Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)
Кафедра вычислительной техники

Выпускная квалификационная работа бакалавра

Реализация и оптимизация неблокирующих связанных списков

Выполнил студент группы 9305

Кашин Андрей Александрович

Научный руководитель: кандидат технических наук, доцент

Пазников Алексей Александрович

Санкт-Петербург

2023 г.

Цель и задачи

- Цель: исследование принципов работы неблокирующих структур и получение параметров для эффективного применения структур для конкретного случая.

Задачи:

- реализация неблокирующего списка Харриса;
- применение и сравнение производительности списков из библиотеки Libcds;
- Исследование методов оптимизаций на примере алгоритмов Libcds.

Актуальность темы и используемые технологии

- Сложность программного обеспечения постоянно растет, а значит повышаются требования к производительности, следовательно, необходимо применять многопоточность.
- Оптимальные параметры неблокирующих структур данных сложно определить без тестов.

- Используемые технологии: C++, Intel Vtune Profiler, Clang, Cmake.

Неблокирующие структуры данных

Неблокирующие структуры данных реализуют алгоритмы, которые позволяют использовать одни и те же разделяемые данные несколькими потоками одновременно без внешней синхронизации.

Виды:

- Без препятствий (англ. obstruction-free);
- Без блокировок (англ. lock-free);
- Без ожиданий (англ. wait-free).

Список Харриса

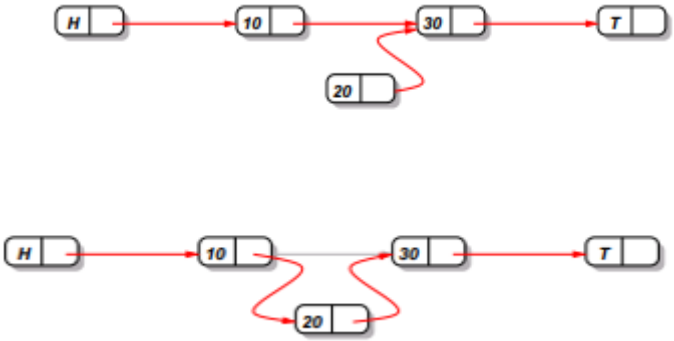
Harris' Non-blocking linked list – структура данных, которая является надстройкой над списком – множеством, т. н. List-based-set, поскольку является отсортированным, без повторов, односвязным списком.

Поддерживает операции:

- Insert;
- Find;
- Delete;
- Search.

Список Харриса, смысл алгоритма

```
bool MyInsert (key) {
    Node*new_node = new Node(key);
    Node*right_node, *left_node;
    do {
        right_node = MySearch (key, &left_node);
//поиск позиции для вставки
        if ((right_node != tail) && (right_node->key ==key))
//если не найдено - выход
            return false;
//присваивание ссылки во вставляемый элемент
        new_node->next = right_node;
        if (CAS(&(left_node->next), right_node, new_node))
//присваивание ссылки в левый элемент.
return true;
    } while (true);
}
```



Операция вставки в список реализовывается просто – применяется атомарная операция по отношению к полю next элемента слева от вставляемого элемента.

Если операция CAS по присваиванию поля next левого элемента списка в данный проход цикла не выполнилась, цикл идет заново, пока выполнение не будет успешным.

Список Харриса, смысл алгоритма

```
private Node *List::search (KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;
    search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;
// 1: находим левый и правый элементы, правый является искомым
if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail) break;
            t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key<search_key));
        right_node = t;
/* 2: проверяем, что элементы являются соседями */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;
/* 3: удаляем физически*/
        if (CAS (&(left_node.next), left_node_next, right_node))
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again;
            else
                return right_node;
    }
```

Операция поиска находит по ключу элемент, ключ которого больше или равен искомому. Он является правым элементом в поиске. Далее находится первый элемент слева, т.е. ключ у которого меньше, чем у правого элемента.

Обязательные условия – оба элемента не должны быть маркированы, должны находиться рядом, левый элемент меньше ключа поиска, правый – больше.

Список Харриса, смысл алгоритма

```
public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;
    do {
        right_node = search (search_key, &left_node);
            if ((right_node == tail) || (right_node.key != search_key))
//элемент не найден
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS (&(right_node.next),
                 right_node_next, get_marked_reference  (right node next)))
//логическое удаление
            break;
    } while (true);
    if (!CAS (&(left_node.next), right_node, right_node_next)
//физическое удаление
        right_node = search (right_node.key, &left_node);
    return true;
}
```

Функция удаления пытается удалить элемент с заданным ключем. Удаление происходит в два этапа – логическое и физическое.

Характеристики тестов

Выполнены на компьютере, имеющем следующие характеристики:

- Intel Core i7-8565U @ 1.80ГГц, 4 ядра, 8 логических процессоров
- оперативная память: 8 ГБ;
- дисковое пространство: 960 ГБ.

Порядок проведения:

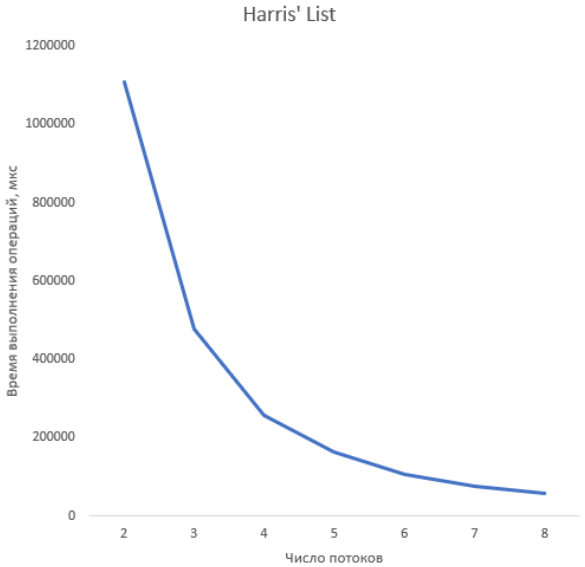
1) Перед каждым тестом все логические процессоры совершают «пустую» работу.
2) 3200 одновременных операций вставки и удаления случайного элемента в каждом потоке.
3) От 1 до 8 потоков.
4) Все тесты проводились по 5 раз, в результате получалось среднее значение времени.

Отличие между экспериментами заключается в структурах и их параметрах.

Список Харриса тест, результаты и производительность

Тип ключа элемента – int. 3200 операций.

```
Run:  selfmadeHarris
C:\Users\akash\CLionProjects\selfmadeHarris\cmake-build-debug
threads number = 1 duration = 4728530 microsec
threads number = 2 duration = 1106979 microsec
threads number = 3 duration = 476624 microsec
threads number = 4 duration = 255799 microsec
threads number = 5 duration = 162381 microsec
threads number = 6 duration = 106434 microsec
threads number = 7 duration = 77064 microsec
threads number = 8 duration = 56989 microsec

Process finished with exit code 0
```



Введение в Libcds

Libcds – библиотека неблокирующих структур данных, разработанная Максимом Хижинским, ведущим инженером компании “VasExperts” в 2009 году.

Структура программы с Libcds:

```
#include <cds/init.h>  //cds::Initialize и cds::Terminate
#include <cds/gc/hp.h> //cds::gc::HP (Hazard Pointer)

int main(int argc, char** argv)
{
   // Инициализируем libcds
   cds::Initialize() ;
   {
      // Инициализируем Hazard Pointer синглтон
      cds::gc::HP hpGC ;
      cds::threading::Manager::attachThread() ;

      // Всё, libcds готова к использованию
      // Далее располагается ваш код
      ...
   }
   // Завершаем libcds
   cds::Terminate() ;
}
```

Каждая структура имеет набор параметров, определяющих поведение структуры данных и алгоритм её работы.

Рассматриваемые параметры:

- Safe-Memory-Reclamation схема
- Back-off стратегия

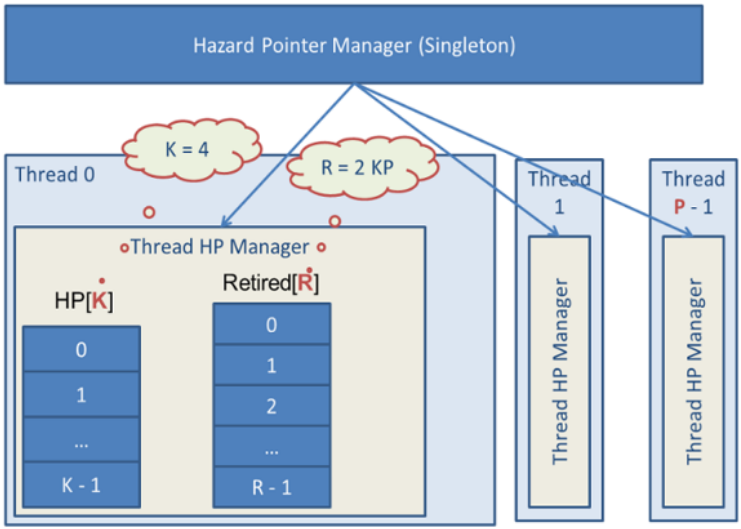
Описание параметров структур

SMR схемы:

- Hazard Pointers (HP);
- Dynamic Hazard Pointers (DHP);
- User-spaced Read-Copy-Update (URCU).

Hazard Pointer

У каждого потока есть локальный массив указателей HP. Прежде, чем работать с указателем на элемент, поток помещает его в этот массив, при этом заполнять массив может только поток-создатель, а читать его могут все потоки. В среднем размер массива для операций с разными структурами не превышает 5 указателей.



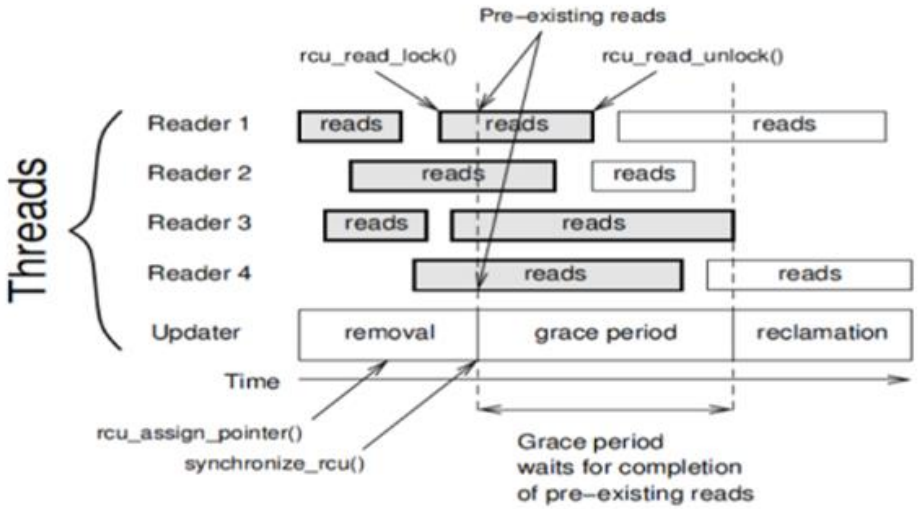
При удалении поток помещает нужный указатель в массив Retired. Как только массив Retired полностью заполняется, вызывается Scan(). В этой функции сначала со всех потоков собираются HP, не равные нулю, потом из массива Retired удаляются указатели, не являющиеся HP, т.е. которые можно безопасно удалить, т.к. с ними не работает ни один поток.

Сравнение HP и DHP

Особенность	HP	DHP
Максимальное число HP указателей на поток	Ограничено, указывается во время инициализации	Изменяется, память выделяется, когда необходимо
Максимальное число элементов на удаление	Ограничено, указывается во время инициализации	Ограничено, изменяется, зависит от числа потоков и размера массива HP в каждом потоке
Поддерживаемое число потоков	Ограничено, верхний порог указывается при инициализации.	Нет ограничений

User-spaced Read-Copy-Update

- URCU instant;
- URCU buffered;
- URCU threaded.



Back-off стратегия

Смысл back-off в том, что в ситуации, когда поток не может выполнить атомарную операцию, он приостанавливает свое действие.

В случае большой нагрузки на элемент только один поток сможет выполнить необходимую операцию, остальные будут впустую использовать ресурсы.

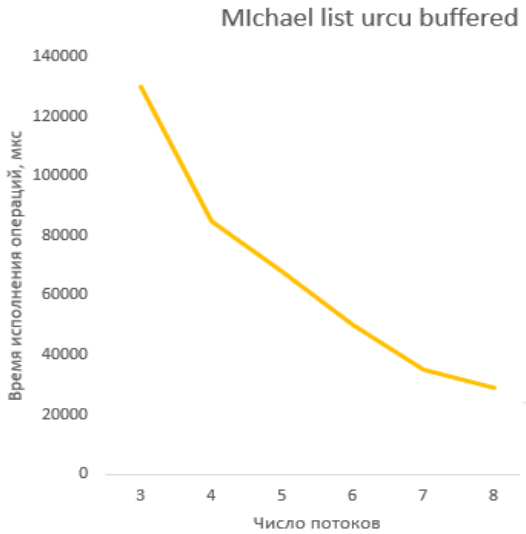
Back-off стратегии выгодны, поскольку обычно время на смену контекста гораздо больше, чем время на выполнение CAS.

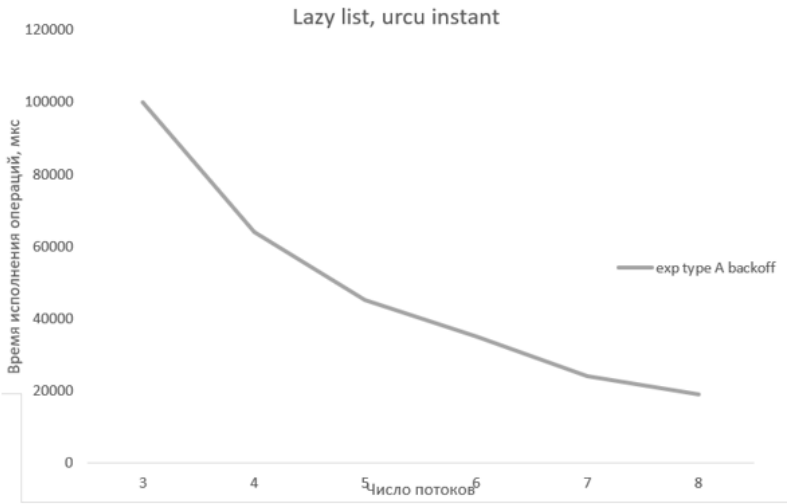
- Back-off delay – конкретное время простоя потока;
- Exponential-backoff – поток входит в режим ожидания на промежуток времени, определяемый параметрами min, max, причем каждый последующий период ожидания, начиная с min, увеличивается в два раза.

Результаты первичных тестов

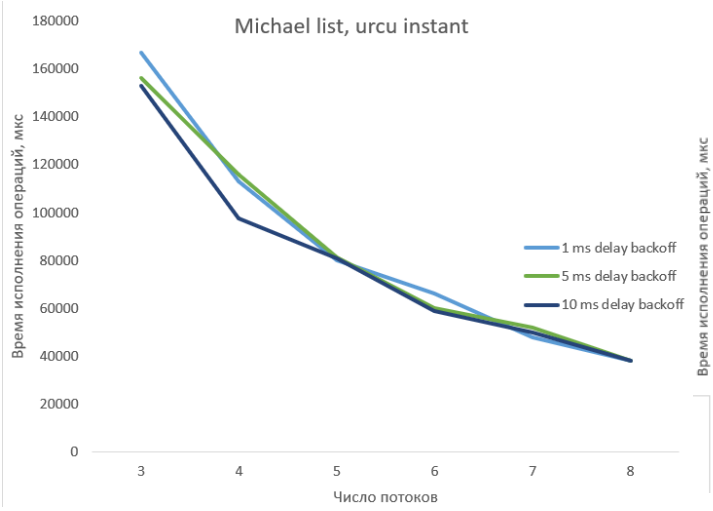
Сравнивались 2 структуры Michael List и Lazy List со следующими параметрами:

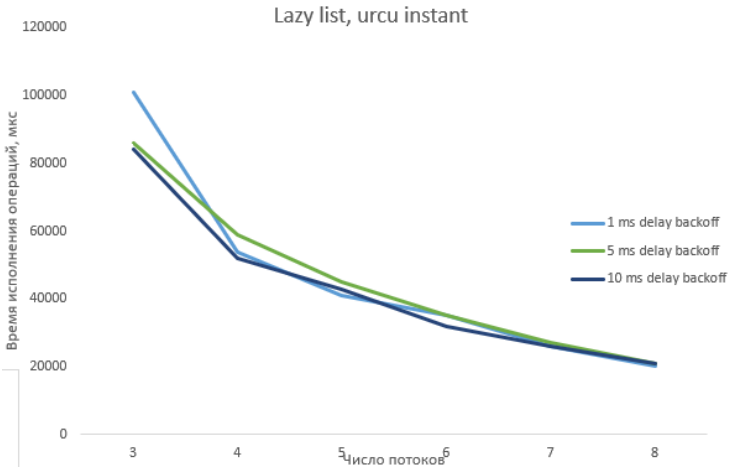
- 3 вида SMR схем;
- 5 различных back-off стратегий;
- Тип ключа элемента – int. 3200 операций.





Back-off





Сравнение back-off стратегий показало, что среднее время выполнения мало зависит от параметра задержки, что позволяет предположить, что потоки почти не нагружают структуру, т. е. очень редко происходит одновременное обращение потоков к одному и тому же элементу

Результаты оптимизаций Clang

Из всех вариантов опций наилучший результат быстродействия показал Lazy List со следующими опциями:

- Backoff стратегия exponential type A;
- URCU SMR схема с алгоритмом<instant>.

Результат - 18990 микросекунд.

Повысим масштаб – увеличим число операций до 32000.

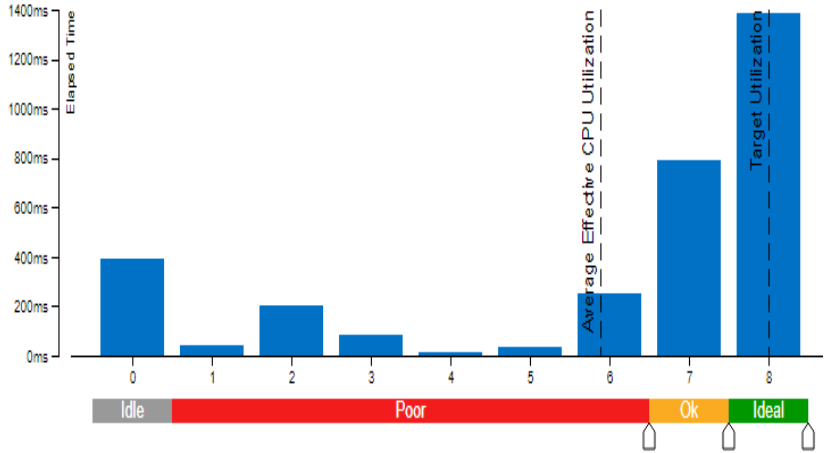
Результат в 1725006 микросекунд. После оптимизаций Clang – 521003 микросекунды – выйгрыш в 69,8%.

Рассмотрим проходы LLVM opt:

- 850 проходов всего;
- из них: 762 –inline, 18 –gvn;
- остальные проходы связаны с оптимизацией алгоритмов, не связанных с libcds.

```
remark: C:/libcds_dir/libcds\cds/urcu/details/base.h:357:0:

'?load@?$_Atomic_storage@PEAU?$thread_data@Ugeneral_instant_tag@urcu@cds
@@@details@urcu@cds@@
$07@std@@QEBAPEAU?$thread_data@Ugeneral_instant_tag@urcu@cds
@@@details@urcu@cds@@W4memory_order@2@@Z'

inlined into

'?detach_all@?$thread_list@Ugeneral_instant_tag@urcu@cds
@@V?$allocator@H@std
@@@details@urcu@cds@@QEAAXXZ'

with (cost=-30, threshold=375)
at callsite
?detach_all@?$thread_list@Ugeneral_instant_tag@urcu@cds
@@V?$allocator@H@std@@@details@urcu@cds@@QEAAXXZ:5:0;

[-Rpass=inline]
```

VTune Profiler



Профилировщик показал, что наибольшее процессорное время, 77.5%, 14.6 секунд занимает операция сравнения std::less(), а все функции структуры данных не превышают 2.529 секунд.

MyThreadFunctionExpACo	79.2%	0s	test.exe	MyThreadFunctionExpA
std::less<int>::operator()	77.5%	14.626s	test.exe	std::less<int>::operator(
cds::opt::details::make_cor	77.5%	0s	test.exe	cds::opt::details::make_

Заключение

- Был реализован и протестирован алгоритм списка Харриса
- выполнено сравнение методов Safe Memory Reclamation: наилучшее быстродействие показал алгоритм URCU Instant;
- в результате сравнения алгоритмов выяснилось, что в данных условиях оптимальная back-off стратегия не является решающим фактором в производительности структуры;
- Общий результат сравнения рассмотренных структур показал, что список Харриса уступает в производительности алгоритмам Libcds, среди алгоритмов Libcds максимальную скорость показал Lazy List с SMR URCU instant;
- профилировщик показал, что наибольшее время в алгоритмах Libcds занимает операция сравнения, поэтому выбор оптимальной функции позволяет сократить время выполнения программы;
- разбор логов оптимизатора LLVM показал, что наибольший прирост производительности дает инлайнинг методов и упрощение ненужных оптимизаций присваивания;
- разработчикам неблокирующих структур, в частности, Libcds, стоит обратить внимание на классические методы оптимизации, такие как векторизация циклов и применение специфичных для платформы SIMD инструкций. Сложные алгоритмы, представленные в исходном коде Libcds, не позволяют автоматически применять такие оптимизации, поскольку реализованы в большинстве своем на проходах по указателям.