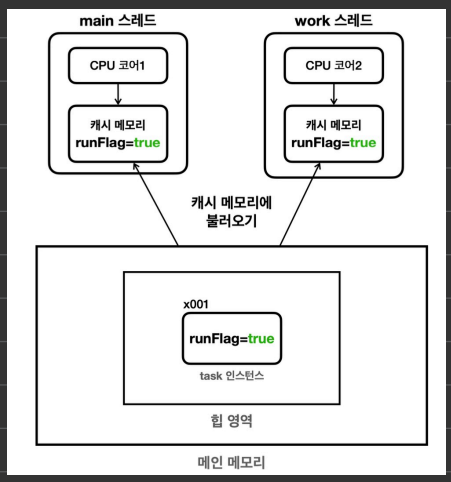
Section 6 메모리 가시성

- Memory Visability
  - 멀티스레드 환경에서 한 스레드가 변경한 값이, 다른 스레드에 언제 보여지는가?
    - ↳ 각 스레드는 캐시메모리에 load하여 resource를 사용
    - ↳ Context Switching 시 Cache Update



캐시는 직접접근보다 데이터 많음.

↳ 각 스레드의 Memory Visability를 위해 성능을 포기하고 Volatile 사용

- JVM

*happens-before : 스레드간 작업순서의 규칙, 순차적인 실행보장

↳ volatile 이나, 스레드 동기화 기법을 사용하면 Memory Visability 보장

#Section 7 동기화

1. Critical Section
   - 여러 스레드가 동시에 접근하는 공유자원
     - 한번에 한 스레드가 접근 가능하도록 보호해야함
     - Read시엔 OK,
2. Synchronized
   - 공유자원에 대해서 일관성있고 안전하게 관리하게함
     - Lock을 얻은 스레드만이 접근가능
     - Lock을 얻는 순서는 알 수없음.

메서드 동기화: 메서드를 `synchronized`로 선언해서, 메서드에 접근하는 스레드가 하나뿐이도록 보장한다.

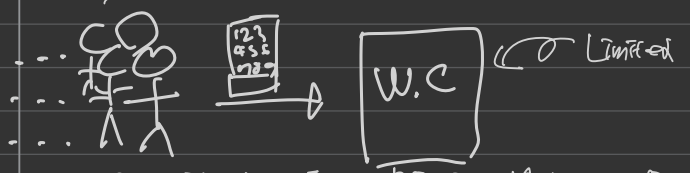
```
public synchronized void synchronizedMethod() {
     // 코드
 }
```

블록 동기화: 코드 블록을 `synchronized`로 감싸서, 동기화를 구현할 수 있다.

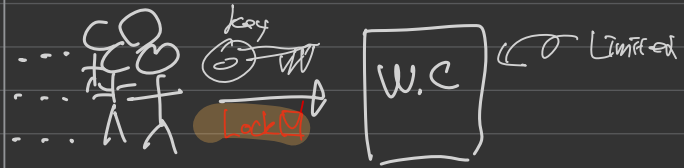
```
public void method() {
    synchronized(this) {
        // 동기화된 코드
    }
}
```

* 단점
  - Lock을 얻지 못한 스레드들 BLOCKED 상태
    - 특정시간 X, Interrupt X
  - 공정성 문제

* 그림예제



e.g) 화장실에 3이라는 번호를 이용하여 여러사람들이 입...
- Solution : 화장실 key를 입.



- key가 Lock의 역할을 수행

#Section 8 고급동기화

- Synchronized의 단점인 무한대기와 공정성을 해결하기 위함.
  1) LockSupport
  2) ReentrantLock

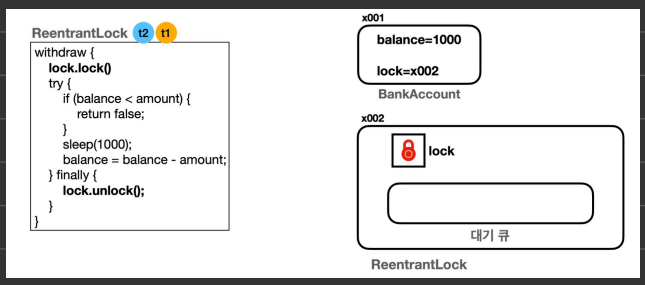
1. LockSupport

- 스레드를 WAITING 상태로 만들 수 있음
  ↳ Interrupt가 가능
- park(), parkNanos(), unpark(thread)
  ↓ WAITING / ↓ TIMED_WAITING / ↓ RUNNABLE

↳ Synchronized의 무한대기 문제를 해결가능 But, 구현하기 어려움

2. ReentrantLock

- 무한대기, 공정성문제 해결가능
- 여러가지 구현체를 통해 쉽게 상호배제, 동기화 기능을 제공



1. Lock을 얻기를 실패한 스레드는 Queue를 통해 관리됨
2. Lock을 얻지못한 스레드는 WAITING 상태 (TIMED_WAITING 가능)
3. ReentrantLock은 fair/unfair를 통해 공정성 해결
   ↳ FIFO
   ↳ Best Effort FIFO?

↳ 결과적으로, Lock을 관리하는 Queue와 TIMED_WAITING / Interrupt를 이용해 Synchronized의 한계를 극복