Operating System LAB2
Concurrent Data Structure

Ⅰ. 공통 과제 설명

Ⅱ. Queue

Ⅲ. Binary Search Tree

Ⅰ. 공통 과제 설명

1. 과제 목표

(1) Concurrent Data Structure (Binary Search Tree, Queue)를 구현한다.

- Binary Search Tree 혹은 Queue 중 하나를 선택하여, 아래 3가지 버전으로 구현한다.
  1) Without Lock, 2) Coarse-grained Lock, 3) Fine-grained Lock

(2) 다양한 Workload를 분석하고, 이에 따른 결과를 분석한다.

(3) Google Test Framework을 경험한다.

2. 과제 배경

멀티스레드 환경에서는 여러 스레드가 공유 자원에 동시에 접근할 수 있기 때문에, 경쟁 상태(Race Condition)가 발생할 수 있습니다. 이러한 문제를 해결하기 위해 개발자들은 락(Lock)을 사용하여 공유 자원에 대한 접근을 제어합니다. 그러나 락의 사용 방식에 따라 성능이 크게 달라질 수 있습니다.

락을 사용하는 방식은 크게 두 가지로 나눌 수 있습니다: Coarse-grained Locking과 Fine-grained Locking입니다. Coarse-grained Locking은 임계 영역(Critical Section)의 범위를 크게 설정하여 많은 양의 데이터를 한 번에 보호하는 방식입니다. 이 방식은 구현이 간단하지만, 동시성이 떨어질 수 있습니다. 반면에 Fine-grained Locking은 임계 영역을 세분화하여 각 부분마다 락을 사용하는 방식입니다. 이 방식은 동시성을 높일 수 있지만, 구현이 복잡해질 수 있습니다.

각 방식에는 장단점이 있으므로, 상황에 맞게 적절한 방식을 선택해야 합니다. 이번 과제에서는 Coarse-grained Locking과 Fine-grained Locking을 직접 구현해보고, 실행 시간을 측정하여 두 방식의 성능을 비교해 보겠습니다. 이를 통해 락의 사용 방식이 성능에 미치는 영향을 이해하고, 효율적인 동시성 제어 방법을 익힐 수 있을 것입니다.

3. 과제 환경 구성

(1) OS 및 Platform(HW)

- OS : Ubuntu 22.04.4 LTS
- Platform : Virtual Box, WSL, Native Linux

LAB0를 참고하여 구축한다.

(2) 소스코드 다운로드 및 실행

```
> git clone https://github.com/DKU-EmbeddedSystem-Lab/2024_DKU_OS.git
> cd 2024_DKU_OS/lab2
```

1) BST
> cd bst
> make
> ./test

(2) Queue
> cd queue
> make
> ./test

4. 보고서 구성

(1) 구현 설명 : 구현한 소스코드 설명

(2) 문제 (**자신의 선택한 자료구조의 문제만 풀이할 것**)

1) Queue

a. Fine-grained Queue가 비어있을 때, 서로 다른 스레드에서 동시에 ENQUEUE, DEQUEUE를 요청한다면, 수행 순서에 따라 수행 결과가 어떻게 달라질 수 있는가?

b. Multi-thread로 Queue를 접근할 때, request의 수행 순서를 일정하게 보장하는 방법을 제시하고, 설명하라.

2) BST

a. Fine-grained BST에서 동일한 키에 대해 서로 다른 스레드에서 동시에 INSERT, DELETE를 요청한다면, 수행 순서에 따라 수행 결과가 어떻게 달라질 수 있는가?

b. Multi-thread로 BST를 접근할 때, request의 수행 순서를 일정하게 보장하는 방법을 제시하고, 설명하라.

(3) Discussion

1) Workload별 성능(Execution Time) 분석

a. Lock Type(Non, Coarse, Fine)과 Thread 수에 따른 실행시간 비교 분석 (1-2가지)

b. 실행시간 분석 시, 과제 환경의 CPU/VirtualBox 코어 개수를 고려할 것

c. 그래프를 작성하는 것을 추천

2) 과제를 하면서 새롭게 배운 점, 어려웠던 점, 더 공부하고 싶은 점

5. 과제 제출

(1) 과제 제출 링크 (구글 폼) : https://forms.gle/d1T9kt1xeg5Dogk59

(2) 과제 제출 기한 : **24년 5월 13일 월요일 23:59:59**

- 재제출 시 가장 마지막에 제출한 과제를 기준으로 채점

(3) 과제 제출 목록

1) 보고서

- 파일명 : os_lab2_학번_이름.pdf

2) 소스코드

A. BST

- bst_impl.h_학번_이름.h
- bst_impl.cpp_학번_이름.h

B. Queue

- queue_impl.h_학번_이름.h
- queue_impl.cpp_학번_이름.h

- 소스코드 파일 상단에 학번과 이름 주석으로 반드시 작성

6. 채점

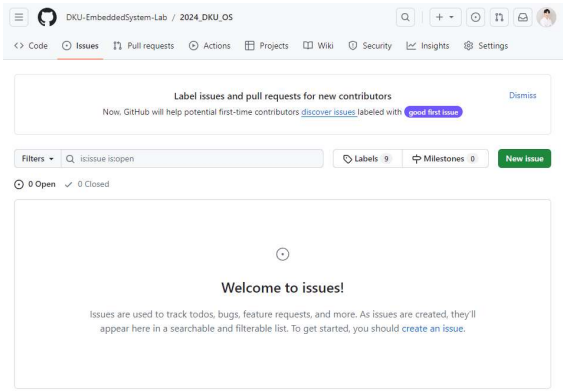
(1) 배점

구분	세부사항		배점	총점
구현	Queue	Queue	10	50
		Coarse Queue	20	
		Fine Queue	20	
	BST	BST	20	70
		Coarse BST	25	
		Fine BST	25	
보고서	구현 설명		10	30
	문제		5	
	Discussion		10	
	형식 점수		5	
- BST/Queue 중 1개를 선택하여 과제 수행 - 구현은 Google Test를 통해, 모두 자동 채점함				

(2) 감정사항

- 지각 제출 시, 하루에 10% 감점
- 소스코드를 텍스트/이미지 등 빌드 불가 파일로 제출 시 → 구현+형식 점수 (0/70점)
- 인적사항 주석 미 작성, 파일명/형식 미 준수 → 형식 점수 (0/10점)
- 코드 표절 시, 관련 학생 모두 일괄 0점 처리 (Moss/Codequiry Program 검사 예정)
  - if/else문 순서 변경, 함수 위치 변경, 변수 명 변경 모두 표절에 해당함
  - GPT 사용가능, 하지만 이로 인한 표절은 모두 제출자 책임

7. 질문



(https://github.com/DKU-EmbeddedSystem-Lab/2024_DKU_OS/issues)

과제 관련한 질문은 기본적으로 Github Issue를 통해 질문하시길 바랍니다. 이메일로 질문 드려도 답변 드리지 않습니다. 단, 자신의 코드 혹은 과제의 정답을 제시하며 질문해야 하는 경우, 반드시 이메일(mgchoi@dankook.ac.kr)로 질문하시길 바랍니다.

Ⅱ. Queue

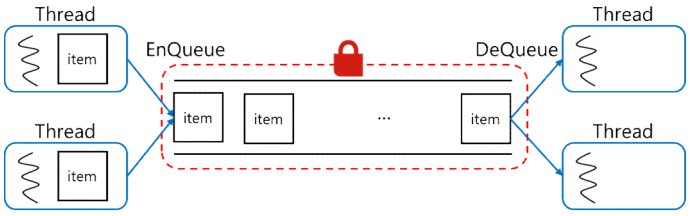
1. Concurrent Queue

큐에서 여러 스레드가 동시에 삽입(enqueue)과 삭제(dequeue) 연산을 수행할 때, 경쟁 상태(Race Condition)가 발생하여 정상적인 동작이 보장되지 않을 수 있습니다. 이러한 문제는 생산자-소비자(Producer-Consumer) 패턴에서 자주 발생합니다.

생산자 스레드가 큐에 새로운 요소를 추가하는 동안, 소비자 스레드가 해당 요소를 삭제한다면 문제가 발생할 수 있습니다. 또한, 두 개 이상의 스레드가 동시에 큐의 front나 rear를 변경하려고 한다면 일관성 문제가 발생할 수 있습니다. 경쟁 상태를 방지하고 큐의 각 연산이 안전하게 수행되도록 보장하기 위해, Coarse-grained Lock과 Fine-grained Lock 방식을 사용할 수 있습니다.

(1) Coarse-grained Lock을 사용한 Queue

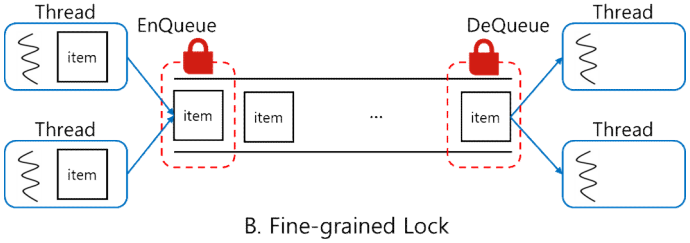
Coarse-grained Lock 방식에서는 큐에 대한 삽입(Enqueue) 및 삭제(Dequeue) 연산을 수행할 때 전체 큐를 임계 영역으로 간주합니다. 이는 모든 삽입 및 삭제 연산이 큐 전체에 대한 독점적인 접근 권한을 요구함을 의미합니다. 예를 들어, 한 스레드가 삽입 작업을 수행하는 동안 다른 모든 스레드는 해당 큐에 대해 어떠한 연산도 수행할 수 없으며, 해당 스레드의 작업이 완료될 때까지 대기해야 합니다. 이 방식은 구현이 간단하며 동시성 문제를 명확히 해결할 수 있지만, 동시성의 수준이 낮아지므로 시스템의 전반적인 성능이 저하될 수 있습니다.



A. Coarse-grained Lock

(2) Fine-grained Lock을 사용한 Queue

Fine-grained Lock 방식에서는 큐의 개별 요소나 구조에 락을 적용하여 세밀한 동시성 제어를 수행합니다. 예를 들어 큐의 머리(head)와 꼬리(rear/tail)에 각각 별도의 락을 적용할 수 있습니다. 삽입 연산은 주로 꼬리에서 이루어지므로 꼬리 부분에 락을 걸고, 삭제 연산은 머리에서 이루어지므로 머리 부분에 락을 걸어 두 연산이 독립적으로 수행될 수 있도록 합니다. 이 접근 방식은 동시성 수준을 높이지만, 데드락과 같은 추가적인 복잡한 문제를 관리해야 할 필요가 있습니다.



B. Fine-grained Lock

각 방식은 상황에 따라 선택되어야 하며, 특히 큐의 사용 패턴과 성능 요구 사항을 고려하여 적절한 방식을 채택하는 것이 중요합니다. Coarse-grained Lock은 보다 단순하고 안전한 동시성 제어를 제공하지만, 성능 저하를 초래할 수 있습니다. 반면, Fine-grained Lock은 성능을 최적화할 수 있지만 구현의 복잡성과 오류 가능성이 높아집니다.

2. 구현 과제

(1) 문제

- “queue_impl.h”와 “queue_impl.cpp”의 Queue, CoarseQueue, FineQueue 클래스를 구현한다.
- 각 클래스의 “enqueue”, “dequeue”, “empty“ 함수는 ”queue.h“의 설명을 참고한다.
- “queue_impl.h”와 “queue_impl.cpp”파일은 ”queue.h“외 다른 헤더파일을 참조할 수 없다. (**다른 파일 및 라이브러리 참조 시, LAB2 전체 0점 처리**)
- “queue_impl.h”와 “queue_impl.cpp”외 다른 파일은 수정, 추가, 제출이 불가하다.
- 반드시 C++로 구현해야 한다.

(2) 입력

1) Workload Type

- ENQ_THEN_DEQ : workload num 만큼 enqueue 한 뒤, 큐가 빌 때까지 dequeue한다.
- ENQUEUE_DEQUEUE : workload num 만큼 en/dequeue 한 뒤, 큐가 빌 때까지 dequeue한다.

2) Request Num : 500,000/1,000,000

3) Number of Threads : 1/2/4/8

(3) 채점

1) 학생이 제출한 ““queue_impl.h”와 “queue_impl.cpp”는 Makefile를 통해, 다른 소스코드와 “test” 프로그램으로 build 되고 정상적으로 수행되어야 한다.

2) test workload를 학생이 구현한 queue로 수행한다. 동일한 workload를 std::queue로 single-thread로 수행한다. 이후 각 queue가 dequeue한 모든 data를 정렬한 후, 비교하여 동일한 지 비교한다. (자세한 내용은 test_util.h, test_util.cpp를 참고.)

3. 소스 코드 구조

*** 모든 과제 소스 코드에 대한 설명은 주석으로 모두 자세히 작성하였으니, 주석을 참고하시길 바랍니다.**

(1) Queue Cass Class 구조

```
/// @brief Queue의 부모 클래스
class DefaultQueue {
    public:
        /**
         * @brief 큐 back에 key-value pair를 삽입한다.
         *
         * @details 큐가 가득 찼다면, 공간을 새로 할당하여 삽입한다.
         *
         * @param key
         * @param value
         */
        virtual void enqueue(int key, int value) = 0;

        /**
         * @brief 큐 front에 존재하는 key-value pair를 반환하고, 이 큐에서 삭제한다.
         *
         * @details 큐 front가 비었다면, {-1,-1}를 반환한다.
         *
         * @return std::pair<int, int>
         */
        virtual std::pair<int, int> dequeue() = 0;

        /**
         * @brief 큐가 비었다면 true, 아니라면 false를 반환한다.
         *
         * @return true
         * @return false
         */
        virtual bool empty() = 0;
};
```

상속

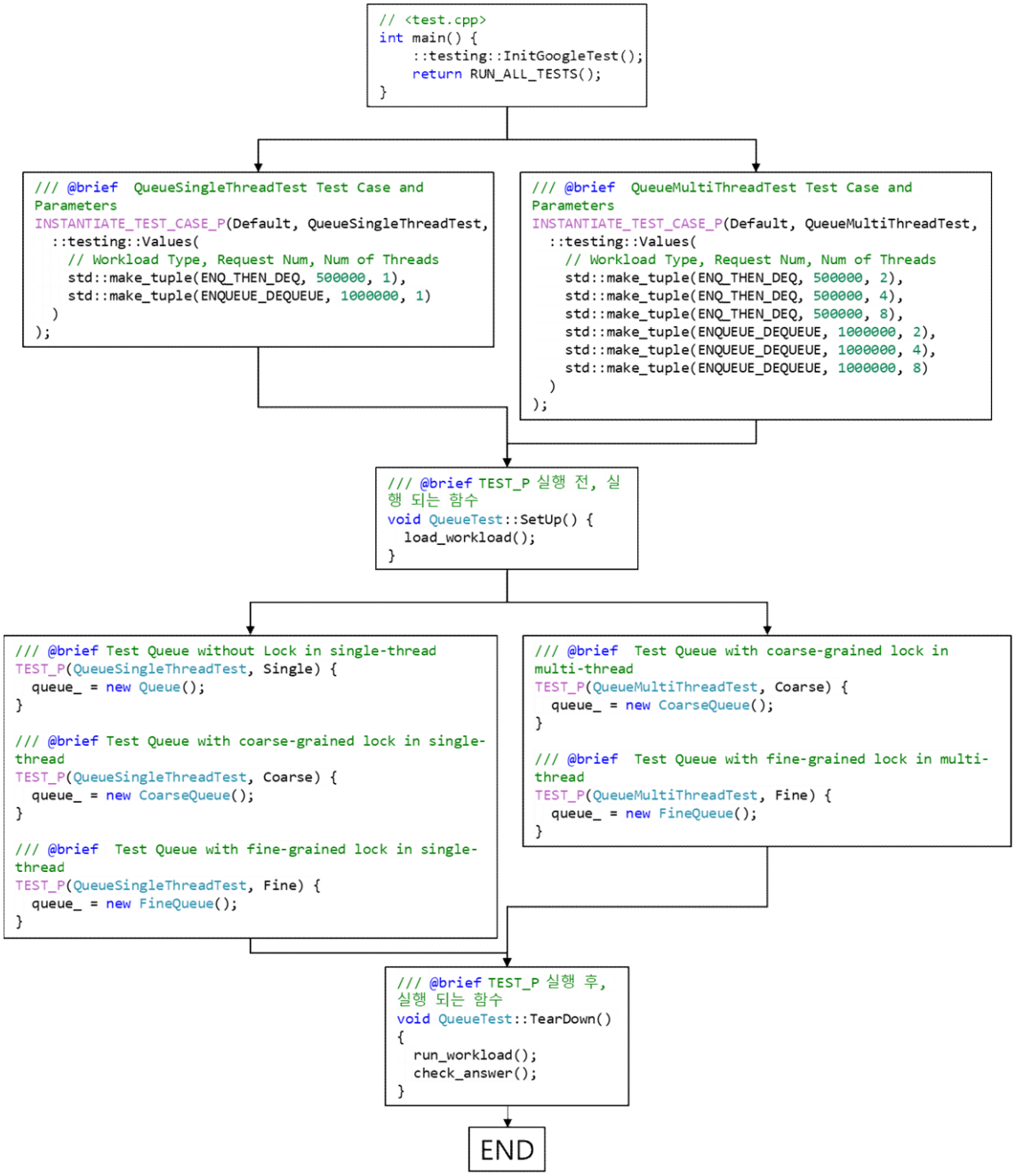
```
/**
 * @brief Queue without lock
 * DefaultQueue의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class Queue : public DefaultQueue {
    private:
        // 멤버 변수 추가 선언 가능
        std::queue<std::pair<int, int>> queue;
    public:
        // 멤버 함수 추가 선언 가능
        void enqueue (int key, int value) override;
        std::pair<int, int> dequeue () override;
        bool empty() override;
};
```

```
/**
 * @brief Queue with coarse-grained lock
 * Queue 전체를 critical section으로 가정하여, 하나의 lock으로 이를 관리한다.
 * DefaultQueue의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class CoarseQueue : public DefaultQueue {
    private:
        // 멤버 변수 추가 선언 가능
        std::queue<std::pair<int, int>> queue;
        pthread_mutex_t mutex_lock;
    public:
        // 멤버 함수 추가 선언 가능
        CoarseQueue (){
            pthread_mutex_init(&mutex_lock, NULL);
        }
        void enqueue (int key, int value) override;
        std::pair<int, int> dequeue () override;
        bool empty() override;
};
```

```
/**
 * @brief Queue with fine-grained lock
 * Queue 내부의 critical section을 개별적으로 lock으로 관리한다.
 * DefaultQueue의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class FineQueue : public DefaultQueue {
    private:
        // 멤버 변수 추가 선언 가능

    public:
        // 멤버 함수 추가 선언 가능
        void enqueue (int key, int value) override;
        std::pair<int, int> dequeue () override;
        bool empty() override;
};
```

(2) Test Code Flow



(3) Test Cass 구조

```
// 부모 테스트 클래스
class QueueTest : public ::testing::TestWithParam<std::tuple<WL_TYPE, int, int>> {
 protected:
  // 테스트 워크로드
  Req* workload_;
  // 테스트할 Queue 포인터
  DefaultQueue* queue_;
  // 각 스레드용 dequeue한 데이터를 저장하는 벡터
  std::vector<std::pair<int, int>>* th_vec_arr_;
  // 정답용 std::map <키, 값>
  std::queue<std::pair<int, int>> ans_queue_;
  // 정답용 std::vector <키, 값>
  std::vector<std::pair<int,int>> ans_vec_;
  // workload type
  WL_TYPE workload_type_ = std::get<0>(GetParam());
  // workload 총 횟수
  int workload_num_ = std::get<1>(GetParam());
  // workload thread 수
  int thread_num_ = std::get<2>(GetParam());

 public:
  // 생성자
  QueueTest (){
    // 스레드 수 만큼 벡터를 동적 할당한다.
    th_vec_arr_ = new std::vector<std::pair<int, int>>[thread_num_];
  }

  /// @brief TEST_P 실행 전, 실행 되는 함수
  void SetUp() override;

  /// @brief TEST_P 실행 후, 실행 되는 함수
  void TearDown() override;

  /**
   * @brief 워크로드를 불러오는 함수
   * Workload의 type과 num을 바탕으로, request을 생성한다.
   * Request의 key와 value는 1 ~ 100,000 사이의 랜덤한 값을 가진다.
   * 생성된 request들은 workload_에 저장된다.
   */
  void load_workload();

  /**
   * @brief 워크로드를 실행하는 함수
   * Workload를 스레드 수 만큼 분할하고, 각 스레드에 할당한다.
   * 스레드와 스레드 인자를 생성하고, 실행하고, 조인한다.
   */
  void run_workload();

  /**
   * @brief 테스트 케이스의 수행결과를 확인하는 함수
   * std::queue를 통해 single thread로 workload를 수행한다
   * std::queue가 dequeue한 data를 ans_vec_에 저장한 뒤 정렬한다.
   * 각 스레드의 vector를 합병하여 정렬하고, ans_vec_과 내부 요소가 모두 동일한지 확인한다.
   */
  void check_answer();

  // 소멸자
  ~QueueTest (){
    // 스레드 별 할당한 벡터 해제
    free(th_vec_arr_);
  }
};
```

상속

```
// Test Queue with Single Thread
class QueueSingleThreadTest : public
QueueTest {};
```

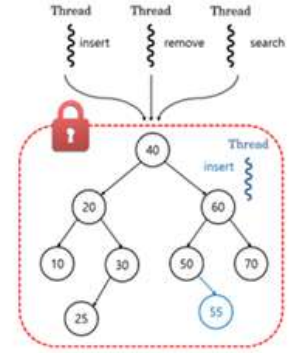
```
// Test Queue with Multi Thread
class QueueMultiThreadTest : public
QueueTest {};
```

Ⅲ. Binary Search Tree

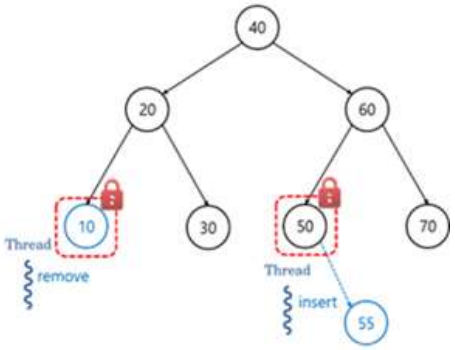
1. Concurrent Binary Search Tree

이진 탐색 트리(Binary Search Tree, BST)에서 여러 스레드가 동시에 삽입(insert), 삭제(remove), 검색(search) 등의 연산을 수행할 때, 경쟁 상태(Race Condition)가 발생하여 정상적인 동작이 보장되지 않을 수 있습니다. 이러한 문제는 생산자-소비자(Producer-Consumer) 패턴에서 자주 발생합니다.

생산자 스레드가 BST에 새로운 노드를 추가하는 동안, 소비자 스레드가 해당 경로의 특정 노드를 삭제한다면 문제가 발생할 수 있습니다. 또한, 읽기 스레드(Reader Thread)가 특정 노드의 데이터를 읽는 동안 쓰기 스레드(Writer Thread)가 해당 노드의 값을 변경한다면 일관성 문제가 발생할 수 있습니다. 경쟁 상태를 방지하고 BST의 각 연산이 안전하게 수행되도록 보장하기 위해, Coarse-grained Lock과 Fine-grained Lock 방식을 사용할 수 있습니다.



A. Coarse-grained Lock 방식



B. Fine-grained Lock 방식

(1) Coarse-grained Lock을 사용한 BST

Coarse-grained Lock 방식에서는 BST에 대한 삽입, 삭제, 검색 등의 연산 수행 시 트리 전체를 임계 영역(Critical Section)으로 간주하고, 해당 연산이 완료될 때까지 다른 스레드의 접근을 차단합니다. 아래 그림은 이 방식의 예시로, 현재 삽입 연산을 수행 중인 스레드가 트리를 독점하고 있으므로 다른 스레드는 대기해야 합니다.

(2) Fine-grained Lock을 사용한 BST

Fine-grained Lock 방식에서는 BST의 각 노드별로 임계 영역을 설정하여, 연산 수행 시 접근하는 노드에 대해서만 다른 연산의 접근을 제한합니다. 아래 그림은 이 방식의 예시로, 각 노드마다 락을 사용하여 세밀한 동시성 제어를 수행합니다.

Coarse-grained Lock과 Fine-grained Lock 방식은 각각 장단점이 있습니다. Coarse-grained Lock은 구현이 간단하지만 동시성이 떨어지는 반면, Fine-grained Lock은 동시성을 높일 수 있지만 구현이 복잡해질 수 있습니다. 상황에 맞는 적절한 방식을 선택하여 사용해야 효율적인 동시성 제어가 가능합니다.

2. 구현 과제

(1) 문제

- “bst_impl.h”와 “bst_impl.cpp”의 BST, CoarseBST, FineBST 클래스를 구현한다.
- 각 클래스의 “insert”, “lookup”, “remove“, ”traversal“ 함수는 ”bst.h“의 설명을 참고한다.
- “bst_impl.h”와 “bst_impl.cpp”파일은 ”bst.h“외 다른 헤더파일을 참조할 수 없다. (**다른 파일 및 라이브러리 참조 시, LAB2 전체 0점 처리**)
- “bst_impl.h”와 “bst_impl.cpp”외 다른 파일은 수정, 추가, 제출이 불가하다.
- 반드시 C++로 구현해야 한다.

(2) 입력

1) Workload Type

- INSERT_ONLY : workload num 만큼 insert한 후, single-thread로 중위 순회한다.
- INSERT_LOOKUP : workload num 만큼 insert(50%). lookup(50%)한 후, single-thread로 중위 순회한다.
- INSERT_LOOKUP_DELETE : workload num 만큼 insert(60%), lookup(20%), delete(20%)한다. Single-thread인 경우는 중위 순회를 수행하나, multi-thread인 경우 중위 순회를 수행하지 않는다. (**Single-thread를 통과하지 못하는 경우, multi-thread 또한 통과하지 못하는 것으로 채점**)

2) Request Num : 1,000,000

3) Number of Threads : 1/2/4/8

(3) 채점

1) 학생이 제출한 “bst_impl.h”와 “bst_impl.cpp”는 Makefile를 통해, 다른 소스코드와 “test” 프로그램으로 build 되어야 한다.

2) “test” 프로그램으로 test workload를 학생이 구현한 bst로 수행한다. 동일한 workload를 std::map으로 single-thread로 수행한다. 이후 두 중위 순회의 결과가 동일한 지 비교한다. (자세한 내용은 test_util.h, test_util.cpp를 참고.)

3. 소스 코드 구조

*** 모든 과제 소스 코드에 대한 설명은 주석으로 모두 자세히 작성하였으니, 주석을 참고하시길 바랍니다.**

(1) BST Cass Class 구조

```cpp
/// @brief Key, Value, (Update) Count 구조체
struct KVC {
    int key;
    int value;
    int upd_cnt;
};
```

```cpp
/// @brief BST 노드 구조체
struct Node{
    // 키
    int key;
    // 밸류
    int value;
    // 업데이트 횟수
    int upd_cnt;
    // left child 노드
    Node* left;
    // right child 노드
    Node* right;
};
```

상속

```cpp
/// @brief FineBST는 FineNode를 정의하여 사용하길 권장한다.
struct FineNode : public Node {
    // 멤버 변수 추가 가능
};
```

```cpp
/// @brief BST 부모 클래스.
class DefaultBST {
    protected:
        // traverse시, 인자로 받은 KVC 배열 포인터 저장
        KVC* traverse_arr_;
        // traverse시, KVC 배열에 data를 삽입할 위치 계산
        int arr_index_;

    public:
        /**
         * @brief 삽입 함수 : 인자로 받은 key와 value를 저장한다
         * 이미 key가 존재하는 경우, 기존 value에 새로운 value를 더하고 update count를 1 증가시킨다.
         *
         * @param key
         * @param value
         */
        virtual void insert(int key, int value) = 0;

        /**
         * @brief 탐색 함수 : 인자로 받은 key의 value를 반환한다.
         * BST에 key가 존재하지 않을 경우, 0을 반환한다.
         *
         * @param key
         * @return int
         */
        virtual int lookup(int key) = 0;

        /**
         * @brief 삭제 함수 : 인자로 받은 key가 저장된 노드를 삭제한다.
         *
         * @param key
         */
        virtual void remove(int key) = 0;

        /**
         * @brief 순회 함수 : 중위 순회하여, KVC 배열에 BST에 존재하는 모든 데이터를 저장한다.
         * KVC 배열에 key, value, upd_cnt가 key의 오름차순으로 저장한다.
         * KVC 배열은 함수 호출 전 이미 메모리에 할당된다.
         * KVC 배열의 길이는 그 길이는 std::map으로 single-thread로 수행한 것과 동일하다.
         * KVC 배열에 데이터 삽입 시, arr_index_를 증감 연산하며 활용하길 권장한다.
         * @param arr
         */
        virtual void traversal(KVC* arr) = 0;
};
```

상속

```cpp
/**
 * @brief BST without lock
 * DefaultBST의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class BST : public DefaultBST {
    private:
        // 멤버 변수 추가 선언 가능

    public:
        // 멤버 함수 추가 선언 가능
        void insert(int key, int value) override;
        int lookup(int key) override;
        void remove(int key) override;
        void traversal(KVC* arr) override;
};
```

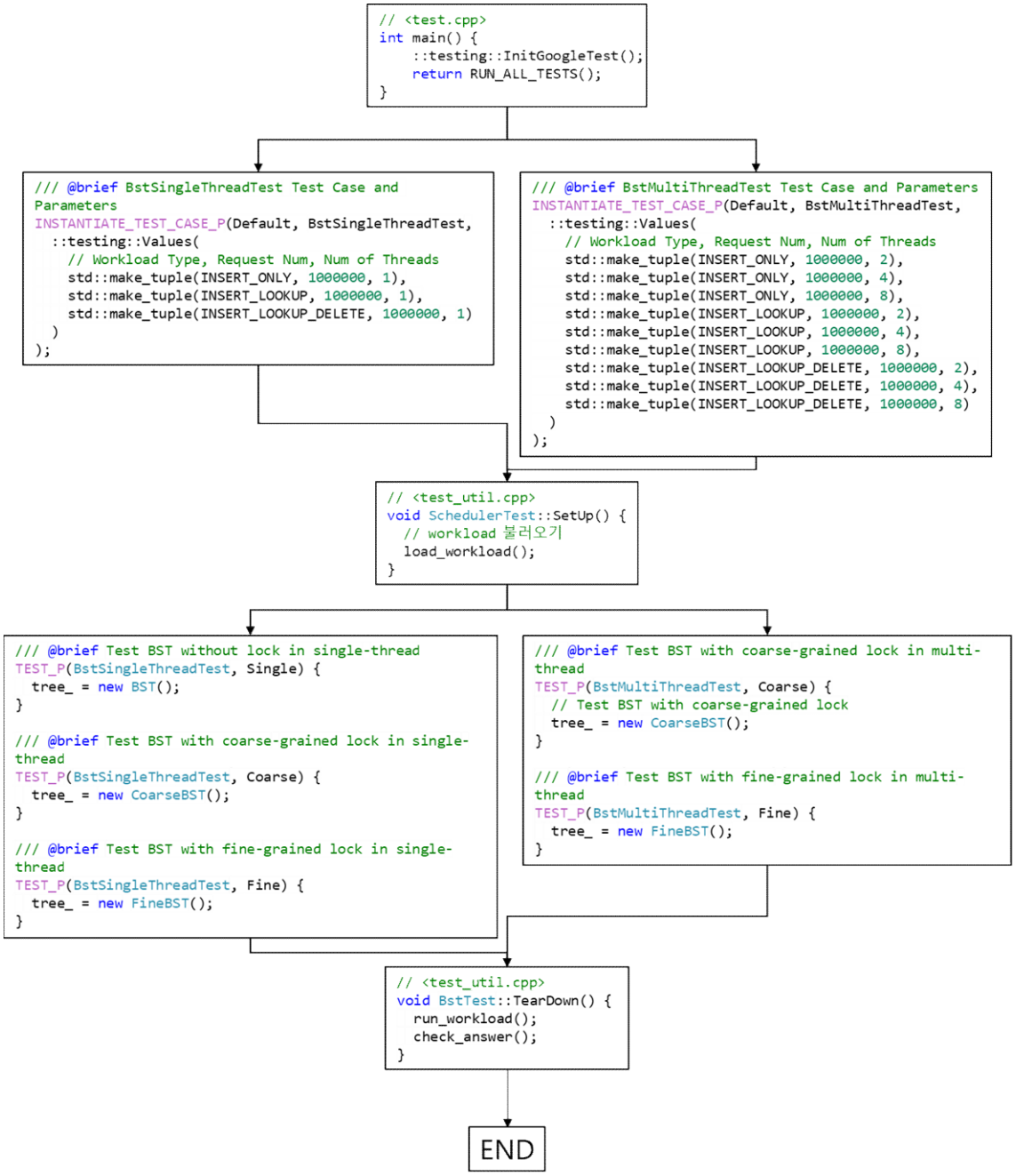
```cpp
/**
 * @brief BST with coarse-grained lock
 * BST 전체를 critical section으로 가정하여, 하나의 lock으로 이를 관리한다.
 * DefaultBST의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class CoarseBST : public DefaultBST {
    private:
        // 멤버 변수 추가 선언 가능

    public:
        // 멤버 함수 추가 선언 가능
        void insert(int key, int value) override;
        int lookup(int key) override;
        void remove(int key) override;
        void traversal(KVC* arr) override;
};
```

```cpp
/**
 * @brief BST with fine-grained lock
 * BST 내부의 critical section을 Node 단위로 lock으로 관리한다.
 * Node를 상속받은 FineNode를 정의하여 사용하길 권장한다.
 * DefaultBST의 함수를 오버라이드하여, 클래스를 완성한다.
 * 구현에 필요한 멤버 변수/함수를 추가하여 구현한다.
 */
class FineBST : public DefaultBST {
    private:
        // 멤버 변수 추가 선언 가능

    public:
        // 멤버 함수 추가 선언 가능
        void insert(int key, int value) override;
        int lookup(int key) override;
        void remove(int key) override;
        void traversal(KVC* arr) override;
};
```

(2) Test Code Flow



(3) Test Cass 구조

```cpp
/// @brief 부모 테스트 클래스
class BstTest : public ::testing::TestWithParam<std::tuple<WL_TYPE,
int, int>> {
 protected:
  // 테스트 워크로드
  Req* workload_;
  // 테스트할 BST 포인터
  DefaultBST* tree_;
  // 정답용 std::map <키, 값, 업데이트 횟수>
  std::map<int, std::pair<int, int>> map_;
  // workload type
  WL_TYPE workload_type_ = std::get<0>(GetParam());
  // workload 총 횟수
  int workload_num_ = std::get<1>(GetParam());
  // 스레드 개수
  int thread_num_ = std::get<2>(GetParam());

 public:
  /// @brief TEST_P 실행 전, 실행 되는 함수
  void SetUp() override;

  /// @brief TEST_P 실행 후, 실행 되는 함수
  void TearDown() override;

  /**
   * @brief 워크로드를 불러오는 함수
   * Workload의 type과 num을 바탕으로, request을 생성한다.
   * Request의 key와 value는 1 ~ 100,000 사이의 랜덤한 값을 가진다.
   * 생성된 request들은 workload_에 저장된다.
   */
  void load_workload();

  /**
   * @brief 워크로드를 실행하는 함수
   * Workload를 스레드 수 만큼 분할하고, 각 스레드에 할당한다.
   * 스레드와 스레드 인자를 생성하고, 실행하고, 조인한다.
   */
  void run_workload();

  /**
   * @brief 테스트 케이스의 수행결과를 확인하는 함수
   * std::map를 통해 single thread로 workload를 수행한다
   * BST와 std::map을 (중위)순회한 결과가 모두 동일한지 확인한다.
   */
  void check_answer();
};
```

상속

```cpp
/// @brief Test BST with Single Thread
class BstSingleThreadTest : public BstTest
{};
```

```cpp
/// @brief Test BST with Multi Thread
class BstMultiThreadTest : public BstTest {};
```