Лекция 10. Атомарные операции и модель памяти

ИУ8

November 11, 2016

Содержание

На текущей лекции рассмотрим:

- атомарные операции
- новая модель памяти C++11

Базовая модель

```
1  struct Test
2  {
3      int a;
4      int b;
5  };
6
7  Test first;
8  first.a = 5; // thread 1
9  first.b = 4; // thread 1
```

```
1  Test first;
2
3  { // thread 1
4  Test tmp = first;
5  tmp.a = 5;
6  first = tmp;
7  }
8
9  { // thread 2
10 Test tmp = first;
11 tmp.b = 4;
12 first = tmp;
13 }
```

Итак, вместо ожидаемого пользователем firsta=5, b=4 мы получим firsta=5, b=?? (или a=??, b=4, или еще что-то). И это абсолютно легально в C++03, т.к. старая модель не подразумевает никакой многозадачности.

Атомарные объекты

Появление многопоточности вносит сложность с загрузкой и сохранением данных.

Не любая структура данных может быть загружена/сохранена в память одной процессорной операцией. Более того, то, что может быть загружено/сохранено в память одной операцией на одной архитектуре, не может быть на другой

Атомарный объект – это такой объект, операции над которым можно считать неделимыми, т.е. такими, которые не могут быть прерваны или результат которых не может быть получен до окончания операции.

Атомарные объекты в C++11

Когда один поток сохраняет данные в объекте атомарного типа, а другой хочет их прочитать, поведение программы определено стандартом, в отличие от ситуаций, когда используются не атомарные типы.

В C++11 появилось два типа атомарных объектов: std::atomic<T> и std::atomic_flag

Эффективность atomic vs mutex

Конечно же, атомарные операции можно реализовать с помощью мьютекса.

Недостатком такой реализации является вероятное снижение эффективности, так как захват и освобождение мьютекса не самые простые операции.

Атомарные объекты **могут** быть гораздо эффективнее.

Атомарный доступ может понадобится к структуре любой сложности и размера, но настоящей атомарной операцией над данными можно считать лишь ту, которую процессор определенной архитектуры может выполнить одной командой.

Чтобы оставить возможность атомарного доступа, атомарность должна эмулироваться для всех типов, которые процессор не может обрабатывать атомарно. Возможно, за счёт тех же самых мьютексов.

is_lock_free

Стандарт предоставляет возможность разработчику точно знать, является атомарный объект требуемого типа свободным от блокировок или нет.

```
1 std::atomic<int> var;
2 std::atomic_uint64_t var2;
3 std::cout << std::boolalpha << var.is_lock_free();
4 std::cout << std::boolalpha << var2.is_lock_free();
```

std::atomic_flag единственный гарантированно является свободным от блокировок.

Исходя из вышесказанного можно предположить, что остальные атомарные типы, для которых не существует свободной от блокировок версии на той или иной архитектуре, будут реализованы посредством atomic_flag

atomic_flag

Содержит всего две операции: test_and_set и clear, что вполне достаточно для флага, ведь он может быть либо поднятым, либо опущенным.

Еще одним важным свойством atomic_flag, которое необходимо упомянуть, является его неопределенность при создании.

Поэтому для получения предсказуемого результата есть смысл всегда инициализировать флаг. Для этих целей существует специальный макрос ATOMIC_FLAG_INIT.

mutex

```
1  class CustomMutex {
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 };
```

Методы std::atomic<T>

- store – Кладет новое значение в объект.
- load – Извлекает значение из объекта.
- exchange – Заменяет значение в объекте на новое и возвращает старое.
- compare_exchange_strong[weak](expected, desired) – Если object равен expected, тогда desired помещается в object. В противном случае object помещается в expected.

std::atomic<integral>

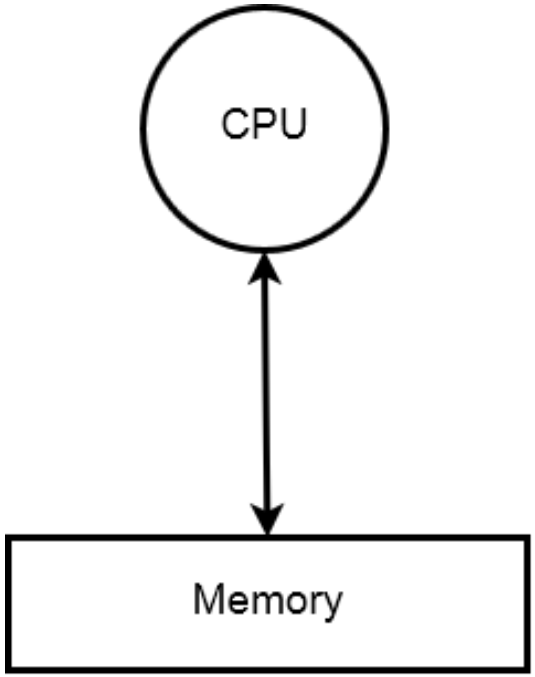
Отличительной особенностью этой версии является наличие дополнительных операций, которые можно осуществлять с атомарным интегральным объектом: fetch_add, fetch_sub, fetch_and, etc.

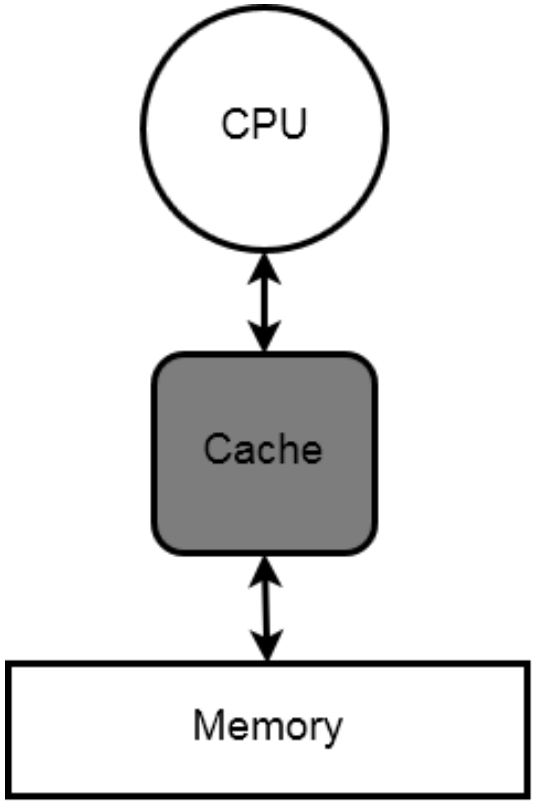
std::atomic<T*>

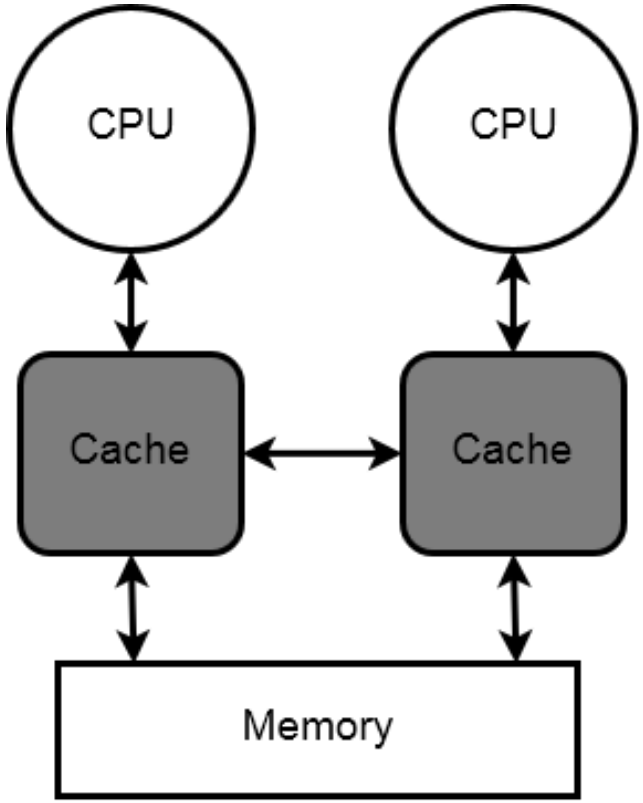
Этот тип используется для всех указателей, работа с которыми должна быть атомарна. В этом типе есть оператор разыменовывания указателя.

Новая модель памяти

Welcome to hell


CPU
Memory


CPU
Cache
Memory


CPU
CPU
Cache
Cache
Memory

Пример оптимизаций

```
1  int data;
2  volatile bool ready = false;
3
4  void thread1() {
5    data = 42;
6    ready = true;
7  }
8  void thread2() {
9    if(ready) {
10     assert(data == 42);
11   }
12 }
```

Процессор может оптимизировать выполнение кода путем переупорядочивания операции.

```
1  auto t = data;
2  if(ready)
3    assert(t == 42);
```

```
1  ready = true;
2  data = 42;
```

Барьеры памяти

```
1  int data;
2  volatile bool ready = false;
3
4  void thread1() {
5    data = 42;
6    // barrier here: STORE_STORE
7    ready = true;
8  }
9  void thread2() {
10   if(ready) {
11     // barrier here: LOAD_LOAD
12     assert(data == 42);
13   }
14 }
```

Барьер памяти типа XX_YY гарантирует, что все операции XX операции до барьера будут выполнены до того, как начнут выполняться YY операции после барьера.

Acquire (захват)

Барьеры типа Acquire гарантируют, что все операции **после** барьера будут выполнены **после** Load-операций **до** барьера.

Release (освобождение)

Барьеры типа Release гарантируют, что все операции **до** барьера будут выполнены **до** Store-операций **после** барьера.

```
1  int i;
2  std::atomic<bool> ready = false;
3
4  void thread1() {
5    data = 42;
6    ready.store(true, std::memory_order_release);
7  }
8  void thread2() {
9    while(!ready.load(std::memory_order_acquire))
10     ;
11   assert(i == 42);
12 }
```

Какие модели памяти бывают

- sequential consistency
- strongly-ordered
- weakly-ordered
- super-weak (relaxed)

sequential consistency

Никаких переупорядочиваний

Гарантируется, что каждый процессор видит все изменения в системе в том же порядке, что и остальные

strongly-ordered

Все процессоры видят последовательности чтения и записей в одном порядке

Пример

x86, AMD64

weakly-ordered

Гарантируется упорядоченность для операций, зависимых по данным

```
1 int g_var = 0;
2 std::atomic<int*> ptr = &g_var;
3
4 void thread1(){
5   if(ptr){
6     int value = *ptr;
7     // ...
8   }
9 }
```

Пример

ARMv7, POWER

```
enum memory_order{
  memory_order_seq_cst,
  memory_order_acq_rel,
  memory_order_release,
  memory_order_acquire,
  memory_order_consume,
  memory_order_relaxed
};
```

Последовательная согласованность

Наиболее строгой и простой для понимании моделью является последовательно согласованная модель памяти.

Она является моделью по умолчанию во всех функциях, которые принимают параметр типа std::memory_order

При использовании такой модели любые две операции с атомарными операциями не могут завершиться в один и тот же момент времени.

Все операции с параметром std::memory_order_seq_cst выполняются так, как будто существует единый поток исполнения и все операция выстроены в этом потоке относительно друг друга, как и в обычном потоке исполнения с помощью отношения предшествования.

```cpp
std::atomic_int first{0};
std::atomic_int second{0};
std::atomic_int third{0};

void thread1()
{
    first.store(5); // #F-1
    second.load(); //#S-1
    third.store(6); //#T-1
}

void thread2()
{
    second.store(12); //#S-2
    first.store(13); //#F-2
    third.load();  //#T-2
}

void thread3()
{
    third.store(33); //#T-3
    second.load();  //#S-3
    first.load();  //#F-3
}
```

П1	П2	П3
П1-1	П1-1	П1-1
П1-2	П1-2	П1-2
П2-1	П2-1	П2-1
П3-1	П3-1	П3-1
П1-3	П1-3	П1-3
П3-2	П3-2	П3-2
П3-3	П3-3	П3-3
П2-2	П2-2	П2-2
П2-3	П2-3	П2-3

ГП
П1-1
П1-2
П2-1
П3-1
П1-3
П3-2
П3-3
П2-2
П2-3

Порядок чтения

Ещё одним важным свойством последовательно согласованных операций является тот факт, что при использовании операции чтения на атомарном объекте, такая операция гарантировано вернёт последнее записанное значение.

Модель acquire-release

Данная модель памяти является более слабой. Это значит, что она накладывает меньше ограничений на исполняемый код и дает программисту меньше гарантий.

В C++ данная модель присуща операциям, которые промаркированы следующими членами std::memory_order:

- memory_order_acquire – этим членом можно маркировать операции, которые загружают значение атомарного объекта. Операции, использующие данный маркер, являются операциями захвата.
- memory_order_release – этим членом можно маркировать операции записи данных в атомарный объект. Операции, использующие данный маркер, являются операциями освобождения.
- memory_order_acq_rel – этим членом можно маркировать операции чтения/изменения записи. Операции, использующие данный маркер, являются операциями захвата для предыдущих операций и операциями освобождения для последующих операций.

```cpp
std::atomic_int first{0};
std::atomic_int second{0};
std::atomic_int third{0};

void thread1() {
    first.store(5, std::memory_order_release);//P1-1
    second.load(std::memory_order_acquire);//P1-2
    third.store(6, std::memory_order_release);//P1-3
}

void thread2() {
    second.store(12, std::memory_order_release);//P2-1
    first.store(13, std::memory_order_release);//P2-2
    third.load(std::memory_order_acquire);//P2-3
}

void thread3() {
    third.store(33, std::memory_order_release);//P3-1
    second.load(std::memory_order_acquire);//P3-2
    first.load(std::memory_order_acquire);//P3-3
}
```

Данная модель позволяет двум независимым операциями завершаться одновременно.

П1	П2	П3
	П2-1	
П1-1		П3-1
	П2-2	
П1-2	П2-3	П3-2
		П3-3
П1-3		

Весь смысл подобной ослабленной модели заключается в том, что при определенных обстоятельствах использование ЭТОЙ МОДЕЛИ может дать прирост в производительности.

Порядок чтения при модели захвата-освобождения

С моделью ПС любая операция чтения загружает последнее сохранённое значение в атомарном объекте.

С моделью захвата-освобождения это не так. Единственным ограничением, которое накладывает данная модель, является следующее: если порядок изменений атомарного объекта A состоит из значений (i1, i2, i3, . . . , in, где iL происходит позже iB, если $L > B$), и если поток П1 загрузил значение iL, то ни при каких обстоятельствах при следующей загрузке он не может получить значение iB, где $B < L$.

Синхронизация операций с помощью атомарных операций

В примере ниже показана вся мощь синхронизационного потенциала атомарных объектов.

```
1  int simpleInt{0};
2  std::atomic_bool flag{false};
3
4  void thread1() {
5      simpleInt = 911;
6      flag.store(true, std::memory_order_release);
7  }
8
9  void thread2() {
10     while(flag.load(std::memory_order_acquire))
11         ;
12     assert(simpleInt == 911);
13 }
```

Задел для Lock-free алгоритмов

Барьер памяти типа Release гарантирует, что запись в **неатомарный** объект произойдет раньше, чем установка значения в **атомарный** объект flag.

Барьер памяти типа Acquire гарантирует, что чтение **неатомарного** объекта произойдет позже, чем получение значения **атомарного** объекта flag.

Тем самым удалось синхронизировать доступ к **неатомарному** объекту без использования высокоуровневых блокировок.

Описанные техники применяются при реализации lock-free алгоритмов.

The end

Самостоятельное изучение

- must read part 4
- must read part 5
- must watch
- Memory Barriers: a Hardware View for Software Hackers

Список литературы

- http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.0
- http://scrutator.me/post/2015/08/14/parallel-world-p4.aspx
- http://scrutator.me/post/2015/10/15/parallel_world_p5.aspx
- https://www.youtube.com/watch?v=SIZmLPtcZiE
- http://preshing.com/20130922/acquire-and-release-fences/
- Уильямс - Параллельное программирование на C++ в действии