REPORT

OS Lab2 Sync

과 목 명 :	운영체제
담당교수 :	최종무 교수님
학 과 :	소프트웨어학과
학 번 :	32200327
이 름 :	김경민
제 출 일 :	2022.05.05

Goal

- Hash-Queue 구조를 이해하고 가시적으로 나타낼 수 있다.
- 데이터의 삽입과 삭제 과정의 흐름을 파악하여 insert/delete 관련 함수를 작성할 수 있다.
- Lab2에서 어떤 소비자/생산자 문제가 발생할 수 있는지 설명할 수 있다.
- Lab2에서 Lock으로 보호해야 할 critical section이 어디인지 확인하고 적절한 lock 변수를 활용해 문제를 해결할 수 있다.
- Non-lock, coarse grained lock, fine grained lock에서 각각 싱글 스레드, 멀티 스레드를 생성해 실험을 수행하고 실험 결과 및 수행 시간을 비교 분석할 수 있다.

Design

Hashlist 설계

미리 정의되어 있는 hashlist 구조체 배열을 사용했다. 배열의 각 인덱스가 bucket이 되어 Bucket별로 노드가 단일 연결 형태로 달린다. 초기화를 통해 각 인덱스에 q_loc이 NULL인 첫 노드를 할당해주어 head 역할을 하도록 하였다. 다음 노드가 추가될 경우 첫 노드의 next는 다음 노드를 가르키면서 연결된다.

Queue 설계

각 Bucket마다 노드가 달리는 hashlist와는 다르게 insert 될때마다 하나의 큐에 삽입된다. 초기화 시, head, rear는 data를 -1로 지정하여 항상 큐의 앞, 뒤를 가르키도록 하였다. prev, next를 사용해 이중연결리스트 형태로 구현하였고, front와 rear로 서로를 가르키는 환형구조로 만들었다. Insert가 되면 이 큐에 rear 바로 앞에 노드가 추가된다. hashlist의 q_loc은 이 큐를 가르키면서 서로 연결된다.

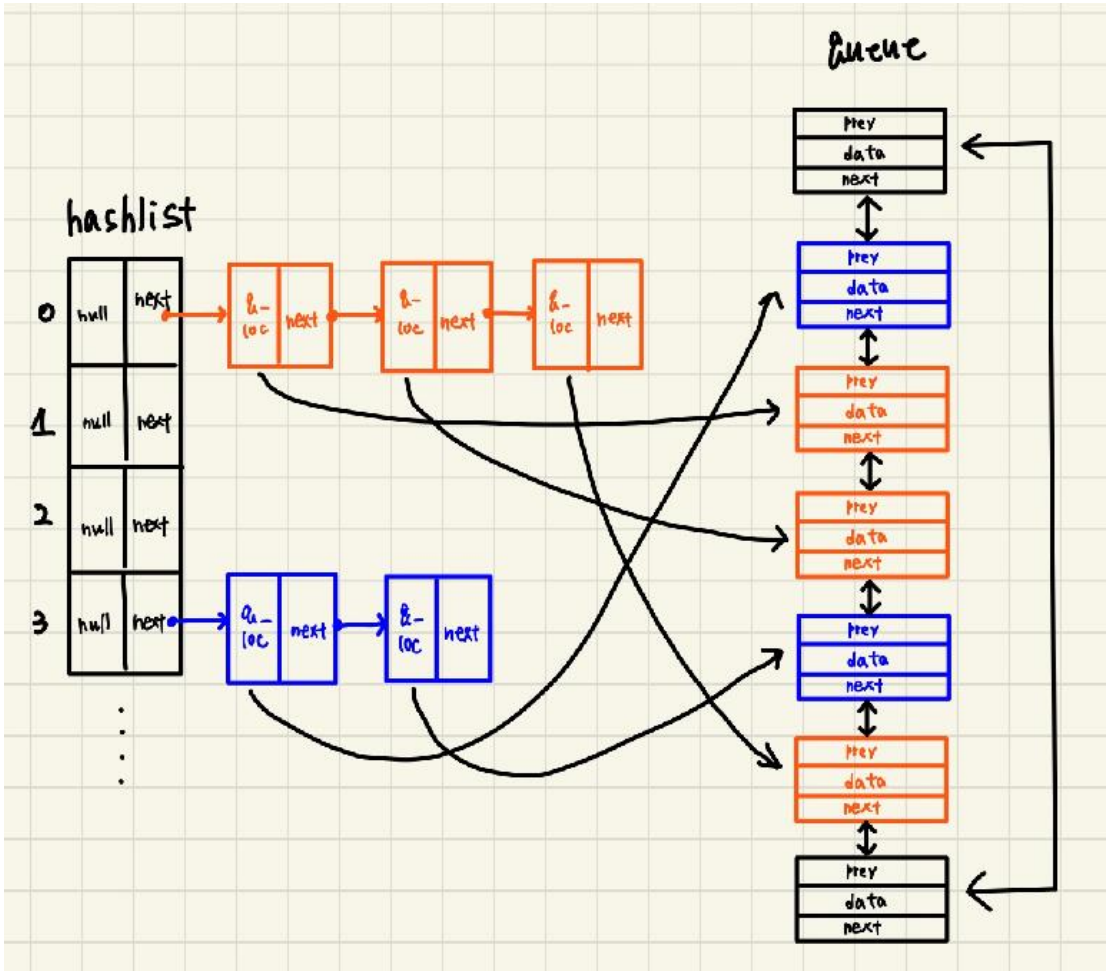
Insert 과정

hash_queue_insert_by_target 함수 안에서 먼저 value_exist 함수를 사용해 넣고자 하는 값이 이미 큐에 들어가 있는 값인지 확인 후, 중복되지 않는 경우 hash 함수를 호출해 bucket을 구한다. 그리고 hashlist[bucket] 과 target을 인자로 하여 hash_queue_add 함수를 호출해 hash에 값을 넣고, enqueue를 호출해 큐에도 값을 넣는다.

Delete 과정

hash_queue_delete_by_target 함수 안에서 bucket을 찾아 해당 hash 리스트에서 삭제하고자 하는 값에 존재하는지 탐색한 후, 존재한다면 dequeue를 호출해 큐에서 연결을 끝고 할당을 해제해주고, dequeue 종료 후 hash 에서도 연결을 끊고 할당을 해제해 삭제한다. 삭제하고자 하는 값을 찾을 때 value_exist 함수는 삭제하고자 하는 값을 찾았을 때 바로 삭제할 수 없기 때문에 중복 탐색 과정을 피하고자 사용하지 않았다.

Hash, Queue 구조 파악을 위해 Design 단계에서 다음과 같은 그림을 그려 참고했다.



기타 함수 설명

1) int hash(int val)

: Target Key를 13로 나눠 나머지를 계산한다. 그 나머지가 Bucket이 되므로 Bucket_list의 인덱스로 사용되기 때문에 다른 함수에서 해당 함수를 호출하여 Bucket 인덱스를 반환값으로 받아 큐에 삽입하는 과정을 수행할 것으로 예상된다.

2) int value_exist(int val)

: hash() 호출해 bucket을 찾고, bucket에 해당하는 hash 리스트를 큐에 찾고자 하는 데이터가 있는지 탐색한다. 데이터가 없으면 1을 반환, 데이터가 있으면 0을 반환하고 바로 종료한다.

Lock 설계

여러 개의 스레드가 queue와 hashlist에 접근하여 데이터를 삽입/삭제할 때, 상호배제를 보장하 위해 queue 와 hashlist를 위한 전역 lock을 선언해주었다.

1) coarse grained

: critical section 범위를 크게 잡기 때문에 노드를 할당 직전에 lock을 걸어주고, 모든 hash관련 혹은 큐 관련 수행이 끝난 후에 unlock을 해준다. 큐 삽입 또한 큐 노드 할당 전에 lock을 걸고 enqueue 함수를 호출하고 돌아와 hash 노드의 q_loc에 큐 노드를 연결 해주는 모든 과정을 거친 후에 unlock해준다. Dequeue를 할 때는 coarse, fine 방식 모두 Dequeue 함수 시작 위치에서 lock을 걸고 끝나는 위치에서 unlock을 걸어주는 것으로 동일하다. 하지만 hashlist에서 노드를 삭제할 때는 삭제할 노드를 찾기 직전부터 lock을 걸어준다. 그리고 만약 노드를 찾았다면 할당 해제 후 unlock을 해주고 함수를 종료하고, 찾지 못했더라고 unlock을 해주고 함수를 종료한다. Critical section의 범위를 크게 잡느라 노드를 찾았을 때, 찾지 못했을 때 따로 unlock을 해줘야 하므로 lock & unlock이 쌍을 이루지 못한다.

2) fine grained

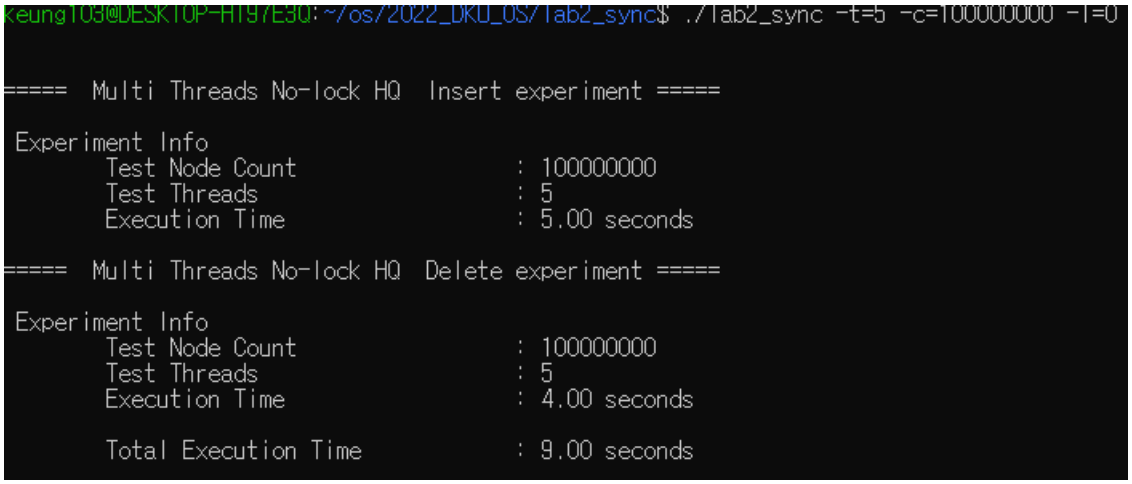
: 꼭 필요한, 즉, 실제 충돌이 일어날 수 있는 부분만을 critical section으로 잡아 lock을 걸어준다. hashlist의 경우 동적할당 수행 다음에 lock을 걸고 hash 노드와 queue 노드를 연결한 후 unlock 해줬다. 큐의 경우 enqueue 함수 내부에 lock을 걸어서 함수를 빠져나와 Hash 노드와 queue 노드를 연결 후에 unlock 해줬다. hash 노드와 queue 노드를 연결하는 수행의 경우 각 노드에 값이 제대로 들어간 후 두 노드를 충돌없이 연결해줘야 한다고 생각해 enqueue를 빠져나온 다음에 수행하도록 하고, hash lock 과 queue lock 두 개로 모두 보호하도록 설계했다. hashlist에서 노드를 삭제할 때는 삭제하는 노드의 앞 뒤를 이어주고 할당을 해제하는 부분에만 lock을 걸어준다.

Discussion

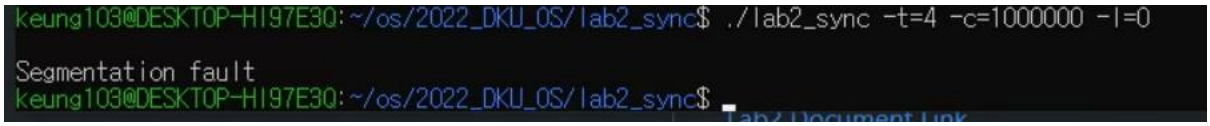
위에서 설계한 대로 대강 구현을 한 다중 스레드가 잘 생성되는지를 가장 먼저 확인했다. 하지만 내 예상과는 조금 다른 결과를 확인했다, 나는 만들고자 하는 스레드의 개수만큼 반복문이 실행되어 pthread_create를 호출하고, 그 다음 인자로 전달된 함수가 실행될 줄 알았는데 실제로는 아직 스레드가 다 만들어지기 전에 중간 중간 스레드 생성 시, 인자로 전달한 함수가 호출되곤 했다. 인자로 호출된 함수에는 데이터를 insert 하거나 delete 하는 함수를 호출하는데 이때 여러 개의 스레드가 동시에 insert를 시도하게 되면서 critical section에 중복 접근하며 충돌이 일어날 수 있다는 것을 생각했다. 처음 과제 안내를 받았을 땐, insert와 delete 스레드 사이의 충돌이라고 생각했는데 그게 아니라 하나의 스레드만이 임계영역에 들어가 데이터 삽입/삭제를 수행할 수 있도록 lock을 걸어줘야 한다는 것을 파악한 것이다.

또한 이 문제를 파악하며 새롭게 얻은 내용도 있었다. 위에서 언급했던 것처럼 언제 어떤 스레드가 호출될지 모른다면 만약 스레드2의 함수가 먼저 실행되어 critical section에 들어가 lock을 걸고 insert를 진행하다 다시 메인 스레드로 넘어가 쭉 진행되어 pthread_join을 실행하면 join의 순서가 스레드 1 -> 스레드 2 이기 때문에 결국 스레드2에 걸려 있는 lock을 영원히 unlock하지 못하게 되는 것이 아닌지에 대해 의문이 들었다. 이 문제에 대해 익명 커뮤니티를 사용해 다른 학생에게 질문한 결과 흥미로운 답변을 얻어낼 수 있었다. 나는 반복문에 따라 스레드1부터 join 연산이 실행되면 해당 스레드가 종료될 때까지 영원히 다른 스레드가 수행되지 않는다고 생각했는데 사실 join이 반드시 순차적으로 실행되지 않을 수 있고 먼저 끝나는 스레드를 먼저 수행한다고 생각하면 된다는 이야기를 들을 수 있었다. 어떻게 보면 이렇게 스레드가 무작위로 실행되기 때문에 lock을 걸어 critical section을 보호하는 것이 당연한 것이라고 연관 지어 이해할 수 있었다. 순서 보장 때문에 join을 사용한다고 했는데 이는 프로세스의 wait와는 조금 다른 부분이고, 여기서 join을 사용하는 이유는 메인 스레드가 다른 다른 스레드의 종료 시점을 조율할 수 있도록 하고, 따라서 수행시간도 측정할 수 있게 하기 위해서라고 배웠다. 아직 완전히 이해하기엔 어려운 내용이지만 기존에 이해하고 있던 지식 외에 더 자세한 내용을 알아갈 수 있어서 뿌듯했던 순간이었다.

본격적으로 non-lock 버전부터 실험을 진행했는데 또다시 예상치 못한 문제가 발생했다. 다중 스레드로 수행을 하면 충돌이 일어나 segmentation fault가 발생하는 것이 정상인데 아무리 스레드를 많이 만들고 데이터 수를 크게 해도 아래 이미지처럼 오류가 발생하지 않는 것이다. 디버깅을 사용해보고자 했지만 gdb 실행 후 코드를 한 줄 한 줄 넘기면서 확인하는 방법을 알지 못해 포기한 점이 매우 아쉬웠고, 추후에 다시 시간을 내어 공부할 필요가 있겠다고 생각했다.



segmentation fault가 발생하지 않았다는 점에서 충돌은 없지만 노드들이 제대로 연결되지 못해서 그럴 것이라고 가정하여 실제 노드가 추가되는 함수가 호출될 때 마다 need_count 변수를 1씩 증가시키고, 해당 함수 종료 후 실제 큐에 들어있는 데이터의 수를 real_count 변수에 저장하여 두 변수를 비교해 보았다. 그 결과 실제 큐에 데이터가 전혀 증가하고 있지 않다는 것을 발견할 수 있었고, insert를 진행하는 코드를 자세히 검토한 결과 value_exist 함수의 return값을 잘못 받아서 발생한 아주 간단한 실수였다. 이 문제를 간단히 해결한 뒤에는 결과가 잘 나오는 것을 확인할 수 있었다.



Non-Lock

본격적으로 실험을 진행한 결과, non-lock 에서 멀티스레드로 수행했을 때는 데이터 수를 100개 정도까지 하여 수행했을 땐 스레드가 4개여도 충돌 없이 잘 수행됐다. 하지만 데이터의 수가 커지면서 위 이미지와 같이 segmentation fault가 발생했다.

싱글 스레드로 수행했을 땐 데이터의 값이 크더라도 잘 수행되는 것을 관찰할 수 있었다.

```
keung103@DESKTOP-HI97E3Q:~/os/2022_DKU_OS/lab2_sync$ ./lab2_sync -t=1 -c=100000 -l=0

=====  Multi Threads No-lock HQ  Insert experiment =====

 Experiment Info
        Test Node Count                 : 100000
        Test Threads                    : 1
        Execution Time                  : 3.00 seconds

=====  Multi Threads No-lock HQ  Delete experiment =====

 Experiment Info
        Test Node Count                 : 100000
        Test Threads                    : 1
        Execution Time                  : 3.00 seconds

        Total Execution Time            : 6.00 seconds
```

Coarse grained Lock

이번엔 Coarse lock을 걸어 수행 후 관찰해본 결과 싱글스레드일 때는 총 시간 1초, 스레드 4개로 멀티스레드로 수행했을 땐 총 2초 정도로 1초 정도 차이가 났다. 데이터를 상당히 크게 했음에도 시간이 굉장히 짧게 측정되어 데이터 수를 더 늘려 보았다. 데이터 개수를 1억개 정도 했을 때 싱글 스레드에서는 inset 5초, delete 2초로 총 7초가 걸렸고, 스레드를 4개로 했을 땐 inset 16초, delete 4초로 총 20초가 걸렸다.

```
keung103@DESKTOP-HI97E3Q:~/os/2022_DKU_OS/lab2_sync$ ./lab2_sync -t=1 -c=10000000 -l=1

=====  Multi Threads Coarse-grained HQ  Insert experiment =====

 Experiment Info
    Test Node Count         : 10000000
    Test Threads            : 1
    Execution Time          : 0.00 seconds

=====  Multi Threads Coarse-grained HQ  Delete experiment =====

 Experiment Info
    Test Node Count         : 10000000
    Test Threads            : 1
    Execution Time          : 1.00 seconds

  Total Execution Time        : 1.00 seconds
```

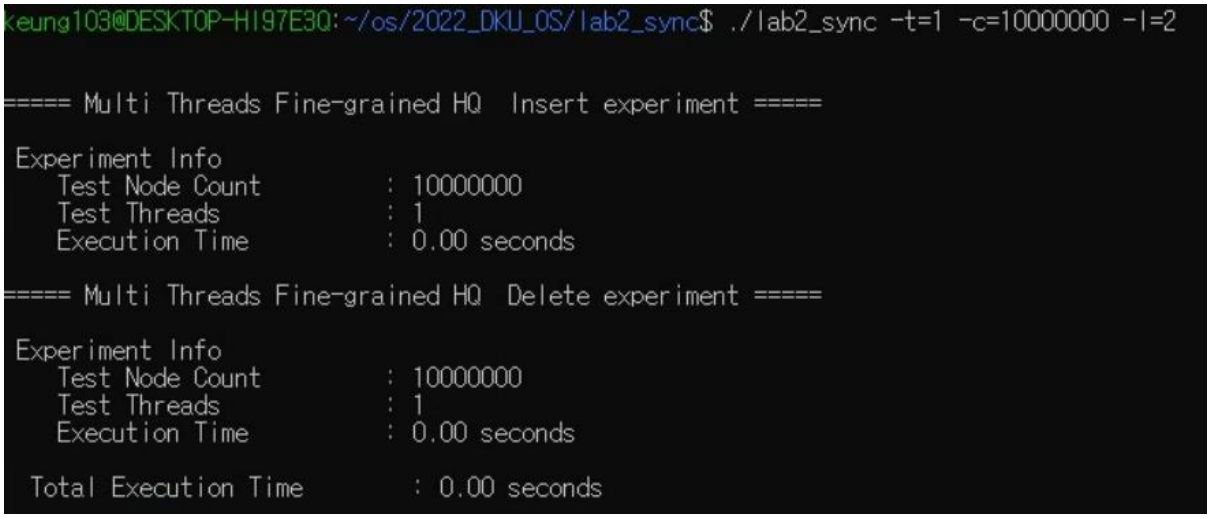
```
keung103@DESKTOP-HI97E3Q:~/os/2022_DKU_OS/lab2_sync$ ./lab2_sync -t=4 -c=10000000 -l=1

=====  Multi Threads Coarse-grained HQ  Insert experiment =====

 Experiment Info
    Test Node Count         : 10000000
    Test Threads            : 4
    Execution Time          : 1.00 seconds

=====  Multi Threads Coarse-grained HQ  Delete experiment =====

 Experiment Info
    Test Node Count         : 10000000
    Test Threads            : 4
    Execution Time          : 1.00 seconds

  Total Execution Time        : 2.00 seconds
```

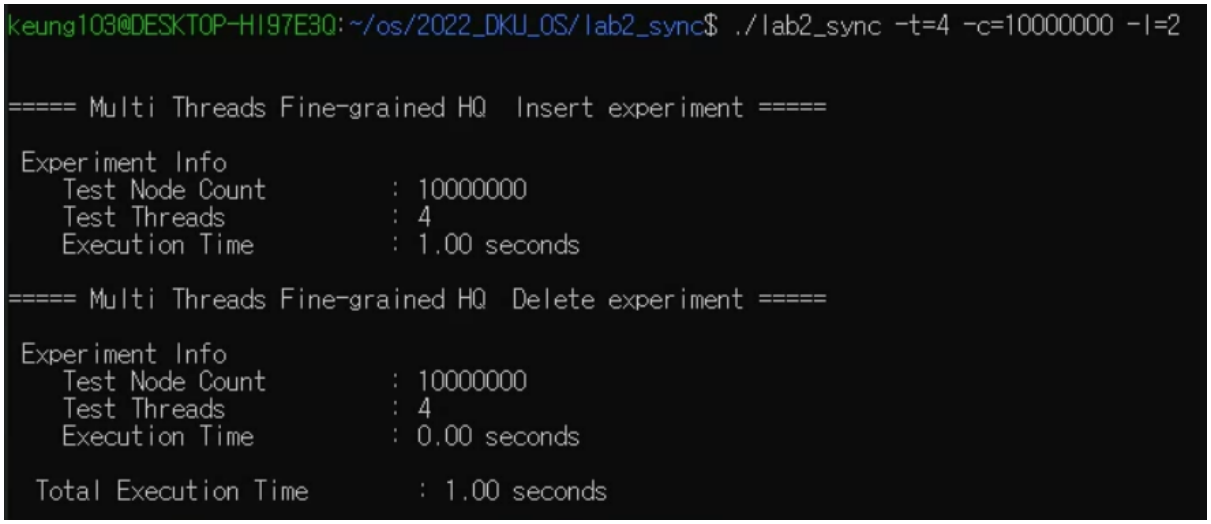
```
keung103@DESKTOP-HI97E3Q:~/os/2022_DKU_OS/lab2_sync$ ./lab2_sync -t=1 -c=100000000 -l=1

=====  Multi Threads Coarse-grained HQ  Insert experiment =====

 Experiment Info
    Test Node Count         : 100000000
    Test Threads            : 1
    Execution Time          : 5.00 seconds

=====  Multi Threads Coarse-grained HQ  Delete experiment =====

 Experiment Info
    Test Node Count         : 100000000
    Test Threads            : 1
    Execution Time          : 2.00 seconds

  Total Execution Time        : 7.00 seconds
keung103@DESKTOP-HI97E3Q:~/os/2022_DKU_OS/lab2_sync$ ./lab2_sync -t=4 -c=100000000 -l=1

=====  Multi Threads Coarse-grained HQ  Insert experiment =====

 Experiment Info
    Test Node Count         : 100000000
    Test Threads            : 4
    Execution Time          : 16.00 seconds

=====  Multi Threads Coarse-grained HQ  Delete experiment =====

 Experiment Info
    Test Node Count         : 100000000
    Test Threads            : 4
    Execution Time          : 4.00 seconds

  Total Execution Time        : 20.00 seconds
```

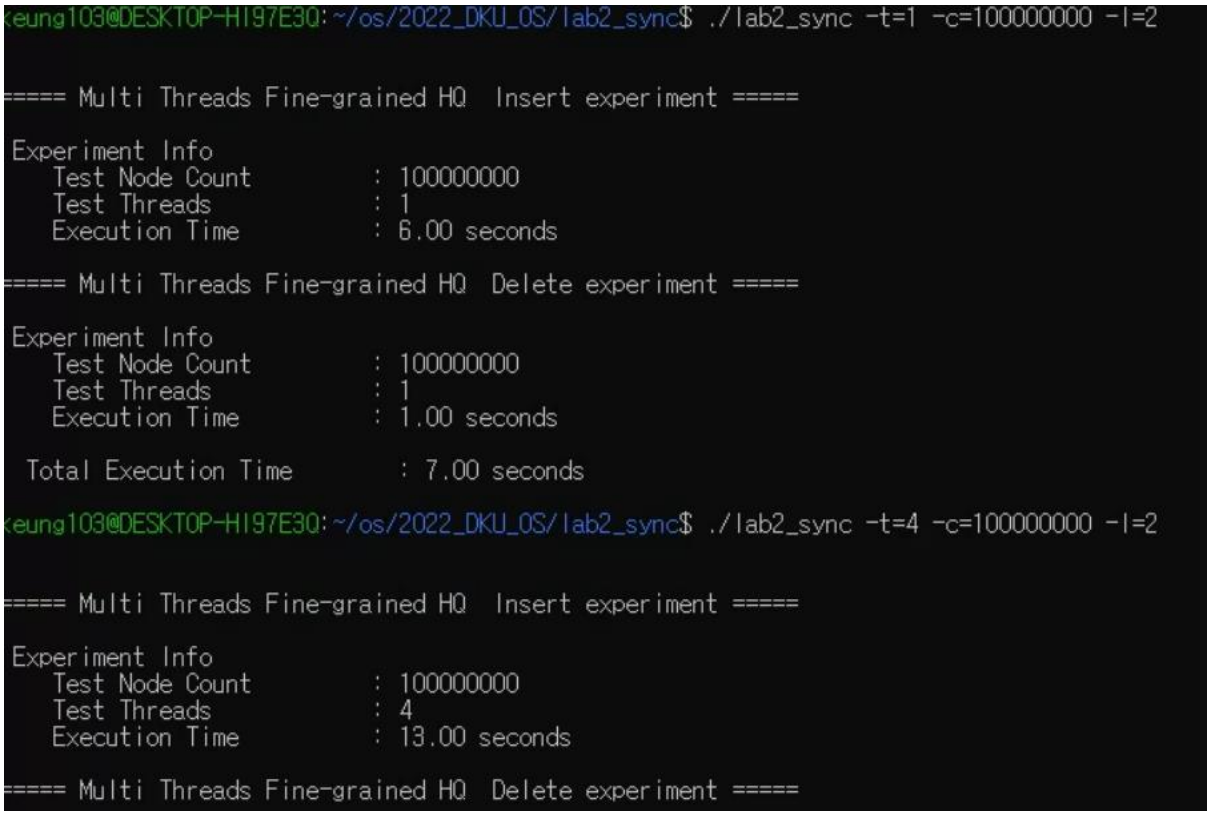
Fine grained Lock

Fine lock을 걸어 수행 후 관찰해본 결과 싱글 스레드일 때는 총 시간 0초, 스레드 4개로 멀티스레드 수행을 했을 땐 총 1초 정도로 1초 정도 차이가 났다. 멀티 스레드에서 Insert와 delete 시간 차이는 coarse 방식과 비슷하지만 전체 시간은 Fine 방식이 더 짧은 것을 관찰할 수 있었다.

이번에도 데이터를 1억개 정도로 늘려 관찰했을 때, 싱글 스레드에서는 inset 6초, delete 1초로 총 7초가 걸렸고, 스레드를 4개로 했을 땐 inset 13초, delete 2초로 총 15초가 걸렸다. 확실히 멀티스레드에서 전체 시간은 Fine 방식이 훨씬 짧다.







Coarse 방식의 경우 critical section이 길어 여러 스레드가 접근할 수 있는 코드가 더 제한적이고 그만큼 많이 대기하기 때문에 시간이 더 걸린다고 생각한다. 또 전체적으로 delete 수행의 수행 시간이 더 짧은 이유도 임계영역이 insert 수행에 비해 길지 않기 때문이라고 생각한다.

사실 Lock을 전역으로만 구현하는 것이 아니라 구조체에 lock 변수를 넣어 사용해보거나 hashlist의 bucket 별로 lock를 사용하는 방식도 시도해보고 싶었으나 시간상 그러지 못했다. 그래도 이번 과제의 기한이 넉넉했던 만큼 충분히 과제를 이해하고 다양한 시도를 해보면서 설계, 구현, 실험을 해볼 수 있었던 것이 아주 큰 도움이 되었고 개념을 이해하는 가장 좋은 방법이라고 느꼈다.

[참고자료]

https://psy-er.tistory.com/61

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=fldrh224728&logNo=220261949525

https://reakwon.tistory.com/98

https://kldp.org/node/20015