메모리 가시성

소유자	종수 김
태그	

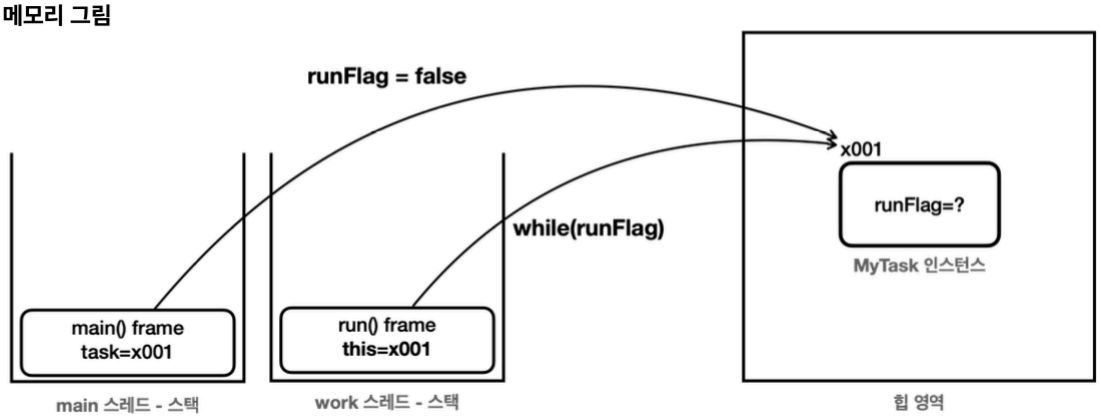
volatile, 메모리 가시성1

```
package thread.volatile1;

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

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("starting");
            }
```

```
            log("task 종료");
        }
    }
}
```

- `work` 스레드는 `MyTask` 를 실행한다.
- 여기에는 `runFlag` 를 체크하는 무한 루프가 있다.
- `runFlag` 값이 `false` 가 되면 무한 루프를 탈출하며 작업을 종료한다.
- 이후에 `main` 스레드가 `runFlag` 의 값을 `false` 로 변경한다.
- `runFlag` 의 값이 `false` 가 되었으므로 `work` 스레드는 무한 루프를 탈출하며, 작업을 종료한다.

메모리 그림



- main 쓰레드, work 쓰레드 모두 MyTask 인스턴스의 runFlag를 사용
- 이 값을 false로 변경하면, work 쓰레드의 작업을 종료

프로그램은 아주 단순하다.

- `main` 스레드는 새로운 스레드인 `work` 스레드를 생성하고 작업을 시킨다.
- `work` 스레드는 `run()` 메서드를 실행하면서 `while(runFlag)`가 `true`인 동안 계속 작업을 한다. 만약 `runFlag`가 `false`로 변경되면 반복문을 빠져나오면서 `"task 종료"`를 출력하고 작업을 종료한다.
- `main` 스레드는 `sleep()`을 통해 1초간 쉰 다음에 `runFlag`를 `false`로 설정한다.
- `work` 스레드는 `run()` 메서드를 실행하면서 `while(runFlag)`를 체크하는데, 이제 `runFlag`가 `false`가 되었으므로 `"task 종료"`를 출력하고 작업을 종료해야 한다.

기대하는 실행 결과

```
15:39:59.830 [     main] runFlag = true
15:39:59.830 [     work] task 시작
15:40:00.837 [     main] runFlag를 false로 변경 시도
15:40:00.838 [     main] runFlag = false
15:40:00.838 [     work] task 종료
15:40:00.838 [     main] main 종료
```

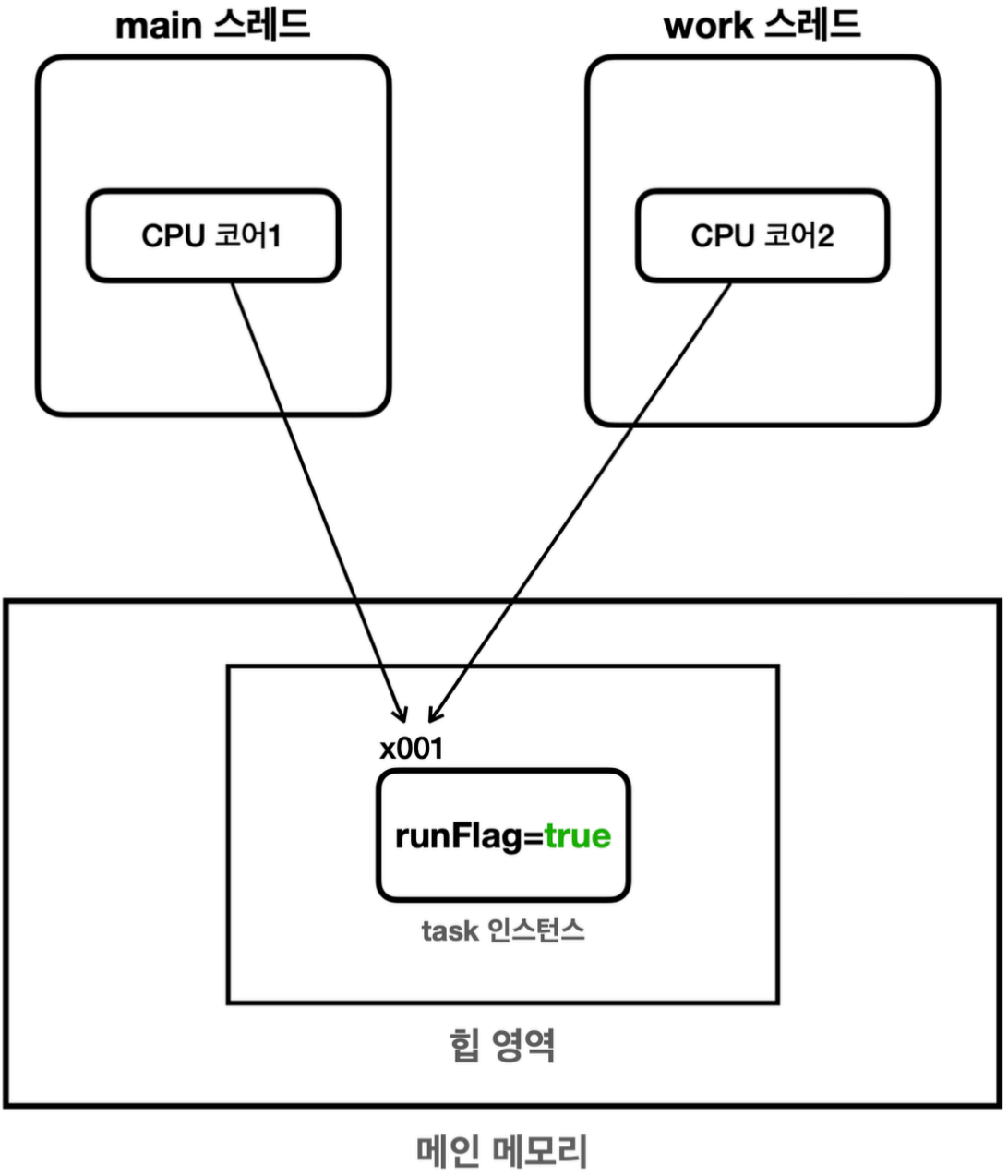
실제 실행 결과는 task가 종료되지 않음, 자바 프로그램 또한 종료되지 않음.

why ?

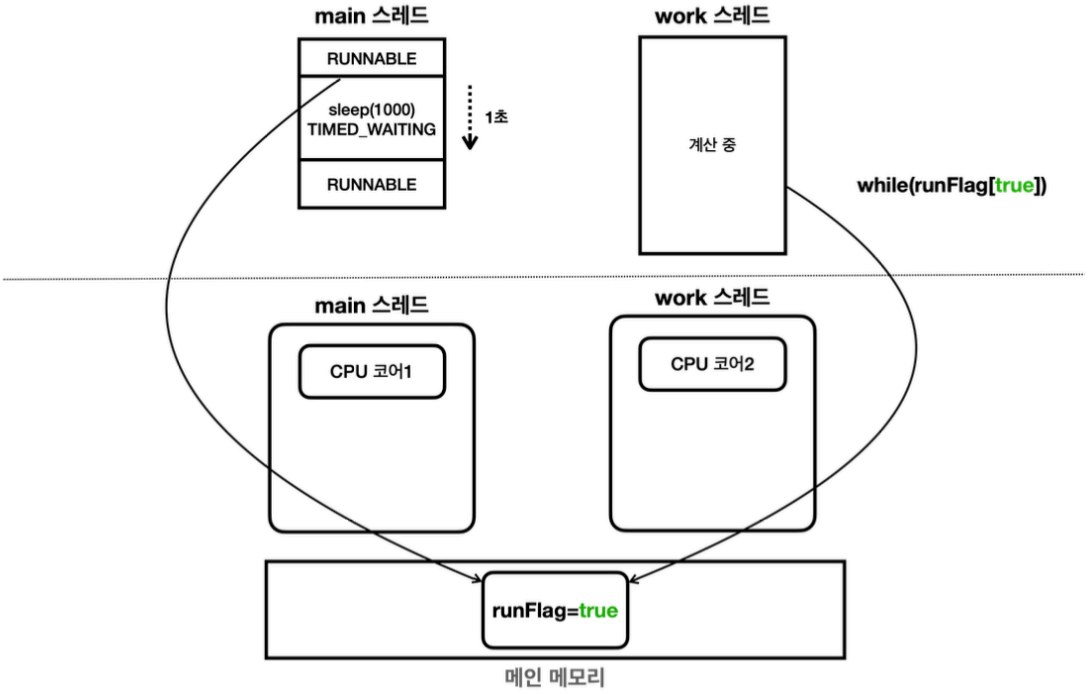
volatile, 메모리 가시성2

메모리 가시성문제.

일반적으로 생각하는 메모리 접근 방식

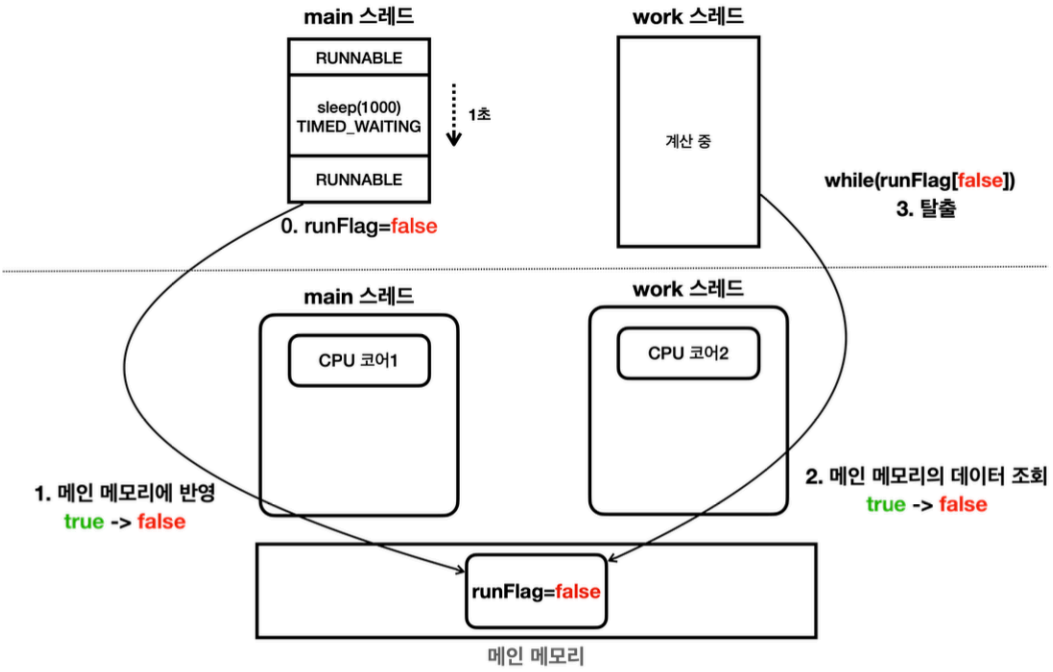


- main 쓰레드와 work 쓰레드는 각각의 CPU 코어에서 실행됨.
- CPU 코어가 한개라면 번갈아가면서 실행될 것.



> 점선 위쪽은 쓰레드의 실행 흐름, 점선 아래쪽은 하드웨어를 나타냄.

- 자바 프로그램을 실행하고 main 쓰레드와 work 쓰레드는 모두 메인 메모리의 runFlag 값을 읽음.
- 프로그램의 시작 시점에는 runFlag를 변경하지 않기 때문에 모든 쓰레드에서 true의 값을 읽음.
    - runFlag = true;
- work 쓰레드는 while(runFlag)이므로 계속 반복해서 수행

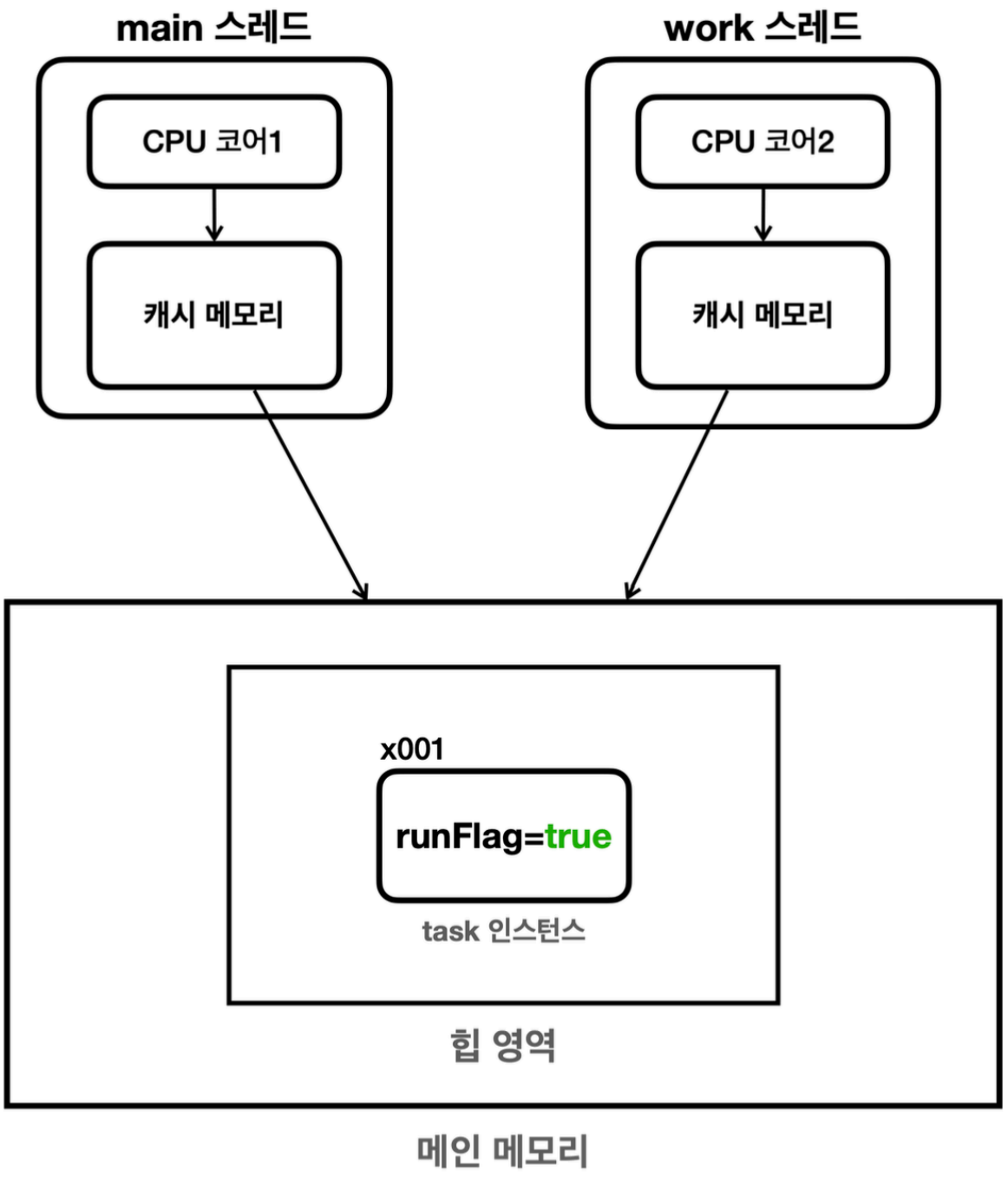


0. main 쓰레드는 runFlag 값을 false로 설정한다.
1. 이 때 메인 메모리의 runFlag = false
2. work 쓰레드의 while(runFlag)를 실행할 때 runFlag의 데이터를 메인 메모리에서 확인한다.
3. runFlag의 값이 false이므로 while문을 탈출하고 "task 종료"를 출력한다.

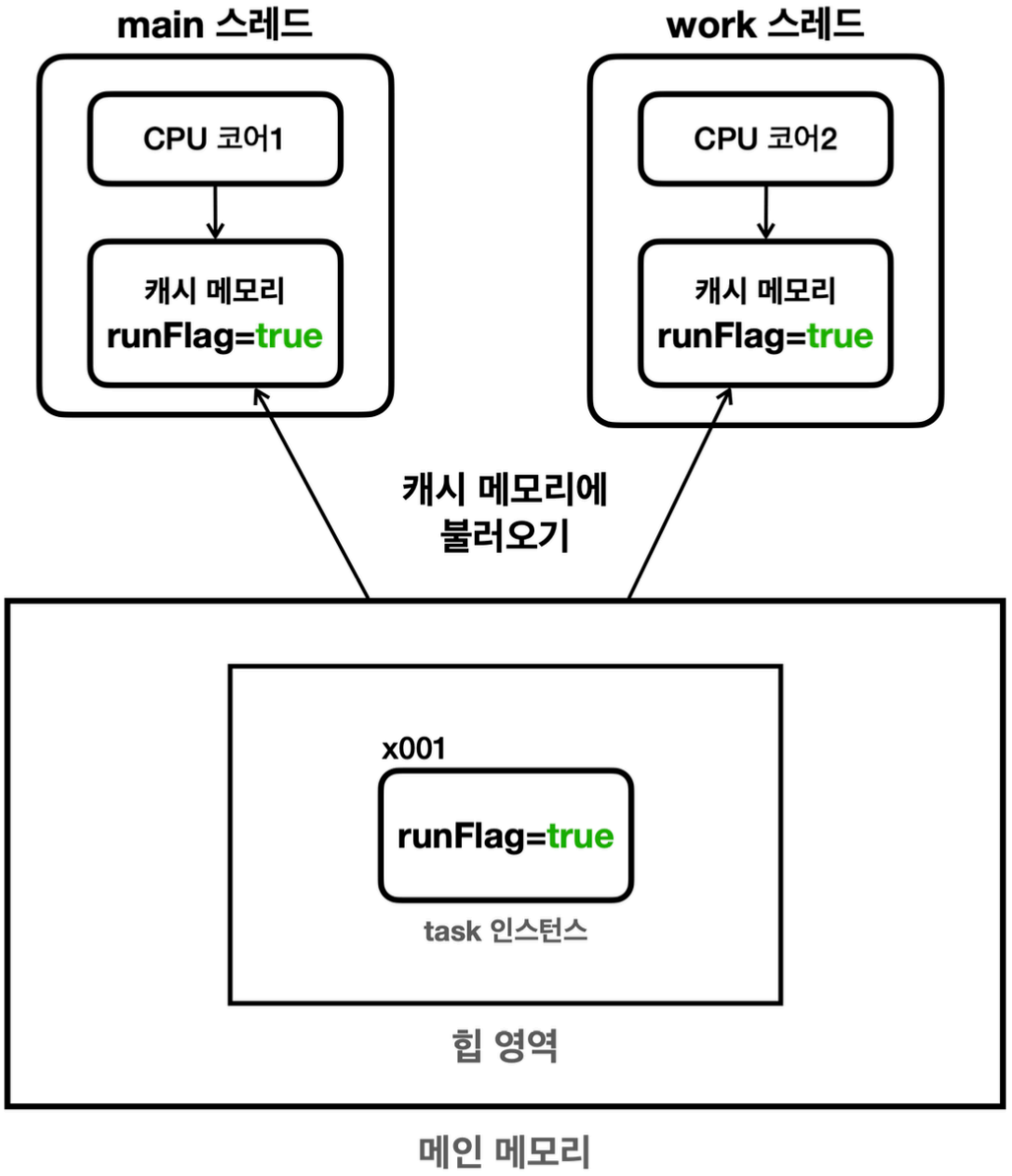
BUT 이렇게 동작하지 않음.

실제 메모리의 접근 방식

> CPU는 처리 성능을 개선하기 위해 중간에 '캐시 메모리'라는 것을 사용

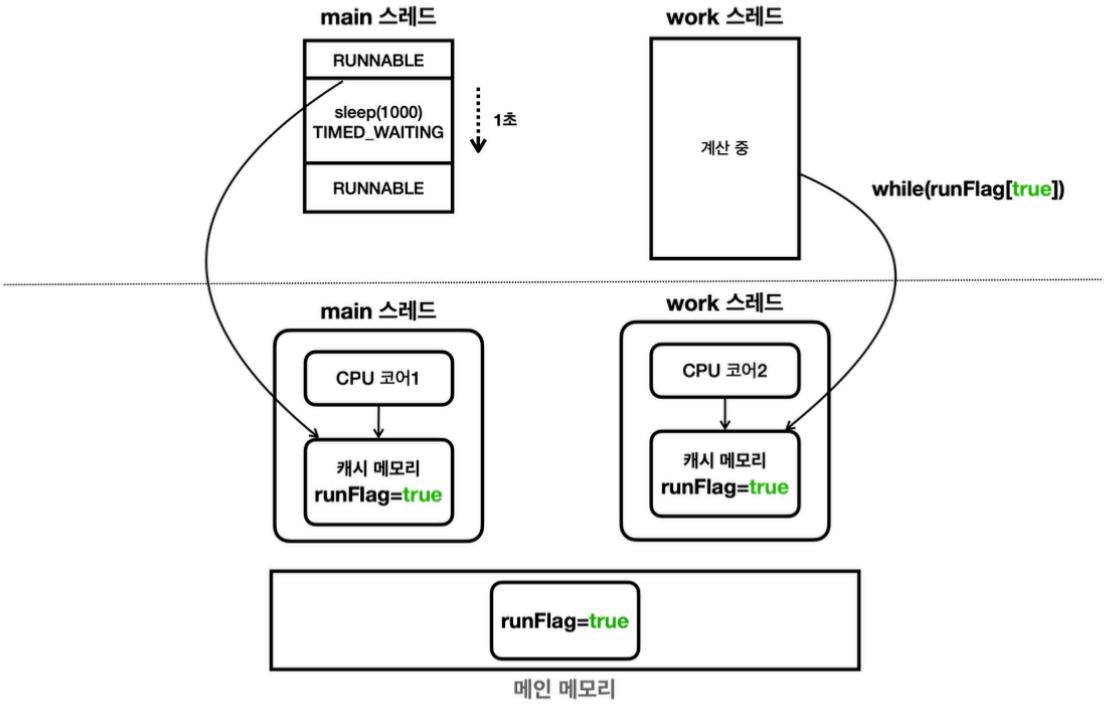


- 메인 메모리는 CPU입장에서는 거리도 멀고, 속도도 상대적으로 느림. 대신에 상대적으로 가격이 저렴하여 큰 용량을 쉽게 구성 가능.
- CPU 연산은 매우 빠르기 때문에 CPU 연산의 빠른 성능을 따라가려면, CPU 가까이에 아주 빠른 메모리가 필요한데 이것이 바로 '캐시 메모리', 캐시 메모리는 CPU와 가까이 붙어있으며, 속도도 매우빠름. 하지만 상대적으로 가격이 비쌈.
- 현대의 CPU 대부분은 코어 단위로 캐시 메모리를 각각 보유하고 있음.
    - 참고로 여러 코어가 공유하는 캐시 메모리도 있음.



- 각 쓰레드가 runFlag의 값을 사용하면 CPU는 이 값을 효율적으로 처리하기 위해 먼저 runFlag를 캐시 메모리에 불러옴.
- 그리고 이후에는 캐시 메모리에 있는 runFlag를 사용.

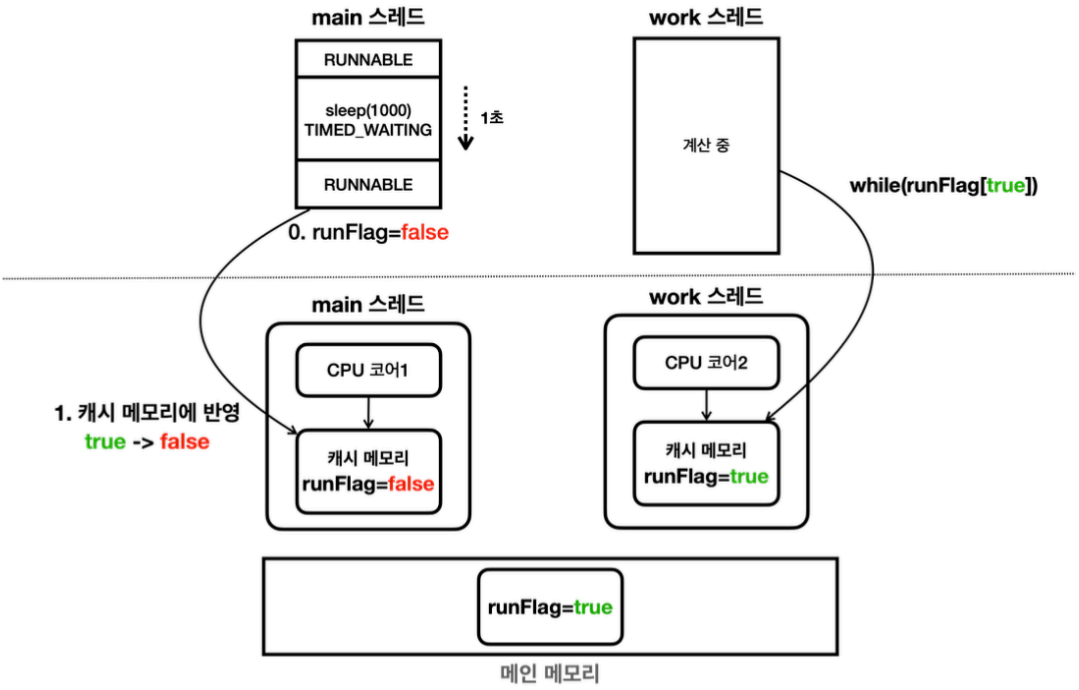
초기 상태



> 점선 위쪽은 쓰레드의 실행 흐름, 점선 아래쪽은 하드웨어를 나타냄.

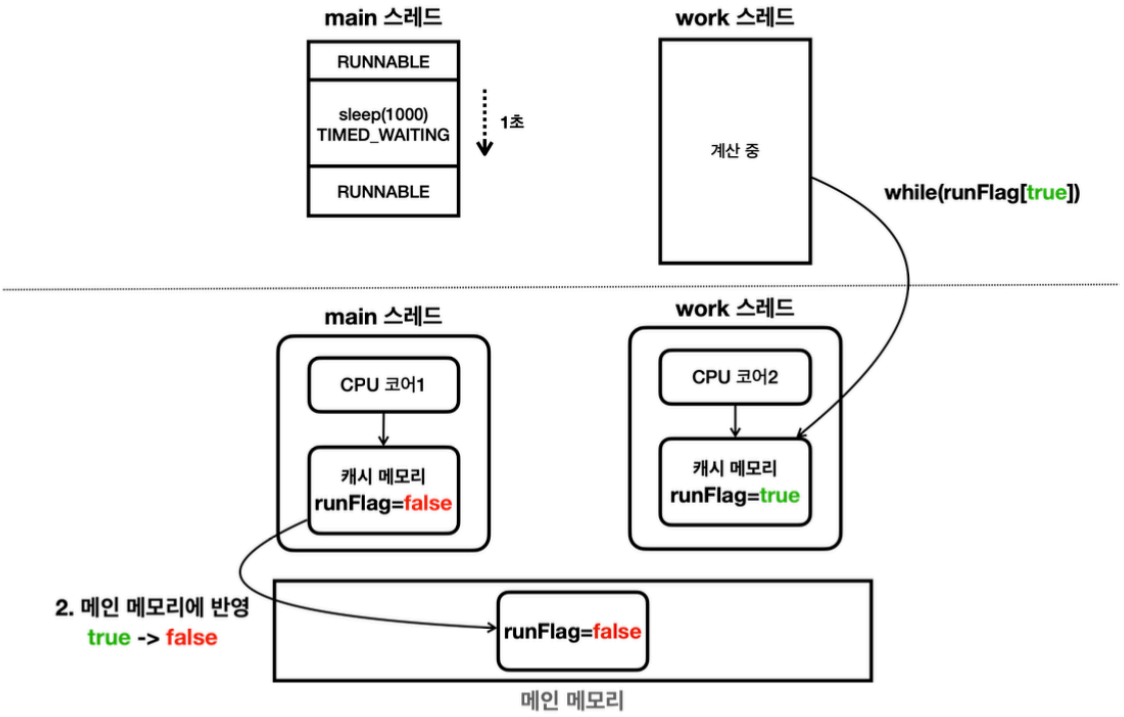
- 자바 프로그램을 실행하고 main 쓰레드와, work 쓰레드는 모두 runFlag의 값을 읽는다
- CPU는 이 값을 효율적으로 처리하기 위해 먼저 캐시 메모리를 불러온다.
- main 쓰레드와, work 쓰레드가 사용하는 runFlag가 각각의 캐시 메모리에 보관된다.
- 프로그램의 시작 시점에는 `runFlag` 를 변경하지 않기 때문에 모든 스레드에서 `true` 의 값을 읽는다.
    - 참고로 `runFlag` 의 초기값은 `true` 이다.
- `work` 스레드의 경우 `while(runFlag[true])` 가 만족하기 때문에 while문을 계속 반복해서 수행한다.

runFlag = false 변경 후

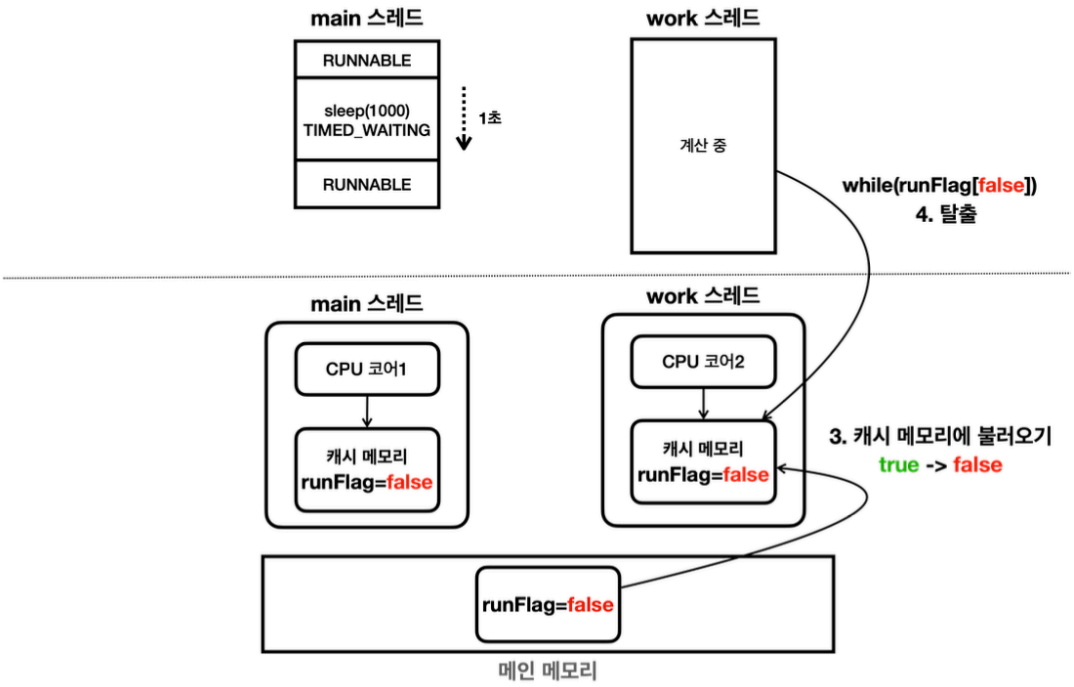


0. main 쓰레드는 runFlag를 false로 설정한다.
1. 이 때 캐시 메모리의 runFlag는 false로 설정

- 핵심은, 캐시 메모리의 runFlag값만 변한다는 것, 메인 메모리에 이 값이 즉시 반영되지 않는다.
    - work 쓰레드가 아닌, main 쓰레드의 캐시 메모리만 변경된 것.
    - main 쓰레드의 캐시 메모리가 언제 메인 메모리에 반영될 지,
      work 쓰레드가 언제 메인 메모리와 캐시 메모리를 싱크할지,
    - 동기화하는 방법도, 언제 동기화 되는지도 '**모름**' 그니까 보장해주는 volatile 키워드가 있는 것.



- 캐시 메모리에 있는 runFlag의 값이 언제 메인 메모리에 반영되는가 ? ⇒ 모름.
- 이 부분에 대한 정답은 '알 수 없음' CPU의 설계 방식과 종류의 따라 다르며, 극단적으로는 평생 안될 수도 있음.
- 캐시 메모리에 있는 값이 메인 메모리에 반영되었다면 ?
  - 끝이 아님, work 쓰레드가 다시 자신의 캐시 메모리로 메인 메모리를 불러와야 하기 때문.
  - 이것도 알 수 없음.



- 메인 메모리에 변경된 `runFlag` 값이 언제 CPU 코어2의 캐시 메모리에 반영될까?
  - 이 부분에 대한 정답도 "알 수 없다"이다.
- CPU 설계 방식과 종류의 따라 다르다. 극단적으로 보면 평생 반영되지 않을 수도 있다!
- 언젠가 CPU 코어2의 캐시 메모리에 `runFlag` 값을 불러오게 되면 `work` 스레드가 확인하는 `runFlag` 의 값이 `false` 가 되므로 while문을 탈출하고, `"task 종료"` 를 출력한다.

> 주로 컨텍스트 스위칭이 될 때, 캐시 메모리도 함께 갱신되는데, 이 부분도 환경에 따라 달라질 수 있다.
> 예를 들어 `Thread.sleep()` 이나 콘솔에 내용을 출력할 때 스레드가 잠시 쉬는데, 이럴 때 컨텍스트 스위칭이 되면서 주로 갱신된다.
> **하지만 이것이 갱신을 보장하는 것은 아니다.**

메모리 가시성 (memory visibility)

이처럼 멀티쓰레드 환경에서 한 쓰레드가 변경한 값이, 다른 쓰레드에서 언제 보이는지에 대한 문제를 '**메모리 가시성**'이라 함. 이름 그대로 **메모리에서 보이는가 ? 보이지 않는가 ?**

그렇다면 한 쓰레드에서 변경한 값이 다른 쓰레드에도 반영되게 하려면 ?

volatile, 메모리 가시성3

캐시 메모리를 사용하면 CPU 처리 성능을 개선할 수 있음

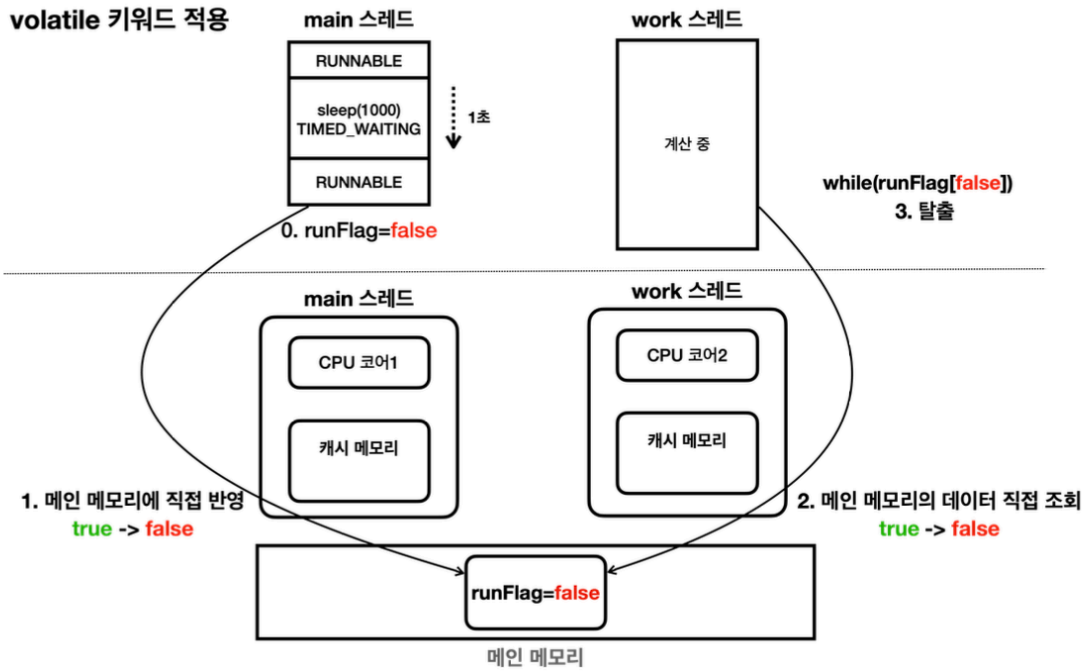
> CPU 캐시 전략 - CPU 캐시와 메인 메모리 간의 쓰기 동기화 방법
> write-back : 캐시에 먼저 쓰고, 나중에 메인 메모리에 반영 (기본 값)
> ㄴ 장점 : 메인 메모리에 접근 최소화 → 성능 좋음
> ㄴ 단점 : 다른 쓰레드에서 최신 값 못봄 → 메모리 가시성 문제
>
> volatile 사용 시
> write-through : 캐시에 쓰는 동시에 메인 메모리에도 바로 반영
> ㄴ 장점 : 항상 메인 메모리와 동기화 → 메모리 가시성 문제 X
> ㄴ 단점 : 메인 메모리에 접근 많아지므로 성능 저하

하지만 때로 성능 향상 보다는 여러 스레드에서 같은 시점에 정확히 같은 데이터를 보는 것이 더 중요할 수 있다.
해결방안은 아주 단순하다 성능을 약간 포기하는 대신에, 값을 읽을 때, 값을 쓸 때 모두 메인 메모리에 직접 접근하면된다.

volatile

```
package thread.volatile1;

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

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("starting");
            }
            log("task 종료");
        }
    }
}

21:42:57.976 [     main] runFlag = true
21:42:57.978 [     work] task 시작
21:42:58.982 [     main] runFlag를 false로 변경 시도
21:42:58.982 [     work] task 종료
21:42:58.982 [     main] main 종료
```



여러 스레드에서 같은 값을 읽고 써야 한다면 `volatile` 키워드를 사용하면 된다. 단 캐시 메모리를 사용할 때 보다 성능이 느려지는 단점이 있기 때문에 꼭! 필요한 곳에만 사용하는 것이 좋다.

- volatile 키워드 사용 시, 메인 메모리에 직접 반영.
  - volatile 변수에 접근 시, JVM이 항상 메인 메모리에서 값을 읽도록 강제함.
  - 따라서, 읽는 쪽 쓰레드 또한 자신의 캐시가 아닌 메인 메모리 값을 확인.
  - 즉, 다른 쓰레드가 읽을 때 메인 메모리를 참조하도록 '보장' 하는 것.
- System.out.println(), log.info()등 쓰레드가 대기 시간으로 들어가서 '컨텍스트 스위칭'이 일어나면, 메인 메모리에 동기화 하기 때문에, volatile 없이도 정상작동 할 '수도' 있음.
  - 하지만, 이건 로직에 따라 변경되는 부분이고 확실히 100% 보장할 수 없기 때문에, 가급적 이런 경우엔 volatile을 사용하는 것이 맞음.

volatile, 메모리 가시성4

```
package thread.volatile1;

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

public class VolatileCountMain {
    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread t = new Thread(task, "work");

        t.start();

        sleep(300);

        task.flag = false;
        log("flag = " + task.flag + " count = " + task.count);
    }
    static class MyTask implements Runnable {
        boolean flag = true;
        long count;

//        volatile boolean flag = true;
//        volatile long count;
```

```
        @Override
        public void run() {
            while(flag) {
                count++;

                if (count % 100_000_000 == 0) {
                    log("flag = " + flag + " count = " + count);
                }
            }
            log("flag = " + flag + " count = " + count + "종료 !!!!");
        }
    }
}

22:11:21.789 [     work] flag = true count = 100000000
22:11:21.866 [     work] flag = true count = 200000000
22:11:21.937 [     work] flag = true count = 300000000
22:11:21.995 [     main] flag = false count = 381478183
22:11:22.008 [     work] flag = true count = 400000000
22:11:22.008 [     work] flag = false count = 400000000종료 !!!!
```

- main 쓰레드가 flag를 false로 변경하는 시점의 count 값은 381478183 임.
- work 쓰레드가 이후에 flag를 확인하지만 flag = true.
  - work 쓰레드가 사용하는 캐시 메모리에서 읽은 것.
  - work 쓰레드는 반복문을 계속 실행하면서 count 값을 증가시킨다.
- work 쓰레드는 이후에 count 값이 400000000이 되었을 때 flag가 false로 변함.
  - 이 시점에 work 쓰레드가 사용하는 캐시 메모리의 flag 값이 false로 변경된 것.

메모리 가시성(memory visibility)

캐시 메모리를 메인 메모리에 반영하거나, 메인 메모리의 변경 내역을 캐시 메모리에 다시 불러오는 것은 언제발생할까?

- 이 부분은 CPU 설계 방식과 실행 환경에 따라 다를 수 있다. 즉시 반영될 수도 있고, 몇 밀리초 후에 될 수도 있고, 몇 초후에 될 수도 있고, 평생 반영되지 않을 수도 있다.

- 주로 컨텍스트 스위칭이 될 때, 캐시 메모리도 함께 갱신되는데, 이 부분도 환경에 따라 달라질 수 있다.
- `Thread.sleep()` , 콘솔에 출력등을 할 때 스레드가 잠시 쉬는데, 이럴 때 컨텍스트 스위칭이 되면서 주로 갱신된다. 하지만 이것이 갱신을 보장하는 것은 아니다.

여기서 정확히 12억에서 변경된 `flag` 값을 읽을 수 있었던 이유는 12억에서 콘솔에 결과를 출력하기 때문이다.

콘솔에 결과를 출력하면, 출력하는 동안 스레드가 잠시 대기하며 쉬는데, 이럴 때 컨택스트 스위칭이 발생하면서 캐시 메모리의 값이 갱신된다.

참고로 이 부분은 주로 그렇다는 것이지 확실하게 캐시의 갱신을 보장하지는 않는다. 따라서 환경에 따라 결과가 달라질 수 있다.

결국 이 상황에서 메모리 가시성 문제를 확실하게 해결하려면 `volatile` 키워드를 사용해야 한다.

```
volatile boolean flag = true;
volatile long count;

22:15:34.002 [    work] flag = true count = 100000000
22:15:34.078 [    main] flag = false count = 134231067
22:15:34.079 [    work] flag = false count = 134231067종료 !!!!
```

- 실행 결과를 보면 2가지 사실을 확인 할 수 있음.
  - main 쓰레드가 flag를 변경하는 시점에 work 쓰레드도 flag의 변경 값을 정확하게 확인할 수 있음.
  - volatile을 적용하면 캐시 메모리가 아닌 메인 메모리에 항상 직접 접근하기 때문에 성능이 상대적으로 떨어짐.
    - 위와 비교했을 때, 성능 차이를 확인할 수 있음.

자바 메모리 모델(Java Memory Model)

메모리 가시성 (memory visibility)

이처럼 멀티쓰레드 환경에서 한 쓰레드가 변경한 값이, 다른 쓰레드에서 언제 보이는지에 대한 문제를 '**메모리 가시성**'이라 함. 이름 그대로 **메모리에서 보이는가 ? 보이지 않는가 ?**

Java Memory Model

Java Memory Model(JMM)은 자바 프로그램이 어떻게 메모리에 접근하고 수정할 수 있는지를 규정하며, 특히 멀티 쓰레드 프로그래밍에서 쓰레드 간의 상호작용을 정의한다. JMM에 여러가지 내용이 있지만, 핵심은 여러 쓰레드들의 작업 순서를 보장하는 happens-before 관계에 대한 정의.

happens-before

happens-before 관계는 자바 메모리 모델에서 쓰레드 간의 작업 순서를 정의하는 개념으로, 만약 A 작업이 B 작업보다 happens-before 관계에 있다면, A 작업에서의 모든 메모리 변경 사항을 B 작업에서 볼 수 있음.

즉, A 작업에서 변경된 내용은 B 작업이 시작되기 전 모두 메모리에 반영됨.

- happens-before 관계는 이름 그대로, 한 동작이 다른 동작보다 먼저 발생함을 보장한다.
- happens-before 관계는 스레드 간의 메모리 가시성을 보장하는 규칙이다.
- happens-before 관계가 성립하면, 한 스레드의 작업을 다른 스레드에서 볼 수 있게 된다.
- 즉, 한 스레드에서 수행한 작업을 다른 스레드가 참조할 때 최신 상태가 보장되는 것이다.

이 규칙을 따르면 프로그래머가 멀티스레드 프로그램을 작성할 때 예상치 못한 동작을 피할 수 있다.

happens-before 관계가 발생하는 경우

- ***프로그램 순서 규칙***
  단일 스레드 내에서, 프로그램의 순서대로 작성된 모든 명령문은 happens-before 순서로 실행된다. 예를 들어, `inta = 1; int b = 2;` 에서 `a = 1` 은 `b = 2` 보다 먼저 실행된다.
- ***volatile 변수 규칙***
  한 스레드에서 `volatile` 변수에 대한 쓰기 작업은 해당 변수를 읽는 모든 스레드에 보이도록 한다. 즉, `volatile`변수에 대한 쓰기 작업은 그 변수를 읽는 작업보다 happens-before 관계를 형성한다.
- ***스레드 시작 규칙***
  한 스레드에서 `Thread.start()` 를 호출하면, 해당 스레드 내의 모든 작업은 `start()` 호출 이후에 실행된 작업보다 happens-before 관계가 성립한다.

```
Thread t = new Thread(task);
```

```
t.start();
```

여기에서 `start()` 호출 전에 수행된 모든 작업은 새로운 스레드가 시작된 후의 작업보다 happens-before 관계를가진다.

- ***스레드 종료 규칙***
  한 스레드에서 `Thread.join()` 을 호출하면, join 대상 스레드의 모든 작업은 `join()` 이 반환된 후의 작업보다happens-before 관계를 가진다. 예를 들어, `thread.join()` 호출 후에 `thread` 의 모든 작업이 완료되어야 하며,이 작업은 `join()` 이 반환된 후에 참조 가능하다. `1 ~ 100` 까지 값을 더하는 `sumTask` 예시를 떠올려보자.
- ***인터럽트 규칙***
  한 스레드에서 `Thread.interrupt()` 를 호출하는 작업이, 인터럽트된 스레드가 인터럽트를 감지하는 시점의 작업보다 happens-before 관계가 성립한다. 즉, `interrupt()` 호출 후, 해당 스레드의 인터럽트 상태를 확인하는 작업이 happens-before 관계에 있다. 만약 이런 규칙이 없다면 인터럽트를 걸어도, 한참 나중에 인터럽트가 발생할 수 있다.
- ***객체 생성 규칙***
  객체의 생성자는 객체가 완전히 생성된 후에만 다른 스레드에 의해 참조될 수 있도록 보장한다. 즉, 객체의 생성자에서초기화된 필드는 생성자가 완료된 후 다른 스레드에서 참조될 때 happens-before 관계가 성립한다.
- ***모니터 락 규칙***
  한 스레드에서 `synchronized` 블록을 종료한 후, 그 모니터 락을 얻는 모든 스레드는 해당 블록 내의 모든 작업을 볼수 있다. 예를 들어, `synchronized(lock) { ... }` 블록 내에서의 작업은 블록을 나가는 시점에 happens-before 관계가 형성된다. 뿐만 아니라 `ReentrantLock` 과 같이 락을 사용하는 경우에도 happens-before 관계가성립한다.

  참고로 `synchronized` 와 `ReentrantLock` 은 바로 뒤에서 다룬다.
- ***전이 규칙 (Transitivity Rule)***

  만약 A가 B보다 happens-before 관계에 있고, B가 C보다 happens-before 관계에 있다면, A는 C보다 happens-before 관계에 있다.

쓰레드의 생성과 종료, 인터럽트 등은 쓰레드의 상태를 변경하므로 당연.

즉, volatile 또는 쓰레드 동기화 (synchronized, ReentrantLcok)을 사용하면 메모리 가시성 문제가 발생하지 않음.