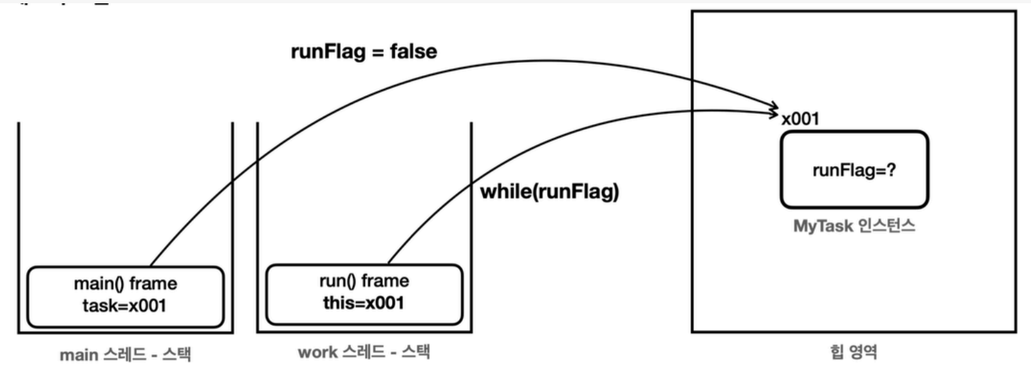
2주차 스터디

목차

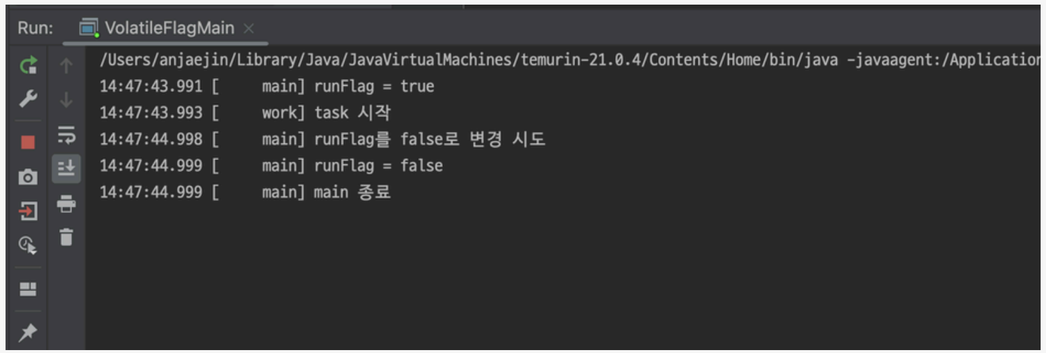
메모리 가시성

```java
public class VolatileFlagMain {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread t = new Thread(task, "work");
        log("runFlag = " + task.runFlag);
        t.start();

        sleep(1000);
        log("runFlag를 false로 변경 시도");
        task.runFlag = false;
        log("runFlag = " + task.runFlag);
        log("main 종료");
    }

    static class MyTask implements Runnable {
        //boolean runFlag = true;
        volatile boolean runFlag = true;

        @Override
        public void run() {
            log("task 시작");
            while (runFlag) {
                // runFlag가 false로 변하면 탈출
            }
            log("task 종료");
        }
    }
}
```

메모리 그림

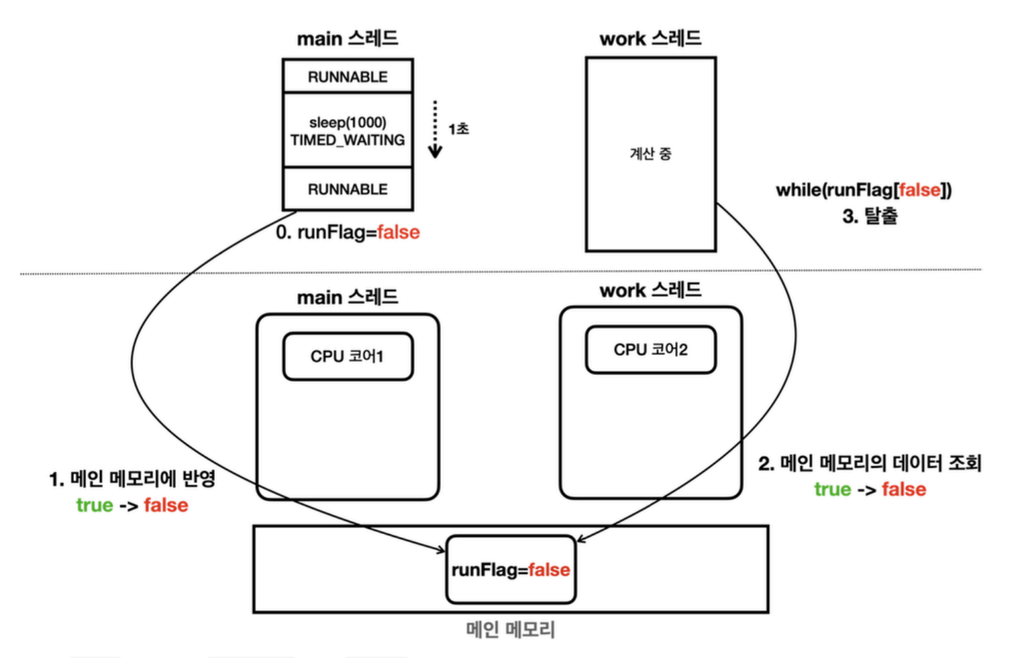


실행 결과

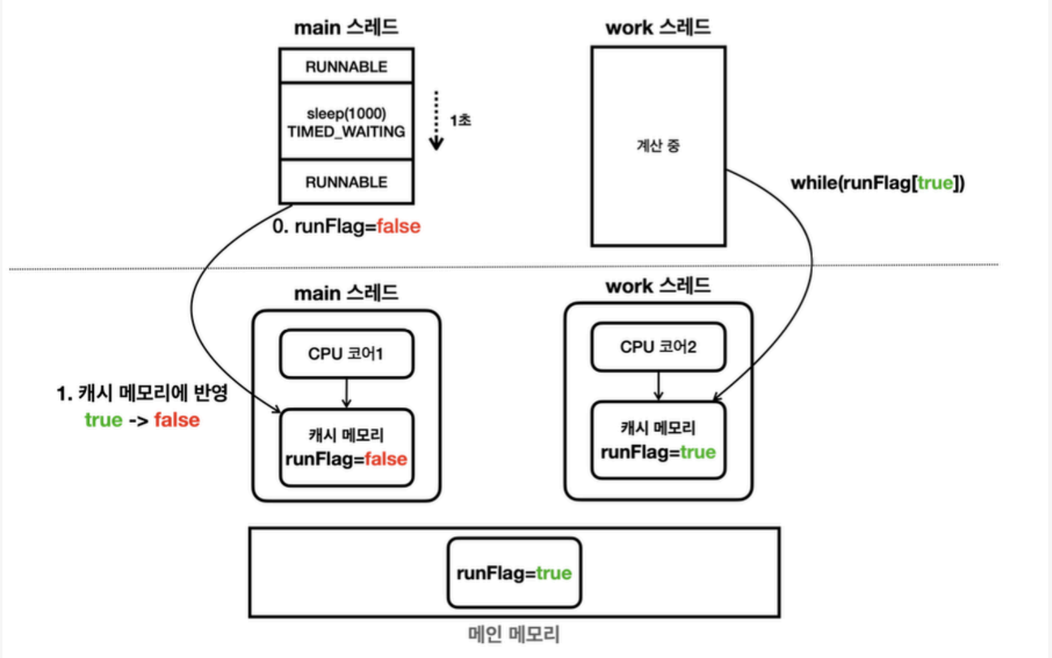


메모리 가시성

일반적으로 생각하는 메모리 접근 방식



실제 메모리의 접근 방식



현대의 CPU 대부분은 코어 단위로 캐시 메모리를 각각 보유

읽기

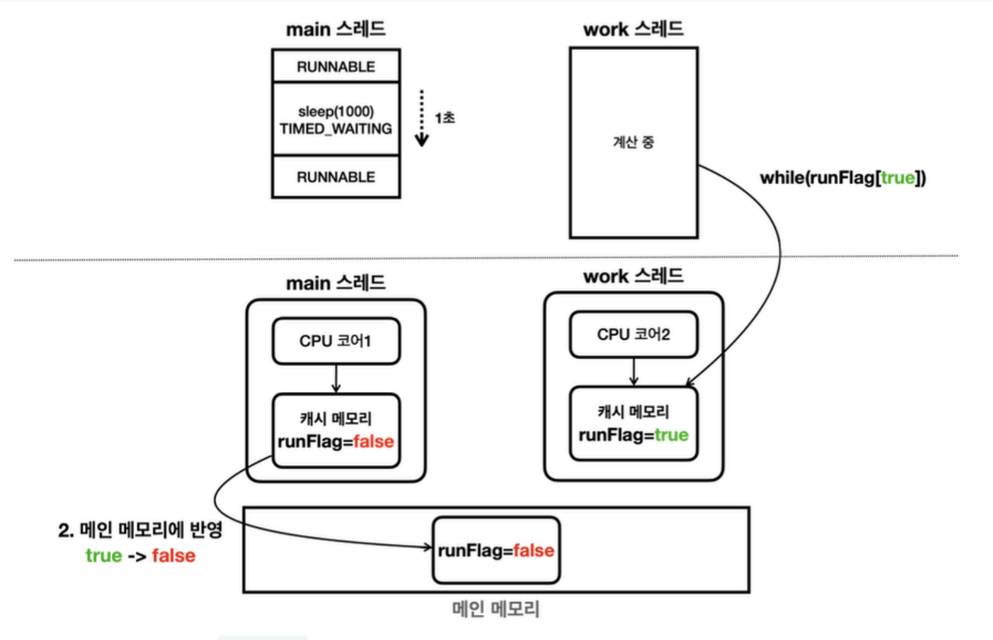
CPU에서 값을 읽을 때 우선 캐시 메모리에 불러오고 이후에는 캐시 메모리에 있는 데이터를 사용

쓰기

CPU에서 값을 쓸 때 캐시 메모리의 값을 변경, 메인 메모리에는 이 값이 즉시 반영X

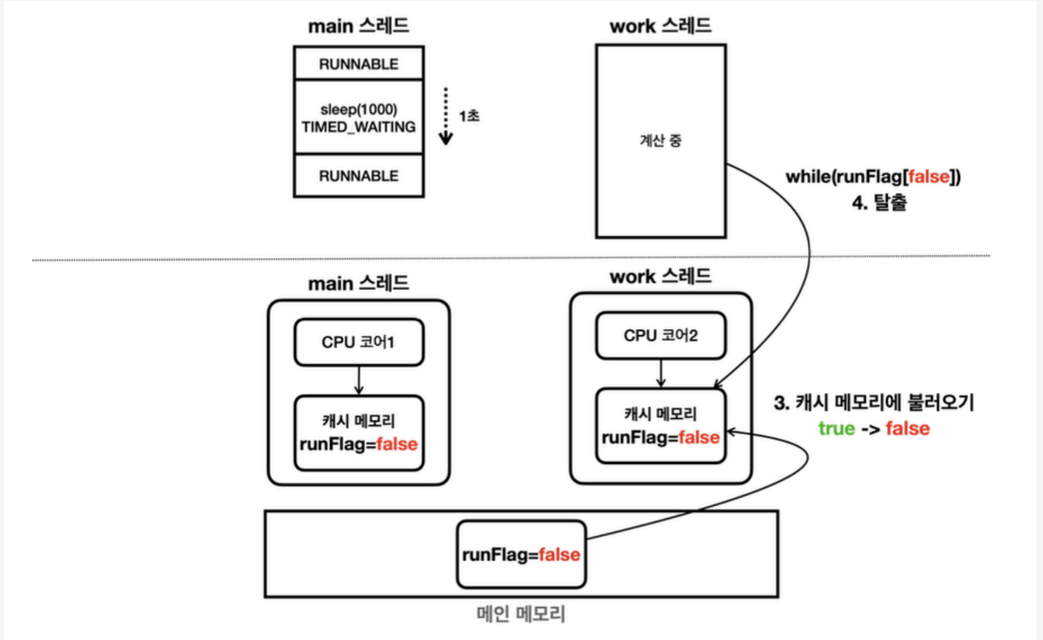
메모리 가시성

캐시 메모리에 있는 runFlag의 값이 언제 메인 메모리에 반영될까?



- CPU 설계 방식과 종류의 따라 달라 알 수 없음
- 컨텍스트 스위칭이 되면서 주로 갱신, 보장 X

메인 메모리에 변경된 runFlag 값이 언제 CPU 코어2의 캐시 메모리에 반영될까?



- CPU 설계 방식과 실행 환경에 따라 다름
- 컨텍스트 스위칭이 되면서 주로 갱신, 보장 X

메모리 가시성

- 멀티스레드 환경에서 한 스레드가 변경한 값이 다른 스레드에서 언제 보이는지에 대한 문제를 메모리 가시성 (memory visibility)이라 한다.
- 이름 그대로 메모리에 변경한 값이 보이는가, 보이지 않는가의 문제이다.
- 그렇다면 한 스레드에서 변경한 값이 다른 스레드에서 즉시 보이게 하려면 어떻게 해야할까?
- 해결방안은 아주 단순하다 성능(캐시)을 약간 포기하는 대신에, 값을 읽을 때, 값을 쓸 때 모두 메인 메모리에 직접 접근하면 된다.
- 자바에서는 volatile이라는 키워드로 이런 기능을 제공

메모리 가시성

```java
public class VolatileFlagMain {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread t = new Thread(task, "work");
        log("runFlag = " + task.runFlag);
        t.start();

        sleep(1000);
        log("runFlag를 false로 변경 시도");
        task.runFlag = false;
        log("runFlag = " + task.runFlag);
        log("main 종료");
    }

    static class MyTask implements Runnable {
        //boolean runFlag = true;
        volatile boolean runFlag = true;

        @Override
        public void run() {
            log("task 시작");
            while (runFlag) {
                // runFlag가 false로 변하면 탈출
            }
            log("task 종료");
        }
    }
}
```

실행 결과

```
23:51:50.769 [     main] runFlag = true
23:51:50.770 [     work] task 시작
23:51:51.776 [     main] runFlag를 false로 변경 시도
23:51:51.776 [     work] task 종료
23:51:51.776 [     main] runFlag = false
23:51:51.776 [     main] main 종료

Process finished with exit code 0
```

- volatile boolean runFlag = true 코드를 사용
- 캐시 메모리를 사용하지 않고, 값을 읽거나 쓸 때 항상 메인 메모리에 직접 접근
- 여러 스레드에서 같은 값을 읽고 써야 한다면 volatile 키워드를 사용
- 단 캐시 메모리를 사용할 때 보다 성능이 느려지는 단점이 있기 때문에 꼭! 필요한 곳에만 사용하는 것이 좋다.

동시성 문제

```java
public class BankAccountV1 implements BankAccount {

    volatile private int balance;

    public BankAccountV1(int initialBalance) {
        this.balance = initialBalance;
    }

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

        log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
        if (balance < amount) {
            log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
            return false;
        }

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

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

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

계좌 출금시 잔고 체크 로직

```java
if (balance < amount) {
    log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
    return false;
}
```

t1, t2 순서로 실행 가정

```
withdraw {
    if (balance < amount) { t2
        return false;
    }
    sleep(1000); t1
    balance = balance - amount;
}
```

t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

- 연산 후 계좌 잔액 -600원

t1, t2 동시에 실행 가정

```
withdraw {
    if (balance < amount) { t1 t2
        return false;
    }
    sleep(1000);
    balance = balance - amount;
}
```

t1: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.
t2: 잔액[1000]이 출금액[800] 보다 많으므로 검증 로직을 통과한다.

- 연산 후 계좌 잔액 200원

임계 영역

```java
public class BankAccountV1 implements BankAccount {

    volatile private int balance;

    public BankAccountV1(int initialBalance) {
        this.balance = initialBalance;
    }

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

        log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
        if (balance < amount) {
            log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
            return false;
        }

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

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

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

공유 자원

- balance(잔액)는 여러 스레드가 동시에 사용하는 공유 자원
- 출금() 메서드를 호출할 때만 잔액이 변경되므로, 다른 스레드가 동시에 출금하면 잔액이 중간에 변동

한 번에 하나의 스레드만 실행

- 출금() 메서드를 한 번에 하나의 스레드만 실행하도록 제한
- 한 스레드가 출금 로직을 끝낼 때까지 다른 스레드는 대기

임계 영역(critical section)

- 여러 스레드가 동시에 접근하면 문제를 일으킬 수 있는 코드 부분을 임계 영역이라고 한다.
- 출금() 로직의 잔액 검증 및 계산 과정이 임계 영역에 해당
- 임계 영역은 한 번에 하나의 스레드만 접근할 수 있도록 보호해야 한다.

임계 영역 보호

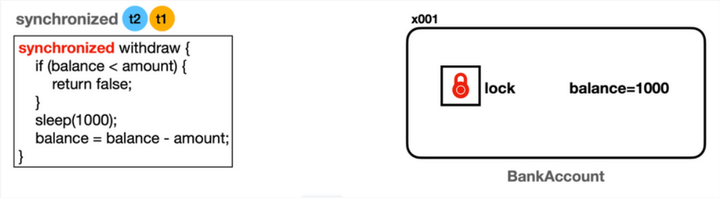
- 자바에서는 synchronized 키워드를 사용하여 간단하게 임계 영역을 보호할 수 있다.

synchronized 메서드

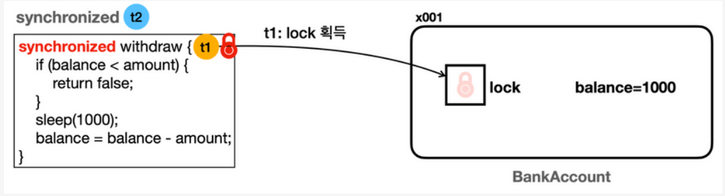
```java
public class BankAccountV2 implements BankAccount {

    private int balance;

    public BankAccountV2(int initialBalance) {
        this.balance = initialBalance;
    }

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

        log("[검증 시작] 출금액: " + amount + ", 잔액: " + balance);
        if (balance < amount) {
            log("[검증 실패] 출금액: " + amount + ", 잔액: " + balance);
            return false;
        }

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

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

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

- synchronized 키워드 사용 시 한 번에 하나의 스레드만 실행할 수 있는 코드 구간을 만들 수 있음
- withdraw(), getBalance() 코드에 synchronized 키워드 추가
  - withdraw(), getBalance() 메서드는 한 번에 하나의 스레드만 실행할 수 있다.

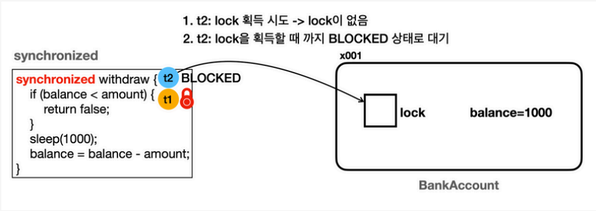
synchronized 분석



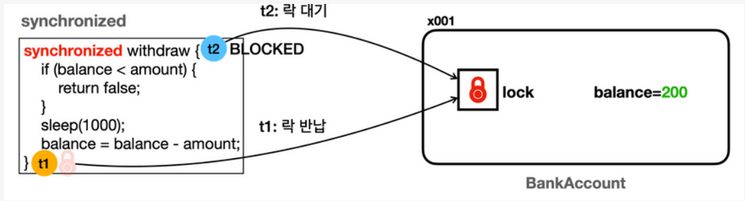
- 모든 객체(인스턴스)는 내부에 자신만의 락(lock) 존재
  - 모니터 락(monitor lock)이라도고 부른다.
- 스레드가 synchronized 키워드가 있는 메서드에 진입하려면 반드시 해당 인스턴스의 락이 필요



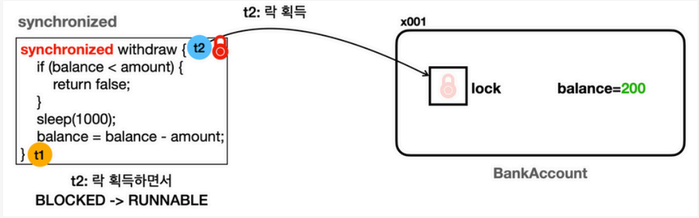
- 스레드 t1이 먼저 synchronized 키워드가 있는 withdraw() 메서드를 호출
- 락이 있으므로 스레드 t1은 BankAccount(x001) 인스턴스에 있는 락을 획득



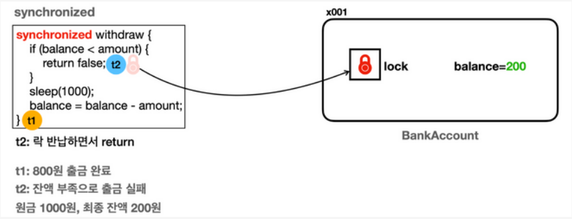
- 스레드 t2가 withdraw() 메서드 호출하면 BankAccount(x001) 인스턴스에 있는 락 획득을 시도
- 하지만 락이 없어 t2 스레드는 락을 획득할 때 까지 BLOCKED 상태로 대기
- t2 스레드의 상태는 RUNNABLE -> BLOCKED 상태로 변하고, 락을 획득할 때 까지 무한정 대기
- BLOCKED 상태가 되면 락을 다시 획득하기 전까지는 계속 대기하고, CPU 실행 스케줄링에 들어가지 않는다.



- t1 : 잔액 1000원에서 800원을 출금하고 계산 결과인 200원을 잔액(balance)에 반영
- t1 : 메서드 호출이 끝나면 락을 반납



- t2 : 인스턴스에 락이 반납되면 락 획득을 대기하는 스레드는 자동으로 락을 획득
- 락을 획득한 스레드는 BLOCKED -> RUNNABLE 상태가되고,다시코드를 실행



- t2 : 출금을 위한 검증 로직을 수행, 조건을 만족하지 않으므로 false를 반환
- t2 : 락을 반납하면서 return

synchronized 동기화 정리

메서드 동기화

- 메서드를 synchronized로 선언해서, 메서드에 접근하는 스레드가 하나뿐이도록 보장

```
public synchronized void synchronizedMethod() {
            // 코드
}
```

블록 동기화

- 코드 블록을 synchronized로 감싸서, 동기화를 구현할 수 있다.

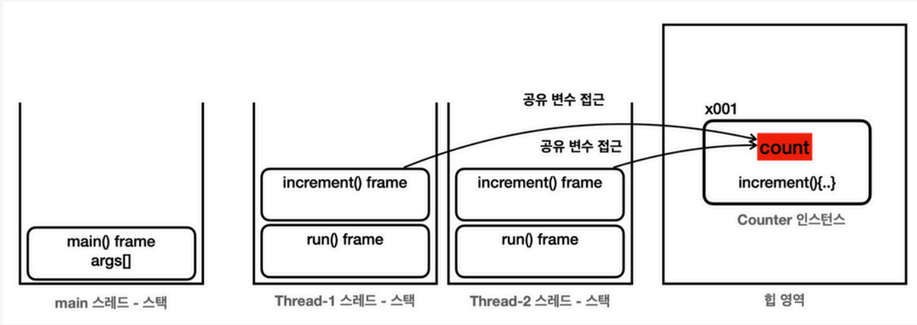
```
public void method() {
    synchronized(this) {
        // 동기화된 코드
    }
}
```

이런 동기화를 사용하면 다음 문제들을 해결

- 경합 조건(Race condition): 두 개 이상의 스레드가 경쟁적으로 동일한 자원을 수정할 때 발생하는 문제
- 데이터 일관성: 여러 스레드가 동시에 읽고 쓰는 데이터의 일관성을 유지

문제1 - 공유 자원

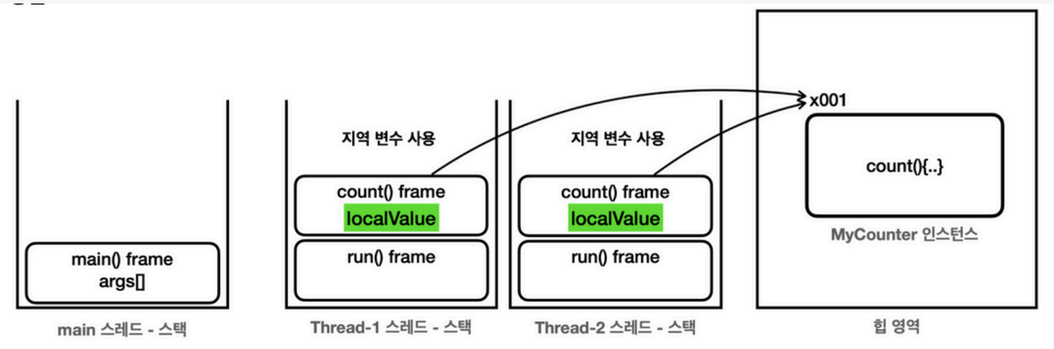
```java
public class SyncTest1Main {

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Runnable task = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    counter.increment();
                }
            }
        };

        Thread thread1 = new Thread(task); // 10000번 호출
        Thread thread2 = new Thread(task); // 10000번 호출

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("결과: " + counter.getCount());
    }

    static class Counter {
        private int count = 0;

        public synchronized void increment() {
            count = count + 1;
        }

        public synchronized int getCount() {
            log("getter 호출");
            return count;
        }
    }
}
```



- 여러 스레드가 동시에 실행되서 공유 자원인 count에 동시에 접근하는 상황
- 각 스레드가 연산한 결과를 count에 동시에 저장하는 문제가 발생
- 따라서 synchronized 키워드를 사용해서 안전한 임계 영역 생성

문제2 - 지역 변수의 공유

```java
public class SyncTest2Main {
    public static void main(String[] args) throws InterruptedException {
        MyCounter myCounter = new MyCounter();

        Runnable task = new Runnable() {
            @Override
            public void run() {
                myCounter.count();
            }
        };

        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");

        thread1.start();
        thread2.start();
    }

    static class MyCounter {

        public void count() {
            int localValue = 0;
            for (int i = 0; i < 1000; i++) {
                localValue = localValue + 1;
            }
            log("결과: " + localValue);
        }
    }
}
```



- 스택 영역은 각각의 스레드가 가지는 별도의 메모리 공간이다.
- 이 메모리 공간은 다른 스레드와 공유하지 않는다
- 지역 변수는 스레드의 개별 저장 공간인 스택 영역에 생성된다.
- 따라서 지역 변수는 다른 스레드와 공유되지 않는다.
- 이런 이유로 지역 변수는 동기화에 대한 걱정을 하지 않아도 된다.

문제3 - final 필드

```
class Immutable {

    private final int value;

    public Immutable(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

- 여러 스레드가 접근 가능한 공유 자원이라도 그 값을 아무도 변경할 수 없다면 문제 되지 않는다
- 필드에 final이 붙으면 어떤 스레드도 값을 변경할 수 없다
- 따라서 멀티스레드 상황에 문제 없는 안전한 공유 자원이 된다.

synchronized 장, 단점

synchronized 장점

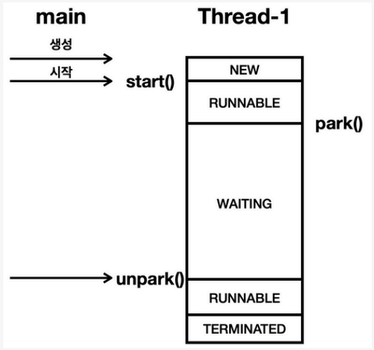
- 프로그래밍 언어에 문법으로 제공
- 아주 편리한 사용
- 자동 잠금 해제: synchronized 메서드나 블록이 완료되면 자동으로 락을 대기중인 다른 스레드의 잠금이 해제 된다.
    - 개발자가 직접 특정 스레드를 깨우도록 관리해야 한다면, 매우 어렵고 번거로울 것이다.

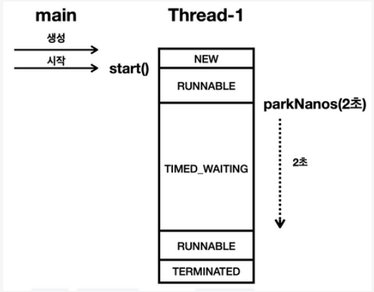
synchronized 단점

- 무한 대기: BLOCKED 상태의 스레드는 락이 풀릴 때 까지 무한 대기한다.
    - 특정 시간까지만 대기하는 타임아웃X
    - 중간에 인터럽트X
- 공정성: 락이 돌아왔을 때 `BLOCKED` 상태의 여러 스레드 중에 어떤 스레드가 락을 획득할 지 알 수 없다.
    - 최악의 경우 특정 스레드가 너무 오랜기간 락을 획득하지 못할 수 있다.

LockSupport

```java
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 준다.
        sleep(100);
        log("Thread-1 state: " + thread1.getState());

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

    static class ParkTest implements Runnable {

        @Override
        public void run() {
            log("park 시작");
            LockSupport.park();
//            LockSupport.parkNanos(2000_000000);  // parkNanos 사용
            log("park 종료, state: " + Thread.currentThread().getState());
            log("인터럽트 상태: " + Thread.currentThread().isInterrupted());
        }
    }
}
```





LockSupport 기능

- park(): 스레드를 WAITING 상태로 변경
- parkNanos(nanos) : 스레드를 나노초 동안만 TIMED_WAITING 상태로 변경
  - 지정한 나노초가 지나면 TIMED_WAITING -> RUNNABLE 상태로 변경
- unpark(thread) : WAITING 상태의 대상 스레드를 RUNNABLE 상태로 변경

인터럽트 사용

- WAITING 상태의 스레드에 인터럽트가 발생하면 WAITING 상태에서 RUNNABLE 상태로 변경

BLOCKED vs WAITING

인터럽트

- BLOCKED 상태는 인터럽트가 걸려도 대기 상태를 빠져나오지 못하고 여전히 BLOCKED 상태
- WAITING, TIMED_WAITING 상태는 인터럽트가 걸리면 대기 상태를 빠져나오고 RUNNABLE 상태로 변한다.

용도

- BLOCKED 상태는 자바의 synchronized 에서 락을 획득하기 위해 대기할 때 사용된다.
- WAITING, TIMED_WAITING 상태는 스레드가 특정 조건이나 시간 동안 대기할 때 발생하는 상태이다.

WAITING 상태

- Thread.join()
- LockSupport.park()
- Object.wait()

TIMED_WAITING 상태

- Thread.sleep(ms)
- Object.wait(long timeout)`
- Thread.join(long millis)
- LockSupport.parkNanos(ns)

ReentrantLock

```
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

void lock()

- 락을 획득
- 다른 스레드가 이미 락을 획득 시, 락이 풀릴 때까지 현재 스레드는 계속 대기(WAITING)
- 인터럽트에 응답하지 않는다.

void lockInterruptibly()

- 락 획득을 시도하되, 다른 스레드가 인터럽트할 수 있도록 한다.
- 만약 다른 스레드가 이미 락 획득 시, 현재 스레드는 락을 획득할 때까지 대기
- 대기 중에 인터럽트가 발생 시 `InterruptedException` 발생 후 락 획득 포기

boolean tryLock()

- 락 획득을 시도하고, 즉시 성공 여부를 반환
- 다른 스레드가 이미 락을 획득했다면 false 반환
- 그렇지 않으면 락을 획득하고 true 반환

boolean tryLock(long time, TimeUnit unit)

- 주어진 시간 안에 락 획득 시 true 반환
- 주어진 시간이 지나도 락을 획득하지 못한 경우 false 반환
- 대기 중 인터럽트가 발생 시 nterruptedException 발생 후 락 획득 포기

공정성

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

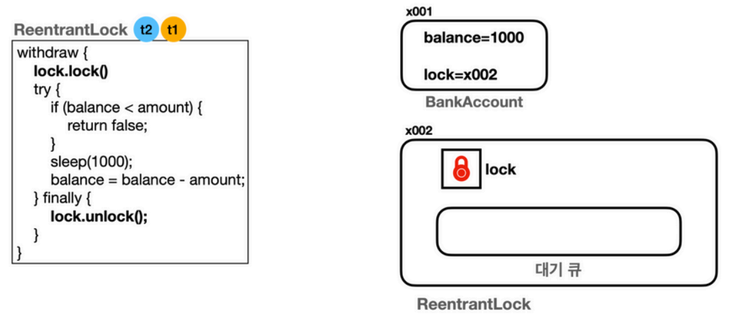
비공정 모드 (Non-fair mode)

- 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.

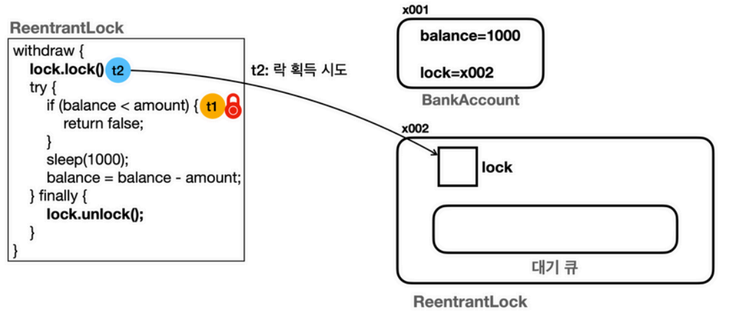
공정 모드 (Fair mode)

- 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.

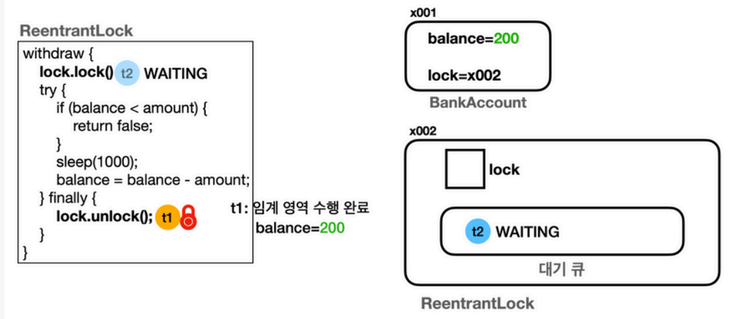
ReentrantLock - 실행 흐름



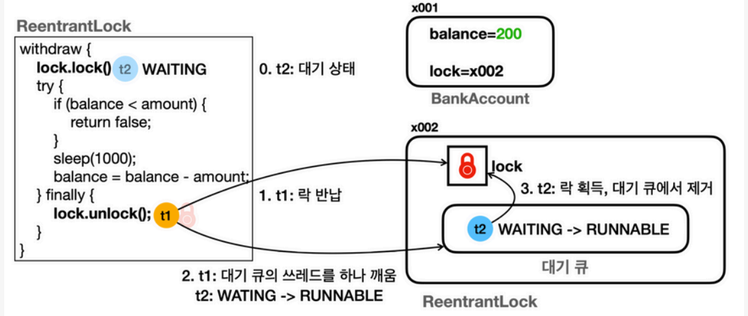
- ReenterantLock 내부에는 락과 락을 얻지 못해 대기하는 스레드를 관리하는 대기 큐가 존재
- 여기서 이야기하는 락은 객체 내부에 있는 모니터 락이 아닌 ReentrantLock 이 제공하는 기능



- t1이 락을 획득 후 임계영역에 진입해 코드 실행
- t2가 ReenterantLock에 있는 락의 획득을 시도 하지만 락이 없다.



- t2는 락을 획득하지 못해 WAITING 상태가 되고, 대기 큐에서 관리된다.
  - LockSupoort.park()가 내부에서 호출된다.
- tryLock(long time, TimeUnit unit)와 같은 시간 대기 기능을 사용하면 TIMED_WAITING 이 되고, 대기 큐에서 관리된다.



- t1이 임계 영역을 수행하고 lock.unlock()을 호출해 락을 반납한다.
- LockSupoort.unpark(thread)가 내부에서 호출 되면서 대기 큐의 스레드를 하나 깨운다.
- 대기 큐에 존재하던 스레드는 RUNNABLE 상태가 되면서 락 획득을 시도한다.
- 락을 획득하면 대기 큐에서 제거되고 락을 획득하지 못하면 대기 큐에 유지
  - 참고로 락 획득을 시도하는 잠깐 사이에 새로운 스레드가 락을 먼저 가져갈 수 있다.
  - 공정 모드의 경우 대기 큐에 먼저 대기한 스레드가 먼저 락을 가져간다.

감사합니다.