Operating System
Lab 2

이름 : 박동학, 홍승기
학번 : 32151648, 32155068
단국대학교 소프트웨어학과

목차

I. 분석

1. 프로젝트 분석

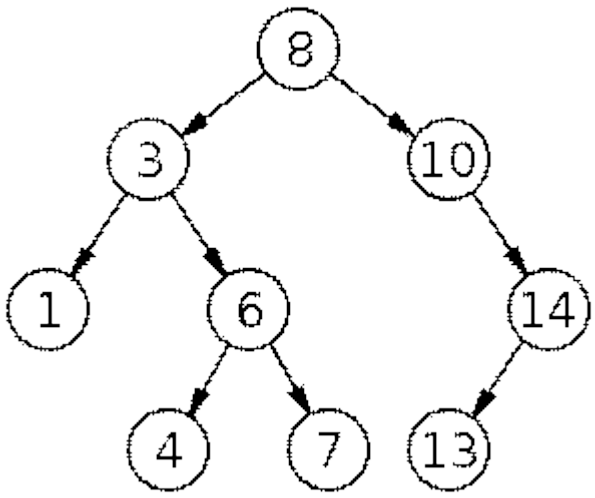
이번 프로젝트는 이진 탐색 트리 구현(Binary Search Tree)과 Thread 를 사용해 이진 탐색 트리의 탐색, 삽입, 삭제 시 병행성을 유지하면서 충돌이 일어나지 않게 Lock mechanism 을 사용해서 여러 Thread 가 원활하게 수행되게 하는 Thread Safe Binary Search Tree Project 이다.

아래에는 이번 프로젝트에 있어서 중심이 되는 이론적 배경을 서술한다.

1) Binary Search Tree

이진 탐색 트리(binary search tree)는 이진 트리의 일종으로서 부모 노드를 기준으로 왼쪽에는 자신보다 작은 데이터를 오른쪽에는 자신보다 큰 데이터를 가지는 자식 노드를 가지는 자료구조이다.

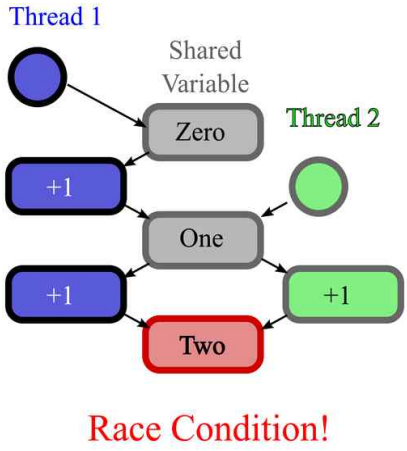
데이터의 삽입, 삭제, 탐색 등의 연산이 있으며 이는 모두 $O(\log n)$의 시간 복잡도를 가진다.



2) Race Condition

Race Condition 이란 Thread 를 사용하는 상황에 있어서 경쟁상태가 발생되는 것을 말한다.

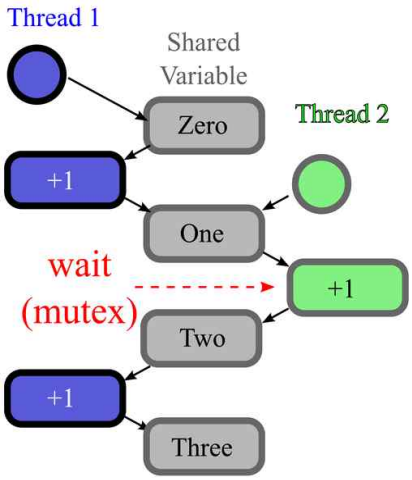
여기서 경쟁상태가 발생된다는 것은 Thread 는 Process 와는 다르게 데이터를 공유하기 때문에 발생한다. 여러 Thread 가 동시에 동작하면 의도치 않게 같은 데이터를 두고 서로 수정을 하려고 해서 원하지 않는 결과가 나오는 것을 의미한다.



3) Critical Section

(2)에서와 같이 경쟁상태가 발생할 수 있는 부분을 Critical Section 즉 임계영역이라고 한다. 임계영역은 공유자원에서 발생하게 되며 이러한 공유자원에 한 순간에 한 개의 Thread 만 수정을 할 수 있게 상호배제를 보장해 주어야 올바른 결과 값을 얻을 수 있다.

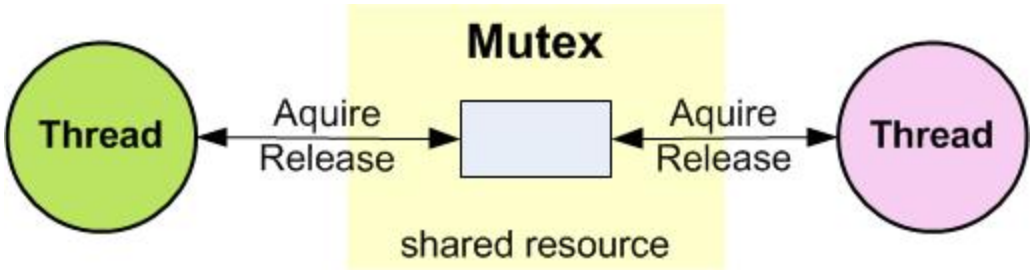
아래 그림에서는 중간의 회색 Shared Variable 이 공유자원이다.



4) Mutex Lock

Mutex Lock 이란 Mutual exclusive Lock 을 의미한다. 즉 상호배제를 보장해 준다는 것이다. 상호배제는 위에서 언급했듯이 원하는 결과를 얻기 위해서는 Race Condition 을 방지해야 하며 이를 위해서는 임계영역에 Mutex 를 통해서 한 순간에 한 개의 Thread 만 접근 할 수 있도록 해야한다.

이와 비슷하게 동기화라는 개념 또한 중요한데 동기화란 순서관계를 보장해 준다는 것이다.



5) Fine-Grained Lock and Coarse-Grained Lock

이러한 상호배제를 보장해 주는 것은 mutex lock 을 이용할 수 있다. 하지만 lock 을 수행하는 범위에 따라서 context switch 나 대기 시간에 따라 성능이 차이가 난다.

여기서 Coarse Grained 는 Lock 를 크게 잡는 것이고 Fine Grained 는 lock 를 작게 잡는 것을 의미한다. 자세한 설계 방향은 아래 설계방향 설정에서 설명한다.

2. 설계 방향 설정

1. BST 구현

우리가 자료구조 수업과 알고리즘 수업때 배운 이진탐색트리를 구현한다.

이진탐색트리의 기본 원칙에 맞춰서 설계를 구성하고 프로젝트 파일에 만들어야 할 코드에 맞춰서 이진탐색트리를 구성을 시도한다, 우리가 기본적으로 배웠던 구성과는 조금 다른 부분이 있어서 구조를 파악하고 설계에 접근 하는 방법을 사용

2. Lock 적용

기본적으로 싱글쓰레드의 경우 lock 을 사용할 필요가 없다. 하지만 쓰레드가 2 개 이상인 멀티쓰레드 환경에 lock 을 사용해보고 또 lock 을 사용하지 않았을 때는 어떤 결과가 발생하는지 확인 해 볼 수 있을 것이다.

Coarse-grained

lock 을 접근하는 함수 전체적으로 감싸서 한 개의 쓰레드가 일을 다 마칠 때까지 다른 쓰레드의 접근을 원천적으로 차단을 시키는 방식으로 설계를 구현해 볼 예정이다. lock 의 위치에 따라 실행시간이 얼만큼 차이가 나는지 확인 해 볼수 있을 것이다. 쓰레드가 많아지면 기아상태에 빠질수도 있는걸 유의하고 설계를 구현 시켜야 한다.

Fine-grained

lock 을 사용함에 있어서 lock 을 거는 위치를 노드에 관한 부분에 대해서 세세한 부분으로 다 나눠 모두 한부분 한부분 lock 을 걸어 보는 방식으로 설계를 시도하기로 한다. lock 의 개수가 많아지면서 프로그램에 context-switch overhead 의 위험이 증가함므로 무분별하게 lock 을 많이 사용하는 방식은 자제하도록 해야한다. 또한 lock 이 많아지면서 프로그래머가 lock 을 알맞은 위치에 사용하지 않으면 deadlock(교착상태)에 빠질 수 있으므로 이점도 유의하면서 fine-grained lock 을 구현을 시도해 보겠다.

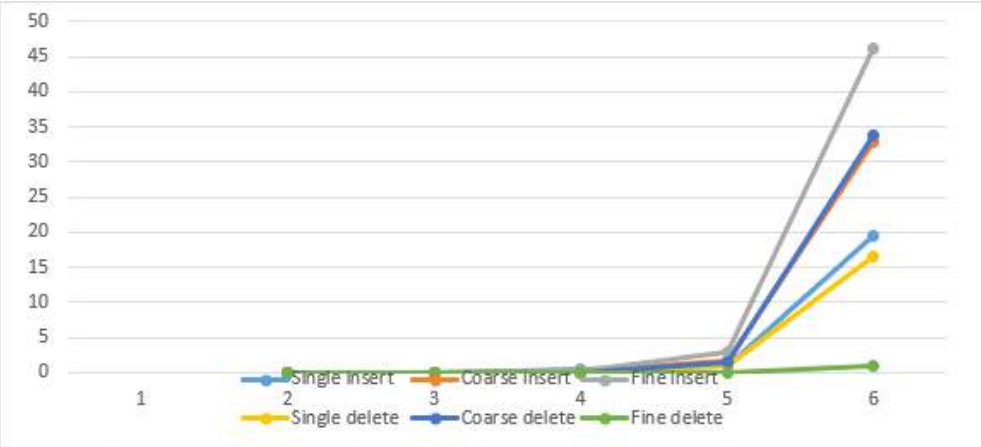
III. 프로젝트

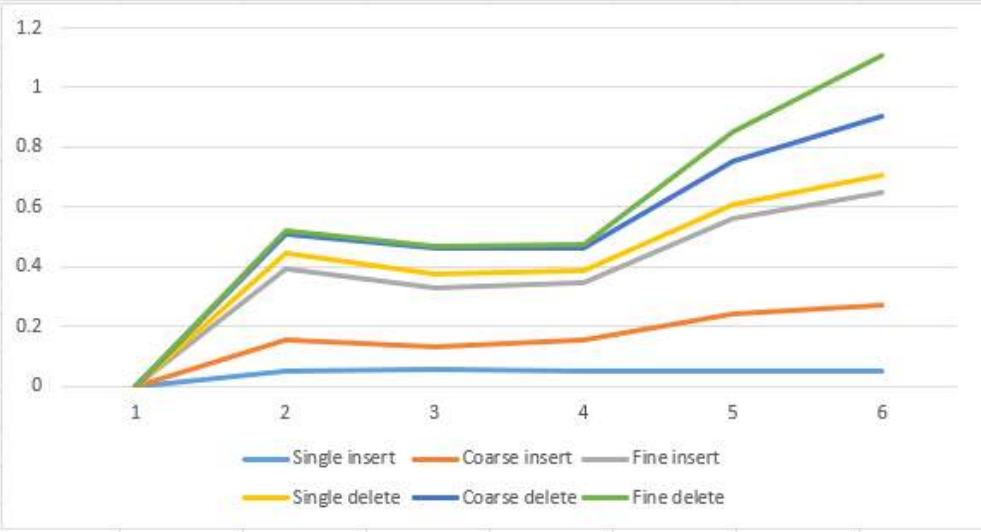
1. 프로젝트 결과 분석

```
=====  Multi thread single thread BST  insert experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 20.114984 seconds


 BST inorder iteration result :
    total node count    : 10000000

=====  Multi thread coarse-grained BST  insert experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 32.006647 seconds


 BST inorder iteration result :
    total node count    : 10000000

=====  Multi thread fine-grained BST    insert experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 47.679908 seconds


 BST inorder iteration result :
    total node count    : 10000000
```

```
=====  Multi thread single thread BST  delete experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 16.048468 seconds


 BST inorder iteration result :
    total node count    : 10000000

=====  Multi thread coarse-grained BST  delete experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 30.251400 seconds


 BST inorder iteration result :
    total node count    : 10000000

=====  Multi thread fine-grained BST    delete experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 0.970276 seconds


 BST inorder iteration result :
    total node count    : 10000000
```

	노드 1000	노드 10000	노드 100000	노드 1000000	노드 10000000
thread =4개					
Single insert	0.000145	0.002072	0.42571	1.034401	19.477282
Coarse insert	0.000674	0.007002	0.113422	1.64581	32.925482
Fine insert	0.000565	0.017807	0.165932	2.935721	46.040992
Single delete	0.000121	0.001876	0.03853	0.829917	16.417006
Coarse delete	0.000562	0.006082	0.071975	1.354575	33.80872
Fine delete	0.000126	0.000874	0.008176	0.046057	1.014889
	thread 4	thread 10	thread 100	thread 1000	thread 20000
Node 100000					
Single insert	0.052009	0.05358	0.051471	0.046882	0.051361
Coarse insert	0.099602	0.08015	0.10572	0.19679	0.21696
Fine insert	0.242483	0.194151	0.18737	0.314669	0.377594
Single delete	0.053025	0.049327	0.044359	0.046947	0.061434
Coarse delete	0.062854	0.083562	0.074663	0.147175	0.19851
Fine delete	0.010264	0.006095	0.008762	0.099076	0.202269





결과 분석

첫 번째 그래프 Thread 고정 Node 수 변경

- Thread 수를 고정하여 Node 수를 고정하여 수행해 본 결과 Node 수가 증가함에 따라 수행시간이 증가하였다.

두 번째 그래프 Node 고정 Thread 변경

- Node 수를 일정하게 하고 Thread 수를 변경하여 수행해 본 결과 일정 수준의 Thread 에서만 더 좋은 결과가 나오고 Thread 가 너무 많거나 적을 경우 수행시간이 증가한 것을 볼 수 있다. 본 과제의 수행에서는 Thread 의 수가 3,4 개 일 때 가장 좋은 결과를 보여 주었다.

전체적인 결과 분석

- Delete -fg 의 경우 다른 부분과는 다른 방식으로 구현하여 좋은 성능을 보였다. 이는 인터넷에서 찾은 알고리즘을 참고하여 구현하였기 때문에 다른 부분보다 좋은 성능을 보인 것으로 보인다.

- 이론적으로는 Coarse > single Thread > Fine 이라고 생각을 하고 구현을 했지만 실제로는 Delete 말고는 뒤죽박죽인 결과가 나왔다,

- 이러한 결과가 나온 이유는 lock 을 올바르게 위치시키지 못했기 때문이라고 생각한다.

IV. 논의

1. 고찰

– 프로젝트를 진행하면서 어려웠던 점 및 프로젝트 고찰

32151648 박동학

이번 과제는 학교를 다니면서 몇 번이나 만들어 봤던 Binary Search Tree 에 대한 과제였기 때문에 이전 과제인 스케줄링 보다는 훨씬 쉬울 것이라고 생각했다. 단순하게 BST 를 구현하고 Coarse Grained 는 Tree 전체를 Fine Grained 는 노드 하나씩을 Lock 을 걸어서 Concurrency 를 유지해 주면 된다고 생각했다. 하지만 실제로 구현을 함에 있어서 Lock 의 순서 관계에 따라서 다양한 오류가 발생하였고 이를 극복하는데 있어서 많은 시간과 노력이 들어갔다.

1. BST(Binary Search Tree) 구현에 있어서의 어려움

- 이미 이전 수업에서 몇 번 구현을 해보아서 손 쉽게 구현할 수 있을 거라고 생각했었다. 하지만 정해진 틀과 필요한 기능들을 충족시키면서 구현하는데 어려움이 있었다.
- 특히 처음에는 이후에 LOCK 을 걸기 편하게 구현을 해야겠다는 생각으로 처음부터 무리하게 포인터를 피하면서 작성하여서 나중에 오류를 찾는데 힘이 들었다.
- 또한 BST 를 처음부터 완전하게 구성하고 넘어갔어야하는데 빨리 LOCK 을 걸고 싶다는 생각에 변수명을 신중하게 선택하지 않았고 이는 나중에 코드를 이해하는데 어려움을 주었다.

2. 처음 LOCK 을 잡았을 때 여러 가지 오류들

- 처음 BST 를 구성하고 LOCK 을 하나씩 잡아가는데 있어서 대충 포인터를 사용하거나 변수를 수정하는 부분을 순서관계 없이 LOCK 의 특성에 대한 이해 없이 무분별적으로 사용하여 구성하였다.
- 그렇게 구성을 함에 있어서 처음 몇 번에 실행에는 성공한 것처럼 보였지만 반복적으로 실행 해보면 우연히 몇 번 성공한 것이라는 걸 알게 되었다.
- 이때 발생한 대표적인 오류는 Coarse Grained 에서 처음부터 끝까지 Lock 을 걸어서 발생하는 문제였다. **조건문을 통하여 return 하는 부분**에 당연히 unlock 을 해주어야

함에도 불구하고 이를 해주지 않아 lock 이 풀리지 않고 thread 들이 접근을 하지 못하는 문제가 발생하였다.

- 또한 순서 관계를 지키지 못해 여러 구문 오류들이 발생하였고 이를 GDB 를 통해서 위치를 파악하고 이를 수정하는 식으로 수정을 하였다.

- 이러한 오류들 뿐만 아니라 단순하게 lock 을 걸어 놓은 부분부터 풀릴 때 까지 기다린다는 생각으로 작성을 하였기 때문에 여러 Thread 가 들어감에 있어서 Node 수가 증가함에 있어서 많은 오류 들이 발생하였다. 예를 들자면 여러 번 **free** 를 시켜줘 생기는 double free 같은 것들이 있다.

3. Fine-Grained 에 있어서 노드 수 증가에 따른 문제점

- 삽입의 경우 생각해야 하는 경우가 적었기 때문에 비교적 수월하게 해결 할 수 있었지만 삭제에 있어서 FG 를 구현하는 것이 많은 문제가 되었다. 어디서 오류가 발생하는지 파악하는 것도 어려웠으며 어떤 순서를 지켜줘야 하는지도 문제가 되었다.

- 노드 하나마다 lock 를 시키면 우리가 구현한 코드에서는 p, pp, s, ps 같이 tree 를 따라 내려가는 4 개의 포인터에 대해서 모든 경우에 대해서 lock/unlock 을 고려해 주어야 했고 이 부분에서 많은 시간을 보내게 되었다.

- 여러 번의 수정을 통해 어느 정도 노드수에 대해서는 정상적으로 수행이 되었지만 노드의 수가 증가하고 트리의 크기가 커지면 thread 당 수행시간이 길어지고 처음 부분과 끝부분의 lock/unlock 관계의 고려 없이 눈에 보이고 나오는 결과를 수정하기 급급했기 때문에 thread 수와 node 수가 증가함에 따라서 오류가 발생 했습니다.

4. Delete -FG 를 구현하는데 있어서 어려움

- 수 많은 시도와 수정에도 불구하고 delete 의 fg 을 정확하게 작동시킬 자신이 없어 승기와 함께 구글링을 통해서 Swapnil Pimpale 와 Romit kudtarkar 이 카네기 멜론 대학의 대학원 시절에 작성한 Fine-Grained 와 Lock-free 에 대한 보고서를 Github 에서 발견하고 이를 읽어보고 문제가 되는 부분을 외부 함수로 만들어 처리한다는 것에 아이디어를 얻어 수정을 했다.

- 물론 우리가 작성한 트리의 구조라던가 프로그램 전체의 수행 흐름이 달라 완벽하게 적용할 수는 없지만 포인터와 다른 변수들을 선언해 줌으로서 보고서에 나와 있는 아이디어를 이용해 수정하는데 성공했다. 비록 우리가 직접 생각을 해서 구현한 부분이 아니라서 아쉽지만 그래도 오랫 동안 고생한 코드가 성공했다는 사실이 기뻤다.

정말 오랜 시간을 들여서 많은 수정을 통해서 작성한 코드인 만큼 수많은 테스트를 해보았고 고칠 때 마다 결과가 다르게 나오고 시간도 다르게 나왔기 때문에 과제를 통해서 공부하고 검색하고 성장 했던 과제 였던 것 같다.

32155068 홍승기

이번 프로젝트는 이진탐색트리를 구현하고, lock 을 사용해서 thread 를 여러 개 사용해서 삽입과 삭제를 동시에 실행시키는 환경에서 충돌이 일어나지 않고 프로그램의 수행시간을 측정하는 프로젝트였다.

프로젝트를 처음 교수님께서 설명해 주실 때는 스케줄러과제보다 쉽게 끝낼 수 있을 줄 알았는데 이진탐색트리를 구현하는 것부터 문제가 많이 생겼었다. 자료구조 수업 때와 알고리즘 수업 때 배운 이진탐색트리와 약간 구조가 다르고 이미 주어진 틀에 맞춰서 이진탐색트리를 구성해야했고, 그 틀에 맞춰서 작성 하는 게 생각 보다 시간이 오래 걸리고 원래 해왔던 이진탐색트리의 구성과 약간 달라서 정말 애를 많이 먹었다.

이진탐색트리를 구성한 후 lock 을 사용해봤더니 수많은 오류가 처음에 발생 했었다. 단일 thread 환경에서는 실행 되던 게 lock 을 사용하고 multi thread 환경에서 수행하면 프로그램이 멈추거나, 세그멘테이션 오류가 빈번하게 발생했고 이 문제 해결하기 위해서 정말 수없이 lock 위치를 바꾸고 실행시켜 본 것 같다. 리눅스환경이 비쥬얼스튜디오 환경과 많이 달라서 오류를 찾는 과정과 수정하는 것이 시간이 오래 걸린 것도 힘든 부분이었다면 힘든 부분이었던 것 같다.

coarse-grained lock 과 fine-grained lock 사용 시 lock 의 위치를 구분하는 것도 심각한 문제였다. coarse-grain 의 경우 임계영역에 상관없이 그냥 삭제나 삽입 함수 전체에 lock 을 걸어주면 해결 가능했는데, fine-grain lock 이 정말 문제가 많이 생겼다. fine-grain 삽입의 경우는 lock 위치를 잘 조정해서 해결했지만, 삭제의 경우 처음에 노드 하나하나 lock 을 걸어주는 생각을 하지 못해서 프로그램이 아예 실행이 되지 않고 deadlock 상태에 빠지거나 세그멘테이션 오류가 빈번히 발생했다... 이 문제를 해결하려고 계속 시도하다가 접근하는 노드 하나하나에 모두 lock 을 걸어주는 방식으로 시도해보려고 했으나, 이 방법으로 해결을 시도했을 때는 코드에 lock 이 너무 많이 쓰여서 코드의 가독성이 현저히 떨어지고, lock 을 많이 쓰다 보니 어느 부분에서 deadlock 에 빠지는지 확실하게 알 수 없었지만 프로그램이 실행되고는 있지만 정지된 상태가 계속해서 발생했었다.

위의 문제를 해결 할려고 정말 밤새면서 코드수정하고, lock 위치를 바꾸고, 다시 설계해보는 과정을 수없이 반복했지만 도저히 해결되지가 않아서, 구글에 여러 자료들을 검색해보니 Swapnil Pimpale 와 Romit kudtarkar 가 연구한 자료를 발견해서 코드에 적용시켜보았다.

fine-grain lock 을 구현할 때 문제가 될 만한 부분들을 다 따로 함수로 만들어서 밖에서 처리하게하고 그앞에 lock 걸어주는 형식으로 아주 간단하게 문제를 해결한 것을 보고 정말 신선한 충격으로 다가왔다. 왜 나는 이렇게 간단한 방식을 생각하지 못하고 꼭 노드에 lock 을 하나하나 다 걸어주려고 했는지 약간 허무하기도 했지만, 프로그래밍에 정답 이란게 없다는 걸 새삼스레 느끼는 계기가 된 것 같다.

저 분들의 해결방법을 확인하고 우리의 문제에 적용시켜 한번 fine-grain lock 을 구현해봤더니 정말 상상 이상의 결과가 나왔다. 앞에 우연히 성공했던 결과와는 차원이 다르게

빠르게 결과물이 산출되는 것을 확인 할 수 있었는데 왜 멀티쓰레드가 싱글쓰레드보다 빠르다고 하는지 실제로 확인 해볼 수 있어서 신기했었다.

이번 과제를 통해서 정말 또 한번 프로그래밍이라는 것이 정말 복잡하고 해결해야 할 문제에 대해서 많은 배경지식이 요구된다는 걸 또 다시 느끼게 되었고, 구글링을 통해서 위에 저 분들이 해결한 방법을 보고 꼭 프로그래밍에 정답이 없고, 많은 답이 생길 수 있다는 걸 느끼는 프로젝트였던 것 같다. lock 이 왜 멀티쓰레드 환경에서 필수적으로 필요한지도 정말 뼈저리게 느끼는 시간이 된 것 같다.