생산자가 생산자를 깨우고 소비자가 소비자를 깨우는 비효율을 해결하자

<해결방안>

생산자 스레드 대기집합과 소비자 스레드 대기집합으로 나누자

(Lock, ReentrantLock 이용)

아직 condition을 분리하지 않은 예제

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

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

- synchronized → Lock lock = new ReentrantLock()
- Condition : ReentrantLock을 사용하는 스레드가 대기하는 스레드 대기공간

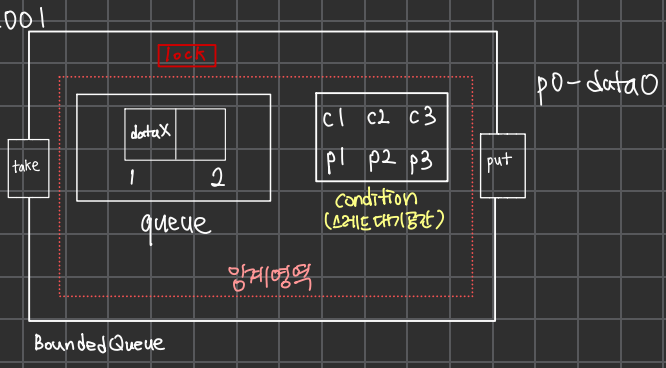
  lock.newCondition() : 스레드 대기 공간 생성 → 직접 만들어서 사용해야함

  condition.await() : 지정한 condition에 현재 스레드를 대기 상태로 보관

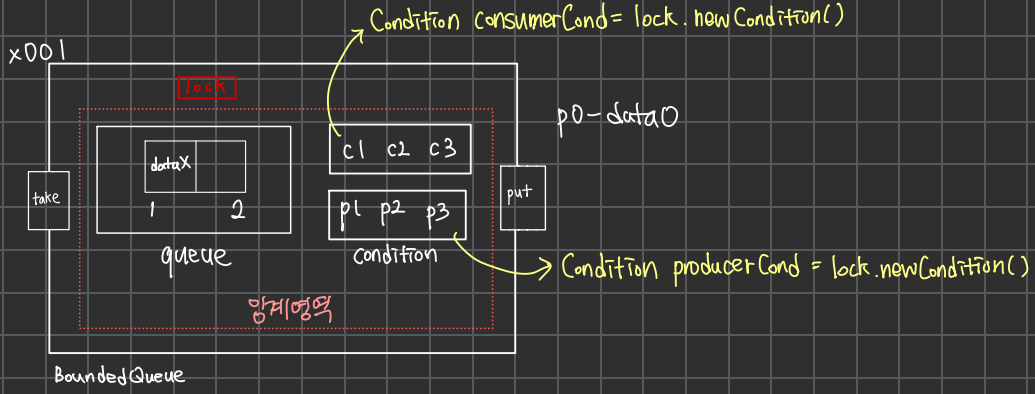
  ReentrantLock에서 획득한 락을 반납하고 대기 상태로 condition에 보관

  condition.signal() : 지정한 condition에서 대기중인 스레드를 깨운다.

  깨어난 스레드는 condition에서 빠져나온다.



생산자와 소비자 Condition 분리



- put(data) → producerCond.await(), consumerCond.signal()
- take() → consumerCond.await(), producerCond.signal()

Object.notify()	Condition.signal()
· 대기중인 임의의 스레드 하나를 깨움 · synchronized 블록내에서 모니터락을 가진 스레드가 호출	· 일반적으로 FIFO 순으로 하나를 깨움 (Condition은 Queue사용) · ReentrantLock을 가진 스레드가 호출

<스레드의 대기>

Synchronized 대기

Blocked 상태의 스레드를 락 대기 장소에서 관리함

① 락 획득 대기
BLOCKED, 락이 없으면 대기
다른 스레드가 synchronized를 빠져나갈때 대기가 풀리며 락 획득을 시도함

② wait() 대기
WAITING 상태로 대기
wait() 호출하면 스레드 대기 장소에서 대기
다른 스레드가 notify() 호출하면 빠져나옴

스레드 대기 장소에서 빠져나온뒤, Lock 획득시도 있다면 Blocked 상태로 락 대기장소 추가, 있다면 락 획득후 임계영역에서 작업수행

ReentrantLock 대기

① ReentrantLock 락 획득대기
- ReentrantLock의 대기 큐에서 관리
- waiting 상태로 대기
- lock.lock() 호출시 락이 없으면 대기
- 다른 스레드가 lock.unlock()을 호출했을때 대기가 풀리며 락 획득 시도, 락 획득시 대기 큐에서 빠져나감

② await() 대기
- condition.await() 호출시 condition 객체의 스레드 대기 공간에서 관리, waiting 상태로 대기
- 다른 스레드에서 condition.signal()을 호출 했을때 condition 객체의 스레드 대기 공간에서 빠져나감

BlockingQueue (java.util.concurrent.BlockingQueue)

: 스레드 관점에서 보면 큐가 특정 조건이 만족될 때까지 스레드의 작업을 차단

데이터 추가 차단 : 큐가 가득차면 데이터가 put()을 시도하는 스레드는 공간이 생길 때까지 차단

데이터 획득 차단 : 큐가 비어있으면 take()을 시도하는 스레드는 큐에 데이터가 들어올 때까지 차단

```
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);

    //...
}
```

데이터추가 메서드

데이터 획득 메서드

구현체

- ArrayBlockingQueue : 배열기반, 버퍼 크기 고정
- LinkedBlockingQueue : 링크 기반, 버퍼 크기고정 or 무한정

Condition notEmpty : 소비자 스레드 대기

Condition notFull : 생산자 스레드 대기

put(data) : lock.lockInterruptibly(), 락찾아면 await() → enqueue() 호출

enqueue(data) : 데이터추가후 notEmpty.signal()

BlockingQueue가 제공하는 메서드

Operation	Throw Exception	Special value	Blocks	Time out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	-	-
	대기시 예외	즉시 반환	대기	시간 대기