LockSupport.unpark(thread1); // 1. unpark 사용
    }
    static class ParkTest implements Runnable {
        @Override
        public void run() {
            log("park 시작");
            LockSupport.park();
            log("park 종료, state : " + Thread.currentThread().getState());
            log("인터럽트 상태 : " + Thread.currentThread().isInterrupted());
        }
    }
}
```

실행 상태 그림




- main 쓰레드가 Thread-1을 start()하면 Thread-1은 RUNNABLE
- Thread-1은 Thread.park()를 통해 RUNNABLE → WAITING 상태가 되면서 대기
- main 쓰레드가 Thread-1을 unpark()로 깨우며 Thread-1은 대기 상태에서 실행 가능 상태로 변경
    - WAITING → RUNNABLE

이처럼 `LockSupport` 는 특정 스레드를 `WAITING` 상태로, 또 `RUNNABLE` 상태로 변경할 수 있다.
그런데 대기 상태로 바꾸는 `LockSupport.park()` 는 매개변수가 없는데, 실행 가능 상태로 바꾸는

`LockSupport.unpark(thread1)` 는 왜 특정 스레드를 지정하는 매개변수가 있을까?
왜냐하면 실행 중인 스레드는 `LockSupport.park()` 를 호출해서 스스로 대기 상태에 빠질

수 있지만, 대기 상태의스레드는 자신의 코드를 실행할 수 없기 때문이다. 따라서 외부 스레드의 도움을 받아야 깨어날 수 있다.

### 인터럽트 사용

```
//        LockSupport.unpark(thread1); // 1. unpark 사용
        thread1.interrupt(); // 2. interrupt() 사용

20:29:36.191 [ Thread-1] park 시작
20:29:36.270 [     main] Thread-1 state: WAITING
20:29:36.271 [     main] main -> unpark(Thread-1)
20:29:36.271 [ Thread-1] park 종료, state : RUNNABLE
20:29:36.273 [ Thread-1] 인터럽트 상태 : true
```

BLOCKED의 쓰레드 상태는, 인터럽트를 통해 깨울 수 없으나, WAITING 상태의 쓰레드는 인터럽트를 통해 깨울 수 있음.

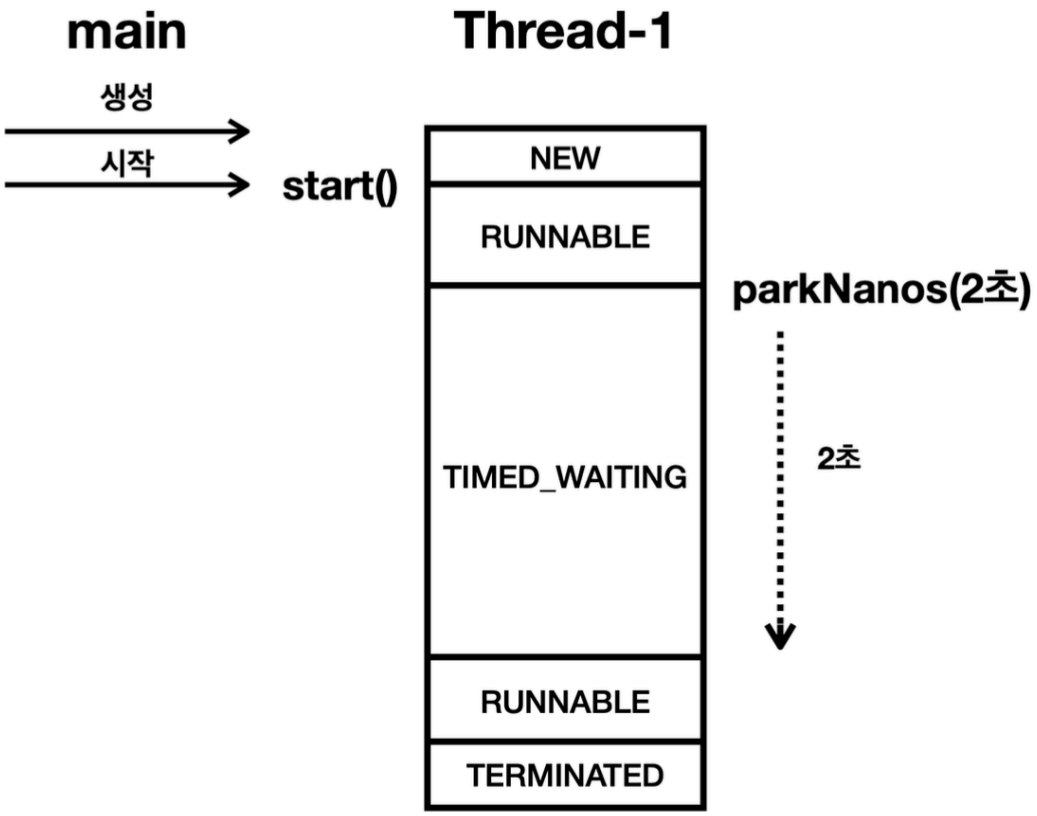
## LockSupport2

### 시간 대기

- `parkNanos(nanos)` : 스레드를 나노초 동안만 `TIMED_WAITING` 상태로 변경한다. 지정한 나노초가 지나면 `TIMED_WAITING` 상태에서 빠져나와서 `RUNNABLE` 상태로 변경된다.
- 참고로 밀리초 동안만 대기하는 메서드는 없다. `parkUntil(밀리초)` 라는 메서드가 있는데, 이 메서드는 특정 에포크(Epoch) 시간에 맞추어 깨어나는 메서드이다. 정확한 미래의 에포크 시점을 지정해야 한다.

```
package lock;

import java.util.concurrent.locks.LockSupport;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class LockSupportMainV2 {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new ParkTest(), "Thread-1");
        thread1.start();
```

```
        // 잠시 대기하여 Thread-1이 park 상태에 빠질 시간을 줌.
        sleep(100);
        log("Thread-1 state: " + thread1.getState());

    }
    static class ParkTest implements Runnable {
        @Override
        public void run() {
            log("park 시작");
            LockSupport.parkNanos(2000_000000);
            log("park 종료, state : " + Thread.currentThread().getState());
            log("인터럽트 상태 : " + Thread.currentThread().isInterrupted());
        }
    }
}

20:32:47.196 [ Thread-1] park 시작
20:32:47.280 [     main] Thread-1 state: TIMED_WAITING
20:32:49.202 [ Thread-1] park 종료, state : RUNNABLE
20:32:49.205 [ Thread-1] 인터럽트 상태 : false
```




- `Thread-1`은 `parkNanos(2초)`를 사용해서 2초간 `TIMED_WAITING` 상태에 빠진다.
- `Thread-1`은 2초 이후에 시간 대기 상태(`TIMED_WAITING`)를 빠져나온다.

## BLOCKED vs WAITING

WAITING 상태에 특정 시간까지만 대기하는 기능이 포함된 것이 TIMED_WAITING임.

둘을 묶어서 WAITING 상태

**인터럽트**

- BLOCKED 상태는 인터럽트가 걸려도 대기 상태를 빠져나오지 못함. 여전히 BLOCKED
- WAITING 상태는 인터럽트가 걸리면 대기 상태를 빠져나온 후, RUNNABLE 상태로 변함

**용도**

- BLOCKED 상태는 자바의 synchronized에서 락을 획득하기 위해 대기할 때 사용
- WAITING, TIMED_WAITING 상태는 쓰레드가 특정 조건이나 시간 동안 대기할 때 발생하는 상태

***대기(** `WAITING` **) 상태와 시간 대기 상태(** `TIMED_WAITING` **)는 서로 짝이 있다.***

- `Thread.join()` , `Thread.join(long millis)`
- `LockSupport.park()` , `LockSupport.parkNanos(long nanos)`
- `Object.wait()` , `Object.wait(long timeout)`

***참고***: `Object.wait()` 는 뒤에서 다룬다.

`BLOCKED` , `WAITING` , `TIMED_WAITING` 상태 모두 스레드가 대기하며, 실행 스케줄링에 들어가지 않기 때문에,CPU 입장에서 보면 실행하지 않는 비슷한 상태이다.
`BLOCKED` 상태는 `synchronized` 에서만 사용하는 특별한 대기 상태라고 이해하면 된다.

`WAITING` , `TIMED_WAITING` 상태는 범용적으로 활용할 수 있는 대기 상태라고 이해하면 된다.

## LockSupport 정리

LockSupport를 사용하면 쓰레드를 WAITING, TIMED_WAITING 상태로 변경할 수 있으며, 인터럽트를 받아서 쓰레드를 깨울 수도 있음.

이런 기능 들을 통해 synchronized의 단점인 무한 대기 문제를 해결할 수 있음.

```
if (!lock.tryLock(10초)) { // 내부에서 parkNanos() 사용
log("[진입 실패] 너무 오래 대기했습니다.");
return false;

}

//임계 영역 시작
...
//임계 영역 종료
lock.unlock() // 내부에서 unpark() 사용
```

락( `lock` )이라는 클래스를 만들고, 특정 스레드가 먼저 락을 얻으면 `RUNNABLE` 로 실행하고, 락을 얻지 못하면`park()` 를 사용해서 대기 상태로 만드는 것이다. 그리고 스레드가 임계 영역의 실행을 마치고 나면 락을 반납하고,
`unpark()` 를 사용해서 대기 중인 다른 스레드를 깨우는 것이다. 물론 `parkNanos()` 를 사용해서 너무 오래 대기하면 스레드가 스스로 중간에 깨어나게 할 수도 있다.

하지만 이런 기능을 직접 구현하기는 매우 어렵다.

예를 들어 스레드 10개를 동시에 실행했는데, 그중에 딱 1개의 스레드만 락을 가질 수 있도록 락 기능을 만들어야 한다. 그리고 나머지 9개의 스레드가 대기해야 하는데, 어떤 스레드가

대기하고 있는지 알 수 있는 자료구조가 필요하다. 그래야 이후에 대기 중인 스레드를 찾아서 깨울 수 있다.

여기서 끝이아니다. 대기 중인 스레드 중에 어떤 스레드를 깨울지에 대한 우선순위 결정도 필요하다.

한마디로 `LockSupport` 는 너무 저수준이다. `synchronized` 처럼 더 고수준의 기능이 필요하다.

`ReentrantLock` 은 `LockSupport` 를 활용해서 `synchronized` 의 단점을 극복하면서도 매우 편리하게 임계 영역을 다룰 수 있는 다양한 기능을 제공한다.

# ReentrantLock - 이론

자바는 1.0부터 존재한 `synchronized` 와 `BLOCKED` 상태를 통한 통한 임계 영역 관리의 한계를 극복하기 위해 자바 1.5부터 `Lock` 인터페이스와 `ReentrantLock` 구현체를 제공한다.

**Lock 인터페이스**

```
package java.util.concurrent.locks;
 public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

Lock 인터페이스는 동시성 프로그래밍에서 쓰이는 안전한 임계 영역을 위한 락을 구현하는데 사용.

Lock 인터페이스는 다음과 같은 메서드를 제공하며, 대표적인 구현체로 ReentrantLock이 있음.

1. void lock()
    - 락을 획득한다. 만약 다른 스레드가 이미 락을 획득했다면, 락이 풀릴 때까지 현재 스레드는 대기( `WAITING` )한다. 이 메서드는 인터럽트에 응답하지 않는다.
    - 예) 맛집에 한번 줄을 서면 끝까지 기다린다. 친구가 다른 맛집을 찾았다고 중간에 연락해도 포기하지 않고 기다린다.

> 참고
> 여기서 사용하는 락은 객체 내부에 있는 모니터 락이 아니다! `Lock` 인터페이스와 `ReentrantLock` 이 제공하는 기능이다!
>
> 모니터 락과 `BLOCKED` 상태는 `synchronized` 에서만 사용된다.

2. void lockInterruptibly()
    - 락 획득을 시도하되, 다른 스레드가 인터럽트할 수 있도록 한다. 만약 다른 스레드가 이미 락을 획득했다면, 현재스레드는 락을 획득할 때까지 대기한다.
    - 대기 중에 인터럽트가 발생하면 `InterruptedException` 이 발생하며 락 획득을 포기한다.
    - 예) 맛집에 한번 줄을 서서 기다린다. 다만 친구가 다른 맛집을 찾았다고 중간에 연락하면 포기한다.
3. boolean tryLock()
    - 락 획득을 시도하고, 즉시 성공 여부를 반환한다. 만약 다른 스레드가 이미 락을 획득했다면 `false` 를 반환하고,그렇지 않으면 락을 획득하고 `true` 를 반환한다.
    - 예) 맛집에 대기 줄이 없으면 바로 들어가고, 대기 줄이 있으면 즉시 포기한다.
4. boolean tryLock(long time, TimeUnit unit)
    - 주어진 시간 동안 락 획득을 시도한다. 주어진 시간 안에 락을 획득하면 `true` 를 반환한다. 주어진 시간이 지나도락을 획득하지 못한 경우 `false` 를 반환한다.
    - 이 메서드는 대기 중 인터럽트가 발생하면`InterruptedException` 이 발생하며 락 획득을 포기한다.
    - 예) 맛집에 줄을 서지만 특정 시간 만큼만 기다린다. 특정 시간이 지나도 계속 줄을 서야 한다면 포기한다. 친구가다른 맛집을 찾았다고 중간에 연락해도 포기한다.
5. void unlock()
    - 락을 해제한다. 락을 해제하면 락 획득을 대기 중인 스레드 중 하나가 락을 획득할 수 있게 된다.
    - 락을 획득한 스레드가 호출해야 하며, 그렇지 않으면 `IllegalMonitorStateException` 이 발생할 수 있다.
    - 예) 식당안에 있는 손님이 밥을 먹고 나간다. 식당에 자리가 하나 난다. 기다리는 손님께 이런 사실을 알려주어야한다. 기다리던 손님중 한 명이 식당에 들어간다.
6. Condition newCondition()

- `Condition` 객체를 생성하여 반환한다. `Condition` 객체는 락과 결합되어 사용되며, 스레드가 특정 조건을기다리거나 신호를 받을 수 있도록 한다.
- 이는 `Object` 클래스의 `wait` , `notify` , `notifyAll` 메서드와 유사한 역할을 한다.
  참고로 이 부분은 뒤에서 자세히 다룬다.

**이 메서드들을 사용하면 고수준의 동기화 기법을 구현할 수 있다.** `Lock` 인터페이스는 `synchronized` 블록보다 더많은 유연성을 제공하며, 특히 락을 특정 시간 만큼만 시도하거나, 인터럽트 가능한 락을 사용할 때 유용하다.
이 메서드들을 보면 알겠지만 다양한 메서드를 통해 `synchronized` 의 단점인 무한 대기 문제도 깔끔하게 해결할 수있다.

***참고***: `lock()` 메서드는 인터럽트에 응하지 않는다고 되어있다. 이 메서드의 의도는 인터럽트가 발생해도 무시하고 락을 기다리는 것이다.

앞서 대기( `WAITING` ) 상태의 스레드에 인터럽트가 발생하면 대기 상태를 빠져나온다고 배웠다. 그런데`lock()` 메서드의 설명을 보면 대기( `WAITING` ) 상태인데 인터럽트에 응하지 않는다고 되어있다. 어떻게 된 것일까?

`lock()` 을 호출해서 락을 얻기 위해 대기중인 스레드에 인터럽트가 발생하면 순간 대기 상태를 빠져나오는 것은 맞다.
그래서 아주짧지만 `WAITING` `RUNNABLE` 이된다.그런데 `lock()` 메서드안에서해당스레드를다시
`WAITING` 상태로 강제로 변경해버린다. 이런 원리로 인터럽트를 무시하는 것이다. 참고로 인터럽트가 필요하면
`lockInterruptibly()` 를 사용하면 된다. 새로운 `Lock` 은 개발자에게 다양한 선택권을 제공한다.

## 공정성

무한대기에 대한 문제는 Lock 인터페이스를 통해 해결할 수 있으나, '공정성' 문제가 발생할 수 있음.

락이 돌아왔을 때 BLOCKED 상태의 여러 쓰레드 중에 어떤 쓰레드가 락을 획득할 지 알 수 없음. 최악의 경우 특정 쓰레드가 너무 오랜기간 락을 획득하지 못할 수 있음.

Lcok 인터페이스의 대표적인 구현체로 ReentrantLock이 있는데, 이 클래스는 쓰레드가 공정하게 락을 얻을 수 있는 메서드를 제공.

```
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockEx { // 비공정 모드 락
    private final Lock nonFairLock = new ReentrantLock(); // 공정 모드 락
    private final Lock fairLock = new ReentrantLock(true);
```

```
    public void nonFairLockTest() {
    nonFairLock.lock();
    try {// 임계 영역

    } finally {
      nonFairLock.unlock();
    }
  }
  public void fairLockTest() {

    fairLock.lock();
    try {// 임계 영역

    } finally {
      fairLock.unlock();
    }
    }
}
```

## 비공정 모드 (Non-fair mode)

비공정 모드는 `ReentrantLock` 의 기본 모드이다. 이 모드에서는 락을 먼저 요청한 스레드가 락을 먼저 획득한다는보장이 없다.

락을 풀었을 때, 대기 중인 스레드 중 아무나 락을 획득할 수 있다. 이는 락을 빨리 획득할 수 있지만, 특정스레드가 장기간 락을 획득하지 못할 가능성도 있다.

- ***비공정 모드 특징***
  - ****성능 우선***: 락을 획득하는 속도가 빠르다.
  - ****선점 가능***: 새로운 스레드가 기존 대기 스레드보다 먼저 락을 획득할 수 있다.
  - ****기아 현상 가능성***: 특정 스레드가 계속해서 락을 획득하지 못할 수 있다.

## 공정 모드 (Fair mode)

생성자에서 `true` 를 전달하면 된다. 예) `new ReentrantLock(true)`

공정 모드는 락을 요청한 순서대로 스레드가 락을 획득할 수 있게 한다.

이는 먼저 대기한 스레드가 먼저 락을 획득하게되어 스레드 간의 공정성을 보장한다.

그러나 이로 인해 성능이 저하될 수 있다.

- ***공정 모드 특징***
  - ****공정성 보장***: 대기 큐에서 먼저 대기한 스레드가 락을 먼저 획득한다.
  - ****기아 현상 방지***: 모든 스레드가 언젠가 락을 획득할 수 있게 보장된다.
  - ****성능 저하***: 락을 획득하는 속도가 느려질 수 있다.

### 비공정, 공정 모드 정리

- 비공정 모드는 성능을 중시하고, 쓰레드가 락을 빨리 획득할 수 있지만, 특정 쓰레드가 계속해서 락을 획득하지 못할 수 있음.
- 공정 모드는 스레드가 락을 획득하는 순서를 보장하여 공정성을 중시하지만, 성능이 저하될 수 있다.

> Lock 인터페이스와 ㅏ ReentrantLock 구현체를 사용하면 synchronized 단점인 무한 대기와 공정성 문제를 모두 해결 가능.

## ReentrantLock - 활용

```
package thread.sync;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class BankAccountV4 implements BankAccount{
    private int balance;

    private final Lock lock = new ReentrantLock();

    public BankAccountV4(int initialBalance) {
        this.balance = initialBalance;
    }

    @Override
    public boolean withdraw(int amount) {
        log("거래 시작 : " + getClass().getSimpleName());
```

```
        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);

        lock.lock(); // ReentrantLock을 이용하여
        try {
            if (balance < amount) {
                log("[검증 실패]");
                return false;
            }

            // 잔고가 출금액 보다 많으면 진행
            log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
            sleep(1000); // 출금에 걸리는 시간으로 가정
            balance -= amount;

            log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);
        } finally {
            lock.unlock();
        }

        log("거래 종료");
        return true;
    }

    @Override
    public int getBalance() {
        return balance;
    }
}
```

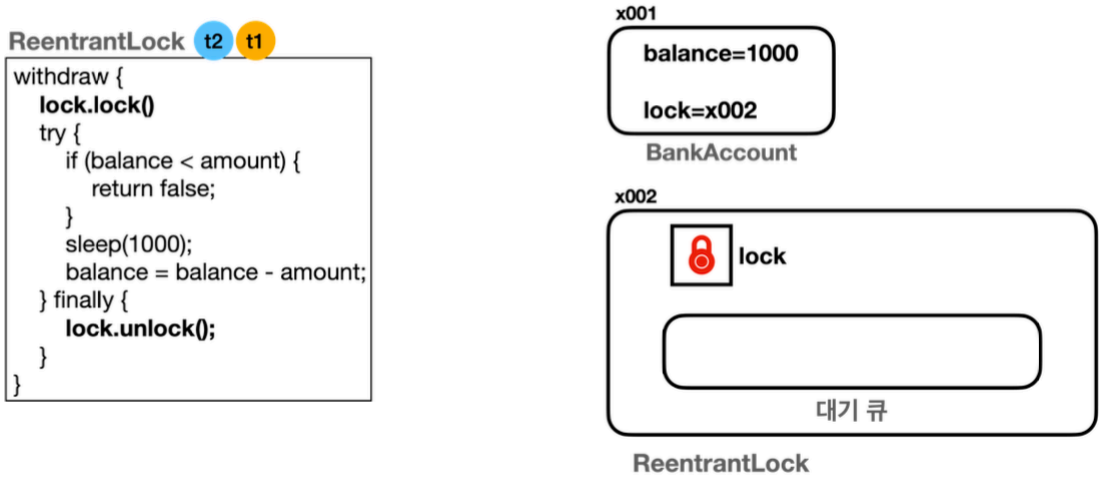
- `private final Lock lock = new ReentrantLock()` 을 사용하도록 선언한다.
- `synchronized(this)` 대신에 `lock.lock()` 을 사용해서 락을 건다.
    - `lock()` `unlock()` 까지는 안전한 임계 영역이 된다.

> **임계 영역이 끝나면 반드시! 락을 반납해야 한다. 그렇지 않으면 대기하는 스레드가 락을 얻지 못한다.**
> **따라서 `lock.unlock()` 은 반드시 `finally` 블럭에 작성해야한다.**

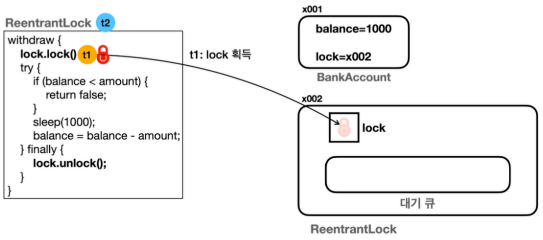
> **이렇게 하면 검증에 실패해서 중간에 `return` 을 호출해도 또는 중간에 예상치 못한 예외가 발생해도 `lock.unlock()` 이 반드시 호출된다.**

***주의!***
여기서 사용하는 락은 객체 내부에 있는 모니터 락이 아니다! `Lock` 인터페이스와 `ReentrantLock` 이 제공하는 기능이다!
모니터 락과 `BLOCKED` 상태는 `synchronized` 에서만 사용된다.

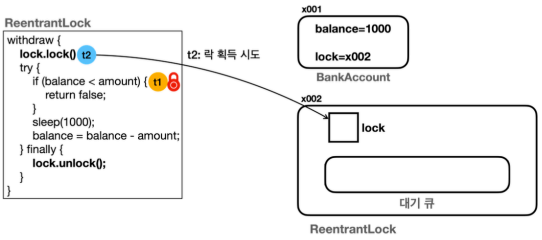
**실행 결과 분석**



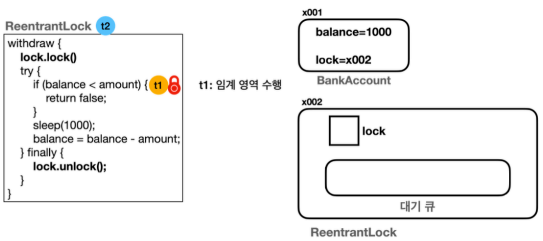

- `t1` , `t2` 가 출금을 시작한다. 여기서는 `t1` 이 약간 먼저 실행된다고 가정하겠다.
- `ReenterantLock` 내부에는 락과 락을 얻지 못해 대기하는 스레드를 관리하는 대기 큐가 존재한다.
- 여기서 이야기하는 락은 객체 내부에 있는 모니터 락이 아니다. `ReentrantLock` 이 제공하는 기능이다.



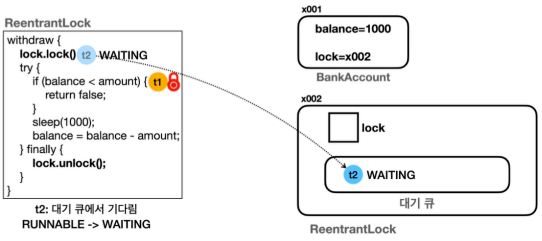

- `t1` : `ReenterantLock` 에 있는 락을 획득한다.
- 락을 획득하는 경우 `RUNNABLE` 상태가 유지되고, 임계 영역의 코드를 실행할 수 있다.



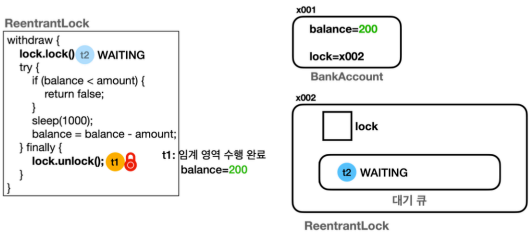

- `t2` : `ReenterantLock` 에 있는 락의 획득을 시도한다. 하지만 락이 없다.



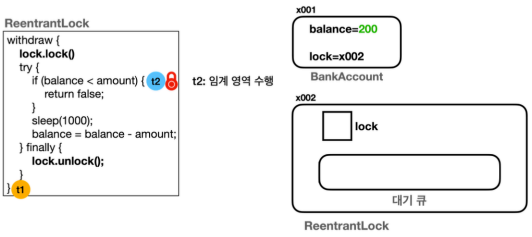

- `t1` : 임계 영역의 코드를 실행한다.



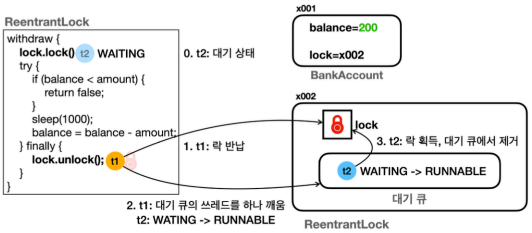

- `t2` : 락을 획득하지 못하면 `WAITING` 상태가 되고, 대기 큐에서 관리된다.
  - `LockSupoort.park()` 가 내부에서 호출된다.
- 참고로 `tryLock(long time, TimeUnit unit)` 와 같은 시간 대기 기능을 사용하면 `TIMED_WAITING` 이 되고, 대기 큐에서 관리된다.



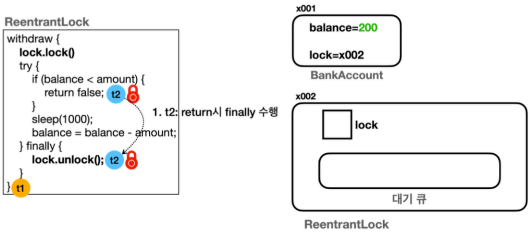

- `t1` : 임계 영역의 수행을 완료했다. 이때 잔액은 `balance=200` 이 된다.



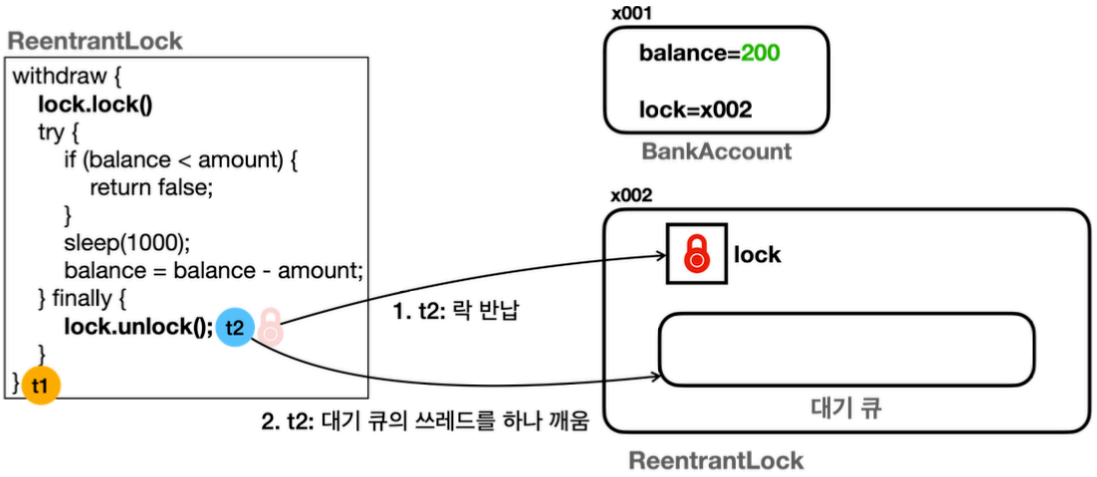

- `t2` : 락을 획득한 `t2` 스레드는 `RUNNABLE` 상태로 임계 영역을 수행한다.



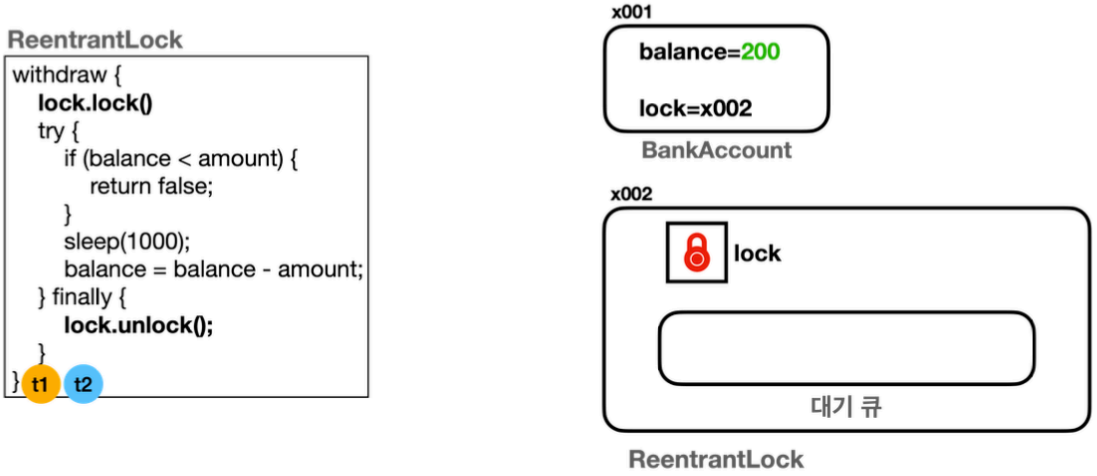

- `t1` : 임계 영역을 수행하고 나면 `lock.unlock()` 을 호출한다.
  - **1. t1**: 락을 반납한다.
  - **2. t1**: 대기 큐의 스레드를 하나 깨운다. `LockSupoort.unpark(thread)` 가 내부에서 호출된다.
  - **3. t2**: `RUNNABLE` 상태가 되면서 깨어난 스레드는 락 획득을 시도한다.
    - 이때 락을 획득하면 `lock.lock()` 을 빠져나오면서 대기 큐에서도 제거된다.
    - 이때 락을 획득하지 못하면 다시 대기 상태가 되면서 대기 큐에 유지된다.
      - 참고로 락 획득을 시도하는 짧은 사이에 새로운 스레드가 락을 먼저 가져갈 수 있다.
      - 공정 모드의 경우 대기 큐에 먼저 대기한 스레드가 먼저 락을 가져간다.




- `t2` : 잔액[200]이 출금액[800]보다 적으므로 검증 로직을 통과하지 못한다. 따라서 검증 실패이다. `return false` 가 호출된다.
- 이때 `finally` 구문이 있으므로 `finally` 구문으로 이동한다.




- `t2` : `lock.unlock()`을 호출해서 락을 반납하고, 대기 큐의 스레드를 하나 깨우려고 시도한다. 대기 큐에 스레드가 없으므로 이때는 깨우지 않는다.




- **완료 상태**

**참고**: `volatile`를 사용하지 않아도 `Lock`을 사용할 때 접근하는 변수의 메모리 가시성 문제는 해결된다. (이전에 학습한 자바 메모리 모델 참고)

- ReentrantLock은 내부에 락과 대기 큐를 가지고 있음.
  - Lock을 획득한 쓰레드는 RUNNABLE
  - Lock획득을 기다리는 쓰레드는 대기 큐에서 WAITING 상태 (tryLock인 경우 TIMED_WAITING)를 유지.
  - 이후 Lock을 획득한 쓰레드가 unlock()을 통해 반납하면,
  - ReentrantLock 인스턴스는, 대기 큐에 있는 WAITING 상태의 쓰레드를 꺼내어 RUNNABLE로 만들고, 해당 쓰레드는 lock()에 의해 새로운 '락을 획득'하여 로직을 진행하는 것.

# ReentrantLock - 대기 중단

ReentrantLock을 사용하면, 락을 무한대기 하지 않고, 중간에 빠져나오는 것이 가능.

심지어, 락을 얻을 수 없다면 기다리지 않고 즉시 빠져나오는 것도 가능.

`boolean tryLock()` **

- 락 획득을 시도하고, 즉시 성공 여부를 반환한다. 만약 다른 스레드가 이미 락을 획득했다면 `false` 를 반환하고,그렇지 않으면 락을 획득하고 `true` 를 반환한다.
- 예) 맛집에 대기 줄이 없으면 바로 들어가고, 대기 줄이 있으면 즉시 포기한다.*

`boolean tryLock(long time, TimeUnit unit)` **

- 주어진 시간 동안 락 획득을 시도한다. 주어진 시간 안에 락을 획득하면 `true` 를 반환한다. 주어진 시간이 지나도락을 획득하지 못한 경우 `false` 를 반환한다.
- 이 메서드는 대기 중 인터럽트가 발생하면`InterruptedException` 이 발생하며 락 획득을 포기한다.
- 예) 맛집에 줄을 서지만 특정 시간 만큼만 기다린다. 특정 시간이 지나도 계속 줄을 서야한다면 포기한다. 친구가다른 맛집을 찾았다고 중간에 연락해도 포기한다.

## tryLcok 예시

```
package thread.sync;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class BankAccountV5 implements BankAccount{
    private int balance;

    private final Lock lock = new ReentrantLock();

    public BankAccountV5(int initialBalance) {
        this.balance = initialBalance;
    }

    @Override
```

```
    public boolean withdraw(int amount) {
        log("거래 시작 : " + getClass().getSimpleName());

        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);

        if(!lock.tryLock()) {
            log("[진입 실패] 이미 처리중인 작업이 있습니다.");
            return false;
        }

        lock.lock(); // ReentrantLock을 이용하여 임계 영역을 설정
        try {
            if (balance < amount) {
                log("[검증 실패]");
                return false;
            }

            // 잔고가 출금액 보다 많으면 진행
            log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
            sleep(1000); // 출금에 걸리는 시간으로 가정
            balance -= amount;

            log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);
        } finally {
            lock.unlock();
        }

        log("거래 종료");
        return true;
    }

    @Override
    public int getBalance() {
        return balance;
    }
}
```

```
21:10:39.605 [       t2] 거래 시작 : BankAccountV5
21:10:39.605 [       t1] 거래 시작 : BankAccountV5
21:10:39.613 [       t1] [검증 시작] 출금액 : 800, 잔액: 1000
21:10:39.614 [       t1] [검증 완료] 출금액 : 800, 잔액: 1000
21:10:39.614 [       t2] [검증 시작] 출금액 : 800, 잔액: 1000
21:10:39.614 [       t2] [진입 실패] 이미 처리중인 작업이 있습니다.
21:10:40.086 [     main] t1 state: TIMED_WAITING
21:10:40.086 [     main] t2 state: TERMINATED
21:10:40.614 [       t1] [출금 완료] 출금액 : 800, 잔액: 200
21:10:40.615 [       t1] 거래 종료
21:10:40.617 [     main] 최종 잔액 : 200
```

1. `t1` : 먼저 락을 획득하고 임계 영역을 수행한다.
2. `t2` : 락이 없다는 것을 확인하고 `lock.tryLock()` 에서 즉시 빠져나온다. 이때 `false` 가 반환된다.
3. `t2` : "[진입 실패] 이미 처리중인 작업이 있습니다."를 출력하고 `false` 를 반환하면서 메서드를 종료한다.`
4. t1` : 임계 영역의 수행을 완료하고 거래를 종료한다. 마지막으로 락을 반납한다.

## tryLock(시간) 예시

```
package thread.sync;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import static util.MyLoggerThread.log;
import static util.ThreadUtils.sleep;

public class BankAccountV6 implements BankAccount{
    private int balance;

    private final Lock lock = new ReentrantLock();

    public BankAccountV6(int initialBalance) {
        this.balance = initialBalance;
```

```
    }

    @Override
    public boolean withdraw(int amount) {
        log("거래 시작 : " + getClass().getSimpleName());

        // 잔고가 출금액 보다 적으면 진행하면 안됨.
        log("[검증 시작] 출금액 : " + amount + ", 잔액: " + balance);

        try {
            if(!lock.tryLock(500, TimeUnit.MILLISECONDS)) {
                log("[진입 실패] 이미 처리중인 작업이 있습니다.");
                return false;
            }
        } catch (InterruptedException e) { // 500ms를 기다리는 동안 인터럽트가
터진다면 여기서 처리
            throw new RuntimeException(e);
        }

        try {
            if (balance < amount) {
                log("[검증 실패]");
                return false;
            }

            // 잔고가 출금액 보다 많으면 진행
            log("[검증 완료] 출금액 : " + amount + ", 잔액: " + balance);
            sleep(1000); // 출금에 걸리는 시간으로 가정
            balance -= amount;

            log("[출금 완료] 출금액 : " + amount + ", 잔액: " + balance);
        } finally {
            lock.unlock();
        }

        log("거래 종료");
        return true;
    }
```

```
    @Override
    public int getBalance() {
        return balance;
    }
}
```

실행 결과

```
16:33:54.246 [       t1] 거래 시작: BankAccountV6
16:33:54.246 [       t2] 거래 시작: BankAccountV6
16:33:54.252 [       t1] [검증 시작] 출금액: 800, 잔액: 1000
16:33:54.735 [     main] t1 state: TIMED_WAITING //sleep(1000)
16:33:54.736 [     main] t2 state: TIMED_WAITING //tryLock(500)
16:33:54.751 [       t2] [진입 실패] 이미 처리중인 작업이 있습니다.
16:33:55.258 [       t1] [출금 완료] 출금액: 800, 변경 잔액: 200
16:33:55.258 [       t1] 거래 종료
16:33:55.261 [     main] 최종 잔액: 200
```

실행 결과를 분석해보자.

- `t1` : 먼저 락을 획득하고 임계 영역을 수행한다.
- `t2` : `lock.tryLock(0.5초)` 을 호출하고 락 획득을 시도한다. 락이 없으므로 0.5초간 대기한다.
  - 이때 `t2` 는 `TIMED_WAITING` 상태가 된다.
  - 내부에서는 `LockSupport.parkNanos(시간)` 이 호출된다.
- `t2` : 대기 시간인 0.5초간 락을 획득하지 못했다. `lock.tryLock(시간)` 에서 즉시 빠져나온다. 이때 `false` 가 반환된다.
  - 스레드는 `TIMED_WAITING` ➜ `RUNNABLE` 이 된다.
- `t2` : "[진입 실패] 이미 처리중인 작업이 있습니다."를 출력하고 `false` 를 반환하면서 메서드를 종료한다.
- `t1` : 임계 영역의 수행을 완료하고 거래를 종료한다. 마지막으로 락을 반납한다.