Lock-free LRU cache

Шикалов Андрей Артемович
ИСП РАН, 4 сентября 2024

01

Введение

Несколько слов про
lock-free и lru.

Введение

LRU (least recently used) - политика вытеснения из кэша, при которой из хранилища удаляются элементы, которые давно не использовались, освобождая место для новых элементов.

Lock-free - способ написания потокобезопасных структур данных, при котором падение или остановка одного из потоков не может привести к остановке других потоков.

Преимущества Lock-free

- **Гарантия прогресса (безопасность):** если какой-то поток завис, система продолжает работать, не дожидаясь его.
- **Отсутствие инверсии приоритета:** поток, выполняющий важную задачу не тратит время на ожидание, пока выполняется менее важная задача.
- **Производительность:** структуры без блокировок как правило более легковесные, так как не нагружают ОС системным вызовом futex и переключениями планировщика “вхолостую”.

02

Цель задачи

Требования к интерфейсу и реализации.

Цель задачи

- Реализовать lock-free структуру данных, хранящую пары ключ-значение.
- Количество элементов в ней ограничено фиксированным числом (capacity).
- Элементы вытесняются по принципу least recently used.
- Время хранения объектов ограничено - задается время жизни объектов (time-to-live или TTL), спустя которое они удаляются, даже если этого не требует политика вытеснения.
- Получившийся вариант должен быть производительнее, чем предыдущий (словарь, защищённый мьютексом).

Требования к API

- В шаблонных параметрах передаются тип ключа `K`, тип значения `V`, хэш-функция для ключа `Hash<K>`.
- В конструкторе передаются: размер `size_t capacity`, время жизни `chrono::duration ttl` и функция для генерации значения по-умолчанию из ключа `function<V(const K)> func`.
- Реализованы вставка по умолчанию `insert(const K)`, вставка `insert(const K, const V)` и создание элемента на месте `emplace(const K, Args… args)`.
- Обращение к элементу `operator[]` возвращает копию. Если элемент не найден, сконструировать элемент по-умолчанию с помощью переданной функции.

03

Обзор решений

Примеры реализации и доступные библиотеки.

Hashmap + iterable list

*псевдо-c++ код

```
list<K, V>                                dq_;
unordered_map<K, list<K, V>::iterator>    mp_;
```

```
void insert(K key, V value) {
    if (!mp_.contains(key) && dq_.size() == capacity_) {
        // Cache miss on full queue -> erase lru
        mp_.erase(dq_.front());
        dq_.erase_at(dq_.begin());
    }
    else {
        // Cache hit -> erase target
        dq_.erase_at(mp[key]);
        mp_.erase(key);
    }
    mp_.insert(key, dq_.insert(key, value));
}
```

Заводится очередь (dq_), задающая приоритет по порядку вставок и хэш-таблица (mp_) для быстрого доступа к итераторам очереди.

При промахе вытесняется последний элемент и вставляется новый. При попадании элемент удаляется и вставляется заново.

Для стандартной библиотеки сложность составляет O(1).

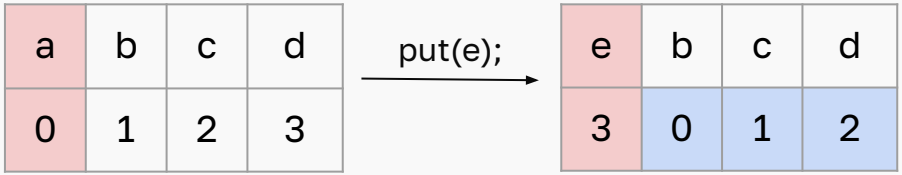
Пример:
github.com/userver-framework/userver/blob/develop/universal/include/userver/cache/impl/lru.hpp

LRU counters

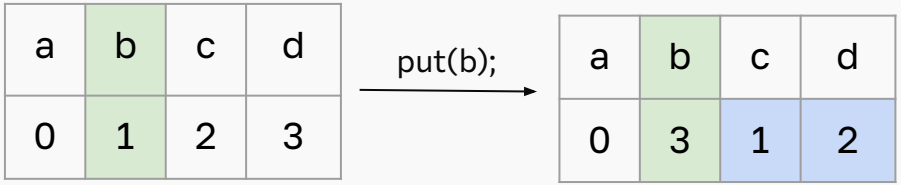
Заводится множество из N элементов, которым присваиваются индексы от 0 до N-1.

Least recently updated (LRU)	0
Most recently updated (MRU)	sizeof(set) - 1

При промахе элемент с индексом 0 заменяется на новый элемент под индексом N-1, а у всех остальных элементов индексы уменьшаются на 1.



При попадании, индекс элемента обновляется до N-1, а все индексы, большие, чем измененный, уменьшаются на 1.



Источник:
http://thebeardsage.com/lru-counter-implementation/

Доступные lock-free структуры

В библиотеке libcds (Concurrency Data Structures by khizhmax), используемой в проекте, есть некоторые подходящие структуры:

- **MichaelHashMap** - хэш-таблица на основе двусвязного списка.
- **IterableKVList** - итерируемый двусвязный список. С помощью методов begin() и end() можно получить первый элемент и служебный элемент в конце (идёт после последнего).
- **FeldmanHashMap** - хеш-таблица на основе многоуровневого массива. Организована как небинарное дерево поиска. Итерируема, доступ к итераторам организован аналогично через begin() и end().

04

Решение в теории

План архитектуры и основные идеи.

Проблемы с libcds

- **Итераторы нельзя передавать другим потокам:** для защиты от обращения по удаленному элементу используется система hazard pointer - каждый поток хранит приватный на запись массив с опасными указателями. Итераторы, например из IterableKVList, содержат в себе копию hazard pointer-а текущего потока. Поэтому конкретный итератор может использоваться только потоком, его создавшим. Значит, идея с хранением итераторов в хэш-таблице не реализуема.
- **Очереди не поддерживают итерирование назад:** это значит, что при поиске места для элемента либо на удаление, либо на вставку (в зависимости от выбора направления очереди) сложность будет составлять O(N), что медленно, даже по сравнению с решением с блокировками.

Решение

Идея - сделать очередь из хеш-таблицы.

Поскольку доступ к элементам FeldmanHashMap происходит достаточно быстро, вместо итераторов можно использовать индексы элементов по принципу “билетов” *.

Заводятся две атомарны величины `atomic<size_t> front` и `atomic<size_t> back`. При вставке элементу присваивается `back.fetch_add(1)`. При удалении итерируемся начиная с `front`.

*en.wikipedia.org/wiki/Ticket_lock

`Map<size_t,pair<K,V>> dq_`

0	1	2	3
[a,A]	[b,B]	[c,C]	[d,D]

`Map<K, size_t> mp_`

a	b	c	d
0	1	2	3

`lru.insert(b, E);`

0	2	3	4
[a,a]	[c,C]	[d,D]	[b,E]

a	c	d	b
0	2	3	4

05

Реализация

Написание кода на C++.

Фундамент класса

```
template<typename K, typename V, typename Hash = SafeHash<K>>
class LRU
{
public:
    V       operator[](const K& key);
    void    insert(const K& key, const V& value);
    void    expire();
private:
    size_t              capacity_;
    emplacer_func       emplacer_;
    std::atomic<size_t> size_;

    std::atomic<size_t> queue_front;
    std::atomic<size_t> queue_back;

    typedef std::pair<K, V> Node;

    cds::container::FeldmanHashMap<GarbageCollector, size_t, Node, container_traits>
            dq_;

    cds::container::FeldmanHashMap<GarbageCollector, K, size_t, map_traits>
            mp_;

    void    pop_front();
};
```

Удаление с конца

```
template<typename K, typename V, typename Hash, ExpirationPolicy Policy>
void
LRU<K, V, Hash, Policy>::pop_front()
{
    QueueIt begin = dq_.begin();

    if (begin == dq_.end()) return;

    auto begin_next = dq_.begin();
    ++begin_next;
    size_t begin_key = begin->first;

    while (!queue_front.compare_exchange_strong(begin_key, begin_next->first))
    {
        ++begin;
        ++begin_next;

        if (begin_next == dq_.end()) return;
    }

    if (begin != dq_.end())
    {
        mp_.erase(begin->second.first);
        dq_.erase(begin->first);
    }
} // LRU::pop_front
```

При вытеснении элемента принципиально сделать ровно столько удалений с конца, сколько было запросов на вытеснение. Ввиду того, что индексы могут идти не по порядку, мы итерируемся с начала и крутимся в CAS, сравнивая ключ начала с прочитанным ключом.

После прохождения через CAS каждый поток получает уникальный индекс на удаление.

Вставка

```cpp
template<typename K, typename V, typename Hash, ExpirationPolicy Policy>
void
LRU<K, V, Hash, Policy>::insert(const K& key, const V& value)
{
    bool need_pop = false;
    auto value_ptr = mp_.get(key); // pair<Key, size_t>*

    // key is not presented in cache
    if (value_ptr.empty())
    {
        if (need_pop = (size_.fetch_add(1) >= capacity_))
            size_.store(capacity_);
    }
    // key is presented in cache
    else
    {
        dq_.erase(value_ptr->second);
        mp_.erase(value_ptr->first);
    }

    size_t current_number = queue_back.fetch_add(1);
    dq_.insert(current_number + 1, Node(key, value));
    mp_.insert(key, current_number + 1);

    if (need_pop) pop_front();
} // LRU::insert
```

Реализуем так же, как в hashmap + list (слайд 9).

При переполнении присваиваем size_ = capacity_. Каждый поток, который выполнил вставку с переполнением получает локальный флажок на удаление элемента с конца.

Таким образом после выхода всех потоков из критической секции, размер очереди корректен.

TTL

```
struct LruIndex
{
    size_t      number;
    std::chrono::time_point<std::chrono::steady_clock>  time;

    // implicit cast to size_t
    operator size_t() const { return number; }
};
```

```
auto it = mp_.begin();
while (it != mp_.end())
{
    if (clock_t::now() - it->second.time >= time_to_live_)
    {
        dq_.erase(it->second.number);
        mp_.erase(it->first);
        size_--;
    }
    else
    {
        return;
    }
    ++it;
}
```

Вместо size_t используем для индексации структуру-пару с неявным приведением к size_t.

Итерируемся по всем элементами и проверяем, не закончилось ли их время жизни.

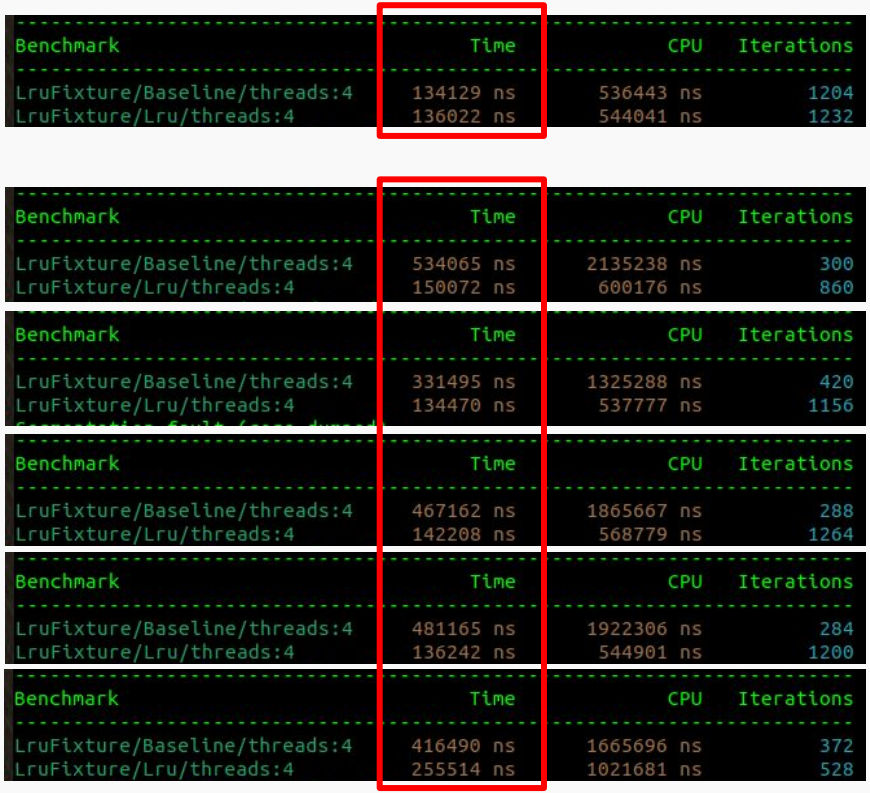
Примечание: планировка и запуск циклов просрочки элементов предоставляется пользователю, т.к. создание отдельного пула для каждой очереди и исполнение циклов на нём достаточно ресурсозатратно.

06

Результаты

Тесты производительности в сравнении с предыдущей версией.

Бенчмарк



Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	134129 ns	536443 ns	1204
LruFixture/Lru/threads:4	136022 ns	544041 ns	1232

Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	534065 ns	2135238 ns	300
LruFixture/Lru/threads:4	150072 ns	600176 ns	860

Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	331495 ns	1325288 ns	420
LruFixture/Lru/threads:4	134470 ns	537777 ns	1156

Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	467162 ns	1865667 ns	288
LruFixture/Lru/threads:4	142208 ns	568779 ns	1264

Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	481165 ns	1922306 ns	284
LruFixture/Lru/threads:4	136242 ns	544901 ns	1200

Benchmark	Time	CPU	Iterations
LruFixture/Baseline/threads:4	416490 ns	1665696 ns	372
LruFixture/Lru/threads:4	255514 ns	1021681 ns	528

- Удачное поведение планировщика (повезло встретить мало блокировок). Результат - одинаковая производительность.
- Стандартное поведение планировщика (в большинстве случаев). Результат - улучшение производительности в ~2-3 раза.

Lock-free LRU cache

Шикалов Андрей Артемович
ИСП РАН, 4 сентября 2024