5. 메모리 가시성

#0.강의/1.자바로드맵/5.자바-고급1편

volatile, 메모리 가시성1

`volatile`과 메모리 가시성을 이해하기 위해, 간단한 예제를 만들어보자.

주의: `volatile`은 자바에서 예약된 키워드이다. 따라서 패키지 이름으로 사용할 수 없다. 대신에 패키지 이름에 숫자를 붙여서 `volatile1`을 패키지 이름으로 사용했다.

```
package thread.volatile1;

import static util.MyLogger.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("task 종료");
        }

    }
}
```

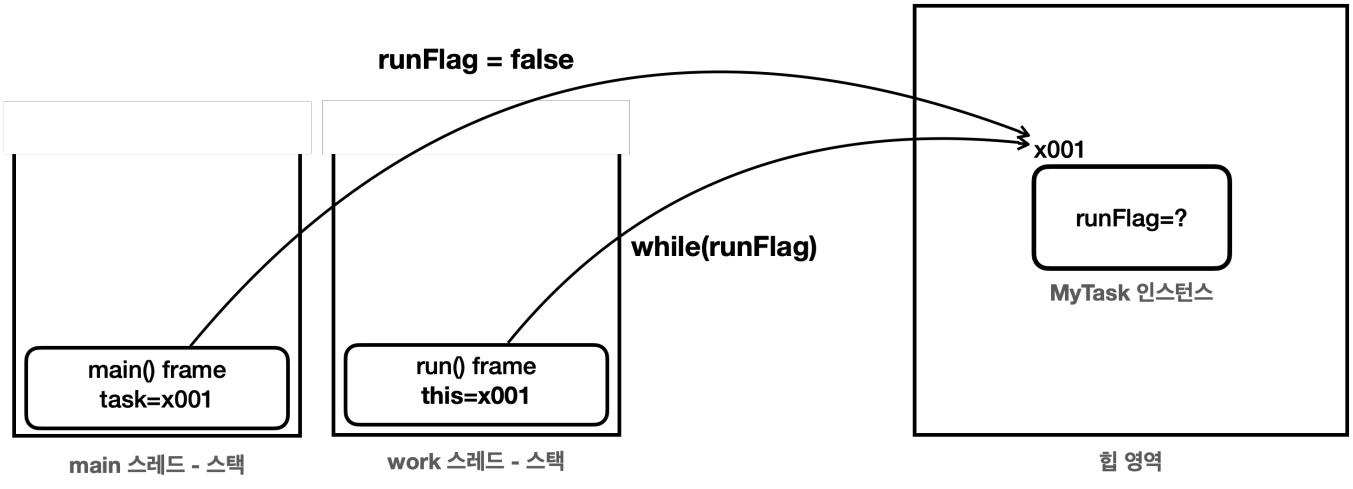
프로그램은 아주 간단하다. `runFlag`를 사용해서 스레드의 작업을 종료한다.

- `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.

주의! - 여기서는 아직 volatile 키워드를 사용하면 안된다! 다음과 같이 꼭! 작성해야 한다.

```
boolean runFlag = true;
```

메모리 그림



- `main` 스레드, `work` 스레드 모두 `MyTask` 인스턴스(`x001`)에 있는 `runFlag`를 사용한다.
- 이 값을 `false`로 변경하면 `work` 스레드의 작업을 종료할 수 있다.

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
```

그런데 실제 실행 결과는 기대하는 실행 결과와 다르게 다음과 같이 출력될 것이다.

실제 실행 결과

```
15:40:55.367 [     main] runFlag = true
15:40:55.367 [     work] task 시작
15:40:56.374 [     main] runFlag를 false로 변경 시도
15:40:56.374 [     main] runFlag = false
15:40:56.375 [     main] main 종료
```

실제 실행 결과를 보면 `task 종료`가 출력되지 않는다! 그리고 자바 프로그램도 멈추지 않고 계속 실행된다. 정확히는 `work` 스레드가 while문에서 빠져나오지 못하고 있는 것이다.

분명히 `runFlag`를 `false`로 변경한 것을 콘솔에 출력해서 두 눈으로 확인했다!

```
15:40:56.374 [     main] runFlag = false
```

이때 `work` 스레드가 실행하는 `while (runFlag)` 조건은 `false`가 된다. 따라서 `work` 스레드는 while문을 빠져나오고 task 종료를 출력해야 한다! 도대체 어떻게 된 일일까?

참고: 실행 환경에 따라 실제 실행 결과는 달라질 수 있다. 어떤 분들은 특정 시간이 지나고 task 종료가 출력될 수 도 있고, 어떤 분들은 기대하는 실행 결과와 똑같이 작동할 수도 있다. 가급적 예제 코드와 똑같이 작성해야 비슷한 결과를 얻을 가능성이 높아진다.

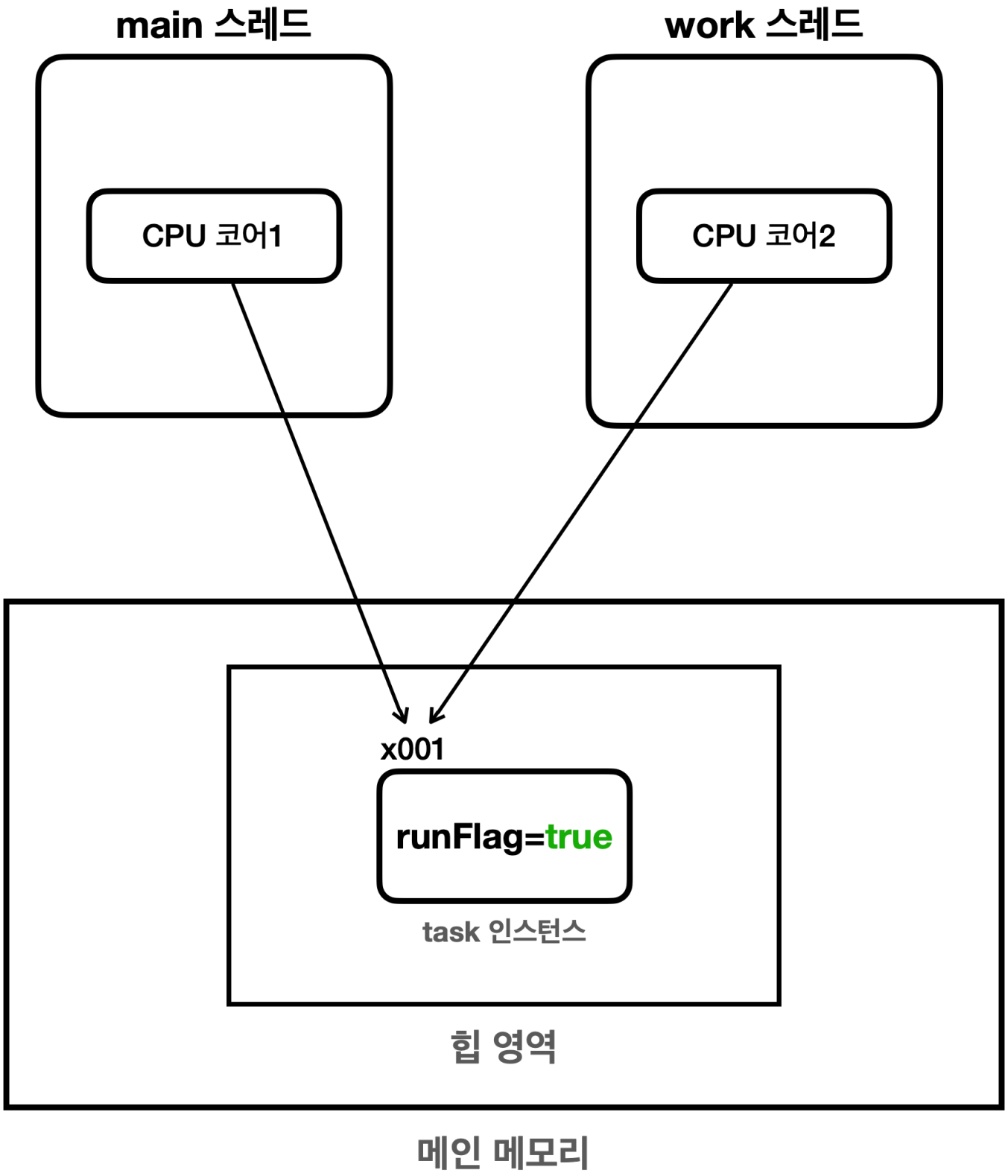
volatile, 메모리 가시성2

메모리 가시성 문제

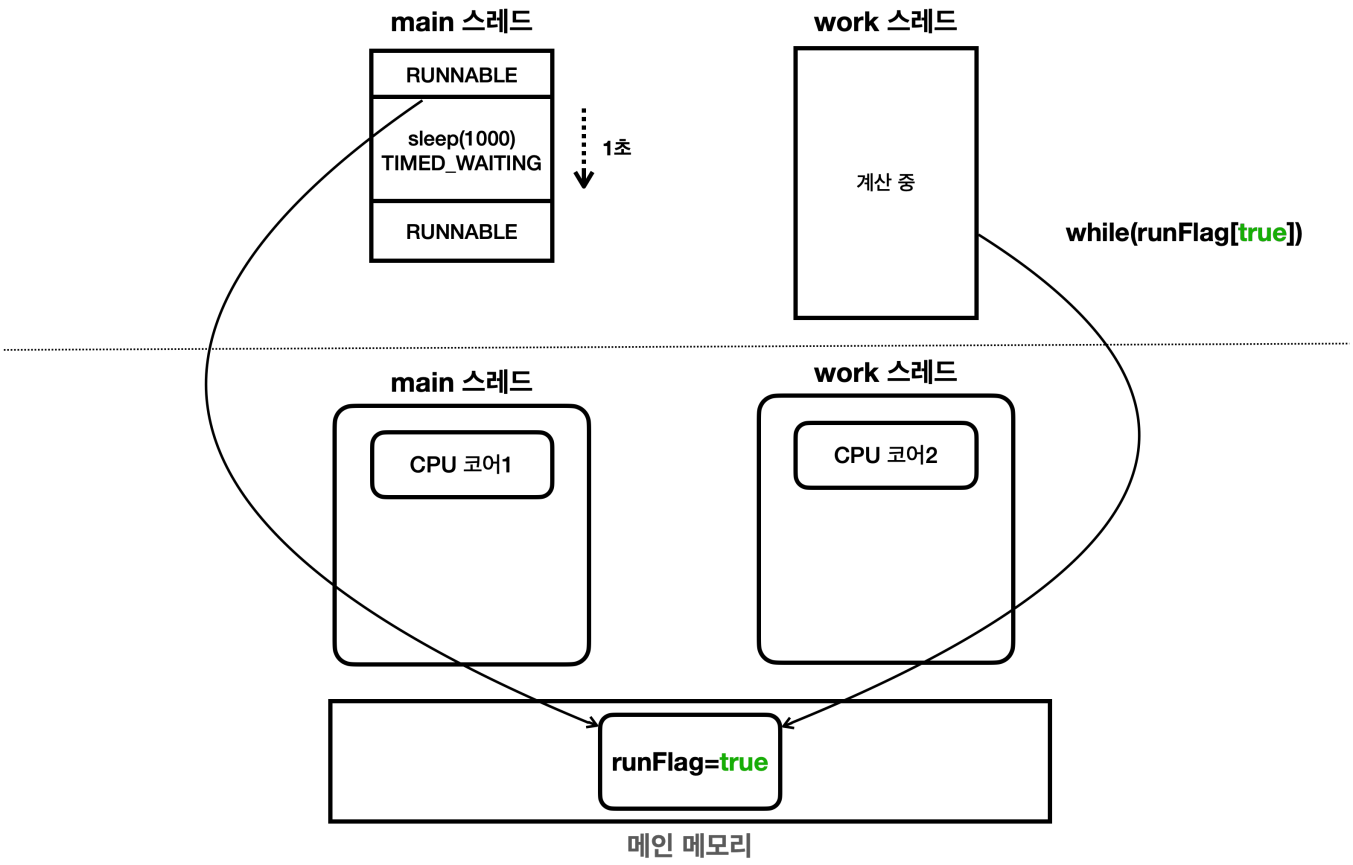
멀티스레드는 이미 여러 스레드가 작동해서 안 그래도 이해하기 어려운데, 거기에 한술 더하는 문제가 있으니, 바로 메모리 가시성(memory visibility)문제이다. 이게 어떤 문제이고, 왜 이런 문제가 발생하는지, 그리고 어떻게 해결하는지 그림으로 차근차근 알아보자.

먼저 우리가 일반적으로 생각하는 메모리 접근 방식에 대해서 설명하겠다.

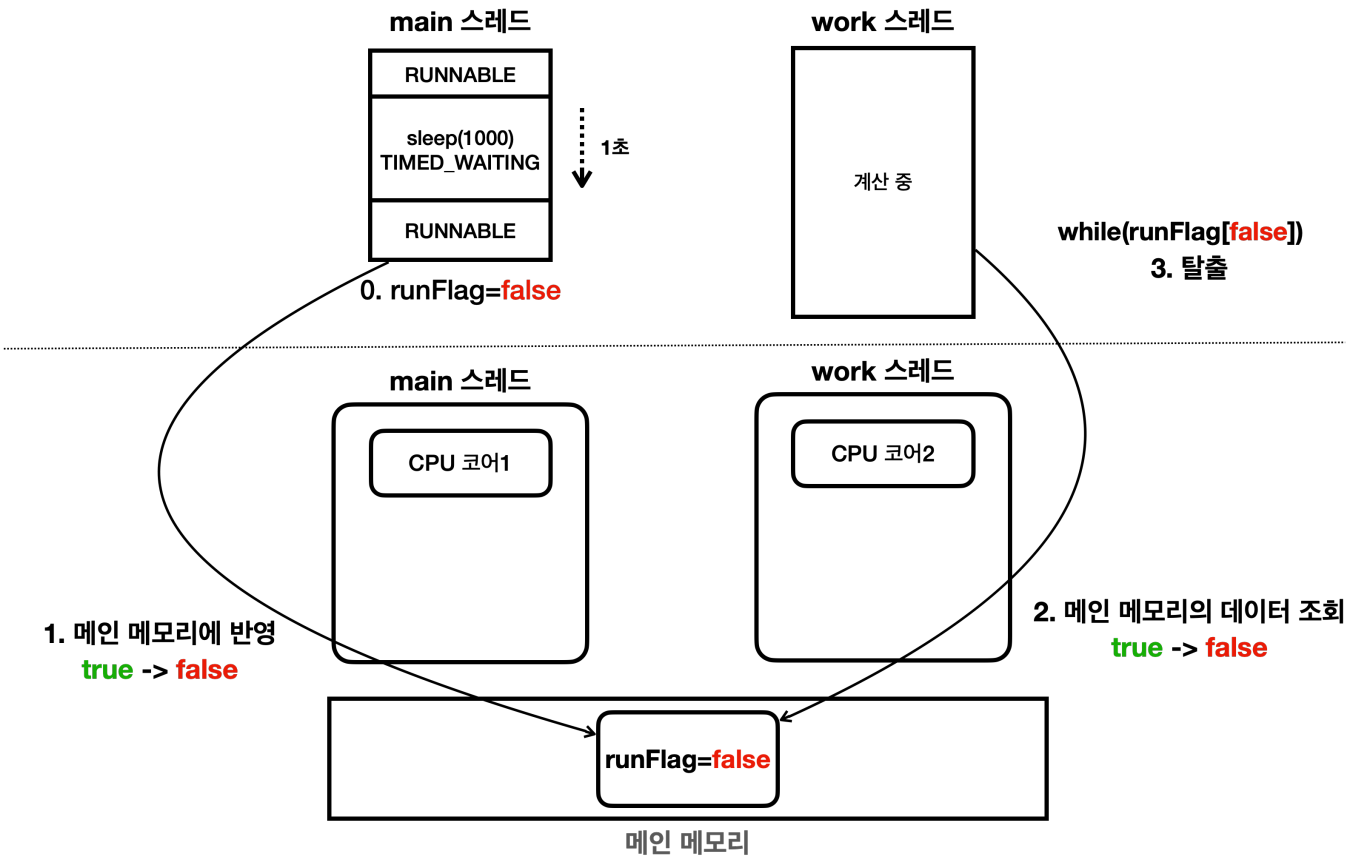
일반적으로 생각하는 메모리 접근 방식



- `main` 스레드와 `work` 스레드는 각각의 CPU 코어에 할당되어서 실행된다.
- 물론 CPU 코어가 1개라면 빠르게 번갈아 가면서 실행될 수 있다.



- 점선 위쪽은 스레드의 실행 흐름을 나타내고, 점선 아래쪽은 하드웨어를 나타낸다.
- 자바 프로그램을 실행하고 `main` 스레드와 `work` 스레드는 모두 메인 메모리의 `runFlag`의 값을 읽는다.
- 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.

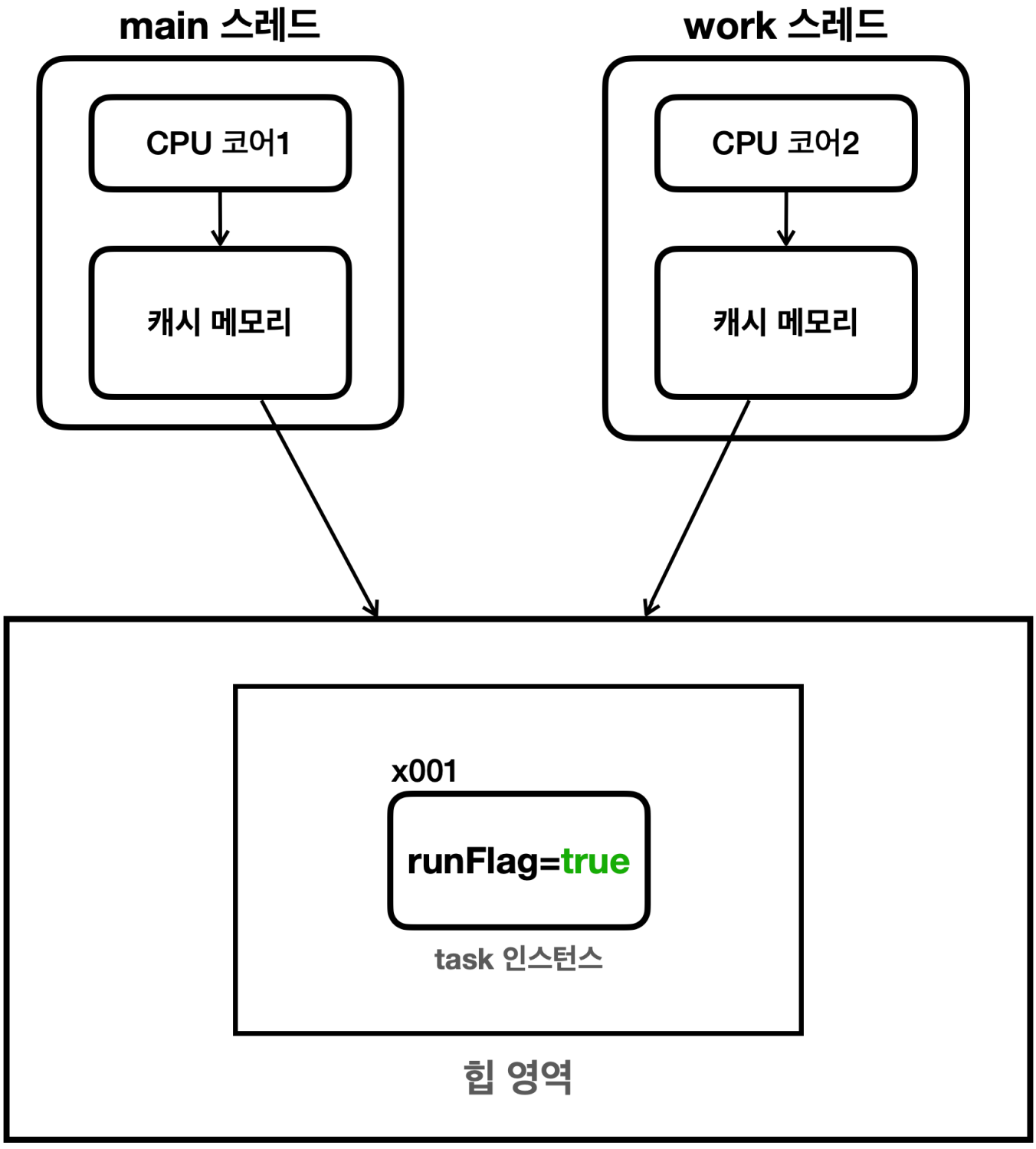


0. `main` 스레드는 `runFlag` 값을 `false`로 설정한다.
1. 이때 메인 메모리의 `runFlag` 값이 `false`로 설정된다.
2. `work` 스레드는 `while(runFlag)`를 실행할 때 `runFlag`의 데이터를 메인 메모리에서 확인한다.
3. `runFlag`의 값이 `false`이므로 while문을 탈출하고, `"task 종료"`를 출력한다.

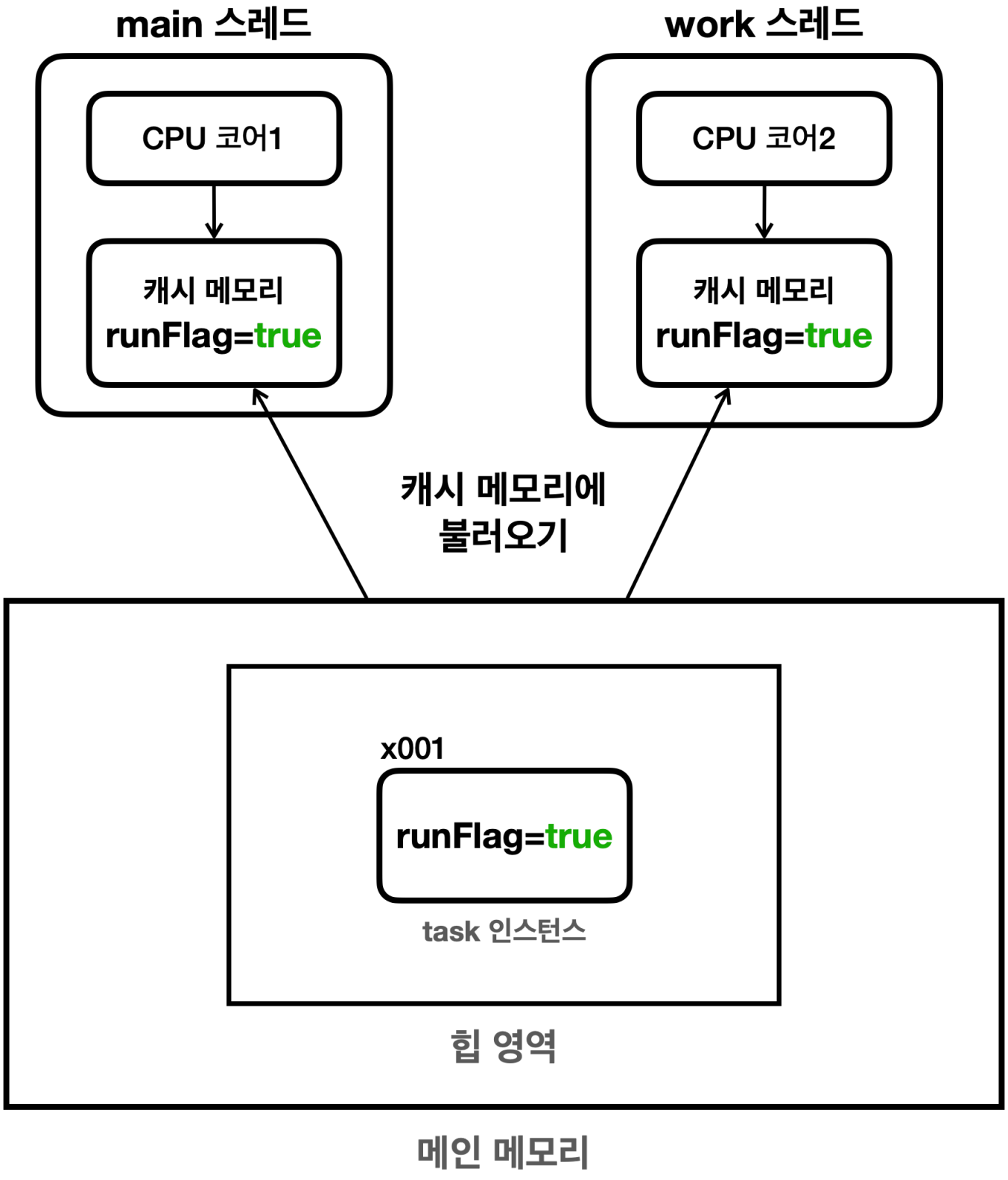
아마도 이런 시나리오를 생각했을 것이다. 그런데 실제로는 이런 방식으로 작동하지 않는다.

실제 메모리의 접근 방식

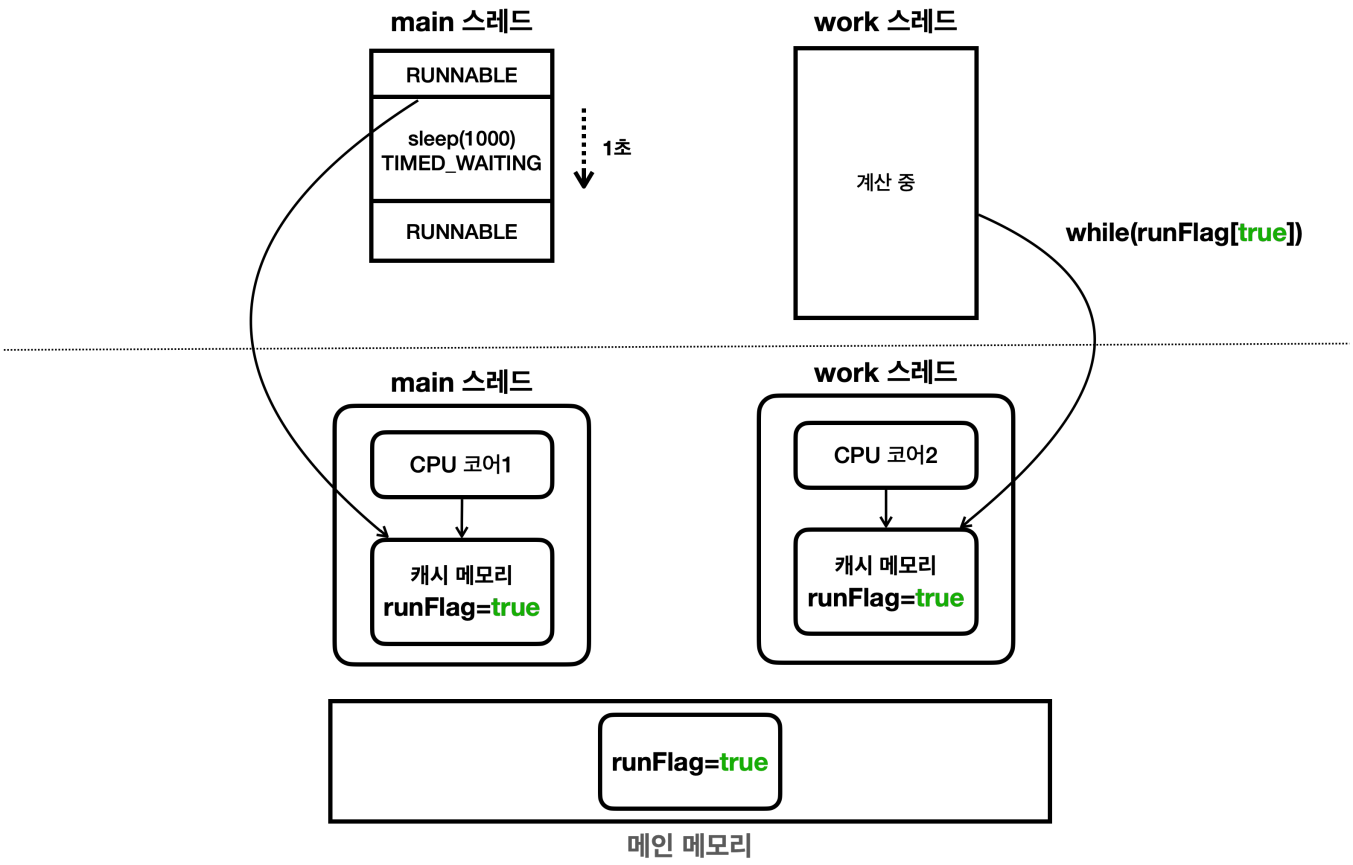
CPU는 처리 성능을 개선하기 위해 중간에 캐시 메모리라는 것을 사용한다.



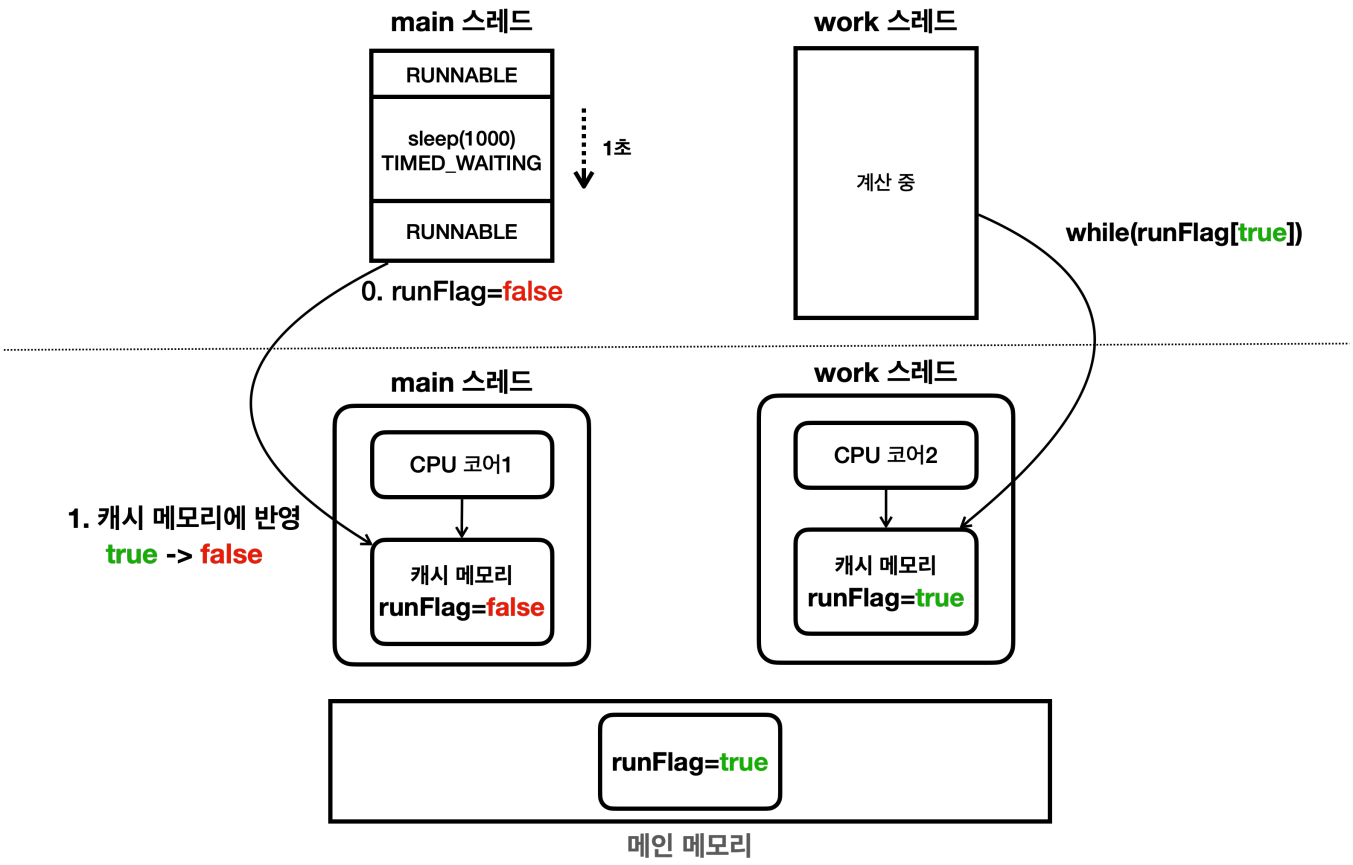
- 메인 메모리는 CPU 입장에서 보면 거리도 멀고, 속도도 상대적으로 느리다. 대신에 상대적으로 가격이 저렴해서 큰 용량을 쉽게 구성할 수 있다.
- CPU 연산은 매우 빠르기 때문에 CPU 연산의 빠른 성능을 따라가려면, CPU 가까이에 매우 빠른 메모리가 필요한데, 이것이 바로 캐시 메모리이다. 캐시 메모리는 CPU와 가까이 붙어있고, 속도도 매우 빠른 메모리이다. 하지만 상대적으로 가격이 비싸기 때문에 큰 용량을 구성하기는 어렵다.
- 현대의 CPU 대부분은 코어 단위로 캐시 메모리를 각각 보유하고 있다.
  - 참고로 여러 코어가 공유하는 캐시 메모리도 있다.



- 각 스레드가 `runFlag`의 값을 사용하면 CPU는 이 값을 효율적으로 처리하기 위해 먼저 `runFlag`를 캐시 메모리에 불러온다.
- 그리고 이후에는 캐시 메모리에 있는 `runFlag`를 사용하게 된다.



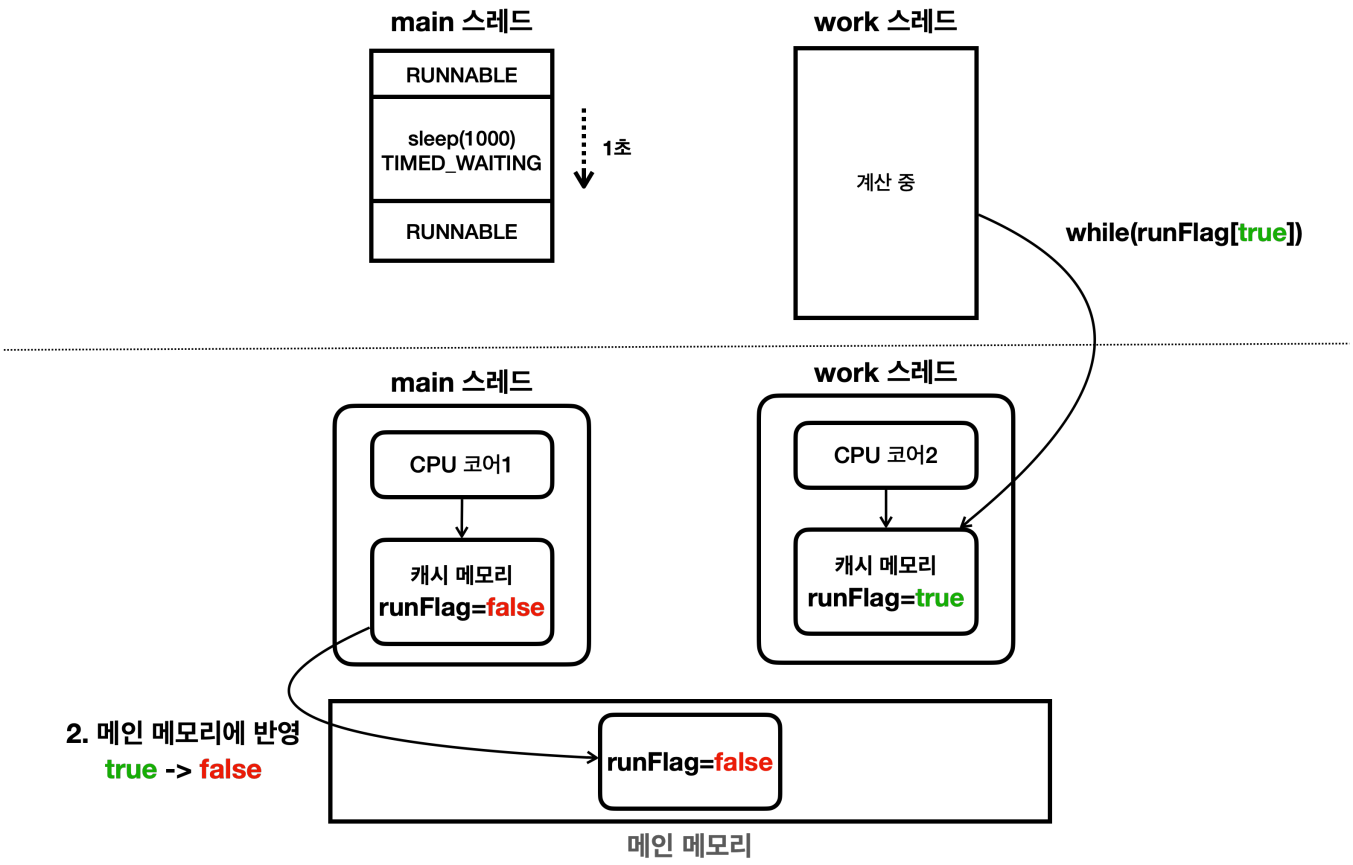
- 점선 위쪽은 스레드의 실행 흐름을 나타내고, 점선 아래쪽은 하드웨어를 나타낸다.
- 자바 프로그램을 실행하고 `main` 스레드와 `work` 스레드는 모두 `runFlag`의 값을 읽는다.
- CPU는 이 값을 효율적으로 처리하기 위해 먼저 캐시 메모리에 불러온다.
- `main` 스레드와 `work` 스레드가 사용하는 `runFlag`가 각각의 캐시 메모리에 보관된다.
- 프로그램의 시작 시점에는 `runFlag`를 변경하지 않기 때문에 모든 스레드에서 `true`의 값을 읽는다.
  - 참고로 `runFlag`의 초기값은 `true`이다.
- `work` 스레드의 경우 `while(runFlag[true])`가 만족하기 때문에 while문을 계속 반복해서 수행한다.



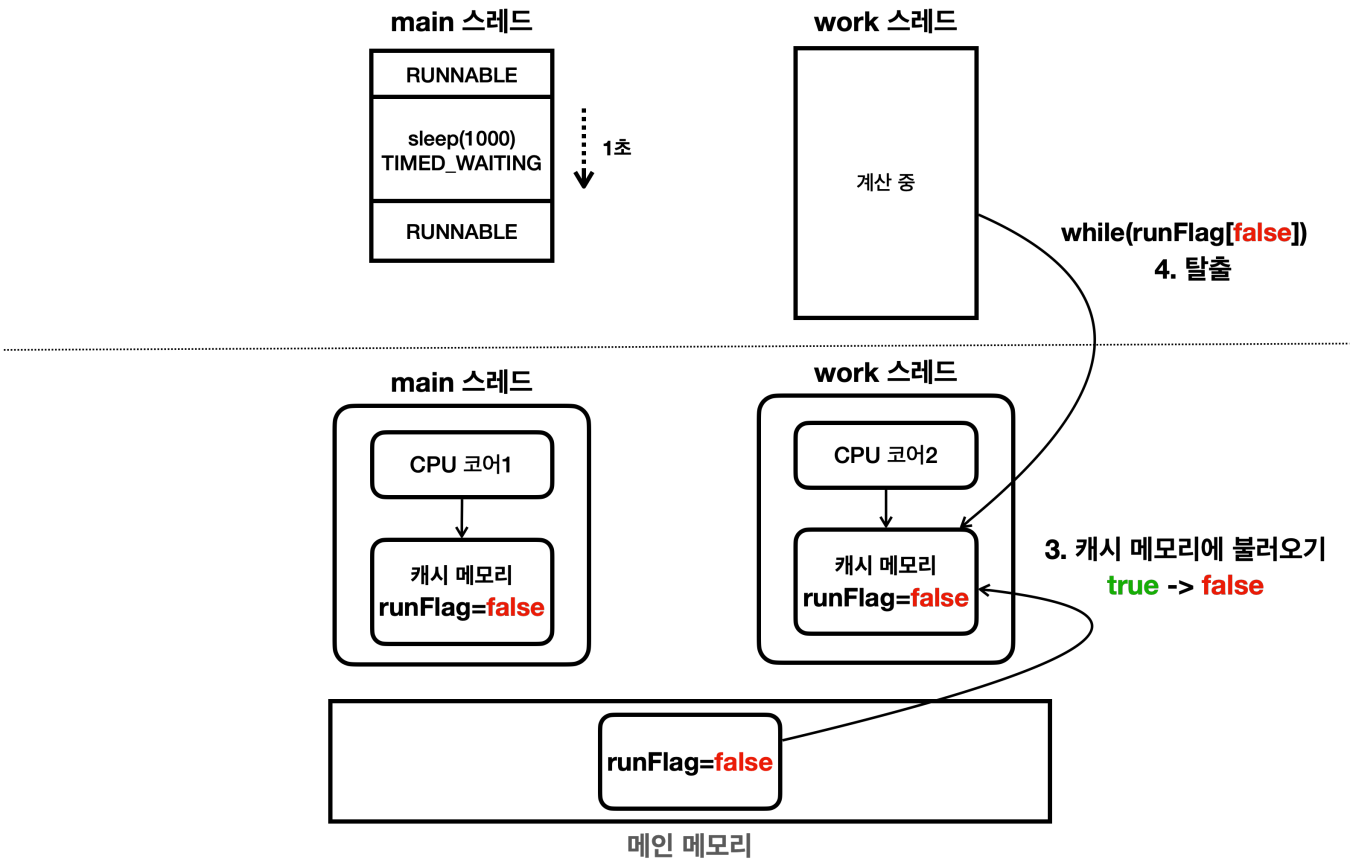
0. `main` 스레드는 `runFlag`를 `false`로 설정한다.
1. 이때 캐시 메모리의 `runFlag`가 `false`로 설정된다.

여기서 핵심은 캐시 메모리의 runFlag 값만 변한다는 것이다! 메인 메모리에 이 값이 즉시 반영되지 않는다.

- `main` 스레드가 `runFlag`의 값을 변경해도 CPU 코어1이 사용하는 캐시 메모리의 `runFlag` 값만 `false`로 변경된다.
- `work` 스레드가 사용하는 CPU 코어2의 캐시 메모리의 `runFlag` 값은 여전히 `true`이다.
- `work` 스레드의 경우 `while(runFlag[true])`가 만족하기 때문에 while문을 계속 반복해서 수행한다.



- 캐시 메모리에 있는 `runFlag`의 값이 언제 메인 메모리에 반영될까?
- 이 부분에 대한 정답은 "알 수 없다"이다. CPU 설계 방식과 종류의 따라 다르다. 극단적으로 보면 평생 반영되지 않을 수도 있다!
- 메인 메모리에 반영을 한다고 해도, 문제는 여기서 끝이 아니다.
- 메인 메모리에 반영된 `runFlag` 값을 `work` 스레드가 사용하는 캐시 메모리에 다시 불러와야 한다.



- 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.

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

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

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 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?

volatile, 메모리 가시성3

캐시 메모리를 사용하면 CPU 처리 성능을 개선할 수 있다. 하지만 때로는 이런 성능 향상보다는, 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 접근하면 된다.

자바에서는 `volatile`이라는 키워드로 이런 기능을 제공한다.

VolatileFlagMain - volatile 적용

```
package thread.volatile1;

import static util.MyLogger.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) {
```

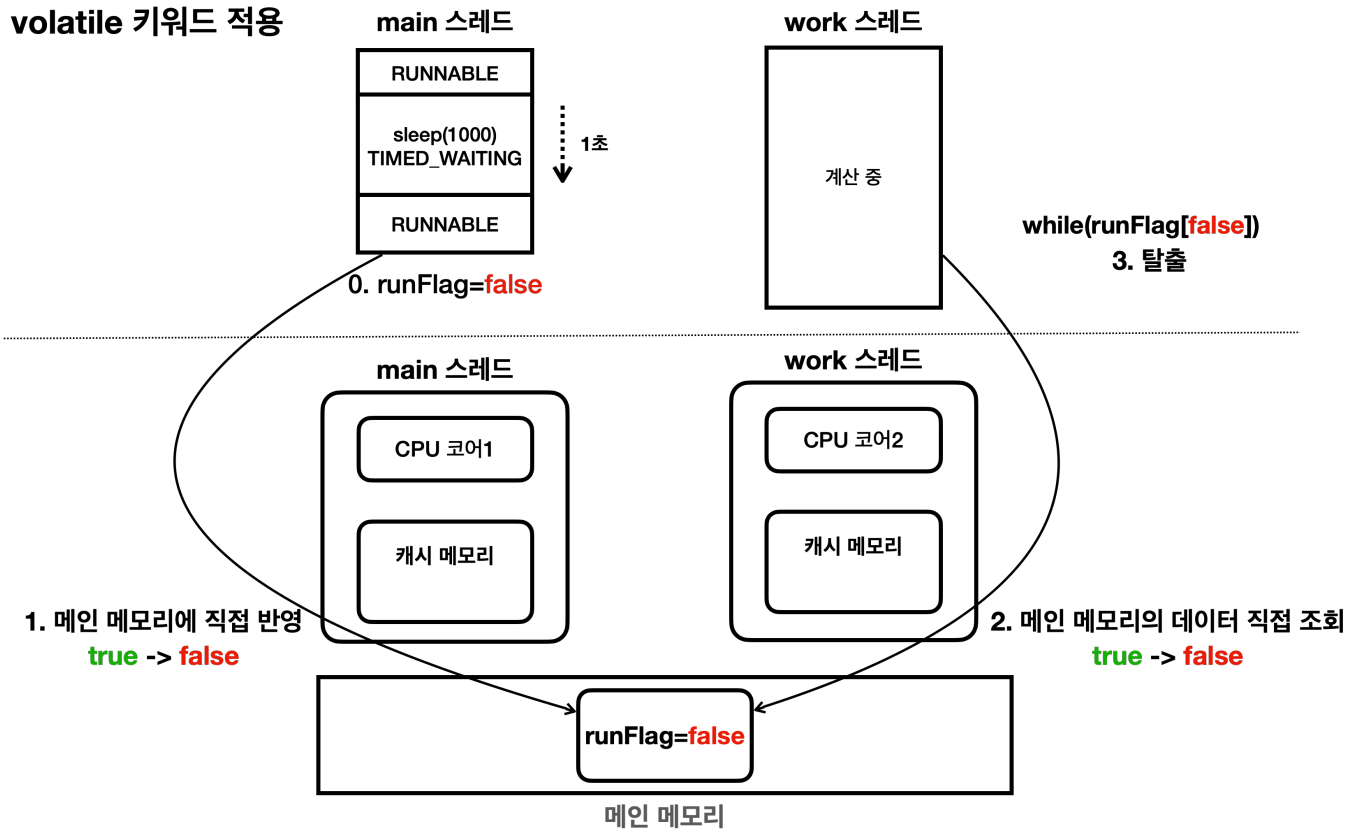
```
                //runFlag가 false로 변하면 탈출
            }
            log("task 종료");
        }

    }
}
```

- 기존 코드에서 `boolean runFlag` 앞에 `volatile` 이라는 키워드만 하나 추가해보자.
  - `volatile boolean runFlag = true;`
- 이렇게 하면 `runFlag` 에 대해서는 캐시 메모리를 사용하지 않고, 값을 읽거나 쓸 때 항상 메인 메모리에 직접 접근한다.

실행 결과

```
15:39:59.830 [     main] runFlag = true
15:39:59.830 [ thread-0] task 시작
15:40:00.837 [     main] runFlag를 false로 변경 시도
15:40:00.838 [ thread-0] task 종료
15:40:00.838 [     main] runFlag = false
15:40:00.838 [     main] main 종료
```

실행 결과를 보면 `runFlag` 를 `false` 로 변경하자마자 `"task 종료"` 가 출력되는 것을 확인할 수 있다.
그리고 모든 스레드가 정상 종료되기 때문에 자바 프로그램도 종료된다.



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, 메모리 가시성4

이번에는 실시간성이 있는 예제로 메모리 가시성을 확인해보자.

volatile 미적용

```
package thread.volatile1;

import static util.MyLogger.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(1000);

        task.flag = false;
        log("flag = " + task.flag + ", count = " + task.count + " in main");
    }

    static class MyTask implements Runnable {
        boolean flag = true;
        long count;
        //volatile boolean flag = true;
        //volatile long count;

        @Override
        public void run() {
            while (flag) {
                count++;
                //1억번에 한번씩 출력
                if (count % 100_000_000 == 0) {
                    //주석 처리 한다면...
                    log("flag = " + flag + ", count = " + count + " in
while()");
                }
            }
            log("flag = " + flag + ", count = " + count + " 종료");
        }
    }
}
```

work 스레드

- `work` 스레드는 반복문에서 `count++`을 사용해서 이 값을 계속 증가한다.
- 반복문을 1억번 실행할 때 마다 한 번씩 `count`의 값을 출력한다.
- `flag` 값이 `false`가 되면 반복문을 탈출하고 `count` 값을 출력한다.

main 스레드

- `main` 스레드는 1초간 대기하다가 `flag` 값을 `false`로 변경한다.
- `flag` 값을 `false`로 변경한 시점에 `count` 값을 출력한다.

실행 결과

```
10:45:04.429 [     work] flag = true, count = 100000000 in while()
10:45:04.518 [     work] flag = true, count = 200000000 in while()
10:45:04.605 [     work] flag = true, count = 300000000 in while()
10:45:04.691 [     work] flag = true, count = 400000000 in while()
10:45:04.775 [     work] flag = true, count = 500000000 in while()
10:45:04.859 [     work] flag = true, count = 600000000 in while()
10:45:04.942 [     work] flag = true, count = 700000000 in while()
10:45:05.024 [     work] flag = true, count = 800000000 in while()
10:45:05.107 [     work] flag = true, count = 900000000 in while()
10:45:05.189 [     work] flag = true, count = 1000000000 in while()
10:45:05.273 [     work] flag = true, count = 1100000000 in while()
10:45:05.338 [     main] flag = false, count = 1176711196 in main
10:45:05.357 [     work] flag = true, count = 1200000000 in while()
10:45:05.357 [     work] flag = false, count = 1200000000 종료
```

- `main` 스레드가 `flag`를 `false`로 변경하는 시점에 `count` 값은 `1176711196`이다.
- `work` 스레드가 이후에 `flag`를 확인하지만 아직 `flag = true`이다.
  - `work` 스레드가 사용하는 캐시 메모리에서 읽은 것이다.
  - `work` 스레드는 반복문을 계속 실행하면서 `count` 값을 증가시킨다.
- `work` 스레드는 이후에 `count` 값이 `1200000000`이 되었을 때 `flag`가 `false`로 변한 것을 확인할 수 있다.
  - 이 시점에 `work` 스레드가 사용하는 캐시 메모리의 `flag` 값이 `false`로 변경되었다.

시점의 차이

- `main` 스레드가 `flag`를 `false`로 변경한 시점에 `count` 값은 `1176711196`이다.
- `work` 스레드가 `flag` 값을 `false`로 확인한 시점에 `count` 값은 `1200000000`이다.

결과적으로 `main` 스레드가 `flag` 값을 `false`로 변경하고 한참이 지나서야 `work` 스레드는 `flag` 값이 `false`로 변경된 것을 확인한 것이다.

메모리 가시성(memory visibility)

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

이 부분은 CPU 설계 방식과 실행 환경에 따라 다를 수 있다. 즉시 반영될 수도 있고, 몇 밀리초 후에 될 수도 있고, 몇 초 후에 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 적용

주석을 변경해서 `flag`, `count`에 `volatile`을 적용해보자.

```
static class MyTask implements Runnable {
    //boolean flag = true;
    //long count;
    volatile boolean flag = true;
    volatile long count;
}
```

실행 결과

```
10:54:11.064 [     work] flag = true, count = 100000000 in while()
10:54:11.508 [     work] flag = true, count = 200000000 in while()
10:54:11.605 [     work] flag = false, count = 222297705 종료
10:54:11.606 [     main] flag = false, count = 222297705 in main
```

- `main` 스레드가 `flag`를 `false`로 변경하는 시점에 `count` 값은 `222297705`이다.
- `work` 스레드가 `flag`를 `false`로 확인하는 시점에 `count` 값은 `222297705`이다.

실행 결과를 보면 2가지 사실을 확인할 수 있다.

- `main` 스레드가 `flag`를 변경하는 시점에 `work` 스레드도 `flag`의 변경 값을 정확하게 확인할 수 있다.
- `volatile`을 적용하면 캐시 메모리가 아니라 메인 메모리에 항상 직접 접근하기 때문에 성능이 상대적으로 떨어진다.
  - `volatile`이 없을 때: `1176711196`, 약 11억(정확한 숫자는 아니고 대략적인 수치다)
  - `volatile`이 있을 때: `222297705`, 약 2.2억
  - 둘을 비교해보면 물리적으로 약 5배의 성능 차이를 확인할 수 있다. 성능은 환경에 따라 차이가 있다.

> **참고: volatile과 최적화**

> 이해를 돕기 위해 `volatile` 키워드를 사용하면 항상 메인 메모리에 직접 접근한다고 단순화해서 설명했지만, 실제로는 내부적으로 다양한 최적화를 수행한다.
> 더 깊은 내부 동작이 궁금하다면 '메모리 배리어'나 'JIT 컴파일러 volatile 최적화' 같은 키워드를 검색해보자.
> 지금 우리는 `volatile` 이 스레드 간의 '**가시성**'을 보장하며, 그 대가로 **일정 수준의 성능 비용이 발생**한다는 핵심만 이해하면 충분하다.

자바 메모리 모델(Java Memory Model)

메모리 가시성(memory visibility)

멀티스레드 환경에서 한 스레드가 변경한 값이 다른 스레드에서 언제 보이는지에 대한 것을 메모리 가시성(memory visibility)이라 한다. 이름 그대로 메모리에 변경한 값이 보이는가, 보이지 않는가의 문제이다.

Java Memory Model

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

happens-before

happens-before 관계는 자바 메모리 모델에서 스레드 간의 작업 순서를 정의하는 개념이다. 만약 A 작업이 B 작업보다 happens-before 관계에 있다면, A 작업에서의 모든 메모리 변경 사항은 B 작업에서 볼 수 있다. 즉, A 작업에서 변경된 내용은 B 작업이 시작되기 전에 모두 메모리에 반영된다.

- 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

이런 부분은 외우는 것이 아니라, 이런 것이 있다 정도로 읽고 넘어가면 된다. 몇몇 규칙은 아직 학습하지 않은 기능이지

만, 강의가 끝나고 나면 자연스럽게 이해가 될 것이다.

프로그램 순서 규칙

단일 스레드 내에서, 프로그램의 순서대로 작성된 모든 명령문은 happens-before 순서로 실행된다. 예를 들어, `int a = 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 관계를 가진다.

스레드 종료 규칙

한 스레드에서 `Thread.join()`을 호출하면, join 대상 스레드의 모든 작업은 `join()`이 반환된 후의 작업보다 happens-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 생성자에서 초기화된 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 볼 수 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 관계가 성립한다.

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.

정리

스레드의 생성과 종료, 인터럽트 등은 스레드의 상태를 변경하기 때문에 어찌보면 당연하다.

그래서 쉽게 한 줄로 이야기하면 다음과 같이 정리할 수 있다.

volatile 또는 스레드 동기화 기법(synchronized, ReentrantLock)을 사용하면 메모리 가시성의 문제가 발생하지 않는다.

이제 스레드 동기화 기법을 배울 시간이 되었다.

정리