2019

Operating System Lab 2

Embedded System Lab.

LAB 2 – SYNCHRONIZATION

CHOI GUNHEE, CHOI JONG MOO

[Lab 2 Synchronization]

운영체제 수업을 통해 Race Condition 의 위험성 및 Synchronization 의 필요성에 대해 숙지하였다. 이를 바탕으로 본 과제에서는 pthread 기반 mutex 를 활용해 Race Condition 상황을 해결하고 추가로 간단한 assembly language 를 통해 atomic 하게 동작하도록 spinlock 을 직접 구현해 본다. 아래 실습 과제 위치와 설명을 적어두었다, 두번째 실습 관련 직접 수정 및 구현해야 하는 파일은 아래 빨간색으로 표기하였으며, 보너스 과제 관련되어 수정 및 구현해야 하는 파일은 **초록색**으로 표기하였다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  include  lab2_bonus.c  lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example.c
```

✓ 과제 구성

- Makefile
  - 실습 자료들을 컴파일 하기 위한 파일
- include/lab2_sync_types.h
  - 실습에 사용할 구조체 및 구현할 함수에 대한 헤더 파일.
- lab2_example.c
  - mutex 사용 예시를 보여주기 위한 파일.
- **lab2_bst.c**
  - **실습에서 구현해야 할, thread-safe BST 부분 파일.**
- lab2_bst_test.c
  - 구현한, thread-safe BST 의 test code
- **lab2_bonus.c**
  - **실습 보너스 과제를 위해 구현해야 할, spin lock 부분 파일.**
- lab2_bonus_test.c
  - 구현한 spin lock 부분의 test code.

A. Race Condition

운영체제 수업을 통해 Race Condition, Critical Section 및 이를 막기 위한 다양한 기법을 숙지하였다. 본과제에서는 그중 하나인 mutex 를 통해 Race Condition 상황을 해결해본다. 따라서 Critical Section & Race Condition 에 대한 간단한 개념을 돌아보고 mutex 의 실행 예시를 통해 간단히 사용 방법을 숙지한다.

1. Critical Section & Race Condition

Race Condition 이란 둘 이상의 입력 또는 조작이 동시에 이루어져, 프로그램의 결과가 입력, 조작의 순서에 의존하여 동작하게 되어버리는 상황을 의미하는 것이며, 동시적 조작에 있어, 공유가 되는 부분을 Critical Section 이라고 한다. Race condition 을 고려해야 하는 상황으로 Multi-thread 가 있다. Multi-thread application 은 어떤 형태로든 thread 간에 정보를 공유하는 것이 필요하다. Thread 간 어느 thread 가 작업을 완료했고, 어느 thread 가 아직 진행 중인지 서로 공유를 할 수도 있으며, 여러 thread 가 같은 공통적인 data 에 대해 작업을 수행할 수 있다. 대표적인 예로 모든 thread 가 동시에 Database 에 접근하여 data 추가, 삭제, 조작 등의 연산을 수행 할 경우, 각 작업의 진척도를 나타내는 counter 를 서로 공유해야 하며, 각 연산들이 완전하게 수행되도록 보장 할 수 있어야 한다.

2. Synchronization Primitives

Race Condition 을 방지하기 위해서는 Critical Section 에 대해 한번에 하나의 입력 또는 조작 등의 연산이 수행되도록 보장 해 주어야 하며 이를 Mutual Exclusion 기법이라고 한다. 이러한 기법에 기본적으로 atomic operation spinlock, semaphore, mutex 등이 있으며, 이를 통해 특정 공유 리소스에 대한 접근을 한번에 한 thread 로 제한하거나, 해당 리소스에 관련된 작업이 완료되기 전에 다른 작업이 시작 되지 않게 Mutual Exclusion 을 보장함으로써, Race Condition 을 방지 할 수 있다. 더 나아가 공유 자원에 대한 reader, writer 구조에서 writer 에 대해서는 Mutual Exclusion 을 보장하고 reader 에 대해서 reader 끼리 동시에 접근이 가능하되 writer 가 존재 한다면, Mutual Exclusion 으로 동작하는 read/write spinlock ,read/write semaphore 등이 있으며, Multi-thread 등의 환경에서 특정 실행 순서를 보장 해 주어야 할 때, condition variable, barrier 등이 사용된다.

i. Mutex lock

Mutex lock 은 한 번에 한 thread 만 특정 code 영역에 접근 할 수 있도록 하는 Mutual Exclusion 을 통해 Synchronization 을 제공하는 기법이며, lock 을 획득하지 못한 thread 는 lock 을 획득한 thread 가 작업을 끝내고, lock 을 반환 할 때까지

대기하게 된다. Mutex lock 을 이용하기 위해서는 변수 선언, 초기화, 적절한 함수를 사용하여야 한다. 실습 자료의 아래와 같은 위치에 mutex 사용법의 예시를 포함하였다.

- mutex 예시 파일.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  include  lab2_bonus.c  lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example.c
```

위 그림은 mutex 예시 파일의 위치를 보여준다. lab2_example.c 소스 파일을 통해 mutex의 사용 예시를 확인 할 수 있다.

```
55 /* static initialization of mutex  */
56 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
57 int shared_variable=0;
58
59 static void* add_shared_variable(void *arg){
60     thread_arg *th_arg = (thread_arg*)arg;
61     int num_iterations = th_arg->num_iterations;
62     int is_sync = th_arg->is_sync;
63     int i=0;
64
65     if(is_sync){
66         for(i=0; i < num_iterations ;i++){
67             /* Acquiring lock for critical section */
68             pthread_mutex_lock(&mutex);
69
70             /* Critical Section */
71             shared_variable++;
72
73             pthread_mutex_unlock(&mutex);
74             /* Release lock for critical section */
75         }
76     }else{
77         for(i=0; i < num_iterations ;i++){
78             shared_variable++;
79         }
80     }
81 }
82
83 int mutex_test(int num_threads, int num_iterations,int is_sync)
84 {
85     pthread_t *pthreads = NULL;
86     int res=LAB2_ERROR, i;
87     long double result= 0.0;
88     thread_arg arg;
89     arg.is_sync = is_sync;
90     arg.num_iterations = num_iterations;
91
92     pthreads = (pthread_t*)malloc(sizeof(pthread_t)*num_threads);
93     memset(pthreads, 0x0, sizeof(pthread_t) * num_threads);
94
95     for(i = 0 ; i < num_threads; i++){
96         /* Create thread  */
97         res = pthread_create(&pthreads[i], NULL, add_shared_variable,(void*)&arg);
98         if(res == LAB2_ERROR){
99             printf(" Error: _perf_metadata - pthread_create error \n");
100            goto TEST_ERROR;
101        }
102    }
103
104    for(i = 0 ; i < num_threads ; i++){
105        /* wait until working threads complete their job  */
106        pthread_join(pthreads[i], NULL);
107    }
108
109    print_result(num_threads, num_iterations, is_sync);
110
111    return LAB2_SUCCESS;
112 TEST_ERROR:
113    free(pthreads);
114    return LAB2_ERROR;
115 }
```

위 그림은 lab2_example.c 파일의 주요 코드 부분이며, 과제를 구현하는데 있어 참고자료로 첨부하였다. 위 코드는 num_threads의 수만 큼 thread를 생성하며, 각 thread는 num_iteration 만큼 반복하며 shared_variable 이라는 변수를 증가시키는 연산을 수행하는 코드이다.

- 초기화

  위 코드에서는 56번째 줄에서 정적 변수인 mutex를 정적인 방법으로 초기화 하였다. 초기화에는 정적 초기화와 동적 초기화가 있다. 정적 초기화는 mutex 변수를 위와 같이 PTHREAD_MUTEX_INIT 을 통해 초기화 한다.

  동적 초기화는 list, tree 등과 같이 heap 영역에 동적으로 생성되는 mutex 등을 초기화할 때 사용하며 아래와 같은 함수를 사용하여 초기화 한다.

  *pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)*

- critical section에 대한 lock 잡기

  위 코드에서는 68번 줄에서 thread가 critical section인 shared_variable 을 증가시키는 부분에 도입하기 전 pthread_mutex_lock함수를 통해 lock 을 잡도록 하였다. pthread_mutex_lock 함수를 통해 이미 다른 thread가 critical section에 진입하여 수행 중일 경우, 현재 thread는 lock을 잡지 못하고 대기 하여야 한다.

- critical section 에 대한 lock 풀기

  위 코드에서는 73번 줄에서 critical section을 수행한 thread가 잡고있던 lock을 풀 수 있도록 하였다. 이를 통해 lock을 기다리고 있던 다른 thread는 critical section 에 진입 할 수 있다.

➢ mutex 예시 파일 컴파일

- ***# make lab2_example***

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ make lab2_example
Compilingi lab2_sync lab2_example.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o lab2_example.o lab2_example.c
gcc -o lab2_example lab2_example.o  -lpthread
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  include  lab2_bonus.c  lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example  lab2_example.c  lab2_example.o
```

위 그림과 같이 make lab2_example 명령어를 통해 컴파일 하면 lab2_example.c 의 실행 파일인 lab2_example 이 생성된다.

➢ mutex 예시 파일 실행

- ***# ./lab2_example –t 4 –i 1000000 –s o***
- ***#./lab2_example –t 4 –i 1000000 –s m***

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_example

 Usage for ./lab2_example :
    -t: num thread, must be bigger than 0 ( e.g. 4 )
    -i: iteration count, must be bigger than 0 ( e.g. 100000 )
    -s: sync mode setting ( e.g. m : pthread mutex , o : original )

 Example :
    #sudo ./lab2_example -t 4 -i 1000000 -s o
    #sudo ./lab2_example -t 4 -i 1000000 -s m
```

위와 같은 lab2_example 실행 파일에는 t, i, s 옵션을 주어 실행할 수 있으며 각 옵션의 의미는 아래와 같다. 옵션을 주지 않고 실행하게 되거나 올바르지 않은 옵션을 주게 되면 위와 같이 사용법을 출력한다.

- t (thread) : critical section 을 동시에 수행 할 thread 의 개수를 설정한다.
- i (iteration) : critical section 에서 shared_variable 을 증가시키는 연산을 반복 수행할 횟수를 설정한다.
- s (sync mode) : critical section 을 수행 할 방법을 설정한다. o 로 설정 할 경우, mutex 를 사용하지 않고 critical section 을 수행하며, m 으로 설정 할 경우 critical section의 수행에 mutex를 사용하여 수행한다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_example -t 4 -i 1000 -s o

 Experiment info
    num_threads         : 4
    num_iterations      : 1000
    experiment type     : original(race condition)

 Experiment result :
    expected result     : 4000
    result              : 4000
```

위 그림은 critical section을 4개의 thread가 동시에 수행하며, critical section 에서 shared_variable에 대해 1000 회 증가 연산을 수행하며, mutex를 사용하지 않도록 설정한 수행 결과이다. 4개의 thread가 각각 shared_variable 증가 연산을 1000 회 수행 한다면, 수행을 결과는 4000이 될 것으로 예상 할 수 있으며, 그 결과 또한 정상적으로 출력이 되어 race condition이 발생하지 않은 것을 확인 할 수 있다. 여러 번 수행하여 결과를 확인 해 보면, race condition이 발생하여 결과가 4000 이 아닌 다르 결과가 출력되는 것을 확인 할 수 있지만, 좀더 확실히 race condition 을 확인하기 위해 반복 횟수를 늘리게 되면 아래와 같은 실행 결과를 볼 수 있다.

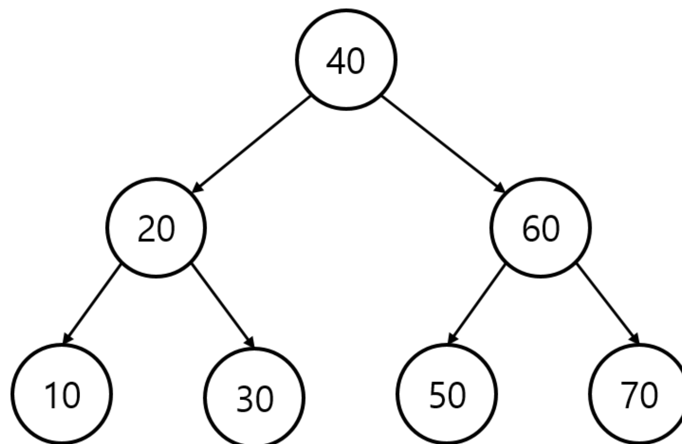
```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_example -t 4 -i 1000000 -s o

 Experiment info
    num_threads         : 4
    num_iterations      : 1000000
    experiment type     : original(race condition)

 Experiment result :
    expected result     : 4000000
    result              : 1518380
```

위 그림은 critical section을 4개의 thread가 동시에 수행하며, critical section 에서 shared_variable에 대해 1000000 회 증가 연산을 수행하며, mutex를 사용하지 않도록 설정한 수행 결과이다. 예상한 결과는 4000000 이지만, 실제 shared_variable 을 출력한 결과는 1518380 이 나온 것을 확인 할 수 있으며 실행할 때마다 그 결과가. 이를 통해 lock을 사용하지 않을 경우, critical section 에 대해 race condition이 발생하게 되어 실행 시 마다 결과가 달라지는 것을 확인 할 수 있다.

B. BST with Coarse-grained Lock & Fine-grained lock

Race Condition 으로부터 보호하기 위해서는 Critical Section 파악하고 어느 정도까지 Lock 을 통해 보호 할 것인지 결정하는 것이 중요하며 이에 따라 성능이 크게 달라질 수 있다. 본 과제에서는 Race Condition 을 해결하기 위해 Tree 자료구조를 사용한다. Tree 는 BST, Btree, B+tree, AVL tree, Red-Black tree 등 다양한 종류의 알고리즘을 사용하는 Tree 가 있으며 이중, 간단한 알고리즘이며, 2 학년 2 학기 자료구조 수업시간에 배운 BST 를 활용해 Multi-thread 수행 시 발생 할 수 있는 Race Condition 을 Coarse-grained locking, Fine-grained locking 라는 두가지 방식으로 해결하며, 그 성능을 비교한다. 따라서 과제 수행 전 BST, Coarse-grained Lock, Fine-grained Lock 의 간단한 개념 및 Multi-thread 환경에서 thread-safe 한 BST 를 구성하는 방법에 대해 숙지한다.

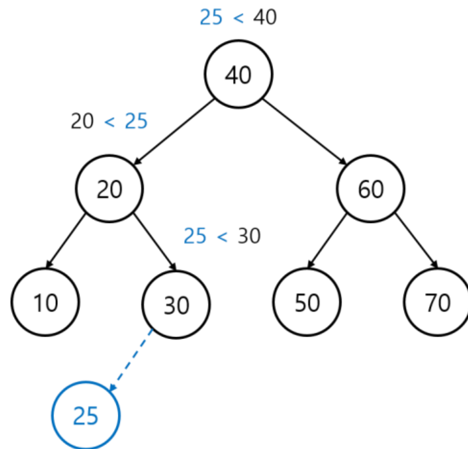
1. Binary Search Tree



BST(Binary Search Tree)는 위 그림처럼 node 마다 key 값을 가지며, left, right 두개의 child link 를 가지는 자료구조이다. 가장 상위에 있는 node 를 root node 라고 하며 가장 하단에 존재하는 node 를 leaf node 라고 한다. 각 node 의 left child link 에는 자신보다 작은 값을 가지는 node 들이 저장되며, right child link 에는 자신보다 큰 값을 가지는 node 들이 저장된다. Tree 에는 기본적으로 node 를 insert, remove, search, traversal 하는 연산 등이 있다.

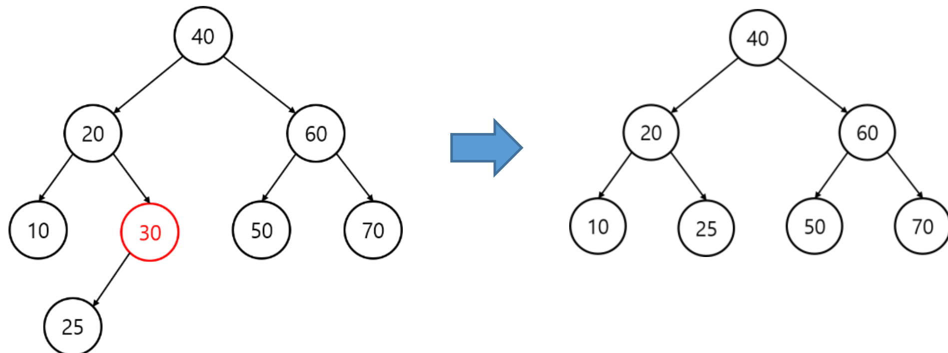
i. node insert

tree 에 새로운 node 를 추가하는 연산으로 root node 부터, key 값을 비교하여 새로운 node 가 저장될 위치를 찾아간다. 위 그림과 같은 tree 에 key 값이 25 인 node 를 추가 한다고 하면 아래그림과 같이 root node 부터 leaf node 에 도달하기 까지 각 node 와 비교하며 위치를 찾아간다.

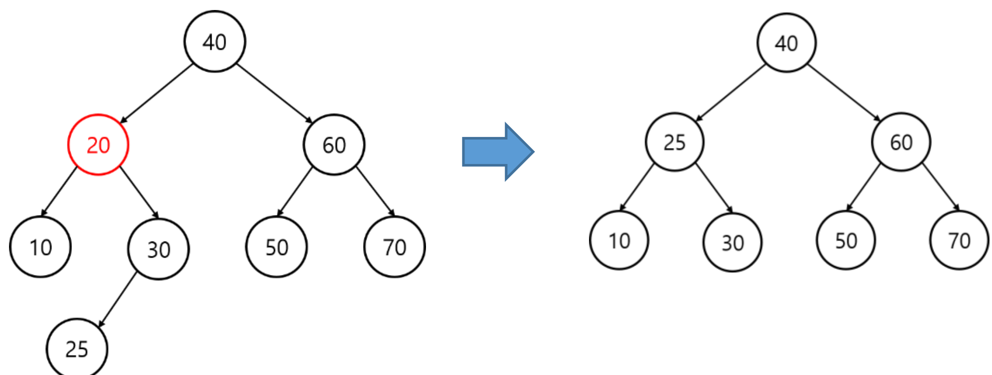


ii. node remove

tree 에 새로운 node 를 삭제하는 연산으로 먼저 삭제할 node 를 찾은 후, BST 의 구조를 유지하기 위해 삭제할 node 의 위치의 값을 재설정 한다. 삭제할 node 가 자식을 한 개만 가지거나, 자식 node 를 가지지 않을 경우는 아래 왼쪽 그림과 같이 수행한다. key 값이 30 인 node 를 삭제 한다고 하면, 오른쪽 그림처럼 삭제 후 node 값이 재설정된다.



자식이 두개인 node 를 삭제 할 때는 BST 구조를 유지하기 위해 삭제할 node 의 왼쪽 자식 sub tree 중 가장 큰 값 또는 오른쪽 자식 sub tree 중 가장 작은 값을 선택하여 삭제할 node 의 값을 재설정 한다. 아래 그림을 통해 key 값이 20 인 node 를 삭제 하려고 할 때의 결과를 볼 수 있다.



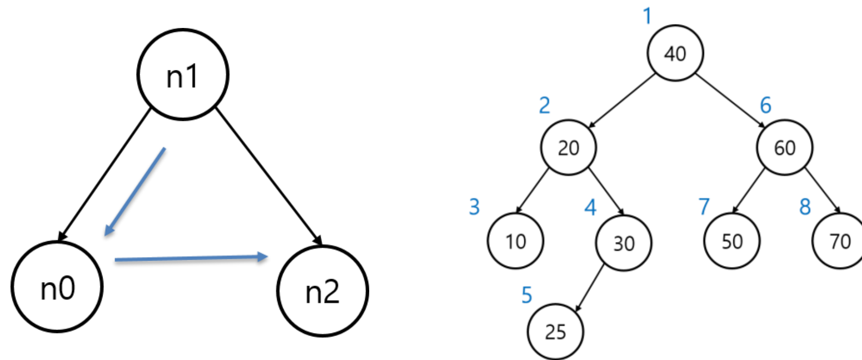
iii. node search

tree 의 node 를 찾는 연산으로 insert 연산에서 추가할 위치를 찾거나 remove 연산에서 삭제할 node 를 찾는 것과 같은 방법으로 tree 에 있는 node 를 찾는다.

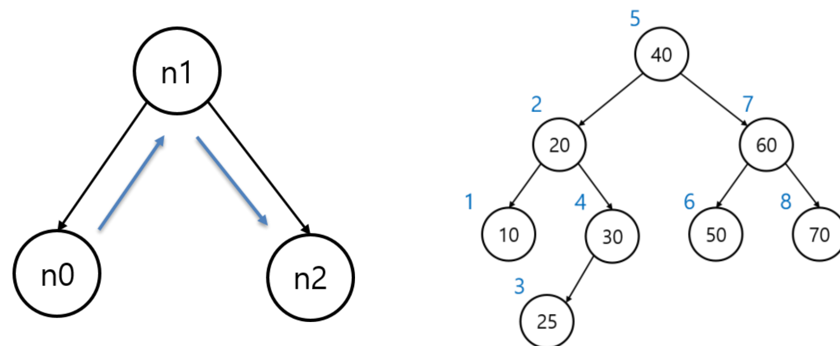
iv. node traversal

tree node 를 순회하는 연산이다. 순회에는 순회 순서에 따라 전위 순회(preorder), 중위 순회(inoder), 후위 순회(postorder)가 있으며 각각 아래와 같이 순회한다.

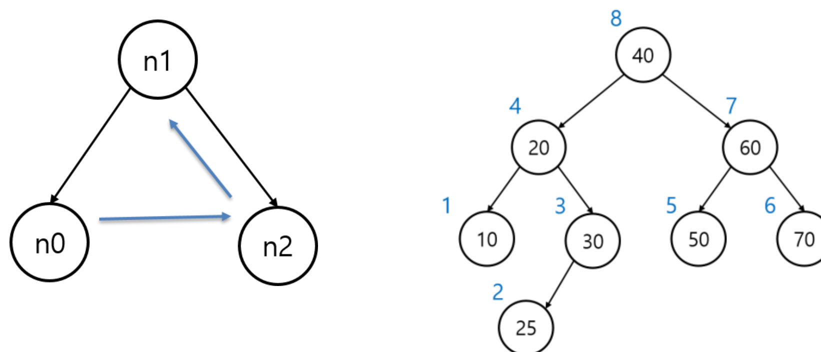
전위 순회는 아래 그림과 같은 순서대로 node 를 순회한다.



중위 순회는 아래 그림과 같은 순서대로 node 를 순회한다.



후위 순회는 아래 그림과 같은 순서대로 node 를 순회한다.



2. Coarse-grained Lock & Fine-grained Lock

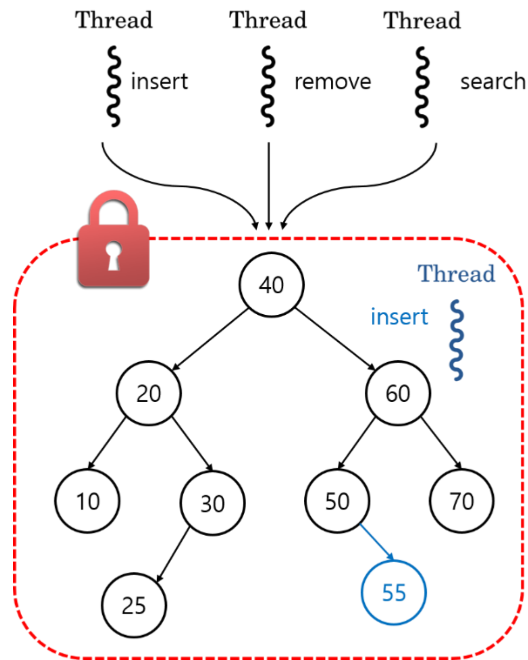
Multi-thread 등의 Critical Section 이 존재하는 환경에서 Race Condition 을 방지하기 위해 Lock 을 사용 할 수 있다. 하지만 Race Condition 이 발생하지 않더라도 Lock 을 사용하는 방식에 따라 그 성능이 크게 좌우될 수 있다. Lock 을 사용하는 방식은 Critical Section 의 선정 범위에 따라 즉 Lock 을 얼마나 세분하게 설정하느냐에 따라 Coarse-grained Locking, Fine-grained Locking 의 두가지 방식으로 사용 될 수 있다. Coarse-grained Locking 방식은 Critical Section 의 범위를 크게 잡아 많은 양의 데이터를 한번에 보호하는 방식이며, Fine-grained Lock 은 Critical Section 의 범위를 세분화하여 각 부분마다 Lock 을 사용하는 방식이다. 각각 장단점이 있으며 본 과제에서는 실행 시간 확인 및 직접 구현해 봄으로써 두가지 방식을 비교해 본다.

3. BST with Coarse-grained Lock & Fine-grained Lock

여러 thread 가 동시에 BST 에 접근하여 insert, remove, search 등의 연산을 수행 할 때, Race Condition 이 발생하여 정상적으로 동작을 하지 못하게 된다. 다양한 상황이 있을 수 있지만 예를 들어 Producer & Consumer 구조를 생각 해 볼 수 있다. Producer & Consumer 구조에서, 즉 BST 에 새로운 node 를 추가하는 producer thread 와 기존의 node 를 삭제하는 consumer thread 가 있다고 할 때, 추가를 하는 도중, consumer thread 가 경로의 특정 node 를 삭제해 버리면 문제가 될 수 있다. 또한. 또는 BST 의 특정 node data 를 읽어 들이는 Reader thread 와 BST 의 특정 node 에 data 를 수정하는 Writer thread 의 Reader & Writer 의 경우 특정 node 의 data 를 읽어 들이는 동안에 Wrier 에 의해 값이 바뀌어 버리면 문제가 발생 할 수 있다. Race Condition 을 막기 위해선 BST 의 각 연산이 안전하게 수행되는 것이 보장되어야 하며 이를 위해 Coarse-grained Lock, Fine-grained Lock 방법을 사용 할 수 있다.

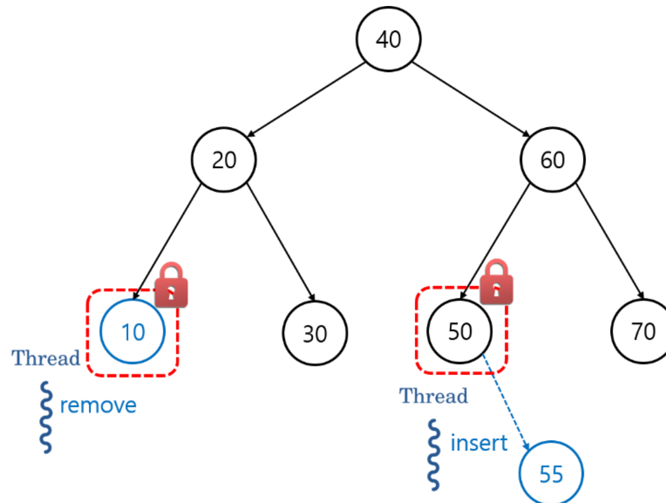
i. BST with Coarse-grained Lock

Race Condition 을 막기 위해 BST 에 대한 insert, remove, search 등의 연산 수행 시 각 연산이 수행되는 동안 다른 thread 가 BST 에 접근 할 수 없도록, Tree 전체를 Critical Section 으로 간주하여 Lock 을 사용 할 수 있다. 아래 그림은 이러한 Coarse-grained Lock 방식을 사용한 예시이다. 현재 insert 연산을 수행하는 Thread 가 Tree 를 사용 중이므로 다른 Thread 는 insert 연산이 완료 될 때까지 기다려야 하는 것을 볼 수 있다.



ii. BST with Fine-grained Lock

Race Condition 을 막기 위해 BST 에 대한 insert, remove, search 등의 연산 수행 시 연산이 수행되는 동안 접근하는 tree 의 node 에 대해 다른 연산들이 접근을 못하도록 Critical Section 의 범위를 Tree 전체가 아닌, 각 node 별로 설정하여 Lock 을 사용 할 수 있다. 아래 그림은 이러한 Fine-grained Locking 방법의 예시이다.



C. Lab2 BST Synchronization

운영체제내에서는 다양한 자료구조들이 사용되며 이들 자료 구조들 간의 또는 자료구조 내부에서 발생하는 Race Condition 을 막기 위해 다양한 기법들이 사용된다. 본 과제에서는 운영체제의 다양한 부분에서 사용되는 Tree 자료구조의 간단한 알고리즘 중 하나인 BST 를 사용하여 동기화기법을 구현한다.

1. 기본 과제 구성

Lab2 BST Synchronization 은 아래 그림과 같은 위치에 있으며 lab2_bst.c 라는 파일의 함수들을 구현해야 한다. 함수 구현은 소스 파일 내에 정해진 형식대로 구현하지 않아도 되나 정해진 형식을 변경하게 되면 include/lab2_fs_types.h 의 함수 선언을 변경해 주어야 한다. 구현해야 하는 소스파일을 빨간색으로 표시하였다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  include  lab2_bonus.c  lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example.c
```

lab2_bst.c 파일의 TODO 로 명시된 함수를 구현하여야 한다.

2. 기본 과제 컴파일 방법

아래 그림과 같이 make lab2_bst 명령어를 통해 구현한 lab2_bst.c 소스파일을 컴파일 하면 컴파일 결과 생성되는 lab2_bst 라는 실행 파일과, object 파일들을 확인 할 수 있다.

- ***# make lab2_bst***

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ make lab2_bst
Compilingi lab2_sync lab2_bst.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o lab2_bst.o lab2_bst.c
Compilingi lab2_sync lab2_bst_test.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o lab2_bst_test.o lab2_bst_test.c
Compilingi lab2_sync include/lab2_timeval.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o include/lab2_timeval.o include/lab2_timeval.c
gcc -o lab2_bst lab2_bst.o lab2_bst_test.o include/lab2_timeval.o -lpthread
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  lab2_bonus.c       lab2_bst    lab2_bst.o       lab2_bst_test.o
include   lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example.c
```

3. 기본 과제 테스트 방법

lab2_bst.c 를 구현한 후 lab2_bst 를 test 하는 시나리오는 fine-grained 와 coarse-grained 를 비교하기 위해 thread 의 수, test 할 node 의 수를 입력 받아, insert 연산과, remove 연산을 test 각각 독립적으로 수행되도록 하여야 한다.

remove 연산의 경우 random key 값 node 를 입력 받은 node 의 수만큼 생성하여 bst 에 추가해 놓고, 입력 받은 thread 의 수만큼 node 를 분할하여 BST remove 연산을 수행 후 결과를 확인한다. insert 연산의 경우 random key 값 node 를

입력 받은 node 의 수만큼 생성하는 연산을 입력 받은 thread 의 수만큼 node 를 분할하여 BST insert 연산을 수행 후 결과를 확인한다.

테스트 실행 파일은 아래와 같은 옵션을 통해 컴파일 할 수 있다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_bst

 Usage for ./lab2_bst :
    -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 ./lab2_bst -t 4 -c 10000000
    #sudo ./lab2_bst -t 4 -c 10000000
```

- t : thread 의 개수를 의미한다.
- c : test 에 생성할 node 의 개수를 의미한다. 지정한 수만큼 난수의 key 값을 만들어 node 를 생성하기 때문에 겹치는 key 값을 가지는 수가 생길 수 있다.

```
root@os-lecture:/home/os-lecture/DKU_OS_LAB/lab2_sync# ./lab2_bst -t 4 -c 10000000

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

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

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

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

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

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

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

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

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

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

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

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

lab2_bst.c 의 함수들이 제대로 구현되지 않는다면, bst 를 Multi thread 에서 실행하는데 있어 오류가 출력되며 lab2_bst test 파일이 종료될 것이다. 하지만, 정상적으로 구현하였다면 위와 같은 출력 결과를 확인 할 수 있을 것이다.(수행 결과 출력되는 execution time 은 레포트 제출을 위해 결과값을 가렸다.) 수행 결과 출력되는 수행 시간을 통해 single thread, multi-thread coarse grained, multi-thread fine grained 로 수행하였을 때의 결과를 비교 할 수 있다.

4. 보너스 과제 구성

기본 과제에서 pthread mutex 를 활용한 Synchronization 해결 기법을 구현하였다. 보너스 과제에서는 Synchronization 에 사용되는 spin lock 을 직접 구현해본다. s 구현해야 하는 소스파일을 빨간색으로 표시하였다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ls
Makefile  include  lab2_bonus.c  lab2_bonus_test.c  lab2_bst.c  lab2_bst_test.c  lab2_example.c
```

lab2_bonus.c 파일의 TODO 로 명시된 함수를 구현하여야 한다.

보너스 과제에서는 spinlock 을 atomic 하게 구현하기 위해 C source code 내에서 시스템 프로그래밍 수업에 배운 assembly language 를 사용하여야 한다. 이를 위해 lab2_bonus.c 에 함수내부에서 assembly language 의 사용 예시로 atomic_add, atomic_sub, atomic_inc, atomic_dec 함수를 첨부하였다. 보너스 과제에서는 이를 참고하여 atomic 하게 동작하는 spinlock 을 구현하여야 한다. spinlock 은 수업시간에 숙지한 test and set 연산 또는 compare and swap 으로 구현 할 수 있다.

5. 보너스 과제 컴파일 방법

아래 그림과 같이 make lab2_bonux 명령어를 통해 구현한 lab2_bonus.c 소스파일을 컴파일 하면 컴파일 결과 생성되는 lab2_bonus 라는 실행 파일과, object 파일들을 확인 할 수 있다.

- ***# make lab2_bonus***

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ make lab2_bonus
Compilingi lab2_sync lab2_bonus.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o lab2_bonus.o lab2_bonus.c
Compilingi lab2_sync lab2_bonus_test.c ...
gcc -c -g -I/home/fs-lecture/DKU_OS_LAB/lab2_sync/include/ -o lab2_bonus_test.o lab2_bonus_test.c
```

6. 보너스 과제 테스트 방법

보너스 과제 실행 파일에는 아래와 같은 옵션을 줄 수 있다.

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_bonus

 Usage for ./lab2_bonus :
    -t: num thread, must be bigger than 0 ( e.g. 4 )
    -i: iteration count, must be bigger than 0 ( e.g. 100000 )
    -s: sync mode setting (e.g. s : your implemted spin lock , o : original)

 Example :
    #sudo ./lab2_bonus -t 4 -i 1000000 -s o
    #sudo ./lab2_bonus -t 4 -i 1000000 -s s
```

-s o 옵션을 통해 spin lock 을 사용하지 않은 결과 Race Condition 이 발생 한 것을 확인 할 수 있다.

- *# ./lab2_bonus –t 4 –i 1000000 –s o*

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_bonus -t 4 -i 1000000 -s o

 Experiment info
    num_threads         : 4
    num_iterations      : 1000000
    experiment type     : original(race condition)

 Experiment result :
    expected result     : 4000000
    result              : 1396913
```

-s s 옵션을 통해 구현한 spin lock 버전으로 확인 한 결과 정상적으로 구현이 되었다면 Race Condition 이 발생하지 않은 것을 확인할 수 있다.

- *# ./lab2_bonus –t 4 –i 1000000 –s s*

```
fs-lecture@fslecture-VirtualBox:~/DKU_OS_LAB/lab2_sync$ ./lab2_bonus -t 4 -i 1000000 -s s

 Experiment info
    num_threads         : 4
    num_iterations      : 1000000
    experiment type     : lab2 spin lock

 Experiment result :
    expected result     : 4000000
    result              : 4000000
```

7. 과제 제출

i. 제출 사항.

- 구현한 lab2_sync 의 압축 파일.
- 구현한 lab2_sync/lab2_bst 의 실행결과 출력되는 single thread execution time, multi-thread coarse grained execution time, multi-thread fine grained execution time 에 대한 비교와 그 이유에 대한 레포트.

ii. 제출 방법.

✓ lab2_sync 구현 제출

아래와 같은 directory 에서 tar 명령을 통해 구현한 lab2_sync directory 에 대해 압축을 수행한 후 압축 결과 생기는 lab2_sync_학번.tar 파일을 choi_gunhee@dankook.ac.kr 으로 제출한다.

- ***# tar cvf lab2_sync_학번.tar lab2_sync***

```
root@os-lecture:/home/os-lecture/DKU_OS_LAB# ls
lab1_sched  lab2_sync  lab3_fuse
root@os-lecture:/home/os-lecture/DKU_OS_LAB# pwd
/home/os-lecture/DKU_OS_LAB
root@os-lecture:/home/os-lecture/DKU_OS_LAB# tar cvf lab2_sync_32111860.tar lab2_sync/
lab2_sync/
lab2_sync/lab2_bonus.c
lab2_sync/lab2_bst.c
lab2_sync/Makefile
lab2_sync/lab2_bonus_test.c
lab2_sync/lab2_example.c
lab2_sync/include/
lab2_sync/include/lab2_sync_types.h
lab2_sync/include/lab2_timeval.c
lab2_sync/lab2_bst_test.c
root@os-lecture:/home/os-lecture/DKU_OS_LAB# ls
lab1_sched  lab2_sync  lab2_sync_32111860.tar  lab3_fuse
```

✓ 레포트 제출

해당 수업 시간에 교수님께 제출.