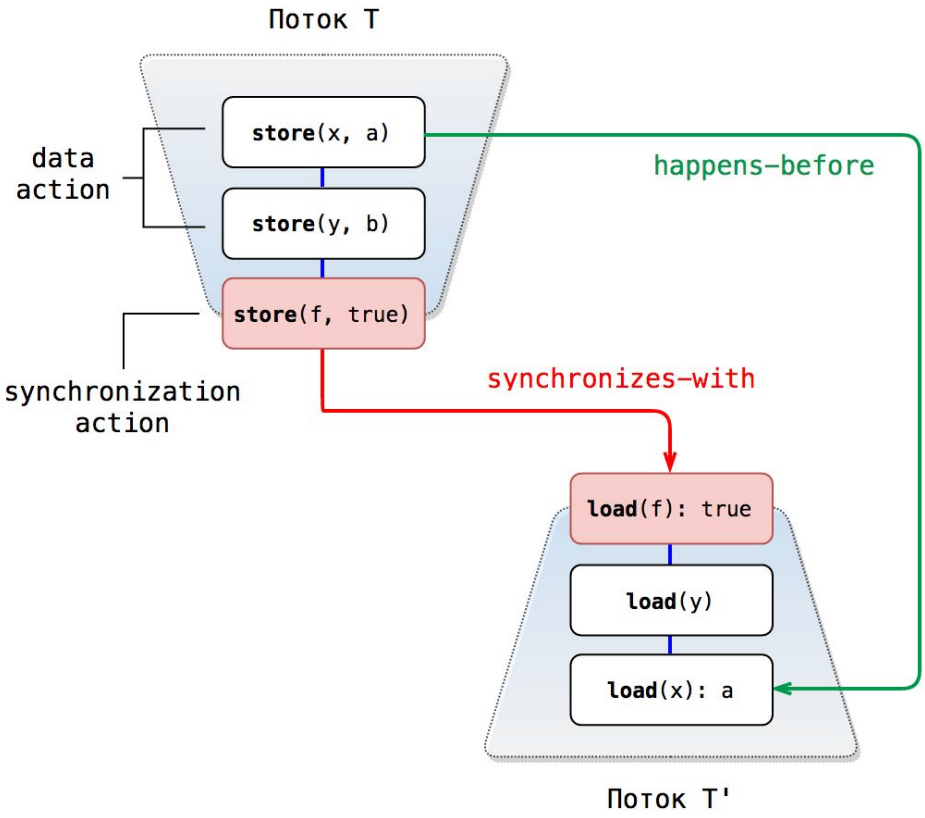
Модель памяти

Главный практический вопрос, на который отвечает модель памяти: **как гарантировать, что запись в память в одном потоке будет доступна чтению в другом потоке?**

Гарантия, которую дает модель памяти: **видимость** записей в память для чтений через отношение **happens-before**:



Замечание: изложение ниже справедливо только для режиме упорядочивания по умолчанию: **std::memory_order_seq_cst**.

Для **атомарных** переменных при исполнении гарантируется глобальный порядок всех обращений к памяти - **synchronization order**. Каждое чтение в этом порядке читает результат последней предшествующей записи.

Порядок **synchronization order** может отличаться от запуска к запуску и в общем случае заранее неизвестен (например, в мьютексе Петерсона мы не можем знать, какой из потоков последним запишет victim и проиграет).

Но важно здесь, что исполнение объясняется некоторым глобальным порядком.

Для **неатомарных** обращений к памяти все чуть сложнее: глобального порядка модель памяти не обещает, зато дает **гарантии видимости записей**:

1) В пределах одного потока модель памяти гарантирует видимость записей в последующих чтениях через отношение **program-order**, которое соответствует порядку инструкций в тексте программы.

2) Запись в одном потоке становится видимой для чтений в другом потоке, только когда между потоками возникает “мост” - стрелка **synchronizes-with**, по этому мосту записи “переходят” в другие потоки.

На картинке выше все записи в верхнем конусе будут видны в нижнем конусе.

Эта стрелка возникает только между атомарной **записью** и тем атомарным **чтением**, которое прочитало записанное значение.

3) Дальше неатомарные записи “растекаются” по транзитивности, двигаясь по стрелкам **program-order** и **synchronizes-with** и становятся доступными для чтений.

Транзитивное замыкание **program-order** и **synchronizes-with** образует частичный порядок **happens-before**.

Описанные гарантии работают только в случае, когда в программе нет гонок. Гонкой называются два конфликтующих обращения к памяти, которые не упорядочиваются во время исполнения отношением **happens-before**.

В этой домашней работе мы рассмотрим три сценария, в которых критично упорядочивание неатомарных обращений к памяти:

1. записи внутри критических секций для спинлока
2. single-producer/single-consumer циклический буфер
3. ленивая инициализация.

Сниппеты кода, в которых вам нужно будет обеспечить/объяснить упорядочивание памяти, находятся в репозитории на bitbucket.

Для каждого сценария нужно:

1. Объяснить, какие неатомарные чтения и записи и зачем должны быть упорядочены.
2. Показать, как алгоритм гарантирует возникновение стрелок **happens-before** между нужными чтениями и записями.
3. Максимально ослабить гарантии упорядочивания.

Все рассуждения нужно проводить в терминах формальной модели памяти, не прибегая к гарантиям конкретной архитектуры процессора и семантике барьеров памяти.

Рассуждать о переупорядочиваниях не стоит, стоит рассуждать **в терминах порядков**: **happens-before** и т.п.

Под ослаблением понимается использование режимов упорядочивания **release/acquire** и **relaxed**.

Гарантии, которые дают ослабленные модели упорядочивания:

1) release/acquire

- теряем глобальный порядок на обращениях к **разным** атомикам,

- сохраняем видимость записей через **happens-before**: если acquire-чтение прочитало результат release-записи, то между ними возникает **synchronizes-with**.

Такого режима может быть достаточно, если разные потоки синхронизируются через **один** атомик, а для каждого атомика в отдельности гарантируется единый порядок модификации.

Примеры, когда упорядочивания **release/acquire** не достаточно: 1) IRIW, 2) протокол взаимного исключения Петерсона для двух потоков.

Например, в примере IRIW с **release/acquire** два потока могут увидеть разный порядок записей в две разные ячейки:

IRIW (Independent Reads of Independent Writes) x = y = 0			
T1	T2	T3	T4
x = 1	y = 1	a = x b = y	c = y d = x

Допускается результат: a = 1, b = 0, c = 1, d = 0
Иначе говоря, потоки T3 и T4 видят разный порядок записей в ячейки x и y.

2) relaxed -- самый слабый режим упорядочивания, гарантирует только, что все потоки наблюдают единый порядок модификации каждой отдельной ячейки.

При этом разные потоки могут видеть две записи в две разные ячейки в разном порядке, никаких гарантий мы здесь не получаем.

При использовании relaxed между записью и чтением не возникает стрелка **synchronizes-with**, предшествующие записи не “перетекают” в другие потоки, так что делать выводы о содержимом других ячеек памяти на основе результата relaxed-чтения нельзя.
