**Задача 1**

Покажем, что если в doorway-секции метода lock поменять местами (want[t].store(true)) и victim (victim.store(t)), то полученный алгоритм не будет гарантировать взаимное исключение.

void lock(int t) {

victim.store(t);

want[t].store(true);

while (want[1 - t].load() && victim.load() == t) {

// wait

}

}

Пусть порядок выполнения команд для потоков A и B такой (поток A - 0, B - 1):

**A:** victim.store(0) -> **B:**victim.store(1) -> **B**:want[1].store(true) -> **B:** while (want[0].load() && victim.load() == 1) -> **A:** want[0].store(true) -> **A:** while (want[1].load() && victim.load() == 0)

Тогда, когда поток B проверяет условие для входа в цикл, он видит, что want[0] == false, значит, он пройдет в критическую секцию. Далее, когда поток A проверяет условие (происходит до того, как вызван unlock() потока B), want[1] = true и victim == 1, значит, он тоже пойдет в критическую секцию, и в ней будет два потока одновременно, что противоречит взаимному исключению.

**Задача 2**

void ticket\_spinlock::lock() {

const ticket\_t this\_thread\_ticket = next\_free\_ticket\_.fetch\_add(1);

while (this\_thread\_ticket != owner\_ticket\_.load()) {

// wait

}

}

Покажем, что при n = 2 частота исполнения критических секций в системе сильно упадёт. Если поток один, то каждый раз, когда он берет билет, его номер совпадает с owner\_ticket (билеты берет только этот поток, значит, когда он проходит в критическую секцию его номер совпадает с owner\_ticket, owner\_ticket увеличивается на 1. Когда поток в следующий раз выполняет lock, он берет номер на 1 больший предыдущего, и он снова совпадает с owner\_ticket ), а значит, он постоянно будет выполнять критическую секцию, и за квант времени будет выполнено большое количество критических секций, так как по условию они короткие.

Рассмотрим случай, когда в системе исполняется n потоков, некоторый поток встал в очередь на спинлоке и был вытеснен планировщиком. Пусть это был поток с индексом 0. Тогда, так как в критическую секцию он не прошёл, owner\_ticket останется равным 0. Следующие потоки возьмут билеты с номерами от 1 до n – 1, и ни один из них не пройдет в критическую секцию. Первым в критическую секцию пройдет поток с индексом 0. Так как за квант времени может исполнится много критических секций, этот поток снова зайдет в doorway-секцию lock, возьмет билет n и будет крутится в цикле, пока не закончится квант времени. Следующий поток будет иметь индекс 1, он пройдет в критическую секцию, затем возьмет билет с номером n + 1, и тоже будет крутится в цикле. Заметим, что так каждый поток будет 1 раз за квант времени проходить в критическую секцию, получать номер на n больший, чем у него был до этого, и затем ждать в цикле. Таким образом, независимо от n за квант времени в среднем будет выполняться 1 критическая секция.

(Строго можно показать по индукции: если для потока выполнено предположение индукции, то есть он имел номер k, прошел в критическую секцию, owner\_ticket стал равным k +1, а номер потока – n + k, то следующий поток имеет номер k + 1(потоки в очереди – по возрастанию номеров), owner\_ticket = k + 1, next\_free\_ticket = n + k + 1, тогда этот поток проходит в критическую секцию, увеличивает owner\_ticket, получает номер n + k + 1.)

Получаем большое количество критических секций за квант времени при 1 потоке и 1 критическую секцию за квант времени при n > 1.

void tas\_spinlock::lock() {

while (locked\_.exchange(true)) {

// wait

}

}

С tas\_spinlock таких проблем не возникнет. Единственное условие, которое проверяет поток – свободен ли мьютекс. Тогда за свой квант времени поток будет захватывать мьютекс, выполнять критичекую секцию, затем возвращаться снова, и мьютекс будет свободен, так как квант времени потока ещё не закончен. Так за квант времени будет выполнено большое количество критических секций.