섹션 9. 생산자 소비자 문제1

여러 스레드가 동시에 데이터를 생산하고 소비하는 상황

- 생산자(Producer) : 데이터를 생산
- 소비자(Consumer) : 생성된 데이터 사용
- 버퍼(Buffer) : 생산자가 생성한 데이터를 일시적으로 저장하는 공간

문제 상황

- 생산자가 너무 빠를 때 : 버퍼가 가득 차버려 데이터를 넣을 수 없는 경우 → 생산자는 버퍼가 빌 때 까지 대기해야 함
- 소비자가 너무 빠를 때 : 버퍼가 비어서 소비할 데이터가 없을 경우 → 소비자는 버퍼에 데이터가 들어올 때까지 대기해야 함

위 두 문제를 재정의 하면 다음과 같다

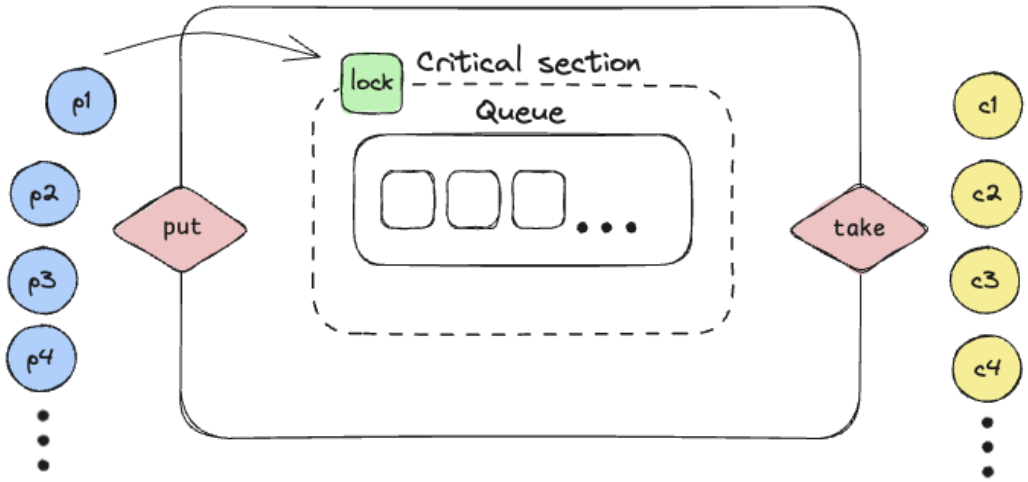
- 생산자 소비자 문제(producer-consumer problem) : 생산자와 소비자가 특정 자원을 함께 생산하고, 소비하면서 발생하는 문제
- 한정된 버퍼 문제(bounded-buffer problem) : 중간에 위치한 버퍼의 크기가 한정되어 발생하는 문제

생산자 소비자 문제 예제 - 생산자 먼저

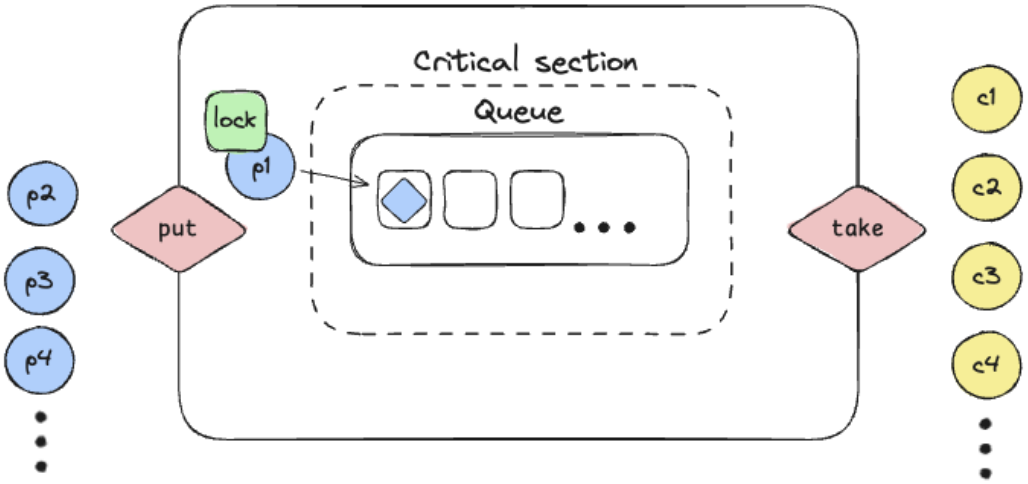
이 예제는 큐의 최대 갯수와 생산자, 소비자 중 어느 스레드를 먼저 실행할지 정해야 한다.

이 예제는 최대 갯수는 사용자가 지정하는 n개, 생산자 먼저 실행하는 예제이다.

다음과 같이 여러 생산자와 소비자, 최대 공간이 n개인 큐가 있다고 하면

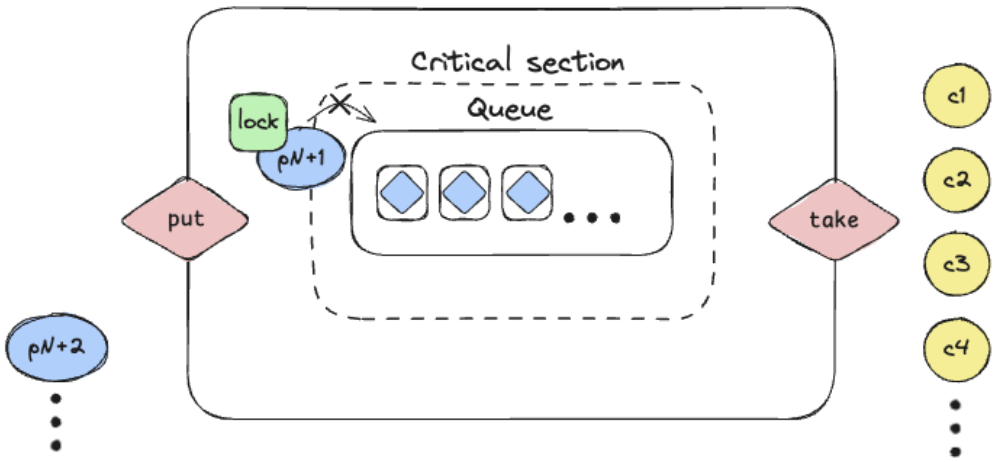


생산자는 임계구역으로 설정된 큐에 데이터를 넣기 위해 lock을 획득해야 한다.

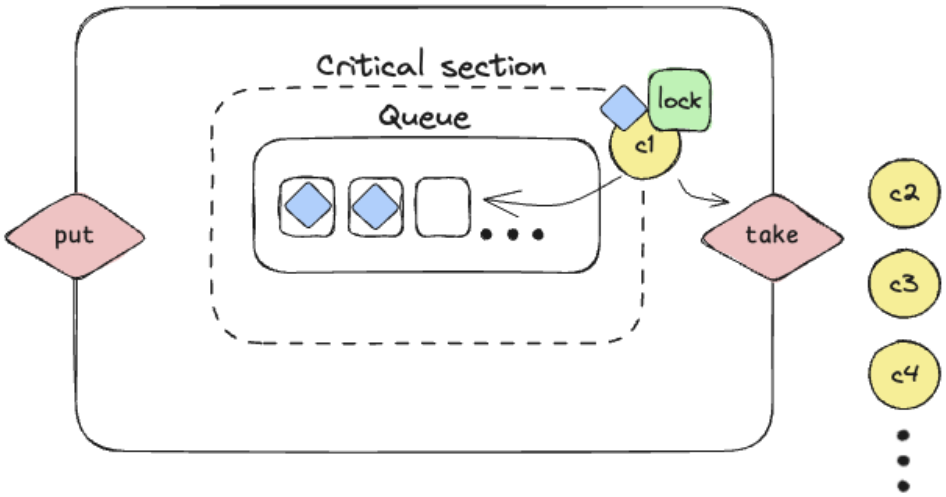


락을 획득한 생산자는 큐에 데이터를 넣고 락을 반납한 후 종료된다.

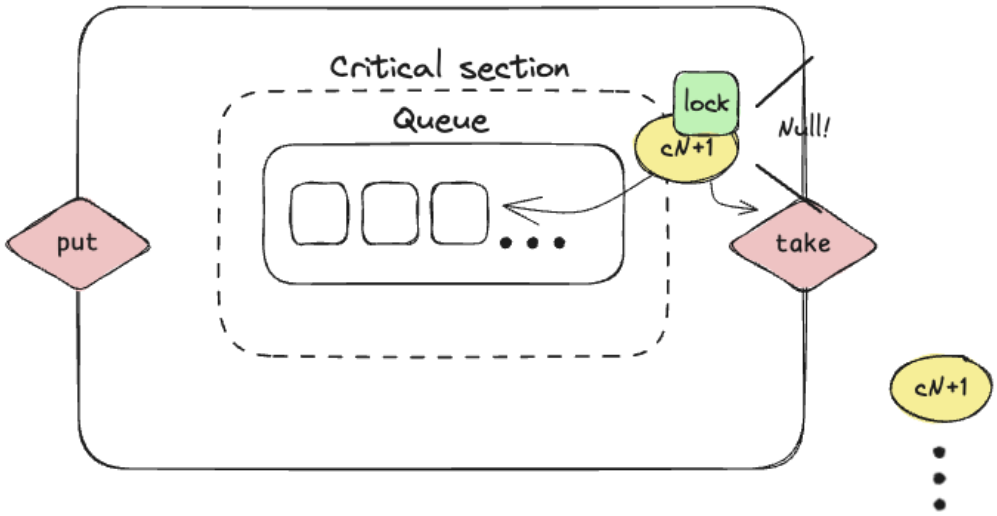
이 과정을 반복하면 큐의 공간이 언젠간 꽉 차게 된다.



큐의 공간보다 생산자가 더 많아 데이터큐가 가득 차있는 경우, 해당 데이터를 큐에 추가하지 못하고 버리게 된다.



소비자 스레드도 마찬가지로 임계구역을 통과하기 위해 lock을 획득하고 Queue에 있는 데이터를 소비한다.



마찬가지로 큐의 데이터보다 더 많은 소비자가 큐에 접근하게 된다면 큐에 데이터는 이미 모두 소비되었기 때문에 데이터를 획득할 수 없어 null을 반환한다.

생산자 소비자 문제 예제 - 소비자 먼저

소비자를 먼저 실행한 경우는 생산자와 반대로

- 소비자가 비어있는 큐에 생산자보다 먼저 접근을 한다.
- 큐는 아직 아무 데이터도 없기 때문에 모든 소비자가 전부 null을 반환한다.
- 이후 생산자가 데이터를 큐에 넣기 위해 접근한다.
- 큐의 데이터 보다 많은 번째의 생산자 부터는 모두 데이터를 넣지 못하고 버리게 된다.

문제점

- 버퍼가 가득 찬 경우 : 소비자가 큐의 데이터를 가져갈 때 까지 생산자가 대기하지 못하고 데이터를 버리게 되는 점
- 버퍼가 빈 경우 : 생산자가 큐의 데이터를 채울 때 까지 대기하지 못하고 null을 반환 해버리는 점

이러한 문제를 해결할 방안으로 일단 대기를 해보는 것도 좋을 것 같다.

생산자 소비자 문제 예제2 - 이번엔 조금 기다려보자

생산자가 데이터를 버리지 않는 대안

- 큐가 가득 차있다면, 큐에 빈 공간이 생길 때 까지 생산자는 대기한다.

- while() 문을 통해 지속적으로 큐에 빈 공간이 생겼는지 체크하고, 없다면 sleep()

소비자가 큐의 데이터를 대기하는 대안

- 큐에 데이터가 없다면, 큐에 데이터가 채워질 때 까지 대기한다.
  - 마찬가지로 while() 문을 통해 지속적으로 큐에 데이터가 있는지 체크 후, 없으면 sleep()

결과

생산자가 종료되지 않고 계속 수행되며, 소비자가 모두 `BLOCKED` 상태가 된다.

- 락을 가지고 대기를 하기 때문에 다른 스레드는 큐에 접근할 수 없다 (DeadLock)

따라서 `TIMED_WAITING` 상태에 걸린 생산자는 비어질리 없는 큐를 계속 대기하고, 소비자는 락을 획득하지 못하고 계속해서 `BLOCKED` 상태에서 대기한다.

Object - wait, notify - 예제3

위의 상황을 해결하기 위해 락을 가지고 있는 스레드가 락을 양보해준다면?

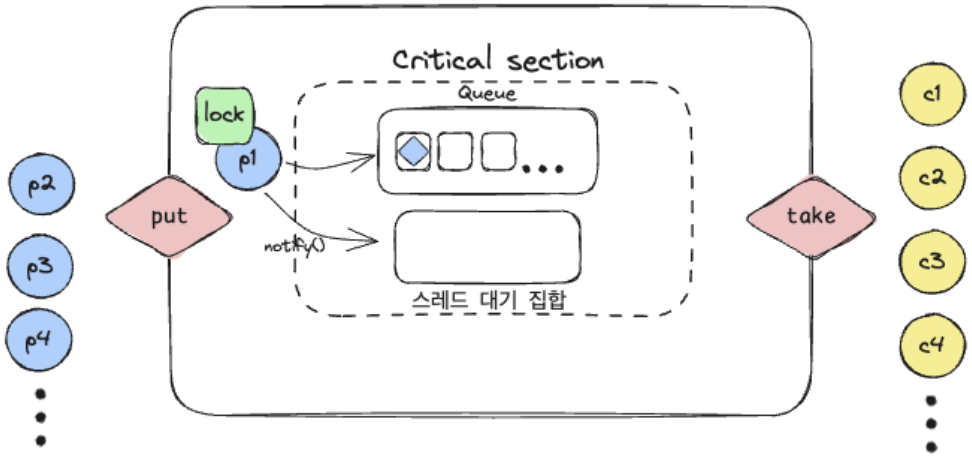
wait(), notify()

- Object.wait()
  - 현재 스레드가 가진 락을 반납 후 `WAITING`
  - 현재 스레드가 `synchronized` 블록이나 메서드에서 락을 소유하고 있을 때만 호출 가능
  - 다른 스레드가 notify() or notifyAll()을 호출할 때 까지 대기
- Object.notify()
  - 대기 중인 스레드 중 하나를 깨운다(`RUNNABLE`)
  - `synchronized` 블록이나 메서드에서 호출되어야 함, 깨워진 스레드는 락을 다시 획득할 기회를 얻음
- Object.notifyAll()
  - 대기 중인 모든 스레드를 깨운다
  - `synchronized` 블록이나 메서드에서 호출되어야 함, 모든 대기 중인 스레드가 락을 다시 획득할 기회를 얻음

스레드 대기 집합(wait set)

- `synchronized` 임계 영역 안에서 Object.wati()를 호출하면 스레드는 `WAITING` 상태에 들어감
- 대기 상태를 관리하기 위해 대기 집합을 가짐
- 모든 객체는 모니터락과 대기 집합을 갖는다.

대기를 해야 할 상황에서 락을 양보해주는 예제 - 생산자 우선

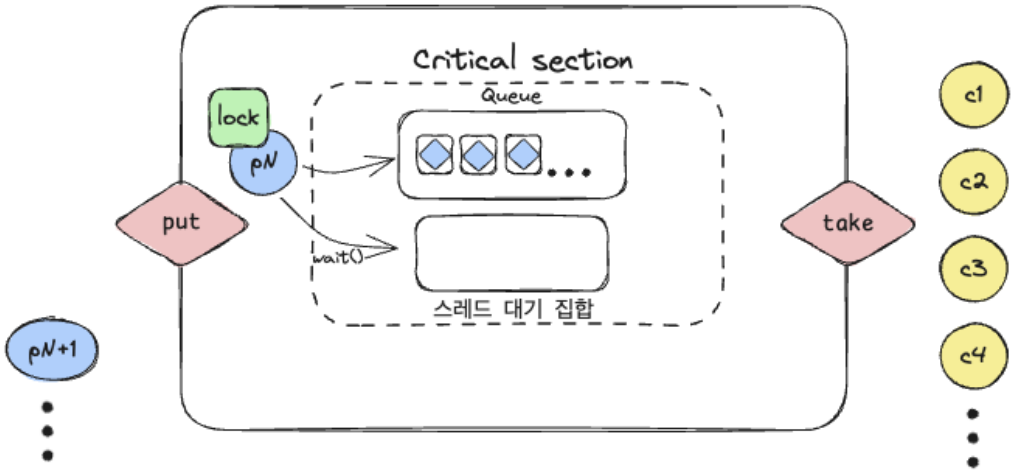


생산자는 락을 획득하고 데이터를 큐에 저장한다.

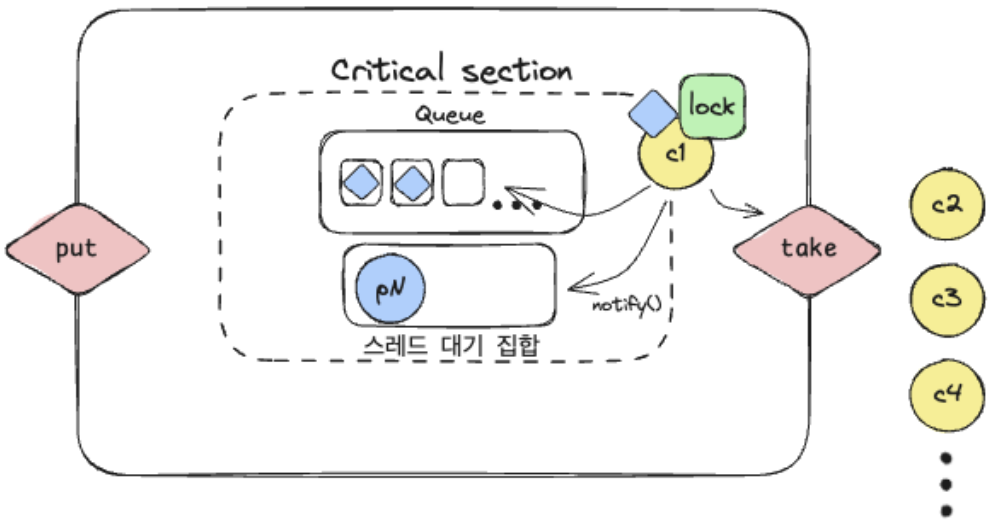
큐에 데이터가 추가되었으므로 notify()를 통해 스레드 대기 집합에서 스레드 깨우기를 시도한다

대기 집합에 아무도 없으므로 그냥 넘어가진다.

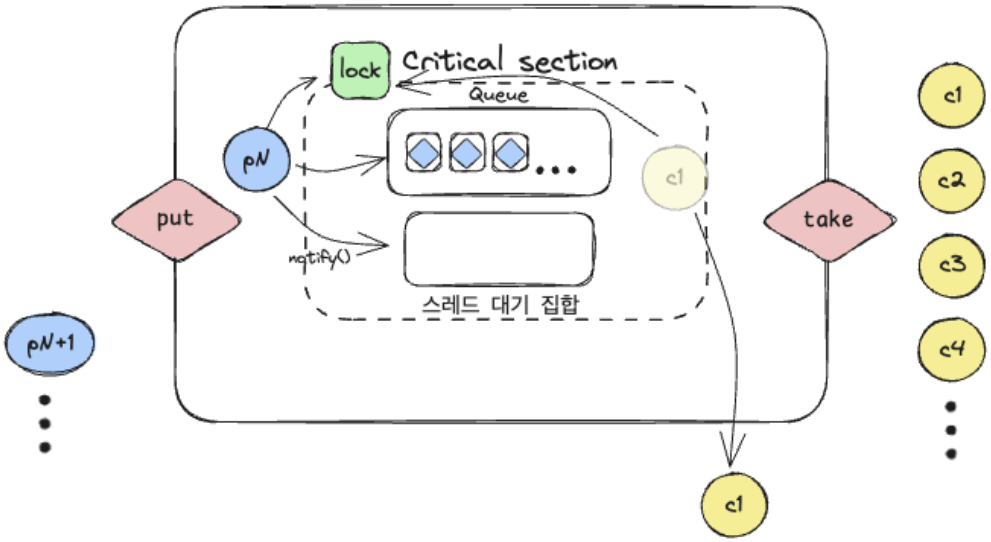
이 과정이 큐에 데이터가 꽉찰 때 까지 반복



큐에 데이터가 꽉차면 wait()를 호출해 스레드를 `WAITING` 상태로 변경하여 큐가 비워질 때 까지 락을 반납하고 대기한다.



소비자 스레드가 실행되면 소비자는 락을 가지고 큐에 접근해 데이터를 가져가고, 큐에 공간이 생겼기 때문에 스레드 대기 집합에 notify() 메서드를 통해 스레드를 깨운다.



소비자 스레드는 생산자를 깨운 뒤, 락을 반납하고 종료되고, 락을 획득할 기회를 얻은 생산자 스레드는 락을 획득해 데이터를 넣고 스레드 대기 집합의 스레드를 깨운다 (아무도 없어서 아무일 안일어남)

이후 순차적으로 소비자 스레드들이 실행 되며 큐의 데이터들이 전부 소비되어 종료된다 → 정상 종료

대기를 해야 할 상황에서 락을 양보해주는 예제 - 소비자 우선

소비자 우선인 경우 위의 상황과 반대로

- 소비자 스레드가 먼저 큐에 접근한다.

- 큐가 비어있으므로 스레드 대기 집합에 `WAITING` 상태로 대기한다.
- 모든 소비자 스레드가 대기 집합에 대기상태로 대기한다.

- 생산자 스레드가 시작되면 큐에 데이터를 저장하고 스레드 대기 집합에 대기중인 스레드 하나를 깨운다.
- 대기 중이던 소비자 스레드중 하나가 깨어 락을 획득하기 위해 `BLOCKED` 상태로 대기한다.
- 락을 획득하면 소비자 스레드는 데이터를 획득하고 큐에 공간이 났다는 것을 알려 대기중인 스레드 하나를 깨운다.
- 하지만 대기중인 스레드는 모두 소비자 스레드이므로 큐를 확인하고 다시 `WAITING` 상태에 들어간다.
- 반복

정리

결과적으로 생산자와 소비자는 모두 정상적으로 동작을 하지만, 소비자가 잘못 일어나 CPU자원만 소모하고 다시 대기하는 경우가 발생 → 비효율적

생산자 → 소비자, 소비자 → 생산자 만 깨울 수 있다면 효율적일 것 같지만 notify()는 불가능

Object - wait, notify 한계

Object.wait(), Object.notify()는 대기 집합 하나에 두 종류의 스레드를 모두 관리

- 원치 않은 스레드(같은 종류의 스레드)가 깨워져 비효율 적인 상황 발생

스레드 기아(thread starvation)

notify()는 어떤 스레드가 깨는 지 알 수 없기 때문에 기아 문제가 발생할 수 있다.

- 대기중인 스레드 중 생산자 스레드가 무수히 많고 소비자 스레드가 하나 있다고 가정해보자
  - 생산자 스레드 하나가 완료되고 소비자 스레드를 깨우기 위해 대기 집합을 깨운다면
  - 1/n 의 경쟁을 뚫고 소비자 스레드가 깨어나기에는 어렵다
  - 경쟁을 뚫지 못하고 계속해서 대기하면 기아 문제 발생

이를 해결하기 위해 notifyAll()을 사용해서 전부 깨워 결국에는 기회를 주는 방법도 있다.

→ 비효율을 막지는 못함