Вся информация, приведённая ниже, является лишь отражением собственного понимания вопросов и сформирована на записях, сделанных во время пар, и данных, доступных в Интернете. Данный документ является лишь обобщением всего изученного на протяжении 3-го семестра, поэтому некоторые вопросы могут быть раскрыты не на 100%, а также могут быть неточности.

Практикум на ЭВМ. Билеты. Осень 2020/2021 уч. г.

1. Таксономия Флинна и типы параллелизма. Процессы, потоки, файберы. Диспетчеризация в операционных системах.

Параллельная программа – программа, отдельные части которой **могут** (необязательно будут, должна быть физическая возможность в виде доступности оперативной памяти, потоков в системе, необходимого количества ядер процессора) исполняться **одновременно** (не требуется ничего извне для этого) и **независимо** (нет зависимости по данным между частями программы) друг от друга.

Конкурентное программирование – вид программирования приложений, конкурирующих за одни и те же вычислительные ресурсы.

Рассмотрим **виды параллелизма**.

1. **Битовый параллелизм**.
   Вид параллелизма, при котором операции над данными выполняются не по одному биту, а по нескольким сразу, то есть **блоками**.
2. **Параллелизм инструкций**.
   Этапы работы с инструкцией, которые могут происходить в один момент времени:

1. Считывание (fetch)
2. Декодирование (decode)
3. Загрузка аргументов (arguments loading)
4. Исполнение (execute)
5. Сохранение аргументов (arguments saving)

   Параллельность состоит в **конвейеризации** работы с инструкциями.

3. **Параллелизм данных**.
   Вид параллелизма, означающий выполнение одной и той же задачи на разных компонентах одних и тех же данных. Он фокусируется на распределении данных между разными вычислительными узлами, которые работают с данными параллельно. Его можно применять к обычным структурам данных, таким как массивы, так и более сложным, как списки, очереди, стеки, хеш-таблицы, деревья, **работая**

с каждым элементом параллельно. Например, задание параллельной обработки данных в массиве из n элементов может быть разделено поровну между всеми n процессорами.
Это контрастирует с параллелизмом задач как другой формой параллелизма.

4. **Параллелизм задач**.
Программист разделяет программу на параллельные задачи. Параллелизм задач фокусируется на **распределении задач**, одновременно выполняемых процессами или потоками, **между разными процессорами**. Для наглядности, можно построить граф зависимостей частей программы и определить, какие из них могут исполняться параллельно, а какие – нет. В отличие от параллелизма данных, который предполагает выполнение одной и той же задачи на разных компонентах одних и тех же данных, параллелизм задач предполагает одновременное выполнение множества разных задач с одними и теми же данными.

Рассмотрим **Таксономию Флинна**.

Классификация по Флинну

	Одиночный поток команд (single instruction)	Множество потоков команд (multiple instruction)
Одиночный поток данных (single data)	SISD (ОКОД)	MISD (МКОД)
Множество потоков данных (multiple data)	SIMD (ОКМД)	MIMD (МКМД)

Архитектура SISD — это традиционный компьютер Фон-Неймановской архитектуры с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. В данном классе **не используется параллелизм ни данных, ни инструкций**.

В машинах с **архитектурой SIMD** один процессор загружает одну инструкцию, набор данных, и выполняет операцию, описанную в этой инструкции, над всем набором данных одновременно. Здесь возникает **параллелизм данных**. Стоит отметить, что одноядерные машины с процессорами, имеющими SIMD-архитектуру, были улучшены компанией Intel до свойств архитектуры MIMD с помощью технологии **Hyper Threading**.

Hyper Threading — это технология для эффективного использования ресурсов ядер процессора (CPU), в первую очередь процессорного времени, позволяя одновременно обрабатывать несколько потоков на одно ядро.

К **архитектуре MISD** относят конвейерные ЭВМ, однако они встречаются редко.

Машины с **архитектурой MIMD** включает в себя многопроцессорные системы, где процессоры обрабатывают множественные потоки данных. Сюда принято относить традиционные мультипроцессорные машины, многоядерные и многопоточные процессоры, а также компьютерные кластеры. Это самая популярная на сегодняшний день архитектура.

1. **Процесс (process)** – характеризуется собственным адресным пространством, ресурсами, это **минимальная единица запуска программ**. На одном компьютере можно сделать параллелизм процессов. Например, при помощи MPI.
    a. **Поток (thread)** – часть процесса. В процессе выделяется главный поток и из него уже - дополнительные потоки. Поток – это **минимальный диспетчеризуемый элемент ОС**. Потоки используют адресное пространство процесса. Чаще ОС оперирует потоками, чем процессами.
        i. **Файбер (fiber)** – минимальный элемент во всей этой вложенной структуре. Любой поток можно преобразовать в набор файберов, диспетчеризацией которых занимается программист.

Диспетчеризация представляет собой распределение процессорного времени между процессами. Целью проведения диспетчеризации является максимальная загрузка процессора, чтобы он не простаивал. В составе операционной системы имеется планировщик, который представляет собой отдельный компонент. Целью деятельности данного компонента является выбор одного или же нескольких загруженных в память и готовых к выполнению процессов и выделение процессора для их исполнения.

2. Закон Амдала. Закон Густафсона-Барсиса.

1. Закон Амдала.

Иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей. В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения **самого медленного фрагмента**.

Согласно этому закону, **ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций**.
В законе Амдала объем работы фиксирован.

Рассмотрим некоторую программу с точки зрения этого закона.

Части программы:

0 < **s** < 1 – доля программы, которая может быть выполнена строго **последовательно**.

1 – s - доля программы, которая может быть выполнена **параллельно** без ограничений.

Время работы:

p – количество вычислительных узлов.

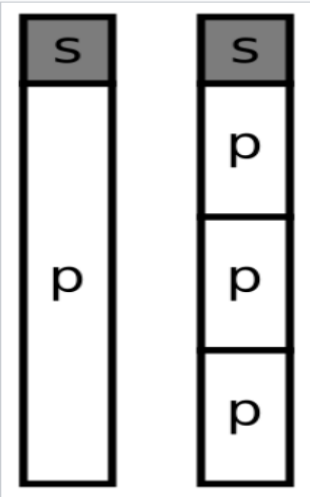
T1 = s * T1 + (1 - s) * T1 – время работы на 1 ядре.

Tp = s * T1 + ((1 - s) * T1) / p – время работы на p ядрах.

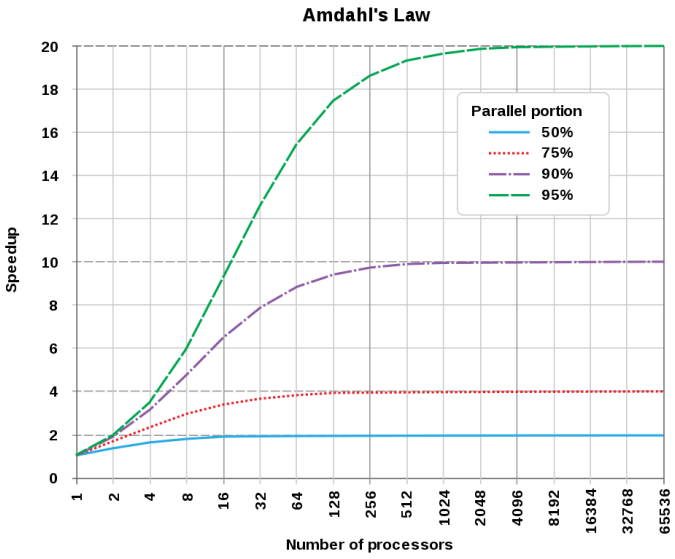
Ускорение:

S(p) = (s * T1 + (1 - s) * T1) / (s * T1 + ((1 - s) * T1) / p)) = 1 / (s + (1 - s) / p) – то есть отношение времени исполнения на 1-м ядре ко времени исполнения на p ядрах даёт нам **ускорение работы на p ядрах относительно 1-го ядра**, которое зависит только размеров последовательной и параллельной частей.

Величина ускорения по закону Амдала показывает, **во сколько раз меньше времени потребуется параллельной программе для выполнения**.



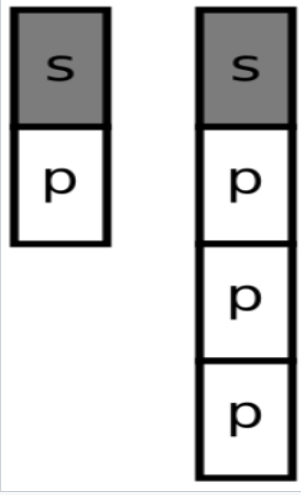
Закон Амдала предполагает, что объем задачи остается неизменным. Задача слева обрабатывается одним потоком, задача справа — тремя. Сравнивается время выполнения.



Ускорение программы с помощью параллельных вычислений на нескольких процессорах ограничено размером последовательной части программы. Например, если можно распараллелить 95% программы, то теоретически максимальное ускорение будет 20-кратным, невзирая на то, сколько процессоров используется.

Предел ускорения на бесконечном числе процессоров – **константа**.

2. Закон Густафсона-Барсиса.



Закон Густафсона предполагает, что время, отведенное на выполнение задачи, остается неизменным. Слева работает один поток, справа — три. Объем выполненной работы увеличен. Сравнивается объем вычислений.

В законе Густафсона-Барсиса время работы фиксировано.

Части программы:

0 < **s** < 1 – доля программы, которая может быть выполнена строго **последовательно**.

1 – s - доля программы, которая может быть выполнена **параллельно** без ограничений.

Объем работы:

p – количество вычислительных узлов.

V1 = s * V1 + (1 – s) * V1 – объем работы на 1 ядре.

Vp = s * V1 + p * (1 - s) * V1– объем работы на p ядрах.

Ускорение:

S(p) = (s * V1 + (1 – s) * V1) / (s * V1 + p * (1 - s) * V1) = s + p * (1 - s) – то есть отношение объёма работы на 1-м ядре к объёму работы на p ядрах даёт нам **ускорение масштабирования на p ядрах относительно 1-го ядра**, которое зависит только размеров последовательной и параллельной частей.

Величина ускорения по закону Густафсона-Барсиса показывает, **насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач**.

3. Операция merge-split. Распараллеливание пузырьковой сортировки.

2. Merge-split.

Рассмотрим работу данной операции на примере 2-ух процессоров.

6 7 1 2 4 5 3 8 – начальный массив данных.

Делим массив на 2 части, затем каждая сортируется на отдельном процессоре.

1 2 6 7 | 3 4 5 8 – распределение чисел по узлам после первого шага.

Далее процессоры пересылают друг другу части массивов: «старшая» половина первого массива и «младшая» половина второго массива меняются местами и также сортируются на своих процессорах.

1 2 3 4 | 5 6 7 8 – распределение чисел по узлам после второго шага.

Далее происходит слияние двух массивов с разных узлов в один.

1 2 3 4 5 6 7 8 - распределение чисел по узлам после последнего шага.

Если в левой части любой элемент меньше любого элемента в правой, то массив отсортирован.

3. Чётная-нечётная сортировка.

p – количество вычислительных узлов.

Таблица 9.5. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Вычислительные элементы			
	0	1	2	3
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
0 чет (1,2),(3.4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
1 нечет (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
2 чет (1,2),(3.4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
3 нечет (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Рассмотрим **схему работы алгоритма**.

Каждая итерация является либо **чётной**, либо **нечётной**.

На **чётных** итерациях логически объединяются в пары и сортируются между собой элементы, находящиеся на чётных узлах

(0 считаем чётным), и следующие за ними, то есть нечётные, узлы путём перекидывания всех элементов на любой из узлов, их сортировки и обратным распределением отсортированных элементов.
На **нечётных** итерациях логически объединяются в пары и сортируются между собой элементы, находящиеся на нечётных узлах, и следующие за ними, то есть чётные, узлы путём перекидывания всех элементов на любой из узлов, их сортировки и обратным распределением отсортированных элементов.
Если количество узлов оказывается нечётным, это **никак не сказывается на алгоритме**, так как в этом случае «лишний» узел будет простаивать на чётной итерации, но будет сортироваться на нечётной.

Подмассивы на каждой итерации можно сортировать каким угодно способом. Всего для работы сортировке требуется **p итераций**.

Заметим, однако, что при выполнении каждой итерации этого алгоритма, **задействованными оказываются только p/2 вычислительных процессоров**, в то время как остальные вычислительные процессоры простаивают. Это приводит к снижению общей эффективности параллельного алгоритма.

4. Распараллеливание quicksort.

1. Базовый алгоритм:

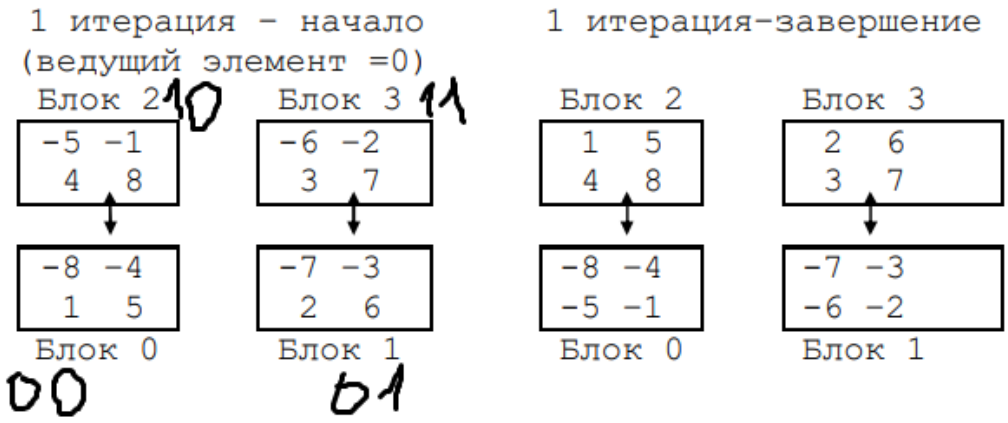
Рассмотрим **схему работы алгоритма**.

Есть **p = 2** процессоров и массив из **n = 16** элементов, который мы разделяем на $\mathbf{N_{бл} = 2p = 4}$ **блоков** – число блоков. Таким образом, на каждом процессоре оказывается **по 2 блока**. Распределение по элементам внутри блоков может быть любым. Каждый блок должен иметь свой **индекс блока** в двоичной СС (00, 01, 10, 11).

$\mathbf{N_{ит}= \log_2 N_{бл} = 2}$ – количество итераций в алгоритме, так как в алгоритме мы каждый раз **делим на 2**.

Далее выбираем любой **опорный элемент**, по которому мы разделяем значения на большие, и меньшие него, и которые попадают в разные части гиперкуба. Причём не обязательно из элементов массива. Если выбирать данный элемент плохо, то **эффективность сортировки снизится**. В данном случае это **0**.

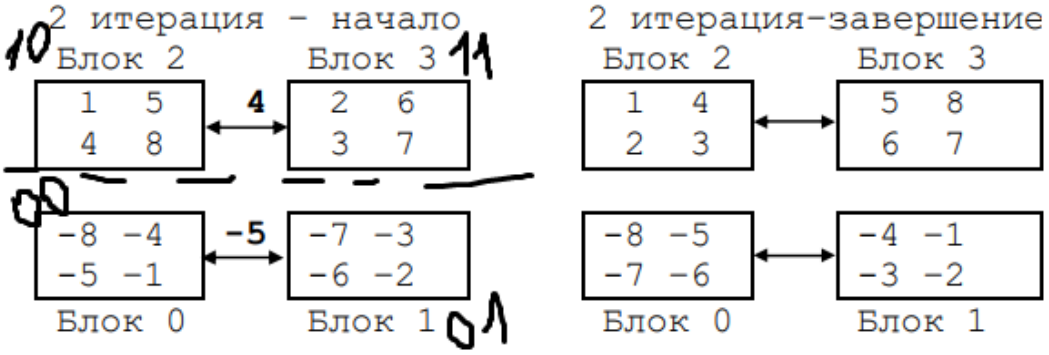
Операции merge-split на **i-ой итерации** подвергаются пары блоков, индекс которых совпадает, за исключением **одного бита**, находящегося в позиции **(кол-во итераций – i)**. После сортировки на одном из узлов, элементы распределяются обратно по блокам относительно опорного элемента. Разряды нумеруются с 0, а итерации – с 1.



Под стрелкой понимаем операцию **merge-split**. Все элементы после неё в блоках 2 и 3 получились больше **0 – опорного элемента**, а в блоках 0 и 1 - меньше него.

Теперь можем разделить этот гиперкуб плоскостью на 2 части, и сортировать теперь по половинкам.

На следующей итерации снова выбираем **опорный элемент в каждой половинке** и по тому же правилу выбираем **блоки** и делаем **merge-split**.



Необходимо заметить, что перехода данных после очередной итерации между плоскостями-разделителями не происходит, чем мы вносим **отношение порядка между отдельными частями исходного массива**.

Если бы блоков было больше, то рассекли бы ещё раз плоскостью, в итоге было бы 4 части, и дальше делали бы то же самое.

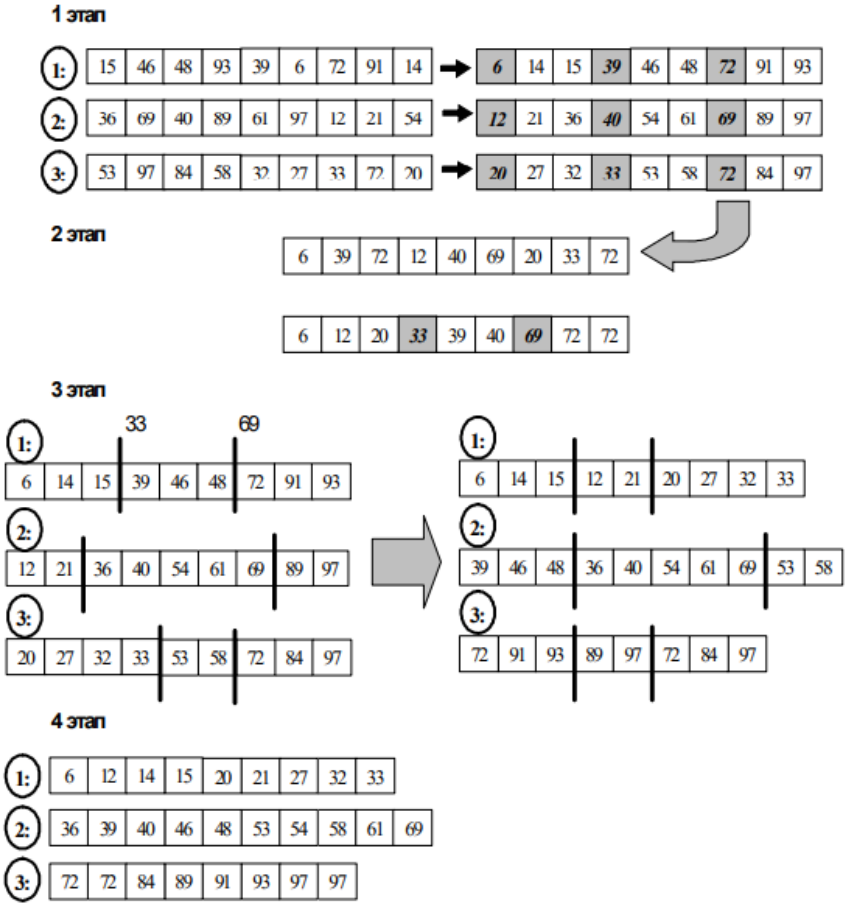
Делим плоскостями до тех пор, пока не рассечём весь гиперкуб так, чтобы **в каждой его части остался один блок**. Тогда **массив отсортирован, если слить блоки по возрастанию/убыванию их индексов**.

2. **Быстрая сортировка с использованием регулярного набора образцов:**
   Есть **p** процессоров и массив из **n** элементов.

   Делим массив на **p блоков**, каждый размером **n/p**, возможно, за исключением последнего блока, если не делится нацело.

   1. Каждый процессор сортирует свой блок любой сортировкой. Далее каждый процессор **формирует набор** из элементов своих блоков с индексами 0, m, 2m, …, (p - 1)m, где **m =** $\frac{n}{p^2}$**.**
   2. Все сформированные процессорами наборы данных **собираются на одном из потоков** (**master thread**) системы и сортируются при помощи быстрого алгоритма, таким образом они **формируют упорядоченное множество**. Далее из полученного множества значений из элементов с индексами $p+\lfloor p/2\rfloor-1,\ \ 2p+\lfloor p/2\rfloor-1,\ldots,(p-1)p+\lfloor p/2\rfloor$ **формируется новый набор ведущих элементов**, который далее используется всеми процессорами. В завершение этапа каждый процессор выполняет **разделение своего блока на p частей с использованием полученного набора ведущих значений**.
   3. На третьем этапе сортировки каждый процессор осуществляет **передачу** выделенных ранее частей своего блока всем остальным процессорам; «передача» выполняется в соответствии с порядком нумерации – часть блока с **номером j**, **0 < j < p**, передаётся процессору с **номером j**.
   4. На четвёртом этапе выполнения алгоритма каждый процессор выполняет **слияние p полученных частей в один блок** и сортирует его.
   5. На последнем этапе **сливаем все блоки в один в нужном порядке**, на чём завершаем сортировку.

Пример:



5. Параллельный алгоритм Флойда.

Алгоритм Флойда применяется для нахождения **кратчайших расстояний между всеми вершинами взвешенного ориентированного графа**.

8. Последовательный алгоритм.

n – число вершин.

A – матрица смежности графа, в которой хранятся длины путей для всех пар вершин.

```
for (k = 0; k < n; k++) // число вершин
    for (i = 0; i < n; i++) // проход по строкам матрицы
        for (j = 0; j < n; j++) // проход по столбцам матрицы
            A[i, j] = min(A[i, j], A[i, k] + A[k, j]); // выбираем наименьшую длину между старым значением и новым
```

Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A

будет храниться требуемый результат - **длины минимальных путей для каждой пары вершин исходного графа**.

Сложность = O(n^3).

9. Параллельный алгоритм.

10.1.2. Разделение вычислений на независимые части

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимального значения (см. Алгоритм 10.1). Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A.

Покажем корректность такого способа организации параллелизма. Для этого нужно доказать, что операции обновления значений матрицы A на одной и той же итерации внешнего цикла k могут выполняться независимо. Иными словами, следует показать, что на итерации k не происходит изменения элементов A_{ik} и A_{kj} ни для одной пары индексов (i,j). Рассмотрим выражение, по которому происходит изменение элементов матрицы A:

$$A_{ij} \leftarrow \min (A_{ij}, A_{ik} + A_{kj}).$$

Для $i=k$ получим

$$A_{kj} \leftarrow \min (A_{kj}, A_{kk} + A_{kj}),$$

но тогда значение A_{kj} не изменится, т.к. $A_{kk}=0$.

Для $j=k$ выражение преобразуется к виду

$$A_{ik} \leftarrow \min (A_{ik}, A_{ik} + A_{kk}),$$

что также показывает неизменность значений A_{ik}. Как результат, необходимые условия для организации параллельных вычислений обеспечены и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы A (для указания подзадач будем использовать индексы обновляемых в подзадачах элементов).

Возможный способ распараллеливания вычислений состоит в использовании **ленточной схемы разбиения матрицы A**. Это означает, что можно разбить матрицу по **столбцам** (вертикальное разбиение) или по **строкам** (горизонтальное разбиение), чтобы обновлять значения в матрице A одновременно, и, как было доказано, независимо друг от друга.

6. Параллельный алгоритм Прима.

Алгоритм поиска **минимального остовного дерева на связном графе**, то есть состоящем из одной компоненты сильной связности. **Остовное дерево** — это ациклический подграф данного графа, с тем же числом вершин, что и у исходного графа. Остовное дерево получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Остовное дерево включает в себя все **n вершин исходного графа и содержит n - 1 ребро**.

1. Последовательный алгоритм.

1. **Выбираем** вершину.
2. На каждой итерации формируем список рёбер, которые выходят из **рассмотренных вершин** (включая выбранную в данный момент вершину) в **нерассмотренные** ранее вершины.
3. Из этого множества выбираем ребро **минимального веса**, помечаем вершину рассмотренной.
4. Повторяем операции 1 - 3 до тех пор, **пока не будут рассмотрены все вершины**.

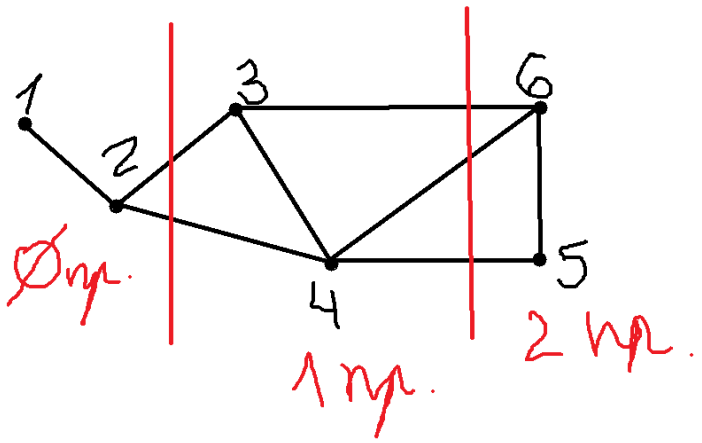
Сложность: O(n^2)

1. Параллельный алгоритм.

Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены. С другой стороны, выполняемые на каждой итерации алгоритма действия **являются независимыми и могут реализовываться одновременно**.

Рассмотрим **схему работы алгоритма**.

Распределяем все вершины по процессорам. Если неудачно провести распределение, алгоритм в худшем случае может выразиться **почти в последовательный**.



Корневым элементом дерева выбираем любую вершину. В нашем случае это вершина **6**.

Итерация 1:

Рассылаем выбранную вершину как начальную на все процессоры и составляем для каждого процессора **список смежных с корневой**

вершиной ребер, **если корневая вершина находится на данном процессоре.** Выбираем только те рёбра, которые идут из **рассмотренных вершин** (включая выбранную в данный момент вершину) **в нерассмотренные**.

Процессоры	0	1	2
Рассматриваемые вершины	6	6	6
Ребра	Пустое множество	Пустое множество	6-3, 6-4, **6-5**

Далее **ищем ребро минимального веса**. Например, это **6-5**.

Итерация 2:

Теперь **рассылаем новую корневую вершину**, **к которой ведёт найденное ребро**, (в нашем случае это **5**) всем остальным процессорам во множество начальных и снова производим построение **списка смежных с корневой вершиной ребер**, **если корневая вершина находится на данном процессоре.** Выбираем только те рёбра, которые идут из **рассмотренных вершин** (включая выбранную в данный момент вершину) **в нерассмотренные**.

Процессоры	0	1	2
Рассматриваемые вершины	6, 5	6, 5	6, 5
Ребра	Пустое множество	Пустое множество	6-3, 6-4, **5-4**

Важно: на 3 процессоре ребра **6-5 больше нет**, потому что мы рассматриваем ребра из **рассмотренных вершин в нерассмотренные**.

Аналогично находим **минимальное ребро**. Пусть это будет ребро **5-4**.

Итерация 3:

Теперь **рассылаем новую корневую вершину**, **к которой ведёт найденное ребро**, (в нашем случае это **4**) всем остальным процессорам во множество начальных и снова производим

построение **списка смежных с корневой вершиной ребер**, **если корневая вершина находится на данном процессоре.** Выбираем только те рёбра, которые идут из **рассмотренных вершин** (включая выбранную в данный момент вершину) **в нерассмотренные**.

Процессоры	0	1	2
Рассматриваемые вершины	6, 5, 4	6, 5, 4	6, 5, 4
Ребра	Пустое множество	4-3, 4-2	**6-3**

Важно: на 3 процессоре рёбер **6-4, 5-4 больше нет**, потому что мы рассматриваем ребра из **рассмотренных вершин в нерассмотренные**.

Аналогично находим **минимальное ребро**. Например, это будет ребро **6-3**.

Итерация 4:

Теперь **рассылаем новую корневую вершину**, **к которой ведёт найденное ребро**, (в нашем случае это **3**) всем остальным процессорам во множество начальных и снова производим построение **списка смежных с корневой вершиной ребер**, **если корневая вершина находится на данном процессоре.** Выбираем только те рёбра, которые идут из **рассмотренных вершин** (включая выбранную в данный момент вершину) **в нерассмотренные**.

Процессоры	0	1	2
Рассматриваемые вершины	6, 5, 4, 3	6, 5, 4, 3	6, 5, 4, 3
Ребра	Пустое множество	**3-2**, 4-2	Пустое множество

Важно: на 3 процессоре рёбер **6-3, 4-3 больше нет**, потому что мы рассматриваем ребра из **рассмотренных вершин в нерассмотренные**.

Аналогично находим **минимальное ребро**. Пусть это будет ребро **3-2**.

Итерация 5:

Теперь **рассылаем новую корневую вершину**, **к которой ведёт найденное ребро**, (в нашем случае это **2**) всем остальным процессорам во множество начальных и снова производим построение **списка смежных с корневой вершиной ребер**, **если корневая вершина находится на данном процессоре.** Выбираем только те рёбра, которые идут из **рассмотренных вершин** (включая выбранную в данный момент вершину) **в нерассмотренные**.

Процессоры	1 процессор	2 процессор	3 процессор
Рассматриваемые вершины	6, 5, 4, 3, 2	6, 5, 4, 3, 2	6, 5, 4, 3, 2
Ребра	**2-1**	Пустое множество	Пустое множество

Важно: на 3 процессоре рёбер **3-2, 4-2 больше нет**, потому что мы рассматриваем ребра из **рассмотренных вершин в нерассмотренные**.

Аналогично находим **минимальное ребро**. Например, это будет ребро **2-1**.

Множество всех выбранных на итерациях алгоритма рёбер – и есть **минимальное остовное дерево**.

7. MPI: основные понятия и операции.

MPI – один из высокоуровневых способов организации параллельных вычислений.

Рассмотрим **схему работы MPI**.

Есть множество **вычислительных узлов**, каждому из которых назначается свой **ранг** - порядковый номер узла. Все узлы в рамках MPI являются **равноправными**, то есть они все исполняют одну и ту же программу. Всё отличие исполнения программы на различных узлах обеспечивается **условными операторами**, проверяющими ранг исполняющего узла, и **дополнительными вызовами MPI**.

Существует **узел Supervisor**, занимающийся управлением, мониторингом всех вычислительных узлов, а также сбором всей информации с них. Он запускает одну и ту же программу на всех узлах.

Коммуникатор – группа из вычислительных узлов, **общее пространство узлов**, где они могут обмениваться информацией друг с другом. Можно создавать свои коммуникаторы.

SELF – коммуникатор, в состав которого входит только **один текущий узел**.

WORLD – коммуникатор по умолчанию, **объединяет все узлы**, входящие в вычислительный кластер.

Такую программу следует запускать из **командной строки или отдельным процессом**, указывая число запускаемых узлов с флагом **-n**, так как если запускать программу через **debug**, то она будет исполняться последовательно, то есть будет выбран один вычислительный узел по умолчанию.

Основные операции:

Чтобы программа работала как программа на MPI, нужно сделать вызов **MPI.Environment.Run**, куда передать аргументы и коммуникатор (по умолчанию используется WORLD).

Также необходимо помнить, что одни процессоры могут работать быстрее, а другие – медленнее, здесь также нет никах гарантий относительно скорости их работы.

1. **Send**.
   Отправление информации с вызывающего узла.
   send(a, b, c)
   a – данные.
   b – на какой узел посылаем.
   c – дополнительный тег (тег нужен, чтобы отличать одно сообщение от другого, если оба сообщения отправляет один и тот же узел).
2. **Receive**.
   Получение информации на вызывающий узел.
   receive(a, b)
   a - ранг ожидаемого отправителя.
   b - дополнительный тег.

В базовом виде **send** и **receive** синхронные.

3. **Reduce**.
   Собирает данные на выбранном узле во всех вычислительных узлов и **применяет** к ним выбранную **операцию**.
   reduce(a, b, c)
   a - данные
   b - операция над этими данными
   c - ранг узла, на котором результат этой операции будет агрегирован.
4. **Gather**.
   Собирает данные со всех вычислительных узлов.
   gather(a, b)
   a – данные.
   b – номер узла, на котором собираем эти данные.
5. **Scatter**.
   Распределяет данные между вычислительными узлами.
   scatter(a, b)
   a - данные.
   b - ранг узла, с которого отправляем данные.
   scatter(a)
   a - ранг узла, с которого ожидаем данные.
6. **Barrier**.
   Ставит барьер, означающий, что выполнение программы не пойдёт дальше, **пока все исполнение на всех узлах не достигнет барьера**.

8. Понятие взаимного исключения и критической секции. Deadlock, livelock, голодание. Инверсия приоритетов. Барьер памяти. Задача об обедающих философах.

Критическая секция – фрагмент или фрагменты кода, которые одновременно может исполнять не **более чем 1 поток**. Если поток уже занимается исполнением кода в критической секции, то никакой другой поток в этой секции **находиться не может**. Когда один поток исполняет инструкцию, другие потоки ждут на входе в критическую секцию, образуя **очередь ожидания**. Фактически это можно назвать сериализацией потоков.

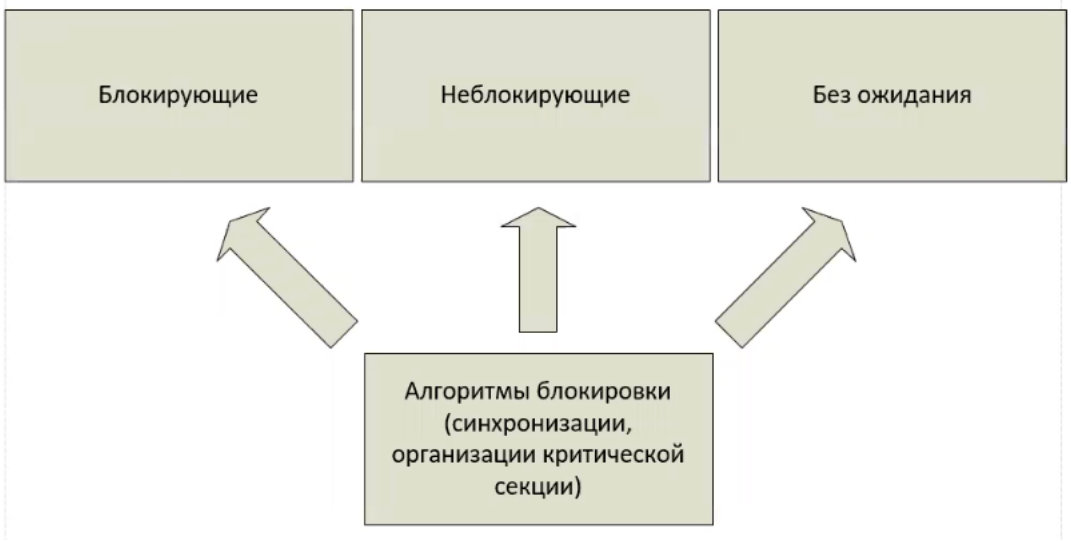
Фрагмент**ы** кода – подразумевается в том смысле, что может быть, например, две функции, где работают несколько потоков, но критическая секция **все равно одна**!

Также критических секций может быть несколько. Может быть так, что один поток будет в двух и более критических секциях одновременно.

Если какой-то поток **забудет открыть критическую** секцию, то возникает **deadlock** – критическая секция **никогда не разблокируется**, другие потоки будут **бесконечно ждать** открытия критической секций.

От примитива синхронизации или синхронизирующего алгоритма мы прежде всего ожидаем свойства **взаимного исключения**, то есть алгоритм не дает пройти в критическую секцию более чем одному потоку сразу.

Классификация алгоритмов блокировок **по времени доступа**:



Другая классификация:

Обе классификации нормально сосуществуют вместе.

Алгоритмы без взаимной блокировки – при доступе в критическую секцию два потока могут блокировать друг друга, но не могут заблокироваться навсегда.

Голодание – каждый поток, запросивший доступ к критической секции, рано или поздно обязательно получит его, кроме «**голодающего**» потока, которому **всегда будет не «везти»**.

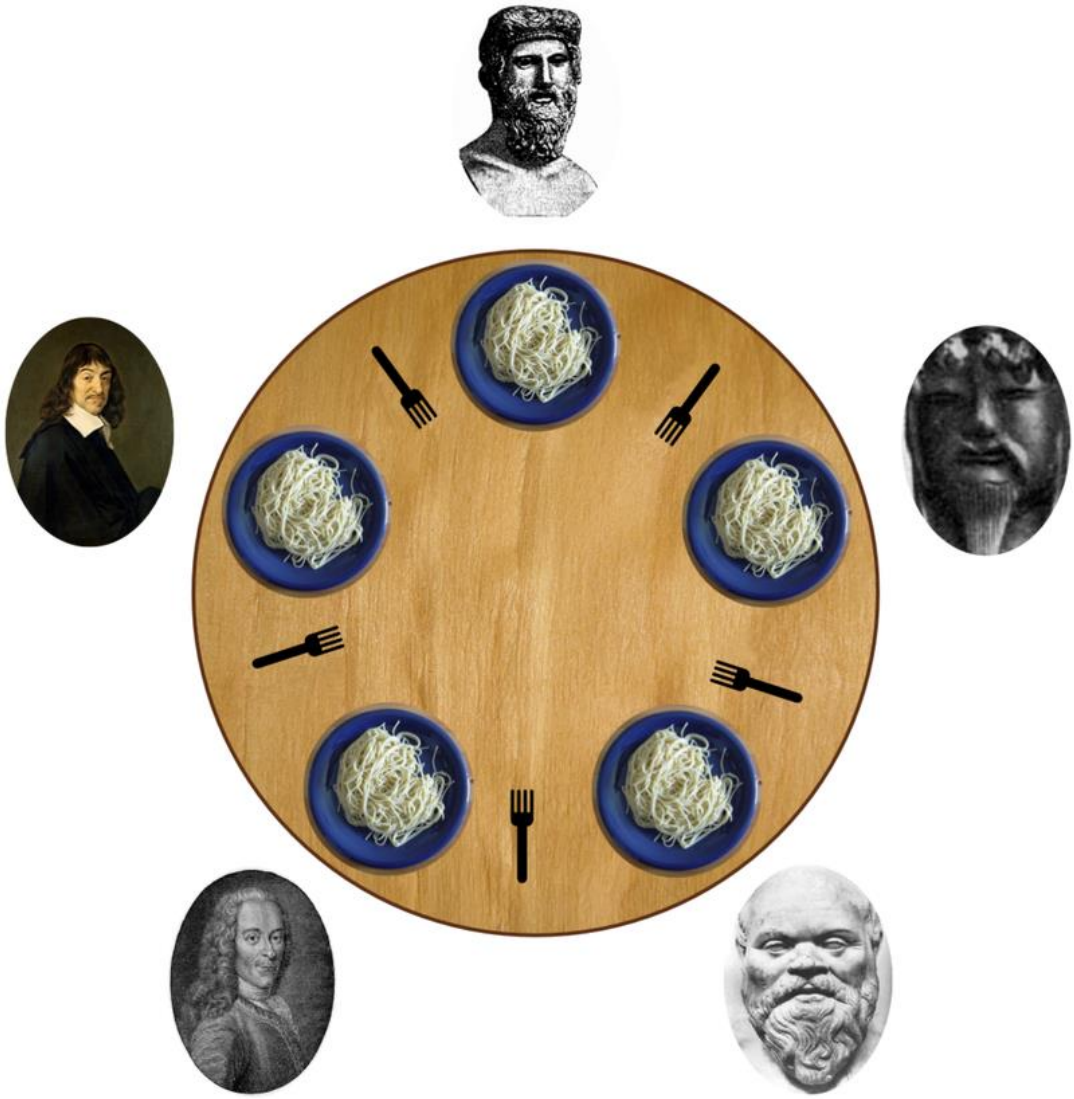
Алгоритм без голодания

Если алгоритм обеспечивает **отсутствие голодания**, то алгоритм гарантирует, что в нём **нет взаимных блокировок**.

Время нахождения потока в критической секции должно быть минимально, иначе планировщик ОС может вытеснить поток.

Задача об обедающих философах.

Задача об обедающих философах — классический пример, используемый в информатике для иллюстрации проблем синхронизации при разработке параллельных алгоритмов.



Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева.

Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим.

Вопрос задачи заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

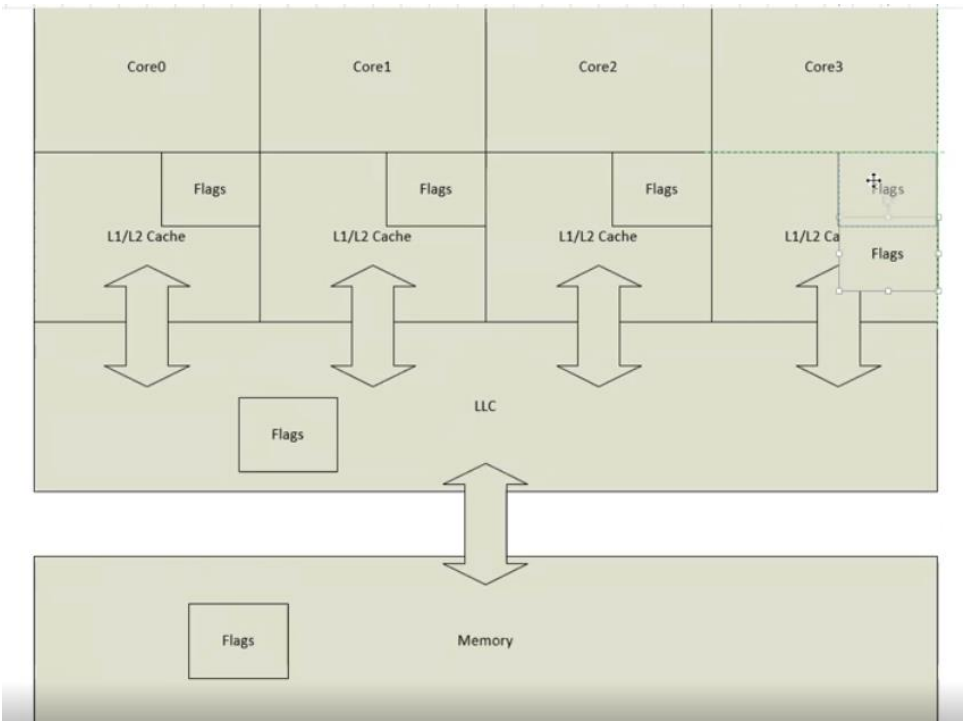
Задача сформулирована таким образом, чтобы иллюстрировать проблему **избежания взаимной блокировки** (англ. deadlock) — состояния системы, при котором **прогресс невозможен**.

Например, можно посоветовать каждому философу выполнять следующий алгоритм:

2. Размышлять, пока не освободится левая вилка. Когда вилка освободится — взять её.
3. Размышлять, пока не освободится правая вилка. Когда вилка освободится — взять её.
4. Есть
5. Положить левую вилку
6. Положить правую вилку
7. Повторить алгоритм сначала

Это решение задачи некорректно: оно позволяет системе достичь состояния взаимной блокировки, когда каждый философ взял вилку слева и ждёт, когда вилка справа освободится.

Барьеры памяти:



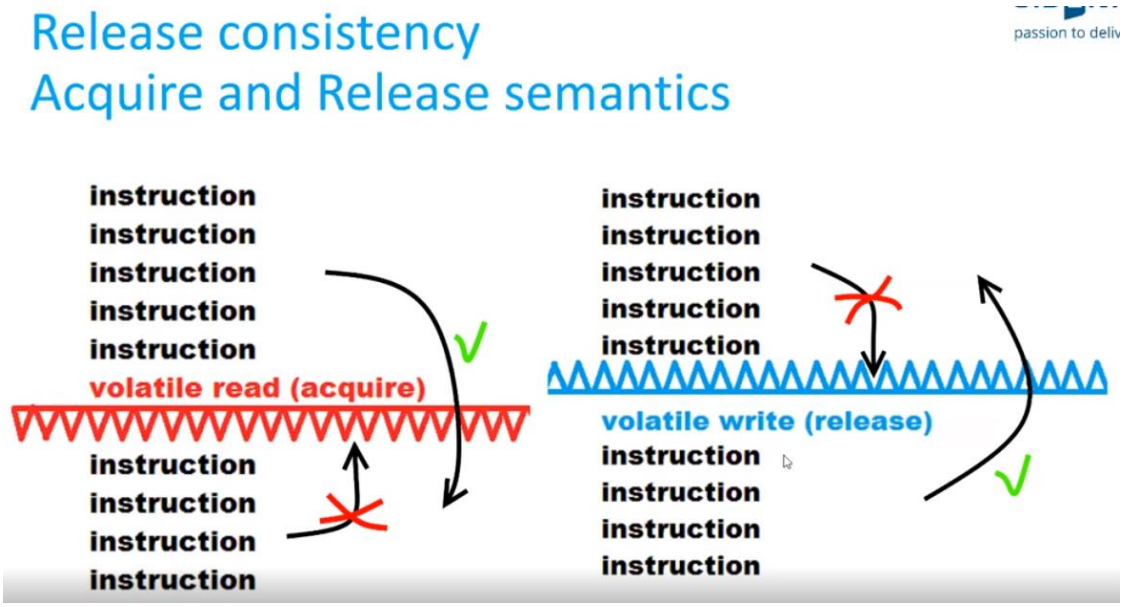
Суть проблемы: если какое-то ядро что-то изменило каких-либо данных, то другое ядро **может не знать об этих изменениях**, так как новое значение находится в кэше другого ядра и недоступно остальным. Таким образом, все другие ядра обладают неактуальным значением.

Это происходит из-за того, что генерируемый компилятором машинный код может не знать, что он работает в многопоточной среде, и применять слишком много оптимизаций. То есть у него нет никакой индикации того, что какие-либо данные являются **областью разделяемой памяти**.

И чтобы сказать компилятору, что **данные являются разделяемыми**, используют ключевое слово **volatile**. Ключевое слово volatile работает с барьерами памяти.

Семантика Aquire/Release:

Компилятор может **переставлять инструкции по порядку их исполнения**, из-за чего поведение программы может измениться.



Acquire (volatile read) - инструкции перед чтением можно перемещать после чтения, но обратно нельзя. Своего рода барьер, пропускающий инструкции дальше по исполнению, но назад нельзя.

Release (volatile write) - инструкции до записи нельзя перемещать за запись, но можно обратно. Своего рода барьер, пропускающий инструкции назад по исполнению, но вперёд нельзя.

volatile = Aquire + Release.

Если обращаемся к области разделяемой памяти внутри критической секции, оформленной с примитивами синхронизации, то **volatile можно не**

использовать, так как **на входе/выходе критической секции барьеры памяти уже есть**.

Если обращаемся к разделяемой памяти за пределами критической секции, то **volatile обязательно должен быть**.

Также **volatile** указывает на необходимость **чтения из источника и записи в источник без кэширования**, то есть изменение данных на одном процессоре приведет к **инвалидации данных** в кэше LLC, и эти измененные данные пометятся на других процессорах как **некорректные** и **будут считаны вновь**. Также это указание компилятору, что **оптимизации для этих данных не нужны**.

9*. Алгоритм Петерсона. Доказательство корректности.

Алгоритм для синхронизации, создания критической секции, но только **двух** потоков.

Интерфейс ILock:

Lock() – пропускает поток, если критическая секция свободна, либо тормозит поток, пока критическая секция занята другим потоком. Если несколько потоков ожидают входа, то выбирается какой-то один.

Unlock() – поток, выходящий из критической секции, сообщает другому потоку об освобождении критической секции.

Получаем **ID потоков**. Создаём **массив флагов**. Если поток входит в критическую секцию, он смотрит, не поднят ли флаг другого потока (значение true), и будет ждать в бесконечном цикле, пока этот флаг не будет опущен. При этом в разблокировке поток просто опускает свой флаг обратно в false.

Если убрать переменную victim, то алгоритм допускает взаимные блокировки. Если оба потока поднимут свои флаги в true, то возникнет **deadlock**, так как никакой поток не успеет войти в критическую секцию и не сделает Unlock(), тем самым не вернет свой флаг в false.

Если убрать массив флагов, то в случае одновременного прихода двух потоков в метод Lock(), один из потоков **точно провалиться в критическую секцию**, так как процессор линеаризует все запросы и всегда будет последний поток, который запишет свой номер в victim. Однако в случае отсутствия конкуренции один единственный поток будет **до бесконечности ждать**.

Если будет и массив флагов и поле victim, то если потоки одновременно установят свои флаги в true, то всё равно только один из них сможет пройти в критическую секцию за счёт проверки **victim == i**, где решается, в каком порядке потоки будут заходить в критическую секцию. Всегда будет возникать “проталкивание” потоков внутрь критической секции.

В таком случае алгоритм гарантирует взаимное исключение, отсутствие взаимной блокировки и отсутствие голодания.

Недостатки алгоритма Петерсона:

1. В данном алгоритме ограниченное число потоков.
2. Преобразование в машинный код зависит от конкретного компилятора.

10*. Алгоритм Лэмпорта. Понятие консенсуса, теорема о консенсусе (без доказательства).

Алгоритм Лэмпорта - предназначен для синхронизации **n** потоков. Общая идея: реализация электронной очереди, где каждому приходящему потоку выдается “талончик”.

Заводим помимо массива флагов, массив талонов. Определяем ID потока - число от **0** до **n - 1** включительно.

В Lock() выставляем для текущего потока флаг в true, и выдаём ему талон с **номером максимального талона + 1**. Условие выбора потока по талону следующее.

1. Нас интересуют все талоны, кроме талона текущего потока, чтобы сравниться с другими потоками.
2. Проверяем, что значения флагов у этих потоков тоже истинна, то есть они также претендуют на вход в критическую секцию.
3. Более приоритетным считается поток, номер талона которого меньше номера талона текущего потока, то есть он пришёл раньше, либо в случае равенства талонов, что может произойти, более приоритетный тот, чей ID меньше ID текущего потока. Здесь фактически происходит арбитраж потоков на вход в критическую секцию.

Как только поток заканчивает работу в критической секции, он сбрасывает в Unlock() свой флаг в false.

Чисто теоретически алгоритм Лэмпорта не свободен от голодания в случае переполнения значений в массиве талонов.

Недостатки алгоритма Лэмпорта:

7. В данном алгоритме ограниченное число потоков.
1. Преобразование в машинный код зависит от конкретного компилятора.

Задача о консенсусе: есть несколько участников, которые могут по какому-то алгоритму предоставить значение **0 или 1**. Если все участники предоставили значение, то **как определить по нему итоговое значение**?

Свести задачу **синхронизации** можно **к задаче о консенсусе**.

Число консенсусов для алгоритма Петерсона равно **2**. Для алгоритма Лэмпорта равно **n**.

11. Понятие блокирующего алгоритма, lock-free, wait-free. Проблема ABA.

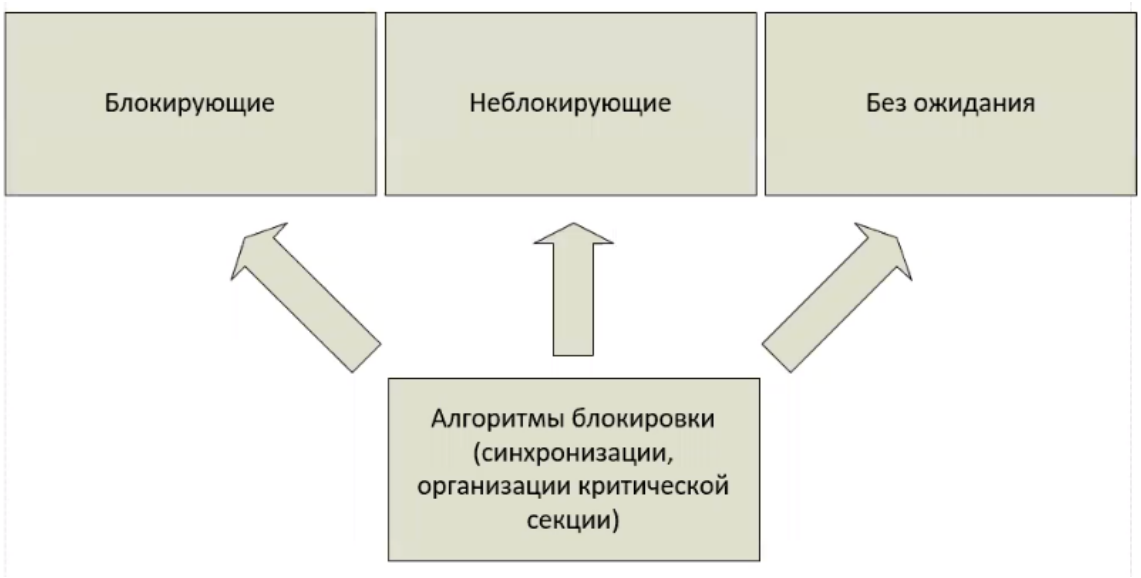
Алгоритм является блокирующим в том случае, если несколько потоков одновременно хотят попасть внутрь критической секции, то внутрь попадёт **только один из этих потоков**, а другие каким-либо образом **приостанавливают свою работу**. Блокирующие алгоритмы – алгоритмы, в которых вход в критическую секцию потенциально может завершаться и **за бесконечное число шагов**, то есть нет гарантии, что мы когда-нибудь можем зайти в критическую секцию, что, естественно может привести программу **в некорректное состояние**.

Неблокирующие алгоритмы – для каких-то, необязательно всех, потоков вход в критическую секцию будет выполняться **за конечное число шагов, а не бесконечное,** и такая ситуация будет повторяться **бесконечное число раз**. Такие алгоритмы построены таким образом, что каким-то потокам будет «**везти**», и они будут получать управление, игнорируя другие потоки. Это **более строгое** требование к алгоритму.

Если **на поток всегда оказывают влияние другие потоки**, то скорее всего алгоритм **блокирующий**, если в каких-то случаях **поток может прорваться в критическую секцию среди своих конкурентов**, то скорее всего **неблокирующий**.

Алгоритмы без ожидания – это класс алгоритмов, в которых потоки заканчиваются всегда, **за конечное число шагов**.

Классификация алгоритмов блокировок **по времени доступа**:



Проблема ABA: поток **сохраняет локально состояние системы во время работы**, затем операционная система **останавливает поток. После возвращения из сна поток не проверяет, изменилось ли состояние системы, и это приводит к проблемам.**

12. Атомарные операции.

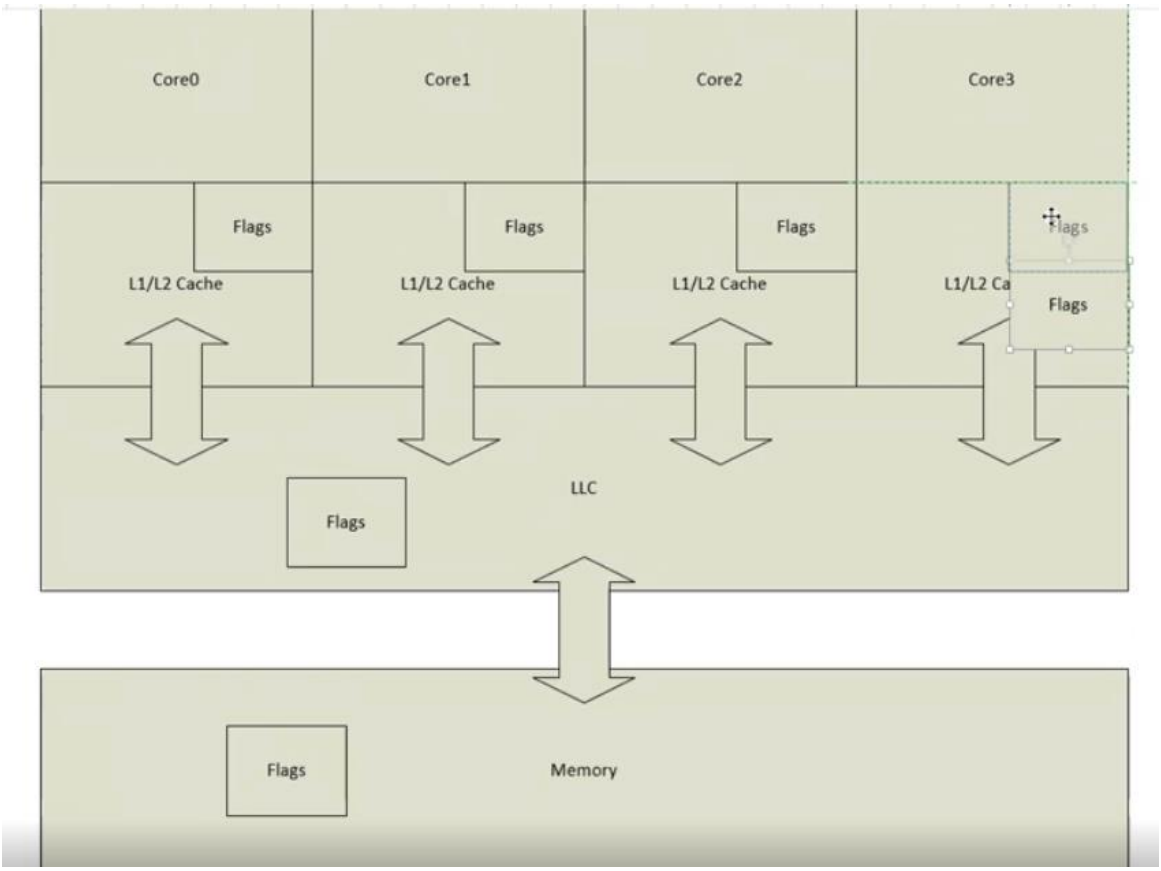
Атомарные операции - операции, которые делают **считывание** данных, **изменение** значения и **запись** обратно как **единую операцию**.

Атомарные операции хороши тем, что на их основе можно построить **алгоритм синхронизации**, алгоритм **разграничения доступа к критической секции**, **обеспечения взаимного исключения**.

Есть **Increment, Decrement, Read, Write** (для **int** - в случае 64-битной, так как 32-битные по умолчанию являются **атомарными**).

Неатомарность операции состоит в том, что проход из памяти до ядра, работа внутри ядра над значением и запись обратно в память - **три разные операции**.

Если **операция атомарна**, то во время, которое ядро тратит, например, на инкрементацию значения, доступа к старому значению в LLC не производится. То есть это значит, что **доступа к этой ячейке памяти в LLC со стороны других ядер не происходит**. Атомарная операция автоматически **инвалидирует** значение кэша в других ядрах и в **лучшем случае** происходит обращение ядра **к LCC, в худшем - к ОЗУ**.



Недостаток: атомарные операции нужно поддерживать на уровне схемотехники ЦПУ, поэтому **не все процессоры поддерживают атомарные операции**.

Для всех алгоритмов синхронизации на атомарных **операциях число консенсусов** ∞!

13*. Понятие Spinlock (спинлок). Примеры реализации спинлоков по принципу Test-And-Set.

Алгоритм TASLock:

Алгоритм синхронизации с использованием **атомарной операции compare-exchange. Test-and-Set/compare-and-set/compare-exchange -** всё одна и та же операция, но с разной реализацией в разных языках. **CompareExchange(ref a, b, c);**

ref a - ссылка на область памяти, к которой **мы получаем эксклюзивный доступ на время атомарной операции.**

b- значение, которое **мы перезапишем**, если в памяти действительно лежит **c.**

c- значение, которое мы **хотим увидеть** в этой области памяти.

Операция **смотрит значение в той или иной ячейке памяти**, если оно **совпадает с ожидаемым значением (c)**, то **подменяет старое значение на новое (b)**; если **совпадения нет**, то **не делает ничего.**

Возвращает **старое значение в памяти (a),** которое было ещё **до начала операции**.

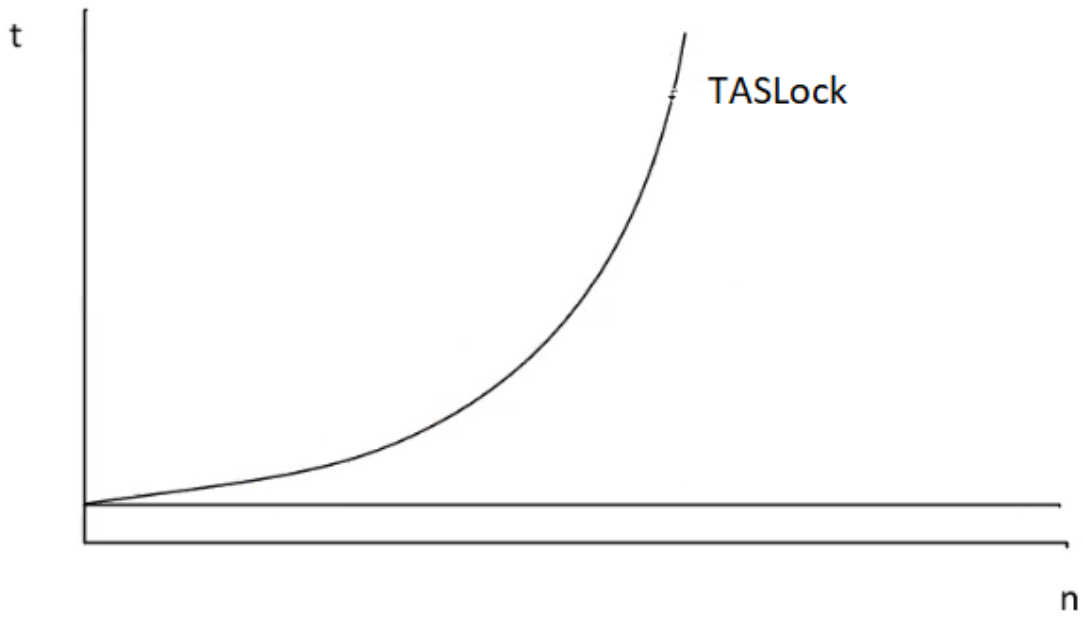
Сам алгоритм:

По умолчанию критическую секцию никто не занимает, поле **state = 0**.

В Lock() поток, который будет занимать критическую секцию, **поменяет 0 на 1**. Все остальные потоки, которые не смогли поменять значение в **state на 1**, так как там **уже было это значение**, будут **крутиться в бесконечном цикле** до тех пор, пока поток, занимающий критическую секцию, **не сбросит** в Unlock() **state в 0,** и кто-нибудь из них снова не установит его в **1**.

Число консенсусов для данного алгоритма **равно** ∞, таким образом он рассчитан на **бесконечное число потоков**.

Зависимость **времени от количества потоков**:



Для идеального, которого на практике не существует, но к нему стоит стремиться, алгоритма зависимость **константно, прямая**.

У **TASLock** зависимость - **экспонента.** Такой экспоненциальный рост связан с тем, что **все множество потоков** начинает работать с одной и той

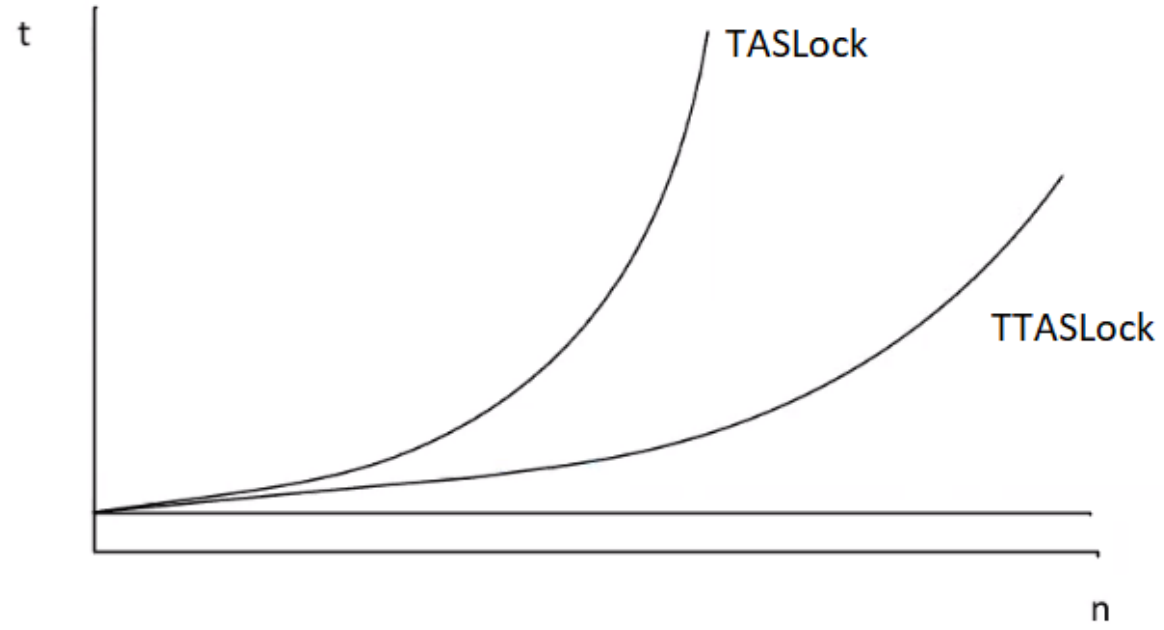
же ячейкой памяти, тем самым происходит **блокировка других потоков**, то есть они эту операцию делать не могут, то есть своего рода формируется **очередь** из атомарных операций.

Алгоритм TTASLock (Test-and-test-and-set):

Является **модифицированным вариантом TASLock.**

В **state** записываем 0.

В дополнение к **TASLock** добавляется цикл **while(state == 1).** Он нужен для того, чтобы организовать активное ожидание. Так как в нём нет атомарной операции, то при проверке мы работаем с **закэшированными данными**, при это **обращаемся к шине памяти** и **совершаем попытку занять критическую секцию,** когда совершаем атомарную операцию, **только если она потенциальна возможна**. То есть мы не пытаемся постоянно делать атомарную операцию, чем **ускоряем алгоритм**. В .NET прироста особа нет, в отличие от Java.



Spinlock (спинлок) - примитив синхронизации с использованием **цикла активного ожидания**. Фактически происходит постоянное “кручение” по какой-либо переменной в ожидании, когда **она переключится на нужное значение**.

И **TASLock** и **TTASLock** - алгоритмы вида **spinlock**.

Спин-блокировки являются аналогами мьютексов, позволяющими тратить **меньше времени на процедуру блокировки потока**, поскольку **не требуется** переводить поток в **заблокированное состояние**. В случае мьютексов может потребоваться **задействование планировщика с переводом потока в другое состояние** и добавлением его в список потоков, ожидающих разблокировки. Спин-блокировки **не задействуют планировщик и используют цикл активного ожидания без изменения состояния потока**, что приводит **к трате процессорного времени на ожидание освобождения блокировки другим потоком**.

Если в алгоритм **TTASLock** добавить **Thread.Sleep(0)** или **Thread.Yield**() внутрь **while(state == 1)**, это сообщит системе, что можно переводить потоки в другое состояние, то алгоритм переведется из **spinlock** в **mutex.**

Разница между **spinlock** и **mutex** состоит в том, что в **spinlock** есть только **активное ожидание**, а в **mutex** по факту делает **Thread.Sleep(0),** дающий указание системе, что поток можно **переводить в другое состояние**, усыплять. Но в этом случае реакция на пробуждение потока будет **долгой**.

Алгоритм BackoffLock:

Основан на TTASLock.

Если нужно добиться большего контроля за тем, когда потоку следует **просыпаться и насколько интенсивно проверять условие**, то можно использовать **алгоритм экспоненциального откладывания**.

Добавляется вызов функции из класса **Backoff()**, делающего откладывание. Алгоритм откладывания включает в себя размер **минимальной и максимальной паузы и лимита**, изначально равного минимальной паузе.

В **DoBackoff()** выбирается **интервал ожидания** от **0** до **лимита.** Затем в **лимит** устанавливаем **минимальное значение между максимальной паузой и удвоенным лимитом**. Таким образом, если после одного откладывания **критическая секция всё ещё занята**, то мы в следующий раз поток засыпает уже **на удвоенное время, по сравнению с предыдущим разом**, но не большее максимальной паузы.

14*. Неявная и явная реализация спинлоков с помощью списков.

Алгоритм ALock:

Предназначен только для конечного числа **n** потоков, то есть его число консенсусов **n**.

При инициализации:

Передаем **capacity**, по которому создаём массив флагов для потоков. Мы считаем, что у нас не более **capacity** потоков. Заводим переменную **tail = 0**. **flag[0]** выставляем в **true**. Таким образом, во всех, кроме нулевого элемента, значения флагов являются ложными.

Массив флагов - массив **разрешений потокам**, которые используют **i-ые** элементы массива соответственно. Если **i-ый** флаг выставлен в **истину**, то **i-ый** поток **может занять критическую секцию**. Если **false**, то другие потоки **занимают эту критическую секцию**. Изначально доступ к критической секции есть у **нулевого потока**.

Lock():

Вычисляется **slot - номер слота,** который будет использоваться потоками **для определения своего значения в массиве флагов.** Это будет происходить таким образом, что для **нулевого** пришедшего потока он будет равен **0**, для **первого - 1** и так далее. Далее происходит установка значения **mySlotIndex.Value**, чтобы **присвоить номер слота потоку**. Таким образом, индекс в массиве флагов и номер потока **не связаны**.

MySlotIndex - уникальное для каждого потока хранилище типа **ThreadLocal**. В нем можно хранить значение, которое будет **уникальным для каждого потока**. Как только происходит обращение к этому хранилищу, будет выставлено значение по умолчанию. В этом случае оно **0**.

Дальше проверка в цикле **while(!flag[slot**]), пока флаг не будет выставлен в **true**, то есть снова используется **активное ожидание -** значит, это **spinlock**.

Unlock():

Берем уникальное значение из хранилища **mySlotIndex**, **сбрасываем** по этому значению **флаг** в массиве флагов в **false**, и выставляем **для другого потока флаг** в **true**.

По факту алгоритм устанавливает **очередь конечной длины из потоков** на основании прибытия потоков в метод **Lock**(), где им выдаётся уникальный номер **через атомарную операцию инкрементирования**. Потоки бегают по флагам по кругу, каждый раз давая **следующему по флагу потоку** попасть в критическую секцию.

Алгоритм CLHLock:

Совершенная версия **ALock**, где используется **не массив, а неявный список**. Данный алгоритм позволяет синхронизировать **бесконечное** число потоков.

Определяется дополнительный класс **QNode**, хранящий переменную **locked** типа **bool** и свойства для обращения к ней.

Есть **узел tail** и **два ThreadLocal** хранилища: одно указывает на **текущий узел**, другое **на предшествующий узел**.

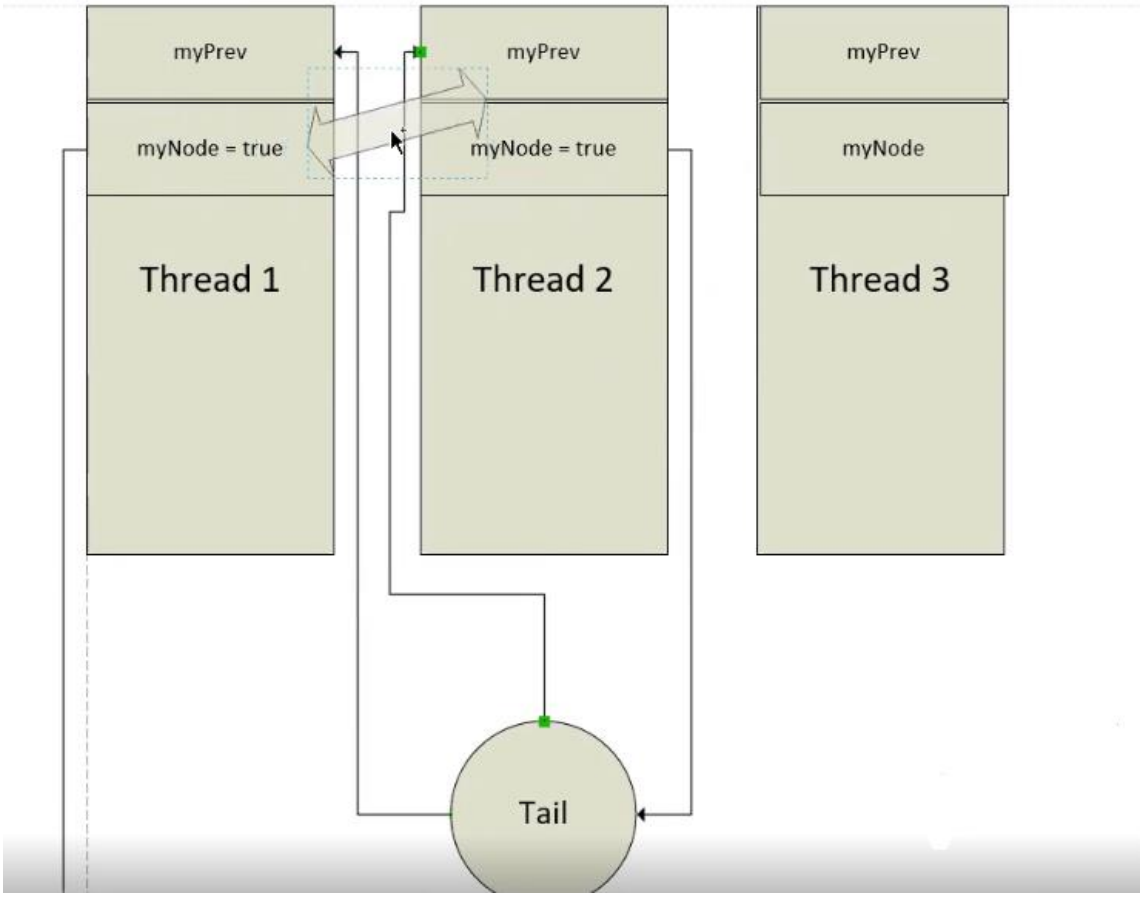
Lock():

Получаем **текущее значение узла qnode**, по умолчанию из локального хранилища придет **новый QNode**, который мы инициализируем через лямбда-выражение.

Далее в **qnode** меняем значение **locked** в **true** и заменяем атомарным обменом **Exchange()** значение **tail** (**фактически это голова условного списка**) на новый **qnode**. Далее в хранилище **myPred** записываем **предыдущую голову списка**, то есть предыдущее значение **tail до его замены**. Далее цикл **while(pred.Locked)**, то есть **активное ожидание - spinlock()**.

Unlock():

Берем **текущее значение узла**, меняем значение **locked** на **false** и в **myNode** сохраняем значение **myPred,** чтобы его **повторно использовать.**



Фактически связь между двумя потоками, формирование списка между двумя потоками делается через два **ThreadLocal** хранилища и поле **tail** (**голова условного списка**). **Tail** просто содержит текущий **myNode,** чтобы атомарно обменять его со следующим и в этот момент присвоить его в **myPred.**

При освобождении критической секции **Thread 1** происходит следующее: **Thread 1** выставляет значение **locked из myNode** в **false**, тоже самое происходит и в **Thread 2** в **myPrev,** так как это **один и тот же объект**. В итоге **Thread 2** видит, что критическая секция свободна и занимает её.

Алгоритм TOLock:

Раньше блокировка критической секции могла длиться **бесконечно**, если поток, например, забыл закрыть секцию. Этот алгоритм призван это исправить.

Данный алгоритм принимает в качестве параметра **timeout** и по истечении него **ждущий поток будет выходить из блокировки** и что-то делать дальше, например, сообщать, что блокировка не удалась.

QNode - класс, в котором содержится **ссылка на предыдущий узел**.

Создаём **QNode AVAILABLE – особый экземпляр** для обозначения пустоты очереди.

Фактически объявляем **троичную логику для очереди**:

8. Если **null**, то **в очереди никого нет**.
9. Если **AVAILABLE**, то можно **очередь занимать и не обращать внимания на этот узел**.
10. Если **другое значение**, то это **полноценный узел какого-то потока**.

Как и в алгоритме **CLHLock** список организован **неявно**, то есть мы храним текущий узел в **ThreadLocal** и заводим переменную **tail**.

То есть оба этих алгоритма - **неявная реализация** с помощью **списков**.

Вместо метода **Lock()** здесь **TryLock()** из интерфейса **IMaybeLock**, возвращающий **true**, если мы зашли в критическую секцию или **false**, если войти не получилось. Он принимает параметр **PatienceInMs** – это и есть **timeout**.

TryLock():

Фиксируем время, когда поток начинает пытаться войти в критическую секцию.

Создаем новый **узел QNode** и **сохраняем его в ThreadLocal хранилище**. В качестве предыдущего значения этого узла ставим **null**, то есть предшественников у этого узла нет.

Далее заменяем значение **tail** на **текущий созданный узел**, фактически встаем в очередь по **tail**. Предыдущее значение **tail** сохраняем в **myPred**.

Если **myPred == null**, то мы являемся **первым потоком, входящим в критическую секцию**, значит **возвращаем true** и **занимаем критическую секцию**.

Если предшественник **myPred.Pred** это **AVAILABLE**, то считаем, что также можем зайти в критическую секцию, это означает, что **предшественник уже освободил критическую секцию и выставил AVAILABLE**.

В противном случае идём дальше:

Крутимся в **while** до тех пор, пока мы не превысили **PatienceInMs** или **не заняли критическую секцию**.

Внутри цикла мы берем узел предшественника **myPred.Pred.** Если он **AVAILABLE**, то, как выше, это значит, что **предшественник уже освободил критическую секцию и выставил AVAILABLE**, значит мы **можем занять критическую секцию**. В противном случае если он **не равен null**, то это значит, что наш предшественник сам завершился с **PatienceInMs**, то есть его больше не нужно учитывать, поэтому в качестве нашего предшественника **myPred** мы ставим этот **predPred**.

Если мы вышли из цикла по **PatienceInMs**, то мы пытаемся заменить **tail** с нашего текущего узла **qnode** на узел предшественник **myPred**. Если это удается, то это означает, что **текущий поток был последним в очереди**, значит никому сообщать, что мы вышли из очереди **не нужно**, поэтому просто выходим. Если заменить не удалось, то есть **за нами уже стоят в очереди**, то мы должны как бы предупредить, что заканчиваем работу, поэтому выставляем в **qnode.Pred** для нашего текущего узла значение **myPred**, чтобы другой ждущий поток увидел, что наш поток можно больше не рассматривать.

Unlock():

Получаем **узел для текущего потока**, далее пытаемся подменить **tail с текущего узла qnode на null**. Если удается, это означает, что **наш поток был первым и единственным в очереди**, а значит за этим потоком никто очередь не занимал, поэтому просто выходим. Если же атомарная операция прошла безуспешно, то сообщаем предшественнику, что он может занять критическую секцию, **выставив значение нашего предшественника в AVAILABLE**.

15. Семафоры и мьютексы. Решение задачи производителей-потребителей с ограниченным буфером на семафорах и мьютексах.

Mutex - примитив синхронизации, обеспечивающий взаимное исключение. Отличается от **spinlock** добавлением **Thread.Sleep(0) или Thread.Yield()** для информирования операционный системы, что поток **можно переводить в другой режим**.

В **mutex** можно отслеживать, **какой поток является владельцем критической секции или владельцем мьютекса**.

Считается, что **mutex - блокирующий алгоритм**, так как он в какой-то момент блокирует выполнение какого-нибудь потока.

Semaphore (семафор - примитив синхронизации, для которого при создании указывается **максимальное число (емкость) capacity**. Если **capacity = 1,** то **semaphore** становится **mutex** и **обладает** свойством взаимного исключения. Семафор можно использовать для организации **не только взаимно-исключающего доступа**, но также и для организации доступа к ресурсам, **которые допускают использование одновременно не более чем n потоками**. В этом случае он свойством взаимного исключения может не обладать.

Как только **поток пытается использовать семафор**, то он **декрементирует емкость,** и **не может этого сделать**, когда число равно **0**.

При освобождении семафора поток инкрементирует емкость.

Так как нет ограничения, какой поток захватывает, а какой освобождает, то **владельца семафора быть не может**.

Вводим поле **state,** означающее **состояние семафора, - число потоков**, которые используют этот семафор. Имеется общий объект синхронизации **sync.**

Acquire():

Входим в монитор.

Если другие потоки **захватили весь семафор**, то **ожидаем**.

Если еще есть место для потоков, **то занимаем его и инкрементируем state**. Затем выходим из критической секции.

Release():

Если пытаемся освободить **пустой семафор**, то **кидаем исключение**.

В противном случае **декрементируем state и делаем PulseAll()**, сообщая другим потокам, что в семафоре **появилось свободное место**.

16. Мониторы и переменные условия.

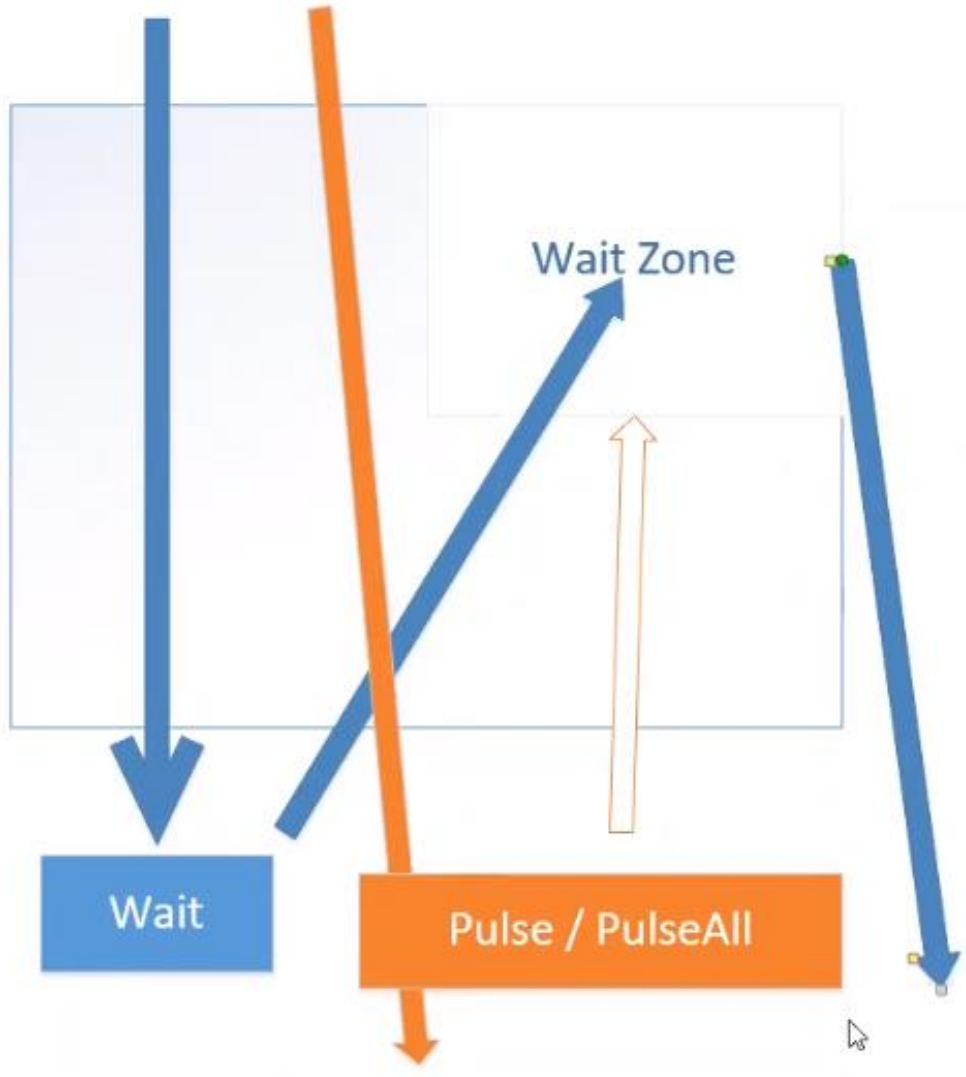
Монитор - примитив синхронизации.

У мониторов есть методы **Enter() -** для входа в критическую секцию **и Exit() -** для выхода из критической секции.

Чтобы отличать один монитор от другого, используется **объект-заглушка** в качестве параметра. Считается, что с каждым объектом связан свой монитор. Входить и выходить из одной и той же критической секции нужно **по одному и тому же объекту**.

Передать в монитор **контейнерный тип нельзя** из-за **boxing**: так как каждый раз будет создаваться **новая ссылка** на контейнерный тип. Необходимо использовать **только ссылочный тип**.

Рекомендуется использовать конструкцию **try-finally** для работы с монитором, чтобы **в любом случае освобождать критическую секцию**.



Wait() - команда внутри монитора, прекращающая работу монитора, **освобождает критическую секцию, не покидая её.** То есть поток как бы **выходит из критической секции, но не покидает монитор**.

Pulse()/PulseAll() - сигнал для потоков из **Wait()**, что критическую секцию можно занимать. Срабатывают только после того, как вызывающий их поток **покинет критическую секцию**.

Pulse() - пробуждает **один** поток.

PulseAll() - пробуждает **все** потоки, при этом потоки все равно проходят в критическую секцию **по одному**.

Wait() и Pulse()/PulseAll() - вызываются **только внутри критической секции.**

Condition Variable (переменная условия):

Если какой-то поток не может продолжить работу в критической секции, то он переходит **в ожидание**, проверяя **при пробуждении в цикле while условие**, из-за невыполнения которого он в ожидание и вошёл. Как только условие выполниться, поток **выйдет из ожидания**.

Алгоритм LockedQueue (решение задачи производителей-потребителей с ограниченным буфером):

Производители и потребители обладают **равными правами**. Имеется **буфер** по типу FIFO **ограниченного размера**, куда производителями складываются товары, а потребителями - достаются. Размер буфера передаётся в конструктор.

В обеих критических секциях используем один и тот же объект **sync.**

Enqueue() - метод для производителей:

Входим в критическую секцию.

Далее, если мы хотим **положить новый объект**, но весь **буфер заполнен**, то мы **переходим в ожидание** до тех пор, **пока не освободится буфер**.

Если же в буфере **есть место**, то добавляем **новый объект в буфер**. **Tail** используется для мониторинга **свободного места** в буфере.

Если **достигли конца буфера**, то переходим в начало буфера при помощи **tail = 0,** и **увеличиваем счетчик количества объектов count**. Далее делаем

PulseAll(), так как нам **не важно, какой поток будет класть или брать из буфера**.

Dequeue() - метод для потребителей:

Входим в критическую секцию.

Если **буфер пустой**, то **поток входит в ожидание**.

Если **буфер не пустой**, то **забираем элемент**. **Head** используется для мониторинга **занятого места** в буфере.

Если **достигли конца буфера**, то переходим в начало буфера при помощи **head = 0,** и **уменьшаем счетчик количества объектов count.** Поскольку мы освободили только **одно место в буфере**, то мы пробуждаем только **один поток** при помощи **Pulse()**.

Проблема: потенциально может возникнуть ситуация, когда **PulseAll()** пробудит **не только производителей, но и потребителей**. В этом случае больших проблем не будет, так как записывающие потоки **прокрутятся пару раз и снова уйдут в сон**. А вот **Pulse()** в **Dequeue()** может пробудить **не производителя, а только потребителя,** и тогда возникнет ошибка. Поэтому лучше писать всегда **PulseAll()**.

17*. Реэнтрантный спинлок. Read-Write Lock.

Алгоритм ReentrantLock (реэнтрантный спинлок):

Если один поток захочет получить доступ к блокировке еще раз, то он навсегда уйдет в ожидание, как, например, в **TASLock.** Чтобы это решить, существует реэнтрантность.

Реэнтрантность - свойство spinlock или mutex, что **если поток захватил spinlock или mutex, то он сможет захватить его вновь без исключений**.

Этот алгоритм можно запускать рекурсивно за счёт этого свойства.

Введем дополнительную переменную **owner**, в которой будем хранить **ID потока**, а также счетчик **holdCount**, в котором считается, **сколько раз один и тот же поток захватывал спинлок или мьютекс**.

Lock():

Запоминаем **ID текущего потока в me**, входим в монитор.

Если **owner == me**, то есть мы **уже находимся в критической секции**, то **увеличиваем счетчик захватов и выходим из метода**. Вызов return вызовет Monitor.Exit.

Если **проверка не прошла**, то **ожидаем**, пока другой поток не освободит все занятые мониторы.

Как только это произойдет, то мы скажем, что это **мы захватили монитор и поставим число захватов 1.**

Unlock():

Входим в монитор.

Если мы пытаемся разблокировать **никем не занятый монитор или монитор, занятый другим потоком**, то **кидаем исключение**.

Если всё корректно, то монитор **занят нами**, поэтому **уменьшаем счетчик захватов**, и в случае равенства **0**, сообщаем другому потоку, что **он может занять критическую секцию,** так как в этом случае мы закрыли все мониторы. Затем выходим из монитора.

Алгоритм ReadWriteLock:

Читатели и писатели **не обладают равными правами**. Есть множество писателей, **пишущих в одну ячейку**, и читателей, **читающих из одной ячейки**.

Требуется организовать работу таким образом, чтобы писатели могли писать, а читатели читать, при этом **частичных** (поломанных) **объектов быть не должно**.

Имеется общий объект синхронизации **sync,** число читателей **readers** и значение писателя **writer**, говорящее, хочет ли он что-то написать.

Читатели:

В ReadLock.Lock(): чтобы суметь прочитать что-то, сначала входим в монитор. Далее пока **есть писатель**, который хочет что-то записать, **читатель переходит в ожидание**. Далее, **если писатель уже не хочет писать, увеличиваем количество читателей и выходим из монитора**.

В ReadLock.Unlock(): входим в монитор, **уменьшаем количество читателей.** Если их стало **0**, то информируем **PulseAll()**, что другие читатели или писатели могут получить доступ.

Писатели:

В WriteLock.Lock(): входим в монитор. Если **есть читатели или уже пишет какой-то писатель**, то **уходим в режим ожидания**. Если **никого нет**, то мы выставляем флаг писателя в **true** и начинаем писать.

В WriteLock.Unlock(): в мониторе выставляем флаг писателя в **false** и делаем **PulseAll**(), чтобы другие читатели или писатели могли получить доступ.

Проблема: читателей может быть так много, что **никогда не возникнет ситуация, что писатель начнет что-то писать**.

Алгоритм FairReaderWriterLock:

Ещё **более неравноправная** модификация **ReadWriteLock,** призванная решить возникающую в нём проблему.

Имеется общий объект синхронизации **sync.** Храним число читателей, вошедших в критическую секцию, - **readAcquires,** и число читателей, освободивших критическую секцию, - **readReleases**.

Читатель:

В **ReadLock.Lock(),** аналогично предыдущему, пока **есть писатель**, который хочет что-то записать, **читатель переходит в ожидание**. Далее, если **никто не хочет писать**, **увеличиваем количество читателей**, **вошедших в критическую секцию**, **и выходим из монитора**.

В **ReadLock.Unlock() увеличиваем количество читателей, вышедших из критической секции,** и в случае **равенства вышедших с вошедшими** делаем **PulseAll()**.

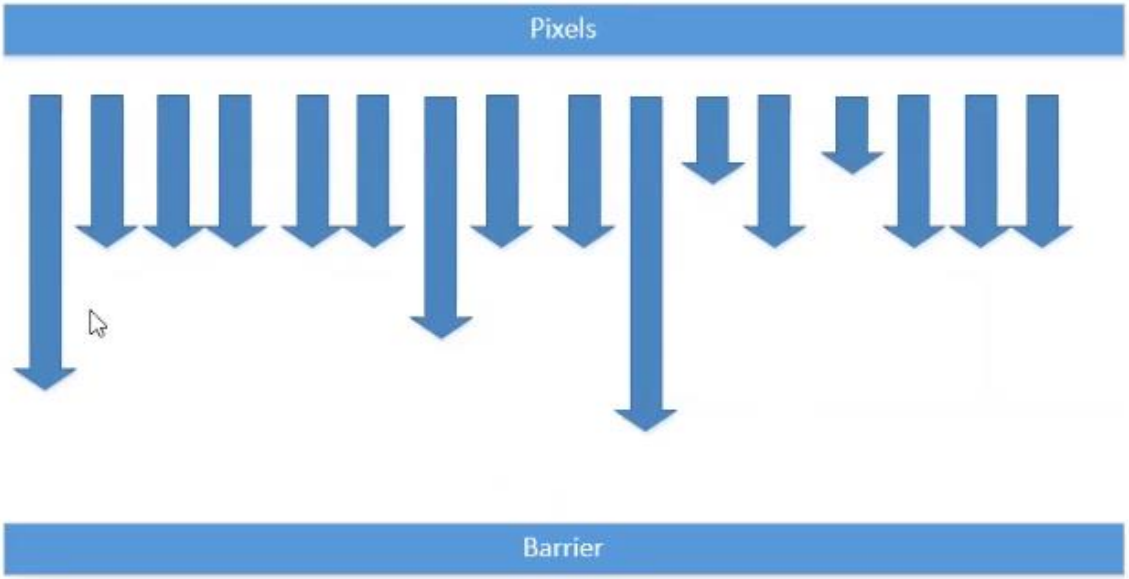
Писатель:

В **WriteLock.Lock()** смотрим, есть ли другой писатель. Если **есть,** то крутимся **уходим в ожидание**. Если **нет другого**, ставим флаг писателя в **true**. Таким образом, с этого момента все читатели и писатели будут становиться в ожидание. Далее **ожидаем**, пока все **читатели, что прошли на чтение перед нами, не покинут критическую секцию**.

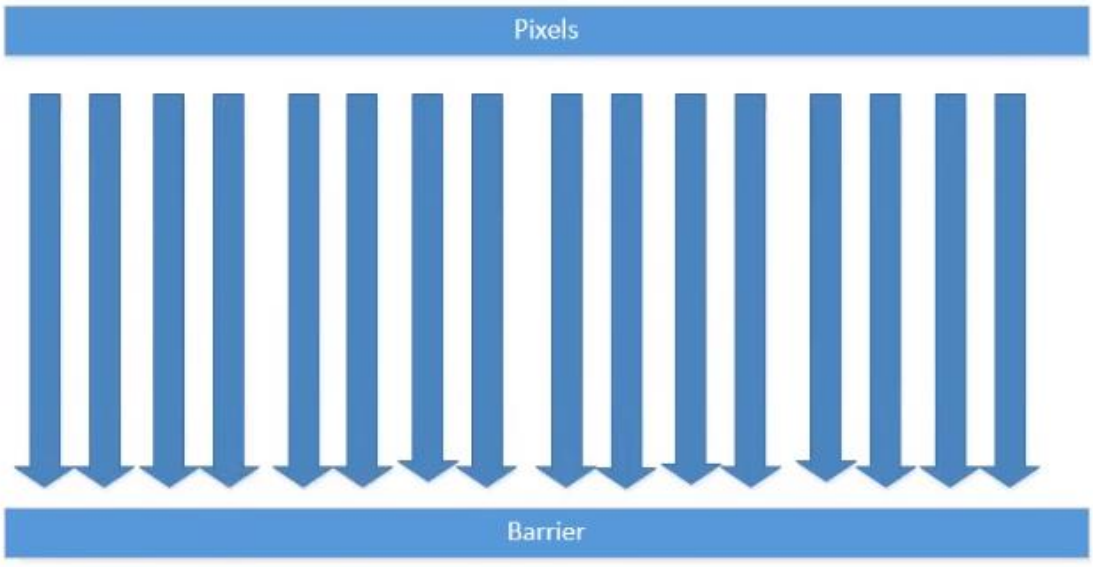
В **WriteLock.Unlock()** сбрасываем флаг писателя в **false** и делаем **PulseAll(),** чтобы другие потоки пытались что-то сделать.

18*. Понятие барьерной синхронизации. Простая барьерная синхронизация.

Барьеры часто используются при работе с **изображениями**. Барьеры есть в **MPI.**



Барьерная синхронизация предполагает, что при достижении некоторой **точки в коде** потоки ждут в ней, пока **все потоки** не достигнут этого барьера. Необходимость наличия барьера в этом случае обосновывается тем, что все потоки работают **с разной скоростью,** поэтому и достигают барьера в разное время.



Когда **все потоки достигают барьера**, они **проходят через него**. Это своего рода точка ожидания всех потоков.

Барьеры создаются для **конечного** числа потоков.

Алгоритм SimpleBarrier (простой барьер):

Данный барьер построен на **атомарных операциях**.

Создаем переменную **count,** которую инициализируем из конструктора числом **n**. Эта переменная означает **число потоков**, для которых будет работать данный барьер. Её же сохраняем в **size** для **перезапуска бареьера**.

Каждый из потоков при достижении барьера вызывает метод **Await(),** который **задерживает потоки,** пока они все не достигнут его.

В начале атомарно декрементируем счётчик числа потоков для барьера **count.** Полученное число **position - позиция потока в барьере.**

Если **position** оказывается **не равен 0,** то отправляем поток в **активное ожидание**, пока все потоки не придут в данный метод, то есть **count не станет равен 0.**

Если **position** оказывается **равен 0,** то **все потоки достигли барьера**. Поэтому барьер открылся при декременте, а мы сбрасываем **count в начальное состояние size.**

Теперь барьер готов **к повторному использованию**.

Проблема: на примере двух потоков. Первый поток уходит в **ожидание**. Второй поток дошел до барьера и **обновил значение count.** Если первый поток **был в состоянии сна** по команде операционной системы, то он может **пропустить момент, когда count был равен 0**, и таким образом, этот **поток останется ждать в цикле**. Поэтому здесь и применяется **активное ожидание**, в нем не может произойти такой ситуации.

Алгоритм SenseReverseBarrier:

Как и в прошлом алгоритме есть **size и count**. Дополнительно вводим переменную **phase - глобальное состояние барьера**, а также **ThreadLocal** хранилище **threadPhase**.

При создании барьера также передаем **размер барьера n**, выставляем **phase** в **false**, в **threadPhase** хранилище ставим **true**.

Await():

Получаем значение **myPhase** из локального хранилища потока. Также декрементируем счетчик для получения **position**.

В случае если **не все потоки дошли до барьера**, то крутимся в **активном ожидании** до тех пор, пока **глобальное состаяние барьера phase** не станет **равно фазе данного потока myPhase**. Таким образом, происходит **ожидания равенства фаз всех потоков глобальной фазе**.

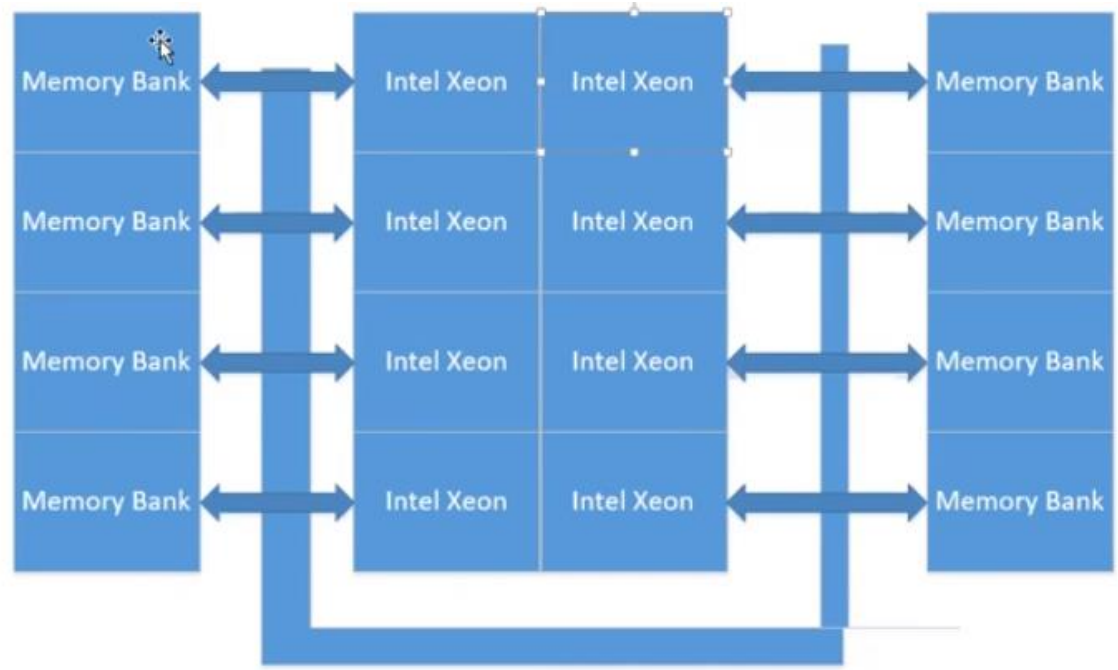
Если же поток пришел **последним в барьер**, то **count** переводим в **изначальное состояние** и **глобальное состояние барьера phase меняем на локальное состояние myPhase**. Фактически глобальная фаза — это фаза последнего пришедшего в барьер потока. Таким образом, потоки, ждущие в цикле, **выйдут из него**.

Далее все потоки, выходя из ожидания, меняют **свое локальное состояние фазы в локальном хранилище на противоположное** и, таким образом, переводят барьер в рабочее состояние: однако теперь всё наоборот, в том смысле, что все начальные значения в локальном хранилище у всех потоков **threadPhase = false,** а глобальная фаза **phase = true.**

Таким образом, этот барьер будет менять свои состояния на противоположные каждый раз для своего **переиспользования**.

Проблемы, как в предыдущем алгоритме **возникать не будет**, так как даже если усыплённый поток проснётся позже других, то он сможет выйти из барьера **за счёт инвертирования всех состояний**.

Недостаток: все потоки крутятся **по одной и той же переменной,** что плохо из-за конкуренции. Также это решение может плохо работать на **NUMA-архитектуре,** так как переменная будет лежать в банке памяти **другого процессора** и обращение к ней будет происходить **дольше по общей шине памяти**, которая нужна для обращения к банкам памяти других процессоров, нежели по **прямой**, по которой обращаются процессоры **к своим банкам памяти**.



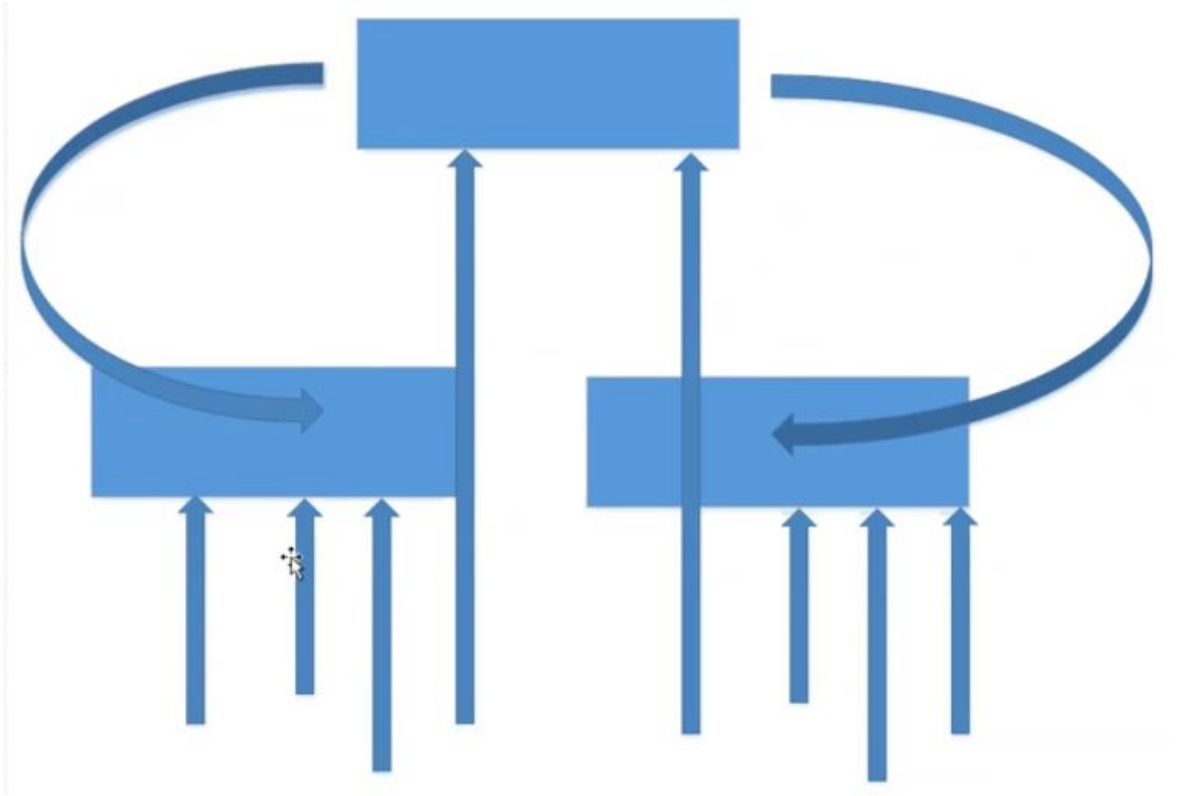
19*. Иерархическая барьерная синхронизация.

Алгоритм TreeBarrier (древовидный барьер):

Чтобы решить проблему **SenseReverseBarrier,** связанную с единственной переменной условия, необходимо построить **иерархию барьеров**.

Может быть несколько барьеров “**низкого уровня**”. Когда барьер такого уровня **заполняется**, это является **сигналом перехода** к работе **над более высокоуровневым барьером**. Когда **высокоуровневый барьер заполняется**, передается сигнал **на барьеры более низкого уровня**, что их можно “**открывать**”.

Древовидный барьер как раз и обеспечивает такую **композицию** барьеров.



При создании такого барьера передаем число **n - общее количество потоков** и **childrenLimit - максимальное количество потоков,** синхронизируемых **на одном узле дерева**.

Далее создаем **массив узлов leaves (листы)** и начинаем **рекурсивно строить дерево,** заранее высчитывая глубину дерева **depth и создавая для родительских узлов дочерние и так далее** в методе **Build(),** пока дерево не будет выстроено. Однако суть алгоритма не в этом, поскольку это исключительно строительство дерева.

Await():

Считаем, что **идентификатор потока** лежит от **0 до n - 1**. Поэтому частное от деления **идентификатора потока на количество потоков** в одном листе дерева даст **индекс листа для соответствующего потока,** и мы получим **myLeaf**. Далее для этого узла вызываем **метод ожидания Node.Await()**.

Класс Node:

Класс строится **двумя путями**: либо **это корневой узел**, либо узел, у которого **есть родитель.**

Node.Await():

Получаем текущую фазу потока **mySense** из локального хранилища **threadSense**, **декрементируем счетчик count** и получаем **position**.

В случае, если мы **не последний поток для этого узла**, то уходим в **активное ожидание** до тех пор, **пока глобальное состояние sence этого узла не станет равно локальному для потока**.

В случае, если мы являемся **последним потоком в данном узле**, то проверяем, **равняется ли родительский узел null**. Если это не так, значит это ещё **не корень**, и **поднимаемся на уровень выше по барьерам,** передавая сигнал. Если же **родительского узла нет**, это значит, что **последний поток во всем дереве дошел до самого конца**, то сбрасываем **count узла в childrenLimit** и **переводим глобальное состояние узла sense в локальное mySense**. Затем ожидающие потоки выйдут из ожидания и **инвертируют** свой **threadSence. Барьер откроется.**

После достижения **корневого узла последним для него потоком,** он изменяет **sence,** чем **открывает и нижележащие барьеры**. Все барьеры также поменяют свое состояние **на противоположное**, чем обеспечат себе возможность **повторного использования** и решат проблему “**проспавшего потока**”.

Недостаток: как и в предыдущем, если используем **NUMA-архитектуру**, то если **локальный барьер находится в нашем банке памяти с быстрым доступом**, то не факт, что такой же быстрый доступ будет к переменной, по которой мы крутимся, **в родительском узле**.

Алгоритм StaticTreeBarrier (статический древовидный барьер):

Идея древовидности **аналогична TreeBarrier**, но в отличие от него, пытаемся **минимизировать издержки на доступ к другим банкам памяти,** если говорить про **NUMA-архитектуру**.

Аналогично предыдущему передаем **общее количество потоков size**, **количество потоков в одном узле childrenLimit**.

Далее создаем все множество узлов, **хранящееся в массиве Node**, вычисляя глубину **depth** и создавая **родительские и дочерние узлы рекурсивно в функции Build().** Инициализируем **globalSense = false, threadSence = !globalSence.**

Await():

Вызываем метод **Node.Await()**, определяя по текущему **ID потока элемент массива узлов**.

Класс Node:

В классе **помимо родителя parent**, мы храним **корень всего дерева root**.

Node.Await():

Получаем текущее значение **mySense** из локального хранилища потока.

Далее узел **активно ждет, обращаясь к своему банку памяти**, пока **число дочерних потоков**, которое **не пришло к этому барьеру, будет больше 0**. Фактически, листовые барьеры синхронизируют **только свой родительский элемент**.

Родительский узел выйдет из ожидания, когда все его **дочерние узлы продекрементируют счетчик детей своего родителя**. Например, если есть узел, у которого два ребенка-листа, то два-ребенка листа пройдут этот цикл, так как у них нет детей, вызовут от родителя метод **ChildDone()** и уменьшат количество детей. Тогда это число станет равно **0**, сам **родитель выйдет из цикла**.

Если **родитель не равен null**, то есть этот **узел - не барьер самого высокого уровня**, то он **сообщает родителю,** что **закончил свою работу** путем вызова метода **ChildDone()**, который **декрементирует атомарно счетчик childCount**, и **начинает активно ждать** в цикле изменения **глобального состояния globalSense**.

Если же **родителей не было**, то есть наш **узел корневой** и **барьер полностью заполнен**, то можем **изменять глобальное состояние барьера globalSence**. Когда потоки выходят из ожидания, они также инвертируют **threadSence,** чтобы обеспечить **повторное использование** алгоритма и решить проблему “**проспавшего потока**”.

Чтобы распространить информацию, что **уровень барьера заполнен**, достаточно вызвать **атомарную операцию декремента.** Таким образом, каждый узел дерева крутится по своей локальной памяти и благодаря этому число тяжелых операций между банками памяти **в NUMA-архитектуре минимизируется**.

В отличие от **TreeBarrier()**, где потоки **работали с листьями дерева и поднимались выше**, то в этом алгоритме **с каждым узлом дерева проассоциирован свой поток**, **у каждого потока есть свой выделенный узел.** Эти узлы можно располагать в памяти так, что вся локальная информация, необходимая для работы потока, будет находиться **в тех банках памяти, которые являются для того или иного процессора локальными**. Это ускоряет работу алгоритма.

20*. Высокоуровневая и детализированная синхронизация для односвязного сортированного списка.

Соглашения для алгоритмов на списках:

1. Когда мы добавляем **новый элемент в множество**, когда **удаляем**, **ищем** элемент, будет искать **не сам элемент**, а его **хэш**.
2. Два элемента в списке всегда будут **опорными**. Первый элемент - **пустой элемент, содержащий минимальное значение хэш-кода**. Второй элемент - **максимальное значение хэш-кода**. Эти значения хэш-кодов **зарезервированы**.
3. Список в данном алгоритме **односвязный**. Наш список **отсортирован по возрастанию**. Но можно по убыванию, поменяются только знаки.
4. **Add() -** возвращает **истину**, если элемент был добавлен, в противном случае **ложь**.
5. **Remove() -** возвращает **истину**, если элемент был удален, в противном случае **ложь**.
6. **Contains() -** возвращает **истину**, если элемент содержится в множестве, в противном случае **ложь**.

Алгоритм CoarseSet:

Данный алгоритм является алгоритмом **грубой или высокоуровневой синхронизации.**

Создаем объект **sync** для **синхронизации мониторами**.

Инициализируем:

Первый опорный элемент **head** - **пустой элемент, содержащий минимальное значение хэш-кода**. Второй опорный элемент **head.Next** - **максимальное значение хэш-кода**.

Add():

Вычисляем **значение хэша элемента key.** Захватываем монитор.

Затем **ходим по списку** до тех пор, **пока значение хэша текущего элемента списка меньше значения хэша элемента, который мы хотим вставить**. В случае **равенства этих значений**, это значит, что этот **элемент уже добавлен**. В случае **неравенства** значений мы вставляем **новый элемент в список** между элементами **pred и curr.**

Отпускаем монитор.

Remove():

Вычисляем **значение хэша элемента key.** Захватываем монитор.

Аналогично прокручиваемся по списку до тех пор, **пока значение хэша текущего элемента списка меньше значения хэша элемента, который мы хотим удалить**. В случае **равенства** этих значений, **удаляем элемент**, перекидывая ссылку **pred.Next на curr.Next**. В **противном случае**, это значит, что **удалять нечего**, так как этого **элемента в списке нет**.

Отпускаем монитор.

Contains():

Вычисляем **значение хэша элемента key.** Захватываем монитор.

Аналогично крутимся по списку до тех пор, **пока значение хэша текущего элемента списка меньше значения хэша элемента, который мы хотим найти**. Далее **возвращаем результат сравнения ключа искомого элемента с ключом найденного элемента**.

Отпускаем монитор.

Минус алгоритма: всякий раз, когда обращаемся к списку, мы **блокируем доступ к нему для других потоков**. То есть, например, при работе метода **Add()**, потоки в **Remove()** и **Contains()** будут **простаивать**. Фактически, единовременно со списком может работать **только один поток**.

Также, наше **множество** является объектом, **к которому можно получать доступ на чтение и на запись**. И получается, что читатели могут мешать писателям, если говорить в этих терминах. Логичным расширением этого алгоритма в таком случае будет **замена монитора на один из Read/WriteLock()**.

Уровень параллелизма в данном алгоритме соответствует **1**.

Алгоритм FineGrainSet:

Данным алгоритм является алгоритмом **детализированной синхронизации.** Он позволяет работать нескольким потокам с разыми частями множества **одновременно**.

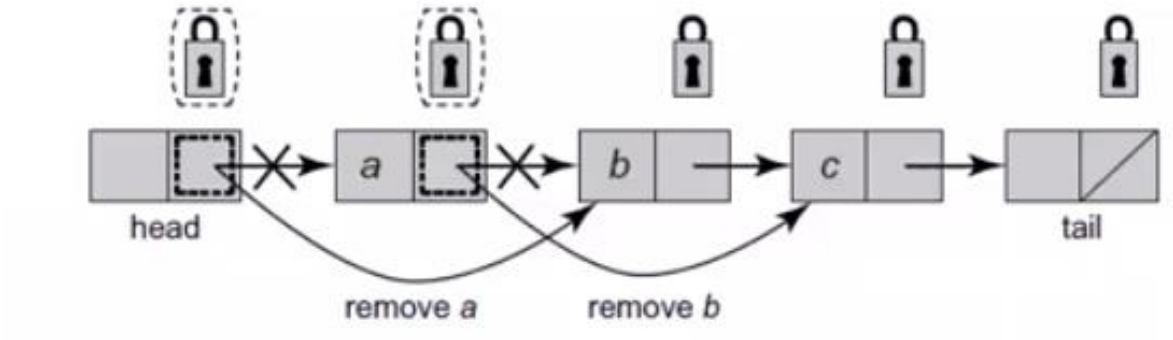
Класс Node:

Помимо **ключа key**, **значения value** и **ссылки на следующий элемент Next,** есть **lock**, **проассоциированный с конктреным узлом.** В качестве **lock** можно использовать любой **spinlock**, **mutex**, **monitor**.

То есть с каждым узлом списка мы ассоциируем **свой примитив синхронизации**.

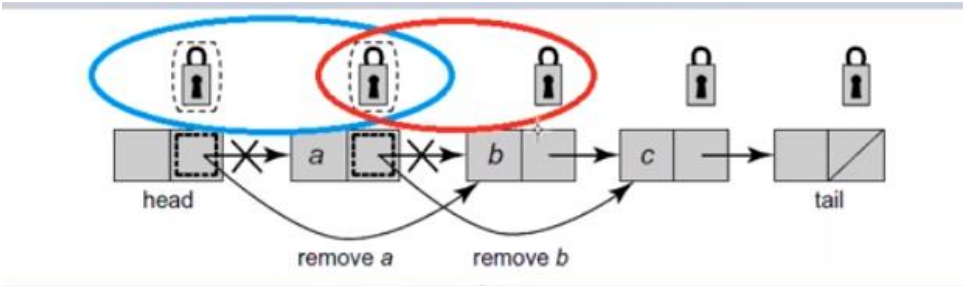
Общая идея: постепенно **проходим по списку от начала в конец** и **захватываем подходящие узлы**, если они **не подходят, отпускаем их**.

Мы захватываем сразу **два узла**. Сделано это для следующего:



Пусть есть два потока, **один хочет удалить узел a, другой узел b**. Если два потока одновременно попробуют удалить свои элементы, **заблокировав только их**, то ссылки переместятся **неправильно** (**head** будет указывать на удалённый элемент **b**). То есть блокировки только одного элемента **недостаточно**.

В случае, если захватывать **по 2 элемента каждому потоку, то подобной конфликтной ситуации не произойдет:** первый поток **захватит свои два узла**, а второй поток, пытаясь захватить свои два узла, **не сможет этого сделать** до тех пор, **пока первый поток не отдаст блокировку**.



Contains():

Вычисляем **значение хэша элемента key.**

Сперва блокируем текущий **головной узел**. Далее блокируем его **последующий узел**.

Далее **идём по списку** до тех пор, **пока ключ текущего элемента меньше ключа элемента, который мы хотим найти**.

Внутри цикла прохода мы **разблокируем предыдущий элемент**, **перемещаем ссылки на следующий элемент, который и так заблокирован,** и **блокируем последующий узел.** Своего рода, ходим по списку **окном из двух элементов**.

При выходе из цикла возвращаем результат проверки **ключа найденного элемента с ключом элемента, который мы хотим найти**. **Снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock**.

Add():

Вычисляем **значение хэша элемента key.**

Аналогично, мы ходим **по списку окном из двух элементов**, блокируя и разблокируя нужные элементы**,** до необходимой позиции в списке.

В случае **равенства ключа найденного элемента с ключом элемента, который мы хотим вставить, выходим из метода**, так как такой элемент **уже здесь содержится**.

В **противном случае добавляем элемент в список между pred и curr** и **снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock**.

Remove():

Вычисляем **значение хэша элемента key.**

Аналогично **ходим по списку окном из двух элементов**, блокируя и разблокируя нужные элементы, до необходимой позиции в списке.

В случае **нахождения нужного элемента - удаляем его**, перемещая ссылки между **pred и curr.Next,** так как мы обладаем эксклюзивной блокировкой на них. **Снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock**.

При этом стоит учесть, что **при добавлении или удалении элемента** никакой другой поток в этом **окне из двух элементов не сможет ни добавить, ни удалить что-либо ещё.**

Уровень параллелизма в данном алгоритме **повышается**, ведь со списком по факту могут работать **одновременно более одного потока**. К примеру,

один поток может работать со списком в конце, другой в начале списка. Однако если происходит **много обращений к началу списка**, то алгоритм **почти вырождается в CoarseSet.**

Проблема алгоритма: может возникнуть ситуация, когда поток, работающий с каким-то **куском списка**, может **блокировать другие потоки**, которые могут **и не работать с этим куском списка** з**а счет хождения окном**. Своего рода, появляется **препятствие** для прохождения по списку. В таком случае потоку придется дожидаться разблокировки для продолжения своей работы.

Однако данное **ограничение сделано специально**, чтобы гарантировать, что если какие-то **узлы не заблокированы**, то никакой другой поток в это время **их не меняет**.

21*. Оптимистическая синхронизация для односвязного сортированного списка.

Метод **Contains()** в **FineGrainSet** не вносит никаких изменений в список, так как лишь ищет элементы, однако блокирует узлы во время поиска из-за **прохода окном**, чем обеспечивает **непротиворечивость данных**, но **замедляет работу алгоритма**. Это пессимистический вариант.

Пессимистический подход к синхронизации звучит так: если конфликты весьма вероятны, то давайте их **вообще не допускать**.

Оптимистический подход к синхронизации предполагает: если конфликт возник, то скорее всего **он быстро решится при помощи повторного обращения к данным**.

Это две принципиально разные стратегии решения конфликтов, организации синхронизации.

Алгоритм OptimisticSet (оптимистичный список):

Данный алгоритм предполагает, что **вероятность конфликта очень мала**, и поэтому лучше **не тратить дополнительных средств для синхронизации**, а дойти до момента проверки, **произошел конфликт или нет**. И только в случае, **если он произошёл, то производить действия.**

Validate():

Он проверяет, что эти узлы действительно **принадлежат списку и находятся в этом же порядке.**

Для этого мы **снова проходим по списку** до первого заблокированного элемента, **всё ещё держа эти элементы как заблокированные**. Таким образом, мы проверяем, **не удалён ли первый из двух узлов**.

Если он был удалён, то з**аблокированная пара не валидна.**

Если указатель из **pred.Next == curr,** то **искомый узел тоже не удалили.** В противном случае, **наоборот**.

Таким образом, наше **окно прошло валидацию,** поэтому мы можем выполнять над ней **любую операцию**.

Contains():

Вычисляем **значение хэша элемента key.** Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Сперва мы просто **проходим по списку** до нужного элемента **без блокировок**. При этом другие потоки, выполняющие другие методы этого списка, **могут блокировать элементы списка**. Только **когда мы находим необходимый элемент**, мы **накладываем блокировку** на **pred** и **curr.** Теперь мы **валидируем оба этих узла методом Validate().**

Если она **прошла успешно**, то **возвращаем результат поиска**.

Если вдруг оказывается, что **данные больше не валидны, то есть произошёл конфликт,** то это означает, что состояние списка с нашей точки зрения **отличается от актуального**. Тогда **снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock**. И теперь нам **необходимо повторить всё сначала**.

С одной стороны, это **выглядит затратно**, но большую часть времени мы **тратим на блокировки, которые мы не делаем тут**, поэтому подход оправдывает себя.

Add():

Вычисляем **значение хэша элемента key.** Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Аналогично **находим необходимый элемент**, затем **валидируем его.**

Если она **проходит успешно**, то возвращаем **false,** если такой **элемент уже есть в списке,** либо **добавляем новый элемент**. Тогда **снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock.**

Если валидация **проваливается**, то **прокручиваемся по списку с самого начала**.

Remove():

Вычисляем **значение хэша элемента key.** Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Аналогично **находим необходимый элемент**, затем **валидируем его.**

Если она **проходит успешно**, то **удаляем элемент из списка**.

В противном случае, **прокручиваемся по списку с самого начала**.

Таким образом, данный алгоритм **лишён главного недостатка FineGrainSet**: **он не блокирует элементы, которые никак не изменяет**.

Степень параллелизма здесь ещё выше, чем в **FineGrainSet**.

22*. Ленивая синхронизация для односвязного сортированного списка.

Алгоритм LazySet (ленивый список):

Построен на почти той же идее, что и **OptimisticSet, и тоже является оптимистичным,** однако:

1. Можно разделить удаление элемента на **логическое и физическое.** Как только мы помечаем узел **логически удалённым**, то это означает, что **когда-нибудь он будет удалён физически**, но **уже сейчас не будет учитываться при работе списка**. Пока элемент **не удалён логически, он не может быть удалён и физически**. Таким образом, удаление разделится на **два этапа**.
2. **Validate()** не предполагает ещё одного прохода **по списку до необходимого элемента**.

Поле **Marked** отвечает за **логическое удаление**.

Validate():

Не предполагает ещё одного прохода **по списку до необходимого элемента**.

Проверяет, что **pred и curr не являются логически удалёнными по Marked — это нововведение,** а также что **следующий за первым заблокированным элементом - искомый,** как и прежде.

Add():

Вычисляем **значение хэша элемента key.** Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Сперва мы просто **проходим по списку** до нужного элемента **без блокировок**. При этом другие потоки, выполняющие другие методы этого списка, **могут блокировать элементы списка**. Только **когда мы находим необходимый элемент**, мы **накладываем блокировку** на **pred** и **curr.** Теперь мы **валидируем оба этих узла методом Validate().**

Если она **проходит успешно**, то возвращаем **false,** если такой **элемент уже есть в списке,** либо **добавляем новый элемент**. Тогда **снимаем** все блокировки в том порядке, в котором и захватывали во избежание **deadlock.**

Если валидация **проваливается**, то **прокручиваемся по списку с самого начала**.

Remove():

Вычисляем **значение хэша элемента key.** Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Аналогично **находим необходимый элемент**, затем **валидируем его.**

Если она **проходит успешно**, то возвращаем **false,** если **удалить не удаётся**, либо приступаем к **операции удаления**. Сперва мы помечаем элемент **логически удалённым Marked = true.** А теперь физически удаляем его из списка при помощи указателей. Теперь **удаление успешно**.

Такой подход к удалению преследует **две цели**:

1. Мы можем **пройтись по логически удалённым в Contains()**, так как не держим никаких блокировок. В этом нет ничего страшного.
2. Если **логически удалённые** элементы будут встречены при **Validate(),** то **он провалится** и в других методах нужно будет **начинать всё сначала**.

Contains():

Вычисляем **значение хэша элемента key.**

Без каких-либо блокировок прокручиваемся до необходимого элемента.

Если **хэш найденного элемента** совпадает с **key** и найденный элемент **не помечен, как логически удалённый,** то возвращаем **true.**

Однако **Contains()** может быть вызван, когда какой-либо поток уже **захватил все необходимые блокировки и приступил к логическому удалению** элемента. Тогда метод может вернуть **неверное значение**, однако это не страшно. Это означает, что ровно в тот момент, когда мы запросили данные об этом элементе, он **ещё был в списке**.

Степень параллелизма здесь ещё выше, чем в **OptimisticSet**.

23*. Неблокирующая синхронизация для односвязного сортированного списка.

Мы рассмотрели **блокирующие алгоритмы**, где каждому узлу так или иначе **сопоставлялся примитив синхронизации.** Если возникал конфликт, и работа больше не могла быть продолжена, то она просто **прерывалась**.

Отличие **неблокирующих алгоритмов** состоит в том, что всегда есть **хотя бы один поток**, который будет **двигаться вперёд**. Остальные **не прерываются**, но **могут стоять на месте**.

В этой ситуации снова хочется использовать **атомарную операцию CompareExchange(),** однако её будет **недостаточно в чистом виде**. Введём новую структуру данных.

LockFreeSet (неблокирующий список):

Является оптимистичным и ленивым == предельно оптимистичный.

Класс AtomicMarkableReference:

Содержит **ссылку на объект в памяти initialValue** и **bool-значение initialMark,** причем оба поля обновляются **одновременно и атомарно**.

Это возможно из-за того, при выделении памяти в языках с управляемой памятью происходит **выравнивание первого байта значащих данных** в памяти **на удобную границу**. Поэтому и происходит так, что мы выделяем 1 байт, а система выдаёт 4 байта.

Благодаря этому, фактически с помощью **битовых операций значимые биты адреса соединяются с bool-значениями переменной**. Поэтому операция **CompareAndSet()** может работать **и с адресом, и с bool-значением как с единым целым**.

В нашем случае **AtomicMarkableReference** хранит **адрес на следующий элемент и bool-значение**, показывающее, **является ли элемент логически удаленным**.

Класс AtomicNode:

Хранит значение **Value,** хэш **Key** и **AtomicMarkableReference-ссылку** на следующий элемент **для текущего узла**.

Add():

Вычисляем **значение хэша элемента key**, который хотим добавить. Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Далее вызываем метод **Find()**, **находящий нужное окно**.

Если **хэш текущего найденного элемента равен искомому ключу**, то значит, что **этот элемент уже был добавлен,** поэтому возвращаем **false**.

В противном случае мы **создаем новый элемент**, передавая **значение, ключ и ссылку на следующий элемент**. Добавление снова происходит **между элементами окна**.

После этого у **предшествующего элемента pred** с помощью операции **CompareAndSet()** мы **подменяем в ссылке текущий узел curr на новый созданный node**, при этом **значение Marked остается false**.

Однако если вдруг какой-то поток **опередил нас и добавил между предшествующим и текущим узлом другой элемент**, то **CompareAndSet() не сработает и мы будем пробовать снова**. С другой стороны, если **другой поток удалил логически текущий узел**, то значит, что его **Marked** уже **true**, поэтому операция снова не сработает, и мы **начнем все сначала**.

Класс Window:

Пара из **предыдущего и текущего** узлов.

Find():

Метод, который **для данного начала списка и искомого ключа вернет пару**, где **ключ предшествующего узла будет строго меньше искомого ключа**, а **текущего узла будет больше или равен искомому ключу**.

Он возвращает объект класса **Window**.

Стоит отметить, что здесь **два цикла while(true)**.

Составляем первоначальную **пару из двух узлов**. Далее **переходим к следующему узлу**.

Если текущий узел **не отмечен как логически удаленный**, то делаем проверку на то, **больше или равен ключ текущего узла по сравнению с искомым ключом.** В случае **успешной проверки** возвращаем окно с **парой текущий элемент и его предшественник**. Если же **проверка не прошла**, то **идем по списку дальше**. Так как **у последнего элемента списка максимальное значение из всех возможных**, значит этот внутренний цикл **обязательно завершится**.

В случае если текущий узел **был отмечен как логически удаленный**, но **физически удален не был**, то мы **сами пробуем подменить узлы с помощью CompareAndSet()**, поменяв ссылку с **pred на succ и оставив в Marked = false**.

Если **операция удалась**, то таким образом мы будем **ходить по списку, удаляя физически логически удаленные элементы**.

Если же эта **операция не удалась**, это значит, что какой-то другой поток **что-то делает с этим узлом**. В этом случае мы должны **выйти из внутреннего цикла и начать проход по нему заново, перенесся удаление данного элемента на следующую итерацию при помощи proceedWithNextCycle.**

Remove():

Вычисляем **значение хэша элемента key**, который хотим добавить. Присутствует особенный для оптимистичного алгоритма цикл **while(true).**

Аналогично **ищем нужное окно** в **Find()**.

В случае, если **не нашли удаляемый элемент**, то выходим с **false**.

Дальше, если все-таки **нашли нужный элемент**, то помечаем его операцией **CompareAndSet() логически удаленным**.

Если **операция не удалась**, например, другой поток поменял порядок узлов или уже удалил этот узел, то **начинаем с самого начала**, при чем метод **Find()** найдет **актуальное на момент вызова окно**.

Если же нам **удалось отметить узел как логически удаленный**, то **удалим его физически** с помощью **CompareAndSet().** При этом мы не смотрим, **удалась операция или нет.** Нам это не важно, ведь **логически мы его уже удалили**. Если что-то пошло не так, то его **удалением** рано или поздно займется метод **Find()**.

Contains():

Вычисляем **значение хэша элемента key.**

Без каких-либо блокировок прокручиваемся до необходимого элемента.

Возвращаем результат проверки на то, что **ключ найденного узла совпадает с искомым** и **найденный узел не удален логически**.

Таким образом, этот неблокирующий алгоритм постепенно **исчерпывает все возникающие конфликты**.

Степень параллелизма данного алгоритма ещё выше, чем у **LazySet.**

24*. Алгоритмы синхронизации хэш-таблицы на списках.

BaseHashSet (множество на основе хэш-таблицы):

Абстрактный класс для реализации хэш-таблицы.

Данная хэш-таблица использует **закрытую адресацию,** то есть присутствует некоторое количество списков с хранящимися элементами, называемые **Bucket (бакет, цепочка).**

У таблицы есть **массив бакетов** и **счетчик общего количества элементов хэш-таблицы**. В конструкторе происходит их инициализация. Также есть свойство **PolicyDemandsResize** - показывающее, **нужна ли перебалансировка таблицы или нет**.

Resize(): перебалансировка хэш-таблицы.

Acquire(): захватывает все необходимые блокировки.

Release(): освобождает все занятые блокировки.

Contains():

Захватываем все необходимые блокировки.

Далее получаем **индекс нужного бакета myBucket**, в котором должен храниться искомый элемент, и проверяем, **хранится ли данный элемент внутри хэш-таблицы**. Возвращаем результат проверки.

В конце освобождаем блокировки.

Можно **перенести блокировки на уровень каждого конкретного бакета,** так как **бакеты — это списки**, а синхронизацию на них мы уже

рассмотрели. Однако здесь мы рассматриваем алгоритм **более высокого уровня**.

Add():

Захватываем все необходимые блокировки. Сбрасываем результат в false.

Далее получаем **индекс нужного бакета myBucket**, в котором должен храниться искомый элемент. Далее проверяем, что д**анного элемента в бакете нет**.

Если **его нет**, то **добавляем элемент в бакет**, **инкрементируем счетчик элементов** и **освобождаем все блокировки**.

Далее делаем **перебалансировку**, если она необходима, то есть прошла проверка **PolicyDemandsResize.**

Возвращаем **результат**.

CoarseHashSet:

Берем единственный **на всю таблицу mutex**. Естественно, минус в том, что при вызове **Add/Remove/Contains** мы будем захватывать этот **mutex**, чем **будем блокировать всю таблицу**.

Resize():

Захватываем блокировку.

Делаем **перебалансировку хэш-таблицы в один поток**.

Освобождаем блокировку.

Проблема: получается, что **Add/Remove/Contains/Resize** являются **критической секцией по одному и тому же мьютексу**.

StripedHashSet (полосатая хэш-таблица):

Улучшение алгоритма CoarseHashSet: **на каждый бакет будет свой мьютекс**.

Инициализация:

Создаём массив **mutex,** соответствующий числу бакетов, **каждый из которых предназначен для своего бакета(ов)**.

Acquire() и **Release():**

Определяем для соответствующего элемента, **мютекс какого бакета нам нужно захватить или же освободить**.

То есть **i-ому бакету в массиве бакетов будет соответствовать i-ый мьютекс в массиве мьютекса**. Однако это только до перебалансировки!

Resize():

Сохраняем **текущую ёмкость таблицы oldCapacity**.

Захватываем **все мьютексы**, то есть нужен **полный эксклюзивный доступ к таблице**.

Если выясняется, что на момент перебалансировки сохраненная нами емкость **не равна актуальной**, это значит, что **другой поток уже сделал перебалансировку**, то мы **отпускаем все мьютексы**.

Если же нас **никто не опередил**, то **удваиваем емкость и количество бакетов в новой таблице. Переносим элементы в новую таблицу**. **Отпускаем все мьютексы**.

Стоит учесть, что **количество мьютексов всегда остается неизменным после перебалансировки**. То есть после первой балансировки **один мьютекс отвечает уже за 2 бакета, после второй - за 4, и т.д.**

Проблема: по мере перебалансировки таблицы из-за того, что мы **не увеличиваем количество мьютексов**, **уровень параллелизма всегда фиксированный и не увеличивается с ростом числа бакетов.**

Однако по сравнению с **CoarseHashSet уровень параллелизма здесь выше**.

RefinableHashSet:

Улучшение алгоритма StripedHashSet: можно **изменять количество мьютексов** при перебалансировке в **соответствии с числом бакетов**, увеличивая уровень параллелизма.

При инициализации:

Создаем **массив мьютексов**, изначально равный **числу бакетов,** и **AtomicMarkableReference owner**, в котором храним **поток,** занимающийся **Resize()**, и **bool-переменную,** означающую**, делает ли кто-то Resize()**.

Acquire():

Получаем **текущий поток выполнения me.**

Далее проверяем, **нет ли другого потока, занимаюшего этот AtomicMarkableReference.** По факту это означает, что другой поток сейчас делает **Resize().**

Далее сохраняем текущий массив мьютексов **oldLocks** и определяем **соответствующий мьютекс и захватываем его**.

Если выясняется, что в тот момент, **когда мы захватывали мьютекс**, кто-то снова **начал Resize()**, то мы проверяем, что **никто другой не затребовал Resize()**, **либо же Resize() затребовали мы** и что **oldLocks равен актуальному locks,** то есть они также **не были изменены Resize()**.

Если эта **проверка проходит**, это значит, что **мы захватили актуальный мьютекс**, поэтому **выходим из метода**.

В противном случае **отпускаем мьютекс** и **повторяем действия заново,** так как **мьютекс более не актуальный**.

Release():

Отпускаем соответствующий мьютекс.

Resize():

Фиксируем в самом начале **старое состояние таблицы oldCapacity** и **новое состояние таблицы newCapacity**. Далее сохраняем **текущий поток me**.

Сообщаем, **что поток хочет начать перебалансировку с помощью операции CompareAndSet()**.

Если это **не получилось**, значит **другой поток опередил нас и уже сейчас делает перебалансировку**, тогда **заканчиваем работу**.

Если **получилось**, то проверяем **актуален ли размер таблицы.** В случае, если **неактуален**, то есть **другой поток уже успел сделать перебалансировку**, то **выходим из метода, снимая блокировку и выставляя (null, false) в owner**. Если же **никто не сделал перебалансировку, тогда ее делаем мы**, то ждем, когда все остальные

потоки **освободят все мьютексы**. После этой точки **никакой поток не будет занимать мьютекс,** так как конкретно эти мьютексы нам **больше не нужны,** но и другие потоки будут видеть, что идет **Resize(),** поэтому синхронизация не нарушится.

Дальше создаем **новый массив бакетов и массив мьютексов одинакового размера** и **перемещаем элементы из старой таблицы в новую**.

В конце **сбрасываем AtomicMarkableReference, выставляя (null, false) в owner.**

25*. Неблокирующая синхронизация хэш-таблицы на списках.

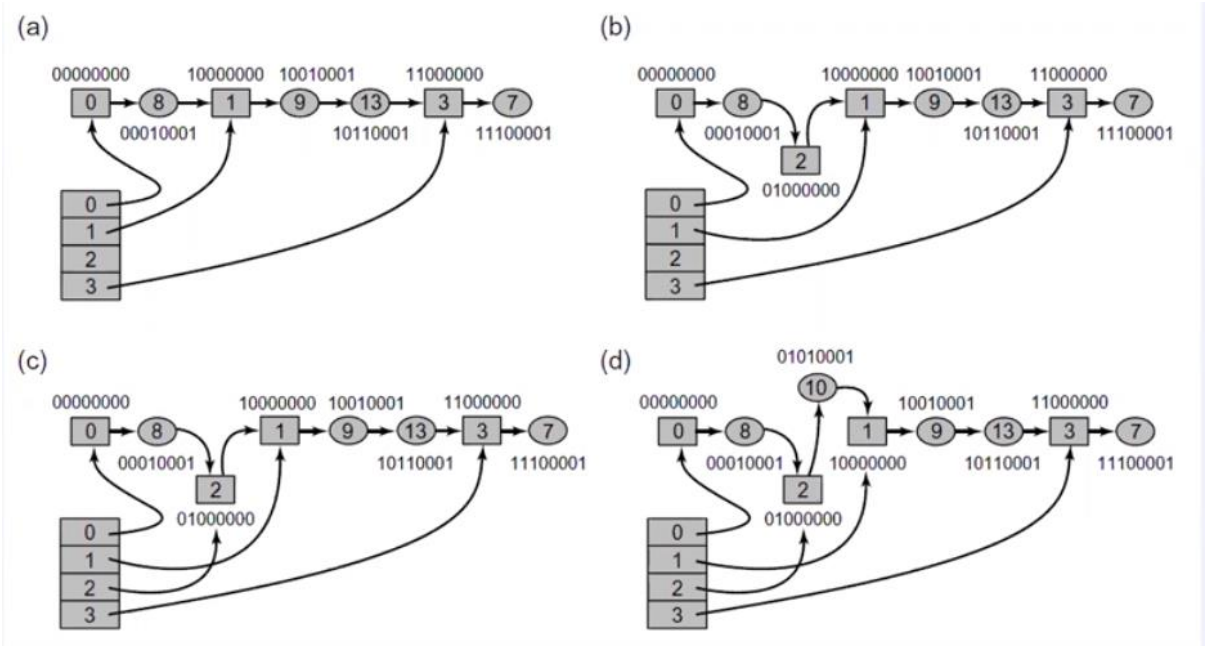
В предыдущих алгоритмах мы использовали мьютексы для **блокирования множеств на списках**. Здесь же предлагается **диаметрально противоположный подход**.

Все элементы хэш-таблицы хранятся в одном **единственном списке.**

Перебалансировка заключается не в перебрасывании **элементов между списками**, а **в перебрасывании списков вокруг элементов**, то есть сами элементы мы не трогаем. Благодаря этому **перебалансировка происходит довольно опреативно**.

Построение:

Вводим **набор опорных элементов**, на рисунке это **прямоугольники**, по факту они обозначают **номера бакетов в хэш-таблице**, а также вводим **особое кодирование**: каждому **элементу сопоставлено свое целое число - ключ**.



Будем использовать **все биты нашего ключа**, кроме **старшего**. Его мы будем считать **служебным**. Соответственно, будем использовать **только положительные ключи**. Кроме того, мы **реверсируем биты нашего ключа** и для всех элементов таблицы теперь уже **младший служебный бит ставим в 1**.

Пример:

8 00001000

8 **0**0001000 // служебный бит

8 0001000**0 //** реверсируем ключ

8 0001000**1 //** выставляем служебный младший бит в 1

В прямом представлении ключа используются только **7 бит, то есть кроме старшего**. В реверснутом представлении ключа используются **все 8 бит**.

Аналогично делаем **с опорными элементами**, но у них **служебный бит будет строго равен 0.**

Таким образом, элемент **8 попадёт в бакет 0**, так как остаток от деления 8 на 4 равен 0, а затем над ним будет проведена **указанная операция по составлению ключа, как над обычным элементом списка,** то есть служебный бит будет равен **1.** При соблюдении всех условий получается, что данный список оказывается **отсортированным по возрастанию**.

Добавление:

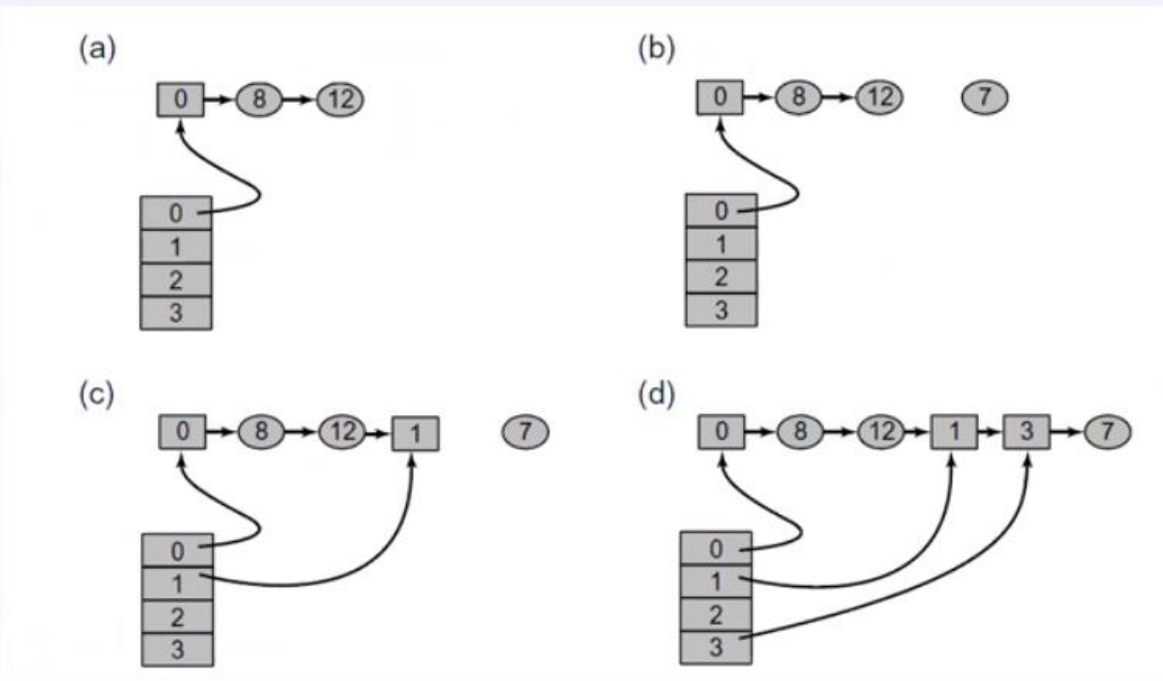
При добавлении элемента, остатка от деления на размер таблицы которого **нет в таблице бакетов, (**у нас это 10, остаток от деления на 4 = 2, такого бакета у нас нет**)** мы **сперва вставляем с правильным ключом опорный элемент, соответствующий новому остатку от деления на размер таблицы**, у нас это **2 с ключом 01000000.** Затем **прокидываем ссылку на новый бакет из таблицы бакетов**. Теперь **вставляем 10 в нужный бакет с ключом 01010001.** Таким образом, добавление **сохраняет свойства отсортированности по возрастанию**.

Поддерживая список в таком отсортированном состоянии на уровне хэш-таблицы, мы можем **перенести все проблемы синхронизации на уровень ниже**, то есть **на уровень списков**, и применить уже известные нам методы. Таким образом, задача построения хэш-таблицы **редуцируется с задачей построения алгоритма неблокирующей синхронизации для множества на списке**.

Перебалансировка:

Число бакетов — это всегда степень двойки, так как мы всегда увеличиваем их **число на 2**. Таким образом, мы можем добавлять либо все новые бакеты сразу, **либо по необходимости**. Здесь рассматривается именно **второй вариант**.

Здесь показаны просто элементы **без ключей.**



При добавлении элемента 7 мы понимаем, что **бакетов не хватает**, поэтому мы **вставляем в таблицу бакет 1**, дальше дробим его, **вставляя бакет 3 в таблицу,** и **после этого добавляем элемент 7**.

Если **размер таблицы 2^n**, то мы можем добавить **$\log_2$(N) новых бакетов при добавлении нового элемента,** если нам это нужно.

Перебалансировка фактически будет **увеличением количества бакетов вдвое** и **добавлением их в структуру**. Таким образом, дополнительных операций по типу **перемещения элементов не требуется,** за счёт чего происходит **ускорение работы этого алгоритма**.

Класс BucketList:

Заметно повторяет **LockFreeSet,** то есть представляет из себя **неблокирующую реализацию множества на списке**.

Мы используем **два вида узлов: опорные узлы и обычные узлы**.

При создании **нового бакета** мы создаем в качестве родительского узла **ключ 0** и **AtomicMarkableReference с AtomicNode.** Это будут **два инварианта данного списка**.

GetSentinel():

Получаем **ключ key** используя метод **MakeSentinelKey().**

Далее ищем **окно Window при помощи метода Find().** Проверяем, есть ли **данный элемент в списке или нет**.

Если **элемент есть, то возвращаем его.**

Если **его нет**, то **пытаемся его добавить node**, используя привычную операцию подмены **CompreAndSet()**.

Если это удалось, то **возвращаем бакет**.

В противном случае, как в оптимистическом алгоритме, **крутимся дальше**.

MakeSentinelKey():

Накладываем на **индекс бакета маску**, **выставляющую старший бит в 0**, и **реверсируем биты полученного результата**.

Contains():

Добавляем к вычислению хэша вычисление ключа при помощи MakeOrdinaryKey().

Ищем окно, **возвращаем результат поиска**.

MakeOrdinaryKey():

Получаем **хэш-код элемента**, **накладываем маску** и **реверсируем биты**, **выставляя старший бит в 1**.

Add():

Как в **LockFreeSet** списке.

Find():

Как в **LockFreeSet** списке.

Класс LockFreeHashSet:

Инициализация:

При инициализации создаем **массив бакетов заданной ёмкости.** Считаем, что изначально **бакетов 2**, **элементов в таблице 0**. То есть подготавливаем массив на большее число бакетов, но изначально используем не все. Работаем **по необходиомсти**.

Add():

Вычисляем хэш для элемента **myBucket**. Далее **достаем соответствующий бакет по хэшу** с помощью метода **GetBucketList.**

Если его нет, то **инициализируем** методом **InitializeBucket().**

Далее, возвращаем **false**, если нам **не удается добавить элемент в бакет**, например, он там уже есть.

Если **элемент добавили**, то **инкрементируем число элементов и проверяем**, **нужна ли перебалансировка**.

Если **нужна**, то пытаемся **подменить при помощи CompareExchange() количество бакетов на удвоенное**. Если этого **не удалось**, то есть кто-то нас опередил, поэтому это не страшно.

Затем возвращаем **true**.

Contains():

Делается по сходной схеме, однако **нет части с перебалансировкой.**

Remove():

Делается по сходной схеме, однако **часть с перебалансировкой не обязательна.**

InitializeBucket():

На основе **идентификатора бакета** получаем **родительский бакет методом GetParent()**.

Если **он null**, то **рекурсивно инициализируем и его**.

Если же он **не null**, то методом **GetSentinel() берём у него опорный элемент** и создаем новый **BucketList** или **возвращаем его, если он уже был создан.**

Теперь **добавляем этот бакет в таблицу**. **Присваивание не делаем защищенным**, так как у нас либо будет **null в таблице бакетов**, либо **это будет один и тот же узел**.

GetParent():

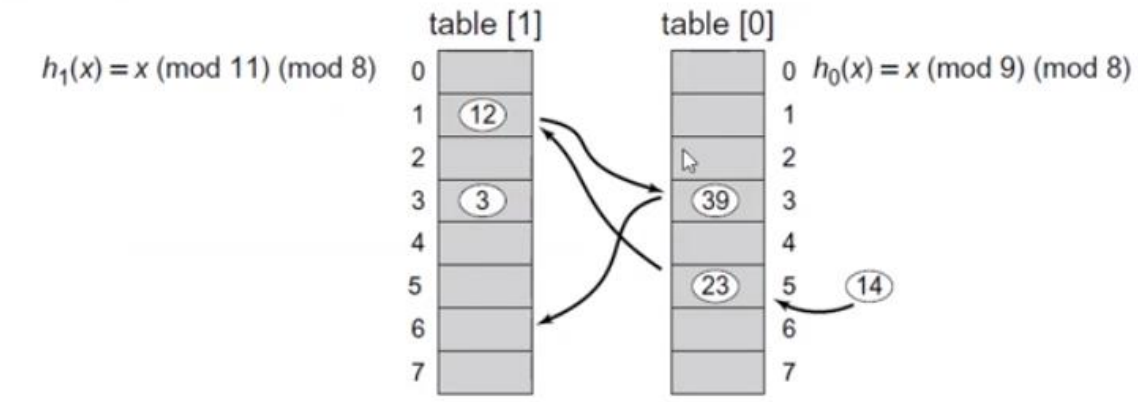
По идентификатору с помощью битовых сдвигов определяется **родительский бакет**.

Таким образом, **на уровне хэш-таблицы синхронизацию мы используем только во время атомарного увеличения счетчика бакетов, то есть при перебалансировке**. То есть все проблемы синхронизации мы **перенесли на уровень ниже, то есть на уровень бакетов.**

26*. Алгоритмы синхронизации хэш-таблицы с открытой адресацией.
Теперь будем рассматривать алгоритмы **с открытой адресацией.**

CuckooHashing (кукушечное хэширование):

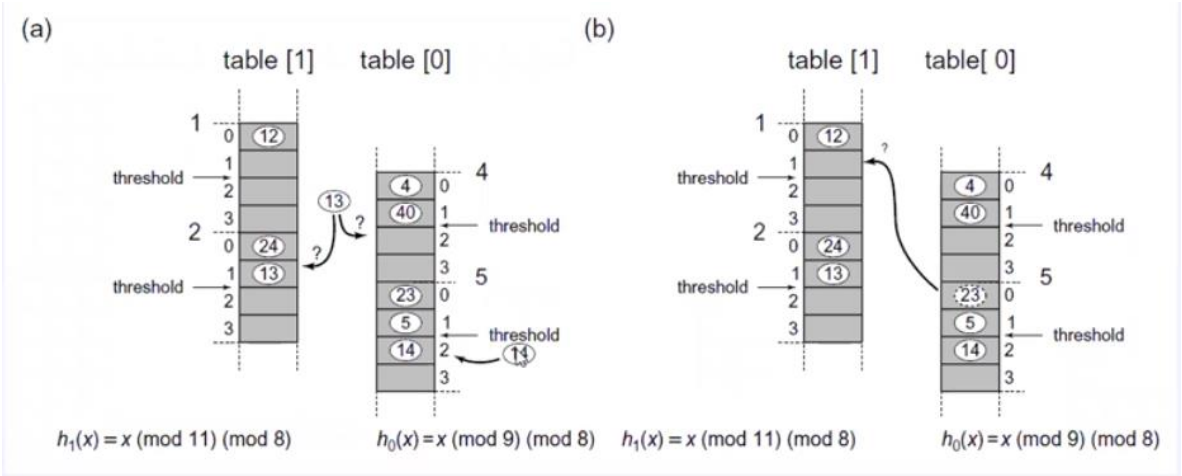
Однопоточный случай:



Есть **не один массив**, а **два массива в таблице**. Для каждого из массивов **своя функцию хэширования**. В каждом массиве индекс одного и того же элемента **будет разным**. Если мы добавляем элемент в массив 0 и соответствующая ячейка **оказывается занята**, то мы **выталкиваем элемент этой ячейки в другой массив**. **Ситуация повторяется и для массива 1**. Циклическое выталкивание продолжается, **пока элемент не найдёт себе место в массивах**.

Проблема: такой цикл выталкивания может оказаться **бесконечным**, однако **вероятность этого очень мала**. Эту проблему можно решить путём **перебалансировки всей таблицы** в таком случае.

Многопоточный случай:



Вместо того, чтобы держать в массиве **одну ячейку на каждый элемент**, будем хранить **список, своего рода блок, из нескольких элементов.** Например, **4 элемента**, как в примере.

Также введем **две метрики**: **частичная и полная заполненность списка**. Если в списке **0 или 1 элемент**, то список **мало заполнен**, если **2 или 3 элемента**, то **заполнен средне**, если **4**, то **заполнен полностью**.

Добавление:

При **добавлении элемента в хэш-таблицу**, определяем, **какой из списков в соответствующей ячейке в обоих массивах менее заполнен**, **в тот и будет добавлять**.

Если же списки **заполнены одинаково**, но **не до конца**, то **добавляем в любой**.

Если оба списка **заполнены полностью**, то нужно **перебалансировать,** чтобы не случилось коллизий.

PhasedCuckooHashSet:

Будем использовать **двумерный массив списков**. Также введем **константы**, показывающие степень наполнения списка: **THRESHOLD = 2 - заполнен наполовину**, **LIST_SIZE = 4 - заполнен полностью, а также число элементов в списках**, а также **LIMIT = 5**, показывающий, что, **если на протяжении 5 итераций перебрасывание элементов не завершилось,** значит **ушли в бесконечность** и **нужно увеличивать таблицу из-за сильных коллизий**.

Также вводим **две хэш-функции**.

Contains():

Берем **блокировку для элемента.**

Дальше смотрим, содержится ли элемент **в соответствующем списке нулевого массива.** Если **нет**, то **первого массива**.

Если нигде не нашли, то **его нет**.

В конце **снимаем** блокировку.

Remove():

Берем блокировку для удаляемого элемента.

Дальше смотрим, **содержится ли элемент в соответствующем списке нулевого массива.** Если **содержится**, то **удаляем его из списка**.

Если же **нет**, то смотрим, **содержится ли во первом массиве,** и **если да**, то **удаляем элемент оттуда**.

В противном случае **не делаем ничего**, так как такого элемента в таблице **нет**. Снимаем блокировку.

Add():

Получаем **блокировку для элемента**, далее **считаем оба хэша**.

Далее если **элемент уже содержится в хэш-таблице**, то **выходим**.

В **противном случае получаем по соответствующему списку из каждого массива**.

Теперь проверяем, **заполнен ли нулевой список на половину**, если **нет, то добавляем элемент**. Если **заполнен**, то аналогично **проверяем первый список**, если же и **туда добавить не удалось**, то **проверяем оба списка на то, заполнены ли они полностью** и **добавляем элемент в список, в котором место есть**, при этом **сохраняем номер таблицы и номер ячейки**, чтобы потом **этот список "разгрузить"**.

Если же **оба списка заполнены полностью**, то значит **нужно увеличить размер таблицы, mustResize = true**. После этих действий мы **снимаем блокировку** и **проверяем, нужно ли делать Resize() таблицы**.

Если **да**, то делаем **Resize()** и **снова добавляем элемент в таблицу**.

Если **Resize() делать не нужно**, то делаем **разгрузку списка методом Relocaate()**, если она **не удалась**, то делаем **Resize()**.

Если всё удалось, возвращаем **true**.

Relocate() (разгрузка списка):

Определяем **противоположную таблицу**, **в которую мы будем разгружать**.

Далее **делаем не более чем LIMIT = 5 попыток на разгрузку**.

На каждой итерации мы **делаем следующее:**

Получаем **нужный список** из того массива**, который мы разгружаем** и **берем его нулевой элемент**. Дальше мы **определяем индекс списка, который примет этот элемент в другой таблице**. Далее **получаем** блокировки.

Теперь **пытаемся удалить элемент из списка, который разгружаем.**

Если это **не получилось**, то есть **другой поток успел удалить этот элемент**, то мы **проверяем, загружен ли этот список как минимум на половину**, и если **это так**, то **продолжаем разгружать список**, в **противном случае выходим из метода с true**.

Если мы **смогли удалить элемент из разгружаемого списка**, то мы **проверяем**, что **другой список**, **в который мы загружаем**, **заполнен меньше, чем на половину**, и **если это так**, то **добавляем в него элемент и заканчиваем с true**.

Если же этот список **заполнен больше, чем на половину**, **но не полностью**, то **переносим проблему другой на этот список**, то есть теперь мы **разгружаем этот список** и **идем на следующую итерацию, чтобы теперь разгружать его.** Таким образом, если **за 5 итераций мы не смогли разгрузить список**, то нужно делать **Resize() таблицы**.

Если же этот список **заполнен полностью**, то **возвращаем элемент назад в старый список и выходим с false**.

StripedCuckooHashSet (полосатая кукушечная хэш-таблица):

Инициализация:

Заводим **двумерный массив мьютексов**, **размер соответствует размеру исходной таблицы**.

Acquire():

Вычисляем **значения хэшей для обоих массивов** и **с помощью остатков от деления определяем нужные два мьютекса и занимаем их**.

Release():

Аналогично **Acquire() отпускаем два мьютекса**.

Resize():

Для того, чтобы сделать перебалансировку, **сохраняем старый размер таблицы** и **занимаем все мьютексы из нулевого массива**. Обусловлено это тем, что если мы получили блокировку **на мьютексы нулевого массива**, то гарантированно **мьютексы для другого массива не займут,** так как мы **сперва получаем все блокировки на нулевой массив, а затем на первый**.

Далее проверяем, не опередил ли нас другой поток **через сравнение сохраненного ранее размера с текущим**.

Если **да,** то **отпускаем все мьютексы и заканчиваем**.

Если **нет**, то мы **обладаем эксклюзивным доступом ко всей хэш-таблице**, то можем сделать **в однопоточном режиме перебалансировку**, **увеличив**

размер таблицы вдвое, **создав новую таблицу со списками** и **перебросив элементы**.

В конце **отпускаем мьютексы**.

Проблема: здесь также **один и тот же мьютекс после перебалансировки отвечает за несколько списков**.

RefinableCuckooHashSet:

Инициализация:

Заводим **двумерный массив мьютексов**, **размер соответствует размеру исходной таблицы**.

Добавляем поле **AtomicMarkableReference owner - поток, занимающийся Resize(), и bool состояние, занимается ли кто-то вообще этим в данный момент**.

Acquire():

Получаем **текущий поток**, а также в цикле **поток, занимающийся перебалансировкой**, если такой есть, и **состояние перебалансировки mark. Проверяем,** если сейчас **другой поток делает перебалансировку**, то мы **ждем ее завершения**.

Далее **сохраняем мьютексы, получаем два соответствующих мьютекса и занимаем их**.

Теперь снова смотрим, **не делается ли сейчас Resize()**. Если **происходит перебалансировка нами или не происходит вовсе**, и **текущий массив мьютексов актуален**, то **выходим**.

В противном случае освобождаем мьютексы и **начинаем все сначала.**

Release():

Освобождаем два соответствующих мьютекса.

Resize():

Получаем **текущий поток и сохраняем размер таблицы.**

Дальше пытаемся с помощью **CompareAndSet() сообщить**, что мы начинаем **Resize()**, **поменяв в переменной owner поток и bool значение на true**.

Если это **не удалось сделать**, это значит, что **другой поток нас уже опередил**, тогда **заканчиваем работу**, сбрасывая **owner** с помощью **Set(null, false)**.

Если **удалось установить право на перебалансировку**, то проверяем, не успел ли кто-то **уже сделать перебалансировку через сравнение с сохранённым ранее размером таблицы**.

Если **другой поток уже сделал перебалансировку**, то **выходим из метода**, сбрасывая **owner** с помощью **Set(null, false)**.

Если же **мы делаем перебалансировку**, то точно также **получаем блокировки только на один массив**, **предотвратив захват блокировок другими потоками** и **отпускаем мьютексы, так как они уже не нужны**. Другие потоки, в свою очередь, **не смогут занять эти мьютексы, так как сейчас уже происходит перебалансировка**. То есть **снова получили эксклюзивный доступ ко всей хэш-таблице**.

Далее **увеличиваем емкость таблицы для данных** и **массива мьютексов вдвое**, **создаем и заполняем их**. Есть тонкость, **при добавлении Acquire() пройдет**, так как **перебалансировкой занимаемся именно мы**.

27. Понятие об асинхронности. Понятие Future/Task. Ключевые слова async и await в языке C#.

Асинхронность в программировании — выполнение процесса в неблокирующем режиме системного вызова, что позволяет **потоку программы продолжить обработку**.

Future - особый класс в ООП языках, предназначенный для реализации **отложенных вычислений**. При создании класса ему передается **делегат**, запуск это делегата происходит **асинхронно**. В **.Net** класс представлен библиотекой **Task Parallel Library**.

Есть **свойства**, **показывающие**, **начались ли вычисления**, **закончены ли вычисления, произошло ли исключение; поле результата, поле исключения, поле делегата, поле потока.**

В **конструктор передаем делегат с методом**.

Отложенность в получении результата вычислений можно построить, если **сначала проверять готовность результата**, которая не будет тормозить проверяющий поток, а **только в случае готовности обращаться**

за результатом, так как в этом случае вызывающий поток бы ждал завершения вычислений.

После запуска метода **Start()** создаются **дополнительные потоки**, которые и исполняют переданный в делегат метод. Потоки берутся из **.Net ThreadPool - пула потоков,** так как создавать каждый раз новый поток на небольшие задачи - **слишком затратно**.

Async/Await - ключевые слова.

Async - обозначает, что **метод является асинхронным**. Такой метод **не должен ничего возвращать**.

Await - можно использовать **только в async методах.** Это ключевое слово заставляет **вернуться вызывающему await потоку из async метода**, но при этом **другой поток,** вероятней всего тот, который мы и ожидаем**, продолжает исполнение данного метода**. Если **вызывающий await поток** существует **только в контексте этого метода, то он завершается в async методе,** но также при этом **другой поток,** вероятней всего тот, который мы и ожидаем**, продолжает исполнение данного метода**.

28*. Понятие о пуле потоков. Дисциплины обработки задач Work Stealing и Work Sharing.

Пул потоков – автоматизированный ссылочный тип-контейнер ссылок на рабочие потоки на языке C#. По сути, выполняет роль посредника между планировщиком задач операционной системы и потоками, реализованными в рамках **.NET-платформы**.

Достаточно часто возникают задачи, связанные с **общей очередью из задач, поэтому их необходимо уметь решать.**

Существует подход, когда **поток-диспетчер распределяет задачи по очередям задач на каждый поток по каким-либо соображениям.** Этот подход не самый лучший, так как **какие-то задачи могут быть быстрее, другие медленнее**. Поэтому необходимо добавить к нему улучшения.

Work Stealing (похищение работы) и Work Sharing (разделение работы) - способы **перебалансировки нагрузки между потоками,** когда выясняется, что существует дисбаланс. При этом **сами потоки** пытаются перебалансировать нагрузку.

Интерфейс IDEQueue (двусторонняя очередь):

Это не очередь в классическом понимании. В этой очереди есть **два указателя Top и Bottom**.

Вставка PushBottom() всегда происходит со **стороны Bottom**. **Рабочий поток из Bottom забирает задачи методом PopBottom()**. Таким образом, очередь работает для потока **как стек.**

Считается, что **если задача была добавлена в пул потоков**, то **порядок обработки задач не гарантируется**, а ожидается, что задача должна быть выполнена **хоть когда-нибудь**.

Получение доступа к **Top стороне очереди методом PopTop()** делается в целях **WorkStealing**.

PopTop() - метод для **потока-вора**.

PopBottom() - метод для **рабочего потока**.

AtomicStampedReference:

Хранит **некоторое value - значение и int stamp - некоторое число.**

CompareAndSet():

Пытается **установить атомарно и новое значение, и новое число, ожидая сначала увидеть то или иное значение и число**.

Если удалось, то возвращает **true.**

Если это **не удалось**, то значит **другой поток изменил ожидаемое состояние объекта**. Возвращает **false**.

WorkStealingThread:

Описание работы **потока-вора**.

Очереди потоков хранятся **в словаре по номеру потока и соответствующей очереди**.

Run():

Считается, что метод **сам выполняется в каком-то потоке и работает до бесконечности**.

Получаем **номер потока** и **задачу из очереди этого потока**.

Если задача успешно получена, то мы **работаем с ней, пока не закончим**. Затем снова **получаем задачу из своей очереди**.

Как только задача в очередной раз **получена не была, то есть очередь данного потока пуста,** то сообщаем операционной системе, что **можно переключаться на другой поток**. Теперь определяем жертву **victim**, **у которой и будем воровать**.

Если **очередь жертвы не пуста**, то **пытаемся своровать задачу у него**.

Если **удалось**, то **работаем над этой задачей**.

Если **нет**, то **попробуем своровать снова**.

BDEQueue (ограниченная двусторонняя очередь):

Рассмотрим случай, когда в **DEQueue не может быть более ограниченного числа задач.**

Как мы уже увидели, **двусторонняя очередь построена на разделяемой очереди между потоками-ворами и потоками-рабочими**.

Инициализация:

Создаём **массив задач конечной емкости**.

Поле **bottom** - показывает, какой **элемент надо изъять рабочему потоку.** Оно **volatile**, так как эти изменения должны видеть все потоки. По факту указывает первый свободный элемент.

Поле **AtomicStampedReference top** - показывает, **какой элемент можно своровать грабителю**.

Изначально и **top и bottom указывают на 0**.

PushBottom():

Помещаем задачу в массив заданий и **атомарно инкрементируем bottom**.

Фактически здесь работаем **со стеком, а не очередью**.

IsEmpty():

Получаем **текущие значение top в oldTop.**

Если **bottom меньше или равен oldTop**, то есть **указатель стека находится ближе к началу, чем указатель очереди**, то это означает, что **очередь пустая**.

В противном случае, это означает, что **хоть одна задача есть, значит можно ее извлекать**.

PopBottom():

Синхронизации здесь нет, так как этот метод вызывает **только один поток, который владеет очередью.**

Если **bottom равен 0**, то есть **очередь полностью пустая**, то мы **возвращаем null**.

В противном случае декрементируем bottom и **берем последнюю задачу для рабочего потока**.

Далее получаем **текущее значение top – oldTop и текущее значение штампа oldStamp.** Получаем **инкрементированием** значение **newStamp**.

Если **все еще значение bottom больше top**, это означает, что **два конца очереди смотрят на разные элементы**, то **есть очередь не пуста, то возвращаем задачу**.

Если **очередь пустая**, то **сбрасываем bottom в 0, чтобы не выйти за пределы ограниченного массива.** Пытаемся сбросить **top в 0 с изменениями нового штампа newStamp при помощи CompareAndSet()**.

Если **получилось, то возвращаем задачу**.

Если это **не удалось**, это означит, что **другой поток уже своровал у нас задачу**. Теперь **сбрасываем top в 0 с новым штампом newStamp**. Возвращаем **null**.

PopTop():

Синхронизации здесь нет, так как этот метод вызывает **только один поток, который владеет очередью.**

Далее получаем **текущее значение top – oldTop и текущее значение штампа oldStamp.** Из них получаем **инкрементированием** значения **newTop и newStamp**.

Если значение **bottom меньше или равно oldTop**, это значит, что **очередь пустая**. Возвращаем **null**.

В противном случае берем задачу, и если нам удается заменить **oldTop на newTop и oldStamp на newStamp**, то **возвращаем задачу**, то есть мы ее своровали.

Если же **это не удалось**, то возвращаем **null**.

Данный алгоритм корректен со стороны взаимодействия **PopTop() и PopBottom().**

Проблема: **работа с массивом задач и полем bottom происходит неатомарно**.

Зачем нам вообще **нужен stamp**?

Проблема ABA: поток **сохраняет локально состояние системы во время работы**, затем операционная система **останавливает поток. После возвращения из сна поток не проверяет, изменилось ли состояние системы, и это приводит к проблемам.**

Если в данном алгоритме не использовать stamp, то может быть следующее:

Поток в **PopTop() взял задачу и до выполнения CompareAndSet() и был уведен в сон**.

За это время **задачи могли обновиться в очереди**. В таком случае, **состояние Top глобально уже неактуально,** и **поток-грабитель получит неактуальную задачу**, которой уже **фактически может и не быть в очереди**.

Чтобы этого не произошло, мы вводим дополнительно **штамп**.

Класс CircularArray:

Представляет **круговой (циклический) массив**. Данная структура **не является потокобезопасной**.

В этом массиве в любой момент времени может содержаться **не более чем Capacity = $2^{logCapacity}$ элементов за счёт битового сдвига влево**.

Инициализация:

В конструкторе создаем массив задач с **начальной указанной емкостью.**

Get():

Чтобы **получить элемент по индексу**, берем **его остаток от деления на Capacity**.

Put():

С добавлением элемента **аналогично**.

Resize():

Принимает **bottom и top**, то есть это **текущий диапазон, в котором находятся данные в массиве**.

Создаем **новый массив с увеличенной на единицу ёмкостью, что приведет к реальному увеличению Capacity в 2 раза,** и **копируем задачи из старого массива в новый**.

UnboundedDEQueue (неограниченная двусторонняя очередь):

Рассмотрим случай, когда в **DEQueue может быть неограниченное число задач.**

Содержит **CircularArray в качестве массива задач**.

Инициализация:

Создаём массив задач.

IsEmpty():

Аналогично **BDEQueue** с ограниченной очередью.

PushBottom():

Сохраняем значения **top и bottom**.

Если определяем, что **вставлять будем уже за границу массива**, то делаем **Resize()**.

Дальше помещаем в **массив новый элемент** и **инкрементируем bottom**.

PopBottom():

Декрементируем bottom, **сохраняем текущее значение top, получаем newTop.** Считаем размер **size**.

Если **размер size меньше 0**, то есть мы **пытаемся изъять элемент из пустой очереди**, то приравняем **bottom к oldTop и заканчиваем работу**.

Если же **очередь не пуста**, то **получаем задачу**, и **еще раз смотрим на size**.

Если он **больше 0**, то **возвращаем задачу**.

В противном случае это значит, что **size = 0**, значит **пробуем атомарно поменять top на newTop при помощи CompareExchange().**

Если **она не прошла**, то есть **нас опередили**, то в качестве задачи **ставим null**.

Далее **инкрементируем bottom и возвращаем задачу**.

PopTop():

Сохраняем текущие top и bottom. Создаем новый newTop. **Определяем size.**

Далее смотрим, **есть ли элементы в массиве**.

Если **нет**, возвращаем **null**.

Если **есть**, то **берем по oldTop задачу и пытаемся заменить top на newTop при помощи CompareExchange()**, то есть **продвинуть вперед указатель очереди**.

Если это **получилось**, то есть **мы своровали задачу**, то **возвращаем ее**.

В противном случае null, так как **нас опередили**.

Почему здесь **нет stamp**?

Здесь **нет необходимости** использовать **штампы**, так как **мы не сбрасываем значение oldTop**, а **идем только вперед за счёт CircularArray.** Таким образом, здесь **решена проблема ABA, потому что пробудившийся поток точно заметит смену контекста.**

Мы рассмотрели ситуацию, когда поток остаётся без задач, то он ворует её у другого потока.

Рассмотрим теперь ситуацию, когда каждый поток **может поменять текущее состояние всей системы**.

WorkSharing - вероятностный алгоритм, то есть чем **меньше элементов, тем больше шанс, что поток начнет перебалансировку**.

WorkSharingThread:

Описание **работы потоков в этом случае**.

Очереди потоков хранятся **в словаре по номеру потока и соответствующей очереди**.

Run():

Считается, что метод **сам выполняется в каком-то потоке и работает до бесконечности**.

Определяем **номер текущего потока**.

Далее **рабочий поток пытается из своей очереди получить задачу по своему номеру** и **запускает** её, **если она не null**.

Далее **поток смотрит на текущий размер своей очереди size.** Если **случайное число** от **0 до size** стало **равно size**, то **случайным образом определяем "жертву" из очереди**. Чем больше элементов, тем ниже вероятность.

Далее определяем **min и max, как victim и me, либо me и victim и получаем блокировки на обе очереди.** Начинаем **перебалансировку**.

Balance():

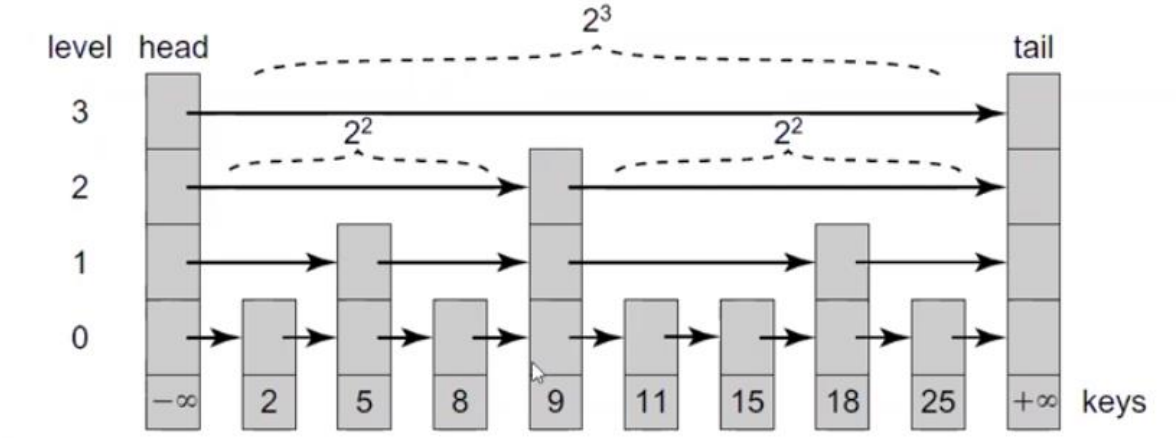
Определяем, **у какой очереди меньше задач и разницу между ними diff.**

Пока разница между количеством элементов в двух очередях больше ограничения THRESHOLD, перемещаем задачи из очереди с большим количеством задач в очередь с меньшим до того, пока они не уровняются.

В итоге, если у потока **слишком много задач, то он отдаёт их кому-то ещё с небольшой вероятностью**. А если у потока **задач нет**, то **он попытается их у кого-то взять**.

29*. Понятие списка с пропусками (skip list). Ленивая синхронизация для списка с пропусками.

Skip list (список с пропусками/скиплист) - вероятностная структура данных, которая используется для **оптимизации работы со списком,** путём добавления **дополнительной поисковой структуры поверх существующего списка**.



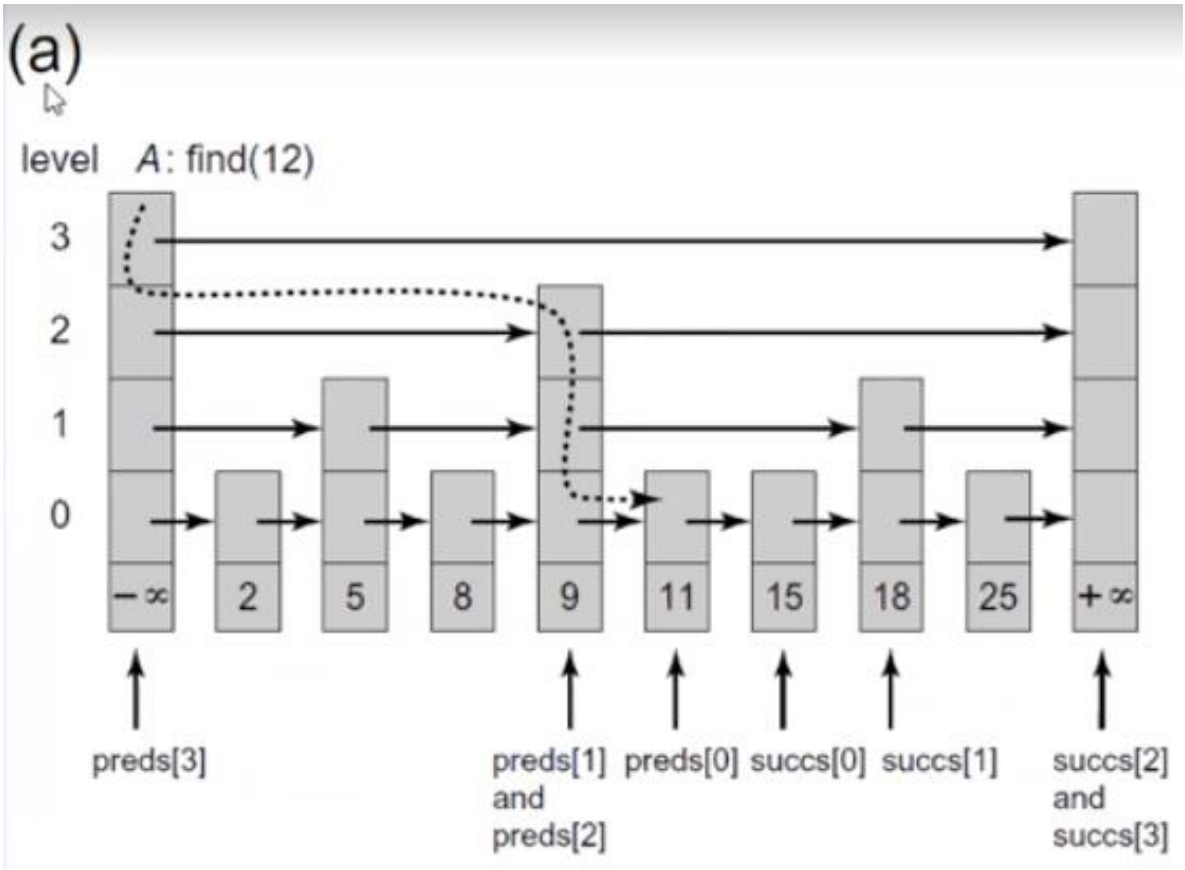
Каждому узлу в соответствие ставится **не только ссылки на следующие узелы**, но и **уровень, задающий число ссылок на последующие узлы**. **Максимальный уровень скиплиста задается при инициализации**. Однако вероятность того, **что у узла будет тот или иной уровень, падает с увеличением уровня**.

Когда мы **добавляем новый узел**, мы с вероятностью **2^x надстраиваем очередной узел, где x - текущий уровень**. Причем **не более максимального уровня скиплиста**. Как только выпадает **отказ к увеличению уровня - останавливаемся**. **Уровни начинаются с 0**.

На нулевом уровне списка мы организовываем **как обычный список, так как этот уровень точно будет у всех узлов**. На первом уровне мы организовываем список **только из тех элементов, у которых есть первый уровень.** На **втором и остальных уровнях аналогично**.

Вводим привычный **инвариант - опорные узлы,** у которых присутствуют **все урони вплоть до максимального. Первый опорный узел хранит элемент -∞, второй опорный узел** +∞, то есть наш **скиплист будет отсортирован по возрастанию**.

Поиск:

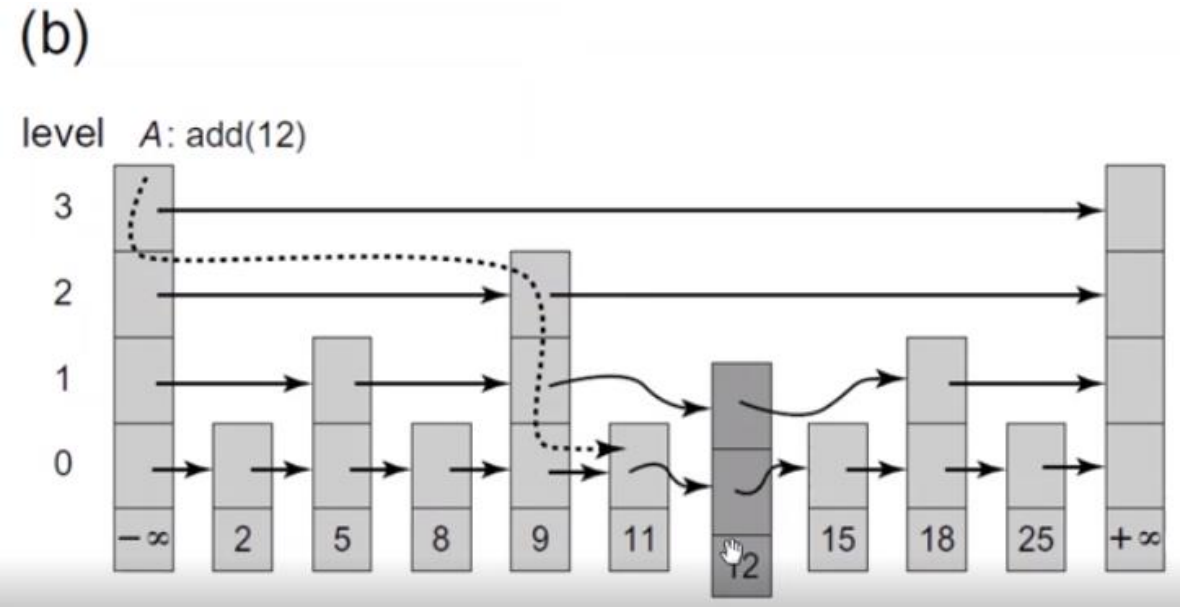


При поиске элемента начинаем **с самого высокого уровня.**

Если **текущий узел меньше**, а **последующий больше искомого элемента, то на этом уровне данный элемент мы точно не найдём**, поэтому **спускаемся на уровень ниже**. Таким образом действуем, **пока не найдём нужный элемент**.

Если **спустились на уровень 0 и не нашли там элемента**, значит **во всем скиплисте его нет**.

Добавление:



При помощи **поиска находим необходимое место на 0 уровне**, куда надо вставить элемент.

Далее **случайным образом определяем**, сколько **уровней будет у нового узла**. Например, у него **есть ещё уровень 1**.

Следовательно, нам нужно **обновить ссылки на всех уровнях данного узла**, которые у него есть. Эта задача нетривиальна.

LazySkipList (ленивый скиплист):

Класс Node:

Представляет **элемент скиплиста**.

Чтобы **обеспечить синхронизацию**, **с каждым узлом сопоставляем мьютекс**. Соответственно **Lock() будет его закрывать**, а **Unlock() - открывать**. Также храним **ключ key** и **значение item для узла** и **массив ссылок на другие элементы**. Размер этого массива **устанавливается при создании узла случайным образом.**

Поле Marked - отметка о **логическом удалении**.

Поле FullyLinked - истина в случае**, если процедура добавления элемента в скиплист произошла успешно на всех уровнях**.\

Также есть **два конструктора - для опорных и обычных узлов**. Для **двух опорных узлов начала и конца скиплиста** используем какие-то **значения по умолчанию**. Также для них устанавливаем **максимальное число**

уровней. Для **обычны узлов** максимальный уровень устанавливается **в соответствии с передаваемым**.

Инициализация:

В самом **LazySkipList** мы создаем **опорные узлы head и tail, с максимальным и минимальным допустимыми значениями, указывая в конструкторе, что head ссылается на tail на всех уровнях**.

Contains():

Создаём **узлы pred - предшествующий узел и succs - текущий узел, охватывающие все уровни, для хождения по скиплисту.**

Вызываем метод **технический метод Find()**.

Далее проверяем, что **lFound не равен -1.** Если это не так, это значит, что узел **не был найден нигде в списке**, тогда вернем **false**. Также проверяем, что этот **узел полностью связан и логически не удален**.

Если **все проверки выполнились**, значит вернем **true**.

Find():

Здесь мы заполняем два массива **предшествующих и последующих узлов**. Также передаем **элемент, который хотим найти.**

Вычисляем **ключ от элемента**.

Далее, в цикле мы идём от **максимального уровня к нулевому, от head**. На каждой итерации мы **считываем последующий узел на текущем уровне**. И делаем так, **пока ключ искомого узла больше ключа текущего узла**. Таким образом мы **проходим по текущему уровню**.

Если мы **нашли узел** и при этом **никто ещё не отметил, на каком уровне нашли этот узел**, то отмечаем с помощью переменной **lFound**, **на каком уровне мы нашли узел**.

Так **заполняем все массивы всеми необходимыми узлами, спускаясь до самого конца.**

Метод возвращает **уровень, на котором впервые этот элемент был найден**. **В противном случае, -1**. При этом в методе были **найдены preds и succs, с которыми и будет вестись работа в других методах.**

Здесь мы **не смотрим ни на какие ограничения для узлов** в плане, удален ли логически узел, полностью ли он связан. Нам необходимо лишь найти нужное место или узел,

Add():

Определяем **количество уровней topLevel у добавляемого элемента случайным образом.** По хорошему, здесь должно быть **экспоненциальное определение вероятности.**

Создаём **узлы pred - предшествующий узел и succs - текущий узел, охватывающие все уровни, для хождения по скиплисту.**

Далее мы вызываем **Find()**. Если мы **нашли узел**, то **берем этот узел** и, если он **не помечен как логически удаленный**, то он **либо уже добавлен, либо его вставляют прямо сейчас**. Тогда **в цикле будем ждать, пока** его не вставят на всех уровнях и **он не станет полностью связным,** чтобы не предоставлять противоречивые данные. И тогда вернем **false**, так как **мы не вставили этот элемент**.

Если же найденный элемент в списке **отмечен как логически удаленный**, значит **скоро его удалят и физически**, поэтому мы **идем на следующую итерацию, чтобы попробовать вставить его вновь**.

Если же **элемента в списке не было найдено**, то будем его честно **добавлять**. В цикле **for с 0 уровня до наибольшего уровня данного узла** будем **блокировать предшествующие узлы на данном уровне** и **проверять**, не являются ли **предшествующие узлы логически удаленными**, также с **текущими узлами,** а также **проверяем**, что **предшествующие узлы все еще указывают на текущие,** и **эта ссылка верна**, то есть, что **никакой другой поток нас не опередил.**

Если какая-то из проверок **не прошла**, то **начинаем все с начала, освобождая блокировки только уровнях,** на которых они и **были захвачены,** что обеспечивается **highestLocked**.

Если же **все прошло успешно**, то создаем **новый узел для вставляемого элемента.**

Далее на **каждом уровне определяем для нового узла следующий элемент,** а затем **у предшествующих узлов определяем ссылки на новый узел.**

В конце **помечаем узел как полностью связный** и **снимаем захваченные блокировки**. Возвращаем **true**.

Вопрос: почему нет блокировки на succ, а есть только на pred?

Remove():

Аналогично **создаем массивы и узел-жертву**. Также **ищем с помощью Find() нужный узел**.

Если **нашли**, то сохраним этот узел в переменную **victim**.

Если же элемент уже отмечен нами как **логически удаленный** или **мы его обнаружили,** он **полностью связный** и его **верхний уровень равен найденному уровню с помощью Find()** и он **не является логически удаленным**, то **проверяем** дальше, что **мы пытаемся удалить узел в самый первый раз**, то есть он еще **не помечен никем как логически удаленный**, то мы **блокируем этот узел** и **еще раз проверяем**, **не отметил ли другой поток его**.

Если другой поток **успел логически удалить этот узел**, то **разблокируем его и выходим**.

В противном случае, **помечаем узел как логически удаленный**.

Далее начинаем **физическое удаление**: как в **Add() поднимаемся снизу вверх и на каждом уровне пытаемся заблокировать предшествующий элемент и проверяем**, что **всё валидно**.

Если **valid не равна true**, то **удаление надо начинать сначала**, так как **состояние списка уже неактуальное**. При этом **victim снова помечать как логически удалённый уже не надо,** и поэтому **работа метода будет ускорена**.

Если же **список актуальный**, то **с максимального уровня victim и вниз мы обновляем ссылки предшествующих узлов на соответствующие узлы следующего за удаляемым узла**. Когда закончили это делать, **разблокируем victim** и **разблокируем все preds,** возвращаем **true**.

А если элемент мы **не обнаружили или с ним ещё что-то случилось из проверок**, то **false**.

30*. Неблокирующая синхронизация для списка с пропусками (skip list).

В отсутствие мьютексов, мы больше не можем блокировать всю стенку ссылок от предыдущих узлов на элемент из-за чего и **не можем всегда поддерживать скиплист в консистентном состоянии,** так как множество

потоков может изменять ссылки в этом случае одновременно. Однако у нас в распоряжении есть **атомарные операции**.

Можем использовать **верхние уровни** исключительно как **сокращённые пути к элементам списка на самом нижнем уровне**. Таким образом, мы немного отходим от привычной структуры скиплиста. Однако по возможности **пытаемся сохранять сокращённые пути**.

LockFreeSkipList (неблокируемый скиплист):

Как и в **LazySkipList,** у нас есть **опорные узлы head и tail с минимальным и максимальным значением, то есть снова будет отсортированность по возрастанию.**

В классе **Node** теперь мы используем структуру **AtomicMarkableReference**, в которой хранятся и **указатели на следующие элементы и значение, является ли данный узел логически удаленным**.

Также есть **два конструктора - для опорных и обычных узлов**. Для **двух опорных узлов начала и конца скиплиста** используем какие-то **значения по умолчанию**. Также для них устанавливаем **максимальное число уровней**. Для **обычны узлов** максимальный уровень устанавливается **в соответствии с передаваемым**.

В остальном аналогично **LazySkipList**.

Find():

Находим **хэш от искомого элемента**.

Далее будем **спускаться по уровням сверху вниз и искать нужный элемент**. В данном методе **в отличие от LazySkipList элементы не только ищутся, но и физически удаляются**. При этом другие потоки также **могут работать со списком.**

Если **элемент логически удалён**, то если мы физически **не смогли удалить элемент с помощью операции CompareAndSet()**, то есть **кто-то нас опередил**, значит **состояние списка неактуальное**, поэтому **начинаем все с самого начала**.

Если нам **никто не мешает**, то мы **физически удаляем нужные элементы на текущем уровне и идём дальше**. Таким образом, **на нижележащих уровнях могут оказаться логически удаленные элементы,** формально содержащиеся в списке. В итоге **постепенно удаляем элементы на всех уровнях**.

Далее, когда **находим нужный диапазон**, **проверяем тот ли это элемент или нет**. Здесь **не важно, на каком уровне мы находим элемент**, потому что **свойство скиплиста здесь не выполняется**, ведь в какой-то момент времени может быть что **на более высоком уровне элемент уже физически удален, а на более низком - нет**. Поэтому нас интересует, нашли мы элемент или нет.

Возвращаем результат проверки и заполненные стенки поиска.

Contains():

Проходим от максимального уровня до нулевого. Пропускаем **логически удаленные элементы**, так как они нас не интересуют. Формально, мы можем их и физически удалять из списка, как в **Find()**, но мы этого не делаем, так как **Contains() должен быть легким.**

Если выясняется, что **текущий элемент на самом нижнем уровне совпадает с искомым по хэшу**, значит **он содержится в списке**.

Add():

Также **определяем максимальный уровень для нового узла и стенки для поиска**.

Если через метод **Find() мы нашли**, что этот **элемент уже есть в списке**, то **false**.

В противном случае, создаем **новый узел с ключом и уровнем**.

Начиная с нижнего уровня и до самого верха, сперва **заполняем в новом узле ссылки на следующие элементы**. Может быть, что **за это время какие-то элементы удалили,** но к этому мы вернёмся**.**

Далее мы **пытаемся вставить новый элемент на самый нижний уровень с помощью операции CompareAndSet()**.

Если это **сделать не удалось**, то либо же наш **предыдущий элемент не указывает на последующий элемент**, который мы **сохранили выше**, то мы **начинаем сначала**, либо же **предшествующий узел удалили**, и мы тоже **начинаем сначала**.

Если же мы **смогли добавить элемент на самом нижнем уровне**, то **пробуем обновить его более высокие уровни**. Идём от **первого уровня до наибольшего на этом узле**.

Если на каком-то уровне **не получилось с помощью CompareAndSet() обновить ссылку**, значит **состояние этого уровня уже неактуальное**,

поэтому **вызываем Find() для актуализации списка** и **обновляем ссылки на этом уровне снова,** и так до тех пор, **пока не смогли обновить ссылки на уровне**.

Если на уровне **получилось с помощью CompareAndSet() обновить ссылку**, то **переходим на следующий уровень, выходя из цикла while(true)**.

В конце возвращаем **true.**

Remove():

Также в цикле с помощью **Find() ищем искомый узел**, если **не находим**, то **false**, так как **такого узла в списке нет**.

В противном случае определяем, что это узел **находится на самом нижнем уровне**. И удаляем **все ссылки на него с самого верхнего уровня до первого уровня**. Для этого мы смотрим **на последующий узел за удаляемым**, и **если наш удаляемый узел уже логически удален другим потоком**, то **не делаем ничего. Иначе помечаем его через CompareAndSet() как логически удаленный**.

После того, как мы **прошлись по всем уровням, кроме последнего**, мы пробуем **логически удалить на нулевом уровне удаляемый узел**.

Если нам **удалось логически удалить узел**, то вызываем **Find() для сборки мусора** и выходим с **true**.

Если же нам **не удалось логически удалить узел**, то снова проверяем, не **удален ли он, но при этом другим потоком**. Если **это так**, то есть **нас опередили, то возвращаем false, так как удалили не мы**. **В противном случае пробуем это сделать снова**.

Таким образом, нам не обязательно поддерживать консистентность на всех уровнях, нам достаточно делать это только **для самого низкого.**

31. Базовые понятия GPGPU и технологии NVIDIA CUDA.

GPGPU - техника использования графического процессора видеокарты, предназначенного для компьютерной графики, в целях производства математических вычислений, которые обычно проводит центральный процессор. Это стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых

конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры видеокарт для выполнения неграфических вычислений.

OpenCL - открытый стандарт, поддерживающий **GPGPU.** Стандарт является платформонезависимым.

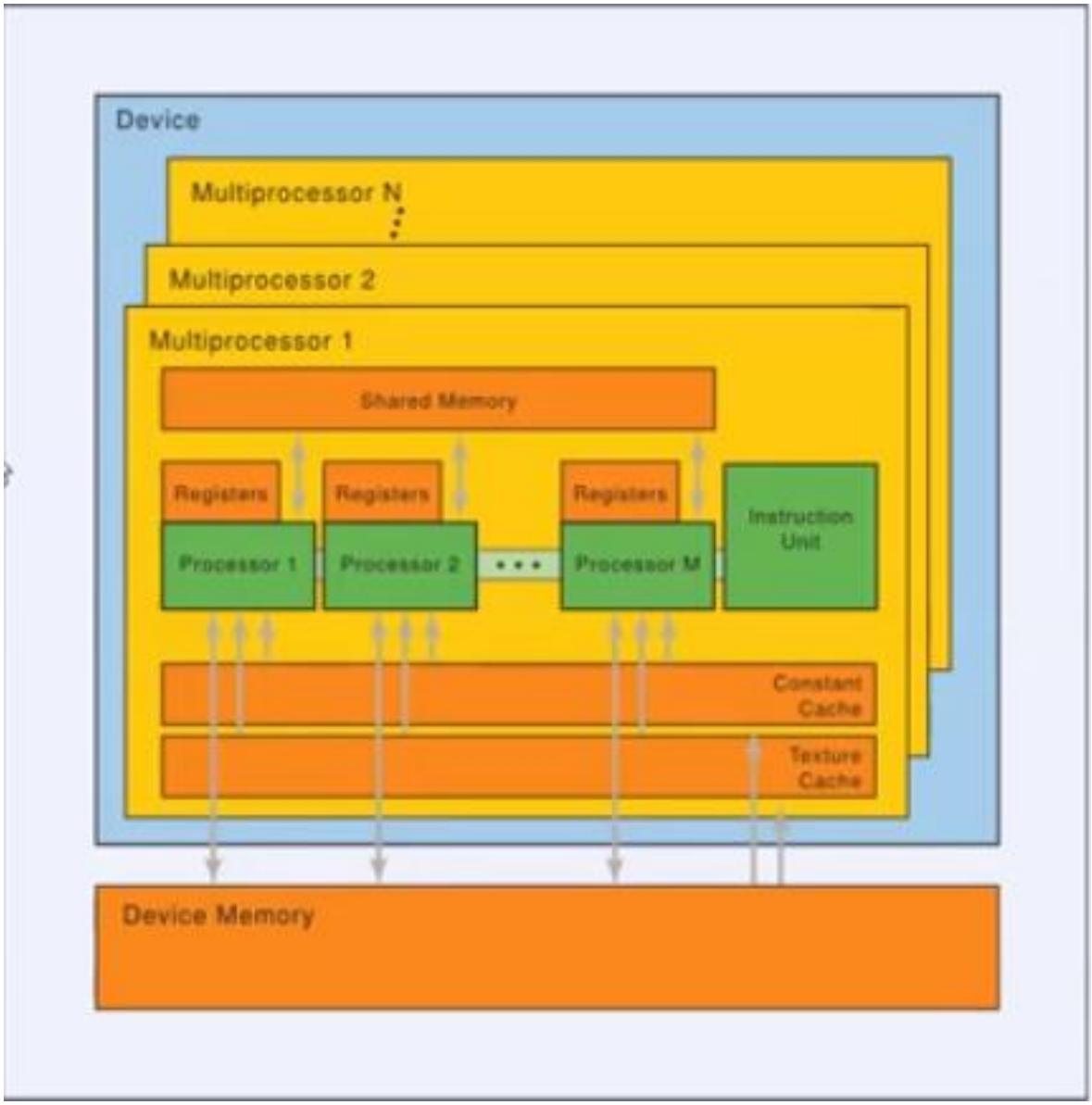
CUDA - программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы **Nvidia**.

CUDA SDK позволяет программистам реализовывать на специальных упрощённых диалектах языков программирования **Си, C++ и Фортран** алгоритмы, выполнимые на графических и тензорных процессорах **Nvidia**.

С точки зрения **программной модели CUDA**:

Видеокарта обозначается как **device,** так как формально необязательно, чтобы программируемое устройство вообще имело видеовыход.

Программируемое устройство всегда является **пассивным**, то есть **контроль его работы ложится на CPU**. Видеокарту можно рассматривать как **периферийное устройство**, которое может **работать одновременно с CPU**.



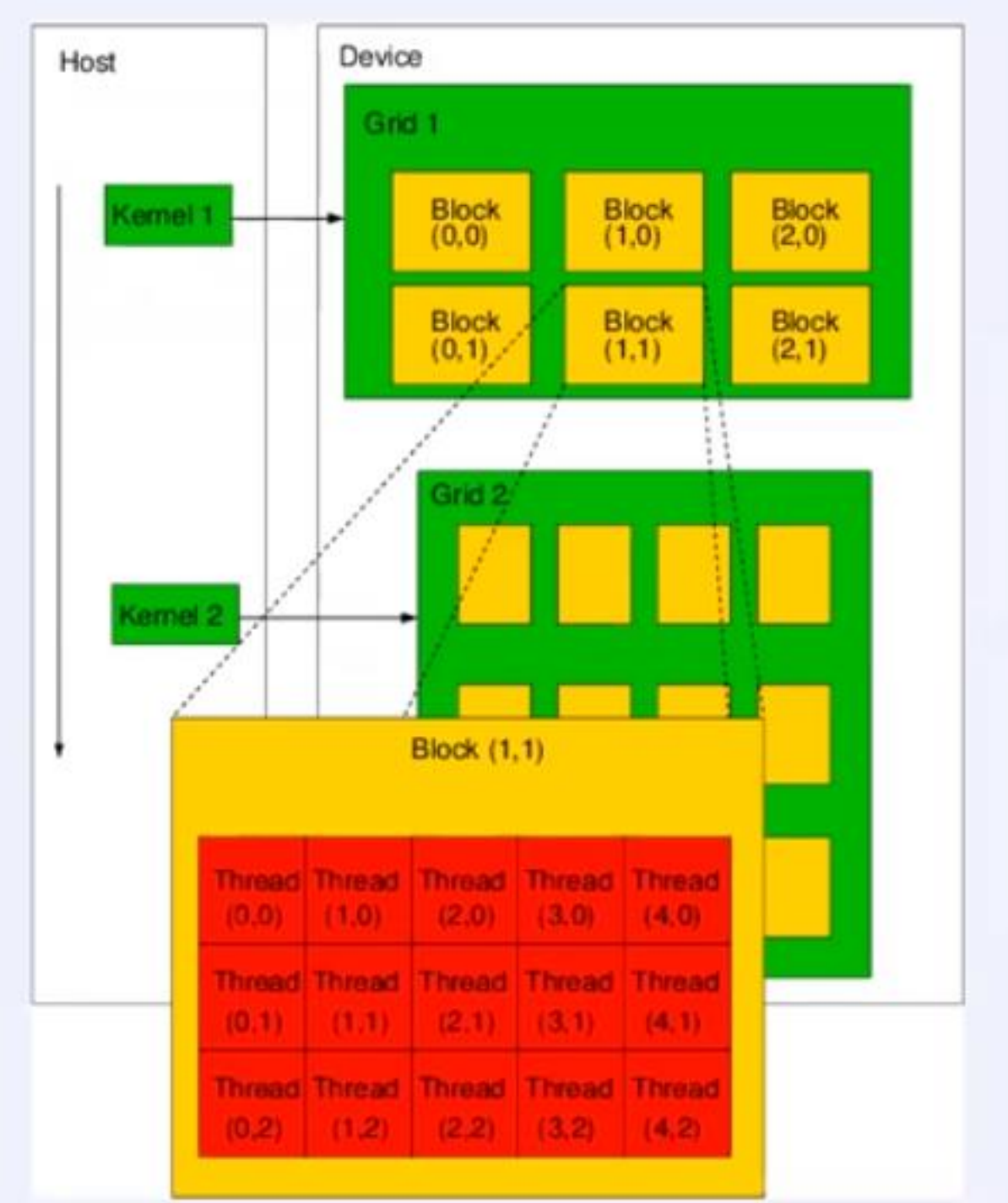
У **device** есть **память,** которая обычно указывается в спецификации**.** Память **device физически отделена от оперативной памяти компьютера**, в том числе и на уровне **адресного пространства.** Например, разыменовывать указатель из оперативной памяти в памяти девайса будет **некорректно**.

У **device** есть от **1 до бесконечности мультипроцессоров**.

В каждом мультипроцессоре есть свой **Insruction Unit - блок управления программы**, который контролирует несколько **АЛУ**. Также у каждого мультипроцессора есть свой объем **регистровой памяти,** которая во время работы и выделяется под нужды каждого **АЛУ**. То есть регистровая память может быть использована в качестве **разделяемой** памяти. Также дополнительно может быть несколько **кэшей** (констант, текстур и других), которые позволяют в некоторые моменты не обращаться к памяти **device.**

Весь объем работы **распределяется** между **мультипроцессорами**, это распределение делается на уровне самого **device**. Каждый **мультипроцессор** выполняет свой объем работы.

Мультипроцессор - далеко не самый быстрый вычислитель, выигрыш в производительности как раз происходит из-за того мультипроцессоров **много**, так и в каждом может быть **16 или 32 АЛУ,** на которых выполняется одна и та же программа в рамках **kernel**.



Программирование на **CUDA** состоит в том, что с **CPU**, обозначается как

host, приходит информация о том, что нужно выполнить **задачу**. Задачи именуется как **kernel**.

Kernel содержит **описание задачи для отдельного потока** и **спецификацию разбиения задачи на блоки** и **каждого блока на отдельные потоки**. То есть каждый **kernel** сопровождается своей **сеткой grid,** которая может быть любой размерности**.** В нашем случае первая задача содержит **сетку 2x3** из **блоков**, при этом каждый **блок делится** на **свою сетку,** которая тоже может быть любой размерности и где **в каждой ячейке содержатся потоки**. В нашем случае, это **сетка 3x5**.

Во время выполнения программы необходимо понять, **за какой кусок задачи отвечает каждый конкретный поток**.

У программы на **CUDA** есть возможность определить, **какому блоку принадлежит данный поток и какой у него идентификатор в этом блоке, а также размерность всей сетки,** в виде глобальных констант, доступных на уровне **CUDA**.

Если умножить **индекс блока**, в котором находится поток, **на размер блока**, а затем **прибавить ему индекс потока внутри блока**, а можно назначить **уникальный идентификатор потока в рамках сетки решаемой задачи**.

Используются расширения языков **C/C++** для программирования на **CUDA,** называемые **CUDA C/C++.** Расширение у таких файлов **.cu.**

Так как никаких исключений в **C** нет, то оповещение об ошибке приходит в виде **кода ошибки**, по которому затем можно понять, что пошло не так.

Host может одновременно работать с **несколькими device**, переключая контексты с одного **device** на другой.

Так как работая с девайсом, мы работаем с **другим адресным пространством**, то привычные операции, как в **C**, **выделения памяти device** здесь не подойдут, поэтому надо использовать операции с **префиксом cuda**. Например, **cudaMalloc()**. Память выделяется из глобальной памяти видеокарты.

С помощью **cudaMemcpy()** можно выполнить пересылку данных с компьютера на **device, с device** на **компьютер**, либо с **device** на **device**.

С помощью функции **_global_ addKernel<<<...>>>()** в угловых скобках можно конфигурировать размер **kernel'a**. Первое параметр - **количество блоков или размер сетки grid**. Второе число - **размер отдельного блока**

в количестве потоков. Далее происходит загрузка кода этого **kernel** на видеокарту. Данная операция **ассинхронная**.

Модификатор _global_ - указание компилятору, что **данная функция является функцией, загружаемой в видеокарту**, следовательно, ее нужно скомпилировать **не в код для CPU**, а **в код для видеокарты**. Фактически это указание, что данная функция **вызывается из hosta, но исполняется на девайсе**.

Таким образом, часть кода компилируется **в код для видеокарты, а часть в код для CPU**.

Тело такой функции представляет из себя программу, которая будет запускаться **на одном единственном АЛУ в рамках мультипроцессора**.

В зависимости от модели видеокарты, **на одном мультипроцессоре может быть либо 16, либо 32 АЛУ**. Поэтому потоки **исполняются** не по отдельности, а **пучками по 16 или 32 потока**. Такие пучки называются **варпами**. **CUDA** сама определяет, как нарезать блок в зависимости от доступности памяти. Если **не все потоки в варпе задействованы, то они автоматически отключаются**.

Компилятор **CUDA** является жестко **оптимизирующим**. При отладке можно заметить, что локальные переменные, **которые больше не используются**, **пропадают после последнего выражения**, **где они используются**.

Мультипроцессор оперирует работой **не отдельного потока, а варпа**, поэтому **исполнение всего блока подразделяется на исполнение отдельного варпа**.

Диспетчер видеокарты каждому мультипроцессору назначает тот или иной блок **от начала его выполнения и до конца**. **Все варпы, входящие в один блок, будут выполнены на одном и том же физическом мультипроцессоре**. Естественно, разные блоки могут выполняться на разных мультипроцессорах.

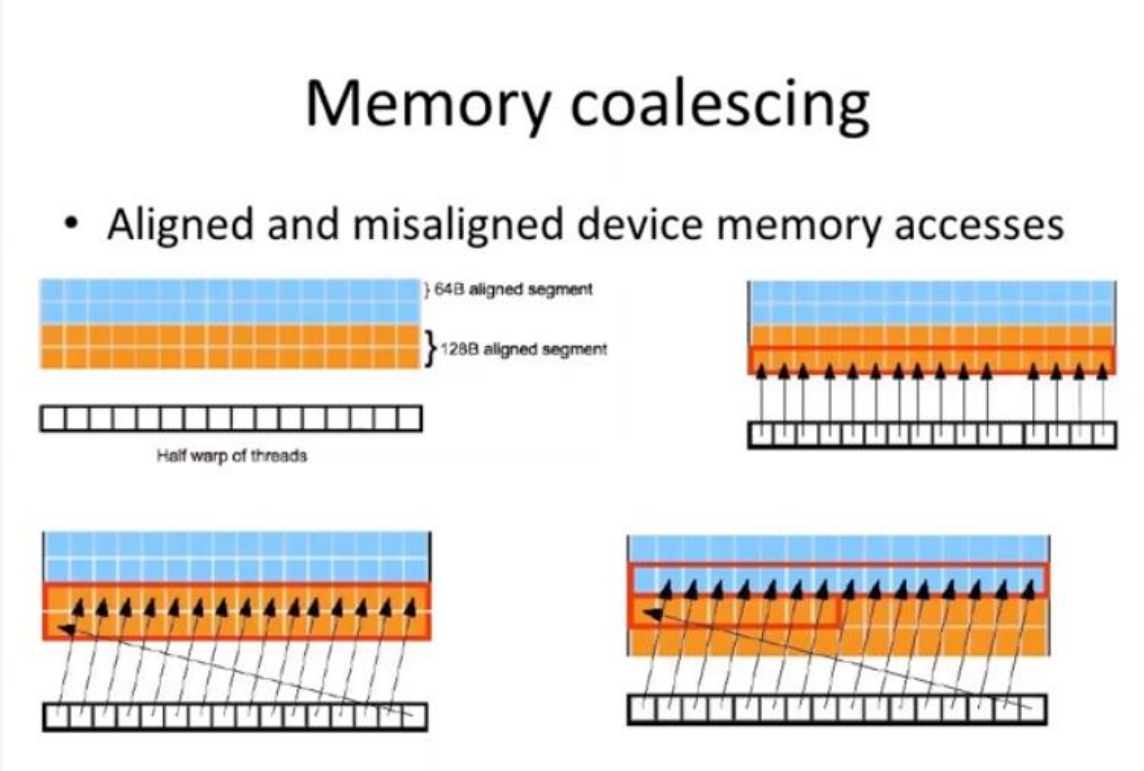
Синхронизация между блоками **существует, но лучше её не допускать**. Внутри блоков **никакой синхронизации, кроме барьерной, не существует**.

Производительность повышается также потому, что **пока один варп простаивает**, **другой варп может производить вычисления**. Массивнейший параллелизм здесь также достигается **грамотным оперированием варпами**, то есть их переключением в **разные состояния**.

С помощью **cudaGetLastError() можно проверить наличие ошибок**.

С помощью **cudaDeviceSynchronize() можно дождаться окончания работы всего kernel и получить код возврата.**

Отладка в **CUDA** позволяет увидеть состояние всех мультипроцессоров и варпов.



Диспетчер памяти склонен выделять память блоками, **выравненными на границу из 64 или 128 байт**. Связано это с тем, что **шина данных**, которая **связывает мультипроцессоры с памятью девайса** имеет размер **64 или 128 байт**.

Если все потоки в варпе обращаются к элементам **одного и того же блока памяти**, то **данные будут получаться из памяти одним запросом**.

Если потоки будут обращаться к блоку памяти **не по порядку**, то это не вызовет проблем, так как **мультипроцессор поймет**, **как раскидать данные по АЛУ**.

Если же потоки одного варпа будут обращаться **к разным блокам памяти**, то здесь уже будет сделано **несколько запросов к оперативной памяти**, что приведет к снижению производительности.

В **условных операторах** сначала происходит **выполнение всех потоков варпа, соответствующих истине**, при этом **остальные потоки спят**. И только затем выполняются потоки, соответствующие **лжи**.

В циклах мультипроцессор будет работать со скоростью того потока, который **выполняет большую часть итераций**.

Система девайса по **таксономии Флинна** является **MIMD**. В самом плохом случае - **SIMD**.