Когерентность кэшей

Задача A. Честный масштабируемый спинлок - 5 баллов

В этой задаче вам нужно реализовать честный масштабируемый спинлок, который основан на явной очереди потоков.

Идея:

Вспомним про устройство Ticket Lock-а:

В Ticket Lock-е потоки выстраивались в очередь, получая от атомарного счетчика последовательные номерки. Очередь была "виртуальной", в спинлоке хранилась "голова" очереди - owner_ticket, номерок текущего владельца спинлока, и "хвост" этой очереди - next_free_ticket, следующий свободный номерок.

Если подумать, по какому паттерну устроена эта очередь, то это Multiple Producers / Single Consumer - сразу несколько потоков конкурируют друг с другом за инкремент next_free_ticket (т.е. за добавление в хвост очереди), но только один поток передвигает вперед голову (извлекает из очереди), передавая тем самым владение блокировкой следующему потоку в очереди.

В спинлоке используется та же самая идея, но очередь потоков материализуется в совершенно явный односвязный список.

Материализация очереди:

Для начала рассмотрим вспомогательную конструкцию - SP/MC очередь, построенную на односвязном списке и на атомарной операции exchange, пока без привязки к спинлокам и очередям ожидания:

Очередь доступна потокам в виде указателя на хвостовой узел, указатели в списке ориентированы от головы к хвосту.

Вставка двухфазная:

Поток, который хочет добавить собственный узел new_node в очередь, первым шагом "захватывает" текущий хвост списка, подменяя его на свой узел с помощью prev_tail = tail.exchange(new_node), вторым шагом - линкует предыдущий хвост и свой узел: prev_tail->next = new_node.

Выполнив prev_tail = tail.exchange(new_node), поток зафиксировал концы ребра в односвязном списке, но еще не провел ссылку между ними.

Хвост, захваченный с помощью tail.exchange(new_node), больше не виден никакому другому потоку.

Заметим, что список, который достраивается таким образом, может распадаться на несколько разорванных сегментов или даже на отдельные узлы, если потоки засыпают между захватом предыдущего хвоста и его линковкой с новым узлом. Но рано или поздно линки возникают. Впрочем, неизвестно, в каком именно порядке...

Сам паттерн MP/SC не является специфичным для задачи взаимного исключения и естественным образом возникает в других (впрочем, весьма нетривиальных) контекстах. Примеры:

- remote deallocation в многопоточных аллокаторах памяти,
- реализация почтовых ящиков (мэйлбоксов) для входящих сообщений в акторных фреймворках

Очередь ожидания и передача блокировок:

Посмотрим, как применить построенный список для реализации очереди потоков в спинлоке.

Каждый поток, который ждет очереди на спинлоке, будет представлен в списке узлом с двумя полями: next - указатель на следующий узел списка, и owner - флаг, который говорит, владеет

ли спинлоком в данный момент поток-хозяин данного узла или не владеет.

Lock устроен так: поток добавляет свой узел в хвост списка, после чего либо крутится на флаге owner своего узла, ожидая, когда в него передадут владение спинлоком, либо (если список был пуст на момент вставки узла) просто проходит в критическую секцию (поскольку других претендентов нет).

Когда поток выходит из критической секции, то он должен передать владение спинлоком следующему узлу в списке (установить в нем флаг owner), либо обнулить tail, если потоков в очереди больше нет.

Тут есть тонкости с передачей владения:

1) Линковка нового узла происходит не мгновенно. Следующий узел уже может быть “логически” добавлен в список (tail указывает не на текущий узел), при этом физическая ссылка next в него может быть еще не проведена. Тем не менее, после Exchange на tail связь между узлами уже зафиксирована в локальной памяти потоков.

2) Если поток увидел, что он был единственным в списке (tail до сих пор указывает на его гард), то передавать владение не нужно, нужно “занулить” tail. Но между этими шагами мог появиться другой поток и прицепить себя в хвост списка.

Гарды на стеках потоков:

Теперь самый нетривиальный тривиальный момент! Если вы хоть раз в жизни писали односвязный список, то наверняка создавали узлы для него в динамической памяти. Здесь мы поступим иначе: в качестве узлов списка мы будем использовать объекты-гарды на стеке потоков!

Пример:

```
SpinLock spin_lock;

В одном из потоков:
{
    SpinLock::Guard guard(spin_lock);
    ... // критическая секция
}
```

У спинлока не будет публичных методов Lock и Unlock. Вместо этого критическая секция будет начинаться в момент создания гарда и завершаться в момент его разрушения, т.е. при выходе исполнения из скоупа, в котором был создан гард.

Для захвата лока поток создает на стеке объект-гард и передает в его конструктор ссылку на спинлок (в котором хранится указатель tail на конец очереди потоков).

Гард и будет тем узлом, который поток добавит в список! В конструкторе гард прицепит сам себя в конец очереди ожидания спинлока, а в деструкторе - передаст блокировку следующему гарду в очереди.

Список потоков в спинлоке называется **интрузивным** - он не владеет памятью и не контролирует время жизни своих узлов, он лишь связывает внешние по отношению к себе объекты, в которые интегрированы ссылки.

Таким образом, очередь ожидания в спинлоке связывает гарды на стеках разных потоков.

Необычно, что список связывает объекты не из хипа, а на стеках потоков, ведь всем известно, что объекты на стеке быстро разрушаются и не должны торчать наружу.

В данном случае ничего плохого не случится, интуиция здесь такая: время пребывания гарда в списке (т.е. время ожидания и освобождения спинлока, в течение которого с этим гардом могут работать другие потоки) привязано к времени жизни гарда: он

добавляется в список при создании, и уходит из него при разрушении.

Масштабируемость:

Под масштабируемостью будем понимать следующее: при увеличении числа ядер и потоков, претендующих на захват спинлока, пропускная способность спинлока (число критических секций в единицу времени) не падает.

Утверждается, что описанная конструкция спинлока является масштабируемой, поскольку число промахов по кэшу при захвате и освобождении спинлока не будет зависеть от числа ядер в процессоре и числа потоков, соревнующихся за владение спинлоком!

Задание:

1) Придумайте реализацию Unlock и реализуйте спинлок.

2) Объясните, почему захват и освобождение описанного спинлока промахиваются по кэшу лишь константное число раз. Почему он не подвержен проблемам cache ping-pong или thundering herd?

3) Объясните, почему TAS, TATAS и Ticket спинлоки не гарантируют константного числа промахов по кэшу?

Все рассуждения про кэши можно проводить, используя самый простой протокол когерентности - **MSI**.

Для простоты будем полагать, что число ядер в процессоре не меньше, чем число потоков, которые соревнуются за спинлок, и каждый поток бежит на отдельном ядре и никогда не меняет его.
