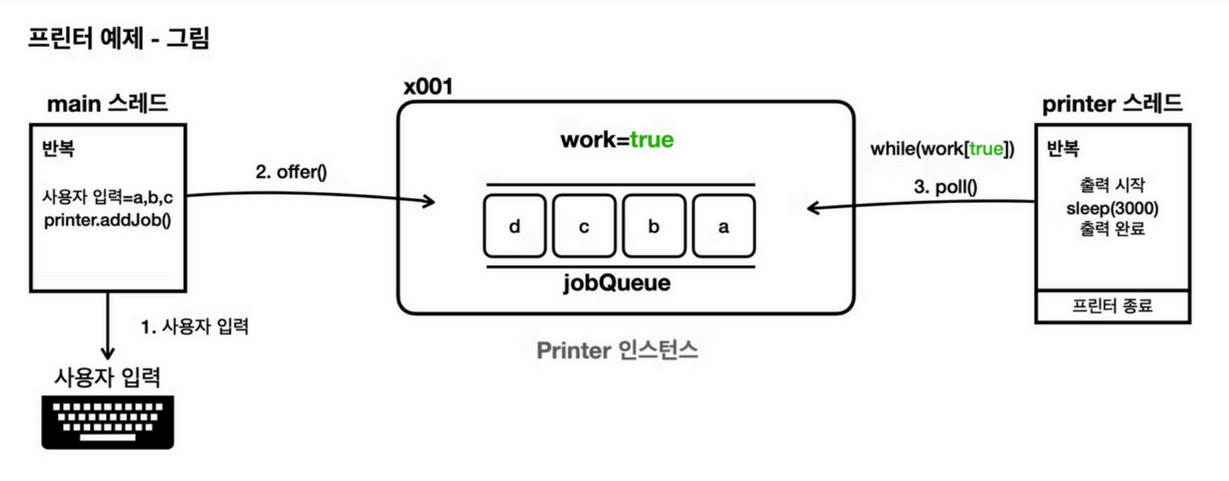
3주차 스터디

목차

- 생산자 소비자 문제
- BoundedQueueV1
- BoundedQueueV2
- BoundedQueueV3
- BoundedQueueV5
- synchronized vs ReentrantLock
- BlockingQueue

생산자 소비자 문제



생산자(Producer)

- 데이터를 생성하는 역할을 한다.

소비자(Consumer)

- 생성된 데이터를 사용하는 역할

버퍼(Buffer)

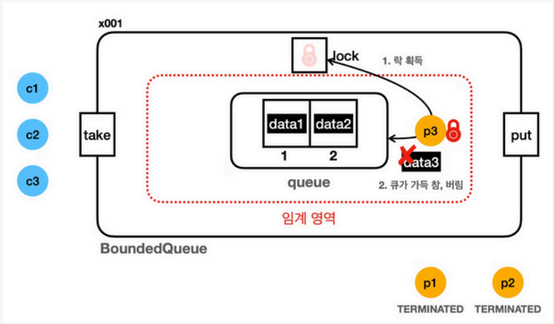
- 생산자가 생성한 데이터를 일시적으로 저장하는 공간
- 버퍼는 한정된 크기를 가지며, 생산 자와 소비자가 이 버퍼를 통해 데이터를 주고받는다.

문제 상황

- **생산자가 너무 빠를 때:** 버퍼가 가득 차서 더 이상 데이터를 넣을 수 없을 때까지 생산자가 데이터를 생성한다.
  - 버퍼가 가득 찬 경우 생산자는 버퍼에 빈 공간이 생길 때까지 기다려야 한다.
- **소비자가 너무 빠를 때:** 버퍼가 비어서 더 이상 소비할 데이터가 없을 때까지 소비자가 데이터를 처리한다.
  - 버퍼가 비어있을 때 소비자는 버퍼에 새로운 데이터가 들어올 때까지 기다려야 한다.

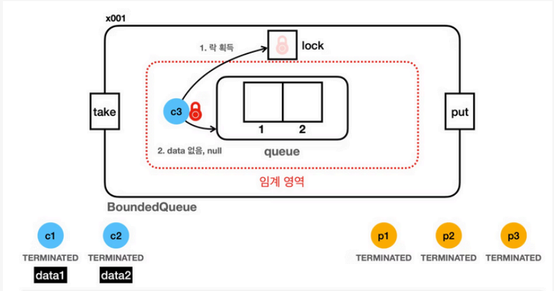
BoundedQueueV1 - 생산자 우선

```java
@Override
public synchronized void put(String data) {
    if (queue.size() == max) {
        log("[put] 큐가 가득 참, 버림: " + data);
        return;
    }
    queue.offer(data);
}

@Override
public synchronized String take() {

    if (queue.isEmpty()) {
        return null;
    }

    return queue.poll();
}
```



생산자(Producer)

- 큐에 데이터가 가득 차 있는 상황
- 생산자 스레드는 데이터를 버림

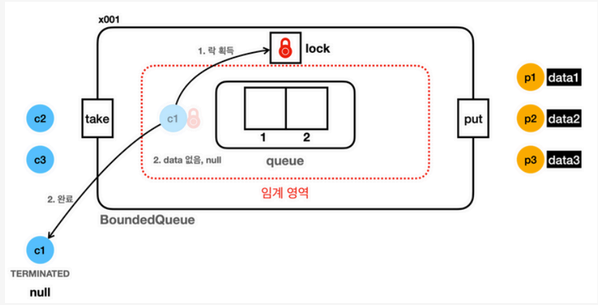


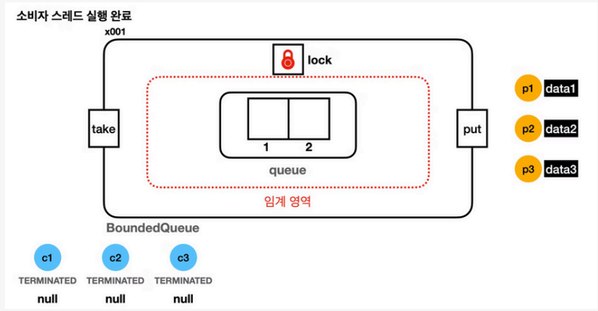
소비자(Consumer)

- 큐가 비어 있는 상황
- 소비자 스레드는 데이터를 가져오지 못함

BoundedQueueV1 - 소비자 우선

```java
@Override
public synchronized void put(String data) {
    if (queue.size() == max) {
        log("[put] 큐가 가득 참, 버림: " + data);
        return;
    }
    queue.offer(data);
}

@Override
public synchronized String take() {

    if (queue.isEmpty()) {
        return null;
    }

    return queue.poll();
}
```



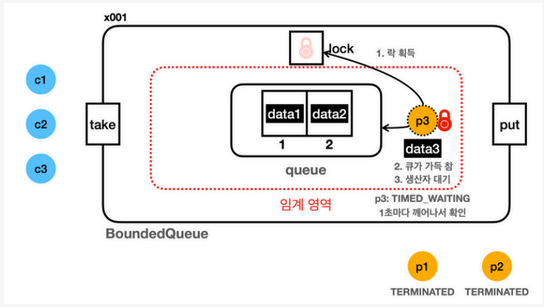


소비자(Consumer)

- 소비자 우선 실행 시 소비자 스레드 모두 데이터를 가져오지 못함
- c1, c2, c3는 데이터를 받지 못한다.(null을 받는다.)
- p3가 보관하는 data3은 버려진다.

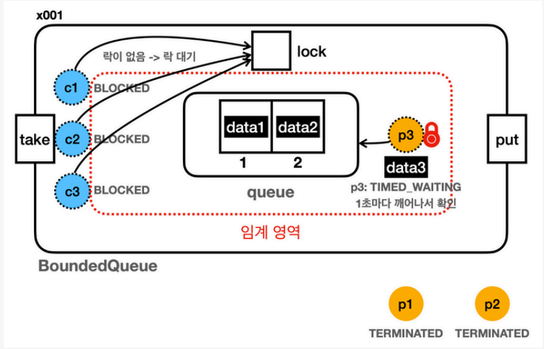
BoundedQueueV2 - 생산자 우선

```java
@Override
public synchronized void put(String data) {
    while (queue.size() == max) {
        log("[put] 큐가 가득 참, 생산자 대기");
        sleep(1000);
    }
    queue.offer(data);
}

@Override
public synchronized String take() {
    while (queue.isEmpty()) {
        log("[take] 큐에 데이터가 없음, 소비자 대기");
        sleep(1000);
    }
    return queue.poll();
}
```



생산자 스레드 실행 시작

- 생산자 스레드인 p3는 임계 영역에 들어가기 위해 먼저 락을 획득
- 큐가 가득 차서 p3는 sleep(1000)으로 대기한다.
- 문제는 p3가 락을 가지고 있는 상태에서 대기를 한다.

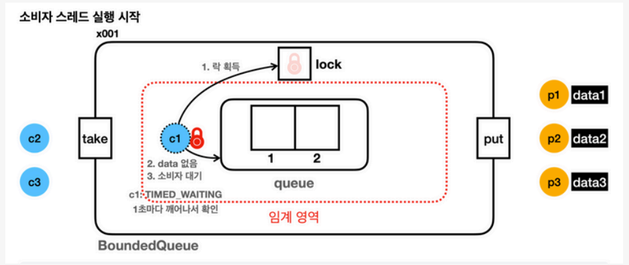


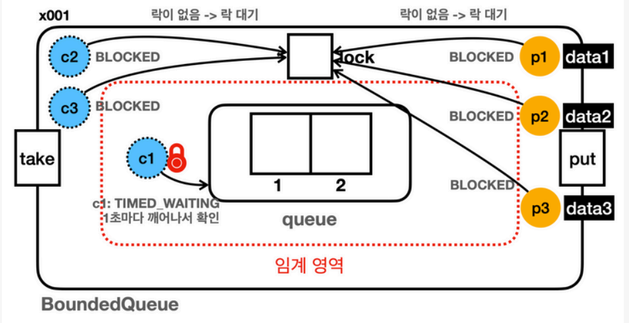
소비자 스레드 실행 시작

- 소비자 스레드가 임계 영역에 들어가기 위해 락을 획득하려 한다.
- 락이 없어 소비자 스레드 모두 BLOCKED 상태에 빠진다.
- 소비자 스레드 모두 무한 대기 상태에 빠진다.

BoundedQueueV2 - 소비자 우선

```java
@Override
public synchronized void put(String data) {
    while (queue.size() == max) {
        log("[put] 큐가 가득 참, 생산자 대기");
        sleep(1000);
    }
    queue.offer(data);
}

@Override
public synchronized String take() {
    while (queue.isEmpty()) {
        log("[take] 큐에 데이터가 없음, 소비자 대기");
        sleep(1000);
    }
    return queue.poll();
}
```





소비자 스레드 실행 시작

- 소비자 스레드 c1은 락을 획득한다.
- 큐가 비어 있어 c1은 sleep(1000)으로 대기한다.
- 문제는 c1이 락을 가지고 있는 상태에서 대기를 한다.
- 다른 소비자 스레드 c2, c3가 락을 획득하려하지만 락이 없어 BLOCKED 상태가 된다.

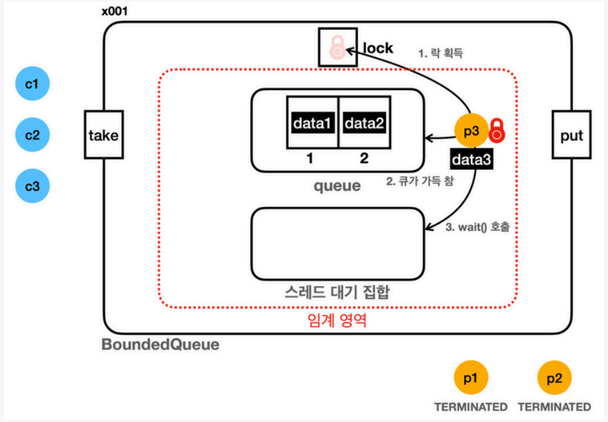
생산자 스레드 실행 시작

- 생산자 스레드들이 락을 획득하려 한다.
- 하지만 락이 없어 생산자 스레드 모두 BLOCKED 상태가 된다.
- c1은 반복문을 돌면서 계속 대기 하므로 c1을 제외한 다른 스레드 모두 무한 대기를 하게 된다.

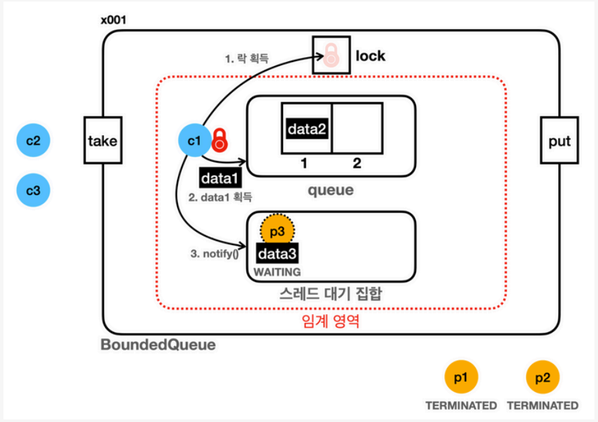
BoundedQueueV3 - 생산자 우선

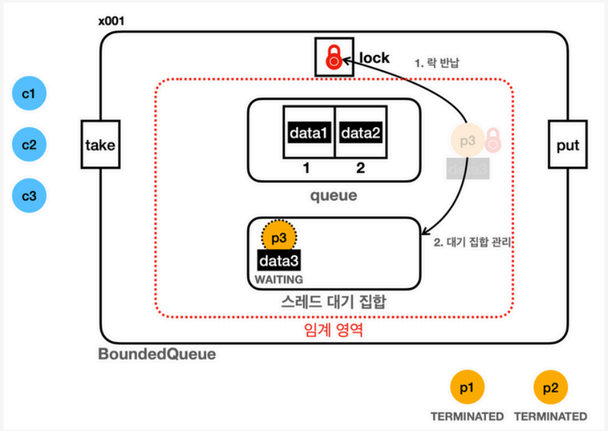
```java
@Override
public synchronized void put(String data) {
    while (queue.size() == max) {
        log("[put] 큐가 가득 참, 생산자 대기");
        try {
            wait(); // RUNNABLE -> WAITING, 락 반납
            log("[put] 생산자 깨어남");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    queue.offer(data);
    log("[put] 생산자 데이터 저장, notify() 호출");
    notify(); // 대기 스레드, WAIT -> BLOCKED
}

@Override
public synchronized String take() {
    while (queue.isEmpty()) {
        log("[take] 큐에 데이터가 없음, 소비자 대기");
        try {
            wait();
            log("[take] 소비자 깨어남");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    String data = queue.poll();
    log("[take] 소비자 데이터 획득, notify() 호출");
    notify(); // 대기 스레드, WAIT -> BLOCKED
    return data;
}
```

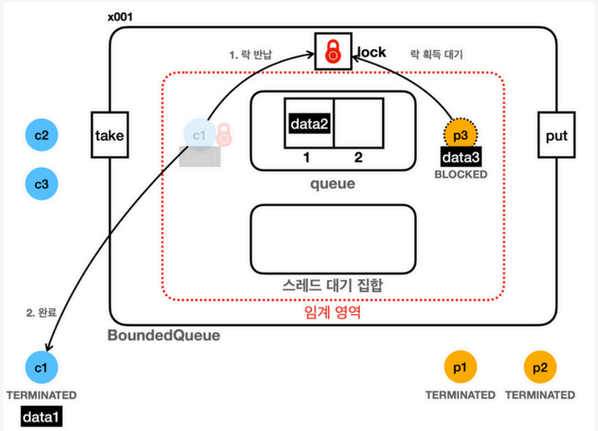
생산자 스레드 실행 시작



소비자 스레드 실행 시작



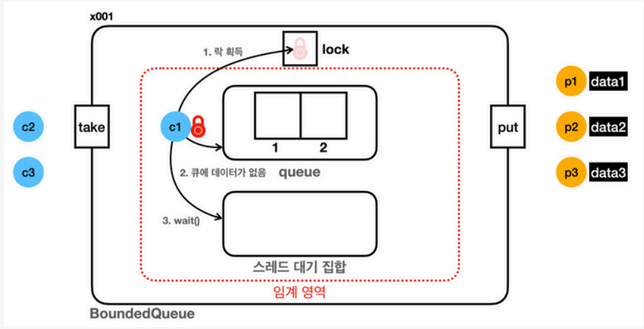


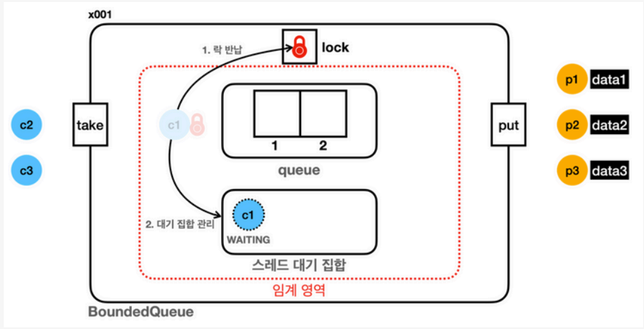


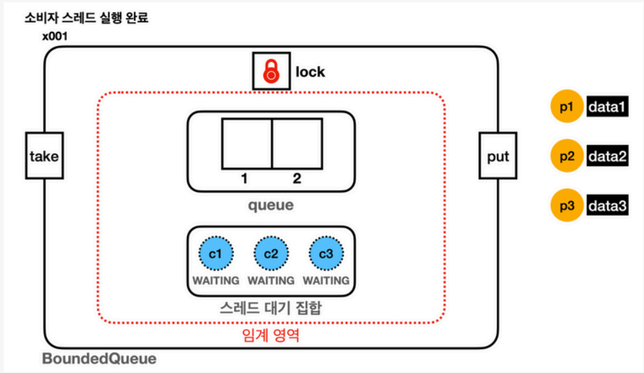
BoundedQueueV3 - 소비자 우선

```java
@Override
public synchronized void put(String data) {
    while (queue.size() == max) {
        log("[put] 큐가 가득 참, 생산자 대기");
        try {
            wait(); // RUNNABLE -> WAITING, 락 반납
            log("[put] 생산자 깨어남");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    queue.offer(data);
    log("[put] 생산자 데이터 저장, notify() 호출");
    notify(); // 대기 스레드, WAIT -> BLOCKED
}

@Override
public synchronized String take() {
    while (queue.isEmpty()) {
        log("[take] 큐에 데이터가 없음, 소비자 대기");
        try {
            wait();
            log("[take] 소비자 깨어남");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    String data = queue.poll();
    log("[take] 소비자 데이터 획득, notify() 호출");
    notify(); // 대기 스레드, WAIT -> BLOCKED
    return data;
}
```

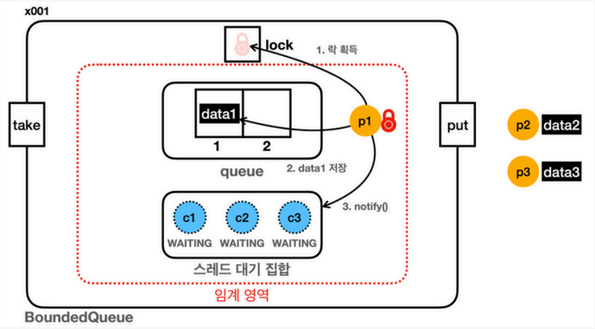
소비자 스레드 실행 시작

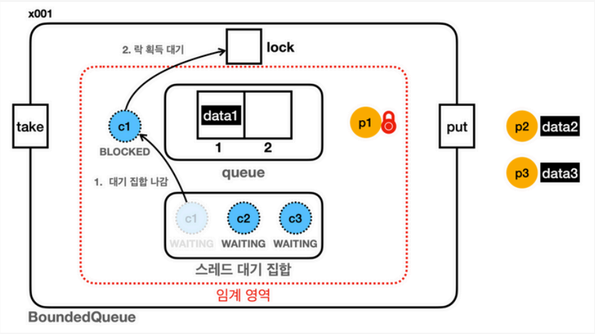


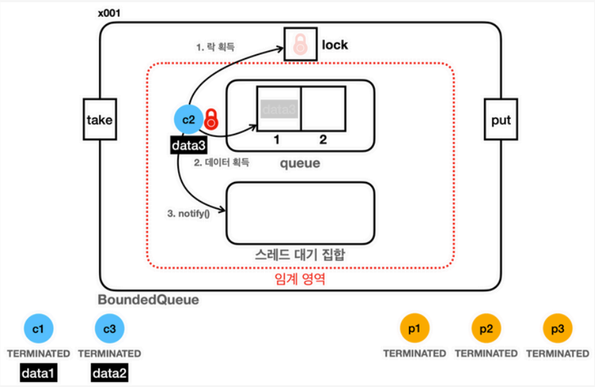




생산자 스레드 실행 시작

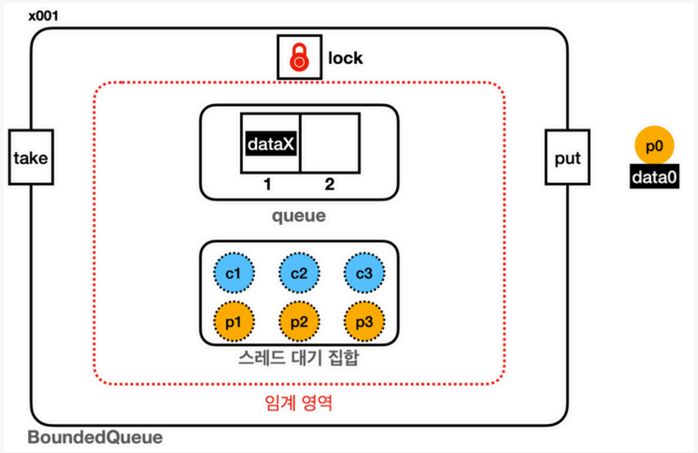




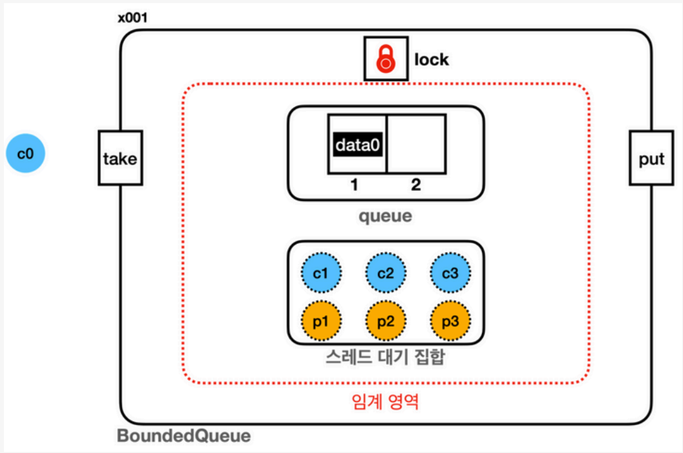


Object - wait, notify - 한계

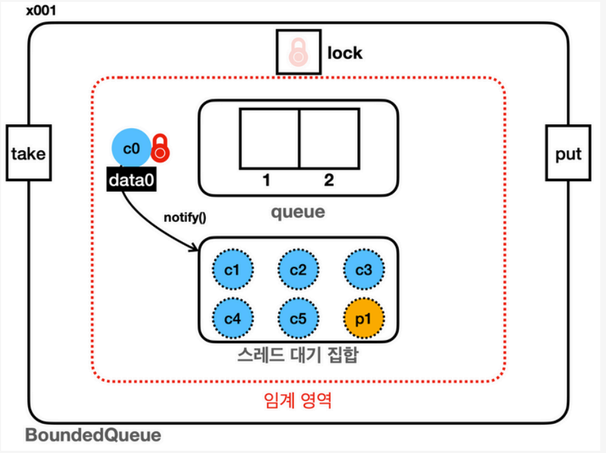
비효율 - 생산자 실행 예시

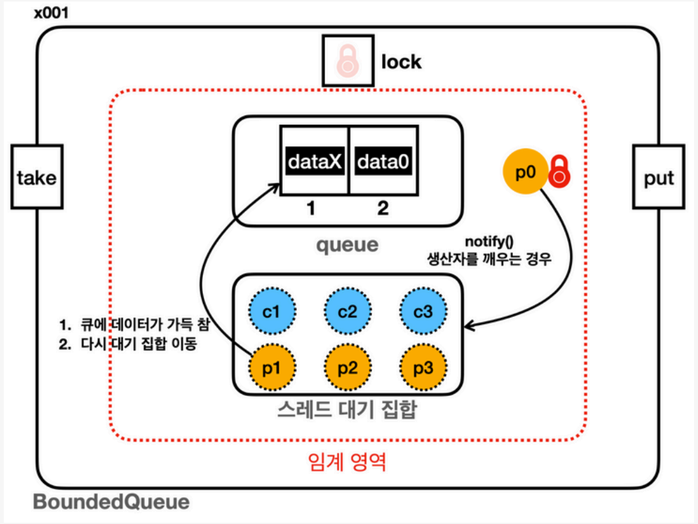


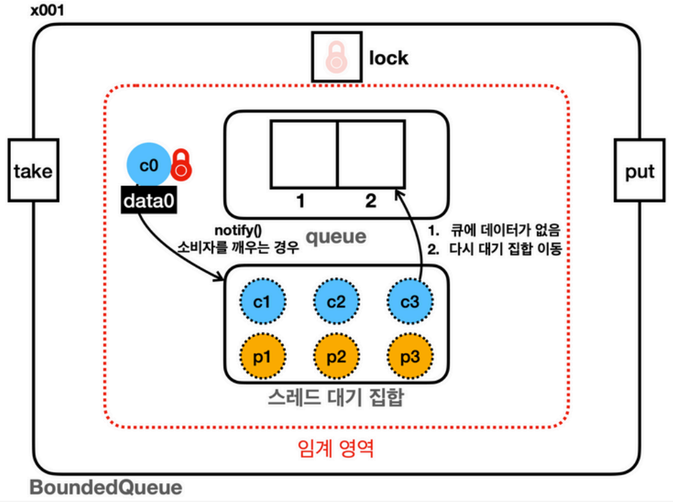
비효율 - 소비자 실행 예시

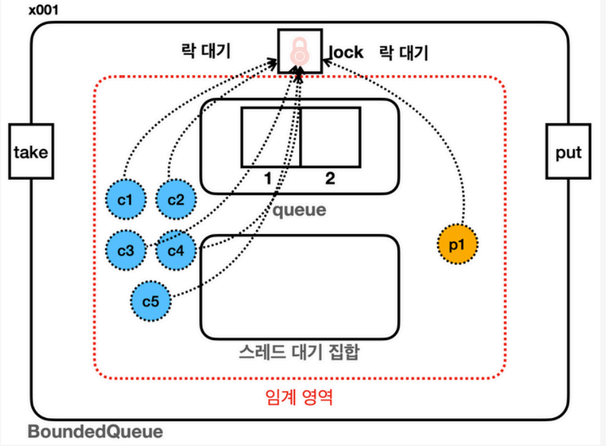


스레드 기아(thread starvation)





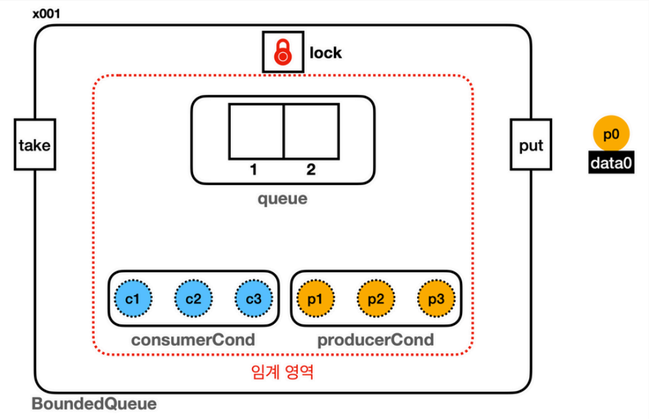




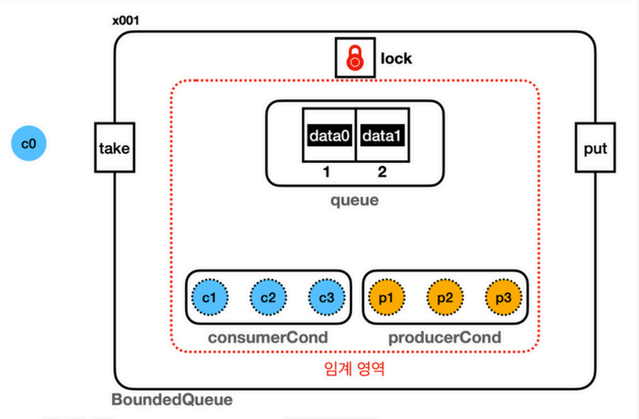
BoundedQueueV5

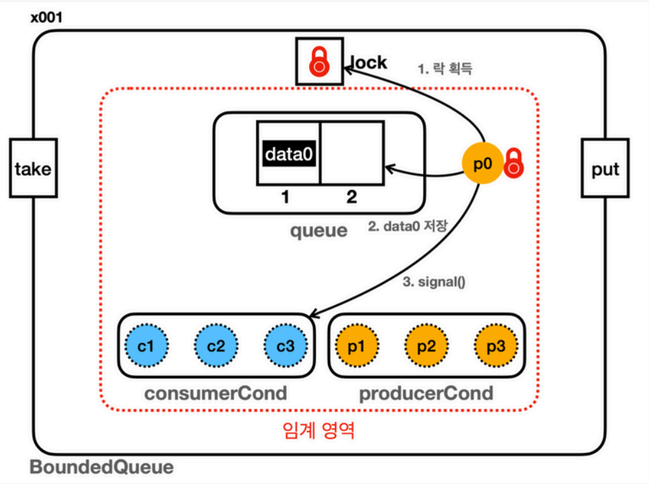
```java
public class BoundedQueueV5 implements BoundedQueue {

    // 락은 하나인데 대기공간을 분리할 수 있다.
    private final Lock lock = new ReentrantLock();
    private final Condition producerCond = lock.newCondition();
    private final Condition consumerCond = lock.newCondition();

    private final Queue<String> queue = new ArrayDeque<>();
    private final int max;

    public BoundedQueueV5(int max) {
        this.max = max;
    }

    @Override
    public void put(String data) {
        lock.lock();
        try {
            while (queue.size() == max) {
                log("[put] 큐가 가득 참, 생산자 대기");
                try {
                    producerCond.await();
                    log("[put] 생산자 깨어남");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            queue.offer(data);
            log("[put] 생산자 데이터 저장, consumerCond.signal() 호출");
            consumerCond.signal();
        } finally {
            lock.unlock();
        }

    }

    @Override
    public String take() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                log("[take] 큐에 데이터가 없음, 소비자 대기");
                try {
                    consumerCond.await();
                    log("[take] 소비자 깨어남");
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            String data = queue.poll();
            log("[take] 소비자 데이터 획득, producerCond.signal() 호출");
            producerCond.signal();
            return data;
        } finally {
            lock.unlock();
        }
    }
}
```

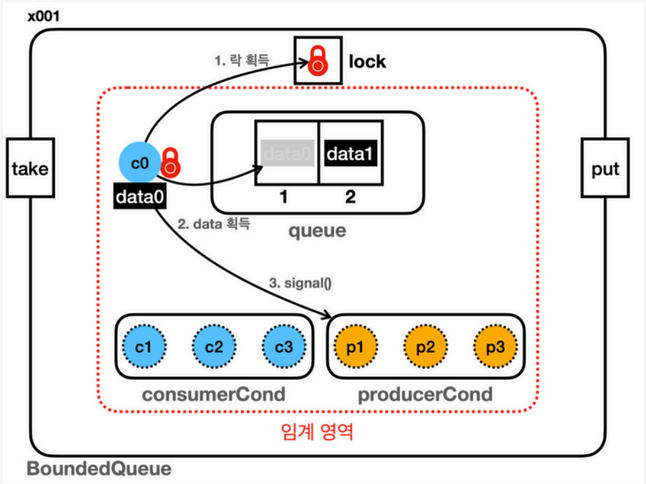
생산자 실행



소비자 실행

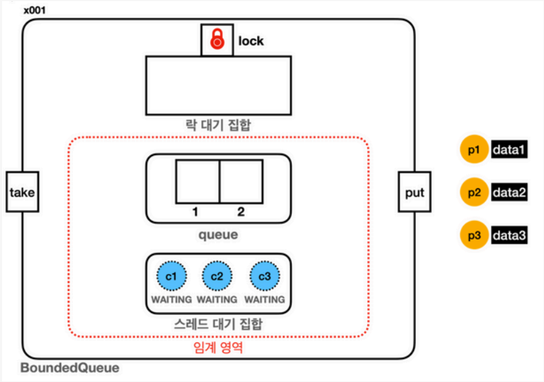




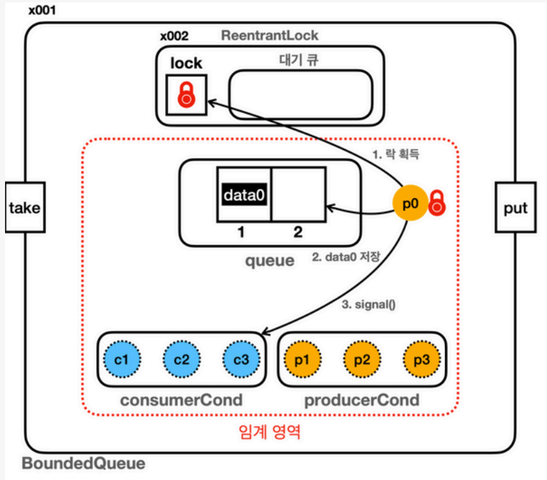


synchronized vs ReentrantLock

synchronized 대기



Lock(ReentrantLock)



비교 항목	synchronized	ReentrantLock
락 획득 및 반납 방식	자동으로 락을 획득하고 반납함	lock()과 unlock() 메서드를 호출하여 직접 관리
인터럽트 처리	지원하지 않음	인터럽트를 받을 시 대기 큐에서 나올 수 있음
사용 대기 공간	자바 객체 내부의 모니터 락, 락 대기 집합, 스레드 대기 집합 사용	ReentrantLock에서 관리하는 락, 락 대기 큐, condition 객체의 스레드 대기 공간 사용
락 대기 공간에서의 스레드 상태	락 대기 집합에서 BLOCKED 상태로 대기	락 대기 큐에서 WAITING 상태로 대기하며 인터럽트를 받으면 대기에서 나올 수 있음
스레드 대기 흐름	wait(), notify() 메서드를 사용하여 스레드 대기 공간 관리	await(), signal() 메서드를 사용하여 스레드 대기 공간 관리
스레드 대기 공간 수	하나의 스레드 대기 집합만 사용	condition 객체를 여러 개 생성하여 여러 대기 공간을 사용할 수 있음
스레드 대기 공간에서 깨어난 후 동작	스레드가 대기에서 깨어나도 바로 실행되지 않고, 락을 획득해야 실행 가능	스레드가 대기에서 깨어나도 바로 실행되지 않고, 락을 획득해야 실행 가능

BlockingQueue

```java
package java.util.concurrent;

public interface BlockingQueue<E> extends Queue<E> {

    boolean add(E e);
    boolean offer(E e);
    void put(E e) throws InterruptedException;
    boolean offer(E e, long timeout, TimeUnit unit) throws InterruptedException;

    E take() throws InterruptedException;
    E poll(long timeout, TimeUnit unit) throws InterruptedException;
    boolean remove(Object o);

    //...
}
```

- 데이터 추가 메서드: add(), offer(), put(), offer(타임아웃)
- 데이터 획득 메서드: take(), poll(타임아웃), remove(..)
- Queue를 상속 받는다. 큐를 상속 받았기 때문에 추가로 큐의 기능들도 사용할 수 있다.

BlockingQueue 인터페이스의 대표적인 구현체

- ArrayBlockingQueue: 배열 기반으로 구현되어 있고, 버퍼의 크기가 고정되어 있다.
- LinkedBlockingQueue: 링크 기반으로 구현되어 있고, 버퍼의 크기를 고정할 수도, 또는 무한하게 사용할 수도 있다.

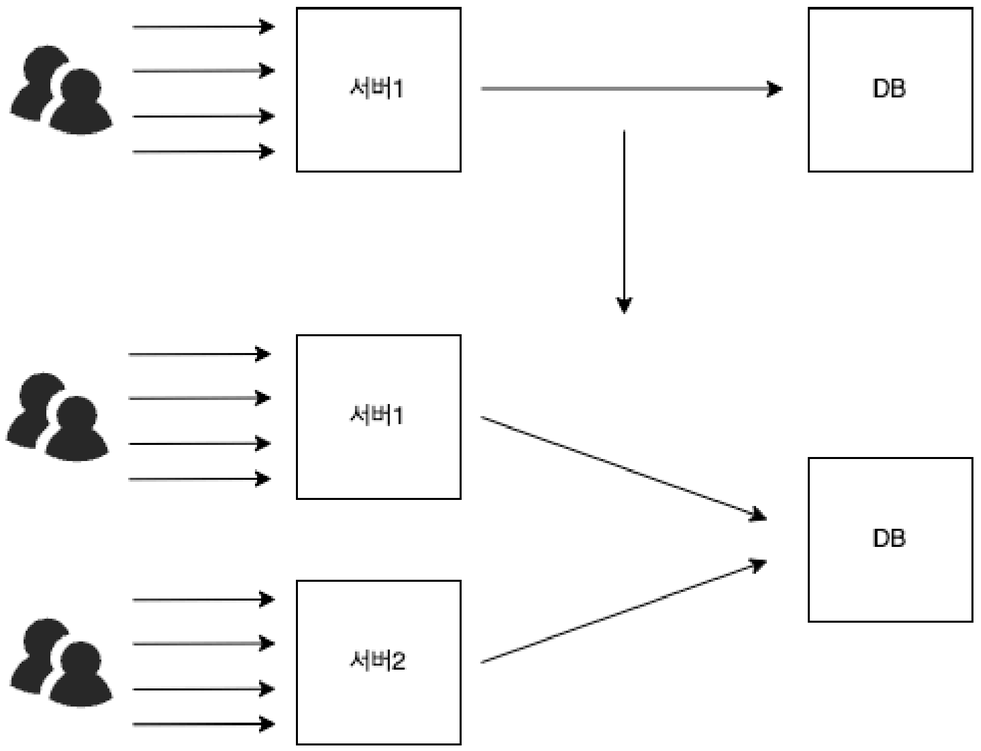
BlockingQueue의 다양한 기능 - 공식 API 문서

Operation	Throws Exception	Special Value	Blocks	Times Out
Insert(추가)	add(e)	offer(e)	put(e)	offer(e, time, ur
Remove(제거)	remove()	poll()	take()	poll(time, unit)
Examine(관찰)	element()	peek()	not applicable	not applicable

감사합니다.

Q&A

분산 서버 환경에서 DB에 있는 데이터를 어떻게 동시성 문제 없이 처리할 수 있을까요?



Q&A

사용자의 요청이 급증해 병목 현상이 발생하는 부분을 어떻게 처리 하면 좋을까요?