Ⅰ. 프로젝트 분석

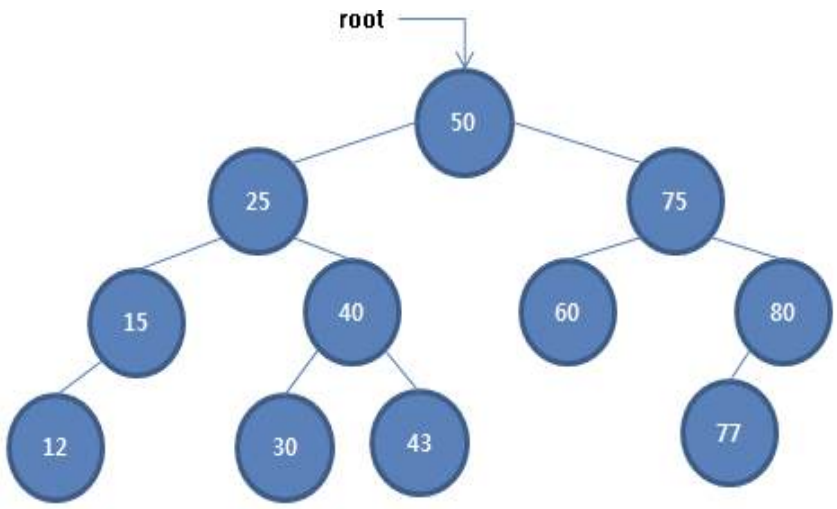
이번 프로젝트는 이진 탐색 트리 구현(Binary Search Tree)과 Multi Thread 를 사용하여 이진 탐색 트리 탐색, 삽입, 삭제 시 No Lock mechanism 일 때의 수행시간과 병행성을 유지하면서 충돌이 일어나지 않게 Coarse-grained Lock mechanism 과 fine-grained Lock mechanism 을 사용하여 여러 Thread 가 원활하게 수행되게 하는 Thread Safe Binary Search Tree 에서의 수행시간을 비교하기 위한 프로젝트이다.

아래에는 이번 프로젝트에 있어서 중심이 되는 이론적 배경을 서술한다.

▶ Binary Search Tree

이진 탐색 트리(binary search tree)의 모든 원소는 키를 가지고 있으며 동일한 키가 없다. 왼쪽 서브트리에 있는 키들은 루트의 키보다 작으며 오른쪽 서브트리에 있는 키들은 루트의 키보다 크다.

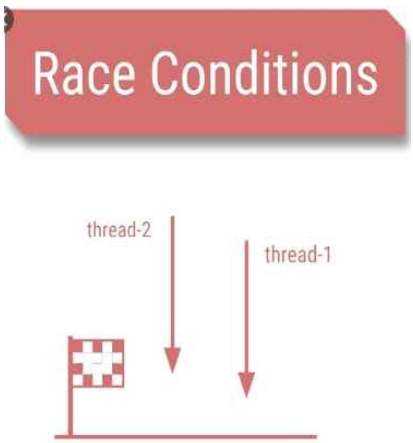
임의의 한 노드 및 데이터를 삽입, 삭제, 탐색하는 데에 유리하며 모두 'O(log n)'의 시간 복잡도를 가진다.



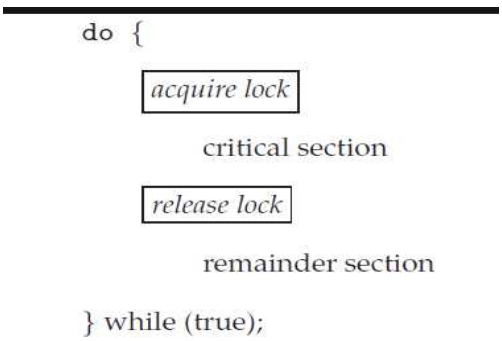
▶ Race Condition & Critical Section

'Race Condition'이란 Thread 를 사용하는 상황에 있어서 경쟁상태가 발생되는 것을 말한다. Thread 는 Process 와는 다르게 데이터를 공유하기 때문에 경쟁상태가 발생한다. 여러 Thread 가 동시에 수행되어지면 같은 데이터를 두고 서로 경쟁을 하기 때문에 원하지 않는 결과가 나올 수 있다.

'Critical Section(임계영역)'이란 동시적 조작에 있어 공유되는 부분을 말한다. 즉, 경쟁상태가 발생할 수 있는 부분이다. 임계영역에서 한 순간에 한 개의 Thread 만 데이터를 사용할 수 있게 상호배제를 보장해 주어야 올바른 결과를 얻을 수 있다.



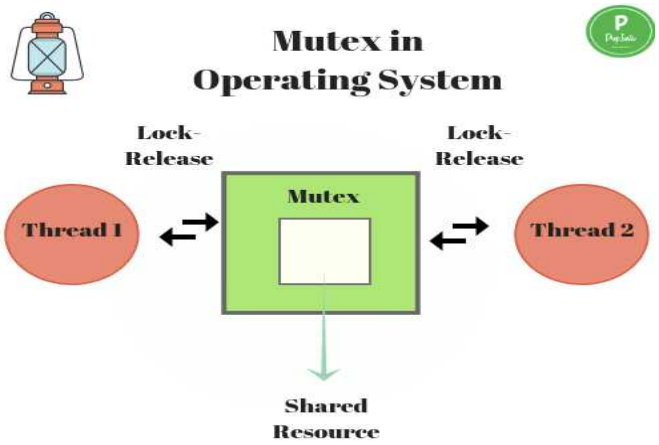
Race condition



Critical section

▶ Mutex Lock

'Mutex Lock(Mutual exclusive Lock)'은 상호배제를 보장해 준다는 의미이다. 상호배제는 위에서 원하는 결과를 얻기 위해서는 Race Condition 을 방지해야한다. Mutex 를 통해서 한 순간에 한 개의 Thread 만 임계영역에 접근 할 수 있도록 해야한다.

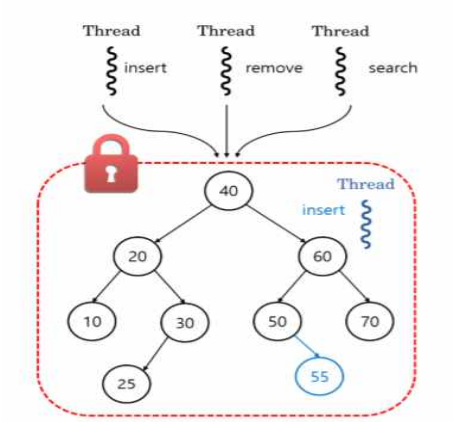


Mutual Exclusive Lock

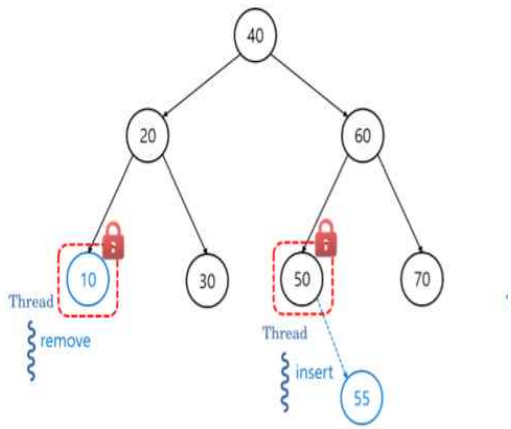
▶ Coarse-Grained Lock & Fine-Grained Lock

'Coarse-Grained Lock' 방식은 Critical Section 의 범위를 크게 잡아 많은 양의 데이터를 한 번에 보호하는 방식이다.

'Fine-Grained Lock' 방식은 Critical Section 의 범위를 작게 잡아 각 부분마다 Lock 을 사용하는 방식이다.



Coarse-Grained Lock



Fine-Grained Lock

Ⅱ. 설계 방향 설정

▶ Binary Search Tree

이진탐색트리의 기본 원칙에 맞춰서 설계를 구성하고 프로젝트 파일에 만들어야 할 코드에 맞춰서 이진탐색트리를 구성을 시도한다. 자료구조 시간에 배웠던 구성과는 조금 다른 부분이 있어서 구조를 파악하고 설계에 접근 하는 방법을 사용한다.

```
void* thread_job_insert(void* arg) {

    thread_arg* th_arg = (thread_arg*)arg;
    lab2_tree* tree = th_arg->tree;
    int is_sync = th_arg->is_sync;
    int* data = th_arg->data_set;
    int start = th_arg->start, end = th_arg->end;
    int i;

    for (i = start; i < end; i++) {
        lab2_node* node = lab2_node_create(data[i]);
        if (is_sync == LAB2_TYPE_FINEGRAINED)
            lab2_node_insert_fg(tree, node);
        else if (is_sync == LAB2_TYPE_COARSEGRAINED)
            lab2_node_insert_cg(tree, node);
        else
            lab2_node_insert(tree, node);
    }
}
```

```
/*
 * multi thread insert test nolock
 */
is_sync = LAB2_TYPE_NOLOCK;
tree = lab2_tree_create();

gettimeofday(&tv_insert_start, NULL);
for (i = 0; i < num_threads; i++) {
    thread_arg* th_arg = &threads[i];
    th_arg->tree = tree;
    th_arg->is_sync = is_sync;
    th_arg->data_set = data;
    th_arg->start = i * term;
    th_arg->end = (i + 1) * term;

    pthread_create(&threads[i].thread, NULL, thread_job_insert, (void*)th_arg);
}

for (i = 0; i < num_threads; i++)
    pthread_join(threads[i].thread, NULL);

gettimeofday(&tv_insert_end, NULL);
exe_time = get_timeval(&tv_insert_start, &tv_insert_end);
print_result(tree, num_threads, node_count, is_sync, LAB2_OPTYPE_INSERT, exe_time);
lab2_tree_delete(tree);
```

본래 작성되어 있던 test 의 single thread 부분을 multi thread 로 바꿔주는 수정이 필요했다.

▶ Lock

쓰레드가 2 개 이상인 멀티쓰레드 환경에서 lock 을 사용하지 않았을 때의 결과와 두 종류의 lock(Coarse-grained, Fine-grained)을 사용하였을 때의 결과를 확인 및 비교할 수 있다.

1) Coarse-grained

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

```
int lab2_node_remove_cg(lab2_tree* tree, int key) {
    // You need to implement lab2_node_remove_cg function.
    pthread_mutex_lock(&mutex); // lock 범위를 함수 전체로 지정
    lab2_node* pp = NULL;  // 부모의 부모노드를  NULL로 초기화
    lab2_node* p = tree->root; // 부모노드를 트리의 루트로 초기화

    if (p == NULL) {  // 빈 트리일 경우
        pthread_mutex_unlock(&mutex);  // lcok 해제
        return LAB2_ERROR;  // 에러 리턴
    }

    while (p != NULL && p->key != key) {  // 삭제 위치 탐색
        pp = p;
        if (p->key < key) {  // 탐색 데이터가 p의 데이터보다 큰 경우
            p = p->right; // p는 p의 오른쪽 자식으로
        }

        else if (p->key > key) {  // 탐색 데이터가 p의 데이터보다 작은 경우
            p = p->left;  // p는 p의 왼쪽 자식으로
        }
    }

    if (p->left != NULL && p->right != NULL) {
        lab2_node* s = p->left;  // s를 p의 왼쪽자식으로
        lab2_node* ps = p;       // ps를 p 노드로

        while (s->right != NULL) {  // 왼쪽 서브트리의 최댓값 탐색
            ps = s;
            s = s->right;
        }
        p->key = s->key;
        p = s;
        pp = ps;
    }

    lab2_node* temp = (lab2_node*)malloc(sizeof(lab2_node));  // temp노드 동적 할당

    if (p->left == NULL) {  // p의 왼쪽 자식이 없는 경우
        temp = p->right;  // p의 오른쪽 자식을 temp으로
    }

    else {  // 반대 경우
        temp = p->left;  // p의 왼쪽 자식을 temp로
    }

    if (p == tree->root) { // p가 루트인 경우
        tree->root = temp; // temp를 트리의 루트로
    }

    else {  // 반대 경우
        if (p == pp->left) {  // p가 pp의 왼쪽노드와 같으면
            pp->left = temp;  // temp를 pp의 왼쪽노드로
        }

        else pp->right = temp;  // 반대경우, temp를 pp의 오른쪽 노드로
    }
    lab2_node_delete(p);  // p 노드 삭제
    pthread_mutex_unlock(&mutex); // lock 해제
    return LAB2_SUCCESS;
}
```

예외처리를 위한 unlock을 제외하고 함수의 제일 윗부분에 lock, 제일 아래부분에 unlock 을 사용하여 Coarse-grain을 구현하였다.

2) Fine-grained

lock 을 거는 위치를 노드에 관한 부분에 대해서 작은 부분으로 나눠 모든 부분에 lock/unlock 을 걸어 보는 방식으로 설계를 시도한다. 무분별하게 lock 을 많이 사용하는 방식은 자제하도록 해야한다. lock 이 많아지고 프로그래머가 lock 을 알맞은 위치에 사용하지 않으면 deadlock 에 빠질 수 있으므로 무분별하게 많은 lock 을 사용하는 방식은 자제한다.

```
int lab2_node_remove_fg(lab2_tree* tree, int key) {
    // You need to implement lab2_node_remove_fg function.

    lab2_node* pp = NULL;  // 부모의 부모노드를 NULL로 초기화
    lab2_node* p = tree->root; // p를 트리의 루트로 초기화

    if (p == NULL) {    // 빈 트리인 경우
        return LAB2_ERROR;  // 에러 리턴
    }

    while (p != NULL && p->key != key) {  // 삭제 위치 탐색
        pp = p;
        if (p->key < key) {  // 탐색 데이터가 p의 데이터보다 큰 경우
            p = p->right;  // p는 p의 오른쪽 자식으로
        }

        else if (p->key > key) {  // 탐색 데이터가 p의 데이터보다 작은 경우
            p = p->left;  // p는 p의 왼쪽 자식으로
        }
    }

    pthread_mutex_lock(&mutex);  // 다른 삭제 노드 데이터가 들어오지 않도록 lock
    if (p->left != NULL && p->right != NULL) {
        lab2_node* s = p->left;  // s를 p의 왼쪽자식으로
        lab2_node* ps = p;  // ps를 p 노드로

        while (s->right != NULL) {  // 왼쪽 서브트리의 최댓값 탐색
            ps = s;
            s = s->right;
        }
        p->key = s->key;
        p = s;
        pp = ps;
    }

    lab2_node* temp = (lab2_node*)malloc(sizeof(lab2_node));  // temp노드 동적 할당

    if (p->left == NULL) {  // p의 왼쪽 자식이 없는 경우
        temp = p->right;  // p의 오른쪽 자식 노드를 temp로
    }

    else {  // 반대 경우
        temp = p->left; // p의 왼쪽 자식노드를 temp로
    }

    if (p == tree->root) {  // p가 루트인 경우
        tree->root = temp;  // temp를 트리의 루트로
    }

    else {  //반대 경우
        if (p == pp->left){  // p가 pp의 왼쪽 노드와 같으면
            pp->left = temp;  // temp를 pp의 왼쪽노드로
        }
        // 반대 경우
        else pp->right = temp; // temp를 pp의 오른쪽노드로
    }
    lab2_node_delete(p);  // p 노드 삭제
    pthread_mutex_unlock(&mutex); // lock 해제
    return LAB2_SUCCESS;
}
```

실질적인 값 삭제가 일어나는 부분에만
lock을 걸어주어 Fine-grain을 구현하였다.

Ⅲ. 실행 결과 및 결과 분석

1)

Thread	4	Node	10000

```
root@ubuntu:/home/ukkk/2020_DKU_OS/lab2_sync# ./lab2_bst -t 4 -c 10000
=====  Multi thread no lock BST  insert experiment  =====

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.005966 seconds


 BST inorder iteration result :
    total node count    : 10000

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

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.010080 seconds


 BST inorder iteration result :
    total node count    : 10000

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

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.002657 seconds


 BST inorder iteration result :
    total node count    : 10000
```

```
=====  Multi thread no lock BST  delete experiment  =====

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.002156 seconds


 BST inorder iteration result :
    total node count    : 10000

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

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.001913 seconds


 BST inorder iteration result :
    total node count    : 10000

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

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.001766 seconds
```

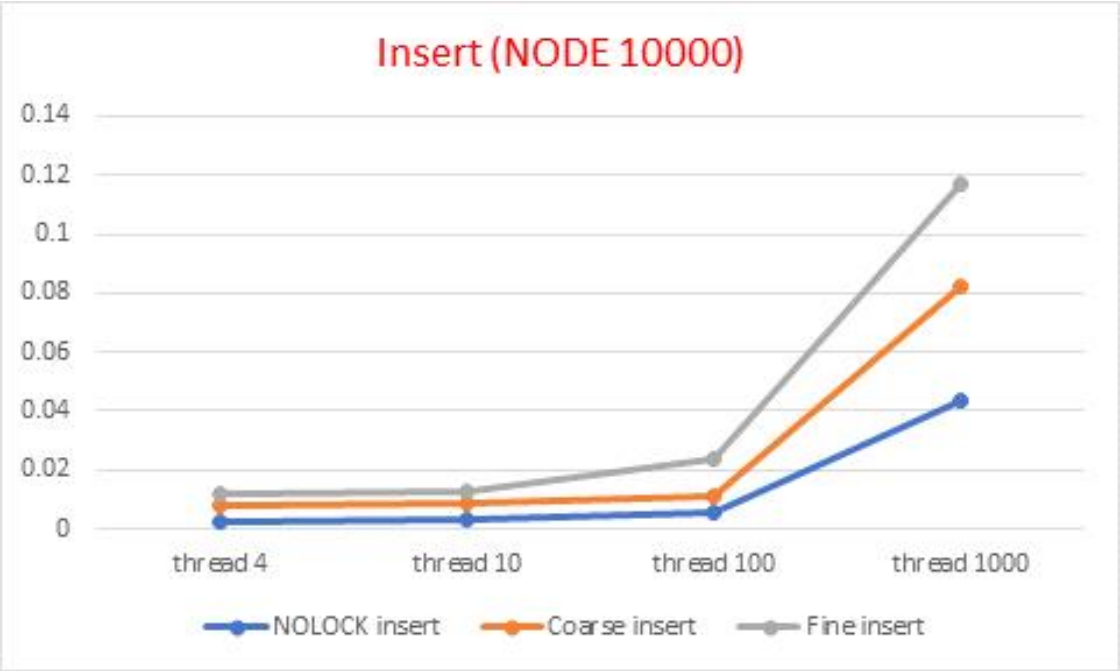
2)

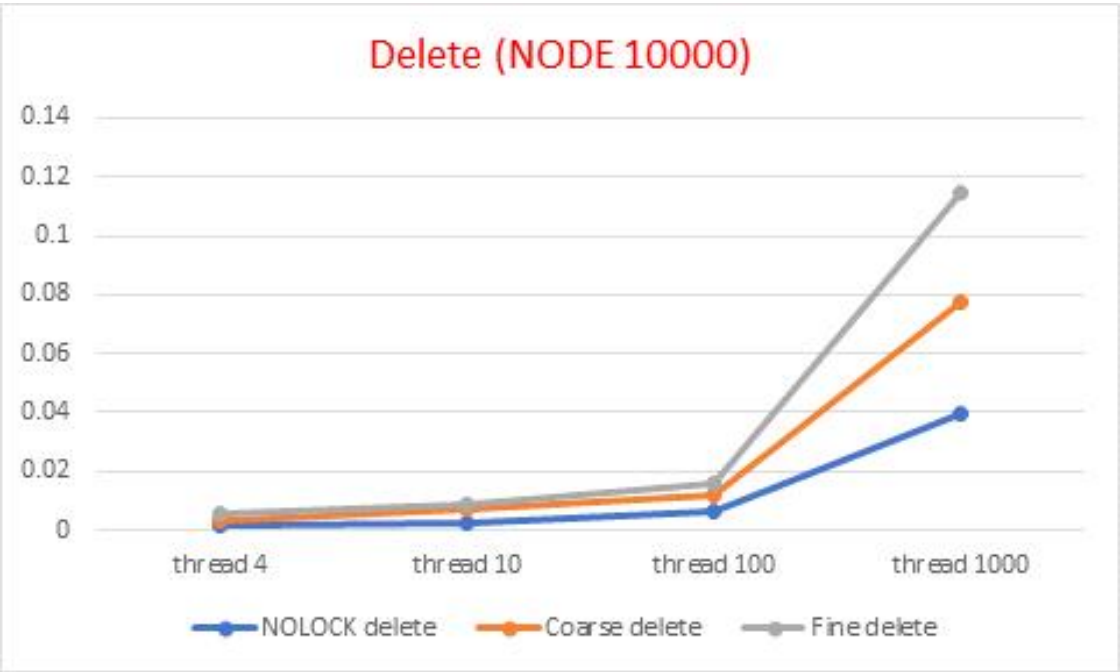
Thread	4	Node	100000

```
root@ubuntu:/home/ukkk/2020_DKU_OS/lab2_sync# ./lab2_bst -t 4 -c 100000
=====  Multi thread no lock BST  insert experiment  =====

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.059880 seconds


 BST inorder iteration result :
    total node count    : 100000

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

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.062166 seconds


 BST inorder iteration result :
    total node count    : 100000

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

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.058164 seconds


 BST inorder iteration result :
    total node count    : 100000
```

```
=====  Multi thread no lock BST  delete experiment  =====

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.047428 seconds


 BST inorder iteration result :
    total node count    : 100000

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

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.054355 seconds


 BST inorder iteration result :
    total node count    : 100000

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

 Experiment info
    test node           : 100000
    test threads        : 4
    execution time      : 0.051253 seconds


 BST inorder iteration result :
    total node count    : 100000
```

▶ 실행 결과 분석





실행을 해보니 node의 개수가 100000이상이 되면 delete부분에서 segment falut가 빈번하게 (대략 80%의 확률) 발생하였다.

우리는 성능이 fine > coarse > nolock 순일 것 이라고 예상했고 결과 역시 우리가 예상한 것의 70~80%정도로 결과가 나왔다.

NODE 100	thread 4	thread 10	thread 100	thread 1000
NOLOCK insert	0.000142	0.000568	0.003189	0.040866
Coarse insert	0.000121	0.000409	0.002738	0.031409
Fine insert	0.000069	0.000386	0.002312	0.031166
NOLOCK delete	0.000063	0.000212	0.002744	0.031021
Coarse delete	0.00072	0.000583	0.003413	0.032153
Fine delete	0.000062	0.000221	0.002259	0.031902
NODE 1000	thread 4	thread 10	thread 100	thread 1000
NOLOCK insert	0.000394	0.000543	0.003121	0.050075
Coarse insert	0.000232	0.000674	0.003135	0.036919
Fine insert	0.000227	0.000371	0.003023	0.031278
NOLOCK delete	0.000332	0.000604	0.006159	0.034058
Coarse delete	0.000394	0.000353	0.003151	0.034371
Fine delete	0.000232	0.000322	0.002983	0.032885
NODE 10000	thread 4	thread 10	thread 100	thread 1000
NOLOCK insert	0.002472	0.003089	0.005771	0.043852
Coarse insert	0.005365	0.005296	0.005206	0.038488
Fine insert	0.004042	0.004328	0.012397	0.034593
NOLOCK delete	0.001737	0.002037	0.006597	0.039251
Coarse delete	0.001805	0.004872	0.004984	0.038057
Fine delete	0.001719	0.002202	0.004635	0.037707

thread의 개수는 10000일 때는 빈번하게 segment fault가 발생하고 10000이 넘어가면 무조건 segment fault가 발생하여 segment fault가 발생하지 않는 범위 내에서 성능을 측정하였다.

Ⅳ. 개인 고찰

< 32163006 이건욱 >

이번 과제는 정말 역대급으로 어려운 과제였던 것 같다. 자료구조 시간과 알고리즘 시간에 bst에 대해서는 공부해봤기 때문에 lock의 위치만 잘 잡아주면 되겠다 생각하고 시작해보았지만 fine grain에서 lock의 위치를 잡기가 쉽지 않았고 여러 시도 끝에 삽입과 삭제가 되는 부분에만 lock과 unlock을 넣어 fine grain을 만들었다.

처음에는 단순히 coarse grain은 시작되는 부분에 lock 끝부분에 unlock, fine grain은 모든 부분에 lock과 unlock이라고 생각했지만 구글에 검색을 하고 이론을 많이 찾아보니 coarse grain과 fine grain이 단순히 lock의 개수 차이가 아니라고 생각이 바뀌게 되었다. 그 결과 지금 우리의 코드처럼 얼핏 보면 비슷해 보이지만 coarse grain은 전체적으로 insert, delete에 락을 걸어주었고 fine grain에는 데이터의 값이 변경되는 부분 즉 검사하는 부분이 아닌 직접적으로 삽입되거나 삭제되는 부분에만 lock을 걸어주어 굳이 검사하는 부분과 같이 lock이 필요치 않다고 생각되는 부분에는 lock을 걸지 않았다. 즉, 실질적인 critical section은 insert와 delete에서의 실질적인 삽입과 삭제가 되는 부분으로 생각하고 lock을 걸어주었다.

또한 coarse grain과 fine grain에서 각각 return이 될 때마다 unlock을 해주었다. 수업 시간에 교수님께 배운 내용 중 lock과 unlock의 개수를 맞춰주는 것이 바람직하고 버그 발생을 줄일 수 있다고 배웠기에 예외처리를 제외한 부분에서는 lock과 unlock의 짝을 맞추어 deadlock이 발생되지 않도록 주의하면서 코드를 작성하였다.

Node의 개수가 100000이 넘어가면 segment fault가 발생하는 부분을 kernel의 개념쪽에서 더 이해하여 수정해보려고 했으나 과제 제출기간 내에 이 문제를 해결하는 것은 시간적으로 부족하다고 느껴 이번에는 코드 자체에 초점을 맞추고 앞으로 이 부분에 대해 더 깊이 공부해볼 생각이다. 또한 이번 과제가 끝나면 코드를 수정하여 segment fault가 발생하면 프로그램이 멈추는 것이 아니라 segment fault가 발생하면 그 부분을 예외처리하고 다른 부분은 정상적으로 진행할 수 있도록 수정하고 싶다.

이번 과제를 수행함으로써 lock과 unlock에 대한 이해와 coarse grain과 fine grain의 차이에 대한 이해를 높이는데 많은 도움이 되었고 bst에 대한 이해도도 높아지는 계기가 되었던 것 같다.

< 32164420 조정민 >

이번 과제는 자료구조 시간 때 배운 이진탐색트리 BST(Binary Search Tree)를 바탕으로 thread와 node수가 변함에 따라 **1) lock을 사용하지 않은 경우, 2) Coarse-grained lock을 사용한 경우, 3) Fine-grained lock**을 사용한 경우에 삽입, 삭제에 대한 실행시간이 어떻게 달라지는지 확인해보는 프로젝트였다.

프로젝트를 시작하기 전, 우리의 생각은 굉장히 단순했다. BST를 구현한 후, 삽입과 삭제에 대한 함수에 lock/unlock의 위치만 잡아주면 된다고 생각했다. 하지만 우리의 생각과는 다르게 BST를 구현하는 것부터 어려움을 겪었다. 교수님께서 올려주신 BST 코드는 자료구조 시간 때 배운 BST의 구조와 달랐고, 그 틀에 맞추서 BST를 구현해야했고 lock을 거는 부분을 찾아줘야 했다.

BST를 구현한 후, lock/unlock의 위치를 잡는 것은 정말 힘들었다. lock/unlock의 확실한 위치를 찾는 것을 하지 못해 위치 변경을 정말 많이 했고 많은 오류를 접했다.

Coarse-grained의 경우에는 임계영역과는 상관없이 삽입과 삭제 함수 전체에 lock/unlock을 걸어주었다. Coarse-grained는 어려움 없이 해결할 수 있었지만 구현하는 와중에 실수 몇 가지가 있었다. 그 중 하나는 조건문을 통하여 return 하는 부분에 당연히 unlock을 해주어야함에도 불구하고 unlcok을 해주지 못해 thread들이 접근하지 못하는 문제가 발생하는 것이었다. 이 문제는 각 조건문의 return 부분을 지워주고 함수가 끝나는 부분에만 return과 unlock을 해주며 해결할 수 있었다. 이로 인해 lock과 unlock의 짝을 맞추는 것도 가능하였다.

Fine-grained의 삽입과 삭제 함수에서 lock을 거는 위치를 찾기가 정말 힘들었다. 처음에 새로 삽입할 데이터의 위치를 찾기 위한 탐색 부분에도 lock을 사용하였다. insert_fg 함수에서는 삽입이 이루어지는 부분에 작은 범위로 lock/unlock을 사용하였지만 remove_fg 함수에서는 lock/unlock을 사용해야하는 부분을 명확하게 찾아내기 어려웠다. 우리의 코드에서는 왼쪽 서브트리의 최댓값을 찾아주는 while문 밖에 lock을 사용하고 함수가 끝나는 부분에 unlock을 사용하였다. 비교적 큰 lock 범위였다. 좀 더 좁은 범위로 lock/unlock을 사용하고 싶었지만 쉽지 않았고, 이렇게 하는 것이 최선이라 생각하였다. 오류없이 더 작은 범위로 lock을 사용하지 못한 것에 대해서는 아직까지 아쉬움이 든다.

실행 결과를 보면, 노드 수에 따라 수행시간이 달라지는 모습을 볼 수 있었다. 노드 수가 많아짐에 따라 수행 시간이 길어졌고, segment fault가 뜨는 결과 또한 확인할 수 있었다. 내 우분투 환경에 대한 문제겠지만 대게 노드 수가 50000 이상인 경우부터 높은 확률로 segment fault가 떴다.

과제를 마무리하면서 돌이켜보니 이번 과제는 정말 어려웠다는 생각이 든다. multi thread 환경에서 필요한 lock 메카니즘을 직접 구현해보면서 Coarse, Fine-grained lock에 대한 전반 적인 이해와 코드에 어떻게 적용을 시켜야하는지 배울 수 있었다. 운영체제 수업이 아니라면 이런 문제를 확인하기 위한 프로젝트는 진행해보지 못했을 것이다. 직접 코드를 작성하고 구글링해서 찾은 부분을 참고하여 코드에 적용시켜보는 것도 우리를 발전시키는 부분 중의 하나라고 생각했다. 완벽하진 않지만 이런 결과를 도출해낼 수 있었다는 부분에 뿌듯함을 느꼈고 나를 한 층 더 성장시킬 수 있었던 좋은 시간이었다.