Домашняя работа №1

Рубаненко Евгений

Март 2017

1 Алгоритм Петерсона

Утверждение. Измененный алгоритм Петерсона не гарнтирует условие взаимного исключения.

Доказательство:

После изменений алгоритм Петерсона для двух потоков будет выглядеть следующим образом:

```
class PetersonMutex
{
public:
    PetersonMutex()
    {
        want[0].store(false);
        want[1].store(false);
        victim.store(0);
    }

    void lock(int threadId)
    {
        want[threadId].store(true);
        victim.store(threadId);
        while (want[1 - threadId].load() &&
                victim.load() == threadId) {
            // wait
        }
    }

    void unlock(int threadId) {
        want[threadId].store(false);
    }

private:
    std::array<std::atomic<bool>, 2> want;
    std::atomic<int> victim;
};
```

Покажем, что при такой реализации нарушается условие взаимного исключения:

```
    Thread[0] : victim = 0;
    Thread[1] : victim = 1;
    Thread[1] : want[1] = true;
    Thread[1] : Enter critical section;
    Thread[0] : want[0] = true;
    Thread[0] : Enter critical section;
```

Таким образом, оба потока оказываются одновременно в критической секции. □

2 Tricky Mutex

Утверждение. Такая реализация мьютекса гарантирует взаимное исключение, но не гарантирует свободу от взаимной блокировки.
Доказательство:
Взаимное исключение: Допустим, что два потока могут одновременно зайти в критическую секцию. Это возможно тогда и только тогда, когда условие

```
thread_count.fetch_add(1) > 0
```

ложно для обоих потоков. Другими словами, когда в каждом из потоков будет проходить эта проверка, счетчик

```
thread_count
```

должен быть равен 0. Допустим, что для первого потока эта проверка происходит раньше. Тогда проверяемое условие окажется ложным и поток зайдет в критическую секцию. Но тогда для второго потока условие окажется верным, ведь сейчас

```
thread_count == 1
```

и потоку предется ждать. Второй поток будет ждать до того момента, пока для первого потока не будет вызван метод

```
unlock()
```

Таким образом, взаимное исключение гарантируется.
Свобода от взаимной блокировки: Очевидно, что если поток всего один, то свобода взаимной блокировки гарнтируется. Допустим, что потоков хотя бы два. В таком случае, приведем последовательность команд, которая покажет, что свобода от взаимной блокировки не гарантируется.

```
Thread[0] : lock();
```

Сейчас первый поток в критической секции.

```
Thread[1] : lock();
```

Второй поток заходит в цикл. В данный момент

```
thread_count == 2
```

Далее

```
Thread[0] : unlock()
```

Сейчас

```
thread_count == 1
```

А теперь вновь попробуем захватить мьютекс первым потоком

```
Thread[0] : lock()
```

В результате этого действия первый поток попадает в цикл. Далее приведем работу планировщика, из которой будет понятно, что ни один поток не сможет попасть в критическую секцию, а будет постоянно находиться в цикле ожидания.

```
1 >>> void lock() {
2 >>>     while (thread_count.fetch_add(1) > 0) {
3 >>>         thread_count.fetch_sub(1);
4 >>>     }
5 >>> }
```

Итак, сейчас оба потока стоят в позиции 3, ее они еще не выполнили,

```
thread_count == 0
```

```
Thread[0] : 3           /* this means that thread #i
                           execute string #j */
                        /* Now thread_count == 1 */
Thread[0] : 4
Thread[0] : 2           /* Now thread_count == 2 */
Thread[1] : 3           /* Now thread_count == 1 */
Thread[1] : 4
Thread[1] : 2           /* Now thread_count == 2 */
```

Т.е. планировщик может выполнять команды в таком порядке, что ни один из потоков не попадет в критическую секцию. Таким, образом, гарантия свободы от взаимной блокирвки отсутствует. □

3 Метод try_lock для ticket спинлока

Приведем реализацию метода try_lock и покажем, что в случае успеха мьютекс будет захвачен, а в случае неудачи - метод не повлияет на другие потоки.

```
bool ticket_lock::try_lock() {
    int oldVal = owner_ticket.load();
    return next_ticket.compare_exchange_strong(oldVal,
                                               oldVal + 1);
}
```

Утверждение. Такая реализация метода try_lock работает корректно.
Доказательство: Мьютекс может быть захвачен тогда и только тогда, когда

```
owner_ticket == next_ticket                    (1)
```

Если же

```
owner_ticket < next_ticket                     (2)
```

то это значит, что сейчас мьютекс захвачен. Отсюда заключаем, что захватывать мьютекс можно тогда и только тогда, когда выполняется условие (1). Отсюда становится понятной реализация метода try_lock: мы смотрим, какой билет будет исполняться следующим (owner_ticket), затем, если ничего не изменилось (например, не были вызваны lock из других потоков), увеличиваем следующий выдаваемый билет (то есть захватываем мьютекс). Если же мьютекс захвачен, то условие в

```
compare_exchange_strong
```

будет ложным и ничего не поменяется (то есть неудачный вызов try_lock не повлияет на другие потоки) □

4 Tournament Tree Mutex

Утверждение. TTM гарантирует свободу от голодания.
Доказательство: Докажем по индукции от корня к листу. Очевидно, что поток, который висит на мьютексе корня, голодать не может по свойству мьютекса Петерсона: впереди него только один поток, который рано или поздно выйдет из критической секции и уйдет вниз в свою ветку, а новые потоки из этой ветки обойти наш поток после этого не смогут, т.к. при попытке это сделать новый поток перезапишет victim в корневом мьютексе, а значит уступит дорогу.

Дальше спускаемся вниз по дереву: если поток завис на внутреннем узле, то поток, который опередил его и поднялся выше, спустится обратно в свою ветку по индуктивному предположению, а новые потоки из этой ветки не смогут обойти ждущий поток по той же самой причине,

что и в случае с корнем. □

Утверждение. TTM не гарантирует честность.

Доказательство: Допустим, что сейчас один из потоков (поток C) находится в критической секции, т.е. корневой мьютекс Петерсона захвачен. Без ограничения общности будем считать, что предыдущий lock этого потока был в левом поддереве. Тогда, если мьютекс попытается захватить поток A из левой половины всего дерева, то он зависнет, не дойдя до корня (деревья пересекутся). Теперь допустим, что после того, как поток A попытался захватить мьютекс, мьютекс попытается захватить поток B, который находился в правой половине дерева. Тогда он зависнет на правом ребенке корня (так как поток, который сейчас заблокирован в корневом мьютексе Петерсона пришел с левой стороны; также считаем, что других потоков нет). Тогда, после того как поток C отпустит мьютекс, его сможет захватить поток B, а потоку A придется ждать, пока разблокируются мьютексы Петерсона на пути, совпадающим с путем потока C. □