

7장

concurrent.lock

- Synchronized의 단점 ↑ parkNanos(ns) 사용하여 해결
- 무한 대기 : BLOCKED 상태의 스케드는 락이 해제될 때까지 무한 대기
 - timeout이 없고 종가지 interrupt도 X → park(), parkNanos(ns)는 인터럽트 가능
 - 공정성 : 락을 어떤 스케드가 획득할지 알 수 없으며 최악의 경우 특정 스케드가 너무 오래 기간 락을 획득하지 못할 수 있다. (마지 starvation 같은?)

↓ 자바 1.5에 도입된
java.util.concurrent 패키지 추가

그리고, LockSupport를 사용하면 synchronized의 무한 대기 문제 해결 가능

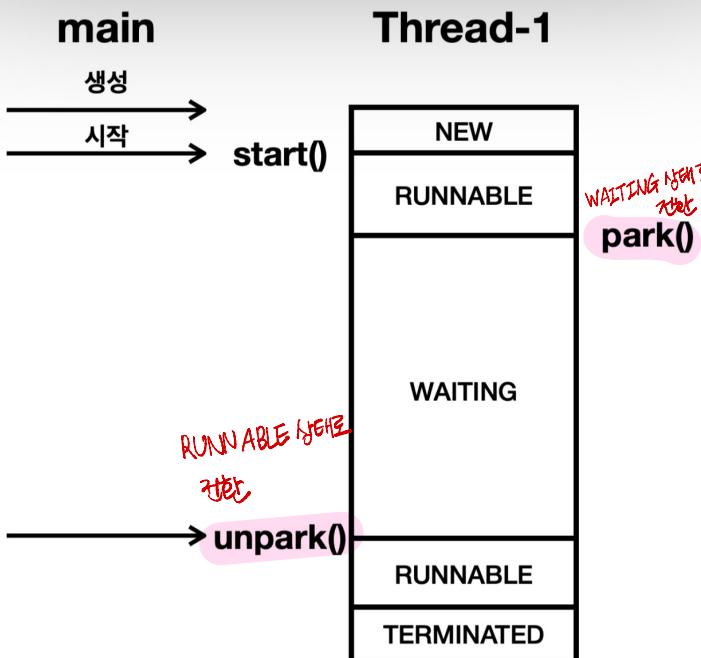
LockSupport의 기능 → 스케드를 WAITING 상태로 변경

- park() : 스케드를 WAITING 상태로 변경 \Rightarrow 고차하다
- parkNanos(nanos) : 스케드를 나온 동안 TIMED-WAITING 상태
- unpark(Thread) : WAITING 상태의 대상 스케드를 RUNNABLE로 변경

```
log("main -> unpark(Thread-1)");
LockSupport.unpark(thread1); // 1. unpark 사용
//thread1.interrupt(); // 2. interrupt() 사용
}

static class ParkTask implements Runnable {
    @Override
    public void run() {
        log("park 시작");
        LockSupport.park();
        log("park 종료, state: " + Thread.currentThread().getThreadState());
        log("인터럽트 상태: " + Thread.currentThread().isInterrupted());
    }
}
```

실행 상태 그림



park()는 파라미터가 없는데, unpark(thread)는 특정 스레드를 지정해야 할까?

→ 스레드는 park()를 호출해서 스스로 대기 상태로

빠질 수 있으나 대기 상태의 스레드는 자신의 코드를
실행할 수 없기 때문 → 외부에서 도움을 받아야만 한다

이터럽트 사용

Waiting 상태 $\xrightarrow{\text{이터럽트}} \text{runnable}$ 상태로 전환되거나 깨어난다.

↳ unpark() 대신 깨울 수 있음

parkNanos(nanos) : TIMED_WAITING $\xrightarrow{\text{nano초 기간 내면}}$ RUNNABLE 상태로 전환

정확한 미래의
epoch 시점 지정

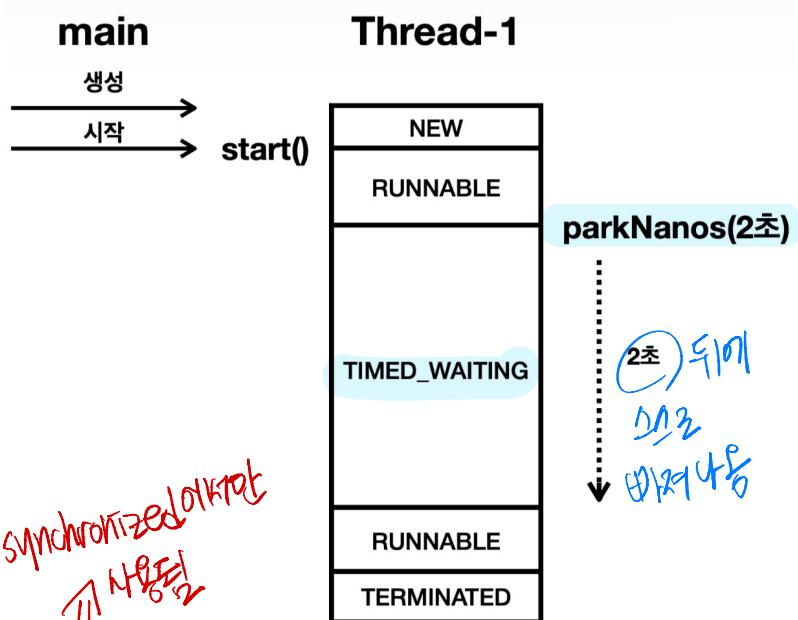
창고로 밀리초 동안 대기하는 바이너드는 없음! \Rightarrow parkUntil(밀리초)는 특정 epoch 시점에 맞춰 깨어나는 메서드

```
static class ParkTask implements Runnable {
    @Override
    public void run() {
        log("park 시작, 2초 대기");
        LockSupport.parkNanos(2000_00000); // parkNanos 사용
        log("park 종료, state: " + Thread.currentThread().getState());
        log("인터럽트 상태: " + Thread.currentThread().isInterrupted());
    }
}
```

✓ nano초 단위에 깨어나도록
unpark() 불필요

↑ 22

실행 상태 그림



BLOCKED와 WAITING(TIMED-WAITING 포함) → 스레드는 대기, CPU 실행 예약됨 포함 X

인터럽트가 걸렸을 때의 차이

- BLOCKED 상태는 인터럽트가 걸려도 상태 편집 불가
- WAITING, TIMED_WAITING 상태는 인터럽트 발생 시 대기 상태 편집과 동시에 RUNNABLE 상태로 편집 가능
- BLOCKED 상태는 synchronized에서 액quires를 위한 대기 상태
- WAITING, TIMED_WAITING 상태는 스레드가 특정 조건이나 시간동안 대기할 때 발생하는 상태
- WAITING: join(), park(), Object.wait() 등 스레드 편집할 때
- TIMED_WAITING: sleep(ms), Object.wait(long timeout), join(long millis), parkNanos(ns) 등 시간 제한이 존재할 때

대기(WAITING) 상태와 시간 대기 상태(TIMED_WAITING)는 서로 짹이 있다.

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

ReentrantLock ~ 자바 1.5 등장

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

※ 주의 ※

이 같은 인스턴스 내부의 monitor lock 이 아니다

→ Lock 인터페이스와 ReentrantLock 이 제공하는 기능

· **void lock()** : 락 획득, 인터럽트에 응답하지 않음 → 인터럽트를 무시하고 락을 기다림(내부적으로 해당 스레드를 WAITING 상태 변경)

↳ 다른 스레드가 이미 락을 획득했다면 락이 풀릴 때까지 현재 스레드는 WAITING 된다.

· **void lockInterruptibly()** : 락 획득 시도 하되, 다른 스레드가 인터럽트 가능

↳ 마찬가지로 다른 스레드가 락을 획득했다면 락을 획득할 때까지 대기

대신에, 대기 중에 인터럽트 발생 시 InterruptedException이 발생하고 락 획득을 포기한다.

· **boolean tryLock()** : 락 획득 시도 하되 즉시 성공 여부 반환

↳ 다른 스레드가 이미 락을 획득했다면 return false, 그렇지 않으면 락을 획득하고 return true

boolean tryLock(long time, TimeUnit unit) : 주어진 시간 동안 락 획득 시도,

↳ 주어진 시간 안에 락 획득하면 return true

주어진 시간 안에 락 획득하지 못하면 return false

대기 중에 인터럽트 발생 시 InterruptedException이 발생하고 락 획득을 포기한다.

void unlock() : 락 해제, 해제하면 대기 중인 스레드 하나가 락 획득 가능

↳ 락을 갖고 있는 스레드에서 호출해야 함 (IllegalMonitorStateException 발생)

Condition newCondition() : Condition 구현 (락과 결합하여 사용, 스레드가 특정 조건을
기다리거나 신호를 받을 수 있도록 한다. (Object 클래스의 wait, notify, notifyAll 메서드와 유사))

synchronized 블록보다 고수준의 동기화 기법 구현 가능 + 특정 시간 만큼 락 대기 시도, 인터럽트 가능한 락 사용

공정성은 어떻게 해결할까?

synchronized의 단점인
무한 대기 및 공정성 문제 해결

↳ Lock 인터페이스의 구현체인 ReentrantLock 클래스로 스레드가 공정하게 락을 얻을 수 있는 모드 제공

사용 예시

```
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();
        }
    }
}
```

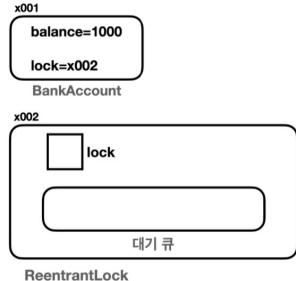


실행 결과 분석

```
ReentrantLock t2 t1
withdraw {
    lock.lock()
try {
    if (balance < amount) {
        return false;
    }
    sleep(1000);
    balance = balance - amount;
} finally {
    lock.unlock();
}
}
```



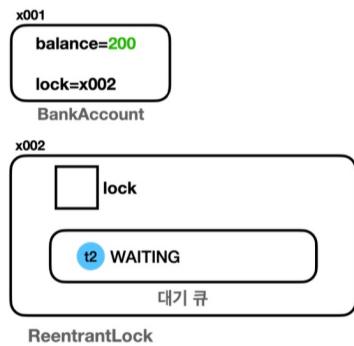
```
ReentrantLock t2
withdraw {
    lock.lock()
try {
    if (balance < amount) { t1 ①
        return false;
    }
    sleep(1000);
    balance = balance - amount;
} finally {
    lock.unlock();
}
}
```



```

ReentrantLock
withdraw {
    lock.lock() t2 WAITING
    try {
        if (balance < amount) {
            return false;
        }
        sleep(1000);
        balance = balance - amount;
    } finally {
        lock.unlock(); t1
    }
}

```

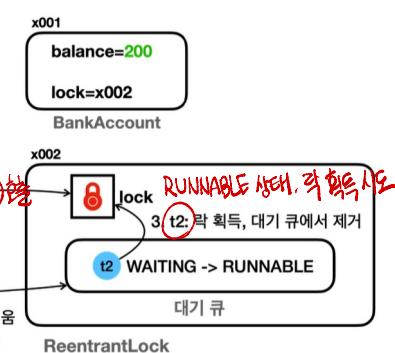


t1은 잠시(2초)의 코드 수행을 끝내고
잔액은 balance = 200 원

```

ReentrantLock
withdraw {
    lock.lock() t2 WAITING
    try {
        if (balance < amount) {
            return false;
        }
        sleep(1000);
        balance = balance - amount;
    } finally {
        lock.unlock(); t1
    }
}

```

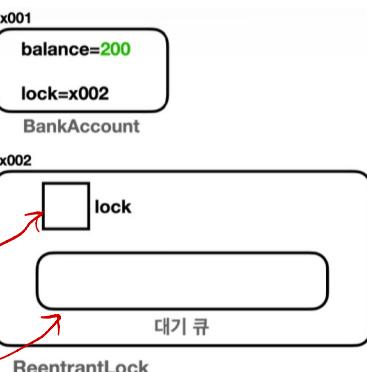


LockSupport.unpark(thread)
내부에서 호출됨

```

ReentrantLock
withdraw {
    lock.lock()
    try {
        if (balance < amount) {
            return false; t2
        }
        sleep(1000);
        balance = balance - amount;
    } finally {
        lock.unlock(); t2
    }
}

```



② 대기 큐에 스레드가 있다면 하나를 깨움

ReentrantLock - 대기 중단

락을 무한정 대기하지 않고 중간에 빠져나올 수 있고 심지어 락을 떠올 수 있다면 기다림 없이 즉시 빠져나올 수 있다.

대기 X

boolean tryLock(); 락 획득 시도 및 즉시 성공 여부 반환 (성공시 true, 실패시 false)

boolean tryLock(long time, ^{TIMED_WAITING} TimeUnit unit) : 지정된 시간동안 락 획득 시도하고 성공시 true

↳ 대기 중에 인터럽트 발생시 InterruptedException 발생 및 락 획득 포기

대기 상태를 빠져나오면 RUNNABLE 이됨