Задание №6. Модель памяти

Рубаненко Евгений

Апрель 2017

1 Single-Producer/Single-Consumer Fixed-Size Ring Buffer (Queue)

Корректная реализация очереди приведена ниже.

```
#pragma once

#include <atomic>
#include <vector>

// Single-Producer/Single-Consumer Fixed-Size Ring Buffer (Queue)
template <typename T>
class SPSCRingBuffer {
public:
    explicit SPSCRingBuffer(const size_t capacity)
        : buffer_(capacity + 1) {
    }

    bool Publish(T element) {
        const size_t curr_head =                          (1)
            head_.load(std::memory_order_acquire);
        const size_t curr_tail =                          (2)
            tail_.load(std::memory_order_relaxed);

        if (Full(curr_head, curr_tail)) {
            return false;
        }

        buffer_[curr_tail] = element;                     (3)
        tail_.store(Next(curr_tail), std::memory_order_release);
        return true;
    }

    bool Consume(T& element) {
        const size_t curr_head =                          (4)
            head_.load(std::memory_order_relaxed);
        const size_t curr_tail =                          (5)
            tail_.load(std::memory_order_acquire);

        if (Empty(curr_head, curr_tail)) {
            return false;
        }

        element = buffer_[curr_head];                     (6)
        head_.store(Next(curr_head), std::memory_order_release);
```

```
        return true;
    }

private:
    bool Full(const size_t head, const size_t tail) const {
        return Next(tail) == head;
    }

    bool Empty(const size_t head, const size_t tail) const {
        return tail == head;
    }

    size_t Next(const size_t slot) const {
        return (slot + 1) % buffer_.size();
    }

private:
    std::vector<T> buffer_;
    std::atomic<size_t> tail_{0};
    std::atomic<size_t> head_{0};
};
```

Неатомарные чтения и записи

В данном случае нам хотелось бы знать, что сейчас находится в очереди. Этого можно добиться, если после каждого добавления или удаления из очереди оповещать другие потоки об изменениях. Таким образом, в данном случае мы будем использовать
$std :: memory\_order\_release$ и $std :: memory\_order\_acquire$. Но заметим следующее - в (2) и (4) используется
$std :: memory\_order\_relaxed$. Это нормально (объяснение смотри в следующем блоке).

Happens-before

Соответсвующие $std :: memory\_order\_release$ и
$std :: memory\_order\_acquire$ проведут стрелку $Synchronized - with$. Корректность использования $std :: memory\_order\_relaxed$
объясняется с помощью $Program - order$ (Точнее, это нормально, так как после соответствующего *store* реордеринги происходить не будут [Еще точнее, проблема может возникнуть, если полученное неатомарное чтение получит плохое значние - но оно может стать плохим, только если отработала вторая функция - а тогда был вызван *store* - то есть будет выдано нормальное значение]). Далее, как и ранее, по транзитивному замыканию получаем отношение $Happens - before$.

Гарантии упорядочивания

Очевидно, что отказаться от
$std :: memory\_order\_acquire$ и $std :: memory\_order\_release$
нельзя, ведь тогда $Consume(T\& \ element)$ сможет не увидеть изменения, внесенные $Publish(T \ element)$.