R E P O R T

<LAB 2 >

Concurrency Binary Search Tree in C

과목명: Operating System

담당교수: 최종무 교수님

소속: 소프트웨어학과 3학년

학번: 32162417

이름: 신정웅

제출일: 2020년 4월 6일

과제를 준비 함에서…

이번 과제를 진행하면서 Git/ GitHub 사용법을 숙지하여 비록 혼자하는 과제지만 https://github.com/JeongShin/OS_lab2_sync 에서 진행 하였습니다. Git을 사용하면서 편리하였던 점을 Discussion에서 다루겠습니다.

1. Code

```
int lab2_node_insert(lab2_tree *tree, lab2_node *new_node)
{
    if (tree == NULL)
    {
        perror("Error: Empty tree insertion!");
        exit(-1);
    }
    lab2_node *curr = tree->root;
    lab2_node *parent = NULL;
    int key = new_node->key;
    if (tree->root == NULL)
    {
        tree->root = new_node;
        return 1;
    }
    while (curr != NULL)
    {
        parent = curr;
        if (key < curr->key)
            curr = (lab2_node *)curr->left;
        else
            curr = (lab2_node *)curr->right;
    }
    if (key < parent->key)
    {
        parent->left = (struct lab2_tree
*)new_node;
    }
    else
    {
        parent->right = (struct lab2_tree
*)new_node;
    }
    return 1;
```

1. 노드 삽입 & NO LOCK

한개의 노드를 BST에 삽입하는 함수 입니다. 어떠한 Lock도 없기 때문에 동시에 여러개의 쓰레드가 접근합니다.

여기서 예로

`curr = (lab2_node *)curr->left;`

노드가 다른 노드로 옮겨갈때

`tree->root = new_node;`

루트에 새로운 노드를 삽입할때

등의 실행이 모드 임계영역 (Critical Section)에 해당 됩니다.

이 과제의 환경은 멀티 쓰레딩 환경이기 때문에 Lock이 없는 접근은 프로그램이 원하는 수행결과를 보장할 수 없습니다.

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

                    Experiment info
               test node          : 1000000
                  test threads        : 4
             execution time      : 1.152346 seconds


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

```
int lab2_node_insert_fg(lab2_tree *tree, lab2_node *new_node)
{
    lab2_node *parent = NULL;
    lab2_node *curr = tree->root;
    int key = new_node->key;
    if (curr == NULL)
    {
        pthread_mutex_lock(&tree->mutex);
        curr = new_node;
        pthread_mutex_unlock(&tree->mutex);
        return 1;
    }

    while (curr != NULL)
    {
        pthread_mutex_lock(&curr->mutex);
        parent = curr;
        if (key < curr->key)
            curr = curr->left;
        else
            curr = curr->right;
        pthread_mutex_unlock(&curr->mutex);
    }

    pthread_mutex_lock(&tree->mutex);
    if (key < parent->key)
    {
        parent->left = new_node;
    }
    else
    {
        parent->right = new_node;
    }
    pthread_mutex_unlock(&tree->mutex);
    return 1;
}
```

2. 노드 삽입 & FG LOCK

Fine Grained Lock의 경우 임계영역에 접근 하는 경우에만 Lock을 걸어 주었습니다.

가장 먼저 루트노드가 없을 경우 (빈 트리) 트리에 Lock을 걸어 새 노드를 할당해주고 풀어주었습니다.

parent->left = new_node;

의 경우 new_node의 삽입이 이루어지기 때문에 tree 전체에 Lock을 걸어 두었습니다.

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

                    Experiment info
              test node           : 1000000
                 test threads        : 4
            execution time      : 0.136470 seconds


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

```c
int lab2_node_insert_cg(lab2_tree *tree, lab2_node *new_node)
{
    pthread_mutex_lock(&tree->mutex);
    lab2_node *parent = NULL;
    lab2_node *curr = tree->root;
    int key = new_node->key;
    if (curr == NULL)
    {
        curr = new_node;

        pthread_mutex_unlock(&tree->mutex);
        return 1;
    }

    while (curr != NULL)
    {
        parent = curr;
        if (key < curr->key)
            curr = curr->left;
        else
            curr = curr->right;
    }

    if (key < parent->key)
    {
        parent->left = new_node;
    }
    else
    {
        parent->right = new_node;
    }
    pthread_mutex_unlock(&tree->mutex);
    return 1;
```

3. 노드 삽입 & CG LOCK

Coarse Grained Lock의 경우 함수가 시작되는 순간부터 끝까지 크게 Lock을 걸어 주었습니다.

가장 먼저 lock을 트리에 걸어주고 함수가 끝나기 직전에 unlock 합니다.

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

                    Experiment info
              test node          : 1000000
                 test threads        : 4
            execution time      : 0.097761 seconds


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

```c
int lab2_node_remove(lab2_tree *tree, int key)
{
    if (tree == NULL)
    {
        perror("Error: Empty tree node deletion!");
        exit(-1);
    }
    lab2_node *curr = tree->root;
    lab2_node *parent = NULL;
    lab2_node *succ;
    lab2_node *par_succ;
```

```
    /*Searching for node to delete*/
    while (curr != NULL && curr->key != key)
    {
        parent = curr;
        if (key < curr->key)
            curr = (lab2_node *)curr->left;
        else
            curr = (lab2_node *)curr->right;
    }
    if (curr == NULL)
        return 0;
    /*Node with no child (leaf node)*/
    if (curr->left == NULL && curr->right == NULL)
    {
        if (parent->left != NULL)
        {
            if (parent->left == curr)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            tree->root = NULL;
        curr = NULL;
    }
    /*Node with left child only*/
    else if (curr->left != NULL && curr->right ==
NULL)
    {
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->left;
            else
                parent->right = curr->left;
        }
        else
            (tree->root = curr->left);
        curr = NULL;
    }
    /*Node with right child only*/
    else if (curr->left == NULL && curr->right != NULL)
    {
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->right;
            else
                parent->right = curr->right;
        }
        else
            (tree->root = curr->right);
        curr = NULL;
    }
    /*Node with right & left child*/
    else
    {
        par_succ = curr;
        succ = (lab2_node *)curr->right;
        while (succ->left != NULL)
```

4. 노드 삭제 & NO LOCK

key에 해당하는 노드를 찾아서 삭제합니다. Lock이 없기 때문에 멀티 쓰레딩 환경에서 여러개의 쓰레드가 같은 값(노드)를 삭제하려는 상황이 생길 수 있습니다. 혹은 삭제되고 있는 노드에 접근해서 잘못된 Sub-tree로 이동될 가능성도 있습니다.

코드가 중복되는 부분이 많지만 에러를 방지하고 이해를 높이기 위해 최대한 하나하나 나열하여 설계하였습니다.

```
        {
            par_succ = succ;
            succ = (lab2_node *)succ->left;
        }
        if (par_succ->left == succ)
        {
            par_succ->left = par_succ->right;
        }
        else
            par_succ->right = succ->right;
        curr->key = succ->key;
        succ = NULL;
    }
    return 1;
}
```

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

                                    Experiment info
                                test node           : 1000000
                                   test threads        : 4
                              execution time      : 0.066769 seconds


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

```
int lab2_node_remove_fg(lab2_tree *tree, int key)
{
    if (tree == NULL)
    {
        perror("Error: Empty tree node deletion!");
        exit(-1);
    }
    lab2_node *curr = tree->root;
    lab2_node *parent = NULL;
    lab2_node *succ;
    lab2_node *par_succ;
    /*Searching for node to delete*/
    while (curr != NULL && curr->key != key)
    {
        parent = curr;
        pthread_mutex_lock(&curr->mutex);
        if (key < curr->key)
            curr = (lab2_node *)curr->left;
        else
            curr = (lab2_node *)curr->right;
        pthread_mutex_unlock(&curr->mutex);
    }
    if (curr == NULL)
        return 0;
    /*Node with no child (leaf node)*/
    if (curr->left == NULL && curr->right == NULL)
    {
        pthread_mutex_lock(&tree->mutex);
```

4. 노드 삭제 & FG LOCK

삽입과 마찬가지로 Fine Grained Lock의 경우 임계영역에 접근 하는 경우에만 Lock을 걸어 주었습니다.

Key에 해당하는 노드를 찾아 내려가는 반복문은 노드에서 다른 노드로 옮겨가기 때문에 curr 노드 기준으로 Lock을 걸어 주었 습니다. 다른 스레드는 curr 노드에 해당하는 곳에 접근할 수 없습니다.

```
        if (parent->left != NULL)
        {
            if (parent->left == curr)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            tree->root = NULL;
        pthread_mutex_unlock(&tree->mutex);
        curr = NULL;
    }
    /*Node with left child only*/
    else if (curr->left != NULL && curr->right ==
NULL)
    {
        pthread_mutex_lock(&tree->mutex);
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->left;
            else
                parent->right = curr->left;
        }
        else
            (tree->root = curr->left);
        pthread_mutex_unlock(&tree->mutex);
        curr = NULL;
    }
    /*Node with right child only*/
    else if (curr->left == NULL && curr->right !=
NULL)
    {
        pthread_mutex_lock(&tree->mutex);
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->right;
            else
                parent->right = curr->right;
        }
        else
            (tree->root = curr->right);
        pthread_mutex_unlock(&tree->mutex);
        curr = NULL;
    }
    /*Node with right & left child*/
    else
    {
        par_succ = curr;
        succ = (lab2_node *)curr->right;
        pthread_mutex_lock(&tree->mutex);
        while (succ->left != NULL)
        {
            par_succ = succ;
            succ = (lab2_node *)succ->left;
        }
        if (par_succ->left == succ)
        {
            par_succ->left = par_succ->right;
```

삭제하려는 노드는 다음중 하나의 노드입니다.

1. 자식이 없는 경우

2. 왼쪽 자식만 있는 경우

3. 오른쪽 자식만 있는 경우

4. 자식이 없는 경우

모든 경우

```
if (parent != NULL)
else
(tree->root = curr->right);
```

즉 parent와 root가 중복되는 경우

```
            3 -> parent, root
      2           5 -> curr
1             4     6
```

```
before remove123456
after remove
124
```

5를 삭제하면 위와 같이 5,6 sub - tree 모두 연결이 끊어지는 상황을 발견하여 tree -> root 를 다시 설정해주는 코드를 짜주었는데 tree -> root 에 대한 접근이 이루어지기 때문이 tree 에 대한 Lock을 걸어 주었습니다.

```
        }
        else
            par_succ->right = succ->right;
        pthread_mutex_unlock(&tree->mutex);
        curr->key = succ->key;
        succ = NULL;
    }
    return 1;
}
```

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

                                    Experiment info
                                test node           : 1000000
                                   test threads         : 4
                               execution time      : 0.003597 seconds


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

int lab2_node_remove_cg(lab2_tree *tree, int key)

```
{
    if (tree == NULL)
    {
        perror("Error: Empty tree node deletion!");
        exit(-1);
    }
    pthread_mutex_lock(&tree->mutex);
    lab2_node *curr = tree->root;
    lab2_node *parent = NULL;
    lab2_node *succ;
    lab2_node *par_succ;
    /*Searching for node to delete*/
    while (curr != NULL && curr->key != key)
    {
        parent = curr;
        if (key < curr->key)
            curr = (lab2_node *)curr->left;
        else
            curr = (lab2_node *)curr->right;
    }
    if (curr == NULL)
    {
        pthread_mutex_unlock(&tree->mutex);
        return 0;
    }
    /*Node with no child (leaf node)*/
    if (curr->left == NULL && curr->right == NULL)
    {
        if (parent->left != NULL)
        {
            if (parent->left == curr)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            tree->root = NULL;
        curr = NULL;
    }
```

5. 노드 삭제 & CG LOCK

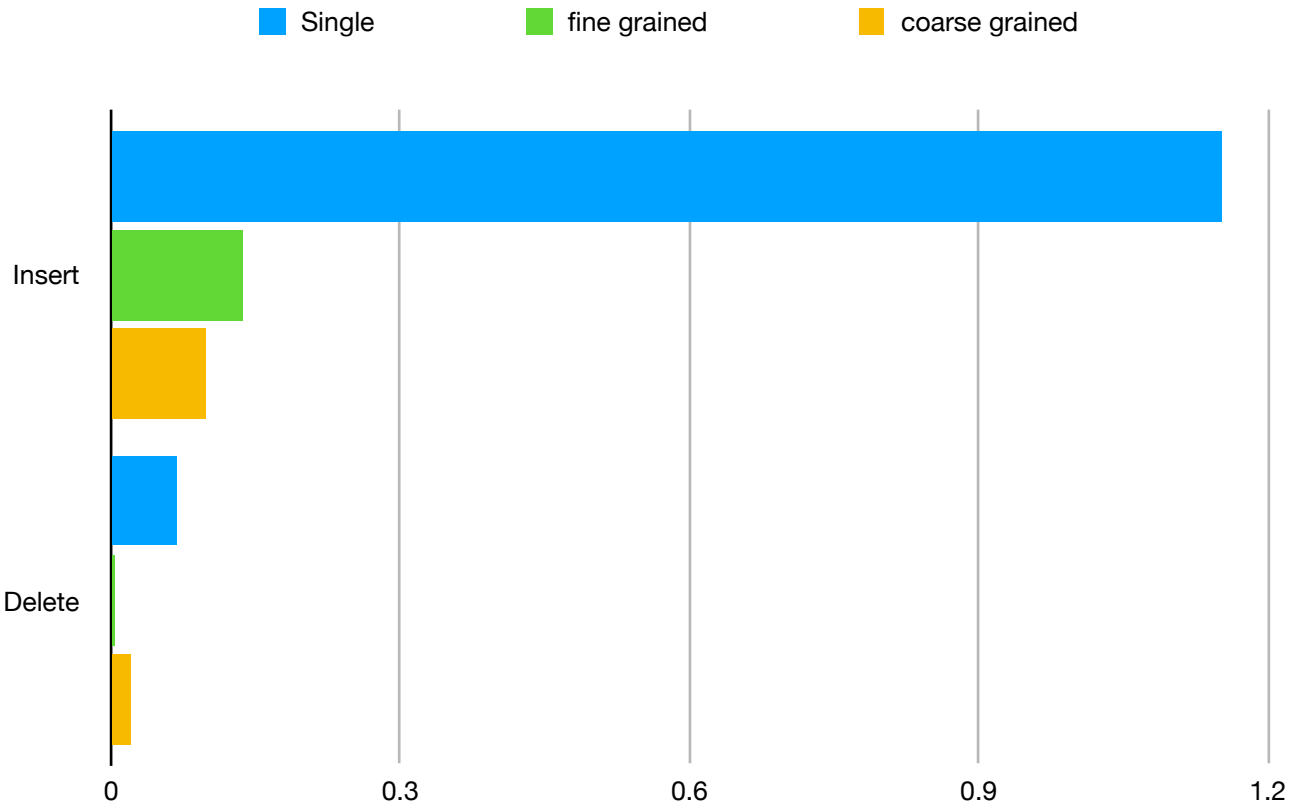
Coarse Grained Lock의 경우 마찬가지로 함수의 시작부터 끝까지 tree에 Lock을 걸어 주었습니다.

```
    /*Node with left child only*/
    else if (curr->left != NULL && curr->right == NULL)
    {
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->left;
            else
                parent->right = curr->left;
        }
        else
            (tree->root = curr->left);
        curr = NULL;
    }
    /*Node with right child only*/
    else if (curr->left == NULL && curr->right != NULL)
    {
        if (parent != NULL)
        {
            if (parent->left == curr)
                parent->left = curr->right;
            else
                parent->right = curr->right;
        }
        else
            (tree->root = curr->right);
        curr = NULL;
    }
    /*Node with right & left child*/
    else
    {
        par_succ = curr;
        succ = (lab2_node *)curr->right;
        while (succ->left != NULL)
        {
            par_succ = succ;
            succ = (lab2_node *)succ->left;
        }
        if (par_succ->left == succ)
        {
            par_succ->left = par_succ->right;
        }
        else
            par_succ->right = succ->right;
        curr->key = succ->key;
        succ = NULL;
    }
    pthread_mutex_unlock(&tree->mutex);
    return 1;
}
```

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

                                  Experiment info
                             test node           : 1000000
                                test threads        : 4
                           execution time      : 0.021954 seconds


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

2. Analysis

2 - 1. 수행 시간 분석



위의 데이터는 다음과 같은 옵션을 주어 수행하였습니다.

```
dku-os-2020@dkuos2020-VirtualBox:~/Desktop/OS_lab2_sync/lab2_sync$ ./lab2_bst -t 4 -c 1000000
```

1. 싱글 스레드 vs 멀티 스레드
2. Lock vs No Lock
3. fine-grained vs coarse gained

2 - 1. Multi-Thread vs Single-Thread

Single Thread 의 경우 하나의 쓰레드로 Tree create, insert, delete 의 작업을 합니다. 장점으로는 구현이 간단하다, 동기화의 필요가 없다 등의 장점이 있습니다. 하지만 위의 그래프와 같이 Multi Thread에 비해 느린 수행 속도를 보여줍니다.

Multi-Thread 의 경우 여러개의 쓰레드가 동시에 병렬적으로 처리하지만 synchronous, concurrency 에 대한 이슈가 있습니다. 여러개의 스레드가 1개의 공유 자원(tree)에서 수행하기 때문에 접근에 대한 제어가 필요 합니다.

2 – 2. Lock vs No Lock

임계영역 (공유된 자원 , tree, 에 접근 하는 영역)에는 mutex lock 제어가 필요합니다.

Lock이 없는 상황에서 10000000개의 데이터로 테스트를 해보았습니다.

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

                    Experiment info
               test node           : 10000000
                  test threads        : 4
             execution time      : 1.674560 seconds


              BST inorder iteration result :
                total node count    : 10000000

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

                    Experiment info
               test node           : 10000000
                  test threads        : 4
             execution time      : 1.290949 seconds


              BST inorder iteration result :
                total node count    : 10000000

                 Killed -> 프로세스가 죽어버림
```

프로세스가 죽어버리는 이유는 다음과 같습니다.

상호배제가 보장되지 않았음.

위의 프로세스는 tree node 삭제 과정에서 죽은걸 예상할 수 있는데 그 이유는 한개의 공유자원 (tree)에 대하여 여러개의 쓰레드가 서로 상호배제가 보장되지 않기 때문에 bst의 특성상 node와 node가 포인터로 연결되어 있는데 이런 포인터 관계가 바뀌는 과정에서 다른 스레드가 접근할 경우 원하지 않는 수행결과를 발생시킵니다.

반면 Lock 이 보장되는 경우 테스트를 진행 해본 결과

```
dku-os-2020@dkuos2020-VirtualBox:~/Desktop/OS_lab2_sync/lab2_sync$ ./lab2_bst -t 3 -c
1000000

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

                    Experiment info
               test node           : 1000000
                  test threads        : 3
             execution time      : 1.045181 seconds


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

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

                        Experiment info
                   test node           : 1000000
                      test threads        : 3
                execution time      : 0.173232 seconds


                  BST inorder iteration result :
                    total node count    : 1000000

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

                        Experiment info
                   test node           : 1000000
                      test threads        : 3
                execution time      : 0.142385 seconds


                  BST inorder iteration result :
                    total node count    : 1000000

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

                        Experiment info
                   test node           : 1000000
                      test threads        : 3
                execution time      : 0.089012 seconds


                  BST inorder iteration result :
                    total node count    : 1000000

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

                        Experiment info
                   test node           : 1000000
                      test threads        : 3
                execution time      : 0.017323 seconds


                  BST inorder iteration result :
                    total node count    : 1000000

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

                        Experiment info
                   test node           : 1000000
                      test threads        : 3
                execution time      : 0.003670 seconds


                  BST inorder iteration result :
                    total node count    : 1000000


dku-os-2020@dkuos2020-VirtualBox:~/Desktop/OS_lab2_sync/lab2_sync$
```

Shell Prompt 가 제대로 뜨면서 완료된 모습을 볼 수 있습니다.

2 - 3. fine grained vs coarse gained

Coarse gained lock의 경우 insert, delete 의 경우 둘다 tree 전체에 대한 lock 을 걸기 때문에 하나의 쓰레드가 tree insert/ delete에 대한 수행이 진행중일 경우 mutex를 가지고 있기 때문에 다른 쓰레드가 tree에 접근하지 못하고 대기합니다.

반면 fine grained의 경우, node -> node, 노드에서 다른 노드로 옮겨가는 경우에는 node에 대한 lock을 구현하고 insert, delete가 이루어 지는 순간에만, 즉, 최소한의 꼭 필요한 구간에만 tree에 lock을 걸어 구현하면 tree에 대해 여러 스레드가 동시에 접근하여 서로가 서로를 방해하지 않는 범위내에서 필요한 작업을 수행하게 됩니다.

위에서 비교한대로 수행시간에서 큰 차이를 보입니다.

3. Discussion

3 - 1. 설계 과정 - Recursive Lock에 대한 분석

“ int lab2_node_remove(lab2_tree *tree, int key) ,,

에 대한 구현 방법을 생각하던중 GitHub에서 다양한 자료를 분석하던 도중 Recursive 하게 구현된 가독성이 좋은 코드들을 분석 해보았습니다. 하지만 과연 Multi-threading 환경에서도 DeadLock 상태에 빠지지 않고 구현할 수 있을까? 에 대한 의문이 들었습니다. 아래와 같은 코드로 두개의 쓰레드로 Recursive 환경을 테스트 해보았습니다.

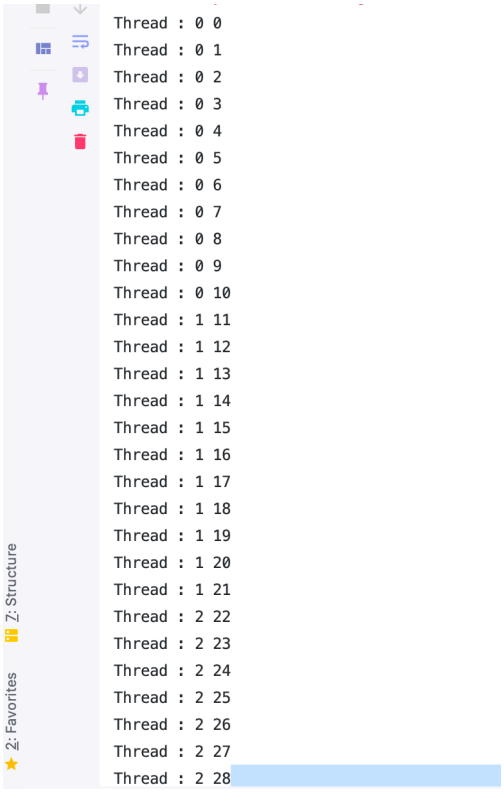
```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

std::recursive_mutex locker;

int bst = 0;

void recursion(char c, int loop)
{
    if (loop < 0)
        return;
    locker.lock();
    cout << "Thread : " << c << " " << bst++ << endl;
    recursion(c, --loop);
    locker.unlock();
}
int main()
{
    thread t1(recursion, '0', 10);
    thread t2(recursion, '1', 10);
    t1.join();
    t2.join();
    return 0;
}
```

수행 결과는 우측의 그림과 같이 bst 라는 변수의 공유 자원에 대해 순서를 지켜가면서 Lock이 제대로 진행됨을 확인했습니다.

의문점은 과연 Multi-Threading 환경에서도 정상적으로 작동할까? 였습니다.

이에 대해 조사 해본 결과 가능하다 였습니다. 실제로 Objective –C 등에서는 Recursive Block 을 위한 API를 제공합니다.

하지만 Recursive Block은 처음 mutex를 획득한 쓰레드가 Recursive하게 호출되는 동안 주도권을 가져가기 때문에 fine grained 와 coarse grained lock을 구현하고 비교하는 이 과제의 취지와는 맞지 않다고 생각하여 iterative 한 방법으로 lab2_node_remove 함수를 구현 하였습니다.



3 – 2. 코드 구현 과정 – 세그멘테이션 오류

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

 Experiment info
    test node           : 10000
    test threads        : 4
    execution time      : 0.001840 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      : 0.001096 seconds


 BST inorder iteration result :
    total node count    : 10000

Segmentation fault (core dumped)
dku-os-2020@dkuos2020-VirtualBox:~/Desktop/OS_lab2_sync/lab2_sync$
```

코드를 구현하는 과정에서 ‘Segmentation Fault (core dumped)’에 시달려야 했습니다. 처음엔 GDB를 이용하여 디버깅해보려 시도하였지만 main문에서 break point를 잡는게 쉽지 않았습니다.

```
    -t: num thread, must be bigger than 0 ( e.g. 4 )
    -c: test node count, must be bigger than 0 ( e.g. 10000000 )

 Example :
    #sudo /home/dku-os-2020/Desktop/OS_lab2_sync/lab2_sync/lab2_bst -t 4 -c 1000
0000
    #sudo /home/dku-os-2020/Desktop/OS_lab2_sync/lab2_sync/lab2_bst -t 4 -c 1000
0000
[Inferior 1 (process 1551) exited with code 0377]
Warning: not running
gdb-peda$ l
281         exe_time = get_timeval(&tv_delete_start, &tv_delete_end);
282
283         print_result(tree, num_threads, node_count, is_sync, LAB2_OPTYPE_DEL
ETE, exe_time);
284         lab2_tree_delete(tree);
285
286         printf("\n");
287
288         free(threads);
289         free(data);
290     }
gdb-peda$
```

구글링하던 도중 세그멘테이션 오류의 경우

``` 세그멘테이션 결함의 주된 원인은 다음과 같다.

1. 프로그램이 허용되지 않은 메모리 영역에 접근 시도

2. 허용되지 않은 방법으로 메모리 영역에 접근을 시도 '''

출처: [https://ho-j.tistory.com/8]

라는 글을 읽고 다시 코드를 검토해본결과 위에 코드 설명부분에서 다루었던 parent node가 root node 와 동일한 경우에 대한 에러처리가 제대로 되지 않고 있음을 알고 수정하여 원하는 결과를 얻었습니다.

```
    if (parent->left == curr)
        parent->left = curr->left;
    if (parent != NULL)
    {
        if (parent->left == curr)
            parent->left = curr->left;
        else
            parent->right = curr->left;
    }
    else
        parent->right = curr->left;

    free(curr);
        (tree->root = curr->left);
    curr = NULL;
```
```

추가 1. 형상 관리 툴 / Git

이 과제를 진행하는 기간동안 Git/ GitHub를 처음으로 사용하여 과제를 진행 하였습니다. 그 전부터 협업의 중요성, 형상 관리의 중요성에 대해 익히 들어왔지만 실제로 사용해본건 이번 과제를 진행하는 주간이 처음이였습니다. 느낀점은 다음과 같습니다.

3-2-1. 운영체제와 무관하게 사용할 수 있다.

Virtual Box 로 Linux 우분투 환경이 느리고 가상 머신이라는 한계점 때문에 불편한 점이 많아서 Mac OS 환경에서 코드를 작성하고 Linux 환경에서 테스트 하는 방식으로 진행하였습니다. Git을 이용하여 운영체제에 상관없이 command line을 통해서 Git push/ pull의 명령으로 쉽고 빠르게 코드를 가져와 테스트, 수정의 과정을 진행하였습니다.

추가 2. 실생활 멀티 스레딩?

과제를 진행하면서 느낀점이 병렬 처리를 위한 멀티 스레드 환경을 구현하는게 더 복잡하고 고려해야할 사항도 많은데 왜 해야하며 어디에 쓰이는지 알아 보았습니다.

멀티 스레드 기반 소켓 프로그래밍

클라이언트/ 서버에서 서버에 다수의 클라이언트 Request가 들어왔을때 서버에서 자식 프로세스를 만들어 클라이언트 요청을 처리하는 '멀티 스레드 기반 소켓 프로그램' 이 서버에 request가 많은 경우 효율적인 작업 수행을 가능하게 하는 프로그래밍이 가능하게 하는게 '멀티 스레드' 란걸 알았습니다.