소프트웨어공학

Homework #3: Concurrent Data Structures 개발/디버깅

2013011800 구장회

[part1]

[0. 코드 구성]
- BST_FGL_REL.java : BST with fine-grained lock (ReentrantLock)
- BST_FGL_RWL.java: BST with fine-grained lock (ReentrantReadWriteLock)
- TEST0.java : test code for 2
- TEST1.java : test code for 3-a
- TEST2.java : test code for 3-b
- TEST3.java : test code for 3-c

[1.BST에 구현한 fine-grained lock에 대해서 operation 별로 (insert, delete, search) 간략히 설명하세요.]
[코드에 주석에 상세히 적었습니다.]
- **fine-grained lock 으로 'java util concurrent locks ReentrantLock' 사용**
- **BST 자체의 lock 1개 , 노드 별 lock 1개씩 사용**
- **void insert(int toInsert)**
  - public void insert(int toInsert) method
    - BST lock()
    - root == null 이면 , 인자로 새로운 노드를 만들어서 루트로 정하고 BST unlock() , return ;
    - 아니면 , BST unlock() , insertHelper(root,toInsert,null) 호출
  - public void insertHelper(Node root , int toInsert , Node parent) method
    - root == null 이면, throw exception
    - 아니면, root.lock() , parent != null 이면 parent.unlock()
    - root.data와 toInsert를 비교해서
      - root.data가 크면 / 왼쪽 자식 노드가 null 이면 , 왼쪽 자식 노드를 toInsert로 새로 만들고 , root.unlock() / 왼쪽 자식 노드가 null 이 아니면 , 왼쪽 자식 노드로 재귀 호출
      - root.data가 작으면 / 오른쪽 자식 노드가 null 이면 , 오른쪽 자식 노드를 toInsert로 새로 만들고 , root.unlock() / 오른쪽 자식 노드가 null 이 아니면 , 오른쪽 자식 노드로 재귀 호출
      - root.data와 같으면 , 이미 데이터가 있으므로 , root.unlock() 하고 , return ;

- **void delete(int toDelete)**
  - public void delete(int toDelete) method
    - BST lock
    - root == null 이면 , BST unlock() , return ;
    - 아니면 , BST unlock() , deleteHelper(root,toDelete,null) 호출
  - public void deleteHelper(Node root , int toDelete , Node parent) method
    - root == null 이면 , return null
    - 아니면, root.lock() 하고 , root.data와 toDelete를 비교해서
    - root.data == toDelete 이면
      - 두 자식노드가 모두 null 일 경우

        - root = null , parent != null 이면 parent.unlock() , (이전상태의)root.unlock() , return root
      - 하나의 자식 노드만 null 일 경우
        - root = null 이 아닌 방향의 자식노드 , parent != null 이면 parent.unlock() , (이전상태의)root.unlock() , return root
      - 두 자식노드 모두 null 이 아닐경우
        - root.data = 오른쪽 자식노드의 가장 왼쪽 자식 노드's data , root의 오른쪽 자식은 그 노드를 삭제한 노드로 변한다.
    - root.data가 작으면 , parent != null 이면 parent.unlock() , 오른쪽 자식노드로 재귀 호출
    - root.data 가 크면 , parent != null 이면 parent.unlock() , 왼쪽 자식노드로 재귀 호출

- **boolean search(toSearch)**
  - public Node findMin(Node root , Node parent) method
    - root == null 이면 , return null
    - 아니면 , root.unlock() , parent != null 이면 parent.unlock()
      - 왼쪽 자식노드 == null 이면 , root.unlock() , return
      - 아니면 , 왼쪽 자식 노드 로 재귀 호출
  - public boolean search(int toSearch) method
    - BST lock
    - root == null , BST unlock() , return false
    - 아니면 , BST unlock() , searchHelper(root,toSearch,null) 호출
  - public boolean searchHelper(Node root , int toSearch , Node parent) method
    - root = null 이면 , throw exception
    - 아니면 , parent != null 이면 parent.unlock()
    - root.data 와 toSearch를 비교해서
      - root.data 가 크면
        - 왼쪽 자식노드가 null 이면 , root.unlock() , return false
        - 아니면 , 왼쪽 자식노드로 재귀호출
      - root.data 가 작으면
        - 오른쪽 자식노드가 null 이면 , root.unlock() , return false
        - 아니면 , 오른쪽 자식노드로 재귀호출
      - root.data 와 같으면
        - root.unlock() , return true

[2.구현한 BST를 어떤 식으로 테스트 했는지 기술하세요. 구현한 BST에 여러 thread가 동시에 접근하여 insert, delete, search 를 할 때 올바르게 동작하는지 어떻게 테스트 하였는지 설명하세요.]

[# of thread = 4]

- testcase1
  - 100만번의 insert —> 평균 2.3118
  - (100만번의 insert 후)100만번의 delete —> 평균 1.4207
  - (100만번의 insert 후) 100만번의 search —> 평균 0.1483
- testcase 2
  - 100만번의 insert/delete/search (1:1:1 비율) —> 평균 1.3638

모든 테스트에 deadlock이 없고 , lock이 걸린상태에서 다시 lock을 try하는 경우가 없으며
평균시간이 옳게 나왔으므로 테스트 결과가 올바르다고 판단했다.

[3.구현한 BST의 성능을 다음과 같이 분석하세요. 본인이 실험한 컴퓨터의 스펙 (core갯수, 메모리 크기, 캐쉬 크기 등)을 기술하고 스펙이 어떻게 성능에 영향을 미치는지함께 분석하세요.]

- 본인 컴퓨터의 스펙
  - 2.6GHz dual-core Intel Core i5 processor
  - 8GB of 1600MHz DDR3L
  - 3MB shared L3 cache

[3-a.100만개의 랜덤한 숫자를 BST에 insert할 때 thread 1개, 2개, 4개, 8개로 나누어서 insert할 경우 실행시간이 어떻게 되는지 그래프로 그리고 설명하세요.]

- 결과(쓰레드의 갯수에 따라)
  - 1개 -> 2개: 시간이 75% 정도로 감소.
  - 2개 -> 4개 : 결과가 거의 같다.
  - 4개 -> 8개 : 오히려 2,4 개 일때의 2배 가까이 늘어났다.

[3-b.100만개의 랜덤한 숫자를 BST에 insert한 후 추가로 100만개의 insert/search operation을 thread 1,2,4,8개로 실행할 때 실행시간에 대해서 그래프로 그리고 설명하세요. Insert 와 search 비율은 1:1, 1:4, 1:9로 해서 실험하세요 (search 가더 많게).]

- 결과(insert/search 의 비율에 따라)
  - 1:1 : 쓰레드의 갯수가 2개일때 가장 빠르고(1개일때의 1.5배), 4,8개 일때는 둘의 중간 수준이다.
  - 1:4 : 1:1의 경우와 비슷한 그래프가 나왔으나(순서가 비슷) , 들쭉날쭉 하였다.
  - 1:9 : 실행시간이 4개 > 2개,8개 > 1개 순서로 나왔다.

[3-c.구현에 사용한 lock을ReadWriteLock으로 바꿔서 b의 실험을 반복하고 성능이 어떻게 달라지는지 설명하세요.]

- fine-grained lock 으로 'java util concurrent locks ReentrantReadWriteLock' 사용
  - insert ,delete 에서의 lock 을 모두 WriteLock 으로 바꿈
  - search 에서의 lock을 모두 ReadLock 으로 바꿈

- 결과(insert/search 의 비율에 따라)
  - 1:1 : 1 > 2,4,8(비슷). but 3-b에 비해 전체적인 시간이 많이 증가했다.
  - 1:4 : 1 > 2,8(비슷) > 4. but 3-b에 비해 전체적인 시간이 많이 증가했다.
  - 1:9 : 8 > 1 > 2,4(비슷). but 3-b에 비해 전체적인 시간이 많이 증가했다.

- ReadWriteLock의 특성으로 3-c 가 3-b 보다 많이 느리다.

[part2]

[0. 코드 구성]
- BST_LF.java : Lock-Free BST
- TEST0.java : test code for 2
- TEST1.java : test code for 3-a
- TEST2.java : test code for 3-b

[1.BST에 구현한 fine-grained lock에 대해서 operation 별로 (insert, delete, search) 간략히 설명하세요.]

[코드에 주석에 상세히 적었습니다.]

- **'java.util.concurrent.atomic.AtomicMarkableReference' 사용**
- **root에 AtomicReference , 노드 별 AtomicMarkableReference 사용**
- **boolean insert(int toInsert)**
  - root를 curNode로 load한다. 삽입할 integer로 새로운 노드를 하나 만든다(newNode)
  - root==null 이면 root를 null로 CAS 하고 newNode로 바꾼다.
    - CAS 성공하면 return true , 실패하면 start from scratch
  - root!=null 이면 integer의 비교연산을 통해 삽입될 위치를 찾는다.
    - toInsert와 같은 값을 가지는 노드를 찾았을때 그 노드가 leaf 이면 return false
    - leaf가 아니면 , 그 노드의 오른쪽 자식노드를 curNode로 지정한다.
  - grand-parent node == null 이면
    - parent node가 marking 되있으면 root.CAS(parent node, null). 하고 start from scratch
    - 아니면 , 새로운 internal node를 만들고 , root.CAS(parent node , new parent node) 하고 성공하면 return true , 실패하면 start from scratch
  - parent node 가 marking 되있을 경우
    - grand-parent node에서 다른 자식노드를 new parent node로 지정하고 그것을 다시 parent node로 지정한다.
  - [삽입]
    - parent node , new node를 자식으로 가지는 subtree를 만들고 grand-parent node에 자식노드로 삽입한다. 이것이 성공하면 return true, 실패하면 start from scratch.

- **int delete(int toDelete)**
  - root를 curNode로 load 한다.
  - root==null 이면 , return null.
  - root != null 이면 integer의 비교연산을 통해 curNode !=null 일때까지 toDelete와 같은 값을 가지는 노드를 찾았을때
    - 그 노드가 leaf 이면
      - parent node == null 이면, root.CAS(curNode,null)
      - grand-parant node == null 이면 , root.CAS(parent node , the other parent node)
      - parent node에 toDelete를 갖는 노드를 삭제한다.
    - leaf 가 아니면 , curNode = curNode's right child node
  - while-loop을 탈출했다면 return false

- **boolean search(int toSearch)**
  - root를 curNode로 load 한다.
  - insert method와 비슷하게 integer의 비교연산을 통해 적절한 노드를 찾는다.
  - 그 노드가 leaf 이면
    - marking 되어있으면 false return , 아니면 true return.
  - leaf가 아니면
    - curNode = curNode's right child node
  - while-loop 탈출한다면 return false

[2.구현한 BST를 어떤 식으로 테스트 했는지 기술하세요. 구현한 BST에 여러 thread가 동시에 접근하여 insert, delete, search 를 할 때 올바르게 동작하는지 어떻게 테스트 하였는지 설명하세요.]

[# of thread = 4]

- testcase1
  - 100만번의 insert —> 평균 1.2238
  - (100만번의 insert 후)100만번의 delete —> 평균 0.5725
  - (100만번의 insert 후) 100만번의 search —> 평균 0.5492
- testcase 2
  - 100만번의 insert/delete/search (1:1:1 비율) —> 평균 1.4109

모든 테스트에 deadlock이 없고 , lock이 걸린상태에서 다시 lock을 try하는 경우가 없으며
평균시간이 옳게 나왔으므로 테스트 결과가 올바르다고 판단했다.

[3.구현한 BST의 성능을 다음과 같이 분석하세요. 본인이 실험한 컴퓨터의 스펙 (core갯수, 메모리 크기, 캐쉬 크기 등)을 기술하고 스펙이 어떻게 성능에 영향을 미치는지함께 분석하세요.]

[실행결과의 표, 그래프 ==> Graph_Part2.pdf.]

- 본인 컴퓨터의 스펙
  - 2.6GHz dual-core Intel Core i5 processor
  - 8GB of 1600MHz DDR3L
  - 3MB shared L3 cache

[3-a.100만개의 랜덤한 숫자를 BST에 insert할 때 thread 1개, 2개, 4개, 8개로
나누어서 insert할 경우 실행시간이 어떻게 되는지 그래프로 그리고 설명하세요.]

- 결과(쓰레드의 갯수에 따라)
  - 대체적으로 실행시간이 쓰레드의 갯수에 따라 1개>2개>4개>8개 이다.
  - 실행시간의 순서가 눈에 보이지만 경우에 따라 2,4,8개의 결과가 큰 차이가 없다.

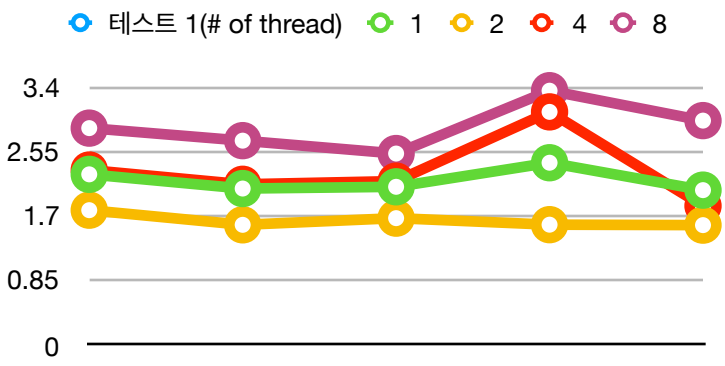
[3-b.100만개의 랜덤한 숫자를 BST에 insert한 후 추가로 100만개의 insert/search operation을 thread 1,2,4,8개로 실행할 때 실행시간에 대해서 그래프로 그리고 설명하세요. Insert 와 search 비율은 1:1, 1:4, 1:9로 해서 실험하세요 (search 가더 많게).]

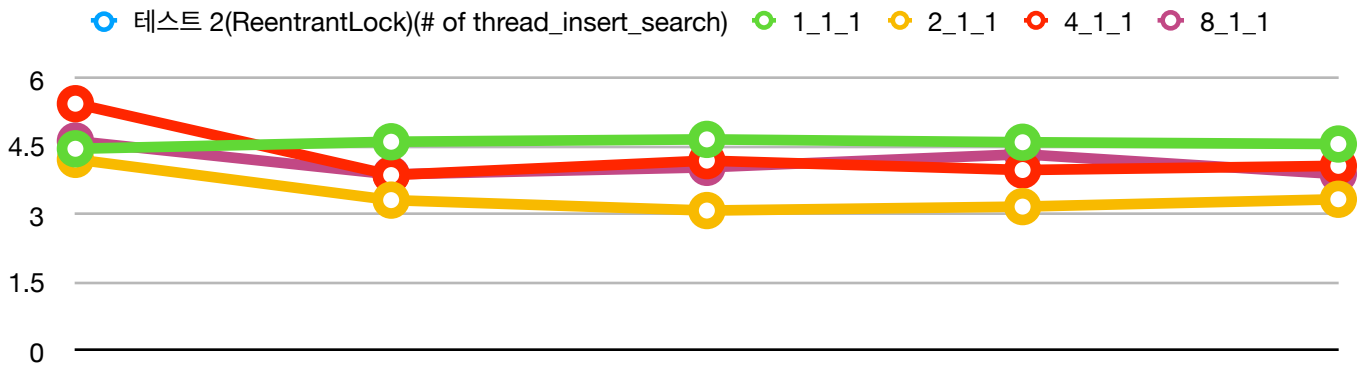
- 결과(insert/search 의 비율에 따라)
  - 1:1 : 실행시간이 1개와2개가 비슷하며 4,8개보다 오래 걸린다. 4,8의 경우는 거의 비슷하다.
  - 1:4 : 1개의 경우가 가장 오래걸린다. 나머지는 큰 차이가 없이 뒤섞인다.
  - 1:9 : 1개의 경우가 가장 오래걸린다. 나머지는 큰 차이가 없이 뒤섞인다.

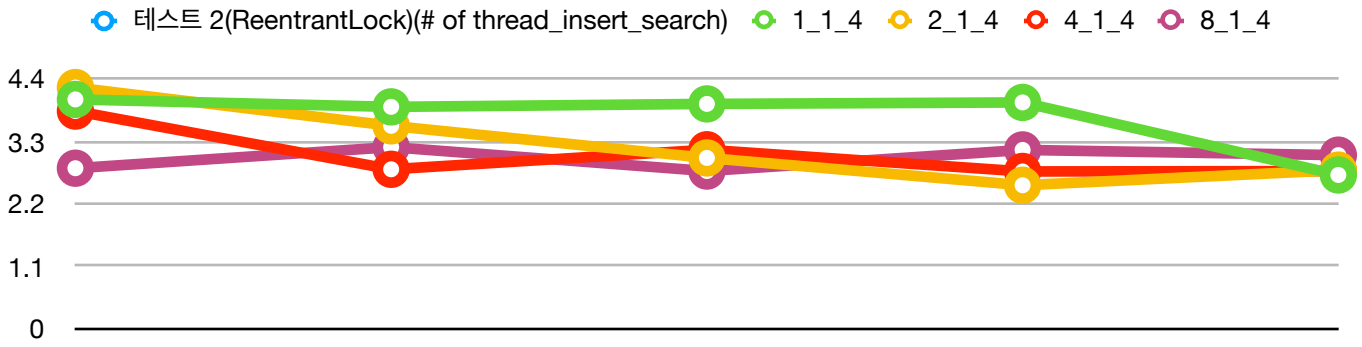
[3-c. BST를 구현한 경우, part 1에서 구현한 fine-grained BST와 비교해서 insert와 search 비율이 달라질때 성능에 대해서 분석하고 설명하세요.]

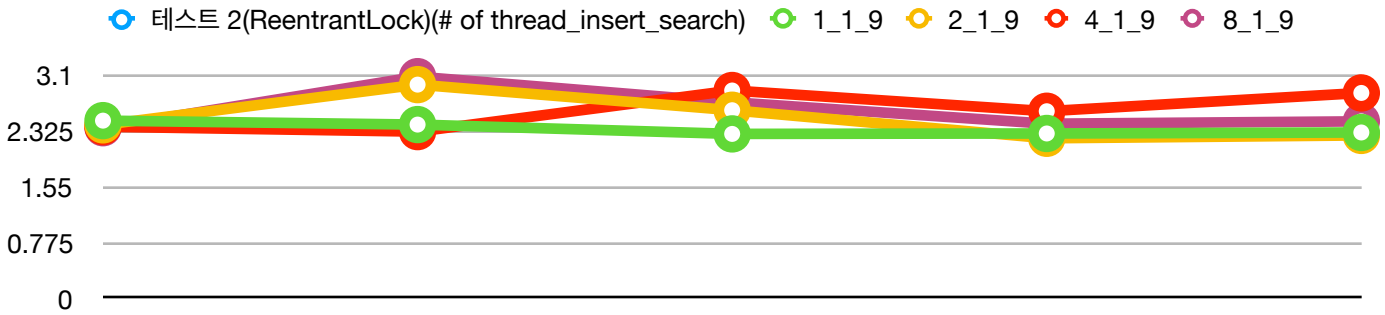
- 결과(insert/search의 비율에 따라)(>> : 시간차가 큰 경우)
  - 1:1
    - fine-grained : 실행시간이 1 >> 2,4(비슷) >> 8 으로 극명하다.
    - lock-free : 실행시간이 1 > 2 > 4,8(비슷) 하지만 시간차가 작아졌다.
  - 1:4
    - fine-grained : 실행시간이 1 >> 2,4,8(비슷) 하다.
    - lock-free : 실행시간이 1 > 2,4,8(비슷) 하다.
  - 1:9
    - fine-grained : 실행시간이 4 > 2,8(비슷) > 1 이다.
    - lock-free : 실행시간이 1 > 2,4,8(비슷) 이다.

[graph-part1]
[3-a]
[3-b]
- insert : search = 1 : 1
- insert : search = 1 : 4
- insert : search = 1 : 9









[graph-part2]
[3-a]
[3-b]
- insert : search = 1 : 1
- insert : search = 1 : 4
- insert : search = 1 : 9