REPORT

교 과 명 : 운영체제 (2분반)

담당교수 : 최종무 교수님

학 과 : 소프트웨어학과

학 번 : 32160204 / 32164172

이 름 : 공채운 / 정유석

DKU 단국대학교 DANKOOK UNIVERSITY

목차

-정유석

-공채운

구현환경

- 언어: C언어
- 운영체제: Ubuntu 18.04
- 분산 버전 관리 시스템: git (github)

[정유석]

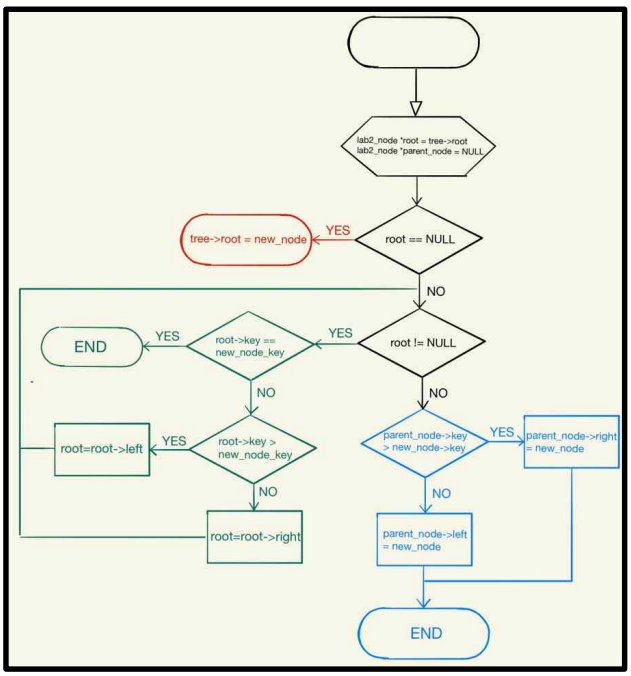
<Insert 함수 구현 및 segmentation fault 확인 >

" lab2_node_insert_cg "함수와 " lab2_node_insert_fg " 함수를 구현하기 위하여

Insert 함수를 먼저 구현하였다. 해당 함수를 3가지 기능으로 나누면 다음과 같다.

1. Tree의 루트에 아무런 노드가 없는가 확인한 조건문
2. new_node를 넣을 트리의 최하 노드를 찾는 반복문
3. new_node를 최하 노드의 left / right을 넣을지 정하는 조건문

이러한 순서에 맞추어 구현한 " lab2_node_insert " 의 알고리즘은 이하와 같다.



<Insert 함수의 알고리즘 >

이렇게 insert함수를 완성시켰다. 해당 함수를 " lab2_node_insert_cg "함수와 " lab2_node_insert_fg" 에도 동일하게 넣고 segmentation fault가 발생되는지 확인하는 과정을 거쳤다.

```
dku-os-2020@dkuos2020-VirtualBox:~/DKU/prac/lab2_sync$ ./lab2_bst -t 1000000 -c 10000000
=====  Multi thread coarse-grained BST  insert experiment  =====

 Experiment info
    test node           : 10000000
    test threads        : 1000000
    execution time      : 4.363387 seconds


 BST inorder iteration result :
    total node count    : 10000000

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

< insert 함수의 segemenation fault >

insert함수로부터 오류를 발생시키기 위하여 thread를 1,000,000개, node를 10,000,000를 몇 차례 실행해 보았고 위과 같은 결과를 얻어낼 수 있었다. 이렇게 Segmentation fault의 발생을 확인하였다. 그렇다면 어느 부분에서 해당 문제가 발생하는 것일까.

< 어느 부분에서 segmentation fault가 발생할까 >

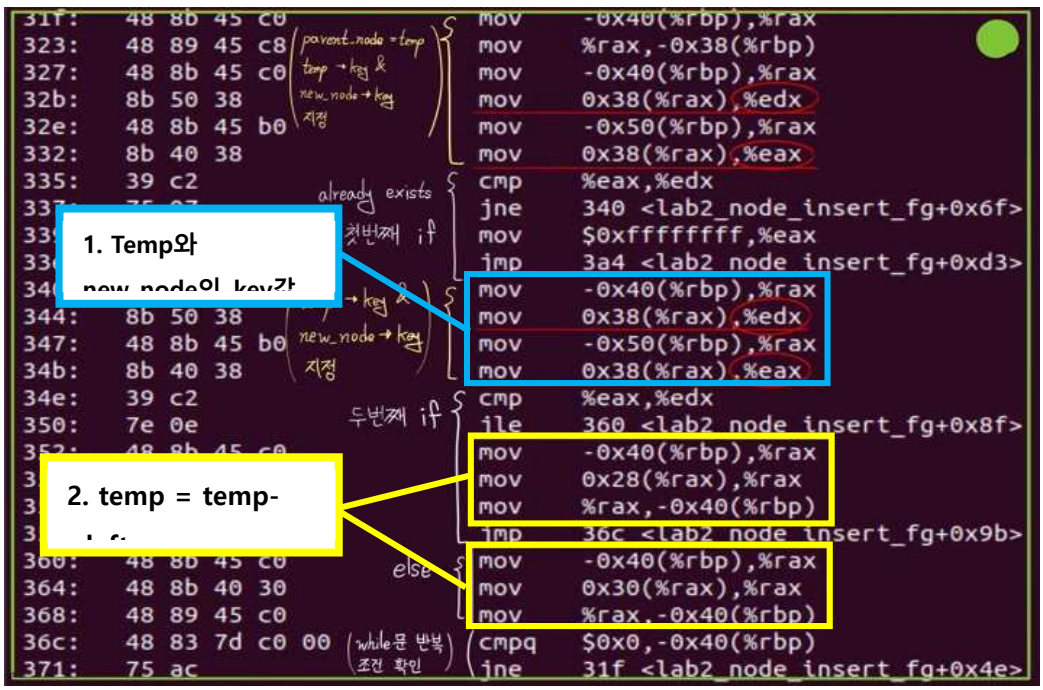
멀티스레드 프로그램이 가지는 문제점은 다수의 스레드가 '**공유데이터**'를 놓고 경쟁을 하게 된다는 것이다. 우리는 이러한 문제를 해결하기 위해 **critical section**을 atomic하게 만들거나 lock을 걸어 올바르게 동기화 시킨다.

그렇기 때문에 나는 해당 과제에서는 (1) critical section을 찾고, 해당 critical section의 공유데이터를 (2)다른 스레드가 간섭할 여지가 있는지 파악할 필요가 있다고 생각하였다.

```
while (temp != NULL) //Goes down until it meets the downmost tree
{
    parent_node = temp;
    if (temp->key == new_node->key) // already exists
        return LAB2_ERROR;
    if (temp->key > new_node->key) //Set up proper location of the root
        temp = temp->left;
    else
        temp = temp->right;
}
```

< bst.c 파일 (insert함수의 2번 과정) >

위의 사진은 insert 함수 2번 과정의 코드이다. 해당 명령어들이 atomic하게 일어나는지 확인하기 위하여 objdump 명령어를 통하여 어셈블리어를 확인하였다.



< bst.o 파일 (insert함수의 2번 과정) >

위의 사진은 insert함수의 일부인 2번 과정의 어셈블리어를 가져온 것이다.

두번째 조건문에서의 고려하여도 첫번째 조건문의 고려사항을 전부 포함한다 판단하였기에 두번째 조건문만을 고려하여 결론을 도출하였다. 고려사항 다음 2가지 였다.

1. key값을 지정하는 과정
2. key값 비교 후, root 포인터의 재지정 과정

1번 과정은 노드의 key값에 접근하는 과정이다. 위의 사진을 보면 알 듯이, atomic하지 않으며 참조하는 과정에서 문제가 발생할 수도 있을 것이다. 하지만 해당 프로그램에서 어떤 함수도 노드의 key값을 수정 및 제거하는 기능을 가지고 있지 않기 때문에, 다른 스레드가 key값의 변동을 주는 일은 일어나지 않을 것이라고 판단하였다. 물론 key값을 얻으려는 과정 중간에 delete함수로 인하여 node의 메모리가 free 되어버린다면 문제가 발생할 수 도 있지 않을까 라는 생각을 하였지만, 해당 프로그램에서는 insert 과정과 delete과정에 동시에 일어나는 일은 존재하지 않기 때문에, lock을 걸지 않아도 공유데이터의 무결성이 보장된다고 판단하였다.

2번 과정은 1번 과정과 다르게 root 포인터를 left 혹은 right로 재지정하는 과정에서 문제가 발생할 수 있다고 판단하였다. 한 스레드 A가 temp 포인터를 temp->left (1)로 옮기려고 하는 과정 도중, 다른 스레드B가 temp 포인터를 다른 temp->left(2)로 옮겨버린다면 스레드A는 temp->left(2)가 되어버린 temp에 temp->left(2)를 덮어버리는 경우가 발생하여 문제가 생길 수 있다고 판단하였다.

이렇게 얻어진 결론은 다음과 같다.

1. Node의 키 값을 참조하는 과정은 동기화 문제가 발생하지 않는다.

   (단, insert함수와 delete함수가 동시에 일어날 수 있게 프로그램이 만들어진다면 동기화를 해줄 필요가 있다. But 해당 프로그램은 이러한 경우에 포함되지 않는다.)

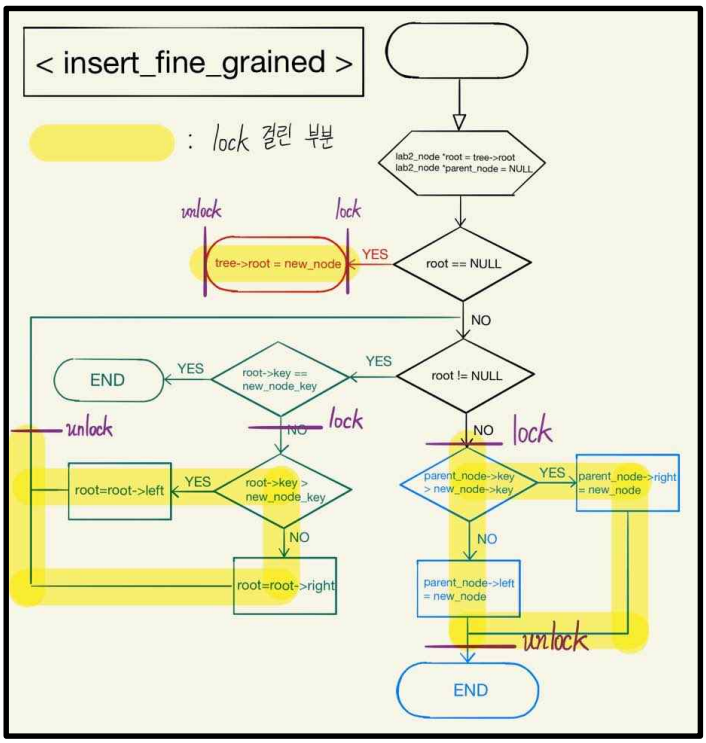
2. Node가 타 node 포인터 (left / right)를 참조하는 과정은 lock을 통하여 동기화를 보장해 주어야한다.

ps. [key 값 참조 과정을 동기화 시켜야 한다면?]

key 값 참조과정에도 동기화를 해주어야 한다면 lock을 통하여도 보장해줄 수 있지만, lock의 구현이 너무나도 복잡해질 것이라고 생각된다. 함수의 지역변수는 다른 스레드에 의해 간섭 받지 못하는 변수이다. 그러므로 함수 내에 지역변수를 선언하여 key값을 따로 대입하는 과정 따로 동기화를 보장해주는 것이 보다 동시성을 향상시킬 거라고 생각한다.

이렇게 도출된 결론들을 이용하여 fine_grained를 구현하였다.

< Insert_fine_grained >

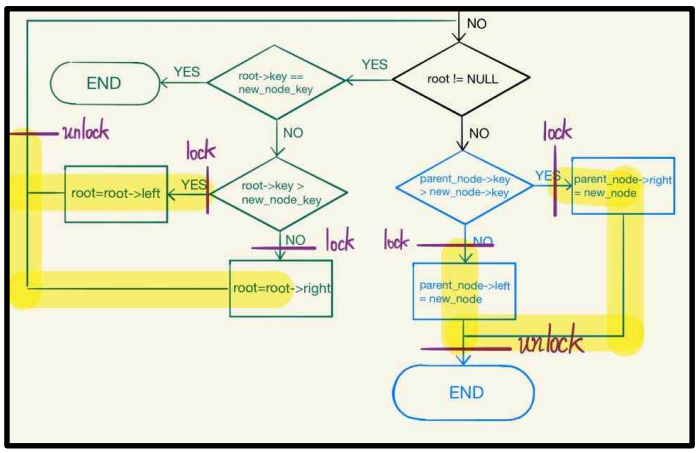


< lab2_node_insert_fg의 알고리즘 >

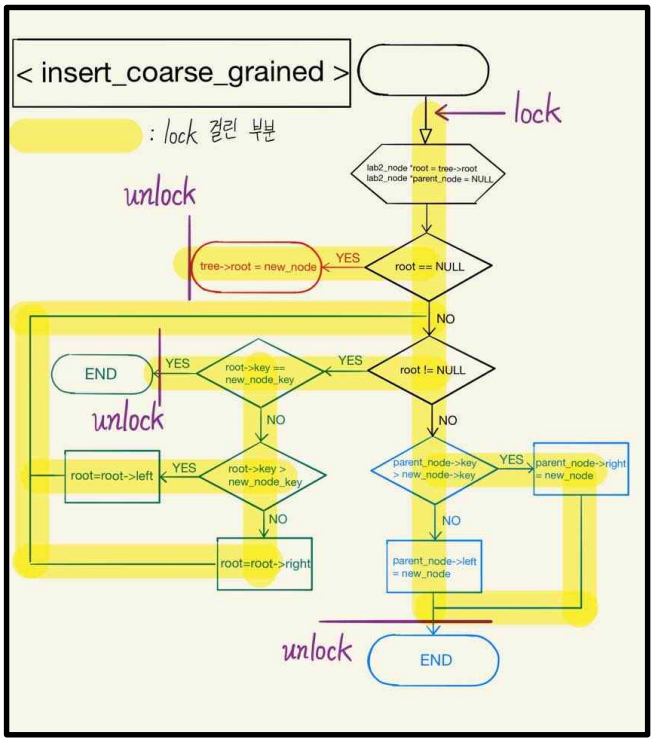
위는 "lab2_node_insert_fg" 알고리즘의 lock의 구현과정이다.

위에서 도출한 결론들을 이용하여

" root=root->left / root=root->right / parent_node->right=new_node / parent_node->left=new_node / tree->root=new_node " 과정에 lock이 필요하다고 판단하여 동기화를 보장하였다. 결론에 맞게 critical section을 더 정밀히 분리하면 좌측의 사진과 같은 구조로 구현을 해야 하지만, lock의 개수와 unlock의 개수가 pair를 이루지 못하면 문제가 생길 수도 있다고 간단히 수업을 통해 듣게 되어 pair를 이루게 하는 위와 같은 구조로 구현하였다.



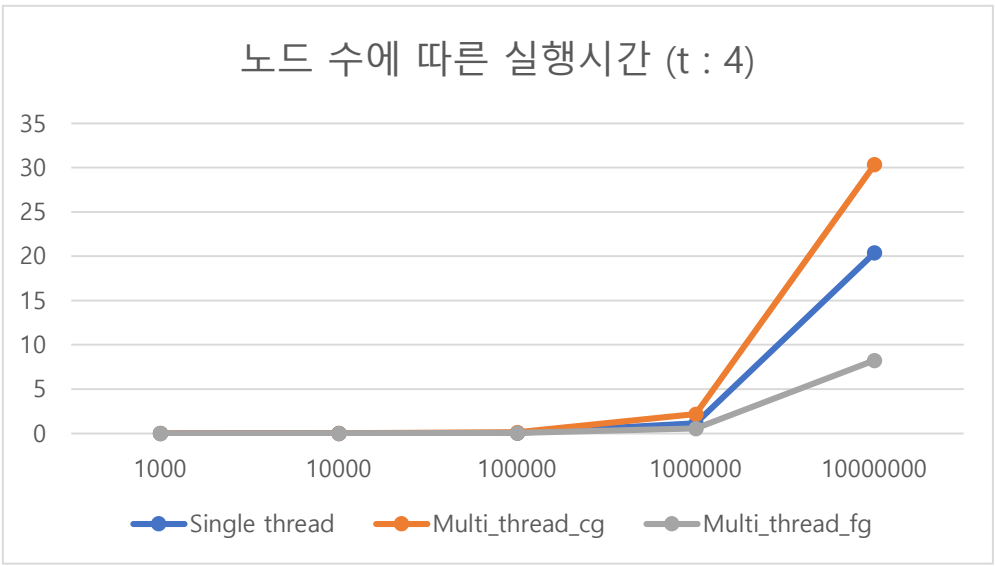
< Insert_coarse_grained>



< lab2_node_insert_cg의 알고리즘 >

위는 "lab2_node_insert_cg" 알고리즘의 lock의 구현과정이다. 동기화의 문제가 생기지 않도록 맨 처음부터 lock을 함수가 리턴 될 때 unlock을 하는 방식으로 coarse grain을 구현하였다. 사실 이렇게 coarse grain을 구현할 시, single thread로 실행했을 때와 별 차이가 없을 뿐 더러, 오히려 lock과 unlock으로 인해 코드만 더 늘어났으니 실행시간이 더 늘어날 것이라고 생각한다.

< 실행 결과>



쓰레드의 수를 4개로 설정하였을 때, 노드 수에 따른 실행시간은 위의 그래프와 같다.

예상과 같이 coarse grained의 경우, 처음부터 끝까지 직렬성을 보장하였기 때문에, single thread와 다를 게 없으며 오히려 code가 더 길어져 실행시간에 안 좋은 영향을 끼쳤다는 것을 확인할 수 있었다. Fine grained의 경우, 노드의 수가 늘어나면 늘어날수록, single thread와 coarse grained과의 성능차이를 보이고 있다. cpu의 한계로 인하여 노드의 개수가 100,000,000일 때를 실험해보지는 못했지만 single / cg / fg 간의 갭은 더욱 넓어질 것이라고 생각한다.

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

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 19.924189 seconds


 BST inorder iteration result :
    total node count    : 10000000

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

 Experiment info
    test node           : 10000000
    test threads        : 4
    execution time      : 27.433291 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      : 8.504079 seconds


 BST inorder iteration result :
    total node count    : 10000000
```

< 실행 예시 >

< Discussion >

- **Segmentation 발생유도**

Segmentaiton fault가 일어나는 것은 결국의 확률적으로 일어나는 오류인데, delete함수에 비하여 공유데이터의 참조가 수가 적어서 그런지 insert함수로부터 segementation fault를 발생시키는데 어려움이 있었다. (실행하는 스레드의 수와 노드의 수를 과하게 늘릴시 프로세스가 killed가 발생하였는데 " dmesg | grep -E -I -B100 'killed process' "명령어를 통하여 프로세스의 중단원인을 파악할 수 있음을 알게 되었다.)

- **어셈블리어 해석**

한 문장의 코드일 지라도 컴퓨터가 실질적으로 실행하는 과정은 여러 줄일 수 있기 때문에 어셈

블리어를 통해 어느 코드에서 문제가 될 수 있는지 확실시하고 싶었는데 어셈블리어가 아직 낯설 뿐만 아니라 포인터의 참조를 어셈블리어로 본적이 없어서 기능들을 구별하는데 몇몇 어려움이 있었다. 인터넷자료들과 이전 학기의 시스템프로그램 자료들을 참고하여 해석하였다. 어셈블리어를 보다보면 느낀 점은 어셈블리어에서 반복문과 조건문 등이 판별하기 쉽기 때문에 jne 등의 코드를 기준으로 파악해 나아가면 내가 파악하고자 하는 부분을 쉽게 찾을 수 있다는 것 이였다.

- **깃허브 사용**

처음으로 깃허브를 사용한 과제였기 때문에 많은 어려움이 있었다. 깃허브의 commit 및 branch의 개념을 이해하느라 다소 어려움이 있었으며, commit을 만들면 자동으로 원격저장소에 저장이 되는 것이 아니며 local 저장소와 원격저장소가 구분되는 사실과 그 구분을 어떻게 분별하는지에 대해서 또한 이해할 수 있게 되었다. jys 라는 branch를 생성한 후, 다시 처음부터 clone을 했었었는데 jys branch는 가져오지 않은 채, master branch만 clone을 하여 내가 작성하였던 코드를 못 가져오는 불상사가 일어났었다. 또한 commit을 하려던 도중 timeline이 맞지 않아 이를 고치려고 pull을 실행하였다가 작성하였던 코드가 모두 덮여져 버리는 일 또한 발생하였었다. 다소 어려운 점들이 있었지만, 팀원과 같이 프로젝트를 진행하는데 매우 편하다는 것을 느꼈다.

[공채운]

1. Mutex를 사용하지 않을 경우의 문제 분석

* 발생하는 오류

- double free or corruption
- Segmentation fault

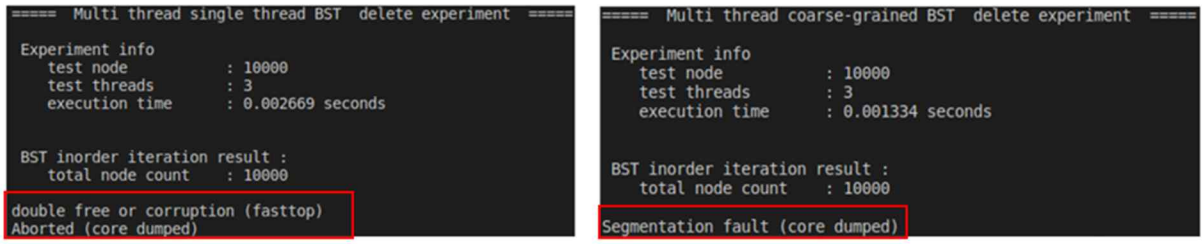


Figure 1. Without mutex

- double free or corruption의 예시

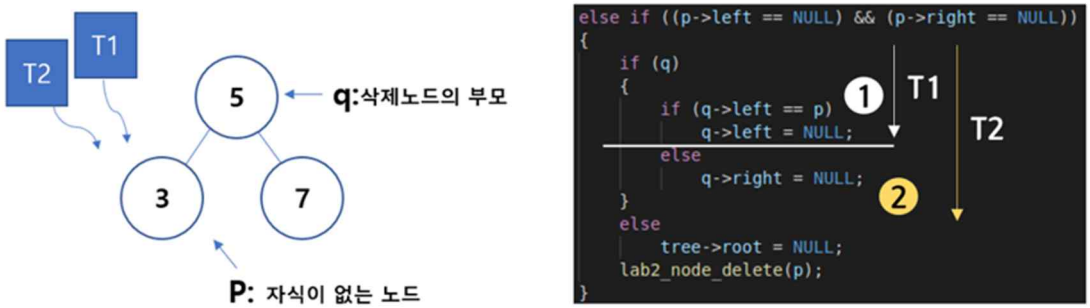


Figure 2. 자식이 없는 노드 삭제

위 코드는 자식이 없는 단말노드를 삭제하는 코드이다. 예를 들어 위의 트리처럼 키 값이 3인 자식이 없는 단말 노드를 삭제한다고 가정하면, 위 코드와 같이 실행된다. 먼저 T1 실행되어 if문의 부모 노드(q)의 왼쪽이 삭제 노드(p)인걸 확인하고 부모노드의 왼쪽을 끊게 된다. 그리고 이때 운이 나쁘게도 T1이 선점되고 T2가 진행된다고 가정하면 if문에서 부모(q)의 왼쪽이 삭제 노드(p)가 아닌걸 확인하게 된다. 결국 오른쪽에 노드가 연결되어 있음으로 인지하고 연결을 끊게 된다.

그리고 이때 계속해서 진행된다면, T2는 우측 노드와의 연결을 끊었음에도 lab2_node_delete(p)에 의해, p가 가리키는 3의 메모리를 반납하게 된다. 언젠가 시간이 흘러 T1이 다시 스케쥴되어 lab2_node_delete(p)를 수행한다면 이미 반납된 자원인 p를 다시 한번 더 반납하려는 시도에 의해 double free or corruption 에러가 발생하게 된다.

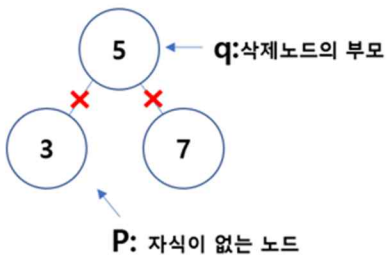


Figure 3. 코드 수행후 결과

- **Segmentation fault의 예시**

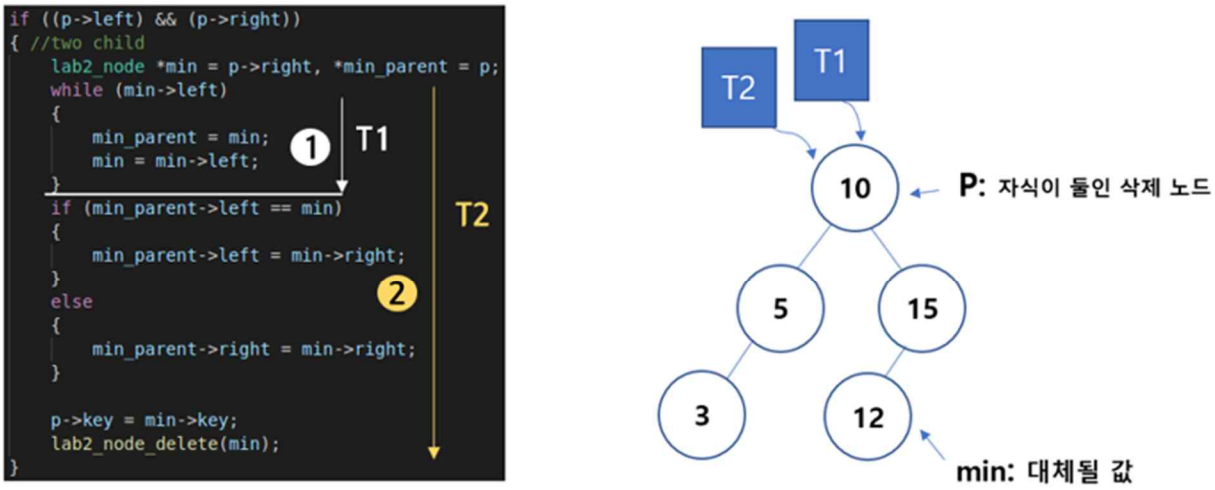


Figure 4. 자식이 둘인 노드의 삭제

위 코드와 같이 자식이 둘인 노드의 삭제를 가정해보자. 먼저 두개의 스레드 중 T1이 실행되어 삭제될 노드인 10에 대체할 값인 우측 서브트리의 최소값인 12를 찾고 선점되었다. 그 이후 T2가 수행되어 대체될 값인 12를 찾고 추가적인 작업을 통해 가리키던 min를 삭제하여 아래 Figure 5와 같은 트리 상태가 되었다.

다시 T1이 스케쥴되어 수행하게 된다면 현재 min은 삭제되어 잘못된 메모리 공간을 참조하고 있고, min->right, min->key 등의 연산을 하게 될 경우 Segmentation fault가 발생하여 프로그램이 종료되게 된다.

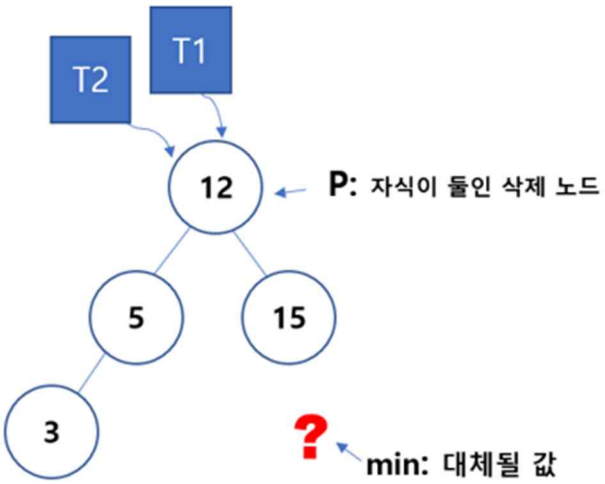
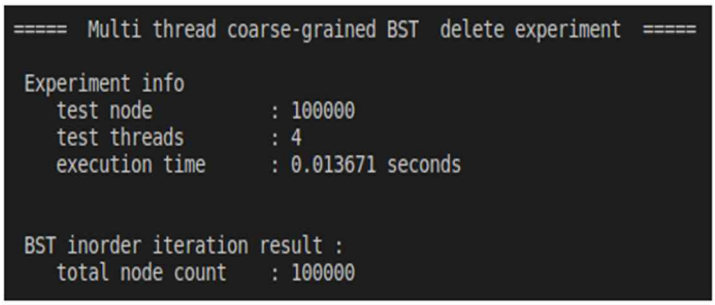


Figure 5. 삭제된 이후

2. Mutex를 사용할 경우

앞에서 확인한 문제들에 의해 Multi thread 상황에서는 필수적으로 병행성이 있는 자료구조를 사용해야한다. 아래 이미지는 lock을 이용하여 상호 배제가 보장되는 이진 탐색 트리의 실행 결과이다. 에러가 없이 정상적으로 실행 됨을 확인할 수 있다.



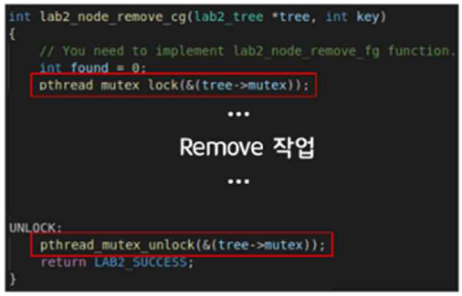


Figure 6. With Mutex. Coarse-grained Lock을 통한 concurrent BST

remove 함수 양 끝에 lock/unlock을 배치하여 스레드 T1이 remove를 실행 중이라면 다른 스레드는 임계영역인 이진 탐색 트리에 접근할 수 없게 된다. T1이 작업을 마치고 unlock을 하게 될 경우 T2가 트리에 접근할 수 있게 된다.

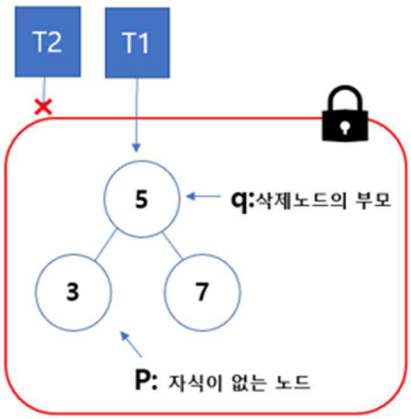


Figure 7. coarse-grained lock의 도식화

3. Fine-grained lock과 Coarse-grained lock의 성능 비교

- 성능 비교 방식

노드 10^7개에 대해서 모두 삭제하는 연산을 10번 반복하여 평균을 낸다.

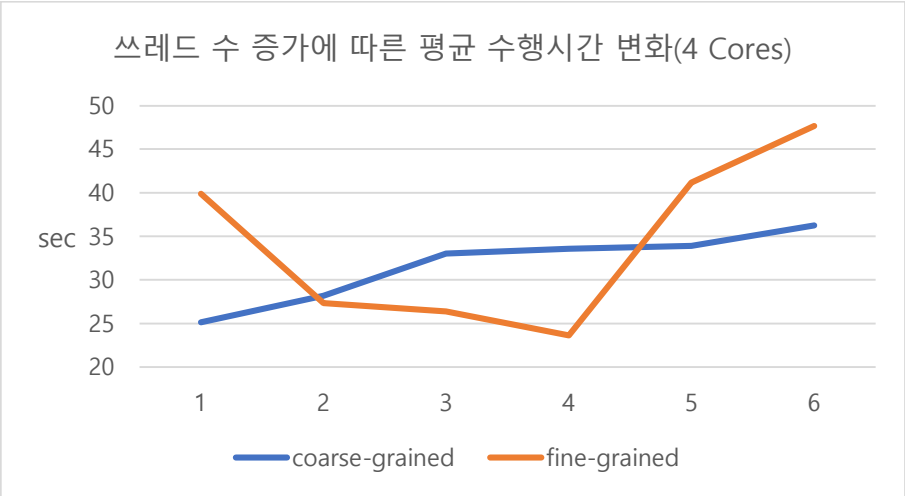
- 성능 분석

1) coarse grained lock과 fine grained lock 수행시간 분석

2) 추가 분석. VirtualBox Core 개수 변화에 따른 fine grained lock의 성능 비교

3) 추가 분석. Multithreaded에서 coarse-grained lock이 fine-grained-lock 보다 좋은 경우

1) coarse grained lock과 fine grained lock 수행시간 분석



*** Coarse-grained lock의 분석**

Coarse-grained lock의 경우 쓰레드 수가 증가함에 따라 점차 수행시간이 늘어났다. Coarse-grained의 경우 사실상 lock이 remove 함수 전체에 적용되어 직렬적으로 수행된다. 따라서 쓰레드 수가 증가할수록 context switch에 대한 부하가 점점 커진 것으로 보인다.

*** Fine-grained lock의 분석**

Fine-grained lock의 경우 쓰레드 수가 증가함에 따라 수행시간이 줄어들다가 4를 기점으로 급격히 성능이 저하되었다. 우선 수행시간이 줄어든 이유는 lock이 필요한 부분에만 걸려있어 lock contention이 적기 때문에 점차 성능이 좋아진 것으로 보인다.

그리고, 4 이후로 급격히 성능이 저하된 이유는 수행 환경(가상머신의 코어 수)과 관련이 있어보인다. 가상머신의 코어 수를 4개로 할당하였는데 1코어당 1개의 쓰레드를 수행할 때 가장 성능이 좋은 것으로 보여진다. 이와 관련하여 추가적인 분석을 아래에서 다루었다.

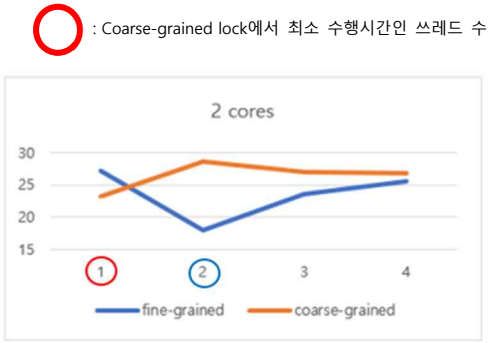
*** Coarse-grained lock과 Fine-grained lock의 비교**

우선 쓰레드 수가 1일 때 coarse-grained lock의 성능이 10번의 시행에서 항상 좋았다. 이는 coarse-grained lock의 경우 lab2_cg_remove_node 함수에서 lock/unlock이 단 한 쌍만 존재하기 때문에 lock에 의한 부하가 적지만, fine-grained의 경우 쓰레드가 1개 일때 불필요한 lock/unlock이 수십번 수행 되기 때문이라고 생각한다. 즉, lock의 부하가 크기 때문에 발생한 결과로 추측한다.

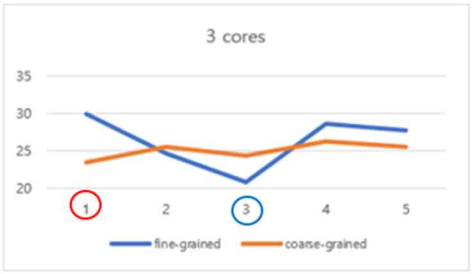
쓰레드 수가 점차 증가하며 lock/unlock의 부하보다 lock contention이 더 많은 영향을 끼치게 된 것으로 보인다. 따라서, 쓰레드 수가 증가함에 따라 fine-grained lock이 성능이 점차 좋아진다.

2) 추가 분석. VirtualBox Core 개수 변화에 따른 fine grained lock의 성능 비교

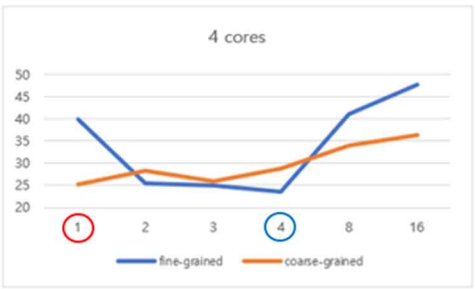
앞선 수행 결과에서 thread 수가 4일때 fine grained lock의 성능이 최적이 되고 그 이후로 성능이 급격히 나빠졌는데 그 원인을 가상 머신의 core 수에서 찾을 수 있었다.



threads	fine-grained	coarse-grained
1	27.156488	23.156487
2	17.954887	28.545412
3	23.667506	26.945012
4	25.585008	26.889652



threads	fine-grained	coarse-grained
1	29.816515	23.448973
2	24.661112	25.456771
3	20.751385	24.342862
4	28.621171	26.17454
5	27.621616	25.408336



threads	fine-grained	coarse-grained
1	39.867728	25.166432
2	25.382731	28.189205
3	24.90219	25.845125
4	23.631353	28.654881
8	41.150175	33.90343
16	47.681591	36.275095

Figure 8. 코어 수 변화에 따른 수행시간 변화

위의 표와 그래프를 보면 coarse-grained lock의 경우 코어 수에 관계없이 쓰레드 수가 1일 때 최적의 성능을 보였다. 이는 쓰레드 수가 많아질 수록 lock contention 및 context switch의 부하가 증가하여 나타난 결과로 보인다.

fine-grained lock의 경우 코어 수와 쓰레드 수가 같을 때 최적의 성능을 보였다. 이는 쓰레드 수가 코어 이상으로 많아질 수록 context switch에 의한 부하가 커지고 cache affinity에 의한 문제도 있을 것이라 추측된다.

3) 추가 분석. Multithreaded에서 coarse-grained lock이 fine-grained-lock 보다 좋은 경우

앞선 결과들을 분석하는 과정에서 추가적으로 특이한 사항을 발견할 수 있었다. 쓰레드 수가 2일 때 간혹 coarse-grained lock이 fine-grained lock보다 좋은 성능을 보였다. Single-thread가 아닌 Multi-threaded 환경에서는 fine-grained lock이 항상 성능이 좋을 것이라 예측되었는데 이와 다른 결과였다.

위와 같은 결과가 나온 이유는 크게 두가지로 볼 수 있을 것 같다. 첫번째로 실험 환경은 context switch가 계속해서 발생하는 상황인데 lab2_bst 프로세스 이외에 다른 프로세스가 많이 끼어들어 수행 결과에 영향을 준 경우. 두번째로 lock/unlock의 부하가 multi-threaded로 인한 성능 향상보다 더 큰 경우가 될 수 있다. 실제로 UNLV(University of Nevada, Las Vegas)의 Edward R. Jorgensen 교수가 작성한 논문[1] 의 119페이지 내용에 따르면, fine-grained lock의 경우 스레드 수가 적을 때 coarse-grained lock에 비해 성능이 낮다. 그 이유는 fine-grained lock의 경우 lock/unlock에 의한 오버헤드가 더 크기 때문이었다. 그리고 스레드 수가 많아져 점차 복잡해질 경우, fine-grained lock가 coarse-grained lock에 비해 성능이 좋아지게 된다는 결과를 도출된 논문이다. 비록 위 논문의 경우 자료구조가 Self-balancing binary search tree이고, search, insert, remove가 동시에 적절한 비율로 수행되는 경우로 가정했지만 일반 binary search tree에서도 유의미할 것으로 추측된다.

Discussion

- Binary Search Tree 구현

우선 기본적으로 BST가 구현되어 있어야 했기 때문에 과거 자료구조 시간에 이용한 코드를 보고 다시 이용하였다. 하지만 다시 실행해보니 그때 작성한 코드가 완벽한 코드가 아니었고, 이를 수정하는데에 약 하루 정도를 쓰게 되었다. 문제점은 root에 대한 삭제를 고려하지 않아서 발생한 문제였다.

- Coarse grained remove 구현

이를 구현하는데에는 크게 어렵지 않았다. 강의에서 배운 것 처럼 remove 함수 앞뒤에 lock/unlock을 배치시키고 적절히 코드를 수정하여 lock/unlock이 pair가 되게 하여 버그도 방지하였다.

---

[1] "Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree"
https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=4733&context=thesesdissertations

- Fine grained remove 구현

가장 많은 시간을 투자한 부분이었다. 구현의 핵심은 삭제 과정에서 쓰이는 노드들에 대해서만 lock을 거는 것이었다. 이 개념을 이해하기 위해 강의도 몇번을 다시 듣고 많은 오픈 소스들을 찾아 보았다. 노드 단위로 락을 거는 것을 결코 쉽지 않았다.

초기에는 tree전체에 대해 lock을 잡는 방식으로 불필요한 코드 부분만 mutex를 사용하지 않았다. 실제 수행해보니 coarse-grained lock과 성능 차이가 전혀 없었다. 사실 어느정도 예상한 결과였다. 성능 비교를 위해서 노드 단위의 mutex가 필수적이라고 생각이 들었다.

우선 search 부분에도 노드 단위의 lock을 설정하였는데, 이때 부터는 에러가 많이 발생하기 시작했고 간간히 성공적으로 수행되기도 했다.

다음에는 방법을 바꾸어 노드를 탐색하는 부분에 대해서만 트리 단위로 lock을 걸었다. 여기서 마주한 특이점이 약 100만번을 수행하는 동안 segmentation fault가 한번도 발생하지 않았다는 점이다. 다만, 몇개의 일부 노드는 삭제 되지 않고 남아있었다. 10000개의 노드 삭제를 수행하면 평균 9998개의 노드가 삭제되고 2개 정도가 남았다. 이후 추가로 삭제 노드 링크를 끊는 부분에도 구현을 시도 했으나 결국 실패했다. (여러 방법으로 구현하려던 시도가 깃허브 'search' branch[2]에 남아있다.)

- Coarse grained remove와 Fine grained remove의 비교

Fine grained remove에 문제가 있지만 오류가 발생하지 않고 전부 삭제된 경우만 취급하여 Coarse-grained remove와 비교가 가능해졌다.

우선 적절한 노드 수를 잡는 데에 고민을 하였다. 10^7으로 결정하게 된 이유는 context switch가 10ms마다 발생하게 되는데 노드 수가 10^7일때 보통 수행시간이 20~50초 단위로 나와 이를 충분히 무시할 정도가 될 수 있을 것이라고 판단하였다.

첫번째로 스레드 수가 1이고 노드가 10^7개일 경우로 설정하여 비교하였다. 여기서 생각한 결과와 정반대의 결과가 나타나게 되는데, fine-grained remove가 오히려 성능이 안좋게 나온 것이다. 기존에는 fine-grained lock이 coarse-grained lock에 비해 성능이 좋은 걸로 학습했기에 이로 인해 혼란에 빠지게 되었다. 왜 이러한 결과가 나타났는지 너무 궁금해서 각종 논문을 찾아보다 위 성능 분석에서 언급한 "Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree"에서 약간의 힌트를 얻을 수 있었다.

---

[2] https://github.com/Gongcu/lab2_sync/tree/search

추가적으로 혼란스러웠던 점은 쓰레드 수가 4개가 될 때 까지는 fine-grained lock의 성능이 점차 증가하다가 4개를 기점으로 급격히 성능이 안좋아졌다. 이유를 생각해보다가 가상머신의 코어 수가 4개로 설정한 것을 기억하게 되었는데 아마 이와 연관이 있을 것이라고 생각이 들었다. 따라서, 이와 관련한 추가적인 분석을 실시하였는데 4장 2)의 결과처럼 코어와 쓰레드의 수가 같을 때 성능이 최적임을 도출할 수 있었다.

-깃허브

항상 단독 프로젝트를 진행하여 단독으로 깃허브를 사용하다가 정유석 학우와 같이 처음으로 팀으로 깃허브를 이용하게 되었다. 단독으로 할 경우에는 commit, push의 개념만 이해하고 사용하였다. 이번에는 팀으로 하게 되어 branch, pull, merge 등의 개념을 학습하는 좋은 기회가 되었다. 어쩌면 앞으로 계속 사용하게 될 것으로 예측하기에 가장 의미있는 학습 내용이라고도 생각이 든다. 깃허브는 제출 마감 시한인 2020.05.06 18:00을 기점으로 private에서 public으로 전환할 예정이다.

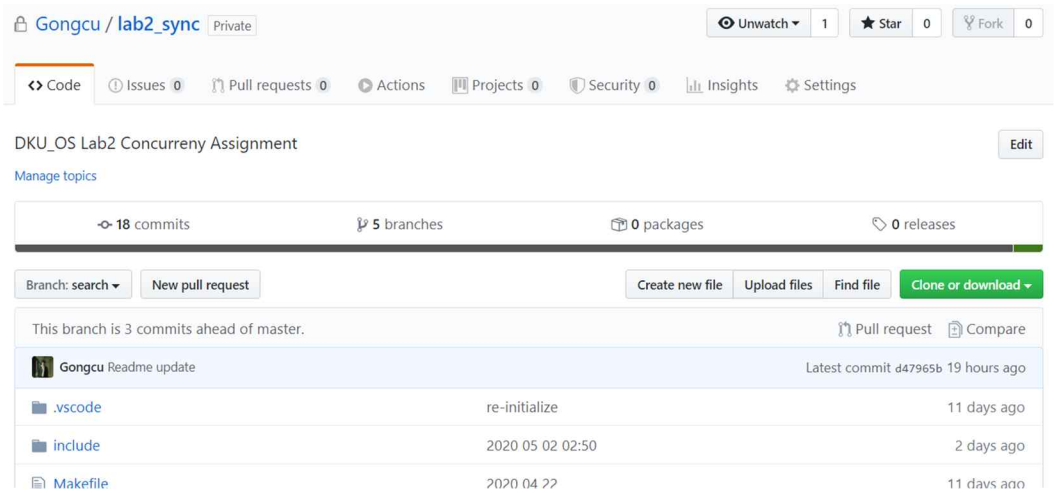


Figure 9. Github repository[3]

-총평

이번 과제를 통해 많은걸 배울 수 있었다. 기본적인 mutex의 개념과 더불어 concurrent data structure에 대해 이해할 수 있었다. 특히 인상깊었던 점은 스레드가 적을 때는 coarse-grained가 오히려 좋은 선택이 될 수도 있다는 점이다. 또한 쓰레드는 코어 수 만큼 할당하는게 최적이라는 것도 좋은 배움이었다.

[3] https://github.com/Gongcu/lab2_sync/

Fine-grained remove를 노드 단위로 mutex를 보장하는건 매우 어려웠다. 대학 생활 과제중 단일 부분에 제일 많은 시간을 투자했으며 결국 완전한 구현에는 실패했다. 절반의 구현정도만 성공하여 수행이 잘 된 케이스로 성능 비교는 가능했다. 아마 이번 과제보다 더 어려운 과제가 대학생활 동안 부여되지 않는다면 아마 가장 어려웠던 과제로 기억될 것 같다.

Fine-grained lock의 구현만큼 성능 비교 분석에도 굉장히 많은 시간을 투자했다. 초기에는 노드 10^5 개를 기준으로 성능을 분석하였는데 보통 0.1~0.9초대 정도의 수행 결과가 나와서 context switch 한번이 결과에 너무 큰 영향을 끼치게 되었다. 그 이후 앞서 말한 것 처럼 노드 10^7의 개수를 할당하게 되었다. 그럼에도 한번의 수행으로는 잘못된 결과가 초래 될 수 있어 5~10회 수행하여 평균을 내는 방식을 택하였다. 기본적으로 한번의 수행에 2분이 걸리고 이러한 수행을 쓰레드 수 변화, 코어 수 변화에 맞게 다양하게 수행하다 보니 총 수행 시간만 7~8시간 가량 걸린것 같다. 프로그램 성능 분석에 있어서 따져야 할 사항이 굉장히 많고 구현만큼 성능 테스트도 쉽지 않다는 것을 깨달았다.

또한, 안드로이드에서 어플을 구현할 때 멀티 스레드인 상황이 종종 있었는데 이 과제를 한 이후에는 좀더 구현의 완성도가 높아질 것으로 기대되고, 나중에 팀프로젝트로 깃허브를 사용하면 일의 능률이 상승될 것에 대한 기대감도 생겼다.

"대학 생활 중 가장 어려운 과제였고, 가장 많은 시간을 투자한 만큼 많은걸 배울 수 있었다."

Reference

"Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree" (UNLV, Edward R. Jorgensen) https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=4733&context=thesesdissertations