Лекция 12. Структуры данных без блокировок

ИУ8

November 25, 2016

Содержание

План лекции:

- что такое структуры данных без блокировок
- примеры структур данных без блокировок
- проблемы с lock free алгоритмами
- тестирование параллельных алгоритмов

Определения

Алгоритмы и структуры данных, в которых для синхронизации используются мьютексы, условные переменные и т.д., называются блокирующими.

Структуры данных и алгоритмы, которые не используют блокирующие функции, называются неблокирующими.

Рассмотрим спинлок-мьютекс

```
1  class spinlock_mutex {
2      std::atomic_flag _locked = ATOMIC_FLAG_INIT;
3  public:
4      void lock() {
5          while(_locked.test_and_set());
6      }
7      void unlock() {
8          _locked.clear();
9      }
10 };
```

Он не является блокирующим, но в то же время он не свободен от блокировок

Структуры данных без блокировок

Разделяемый объект называется lock-free объектом, если он гарантирует, что некоторый поток закончит выполнение операции над объектом за конечное число шагов вне зависимости от результата работы других потоков

Если структура данных открыта для одновременного доступа со стороны нескольких потоков, то считается, что она свободна от блокировок. *Не требуется, чтобы потоки могли выполнять одну и ту же операцию.*

Более того, если один из потоков был приостановлен в середине операции над структурой данных, то остальные потоки должны иметь возможность завершения своих операций.

Wait free

Понятие wait-free объекта (объект, свободный от ожидания) подразумевает, что каждый поток завершит операцию над объектом за конечное число шагов.

Условие lock-free гарантирует продвижение как минимум какого-то одного потока, тогда как более сильное условие wait-free гарантирует успешное выполнение для всех потоков.

Причины использования lock-free структур данных

Примитивы синхронизации иногда являются объектами ядра ОС, поэтому обращение к такому объекту может быть очень дорогим: может потребоваться переключение контекста, переход на уровень ядра ОС, поддержка очередей ожидания доступа к защищаемым примитивам синхронизации данных и пр.

Все описанные шаги в случае со структурами данными, необходимы только для изменения незначительного количества байт. Накладные затраты могут быть (по факту и есть) очень велики.

Ещё одним недостатком синхронизации является слабая масштабируемость. При возрастании числа потоков синхронизированный доступ к данным становится узким местом программы.

За и против

Основная причина использования структур данных, свободных от блокировок - достижение максимального уровня параллелизма.

Вторая причина - надежность. Если поток завершается, не освободив блокировку, то все ожидающие потоки не смогут продолжить работу. В случае с алгоритмами без блокировок все потоки, работающие с одной структурой данных, продолжают свои операции (почти) игнорируя другие потоки.

В lock-free структурах данных невозможны взаимные блокировки, но им на смену приходят активные блокировки.

Реализация структур данных требует больше усилий и знаний.

Хотя lock-free структуры данных позволяют лучше распараллелить операции над структурой данных и сократить время ожидания в каждом конкретном потоке, общая производительность программы вполне может и упасть.

"Суть" структур данных без блокировок

Пытаемся до тех пор, пока не получится

```
1
2  do{
3
4    prepareChanges();
5
6    if (hasConflict() == NO) {
7
8      commitChanges();
9
10     break;
11
12   }
13
14 } while(true);
```

Lock-free Stack

```
bool try_pop(T & data) {
  Node * old_head = _head.load();
  if (old_head == nullptr)
    return false;

  while (old_head && !_head.compare_exchange_strong(old_head, old_head->
      next))
    ; // empty loop

  if (old_head == nullptr)
    return false;

  // if copy throws exception, then the value will be lost
  data = old_head->data;
  return true;
}
void push(const T & data) {
  Node * new_head = new Node(data, _head.load());

  while (!_head.compare_exchange_strong(new_head->next, new_head))
    ; // empty loop
}
```

Проблемы при реализации структур данных без блокировок

Lock-free структуру данных нелегко придумать;
даже если вы её придумали, нелегко её запрограммировать;
даже если вы её запрограммировали, нелегко её отладить;
даже если вы её отладили, нелегко доказать, что алгоритм работает правильно;
наконец, даже если вы это доказали, все равно нет уверенности, что lock-free
структура данных работает правильно (c)

Специфичные проблемы для C++

Специфической проблемой при реализации структур данных без блокировок на C++ (как и на других языках без GC) является управление памятью. Тут стандартно присутствуют две проблемы: использование освобожденной памяти, освобождение всей выделенной памяти.

Чтобы освободить участок памяти, надо быть уверенным, что он нигде не используется. Однако при реализации структур данных без блокировок добиться такой уверенности стоит больших ресурсов: памяти, процессорного времени, умных программистов.

Проблема ABA

Рассмотрим следующую ситуацию:

- поток 1 читает значение A из разделяемой памяти
- ОС приостанавливает поток 1
- поток 2 меняет значение A на B и обратно на A перед вытеснением,
- поток 1 возобновляет работу, видит, что значение не изменилось, и продолжает

Такое поведение приемлемо не всегда. Если из списка удалить элемент, уничтожить его, а затем создать новый элемент и добавить обратно в список, есть вероятность, что новый элемент будет размещён на месте старого. Указатель на новый элемент совпадёт с указателем на старый, что и приведёт к проблеме: равенство признаков не есть равенство данных целиком.

Lock-free Stack без утечек памяти

See 02.cpp и 03.cpp

Схемы управления памятью

Разработано много различных схем для управления памятью, которые можно использовать в lock-free структурах данных. Например,

- дождаться, когда к структуре данных обращается ровно один поток и тогда удалить данные;
- использовать указатели опасности (hazard pointer);
- подсчитывать ссылки на объекты;
- др.

Hazard pointer

Рассмотрим схему управления памятью, предназначенную для защиты локальных ссылок на элементы lock-free структуры данных.

Основная идея

- Когда поток обращается к каким-то разделяемым данным, он помечает в глобальной памяти **HazardPointers**, что эти данные используются. Соответственно никто не может их удалять;
- Когда поток перестает обращаться к разделяемым данным, он помечает в глобальной памяти **HazardPointers**, что эти данные НЕ используются;
- Когда поток собирается удалить элемент, он кладет его в очередь элементов, готовых к удалению;
- При заполнении очереди до определенного размера, пытаемся очистить очередь и удалить элементы. Удаляем только те элементы из очереди, которых нет в **HazardPointers**.

Рекомендации

- используйте std::memory_order_seq_cst для прототипа
- не забывайте о ABA проблеме
- внимательно рассмотрите все возможные схемы управления памятью, выберите подходящую для ваших целей
- помогайте потокам, которые застряли в активных циклах ожидания
- используйте unit тесты

Тестирование

Реализация любого многопоточного приложения требует закалки и психологического здоровья программиста.

Реализация же с использованием алгоритмов без блокировок требует еще больше сноровки, закалки и стальных нервов.

Для сохранения времени и здоровья используйте средства для тестирования параллельных алгоритмов, lock-free структур данных, которые позволяют выявить ошибки, которые трудно воспроизвести и найти обычными средствами.

Relacy Race Detector

Инструмент, автором которого является Дмитрий Вьюков, позволяющий выполнять unit-тесты для параллельных алгоритмов.

Features

- Relaxed ISO C++0x Memory Model. Relaxed/acquire/release/acq_rel/seq_cst memory operations. The only non-supported feature is memory_order_consume, it's simulated with memory_order_acquire.
- Exhaustive automatic error checking (including ABA detection).
- Full-fledged atomics library (with spurious failures in compare_exchange()).
- Memory fences.
- Arbitrary number of threads.
- Detailed execution history for failed tests.
- Before/after/invariant functions for test suites.

Detectable Errors

- Race condition
- Access to uninitialized variable
- Access to freed memory
- Double free
- Memory leak
- Deadlock
- Livelock
- User assert failed
- User invariant failed

Скрестить ужа с ежом

При программировании структур данных без блокировок одной из главных проблем является управление памятью и освобождение выделенных элементов. Поэтому это отличное место для экспериментов с системами сборки мусора. Задача: реализовать стек(или очередь) без блокировок, в которой для освобождения выделенной памяти, будет использоваться gcpp. Первому умельцу бонус на зачете.

The end

Самостоятельное изучение

- Реализация lock-free очереди
- Максим Хижинский - Цикл статей "Lock-free структуры данных"

Список литературы

- Уильямс - Параллельное программирование на C++ в действии
- Максим Хижинский - Цикл статей "Lock-free структуры данных"
- Relacy Race Detector
- Lock-free программирование