Homework #3: Concurrent Data Structures

CSE4065: 소프트웨어 공학 @ Hnaynag Univ.

배지운 2015004584

Part 1. Fine-grained lock (hand-over-hand lock)을 사용하여 여러 스레드가 동시에 접근할 수 있는 Binary Search Tree를 구현한다.

1.

Collections.concurrent.BinaryTree는 과제명세에서 주어진 Fine-grained lock을 사용한 interfaces.Tree 구현체 입니다.

1. Insert

먼저 Tree Head Lock을 획득합니다. 이는 Tree의 Root에 대한 중복 생성을 막고 Root Node의 Lock을 잡기 위함 입니다. Head Lock을 획득했다면 Root가 null인지 체크하고 만약 null이라면 Root에 새 Node를 할당, 아니라면 Root의 Lock을 잠그고 Head Lock을 해제합니다. 이제 Root부터 insert하고자 하는 값을 추적하며 Node를 진행해 갑니다. 이 과정에서 다음 Node의 Lock을 얻고, 이전 Node의 Lock을 풀어주는 Hand-over-hand 방식으로 진행됩니다. 만약 다음 Node가 null이라면 이 위치에 새 Node를 생성하게 됩니다. 반환 값을 돌려주기 전에 획득했던 Node의 Lock을 풀어줍니다.

Read Write Lock을 사용한 Tree를 구현할 때에도 위와 같은 작업을 거치면서 Node를 탐색합니다. 다만 이때는 Read Lock을 사용하여 탐색하므로 다른 Node들이 동시에 Read Lock을 잡을 수 있습니다. 만약 Read Lock을 다른 스레드에서 잡은 Node에 접근해서 Write Lock을 얻고자 한다면 Read Lock을 잡고있는 모든 스레드가 끝나야 Write Lock을 잡을 수 있습니다. 하지만 이 때 Read Lock이 모두 종료하고 condition variable에 Signal을 보내면 Lock을 요청한 스레드가 순서대로 정렬되어 Lock을 얻을 수 없기 때문에 OrderedLock이라는 ReadWriteLock구현체를 새로 만들었습니다. 이는 프로젝트의 concurrent.locks.ReentrantReadWriteOrderedLock에서 확인할 수 있습니다. 이를 이용하여 Write Lock을 획득하는 Critical Section을 격리시켜 안전한 Lock 획득, 해제가 이루어질 수 있도록 했습니다. 이 부분은Collections.concurrent.RWBinaryTree.Node의 write 메서드에 구현 되어있습니다. 이 때 Write Lock을 획득하고 있던 Node의 쓰기가 끝나면 안전한 코드를 위해 다시 Read Lock을 잡은 원래의 상태로 돌려주게 됩니다. 하지만 이를 그대로 WriteLock.unlock(), ReadLock.lock()과 같이 구현하게 되면 Write Lock을 해제하고 Read Lock을 잡기전에 다른 스레드가 Write Lock을 잡아버릴 위험이 있기에 downgrade()라는 메서드를 지원해 Lock 내부적으로 Write Lock을 Read Lock으로 하향시키는 기능을 추가하였습니다.

1. Delete

Insert와 마찬가지로 먼저 Tree의 Head Lock을 얻어 Root를 검사합니다. 만약 아니라면 다음 Node로 Hand-over-hand Lock을 획득합니다. 이때 Node를 삭제하기 위해서는 부모의 left혹은 right을 비워줄 필요가 있으므로 2개의 Node (current, parent) Lock을 동시에 잡고 있어야 합니다. 만약 삭제할 Node를 찾았다면 그 Node를 대신할 값이 있는지 확인합니다. 이 부분은 replacement함수로 구현되어 있습니다. 만약 삭제할 Node의 left가 null이 아니라면 left의 가장 오른쪽 Node를 얻습니다. Left가 null이지만 right가 null이 아닐 경우 right의 가장 왼쪽 Node를 얻습니다. 만약 둘다 null이라면 대신할 값이 필요 없으므로 null을 반환합니다.

Read Write Lock을 이용한 Delete의 구현은 기존의 Delete의 구현과 같지만 탐색과정은 Read Lock으로 처리해서 여러 스레드가 Read Lock을 동시에 잡아 병렬성을 향상시킬 수 있었습니다. 만약 Delete할 Node를 찾았다면 해당 Node의 부모에 대해 Write Lock을 얻습니다. 이후 해당 Node의 Write Lock을 추가로 얻고 이 Node를 대체할 값을 찾습니다. 이 찾는 과정은 일반적인 Lock을 사용한 구현과 같지만 이는 탐색이기 때문에 처음에는 Read Lock을 사용하여 replacement를 찾으려고 했으나 이렇게 구현할 경우 Read Lock을 Write Lock으로 바로 업그레이드 할 수 없기 때문에 Dead Lock이 발생하게 됩니다. 때문에 Replacement를 찾는 과정도 Write Lock을 잡으며 Hand-over-hand locking을 수행하도록 만들었습니다.

1. Search

탐색 과정에서는 값이 변경되어 Dirty write가 발견되는 경우는 없지만 값을 읽는 중의 다른 쓰레드의 쓰기는 막아야함으로 Insert나 Delete 에서 Hand-over-hand Lock 방식으로 Node를 찾아가는 과정처럼 구현합니다.

Read Write Lock을 이용한 Search의 구현은 기존의 Search와 다르지 않습니다. 모든 Lock을 Read Lock으로 잡고 Hand-over-hand로 탐색합니다.

2.

프로젝트의 test모듈의 collections.concurrent.BinaryTreeTest는 위에서 작성한 Tree 구현이 제대로 작동하는지 검증하는 역할을 합니다. Insert, Delete, Search가 하나의 스레드에서, 혹은 여럿의 스레드에서 동시에 작업하고 그것이 참인지 검증해서 Tree의 작업 수행이 정상적으로 종료되었는지 확인합니다. 작성된 BinaryTree 클래스는 테스트를 모두 통과했으므로 올바르게 작동한다고 볼 수 있습니다. 테스트 클래스의 complex테스트는 100만개의 숫자를 insert하고 그 이후 100개의 insert, delete, search 작업을 병렬적으로 처리하여 결과가 맞는지 확인합니다.

3.

테스트는 아래의 두 클라이언트에서 진행되었습니다.

Window: Intel® Core™ i7-6700K CPU @ 4.00GHz, L3 Cache 8.0MB, Memory 32.0GB (4C8T)

MacOS: Intel® Core™ i5-5257U CPU @ 2.70GHz, L3 Cache 3.0MB, Memory 8.0GB (2C4T)

성능 측정 결과는 results에 있습니다. 그래프는 모두 스레드 수를 x축으로 하며 수행 시간을 y축으로 합니다.

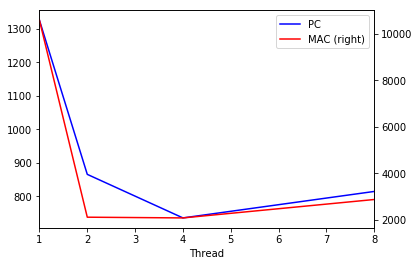


Figure 1 Insert 1M

Figure 1은 위에 설명된 두개의 클라이언트로 1M의 Insert를 실행했을 때의 그래프 입니다. Window버전은 4코어 CPU를 사용하여 4개의 스레드 때 가장 좋은 효율을 보임을 알 수 있습니다. MAC버전은 2코어 CPU를 가지고 있어 2개 이상의 스레드에서 월등한 효율을 냄을 알 수 있습니다. 그 이상의 스레드에서 작동할 경우 성능이 오히려 저하됨을 알 수 있는데 이는 여럿의 스레드가 서로 Lock을 얻으려고 경쟁을 함으로 오히려 대기 시간이 늘어나서 그렇습니다.

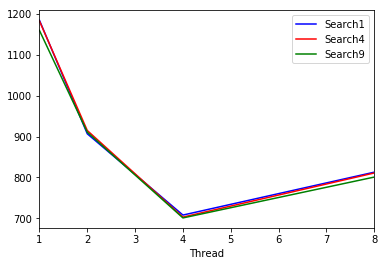


Figure 2 Insert, Search

Figure 2에는 Insert와 Search를 각각 1, 4, 9로 하고 스레드의 수를 1, 2, 4, 8로 했을 때의 그래프 입니다. 이 실행은 Windows에서 진행되어 4코어 CPU를 사용했기 때문에 4스레드에서 가장 좋은 성능을 내고 있습니다. Insert만 했을 때와 마찬가지로 4스레드보다 많아질 경우 오히려 성능이 저하되고 있는데 이는 다수의 스레드가 Lock을 얻기 위해 경쟁을 해서 그렇습니다. Search가 많아질수록 많은 스레드에서 더 좋은 성능을 내고 있지만 큰 차이는 없는 것을 알 수 있습니다. 이는 Search와 Insert가 비슷한 개수의 Lock을 잡고 수행하기 대문에 Insert나 Search에 큰 차이가 없음을 알 수 있습니다.

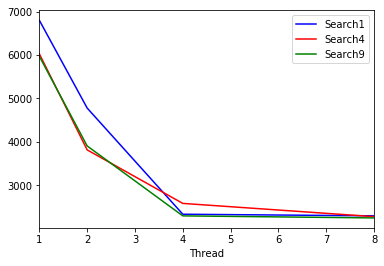


Figure 3 Insert, Search Read Write Lock

Figure 3에는 Figure 2에서 진행했던 실험을 Read Write Lock을 사용하여 구현한 결과 그래프 입니다. 위의 실험과는 다르게 CPU 코어 수인 4개보다 많은 스레드(8개)일 때 성능저하가 일어나지 않았습니다. 또한 1, 2 스레드에서 1:1 비율의 Insert, Search보다 1:4, 1:9비율의 Search가 성능이 더 좋았는데, 이는 Read Lock만 수행하는 Search가 Write락과 비교하여 한번에 더 많은 Lock을 수행할 수 있기 때문으로 보여집니다. 하지만 스레드에서 더 나쁜 결과를 보여주고 있지만, 컴퓨터의 다른 작업이 영향을 끼칠 수 있는 오차 범위 내라고 판단합니다.

Part 2. Lock-Free Sorted Linked List를 구현한다.

1.

Lock-Free Sorted Linked List를 구현해야 했기 때문에 Collections.interfaces.List와 collections.LinkedList를 먼저 구현 했습니다. 하지만 Linked List를 구현하는 과정은 간단하고 과제의 요구사항이 아니기 때문에 생략합니다.

Lock-Free Data Structure를 구현하기 위해 먼저 Atomic한 snapshot을 저장할 Window라는 클래스를 정의했습니다. Window 클래스는 어떤 값의 키(자바에서는 hashCode)를 가지는 Node의 pre, cur, Node의 상태를 Atomic하게 Snapshot한 객체입니다. 구현된 find 함수는 key에 해당하는 Window객체를 돌려주는 함수입니다. Head Node부터 탐색을 시작해서 Atomic Operation인 compareAndSet을 이용해 다음 Node가 바뀌었는지 확인하면서 탐색합니다. 이때 자신보다 크거나 같은 키를 발견하면 그 즉시 pre, cur Node를 갖는 Window객체를 반환합니다. Java의 Atomic Operation을 사용하기 위해 AtomicMarkableReference 클래스를 Node의 래퍼로 사용하여 만들었습니다.

1. Insert

Lock-Free Sorted Linked List의 Insert는 add라는 이름의 메서드로 구현되어 있습니다. Insert 하고자 하는 item의 key를 얻습니다(이 때 포인터가 사용가능한 언어는 포인터를 통해서 구현할 수 있지만 자바는 hashCode를 지원함으로 이를 통해 구현하였습니다). 이 key를 만족하는(key보다 작은 가장 큰 Node)를 얻기 위해 find함수를 사용하여 Window 객체를 얻습니다. 만약 추가하고자 하는 item의 key와 Node의 item의 key가 같다면 이미 존재하는 것 이므로 false를 반환하고, 아니라면 새 노드를 추가하여 이전 노드의 다음에 더합니다. 이 때도 Atomic Operation인 compareAndSet을 사용합니다. Atomic Operation을 사용하므로 이 과정은 실패할 수 있으며 그렇기 때문에 while loop를 통해 값이 추가될 때까지 반복해서 진행하게 됩니다. 즉 획득한 Window객체의 획득 시점과 새 Node를 생성해서 뒤에 이어 붙이는 시점까지 다른 스레드의 접근이 없는 것을 확인하여 없다면 실행하고 있다면 다시 Window를 얻어옵니다.

1. Delete

Lock-Free Sorted Linked List의 Delete도 기본적인 개념은 Insert와 같습니다. Window 객체를 얻고 이 객체에 대해 삭제를 시도할 때까지 변동사항이 없다면 다른 어떠한 스레드도 접근하지 않은 상태로 보고 compareAndSet을 이용하여 삭제합니다.

1. Search

Search는 contains 메서드로 구현되어 있고 Insert나 Delete에 비해 간단한 구조를 가집니다. Search는 Node를 읽기만 하고 쓰기를 하여 Critical Section이 발생하는 일은 없으므로 단순히 Node의 다음을 탐색하는 것으로 쉽게 구현할 수 있습니다.

2.

프로젝트의 test모듈의 collections.concurrent.lockfree.LinkedListTest는 위에서 작성한 Tree 구현이 제대로 작동하는지 검증하는 역할을 합니다. Insert, Delete, Search가 하나의 스레드에서, 혹은 여럿의 스레드에서 동시에 작업하고 그것이 참인지 검증해서 List의 작업 수행이 정상적으로 종료되었는지 확인합니다. 작성된 Lock Free Linked List 클래스는 테스트를 모두 통과했으므로 올바르게 작동한다고 볼 수 있습니다. 테스트 클래스의 complex테스트는 100만개의 숫자를 insert하고 그 이후 100개의 insert, delete, search 작업을 병렬적으로 처리하여 결과가 맞는지 확인합니다.

3.

테스트는 아래의 두 클라이언트에서 진행되었습니다.

Window: Intel® Core™ i7-6700K CPU @ 4.00GHz, L3 Cache 8.0MB, Memory 32.0GB (4C8T)

MacOS: Intel® Core™ i5-5257U CPU @ 2.70GHz, L3 Cache 3.0MB, Memory 8.0GB (2C4T)

성능 측정 결과는 results에 있습니다. 그래프는 모두 스레드 수를 x축으로 하며 수행 시간을 y축으로 합니다.

Linked List 특성상 add, delete Operation이 걸리는 시간은 BST에 비해 매우 길기 때문에 100만의 작업을 모두 수행하지 않고 10만의 작업을 수행한 시간을 측정하였습니다.

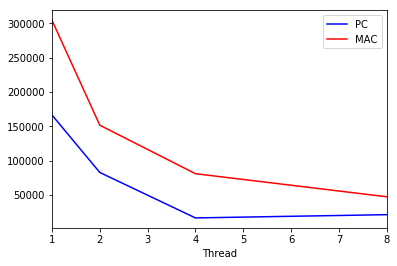


Figure 4 Insert 100K

Figure 4는 10만의 Insert Operation을 수행한 시간을 위에 언급된 두개의 CPU에서 측정한 그래프 입니다. Part1의 BST와 비교해서 CPU의 코어 개수와 같아지는 특정 지점에서 가장 좋은 성능을 내는 것이 아닌 스레드 수가 증가할수록 더 좋은 성능을 내고 있음을 확인할 수 있습니다. 추가적인 실험으로 16 스레드의 경우도 실험해보았지만 Figure 1과 같이 성능이 오히려 저하되는 경우는 나타나지 않았습니다. 이는 Lock Free로 구성된 자료구조의 특성상 다수의 스레드가 실행해도 Lock에 의한 대기가 발생하지 않기 때문으로 보여집니다. 하지만 이는 일정 값으로 수렴하여 이론적인 성능 향상의 최대값이 존재하는 것처럼 보여집니다. 4코어 CPU를 사용한 PC 실험에서는 8스레드 이상일 경우 성능 향상이나 저하가 크지 않고, MAC도 16이상의 스레드에서 그런 경향을 보이고 있습니다.

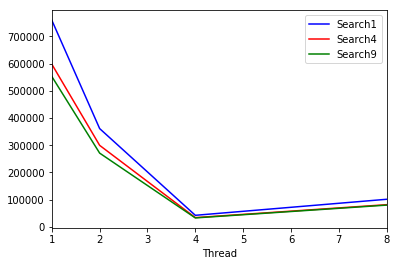


Figure 5 Insert, Search 100K

Figure 3에는 이전의 Figure 2, 4 에서 진행한 실험을 Lock Free Linked List에 대해 적용해 본 그래프입니다. 마찬가지로 CPU코어 수와 같은 4스레드에서 가장 좋은 성능을 보여주었지만 스레드가 물리 코어보다 많은 경우(8 스레드)에 성능 저하가 일어났습니다. 하지만 이 성능 저하는 크지 않고 오히려 Figure 2의 일반적인 Lock을 사용한 BST의 Insert, Search 실험보다 적은 성능 저하를 보입니다. 그러나 Figure 3의 RW Lock으로 구현한 Insert, Search 실험보다는 큰 성능 저하를 보입니다. Lock Free 구조체의 특성상 다수의 스레드가 작업할 경우 성능저하가 심하게 일어나지 않을 것이라 기대했지만, 실험결과가 이와 같이 나온 까닭은 실험한 스레드의 수가 8로 상대적으로 적어 더 많은(32, 64 등) 스레드에서 실행하면 두드러지게 차이가 나타날 것으로 기대합니다. Figure 2나 3에 비해 Search 비율의 증가가 (비교적) 두드러진 차이를 보이고 있습니다. 이는 Lock Free Search는 Critical Section이 존재하지 않아 Write Lock을 위해 Read Lock의 해제를 대기해야 했던 RW, 혹은 일반 Lock 구현의 자료구조보다 Search에서 좀 더 좋은 효율을 낼 수 있기 때문으로 보여집니다.