Мелкогранулярные блокировки: хэш-таблицы и списки

Во всех задачах этого домашнего задания нужно реализовать многопоточный контейнер с интерфейсом **множества**:

Операции:

- bool Insert(const T& element) - добавить элемент, если его нет во множестве, и вернуть true; в противном случае вернуть false
- bool Remove(const T& element) - удалить элемент, если он принадлежит множеству, и вернуть true; в противном случае вернуть false

Иначе говоря, мутирующие операции возвращают true, если контейнер был модифицирован, и false, если не был.

- bool Contains(const T& element) - проверить, принадлежит ли элемент множеству
- size_t Size() - вернуть мощность множества

Задача A. Striped Hash Set - 7 баллов

Напишите хэш-таблицу с цепочками, которая использует технику lock striping для параллельной работы нескольких операций и умеет автоматически расширяться по мере заполнения.

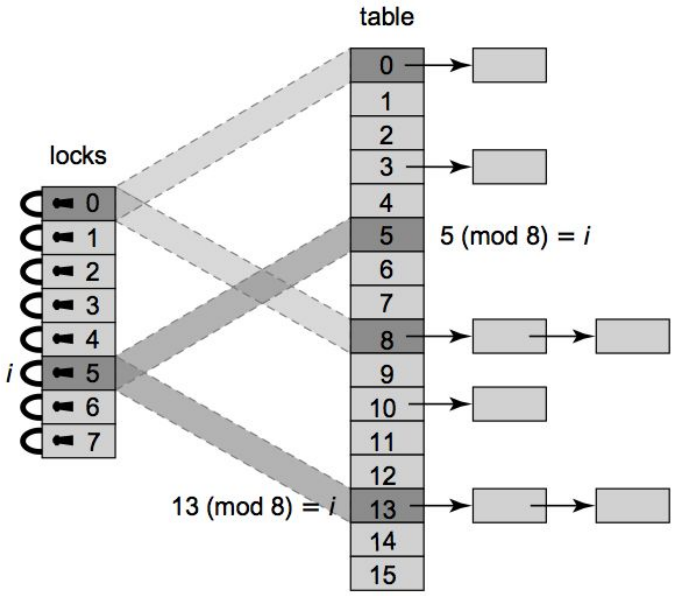
Lock striping:

Хэш-таблица использует массив мьютексов (локов) фиксированного размера, каждый из которых охраняет собственный набор корзин (этот набор будем называть stripe).

Вставки, удаления и поиски в разных страйпах должны работать без синхронизации друг с другом. Операции в пределах одного страйпа должны брать блокировку.

Правильная стратегия расширения хэш-таблицы гарантирует, что каждый элемент после вставки никогда не меняет свой страйп (т.е. навсегда остается привязанным к одному локу), хотя может перемещаться из одной корзины в другую в результате перехеширования.

Привязка к элемента к страйпу - ключевой инвариант, который позволяет потоку захватывать нужный лок, не зная текущее число корзин в хэш-таблице и расположение искомого элемента.



Страйпы в хэш-таблице не материальны, они не представлены никакой отдельной сущностью, они существуют только в виде отображения из корзин хэш-таблицы в локи.

При расширении таблицы поток должен захватить все мьютексы хэш-таблицы, чтобы получить монопольный доступ к внутренностям контейнера и выполнить перехеширование. Чтобы избежать взаимной блокировки, нужно соблюдать монотонность при захвате локов.

Сразу несколько потоков могут одновременно обнаружить, что хэш-таблица переполнилась, и запустить расширение. В такой ситуации хэш-таблица не должна расширяться несколько раз.

Указания по реализации:

В конструкторе StripedHashSet должен получать число локов (страйпов), назовем этот параметр concurrency_level. Если параметр равен 1, то страйпинг вырождается в один толстый мьютекс, который сериализует все операции над хэш-таблицей.

Изначально число мьютексов и число корзин не обязаны совпадать. Число корзин разумно сделать небольшим (несколько десятков).

Конструктор хэш-таблицы должен иметь опциональный параметр growth_factor - мультипликативный коэффициент расширения хэш-таблицы. При этом у него должно быть значение по-умолчанию.

Конструктор хэш-таблицы должен иметь опциональный параметр max_load_factor - пороговое значение для load factor (отношения числа элементов к числу корзин), при превышении которого таблица должна расширяться.

Узнайте, какие пороги / коэффициенты расширения по умолчанию используются в стандартных библиотеках разных языков программирования.

Сделайте контейнер шаблонным, параметр шаблона - тип элементов, которые будут храниться в хэш-таблице.

Для цепочек используйте контейнер std::forward_list<T> - реализацию односвязного списка из стандартной библиотеки. Для поиска в цепочке используйте стандартные алгоритмы - std::find.

В качестве хэш-функции используйте std::hash<T>

Для каждой операции считайте хэш-функцию ровно один раз. Вызов хэш-функции может быть достаточно дорогой операцией.

Для вычисления индекса корзины или страйпа/мьютекса реализуйте два отдельных метода: GetBucketIndex(const size_t hash_value) и GetStripeIndex(const size_t hash_value). Тогда не придется таскать модули по всему коду контейнера.

При расширении таблицы возникает необходимость поменять местами содержимое двух векторов корзин - старого и нового. Для этого можно воспользоваться методом swap у std::vector, который за O(1) перекидывает внутренние указатели на буферы данных, либо использовать move-семантику.

Вопросы на понимание:

Почему важна привязка элемента к страйпу? Приведите пример исполнения, в котором нарушение этого инварианта при расширении таблицы приводит к некорректным результатам операций.

Опишите сценарий, в котором несколько потоков могут одновременно перейти к расширению хэш-таблицы.

Что может пойти не так, если поток, который в ходе выполнения операции решает расширять таблицу, не будет отпускать мьютекс текущего страйпа?

Какие действия предотвращают возникновения взаимных блокировок?

При расширении хэш-таблицы потоку не обязательно захватывать все мьютексы, чтобы понять, что его опередили с расширением. Почему?

Задача A+. Reader/Writer Mutex для страйпов - 3 балла

Реализуйте R/W мьютекс с приоритетом для писателей и используйте его для доступа к страйпам хэш-таблицы:

Потоки, выполняющие поиск в хэш-таблице, будут брать читательскую блокировку на страйп, а вставки/удаления будут работать под эксклюзивной блокировкой страйпа.

Задача B. Оптимистичный список - 5 баллов

Напишите OptimisticLinkedSet - односвязный сортированный список на оптимистичных мелкогранулярных блокировках с двухфазным удалением.

Каждый узел списка должен быть защищен отдельным спинлоком. Поиск ребра, на котором нужно выполнить модификацию (Locate), должен выполняться без блокировок.

Двухфазное удаление (маркировка узла + перелинковка) позволяет быстро валидировать захваченное ребро списка в мутирующих операциях без повторного прохода по списку.

В списке поддерживается следующий **инвариант**: все непомеченные (белые) узлы находятся на главной ветке (master или trunk) списка.

Валидация выполняется после взятия локов на узлах, которые нашел Locate. Если после взятия локов узлы по прежнему образуют ребро и находятся на главной ветке списка, то никакие конкурирующие мутации это ребро уже не поломают.

Валидация проверяет два условия: узлы по-прежнему соседние (после завершения Locate и до взятия локов между ними могли вставить новые узлы) и оба узла по-прежнему находятся в главной ветке списка (в силу инварианта для этого достаточно проверить метки этих узлов).

Про memory management при оптимистичных блокировках:

Заметим, что поток, который отлинковал узел в операции Remove, не может освободить память этого узла, поскольку другие потоки в конкурирующих операциях могут в данный момент проходить через этот узел в операции Locate.

Если мы реализуем оптимистичный список в среде исполнения с автоматической сборкой мусора, то проблема освобождения памяти решается сама собой:

Удаленный узел уже не достижим из головы списка, так что новые операции в него прийти уже не могут, а уже бегущие операции рано или поздно пройдут через него, после этого узел будет подобран сборщиком мусора.

В языке без сборщика мусора жить гораздо сложнее. Умные указатели, использующие подсчет ссылок, не решают задачу, даже несмотря на атомарные инкременты/декременты.

Однопроходной Bump-Pointer Allocator:

В этой задаче мы будем использовать очень простой аллокатор, который при инициализации выделяет арену (буфер) большого размера и при каждой аллокации (нового узла) откусывает от этой арены маленький кусочек, двигая вперед указатель с помощью атомарного Fetch-And-Add. Выделенная память не освобождается и не используется повторно до момента разрушения аллокатора.

При использовании такого аллокатора потокам не нужно явно освобождать память извлеченных узлов, вся арена памяти будет освобождена разом при разрушении аллокатора.
