CSE4006 HW3 Report

2015004257 제갈지혜

1. 과제 설명

Concurrent data structure로 fine-grained lock을 이용한 binary search tree와 lock-free linked list를 구현한다. 각 data structure는 insert(linked-list add), delete(linked-list remove), search 함수를 가진다. 이 함수들은 Intellij 테스트 기능을 이용하여 검증하였고, 체크하는 Invariant들은 후술할 것이다. 성능 측정은 100만개의 데이터 insert를 할 때, thread의 수를 1, 2, 4, 8로 변경하면서 실행시간으로 비교하는 테스트 A와, 100만개의 데이터 insert 후, 추가로 100만개의 insert/search operation(비율은 1:1, 1:4, 1:9)을 실행할 때 thread의 수를 1, 2, 4, 8로 변경하면서 실행시간을 비교하는 테스트 B로 진행한다. 단, Lock-free linked list는 100만개의 데이터를 가지고 성능측정을 할 경우, operation의 수가 n개 일 때 시간 복잡도가 O(n^2)이므로 thread 1개일 때 성능을 측정하는 매우 긴 시간이 필요할 것으로 생각되므로 10만개의 데이터와 operation을 가지고 진행한다. 별도의 jar파일이나 main 프로그램은 존재하지 않으며 ant build시 각 java code가 컴파일 된다.

1. 구현
   1. Fine-grained lock Binary Search Tree

- insert

제일 처음 global lock을 잡고, Root가 null인 경우 root node를 생성한다. 아닌 경우 root lock을 잡고 global lock을 푼다. 그 후부터는 데이터를 삽입할 node를 찾을 때까지 탐색하는데, 현재 탐색하고 있는 node의 data가 insert data 보다 크면 left, 작으면 right child를 다음 node로 선택한다. 이 후 child node의 lock을 잡고 현재 탐색하고 있는 node의 lock을 풀어준다. 이를 이용하여 global lock을 잡고 insert하는 것이 아닌, 삽입할 node의 부모 node만 lock을 잡도록 함으로써 성능을 좋게 할 수 있다. 중복되는 값은 insert하지 않는다.

Lock을 잡는 순서는 현재 node의 lock을 잡고 있는 상황에서, child node의 lock을 잡고, 현재 node의 lock을 풀고, child node를 현재 node로 넣는 순서여야만 한다. 아닌 경우, race가 발생하여 이상 동작을 일으킬 수 있다.

- delete

제일 처음 global lock을 잡고, Root가 null인 경우 삭제할 필요가 없으므로 global lock을 풀어준다. 이 외에는 insert와 같이 탐색을 진행하나, 탐색하고 있는 node와 그 node의 부모 node의 lock을 잡고 있어야 한다. 탐색하고 있는 node가 삭제할 노드가 아니면 부모 node의 lock을 풀어주고, child의 lock을 잡는다. 삭제할 노드를 발견한 경우 대체할 다른 노드를 retrieveData 함수를 통해 찾아 대체해준다.

- search

기본적으로 insert와 동일하게 진행하며 탐색만 진행한다는 점에서 차이가 있다. Insert의 lock을 잡는 순서와 동일하게 진행한다.

* 1. Lock-free sorted linked list

주요 특징은 Operation을 진행할 predecessor와 current를 찾는 getSnapshot함수를 통해 predecessor와 current를 반환받고 atomic operation을 진행한다. Logical deletion을 표시하는 deleteBit를 체크하면서 logical deletion이 진행되지 않았고, 연산을 진행할 부분(sorted이기 때문에 data size로 비교)을 찾으면 snapshot을 반환한다. 그렇지 않으면 성공할 때까지 재시도한다. 이 과정에서 logical deletion bit가 참이라면 pointer 연결 고리를 끊어줌으로써 garbage collector가 메모리를 회수할 수 있도록 한다.

- insert

받아온 snapshot에서 pred에 새로 생성된 Node를 atomic operation인 compareAndSet으로 연결해준다. 실패할 경우 다시 snapshot을 받아오는 것부터 시도한다.

- delete

받아온 snapshot에서 지울 node(curr)에 logical deletion을 atomic operation인 attemptMark로 표시해준다. 실패할 경우 다시 snapshot을 받아오는 것부터 시도한다. 성공한 경우 predecessor의 next에 current의 next인 successor를 atomic operation인 compareAndSet으로 넣어준다.

- search

찾으려는 data보다 현재 탐색중인 node의 data가 크거나 같을 때까지 탐색을 진행한다. 이 때 logical deletion을 표시하는 deleteBit를 update하기 위해 curr.next.get 함수를 사용하며 탐색 결과 현재 node와 찾으려는 data가 같고, logical deletion이 일어나지 않은 node이면 true를 반환한다.

1. 테스트 방법

테스트에 필요한 Thread와 job을 관리하는 Thread pool은 강의실 포털 QnA에 공유된 Pool을 이용하였다. 각 operation별로 invariant를 설정하여 Assert문을 이용해 테스트를 진행하였고, 단일 스레드로 각 operation을 확인하는 테스트와 여러 thread가 동시에 insert/delete/search operation을 진행하는 테스트를 작성하였다. 테스트는 Intellij에서 Junit으로 진행했다.

* 1. Fine-grained lock Binary Search Tree

- insert

준비된 number(0부터 testSize-1)까지의 수를 insert 함수 호출을 통해 tree에 삽입한다. 이 때 사용되는 thread는 1개이고, insert가 끝나면 모든 수들이 잘 삽입되었는지 search 함수를 통해 순차 검증한다. 하나라도 False가 있을 경우 Assert 된다.

- delete

준비된 number(0부터 testSize-1)까지의 수를 insert 함수 호출을 통해 tree에 삽입한다. 이 때 사용되는 thread는 1개이고, insert가 끝나면 삽입했던 수들을 랜덤하게 shuffle하고, shuffle된 순서대로 수를 삭제한다. 모든 수들이 잘 삭제되었는지 search 함수를 통해 검증한다. 하나라도 True가 있을 경우 Assert 된다.

- search

준비된 number(0부터 testSize-1)까지의 수를 insert 함수 호출을 통해 tree에 삽입한다. 이 때 사용되는 thread는 1개이고, insert가 끝나면 삽입했던 수들을 랜덤하게 shuffle하고, 모든 수들이 잘 삽입되었는지 search 함수를 통해 검증한다. 하나라도 False가 있을 경우 Assert 된다.

- testAll

준비된 number(0부터 testSize-1)까지의 수를 insert 함수 호출을 통해 tree에 삽입한다. 동시에 마지막에 모든 연산들이 잘 이루어졌는지 확인하기 위한 Set을 만들어 같은 연산을 실행한다. 이후 삽입했던 수들을 랜덤하게 shuffle하고, 각 수에 대해 random.nextInt(3)을 통해 0인 경우 insert, 1인 경우 delete, 2인 경우 search operation을 job queue에 넣는다. 동시에 Set에도 같은 연산을 실행한다. 사용된 Thread는 4개이다. 모든 작업이 끝난 후에 Set과 Tree가 같으면 테스트 성공, 아닌 경우 Assert된다.

* 1. Lock-free sorted linked list

- insert

준비된 number(0부터 testSize-1)까지의 수를 add 함수 호출을 통해 lfll(lock-free linked list)에 삽입한다. 이 때 사용되는 thread는 4개이고, add가 끝나면 모든 수들이 잘 삽입되었는지 search 함수를 통해 순차 검증한다. 이 때 사용되는 thread는 1개이다. 하나라도 False가 있을 경우 Assert 된다.

- delete

준비된 number(0부터 testSize-1)까지의 수를 add 함수 호출을 통해 lfll에 삽입한다. 이 때 사용되는 thread는 1개이고, add가 끝나면 삽입했던 수들을 랜덤하게 shuffle하고, shuffle된 순서대로 수를 remove한다. 이 때 사용되는 thread는 4개이다. 모든 수들이 잘 삭제되었는지 search 함수를 통해 검증한다. 이 때 사용되는 thread는 1개이다. 하나라도 True가 있을 경우 Assert 된다.

- search

준비된 number(0부터 testSize-1)까지의 수를 add 함수 호출을 통해 lfll에 삽입한다. 이 때 사용되는 thread는 1개이고, add가 끝나면 삽입했던 수들을 랜덤하게 shuffle하고, 모든 수들이 잘 삽입되었는지 search 함수를 통해 검증한다. 이 때 사용되는 thread는 4개이다. 하나라도 False가 있을 경우 Assert 된다.

- testAll

준비된 number(0부터 testSize-1)까지의 수를 add 함수 호출을 통해 lfll에 삽입한다. 동시에 마지막에 모든 연산들이 잘 이루어졌는지 확인하기 위한 Set을 만들어 같은 연산을 실행한다. 이후 삽입했던 수들을 랜덤하게 shuffle하고, 각 수에 대해 random.nextInt(3)을 통해 0인 경우 insert, 1인 경우 delete, 2인 경우 search operation을 job queue에 넣는다. 동시에 Set에도 같은 연산을 실행한다. 사용된 Thread는 4개이다. 모든 작업이 끝난 후에 Set과 Tree가 같으면 테스트 성공, 아닌 경우 Assert된다.

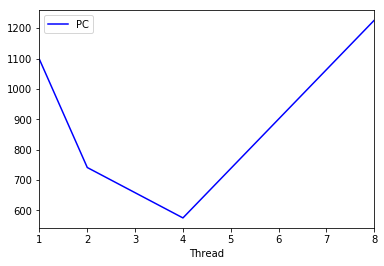
1. 성능분석

성능 테스트는 Intel(R) Core(TM) i5-7600 @ 3.5GHz, 4 core, 4 thread CPU로 진행되었고 L3캐시는 6.0MB의 크기를 가진다. 메모리는 16GB 환경에서 진행되었다.

그래프의 세로축 단위는 millisecond 단위이다.

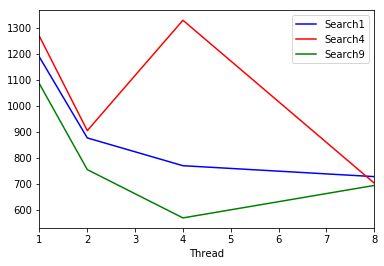
* 1. Fine-grained lock Binary Search Tree

- test A (100만개 데이터 insert with 1, 2, 4, 8 threads)

 논리 thread 수인 4개 일 때 까지는 성능이 증가하는 모습을 보이나 논리 thread 수를 넘어간 8 thread를 실행하는 경우 lock에 병목이 걸려 1 thread보다 못한 성능을 보여주게 된다.

따라서, lock을 사용하는 경우 thread 개수가 논리 thread 수를 넘어가면 성능저하가 심하게 일어난다고 할 수 있다.

- test B (100만개 데이터 insert 후 100만개 추가 operation(insert/search) with 1, 2, 4, 8 threads)

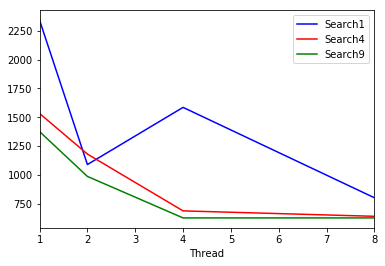
 그래프의 각 line은 insert/search를 1:1, 1:4, 1:9의 비율로 실행했을 때 나온 결과이다.

Search의 비율이 높을수록 한 node에 대해 lock을 잡고 있는 시간이 짧아지기 때문에 성능이 좋아져야 한다. 그러나 Binary search tree를 생성했을 때 tree의 모양에 따라서 성능이 가변적으로 측정될 수 있다.

Search가 1:4 비율일때 4개 thread인 경우 가장 성능이 안 좋게 나왔는데, 백그라운드에서 돌아가던 프로세스가 갑자기 에러메시지를 띄우는 상황이 발생하여 CPU에서 실행되는 우선순위가 밀려 발생한 것으로 추정된다. 대체적으로 thread 수가 논리 thread 수까지 증가할 때는 성능이 좋아지고, lock을 짧게 잡는 operation이 많으면 thread수가 논리 thread수를 넘어갈 때 성능이 저하되게 된다.

* 1. ReadWriteLock Binary SearchTree

- test B (100만개 데이터 insert 후 100만개 추가 operation(insert/search) with 1, 2, 4, 8 threads)

fine-grained lock을 ReadWriteLock으로 변경한 후 test B를 다시 진행해보았다. 이 경우 search는 readLock만을 잡기 때문에 동시에 진행 가능하다.

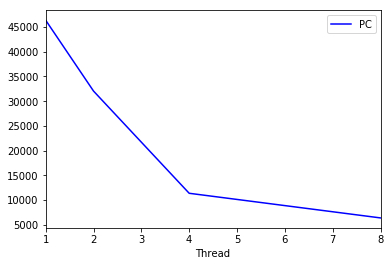
Insert는 삽입하려는 node의 parent까지 readLock을 잡고 writeLock을 잡는 방식으로 구현하려 하였으나, java에서는 lock upgrade (readlock->writelock)이 불가능하기 때문에 처음부터 writeLock을 잡는 방식으로 변경하였다.

Search 비율이 높아질수록 readLock의 비율이 높아지기 때문에 fine-grained와는 다르게 한 노드에 여러 thread가 접근이 가능하여 성능이 좋아지게 된다. 역시 Binary Search Tree의 구조로 인해 insert/search 비율이 1:1이고 thread가 2개일때의 결과는 예외 케이스로 발생했다. Thread 수가 논리 thread 수를 넘어섰음에도 성능 저하가 크게 일어나지않는 것을 보면 Thread 수가 늘어나는 것보다 search 비율이 늘어나는 것이 성능에 더 큰 영향을 미치는 것으로 분석할 수 있다.

* 1. Lock-free sorted linked list

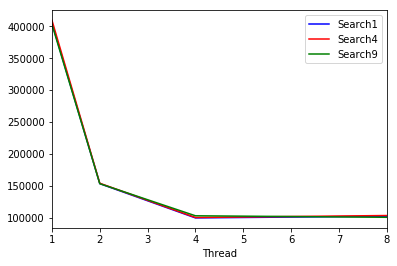
1. 과제 설명 항목에서 설명한 이유에 의해 10만개의 데이터와 operation으로 진행한다.

- test A (10만개 데이터 insert with 1, 2, 4, 8 threads)

Fine-grained lock Binary Search Tree와 가장 대조적인 점은 thread 수가 논리 thread수를 넘더라도 성능저하가 발생하지 않는다는 점이다. 현재 4개의 논리 thread가 존재하고 성능테스트는 8개 thread까지 사용하기 때문에 lock이 있다면 lock을 얻고 푸는데 병목이 있겠지만 lock-free linked list는 lock을 잡지 않기 때문에 thread 수가 늘어도 성능이 저하되는 문제가 발생하지 않는다.

따라서 thread수가 늘어나면서 성능이 증가하는 모습을 보인다고 할 수 있다.

- test B (10만개 데이터 insert 후 10만개 추가 operation(insert/search) with 1, 2, 4, 8 threads)

 역시나 Lock을 잡지 않기 때문에 thread 수가 증가하면서 생기는 lock에 대한 병목현상이 보이지 않는다. Operation별 요구조건의 차이가 fine-grained lock이나 ReadWriteLock에 비해 없기 때문에 Search 비율에 따라서 생기는 성능차이도 없다고 볼 수 있다. 그러나 Linked List라는 자료구조의 한계 때문에 실행시간이 매우 오래 걸리게 되고, 논리 thread 수가 4개이기 때문에 8 thread에서 성능 개선이 거의 없게 된다.

더 많은 논리 thread를 제공하는 환경에서라면 fine-grained structure보다 lock-free structure가 더 좋은 성능을 보일 것으로 분석할 수 있다.