



OPERATING SYSTEM LAB2 ***[LAB 2 SYNCHRONIZATION]***

담당 교수 최종무 교수님

학과 소프트웨어학과

팀원/학번 나현진_32177240

이지언_32173478

제출일 2020년 5월 6일

<https://github.com/realgenie/Lab2>

목차

1. 들어가기에 앞서
2. 접근 설명 및 코드 분석
3. 출력 결과
4. Discussion

역할

나현진	BST에 관한 기본 구조 구현, insert 부분 구현, 보고서 작성
이지언	remove 부분 구현, delete 부분 구현, 코드 오류 분석 및 수정, 보고서 작성 도움

1. 들어가기에 앞서

이번 'Lab2 Synchronization'과제는 운영체제 수업 시간에 배운 'Lecture Note 4. Concurrency: Thread and Lock' 내용을 기반으로 한 과제로, 사전에 개념에 대한 확실한 이해가 필요하다.

공유 자원(Shared resource)에 동시에 Multi-thread 가 접근 하게 될 경우, 경쟁 상태(race condition)에 빠지게 된다. 그렇게 되면 동시성, 병행성(Concurrency)에 문제가 발생하기 때문에 해결 방안으로 병행성 제어가 필요하다.

병행성(Concurrency)에는 2 가지 요구 조건이 존재한다.

첫 번째, **상호 배제(Mutual exclusion)**는 임계 영역(Critical Section)에 한 순간에 하나의 Thread 만 들어갈 수 있도록 해주는 것이다.

두 번째, **동기화(Synchronization)**는 한 thread 가 다른 thread 가 종료될 때까지 기다리는 순서를 보장하는 것이다.

Multi-thread 는 Thread 간 정보를 공유하기 때문에 임계 영역(Critical Section)의 부분을 잘 파악하고, 이 부분에 대해서 상호 배제(Mutual exclusion)를 통해 보호하여 경쟁 상태(race condition)가 일어나지 않도록 해야 한다.

Data Structure 로는 BST(Binary Search Tree)를 이용하여 만드는 것이다. 지난 학기 자료구조 시간에 만들었던 트리를 이용하여 구현하는 것으로, 트리에 여러 Thread 가 동시에 들어와 수행되는 경우, 경쟁 상태(race condition)가 발생한다.

- ⇒ 공유 자원들을 잘 동기화(Synchronization)된 방법으로 접근할 수 있도록, pthread 기반 mutex(lock/unlock)를 잘 구현하여 순서를 주고, 상호 배제(Mutual exclusion)를 보장하여 경쟁 상태(race condition)를 완화 하는 것이 과제의 핵심 포인트이다.

2. 접근 설명 및 코드 분석

lab2_sync 파일 안에 있는 lab2_bst_test.c와 include/lab2_sync_types.h의 부분을 참고하며 **lab2_bst.c**를 수정하는 형식으로 진행했다.

■ Mutex

Mutex 구현에 대한 사전 지식이 필요하다고 생각하여 다음과 설명을 하였다.

Mutex lock은 하나의 Thread만 특정 영역에 접근 할 수 있도록 하는 Mutual Exclusion을 통해 Synchronization을 제공하는 기법이다.

Mutex의 특징은 다음과 같다.

첫 번째, mutex의 lock은 atomic operations으로 작동한다는 것이다. 하나의 Thread가 mutex를 이용해서 lock을 하게 될 경우, 다른 Thread가 lock을 할 수 없다.(Atomicity) **두 번째**, Thread가 lock을 했을 경우, unlock을 해주기 전까지 다른 Thread는 lock을 할 수 없다.(Singularity) **세 번째**, 하나의 Thread가 lock을 하고, unlock을 해주기 전까지 해당 지점에 머물러 CPU의 자원을 소비하지 않는 것이다.(No-Busy Wait) 하지만 Sleep lock의 경우, CPU를 반납하고, unlock시 깨어나지만 Spin lock의 경우, 조건을 계속 확인해주며 CPU를 사용한다(Busy Wait). 위 3가지는 Sleep lock의 경우의 특성을 설명한 것이며, Spin lock의 경우는 CPU를 반납하지 않고 자원을 소비한다.

이에 기존 bonus 문제였던 Spin lock 구현은 critical section이 short 할 경우에 고려하여 구현 했어야 할 것이다.

mutex를 사용하려면 변수 선언, 초기화 등 적절한 함수를 포함하여야 한다.

mutex를 생성하기 위해 mutex에 대한 정보를 저장하기 위한 type인 pthread_mutex_t를 선언해 주고, PTHREAD_MUTEX_INITIALIZER 상수를 할당해 주어 이를 초기화 해주어야 한다. 초기화 방법에는 정적 초기화와 동적 초기화가 있다. 우리는 정적 초기화를 통해 다음과 같이 lab2_bst.c 파일 상단에 mutex를 선언해 주고, 변수를 초기화 했다.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>
#include <string.h>

#include "lab2_sync_types.h"

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

■ BST(Binary Search Tree)

이번 과제는 Concurrent Data Structure를 만드는 것으로, BST(Binary Search Tree)를 이용하였다. 이진 트리(BST)의 기본 속성으로는 left 자식 노드의 값이 부모 노드보다 더 작은 값을 가지고, right 자식 노드의 값이 부모 노드보다 더 큰 값을 가지는 트리이다. 공유 자원(Shared resource)을 update 하는 부분인 BST를 구성 하도록 한다.

int lab2_node_print_inorder(lab2_tree *tree)

이 함수는 tree를 받아 tree에 노드가 insert, delete하는 것을 확인할 수 있도록 해준다. 또한, tree 노드를 순회(Traversal)하는 연산이 들어가는 부분으로, 순회 순서에는 여러 가지가 존재하지만 우리는 중위 순회(Inorder Traversal)를 구현하도록 한다. 중위 순회(Inorder Traversal)는 left->root->right의 순서를 가진다.

root가 NULL인지 확인을 한 후, NULL이 아닐 경우, 노드에 있는 p를 이용하여 left에 p를 가리키게 하고, 그 다음에 root, 마지막으로 right를 가리키게 한다.

lab2_tree *lab2_tree_create()

함수 내에 tree를 포인터 선언해주어 malloc을 통해 구조체의 크기만큼 메모리를 동적 할당 받는다. 또한, tree->root 부분을 초기화 해준다.

처음에 이 부분에 있어서 의문점이 들었는데, tree root부분이 NULL이 아닐 경우, mutex가 초기화가 안되어 있을 경우 문제가 생길 것을 대비하여 mutex_init()을 통해서 tree부분에 대한 mutex를 초기화 해주려고 하였다.

```
lab2_tree *lab2_tree_create() {
    lab2_tree *tree = (lab2_tree *)malloc(sizeof(lab2_tree));
    //pthread_mutex_init(&tree->mutex, NULL);
    tree->root = NULL;
    return tree;
}
```

하지만 위와 같이 mutex_init()을 해줄 경우, 오류가 발생하였다. 그렇기에 노드를 생성하는 부분에서만 mutex_init을 해주도록 하고, 이를 이용할 수 있도록 하였다.

lab2_node *lab2_node_create(int key)

함수 내에서 node를 포인터로 선언해 주어 malloc을 통해 구조체의 크기만큼 메모리를 동적 할당 받는다. 이때, tree에 mutex_init이 되어 있는 노드를 이용할 수 있도록 pthread_mutex_init()를 호출하여, 노드 부분의 mutex를 NULL로 초기화 해주었다. key에는 매개변수로 받은 key를 주며, 나머지 left, right 포인터 부분에 있어서는 NULL로 초기화해준다.

Insert, Remove 부분에서는 임계 영역(critical section)에 대해서 잘 파악하고, lock, unlock을 잘 구현할 수 있도록 해야 한다.

pthread_mutex_lock(), pthread_mutex_unlock() 함수 관련해서는 아래 Coarse-grained lock과 Fine-grained lock에서 구현을 하며 설명을 하도록 하겠다.

■ Insert

BST에서 Insert는 root 노드에서부터 탐색을 하며 insert하는 노드의 값이 현재 가지고 있는 노드보다 더 크면 right로, 작으면 left로 insert된다.

int lab2_node_insert(lab2_tree *tree, lab2_node *new node)

k를 포인터로 선언해 주어, 새로운 노드를 생성할 수 있도록 한다. lab2_node *k = lab2_node_create(new_node->key); 그렇기에 tree->root가 NULL일 경우, 새 노드를 삽입한다.

함수 내에서 p, q를 포인터로 선언해 주어, 각각 tree->root와 NULL이도록 한다. q는 while문 안에 있는 q=p에 있어서 뒤따라 나오는 포인터이다.

p가 NULL이 아닐 경우에 있어서 while문을 해주었다. 동일한 우선 순위가 있을 경우, return 값을 -1로 주었다. 새로운 노드의 key값이 p의 key값보다 작으면 left로 내려가게 된다. 하지만 새로운 노드의 key값이 더 클 경우, right로 내려간다. NULL이 되면 insert가 되고 나서 빠져나온다.

앞에서 k를 포인터로 선언해 주어, 새로운 노드를 생성해 줬었다. 이를 이용하여 새로운 노드가 q의 key 값보다 작게 되면 q의 left에 노드를 insert한다. 그 반대로 크게 될 경우, right에 insert를 하도록 한다.

lock을 임계 영역(Critical Section)의 범위를 얼마나 세부적으로 설정해 주는 지에 따라서 Coarse-grained lock과 Fine-grained lock으로 나눌 수 있다. 이는 우리가 구현하는 insert와 remove 부분에 있어서 동일하게 적용되는 것이다.

Coarse-grained lock은 insert, remove, search 등의 연산 수행에 있어서 각 연산이 수행 될 때, 다른 Thread가 접근하지 못하도록 tree 전체를 임계 영역(Critical Section)으로 간주하여 lock을 사용하도록 한다.

Fine-grained lock은 임계 영역(Critical Section)의 범위를 tree 전체가 아닌 각 노드 별로 설정하여 lock을 사용하도록 한다.

경쟁 상태(race condition)를 막고, BST의 연산이 안전하게 수행될 수 있도록 lock/unlock을 각각 구현하였다.

int lab2_node_insert_fg(lab2_tree *tree, lab2_node *new node)

lab2_node_insert 부분의 함수와 동일하나 임계 영역(Critical Section)의 범위를 노드 별로 설정해 주어야 한다.

다음과 같이 세분화하여 임계 영역(Critical Section) 부분 앞 뒤로 노드 별로, lock, unlock을 구현하였다.

```
if(new_node->key < q->key) {  
    pthread_mutex_lock(&q->mutex);  
    q->left = k;  
    pthread_mutex_unlock(&q->mutex);  
}  
else {  
    pthread_mutex_lock(&q->mutex);  
    q->right = k;  
    pthread_mutex_unlock(&q->mutex);  
}
```

그렇기에 다른 연산들이 접근하기 이전에 tree의 노드에 lock, unlock을 해주어 상호 배제(mutex exclusion)가 보장된다.

int lab2_node_insert_cg(lab2_tree *tree, lab2_node *new node)

lab2_node_insert 부분의 함수와 대부분 동일하지만, while문에서 추가하려는 노드의 key값이 tree에 있으면 return -1을 해 주는데, 이때, coarse grained는 unlock을 해 줘야 한다. 하지만 이렇게 되면, lock과 unlock이 짝이 맞지 않으므로, 추후에 bug가 발생할 수 있다.

따라서, 추가하려는 노드가 tree에 이미 있을 경우, while문에서 break로 나오고, if~else if에 조건이 충족되지 못하므로, 삽입이 이루어 지지 않은 채, unlock되게 된다.


insert_fg와 달리 tree 전체를 임계 영역(Critical Section)으로 lock, unlock을 구현해 주어야 한다. 그렇기 때문에 tree가 시작하는 부분과 끝나는 지점에 각각 lock, unlock을 구현했다. fg와는 다르게 인자가 &tree->root->mutex가 된다. 이 부분이 다른 insert함수와 다른 부분이다.

```
pthread_mutex_lock(&tree->root->mutex);
lab2_node *p= tree->root;
lab2_node *q = 0;

while(p!=NULL){
    q=p;
    if((new_node->key) == (p->key)){
        break;
    }
    if((new_node->key)<(p->key))
        p=p->left;
    else
        p=p->right;
}

if(new_node->key < q->key)
    q->left=k;
else if(new_node->key > q->key)
    q->right=k;

pthread_mutex_unlock(&tree->root->mutex);
return 0;
```



다른 연산들이 접근하기 이전에 tree 전체에 lock, unlock을 해주어 상호 배제(mutex exclusion)가 보장된다.

■ Remove

BST에서 Remove는 Insert 부분보다 복잡하며, tree를 순회하며 삭제해야 하는 값을 발견 시 노드를 삭제한다. 그 자리에 tree에 있는 다른 노드를 대체해주어야 한다.

int lab2_node_remove(lab2_tree *tree, int key)

Tree에 아무 노드도 없다면 return -1 을 한다.

노드 포인터 p에 tree의 root를 가리키게 하고 p->key가 key보다 작으면 left로, p->key가 key보다 크면 right로 이동하면서 p가 찾으려는 key값의 노드를 가리킬 때까지 while문을 실행한다. 이 때, q는 p노드의 부모 노드를 가리키도록 while문에 설정한다.

[단말 노드의 삭제]

삭제할 노드 p가 단말노드라면, 부모 노드인 q가 가리키는 p의 위치의 child node값을 Null로 만들어서 노드를 tree에서 삭제 시킨다.

[하나의 자식을 갖는 노드 삭제]

삭제할 노드 p가 하나의 자식을 가지고 있으면 부모 노드 q가 가리키는 p의 위치를 자식 노드의 위치로 변경한다.

[두개의 자식을 갖는 노드 삭제]

삭제할 노드 p가 두개의 자식을 갖고 있으면, p의 왼쪽 트리 중 key가 가장 큰 노드를 p로 대체한다. 이를 위해 포인터 k를 p의 왼쪽 자식으로 초기화 하고, k를 오른쪽 자식으로 이동하면서 왼쪽 자식 트리의 가장 큰 값을 찾아 가리킨다. 만약 p의 왼쪽자식이 오른쪽 자식이 없을 경우, p->key를 k->key로 변경한 후, k->left가 존재하면 p->left를 k->left로 바꿔주고, 존재하지 않으면, p->left=0으로 한다. 만약 p의 왼쪽 자식이 오른쪽 자식을 가질 경우, p->key를 k->key로 변경한 후, k->left가 존재하면, q->right를 k->left로 바꿔주고, 존재하지 않으면, 0으로 한다.

int lab2_node_remove_fg(lab2_tree *tree, int key)

lab2_node_remove 부분의 함수와 동일하나 임계 영역(Critical Section)의 범위를 노드 별로 설정해 주어야 한다.

다음과 같이 세분화하여 트리의 노드 값이 바뀌는 임계 영역(Critical Section) 부분 앞 뒤로 노드 별로, lock, unlock을 구현하였다.

```

//단말노드 일 경우
if((p->left==NULL)&&(p->right==NULL)){
    if(q->left==p){
        pthread_mutex_lock(&p->mutex);
        q->left = NULL;
        pthread_mutex_unlock(&p->mutex);
    }
    else if(q->right==p){
        pthread_mutex_lock(&p->mutex);
        q->right = NULL;
        pthread_mutex_unlock(&p->mutex);
    }
}

//하나의 자식을 갖는 노드일 경우 (왼쪽자식)
if((p->left!=0)&&(p->right==0)){
    if(q->left==p){
        pthread_mutex_lock(&p->mutex);
        q->left = p->left;
        pthread_mutex_unlock(&p->mutex);
    }
    else if(q->right==p){
        pthread_mutex_lock(&p->mutex);
        q->right = p->left;
        pthread_mutex_unlock(&p->mutex);
    }
}

//하나의 자식을 갖는 노드일 경우 (오른쪽 자식)
if((p->left==0)&&(p->right!=0)){
    k=p;
    if(q->left==p){
        pthread_mutex_lock(&p->mutex);
        q->left = p->right;
        pthread_mutex_unlock(&p->mutex);
    }
    else if(q->right==p){
        pthread_mutex_lock(&p->mutex);
        q->right = p->right;
        pthread_mutex_unlock(&p->mutex);
    }
}
    
```

//두개의 자식을 갖는 노드일 경우

```
else if((p->left!=0)&&(p->right!=0)){
    k=p->left;
    while(k->right != 0){
        q=k;
        k=k->right;
    }
    if(q==p){
        pthread_mutex_lock(&p->mutex);
        p->key = k->key;
        if(k->left!=0)
            p->left = k->left;
        else
            p->left = 0;
        pthread_mutex_unlock(&p->mutex);
    }
    else{
        pthread_mutex_lock(&p->mutex);
        p->key = k->key;
        if(k->left!=0)
            q->right = k->left;
        else
            q->right = 0;
        pthread_mutex_unlock(&p->mutex);
    }
}
return 0;
```

Critical section

Critical section



int lab2_bst_remove_cg(lab2_tree *tree, int key)

lab2_node_remove 함수에서, 찾으려는 key값이 tree에 없을 때, 중간에 return 을 해 주는데, 그렇게 되면, return전에 unlock을 해 줘야 하므로 lock과 unlock의 짝이 맞지 않는다. 이를 위해, while문에서 key값이 tree에 없을 때, 우선 while문에서 break하고, 노드를 삭제하는 구현 부 전에 if(p->key==key)라는 조건을 줘서 key값을 찾지 못했을 경우, if문에 들어가지 못하고 바로 unlock과 return을 시킨다. 만약 key값에 해당하는 노드를 찾게 되면 if문 안으로 들어가서 해당 노드를 삭제하게 된다. 이 부분은 다른 remove함수와 다른 점이다. 또한 fg와는 다르게 인자가 &tree->root->mutex가 된다.

```
int lab2_node_remove_cg(lab2_tree *tree, int key) {
    // You need to implement lab2_node_remove_cg function.
    if(!tree->root)
        return -1;
    pthread_mutex_lock(&tree->root->mutex);
    lab2_node*p=tree->root;
    lab2_node*q=0;
    lab2_node*k=0;

    //트리의 루트부터 키에 해당하는 노드까지 포인터 이동
    while((p!=NULL) && (p->key != key)){
        q=p;
        if((p->left == NULL)&&(p->right ==NULL)){
            break;
        }
        if(key<p->key){
            if(p->left){
                p=p->left;
            }
            else{
                break;
            }
        }
        else if(key>p->key){
            if(p->right){
                p=p->right;
            }
            else{
                break;
            }
        }
    }
    if(p->key==key){
        ...
    }
    pthread_mutex_unlock(&tree->root->mutex);
    return 0;
}
```

Critical section

■ Delete

int lab2_tree_delete(lab2_tree *tree)

tree를 받아 tree->root가 NULL이 아닐 경우에 있어서 lab2_node_delete()의 함수를 통해 tree->root를 delete 해준다.

또한, root를 NULL로 대입하고, 마지막으로 free()를 메모리를 해제할 수 있도록 하여, malloc으로 할당된 메모리를 해제할 수 있도록 해줘야 한다.

int lab2_node_delete(lab2_node *node)

노드를 받아 left가 NULL이 아닐 경우, lab2_node_delete()함수를 통해 left 값을 delete를 해줬다. right의 경우도 이와 같이 동일하다.

또한, key, left, right를 0과 NULL, NULL로 대입하고, 마지막으로 free()를 메모리를 해제할 수 있도록 하여, malloc으로 할당된 메모리를 해제할 수 있도록 해줘야 한다.

3. 출력 결과

① lock 이 있을 경우와 없을 경우의 수행 결과 차이

■ lock 이 없을 경우

```
dku-os-2020@dkuos2020-VirtualBox:~/2020_DKU_OS/lab2_sync$ ./lab2_bst -t 4 -c 100
000000000000
Segmentation fault (core dumped)
```

lock 을 구현해 주지 않고, 값을 10 만개 정도 넣을 경우 Segmentation fault 가 발생한 것을 확인 할 수 있다. 이는 상호 배제(Mutex exclusion)가 보장 되지 않아 동기화(Synchronization)가 되지 않았기 때문이다.

■ lock 이 있을 경우 _mutex 사용

```
==== Multi thread single thread BST insert experiment ====

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 1.791173 seconds

BST inorder iteration result :
  total node count : 1000000

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

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 2.419921 seconds

BST inorder iteration result :
  total node count : 1000000

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

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 0.450845 seconds

BST inorder iteration result :
  total node count : 1000000
```

```
==== Multi thread single thread BST delete experiment ====

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 0.698465 seconds

BST inorder iteration result :
  total node count : 1000000

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

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 1.100632 seconds

BST inorder iteration result :
  total node count : 1000000

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

Experiment info
  test node      : 1000000
  test threads   : 4
  execution time : 0.151048 seconds

BST inorder iteration result :
  total node count : 1000000
```

Coarse grained 와 Fine grained 가 각각 임계 영역(critical section)의 범위를 다르게 잡아주어 lock, unlock 을 구현하게 될 경우, 위와 같이 출력이 잘 되는 것을 알 수 있다.

lock, unlock 을 통해 상호 배제(Mutual exclusion)가 보장이 되도록 하여, 경쟁 상태(race condition)가 일어나지 않도록 한 것인데 이에 따라 공유 자원(shared resource)들이 동기화(Synchronization)가 잘 된 방식으로 접근하고 있는 것을 유추할 수 있다.

② lock 구현 시, Fine-grained 와 Coarse-grained 의 차이에 따른 성능 비교

다음은 Fine grained 와 Coarse grained 의 차이에 따른 성능 비교를 하기 위해, test node 의 값을 달리하여 실행한 결과 값들을 표로 나타냈다.

Insert 부분에 대한 execution time 비교 (test threads 4)

test node	Single thread	Coarse grained	Fine grained
1000	0.000264	0.000975	0.000380
10000	0.003469	0.009484	0.003175
100000	0.084241	0.170471	0.021049
1000000	1.791173	2.419921	0.450845

Delete 부분에 대한 execution time 비교 (test thread 4)

test node	Single thread	Coarse grained	Fine grained
1000	0.000108	0.000349	0.000207
10000	0.001142	0.006533	0.000549
100000	0.023841	0.078122	0.010290
1000000	0.698465	1.100632	0.151048

lock 은 정확성(Correctness) 및 병행성(Concurrency)을 보장하기 위하여 구현되는 것으로, 잘못 구현 시 성능(Performance)이 낮아지게 된다.

Single thread 의 경우, 병행성(Concurrency)이 좋지 않아 Coarse grained 보단 빠르지만 Fine grained 보단 느린 것을 알 수 있다.

Fine grained 은 Coarse grained 보다 임계 영역(Critical Section)의 범위를 세부적으로 잡았다. 이는 연산이 수행되는 부분에 있어서만 임계 영역(Critical Section)의 범위를 잡아준 것이다.

Fine grained 의 경우, lock 의 개수가 많아 병행성(Concurrency)를 좋게 한다. 그렇기 때문에 위의 Insert 와 Delete 부분의 execution time 을 봤을 때, Fine grained 의 경우가 압도적으로 시간이 빠르 기에, 성능(Performance)이 우수한 것을 알 수 있다.

그에 비해, Coarse grained 의 경우, 임계 영역(Critical Section)의 범위를 크게 잡아 Fine grained 보다 execution time 이 크며, 성능(Performance)이 떨어지는 것을 알 수 있다.

결국, Fine grained 가 Coarse grained 보다 병행성(Concurrency)가 좋아 성능(Performance)이 우수한 것을 확인할 수 있다.

또한, Fine grained 는 여러 thread 가 multi core 에서 병렬적으로 수행이 가능하다. 병렬성(parallelism)은 프로그램 수행에 있어서 중요한 요인으로, 병렬적으로 처리 시, 성능(performance)이 좋아지게 된다. 그렇기 때문에 Fine grained 처럼 임계 영역(Critical section)을 보호하는 부분을 작게 잡아 성능(Performance)을 좋게 할 수 있다. 하지만 이는 프로그램이 복잡하다는 단점이 있다.

Coarse grained 는 관리 lock 을 적게 주고, 임계 영역(Critical section)을 크게 잡아 프로그램이 관리가 쉽지만 병렬성(parallelism)이 낮아진다는 단점이 존재한다.

③ 발생했던 오류와 해결 방안 제시

■ 결과 출력 시, 멈춤 현상에 관하여 (+Deadlock)

```
==== Multi thread single thread BST insert experiment ====
Experiment info
  test node           : 100000
  test threads        : 4
  execution time      : 0.071006 seconds

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

결과 출력에 있어서 Segmentation fault 의 문제는 뜨지 않으나, 멈춤 현상이 발생했다. 하지만 몇 번 다시 돌려보게 될 경우, 다음과 같이 출력이 된다.

```
==== Multi thread single thread BST insert experiment ====
Experiment info
  test node           : 100000
  test threads        : 4
  execution time      : 0.054817 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      : 0.132674 seconds
```

몇 번의 실행 반복을 통해서 결과 값이 나오기는 하지만 멈춤 현상이 잦았고, 이에 따라, 어떤 함수의 호출을 기다리고 있거나 Deadlock 이 발생할 수 있다는 것을 예측할 수 있었다.

그렇기에 Deadlock 에 대해서 찾아보게 되었다. 이는 현재 수업 중인 'Lecture Note 5. Concurrency Semaphore and Deadlock'과 관련된 개념이다. 현재 수업 전이기에 Deadlock 과 관련해서 개념을 찾아보았다.

Deadlock 은 2 개 이상의 Process 혹은 Thread 가 서로 끝나기를 기다리는 상태이다. 이는 공유 자원을 사용하기 때문에 발생한다. Deadlock 이 발생하는 조건으로는 첫 번째, Mutual exclusion 은 한 번에 여러 Process / Thread 가 한 자원에 접근하지 못하도록 막은 것이다. 두 번째, Hold and wait 은 자원을 가지고 있는 상태에서 다른 Process / Thread 가 쓰는 자원을 반납하기를 기다리는

상태이다. 세 번째, No Preemption 으로 이는 다른 Process / Thread 가 이미 점유한 자원을 뺏어 오지 못하게 하는 것이다. 네 번째, Circular Wait 로 Process / Thread 가 일이 끝나기를 계속 기다리다가 마지막에는 자신이 나오는 상황이 되는 것이다.

그렇기 때문에 Deadlock 은 위의 4 가지 조건이 성립될 경우 발생하며, 이 중 하나라도 발생하지 않게 하면 예방이 된다. 회피하는 방법으로는 Deadlock 을 막는 알고리즘을 적용하여 해결을 하면 된다. 하지만 Deadlock 을 해결을 하기 위해서는 성능 상의 손해를 볼 수 있다.

Deadlock 에 대해서 아직 개념에 대한 이해도가 부족하기 때문에 다른 부분에 문제가 있는지 확인을 하기 시작했다. lock / unlock 을 주석 처리시, 출력 시 잘 출력이 됨을 확인했고, 이에 따라, mutex lock 의 문제가 있다고 생각을 했다.

Lab2_bst_test.c 에 따라 single thread 다음에 coarse grained, fine grained 가 차례대로 출력이 된다. 그렇기 때문에 coarse grained 부분에서 멈추었기 때문에 나머지도 출력이 안됐다. 그리하여 coarse grained 부분의 mutex 부분을 살펴 보았다.

이는 insert_cg 의 tree_root 부분이 NULL 일 경우, mutex 가 초기화 되지 않은 new_node 로 받는 것에 있어서 문제가 됐다. 그렇기 때문에 tree_root 부분에 mutex_init 이 되어 있는 새로운 노드를 생성해 줬다.

그렇게 될 경우 mutex_init 이 되어 있는 노드가 들어오기 때문에 lock 이 가능하고, 멈춤 현상 없이 나머지도 잘 출력 되는 것을 확인할 수 있었다.

■ 큰 값을 넣어줄 경우 멈추는 상황과 killed 되어버리는 상황 발생

위와 같은 현상이 해결 됐으나, node_count 에 큰 값을 넣어주게 될 경우 다시 한번 멈추는 현상이 발생했다. 이에 따라 mutex 가 받는 인자를 확인하여야 했고, 여러 return 값에 따라 자원들이 계속 기다리게 된다는 것을 확인했다. 그렇기 때문에 위와 같이 멈추는 현상이 발생했던 것이다.

다음과 같이 coarse grained 의 경우, unlock 부분을 하나 더 넣어주어 자원들이 기다리지 않도록 해주었다.

```

while(p != NULL) {
    q = p;
    if(new_node->key == p->key) {
        pthread_mutex_unlock(&tree->root->mutex);
        return -1;
    }

    if(new_node->key < p->key) {
        p = p->left;
    }
    else {
        p = p->right;
    }
}

```

위와 같이 구현을 할 경우, 멈추는 현상이 발생하지 않고, 출력이 잘 되는 것을 볼 수 있었다.

하지만 기존의 수업 시간에 coarse grained 와 fine grained 를 배울 때, lock 과 unlock 을 fair 하게 쌍으로 구현하여야 less bug 가 된다고 했다. 위와 같이 수정을 하게 될 경우, coarse grained 의 lock 이 쌍으로 구현 된 것이 아니라는 것에서 의문점이 들었다. 이에 있어서는 생각을 더 해 볼 필요성을 느끼고, 코드를 다음과 같이 수정했다.

첫 번째로, 추가하려는 노드가 tree에 이미 있을 경우, while문에서 return 값을 -1로 주는 것이 아니라 break로 나오게 하였다. 그리고 lock과 unlock을 쌍으로 구현할 수 있도록 해줬다.

마지막 실행 과정에 있어서 다시 한 번 엄청나게 큰 노드의 개수를 받게 될 경우(e.g 10 만 개의 경우), killed 가 되어버리는 오류가 발생했다.

처음에는 이에 Deadlock 이 발생한 문제가 있는 것인지 의문이 들었다. Deadlock 의 4 가지 발생 조건에 따라, Mutual exclusion 으로 한 번에 여러 Thread 가 자원에 접근하지 못하도록 막았기 때문에 엄청 큰 노드의 개수를 받았을 경우 계속해서 자원을 반납하기를 기다리다가 멈추게 되어 결국엔 killed 가 된 것이 아닐지 생각을 했다.

하지만 이와 관련해서 gdb 를 돌려봤고, thread 는 종료가 됐지만 signal SIGKILL 을 보낸 것을 확인했다.

```
[New Thread 0x7ffff519e700 (LWP 14699)]
[New Thread 0x7ffff499d700 (LWP 14700)]
[New Thread 0x7ffffeffff700 (LWP 14701)]
[New Thread 0x7ffffef7fe700 (LWP 14702)]
[Thread 0x7ffffef7fe700 (LWP 14702) exited]
[Thread 0x7ffffeffff700 (LWP 14701) exited]
[Thread 0x7ffff499d700 (LWP 14700) exited]
[Thread 0x7ffff519e700 (LWP 14699) exited]

Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb)
```

dmesg | grep -E -i -B100 'killed process'를 통해 오류의 내용을 확인했다.

```
[49986.636616] Out of memory: Killed process 9785 (lab2_bst) total-vm:3213896kB, anon-rss:3118340kB, file-rss:0kB, shmem-rss:0kB
```

Out Of Memory killer(OOM killer)은 메모리가 부족할 경우, 특정 프로세스를 강제로 종료 시키는 것이다. 실제 Physical 메모리보다 큰 프로그램을 구동할 경우, (OverCommit 된 메모리에 쓰여진 경우) 메모리가 모자라 Out Of Memory 가 발생한 것이다. 하지만 OOM killer 는 메모리에 있어서 badness 의 task 를 kill 을 해주며 goodness 한 상태로 만드는 것이 목표이기 때문에 나쁜 것은 아니다.

이에 따라, 두 가지를 생각하였다. 포인터를 제대로 초기화 시키지 않아 문제가 발생할 수 있다는 것과 코드 내에서 사용된 포인터가 잘못된 메모리 영역의 주소를 반환하여 Physical 메모리의 용량보다 초과가 되는 상황이 발생한 것 같았다.

이는 포인터를 사용 시 참조하는 주소의 파악에 있어서 주의를 기울일 필요성이 있고, 포인터의 초기화의 중요성을 파악했다.

4. Discussion

① 회의 방법

직접 만나지 못하는 상황이기 때문에 회의를 원활하게 진행할 수 있는 방안을 생각하여야 했습니다. 주기적으로 서로의 시간을 맞춰 구글의 '행아웃'을 통해 서로의 화면을 공유하며 통화를 하는 형식으로 회의를 진행했습니다. 또한 행아웃을 하지 못하는 상황일 경우, 카카오톡 채팅을 통하여 문제점에 대해서 찾아보고 공유를 하는 형식으로 진행했습니다.

② Git

GitHub Desktop을 노트북 내에 설치하여 Visual Studio Code와 연결하였습니다. 우분투에서 작성한 코드를 Visual Studio Code에서 불러와 더 편하게 보고 수정을 했습니다. Visual Studio Code에서 수정한 부분을 우분투에서 재 작성하여 결과 값을 실행했습니다. 오류가 날 경우, 확인을 하고 우분투에서 수정한 부분을 재 작성하고, git에 commit을 하여 repository에 push를 하는 과정을 반복했습니다.

③ 각 구현에 있어서 어려웠던 점과 프로젝트를 하며 느낀 점

나현진

저는 자료구조 시간에 만들었던 BST를 기반으로 inorder, node와 tree에 대한 create, insert 부분을 구현했습니다. 자료 구조 시간에 만들어 본적이 있기 때문에 구현이 쉬울 것이라고 생각을 했었습니다. 하지만 C++로 구현을 했었기 때문에 C언어로 바꾸는데 있어서 시간이 걸렸습니다. 그렇기 때문에 BST에 대한 개념을 다시 공부하며 노드의 생성, 삽입, 삭제 등의 구현을 이해해 나갔습니다.

두 번째로, lock 구현에 있어서 처음에는 Coarse grained와 Fine grained의 lock의 범위 차이만을 생각하고 lock 구현을 했던 것 같습니다. 하지만 단순히 lock의 위치를 다르게 해주는 것만으로는 맞지 않다는 것을 다시 한 번 오류를 통해서 깨달았습니다. mutex에 대해서 많은 정보를 찾아보고, Critical Section의 부분을 파악하는 것이 필요했습니다.

lab2 실습을 구현하면서 많은 실패를 겪으며 교수님께도 문의 메일을 여러 번 드리기도 하고, 많은 정보를 검색 후에 해결해 나갔습니다. 또한 무엇보다도, 결과 값이 나오지 않는 경우에 있어서 중간에 출력 문을 생성하여 값이

어디까지 들어갔나 확인하는 방법을 여러 번 반복했습니다. 하지만 결과가 출력이 되어도 문제였습니다. 잘못된 결과가 나오게 될 경우, 문제점을 찾기가 힘들기 때문입니다. 이는 구현에 있어서 전체적으로 처음부터 코드를 봐야하는 상황이 되어 버렸습니다. Segmentation fault가 날 경우, gdb를 통해 확인을 하는 과정을 반복했습니다. 한 부분의 오류가 다른 부분에 오류를 미칠 수 있다는 것도 깨달았고, 전체적인 부분에 있어서 다시 한 번 생각을 하며 오류를 고칠 수 있도록 해야 한다는 것을 느꼈습니다.

코드 수행 결과에 있어서는 처음에 coarse grained가 lock을 잡아주는 갯수가 fine grained보다 적기 때문에 Performance인 execute time이 더 짧을 것이라고 생각을 했었습니다. 하지만 이는 개념에 대한 이해가 잘못 됐었고, 수행 결과에 대한 비교 분석과 개념을 다시 파악하며 바로 잡을 수 있었습니다. 즉, 임계 영역(Critical section)을 보호하는 부분을 작게 잡아 성능(Performance)을 좋게 하는 것이 좋다는 것을 깨달았습니다.

코드를 완벽하게 작성하지 않은 것 같아 아쉬움이 남습니다. 프로젝트가 끝나고, 시간을 가져 더 완벽하게 구현을 하고 git에 올릴 예정입니다. 코드 구현이 중요하지만 오류에 있어서 정보를 찾아보는 능력도 중요하다고 생각했습니다. 마지막으로 OOM killer의 문제에 있어서는 시간을 가지고 더 찾아볼 예정입니다. 실제로 오류에 대한 부분을 분석하는 것과 이에 대한 정보를 찾아보는 시간을 가지며 많은 것을 알아가는 과정을 가졌던 것 같습니다.

지난 lab1보다 보고서 작성에 있어서 많은 시간을 들였으며 작성에 있어서 또한 기존 수업에 내용을 복습하고, 많은 정보들을 찾아보았습니다. 그렇기에 이와 같이 보고서를 작성할 수 있었고 많은 지식들이 정리가 됐던 것 같습니다.

이번 과제를 통해서 수업 시간에 배운 개념 이외에도 직접 실습을 통해서 확인을 해가며 많은 것을 얻었습니다. 무엇보다도 팀원 분과 소통을 통해서 제가 해결할 수 없던 것들에 대한 부족한 부분들을 도와주셔서 많은 도움이 됐던 것 같습니다. 다시 한 번 팀 프로젝트의 소통의 중요성과 적극적인 참여 필요성을 느꼈습니다.

이지연

저는 자료구조 수업에서 작성했던 BST를 갖고, 트리 노드 삭제함수인 remove 함수 구현, 노드 삭제인 delete 부분 구현을 했습니다. 이 때, c++로 구현 했어서, lock과 c언어의 특성을 고려하여 재 구성하는 것이 조금 어려운 부분이 있었습니다. 하지만 나현진 학생과 함께 bst부분을 다시 공부하고 최

대한 배운 범위에서 구현하려고 노력했습니다.

bst_test.c 함수와 비교하여 작동원리를 이해하고 bst.c 내부에 오류 원인을 bst_test.c와 분석하여 수정하였습니다. 오류를 수정하고 컴파일은 되었지만, 큰 수를 넣으면 동작이 되지 않았습니다. 그 이유를 lock & unlock의 짝이 맞지 않아서 인 것으로 판단하여, lock과 unlock의 짝을 맞춰주기 위해 코드를 재 작성했습니다. 특히 coarse grained 부분에서 짝이 맞지 않았는데, 그 이유는 중간에 return 하면서 unlock을 수시로 해 줬기 때문이라고 생각되어 우선 while 문을 벗어나고 unlock을 한 뒤 모든 return을 한번만 하도록 수정했습니다. 그 결과, coarse grained와 fine grained의 특성을 잘 맞추고, 결과도 더욱 확실하게 대비되어 나타남을 알 수 있었습니다. 이를 통해 코드 최적화에 대한 중요성을 알게 되었습니다. 하지만 아직도 큰 수를 넣으면 동작이 안되는 것이 매우 안타까웠고, 아무리 고쳐봐도 동작원리에는 이상이 없는 것 같아서, physical memory의 부족인 것으로 예상이 됩니다. 추후에 이 부분은 수정을 하는 것이 이번 학기에서 목표로 삼고 있습니다.

또한 lock 구현을 하면서 함수 내부에서 critical section에 대한 관리가 매우 섬세 하다는 것을 알게 되었습니다. 함수 내부에서 critical section을 잘못 설정하거나, lock하고 unlock이 되지 않게 되면 segmentation fault가 발생하는 경우가 매우 많았습니다. 그래서 과제 설명 pdf파일을 다시 보면서 bst에서 lock을 생성해야 하는 critical section에 대한 범위를 제대로 다시 파악하고 생산자&소비자 관점에서 lock& unlock을 확실하게 구현하는 것이 관건이었습니다. 이를 통해 lock과 unlock을 구현하면서 동작 단계를 상상하면서 전체적인 thread의 동작에 대한 경우의 수를 고려해야 한다는 것을 알게 되었습니다.

코드 수행결과에서 확실히 count를 늘려가면서 비교해 보면 single thread의 경우 확실히 multi-thread보다 느린 것을 파악해서, 병행성에 대한 중요성을 느끼게 되었고, coarse-grained보다 fine-grained가 수행속도가 훨씬 빠르게 나타나서 lock의 범위에 대한 중요성을 알게 되었습니다.

그리고 팀원인 나현진 학생과 협업하면서 제가 부족했던 부분에 대해서 많이 도움이 되었고, 오류를 해결하는 과정에서 정보를 많이 찾아 주셔서 도움이 많이 되었던 것 같습니다. 그리고 찾아주신 정보들이 대부분 새로운 정보들이어서 추후에 코딩할 때 도움이 많이 될 것 같습니다.

이 외에도 교수님 강의와 과제 해설을 다시 보면서 lock구현에 있어서 핵심을 다시 파악하고, 함수를 구현한 것이 가장 도움이 많이 된 것 같습니다.