Линеаризуемость и семантика конкурентных объектов

Как описать класс многопоточных исполнений, которые допускает структура данных?

На поставленный вопрос можно ответить, указав:
1) последовательную спецификацию структуры данных и
2) модель согласованности, которую обеспечивает структура данных.

Последовательная спецификация описывает, как структура данных ведет себя при последовательном выполнении операций.

Модель согласованности говорит о том, какими последовательными историями допускается объяснять конкурентные истории выполнения операций на объекте.

Самая сильная (и самая естественная) модель согласованности - **линеаризуемость.** Синоним линеаризуемости - **атомарность.**

Линеаризуемость требует, чтобы любую конкурентную историю можно было объяснить последовательной историей

1) в которой сохранен **порядок всех неконкурентных вызовов** (в частности - порядок вызовов в каждом отдельном потоке, так что из линеаризуемости автоматически следует последовательная согласованность).

2) которая соответствует **последовательной спецификации** структуры данных (set, register, queue, stack и т.п.)

Если в исходной истории один вызов завершился до начала другого вызова (и программа могла наблюдать этот порядок, потому что упорядочила эти вызовы через happens-before), то в последовательной истории, которой мы объясняем исходное конкурентное исполнение, эти два вызова должны сохранить относительный порядок.

В то же время линеаризуемость не фиксирует порядок применения конкурирующих операций. Если программа не упорядочила два вызова, то линеаризуемая структура данных может применять их в произвольном порядке.

Например, если вставка элемента конкурировала с поиском этого элемента, то структура данных может применить поиск как после вставки, так и до нее. Внешний наблюдатель не упорядочил вызовы, а значит не может рассчитывать на какой-то конкретный порядок их исполнения.

Линеаризуемость позволяет рассуждать о конкурентных исполнениях не на уровне отдельных обращений к памяти, а на уровне сложных составных операций.

Точки линеаризации - метод доказательства линеаризуемости.

Точки линеаризации для вызовов операций над объектом - это такие атомарные операции, порядок исполнения которых определяет (отражает) порядок применения операций к конкурентному объекту.

Точки линеаризации для каждого метода должны быть выбраны так, чтобы порядок их исполнения автоматически задавал последовательную историю: исполнив в том же порядке сами вызовы, мы должны получить те же самые результаты (конечно же, согласованные с последовательной спецификацией).

Точки линеаризации должны быть выбраны между началом и завершением любого вызова операции, тогда результат “стягивания” вызовов в точки линеаризации не будет нарушать частичный порядок неконкурентных вызовов в исходной истории.

В общем случае точка линеаризации вызова не обязательно должна находиться в том потоке, в котором выполнялся вызов.

Чтобы доказать, что объект линеаризуем, нужно предъявить алгоритм линеаризации (в частном случае - выбрать и обосновать точки линеаризации для каждого вызова) для произвольной истории исполнения.

Чтобы доказать, что объект нелинеаризуем, нужно привести пример конкурентной истории, для которой нельзя построить линеаризацию, согласованную с последовательной спецификацией объекта.

A. Распределенный счетчик - 2+3 балла

Рассмотрим реализацию монотонно растущего счетчика, значение которого распределено по нескольким атомарным ячейкам памяти для параллельного применения приращений.

Пусть со счетчиком работают n потоков, пронумерованных от 0 до n-1.

Счетчик представляем массивом V[1..n], который изначально заполнен нулями.

Счетчик поддерживает две операции:

- **Add(x):** поток с номером t прибавляет x к значению ячейки V[t]

- **Get()**: поток последовательно считывает и суммирует значения всех ячеек массива V

1) Будет ли такая реализация линеаризуемой? Сможет ли наблюдатель / программа отличить такой счетчик от атомарного?

2) Будет ли счетчик линеаризуемым, если заменить операцию **Add(x)** на операцию **Increment()**, которая увеличивает V[t] на +1?

Подсказка: в этой задаче придется придумать явный алгоритм построения линеаризуемой истории.

A′. Слабый и сильный счетчик - 1 балл

Описанный выше счетчик назовем **слабым** (weak), поскольку операция Increment не возвращает значения счетчика. Чтобы получить значение, нужно отдельно позвать Get.

Назовем счетчик **сильным** (strong), если он может атомарно выполнить инкремент и вернуть прежнее (или новое, не суть) значение, т.е. его публичный контракт состоит из единственного метода GetAndIncrement.

Можно ли доработать описанный выше слабый счетчик до сильного, используя только операции чтения и записи?

B. Очередь - 2 балла

Рассмотрим очередь для нескольких продьюсеров и одного консьюмера:

```
void Queue::Initialize() {
    dummy = new node();
```

```
    head = tail = dummy;
}

void Queue::Enqueue(T item) {
    new_tail = new Node(item);
    prev_tail = tail.exchange(new_tail);
    prev_tail->next = new_tail;
}

bool Queue::Dequeue(T& item) {
    if (!head->next) {
        return false;
    }
    item = head->next->item;
    head = head->next;
    return true;
}
```

Очередь представляет из себя односвязный список с фиктивным узлом в начале. Фиктивный узел позволяет избежать неудобного частного случая при работе с пустой очередью.

Добавление узла выполняется в два шага: первым шагом с помощью атомарного exchange поток “перекидывает” tail на новый узел и “захватывает” указатель на прежний хвостовой узел (prev_tail), вторым шагом - провязывает ссылкой прежний и новый хвост.

Добавлять новые узлы могут сразу несколько потоков.

Извлекать узлы из очереди разрешается только одному потоку.

При извлечении поток проверяет, есть ли за первым фиктивным узлом еще один узел (читает head->next). Если есть, то поток возвращает элемент из этого узла и переставляет на него указатель head (таким образом, этот узел становится новой

фиктивной головой очереди). Идея здесь та же самая, что и в очереди Майкла-Скотта.

Является ли описанная очередь линеаризуемой? Если да, то предъявите точки линеаризации каждого метода. Если нет, то приведите пример нелинеаризуемой конкурентной истории.

C. Поиск без блокировок в оптимистичном сортированном списке - 2 балла

Вспомним реализацию односвязного сортированного списка с оптимистичными блокировками и маркерами на узлах:

См статью. **A Lazy Concurrent List-Based Set Algorithm / Heller, Herlihy, ...**

Поиск в таком списке (операция Contains) работает без блокировок, двигаясь вперед по ссылкам ->next до тех пор, пока не найдет узел, в котором ключ не меньше искомого.

Пусть поиск остановился в узле, ключ в котором равен искомому ключу. В таком случае результат вызова определяется значением флага marked.

Можно ли в таком случае выбрать чтение флага marked в качестве точки линеаризации вызова Contains?

D. Точка линеаризации для Enqueue в очереди Майкла-Скотта - 2 балла

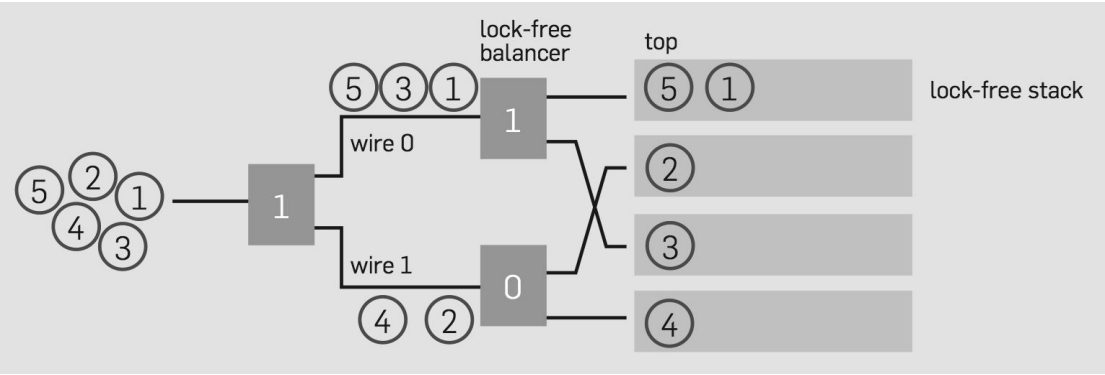
Можно ли в качестве точки линеаризации для вызова Enqueue в очереди Майкла-Скотта выбрать тот успешный CAS, который передвигает указатель tail на вставленный узел?

Заметим, что его не обязательно выполняет тот же поток, который выполняет Enqueue.

Вопрос на понимание: почему точка линеаризации должна быть выбрана между началом и завершением вызова операции?

E. Стек с балансирующим деревом - 2 балла

Покажите, так именно возникает нелинеаризуемая (с точки зрения последовательной спецификации стека - LIFO) история в “стеке” с балансирующим деревом.



F. Точки линеаризации для очереди Herlihy-Wing - 2 балла

В этой задаче вам нужно показать, что ни один из двух шагов в методе Enqueue (первый шаг - инкремент tail, второй - запись значения в выбранный слот массива) в очереди Herlihy-Wing не может быть наперед выбран точкой линеаризации.

Иными словами, ни порядок инкрементов tail-а, ни порядок записей в массив не определяет логический порядок применения вставок в очередь.

Тем не менее, очередь Herlihy-Wing является линеаризуемой! К сожалению, доказательство не умещается на полях этой задачи...

Мораль задачи:

1) Не всегда можно доказать линеаризуемость объекта с помощью точек линеаризации.

2) Невозможность указать точки линеаризации не означает, что объект не является линеаризуемым. Мир устроен сложнее ;)

G. Тестирование линеаризуемости - 2 балла

Рассмотрим счетчик с операциями Add(x) и Get().

Существует ли полиномиальный (по числу вызовов и ширине машинного слова) алгоритм, который получает на вход произвольную конкурентную историю такого счетчика и проверяет её на] линеаризуемость?
