УДК 004.043

РАЗБОР И РЕФАКТОРИНГ НЕОГРАНИЧЕННОЙ ОЧЕРЕДИ С ОДНИМ ПРОИЗВОДИТЕЛЕМ И ОДНИМ ПОТРЕБИТЕЛЕМ

Стухальский А. Л., магистрант кафедры ПОИСиТ

Научный руководитель – Прихожий А.А., профессор, д.т.н.

Ключевые слова: структура данных, очередь, многопоточность, атомарность, C++17, datastructure, spscqueue, multithreading, atomics.

В качестве объекта рассмотрения в статье выступает неограниченная очередь с одним потребителем и одним производителем (далее очередь). Интерфейс очереди состоит из конструктора, деструктора, методов для ввода/вывода в/из очередипроизводителем/потребителем соответственно.

В основе очереди [1]лежит идея повторного использования ячеек. Для этого используются два дополнительных указателя. Один из них указывает на начало диапазона неиспользуемых ячеек (конец диапазона перед хвостом очереди).Другой дополнительный указатель содержит старую копиюхвоста очереди и используется для уменьшения числа обращений кхвосту очереди.Т.к. хвост используется преимущественно потребителем, он находится в кэш-линии потока потребителя, то производителю из другой кэш линии нужно будет синхронизироваться с потребителем для получения последнего значения хвоста, что дольше, чем работа с локальной копией.

Схема очереди приведена на рисунке 1.

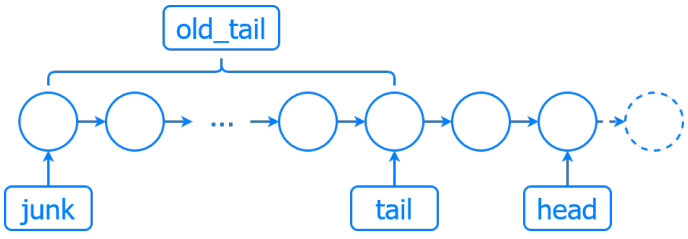

Рисунок 1 Схема очереди Дмитрия Выкова

Ещё одна оптимизация в оригинальном программном коде заключается в разделении кэш-линий для потоков производителя и потребителя таким образом, чтобы они не пересекались. Для этого в оригинальном коде использовалось свойство C++ сохранять порядок переменных при компиляции (переменные следуют в памяти в том же порядке, как они шли в исходном коде слева-направо сверху-вниз) и промежуточный буфер размером с кэш-линию. Таким образом в кэше по порядку шли tail(часть потребителя), отступ (чтобы перенести часть производителя на отдельную кэш-линию), head, junki old_tail (часть производителя).

Начиная с C++11 выравнивание в памяти можно осуществлять с помощью сочетания спецификатора alignas[2]и анонимных структур (см. листинг 1).Спецификатор alignasприменяется для типа, в т.ч. структуры (как в нашем примере), а анонимные структуры позволяют обернуть несколько переменных не меняя способ обращения к ним во внешнем коде. Схема выравнивания указателей в памяти приведена на рисунке 2.

```
struct alignas(cache_line_size) {
node *tail;
};
struct alignas(cache_line_size) {
node *head, *junk, *old_tail;
};
```

Листинг 1программный код выравнивания в памяти через спецификатор alignas

Рисунок 2 Выравнивание указателей очереди в памяти с целью разделения потоков производителя и потребителя по разным кэш-линиям

С целью безопасного использования очереди, в оригинальном коде конструктор копирования и перегрузка оператора присвоения оставлены с пустой реализацией в приватной области видимости, т.о.объект очереди нельзя будет скопировать, что и не имеет смысла делать для многопоточных структур данных.Начиная C++11 для явного удаления функций используется спецификатор delete, который явно указывает, что реализации нети при вызове данной функции будет выдана ошибка компиляции. Пример явного удаления из обновлённого кода приведен на листинге 2.

В оригинальном коде деструктора указатель junk копировался в переменную, и в цикле удалялось содержимое очереди. От использования локальной переменной можно отказаться, а тело цикла заменить вызовом функции std::exchange введенной в C++14 в пакете utility. Сравнение реализаций деструктора приведено на рисунке 3.

```
spsc_queue(spsc_queue const&) = delete;
spsc_queue& operator=(spsc_queue const&) = delete;
```

Листинг 2 Явное удаление конструктора копирования и оператора присвоения очереди

```
~spsc_queue()
{
  node* n = junk;
  do {
    node* next = n->next;
    delete n;
    n = next;
  }
  while (n);
}
```

```
~spsc_queue() {
  while (junk)
    delete std::exchange(
      junk, junk->next);
}
```

Рисунок 3 Сравнение реализаций деструктора

В оригинальном коде в методе вывода из очереди некорректно использовался указатель на хвост. Будучи единственной разделяемойпеременной между двумя потоками, в методе напрямую использовалось его значение. Лучше загрузить значение в локальную копию и в дальнейшем работать с ней. Так же в методе присутствовал избыточный барьер памяти для сохранения нового значения хвоста. Он не нужен поскольку с этим

указателем работает только потребитель, синхронизировать операцию записи не с чем, следовательно и порядок операций не важен.

В методе enqueue в качестве аргумента было фактическое значение, т.е. содержимое копировалось и при передаче в функцию и при записи в очередь. Во избежание копий, начиная с C++11, можно использовать механизм перемещения (rvaluereference[3]), что позволяет избежать промежуточных копий. При передаче аргументов черезrvaluereference,копирования аргумента при передаче в функцию не будет, а внутреннее содержимое аргумента будет перемещено в ячейку без копирования.

```
#pragma once
#include <utility>
#include <atomic>
static constexpr std::size_t cache_line_size = 64;
template<typename T>
class spsc_queue {
public:
spsc_queue() {tail = head = junk = old_tail = new node();}
~spsc_queue() {
while (junk) delete std::exchange(junk, junk->next);
}
void enqueue(T&& v) {
node *n = alloc_node();
n->next = nullptr;n->value = v;
head = head->next = n;
std::atomic_thread_fence(std::memory_order_release);
}
bool dequeue(T& v) {
auto t = tail;
std::atomic_thread_fence(std::memory_order_consume);
if (!t->next) return false;
v = t->next->value;
tail = t->next;
return true;
}
private:
struct node{node *next = nullptr;T value;};
struct alignas(cache_line_size) { node *tail; };
struct alignas(cache_line_size) {
node *head, *junk, *old_tail;
};
node* alloc_node() {
if (junk == old_tail) {
old_tail = tail;
if (junk == old_tail) return new node();
}
return std::exchange(junk, junk->next);
}
spsc_queue(spsc_queue const&) = delete;
spsc_queue& operator=(spsc_queue const&) = delete;
};
```

Листинг 3 Полный код переработанной очереди

Например, вместо копирования массива по указателю, будет скопирован только указатель.

Упрощена реализация функции alloc_node от повторения кода, выделявшего ячейку из числа неиспользуемых, за счёт инвертирования условий. Сравнение реализаций приведено на рисунке 4.Проверить идентичность работы реализаций можно по графам вариантов исполнения, приведенным на рисунке 5, где условия обозначены через используемые операторы сравнения, загрузка хвоста словом load, выделение ячейки из диапазона неиспользуемых словом junk и выделение новой ячейки словом new.

Полный программный код очереди приведен на листинге 3.

```
node* alloc_node() {
  if (junk != old_tail) {
    node* n = junk;
    junk = junk->next;
    return n;
  }
  old_tail = load_consume(&tail);
  if (junk != old_tail) {
    node* n = junk;
    junk = junk->next;
    return n;
  }
  node* n = new node;
  return n;
}
```

```
node* alloc_node() {
  if (junk == old_tail) {
    old_tail = tail;
    if (junk == old_tail)
      return new node();
  }
  return std::exchange(junk, junk->next);
}
```

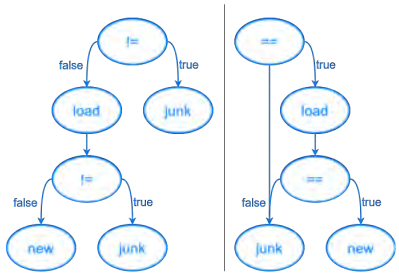

Рисунок 5 Графы вариантов исполнения

Рисунок 4 Сравнение реализаций метода alloc_node реализаций метода alloc_node

Литература

1. Unbounded SPSC Queue [Электронный ресурс]. – Режим доступа: http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue. – Датадоступа: 20.04.2020.

2. alignasspecifier [Электронный ресурс]. – Режим доступа: https: //en.cppreference.com/w/cpp/language/alignas. – Датадоступа: 28.04.2020.

3. Referencedeclaration [Электронный ресурс]. – Режим доступа: https:// en.cppreference.com/w/cpp/language/reference. – Дата доступа: 15.04.2020.