Министерство образования и науки Российской Федерации

Санкт-Петербургский государственный университет

информационных технологий, механики и оптики

Факультет Информационных технологий и программирования
Направление (специальность) Прикладная математика и информатика
Квалификация (степень) Бакалавр прикладной математики и информатики
Специализация ----------
Кафедра Компьютерных технологий **Группа** 4539

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К БАКАЛАВРСКОЙ РАБОТЕ

Разработка и анализ параллельных поисковых структур данных, нечувствительных к размеру кеша

Автор квалификационной работы Акишев И. Р.

Руководитель Елизаров Р. А.

Санкт-Петербург, 2008 г.

ВВЕДЕНИЕ

В настоящее время существует великое множество алгоритмов реализующих различные структуры данных. В зависимости от поставленных задач эти структуры данных могут реализовывать поддержку различных операций, причем для выполнения тех или иных операций они в разной степени потребляют различные ресурсы (такие, как время работы, память и т. д.).

Для того чтобы оценивать эффективность алгоритмов, классифицировать их, а так же изучать их свойства вводятся различные математические модели вычислений. Почти все классические алгоритмы традиционно исследуются в рамках модели, получившей название машины с произвольным доступом, или *RAM*-машины (*RAM* – random access memory) [1], которая довольно проста и в достаточном смысле хорошо соотносится с реальной архитектурой современных компьютеров. В такой модели вычислений вся память представляется в виде бесконечного последовательного массива, к любой ячейке которого программа может обратиться за единицу времени. При использовании этой модели эффективность алгоритмов традиционно измеряется двумя основными критериями: *асимптотическим временем работы* и *асимптотическим объемом используемой памяти*.

Однако, существует множество различных алгоритмов и структур данных, которые, хотя и имеют хорошие теоретические оценки времени работы и используемой памяти, но на практике оказываются слишком ресурсоемкими, громоздкими и медлительными, что не позволяет их использовать в реальных вычислительных системах (яркий пример – Фибоначчиевые кучи). Таким образом, двух указанных критериев оценки эффективности алгоритмов часто бывает недостаточно.

Другим недостатком модели *RAM* и классического подхода разработке и оценке сложности алгоритмов является тот факт, что они не

учитывают важнейшую современную тенденцию сферы компьютерных технологий – переход от *последовательных* систем к *параллельным*.

Цель работы – изучение новых, альтернативных моделей исполнения программ и разработка на их основе эффективных структур данных и алгоритмов. Эти алгоритмы, с одной стороны, должны иметь хорошие теоретические оценки эффективности работы, а с другой – обладать некоторыми особыми свойствами и быть достаточно простыми, для того чтобы быть эффективно реализованными на практике с учетом архитектуры реальных современных систем.

В главе 1 рассматриваются различные теоретические модели вычислений, в частности различные модели доступа к памяти, а так же модели параллельных вычислений, и связанные с ними примитивы. Также в этой главе дается обзор различных существующих алгоритмов.

В главе 2 приводится реализация параллельного алгоритма очереди на основе развернутого связанного списка, доказывается его корректность, а так же исследуются и доказываются некоторые его свойства, в частности отсутствие блокировок, а так же теоретически оценивается эффективность работы с памятью в данном алгоритме.

В главе 3 приводится практический анализ производительности описанного алгоритма в сравнении с предыдущими, произведенный на различных вычислительных системах, а также исследуется зависимость различных параметров на его эффективность.

Разработанный в рамках данной работы алгоритм имеет широкую сферу для применения на практике, так как использованные в нем идеи и свойства, которыми он обладает, позволяют ему показывать высокую производительность на реальных существующих системах.

ГЛАВА 1. МОДЕЛИ ВЫЧИСЛЕНИЙ

В этой главе кратко описываются основные существующие модели доступа к памяти, а также делается обзор параллельных моделей вычислений и теоретических примитивов, связанных с ними.

1.1. Модели доступа к памяти

В этом разделе будут рассмотрены различные модели доступа к памяти, которые соответствуют различным способам оценки эффективности работы алгоритмов.

1.1.1. Модель произвольного доступа к памяти

Модель *RAM*-машины [1] (random access memory machine, равнодоступная адресная машина, машина с произвольным доступом) – классическая модель вычислений, относящаяся к классу регистровых машин. Она является достаточно простой, естественной, и в то же время хорошо отражающей базовую архитектуру фон Неймана. *RAM*-машина родилась как модификация регистровой машины, в которую были добавлены операции косвенной адресации.

RAM-машина является одной из многих эквивалентных машине Тьюринга модели вычислений. Существуют разные теоретические модификации *RAM*-машины, однако в этой работе не будут рассматриваться формальные подробности ее устройства. Нас будет в основном интересовать модель доступа к памяти, которая используется в *RAM*-машине.

Вся память в *RAM*-машине представляет собой бесконечный массив, все ячейки которого пронумерованы натуральными числами. Каждая из ячеек может хранить целые числа из некоторого диапазона. Программа имеет доступ к ячейкам памяти посредством *прямой* или *косвенной адресации*. Она может записывать в свои регистры значение некоторой ячейки памяти и наоборот, при этом номер ячейки, к которой она обращается, может

задаваться как непосредственно (прямая адресация), так и браться из другой ячейки или регистра (косвенная адресация). Это так называемая *модель произвольного доступа к памяти* (random access memory).

Операция доступа к любой ячейке памяти в такой модели занимает *O(1)* времени, а основным критерием оценки эффективности алгоритма в ней является *асимптотическое время работы*, которое выражается, как функция от размера входных данных для задачи *t(N)*. Вторым важным критерием является *асимптотический объем используемой памяти*, то есть количество ячеек, к которым алгоритм будет обращаться в процессе своей работы *s(N)*.

Схематическое изображение модели произвольного доступа к памяти изображено на рис. 1.1.

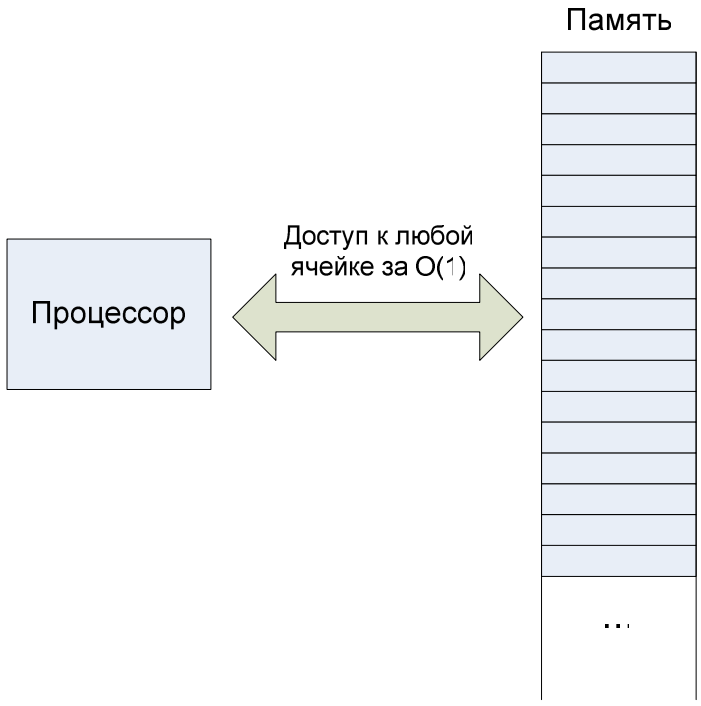

Рис. 1.1. Схема модели *RAM*

Подавляющее большинство классических алгоритмов составляется в терминах модели произвольного доступа к памяти, а их эффективность характеризуется временем работы и объемом используемой памяти. Не смотря на то, что такой анализ алгоритмов является общепринятым и классическим, он не лишен некоторых недостатков.

1.1.2. Кеширование

Основным недостатком модели произвольного доступа к памяти является то, что она не учитывает наличия в реальных системах такого промежуточного звена в иерархии памяти, как *кеш*.

Реальные вычислительные системы устроены так, что скорость передачи данных из памяти в регистры процессора заметно зависит от размера этой памяти. Чем больше доступное программе адресное пространство, или, иными словами, чем больше число различных возможных доступных ячеек памяти, к которым можно обратиться, тем сложнее становятся системы связывающие память с процессором, и тем больше тактов требуется процессору на то, чтобы прочитать или записать значение в некоторую ячейку памяти. Поэтому в современных компьютерах кроме «основной» памяти существует еще несколько уровней памяти, называемых *кешами*. Чем ближе кеш находится к центральному процессору в иерархии памяти, тем меньше его объем, и тем выше скорость доступа к нему.

Данные, которые требуются по мере работы программы, копируются из основной памяти в кеши разных уровней по очереди, и лишь затем попадают в регистры процессора.

Сам термин «кеш» («*cache*») произошел от английского слова «*cash*» – «наличные» (в противопоставление основной памяти, которая как бы соответствовала счету в банке), и был впервые использован еще в 1967 году при публикации статьи об эффективной организации памяти в журнале *IBM Systems*. В современных компьютерах обычно есть два-три уровня кеш-памяти.

Данные между соседними слоями иерархии памяти обычно переносятся не по одной ячейке, а блоками некоторого фиксированного размера. Если запрашиваемые процессором данные уже имеются в кеш-памяти данного уровня, то они не копируются повторно из памяти предыдущего уровня, а сразу же используются. Такая ситуация называется *попаданием кеша* (*cache hit*). Если же блока, в котором находится

запрашиваемая ячейка памяти, нет в кеше, то его загрузка занимает продолжительное время (по сравнению с выполнением такта процессора или с обращением к ячейке, присутствующей в кеш-памяти). Такая ситуация называется *промахом кеша* (*cache miss*), так как требует *переноса блока памяти* (*memory transfer*). Если в кеш необходимо загрузить блок памяти, но свободного места в нем уже нет, то из него стирается один из ранее закешированных блоков, и новый записывается взамен него. Существуют различные алгоритмы выбора блока для удаления (*алгоритмы вытеснения*), подробный обзор которых можно найти в работах [2, 3].

Подобная кешу структура используется не только как промежуточное звено между процессором и оперативной памятью, но также на уровне операционных систем при работе с виртуальной памятью (которая аналогичным образом постранично загружается в оперативную память по мере надобности), при доступе к дисковой файловой системе, или к сетевым ресурсам.

1.1.3. Модель идеального кеша (модель ввода/вывода)

Рассмотрим более формально модель памяти с одним дополнительным уровнем кеша. Она состоит из бесконечного индексированного массива основной, медленной памяти (как и в случае классической модели произвольного доступа), и быстрого кеша. Вся память разбита на блоки ячеек памяти фиксированного размера, по L ячеек памяти каждый. Размер кеша ограничен Z ячейками памяти.

Процессор может работать только с памятью, находящейся в кеше. Память загружается в кеш цельными блоками. Предполагается, что кеш *полностью ассоциативен*, то есть любой блок основной памяти может быть записан в произвольное место кеша (что на практике иногда бывает неверно). Таким образом, в каждый момент времени в кеше может находиться Z/L произвольных блоков из основной памяти.

Предполагается, что кеш достаточно *длинный*, и количество блоков, которое в нем может поместиться соизмеримо с размером самого блока – $Z = \Omega(L^2)$.

При необходимости очистки места для нового блока, загружаемого в кеш, при выборе очередного блока, который будет удален, используется так называемая *оптимальная оффлайн стратегия*. Из кеша удаляется тот блок, который в будущем понадобится в следующий раз позже всех. Такой подход подразумевает, что управлением памятью и кешом занимается сам алгоритм, так как никакой отдельный менеджер памяти не может заранее предсказать, к каким блокам памяти и когда будут происходить обращения в будущем.

Подобный алгоритм вытеснения очень удобен в теории, однако на практике практически не существует систем, в которых программа может на столь низком уровне управлять кешом. На практике используется так называемая стратегия вытеснения *LRU* (*Least Recently Used* – позже всех использованный), либо ее различные эвристические модификации. Она заключается в том, чтобы удалять тот блок памяти, последнее обращение к которому происходило позже всех. В работах [3, 4] даются различные оценки, связывающие число переносов блоков памяти при использовании оптимальной стратегии и при использовании стратегии *LRU*. Суть этих оценок сводится к тому, что можно моделировать оптимальную оффлайн стратегию при помощи *LRU* в аналогичной системе, но с измененным в константу раз параметром *Z*. При этом число переносов блоков памяти, которое потребуется для этого, увеличится лишь в константу раз. Таким образом, при асимптотических оценках эти стратегии оказываются эквивалентными.

В качестве критерия сложности алгоритма в терминах данной модели выступает не только количество производимых операций (которое в классической модели *RAM*-соответствовало времени работы) *t(N)*, но и количество произведенных при этом переносов блоков памяти *q(N)*, или, что то же самое – число промахов кеша.

Описанная выше модель доступа называется *моделью идеального кеша* (*ideal-cache model*). Ее схематическая структура изображена на рис. 1.2.

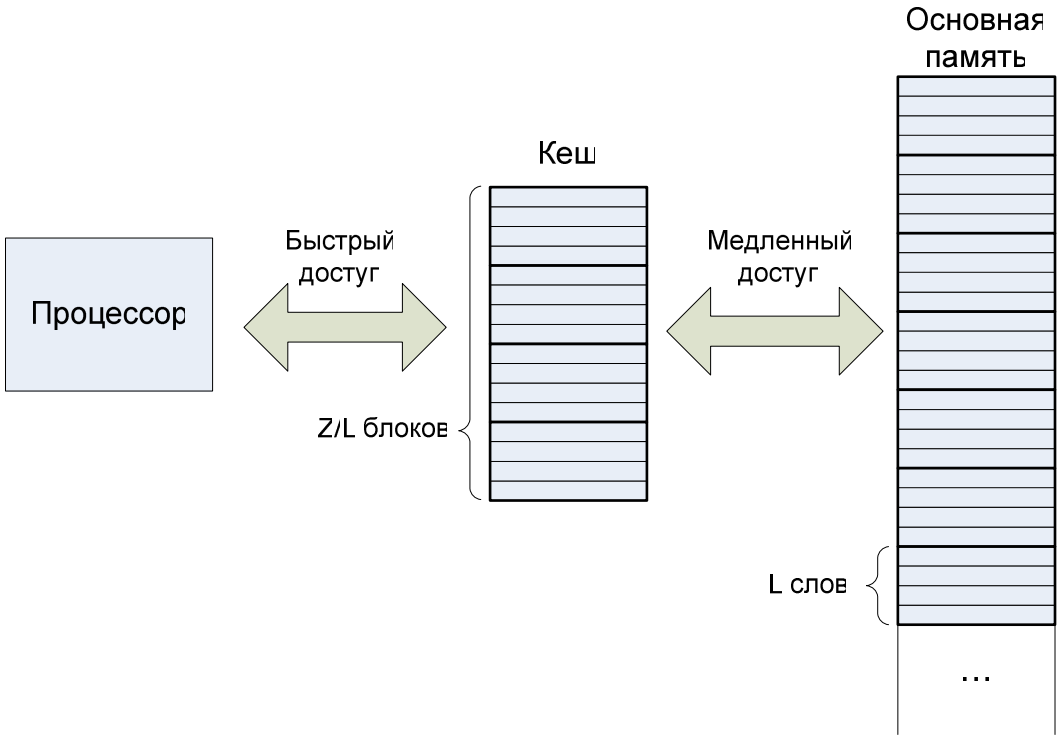

Рис. 1.2. Схема модели идеального кеша

Другое ее название – *модель ввода-вывода* (*I/O model*), так как в первых работах, которые изучали эту модель, она рассматривалась не с точки зрения уровней оперативной памяти и кеша, а с позиций уровней файловой системы и оперативной памяти. Эта модель была, в частности, формально изложена в работе [5], где приводились также алгоритмы для различных задач, таких как быстрое преобразование Фурье, транспонирование матриц, различные алгоритмы сортировки, а также их оценки в терминах данной модели.

Попытки проектировать алгоритмы и измерять производительность в терминах переносов блоков памяти проводились и раньше. Так, наиболее известна классическая работа Байера и МакКрейта [6], в которой была описана поисковая структура данных, известная, как *Б*-дерево, сконструированная для того, чтобы служить эффективным индексом больших объемов данных, хранимых во внешней памяти, доступ к которым

осуществляется блоками. Позднее эта структура стала классической, и было изобретено множество ее модификаций.

1.1.4. *Cache-oblivious* модель

Описанная в предыдущем пункте модель доступа к памяти предполагает наличие лишь одного уровня кеша. Кроме того, подразумевается, что составителю алгоритма для данной модели известны ее параметры Z и L, так что он может использовать их в качестве устанавливаемых параметров для оптимизации алгоритма под ту или иную конкретную систему. Можно попытаться расширить эту модель для случая, когда в системе присутствуют несколько уровней кеш-памяти с разными параметрами, однако полученные модели будут громоздкими и сложными для практического применения при доказательстве свойств тех или иных алгоритмов. Необходимость знания параметров памяти конкретных систем приводит к тому, что алгоритмы, эффективные в терминах модели идеального кеша, приходится программировать под конкретную систему и всякий раз перенастраивать при ее модификации или портировании на другие системы.

В 1999 году четырьмя учеными из MIT была предложена новая модель для оценки сложности алгоритмов, которая получила название *cache-oblivious модели* или *модели, нечувствительной к параметрам кеша* [4].

Идея данной модели заключается в том, что алгоритм строится в терминах одноуровневой модели идеального кеша, но конкретные параметры этой модели заранее неизвестны. Производительность алгоритма с точки зрения работы с памятью, так же как и в предыдущей модели, оценивается, как функция, выражающая число переносов блоков из памяти в кеш. Однако теперь эта функция зависит уже не только от объема данных для задачи N, но и от параметров модели идеального кеша: $q(N, Z, L)$.

При этом сам алгоритм не знает значений параметров Z и L во время своей работы, и формулируется независимо от них.

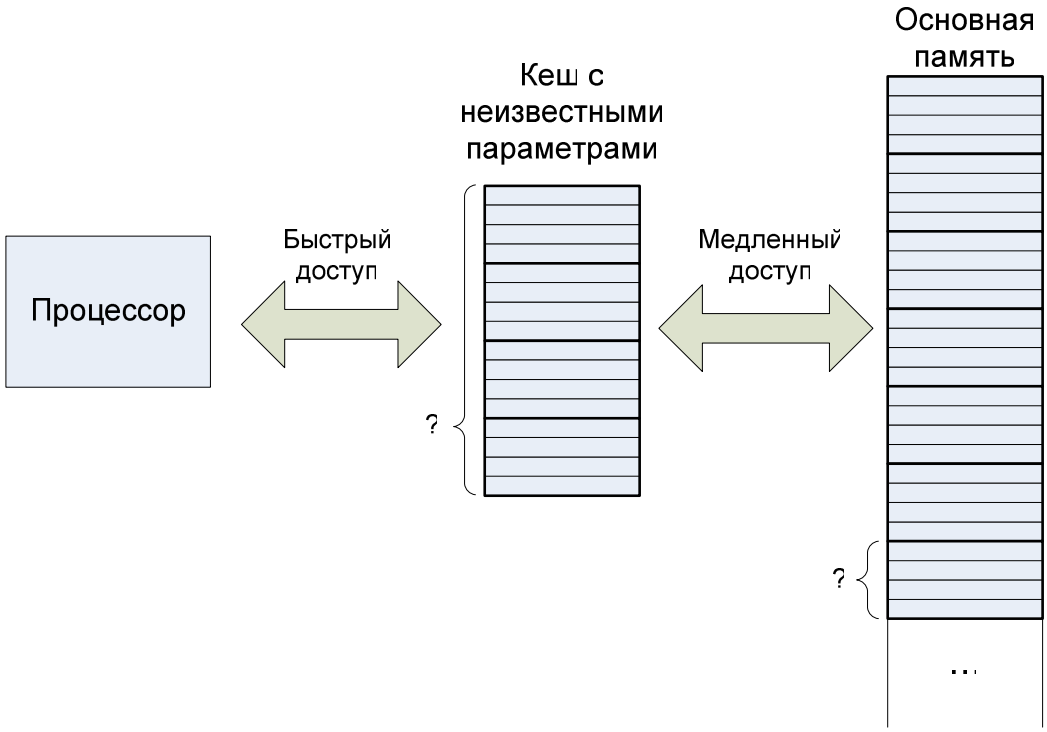

Рис. 1.3. Схематическое изображение *Cache-Oblivious* модели памяти

Алгоритм (или структура данных) считается *нечувствительным к параметрам кеша* (*cache-oblivious*), если для любых параметров кеша он работает за асимптотически оптимальное число переносов блоков памяти. В противном случае, если алгоритм для оптимальной производительности требует настройки дополнительных параметров, то он считается *чувствительным к параметрам кеша* (*cache aware*).

В частности, *Б*-деревья не обладают свойствами *Cache-Oblivious*, так как, хотя они и работают эффективно в модели идеального одноуровневого кеша, но это требует использования узлов в *Б*-дереве размера того же порядка, что и размер одного блока памяти L.

Характерным и важным достоинством *cache-oblivious* модели является тот факт, что если алгоритм является *cache-oblivious*, а следовательно, асимптотически эффективно работающем на машинах с любым одноуровневым кешом, то он также будет работать асимптотически оптимально и на машинах с произвольной многоуровневой иерархией кеш-памяти [4].

Cache-oblivious модель памяти схематично изображена на рис. 1.3.

В настоящее время теория *cache-oblivious* алгоритмов бурно развивается, и уже придумано множество *Cache-Oblivious* алгоритмов. Помимо уже упомянутых алгоритмов, рассмотренных в работе [4], особый интерес представляют различные *Cache-Oblivious* поисковые структуры данных.

Наиболее интересна модификация сбалансированного дерева поиска, которая получила название *COB-дерево* (*Cache-Oblivious B*-tree) [7]. Основная идея, использованная в этой структуре данных, заключается в использовании так называемой *укладки ван Эмде Боаса*, которая позволяет разместить полное *k*-ичное дерево в линейной памяти «*Cache-Oblivious* образом» (рис. 1.4).

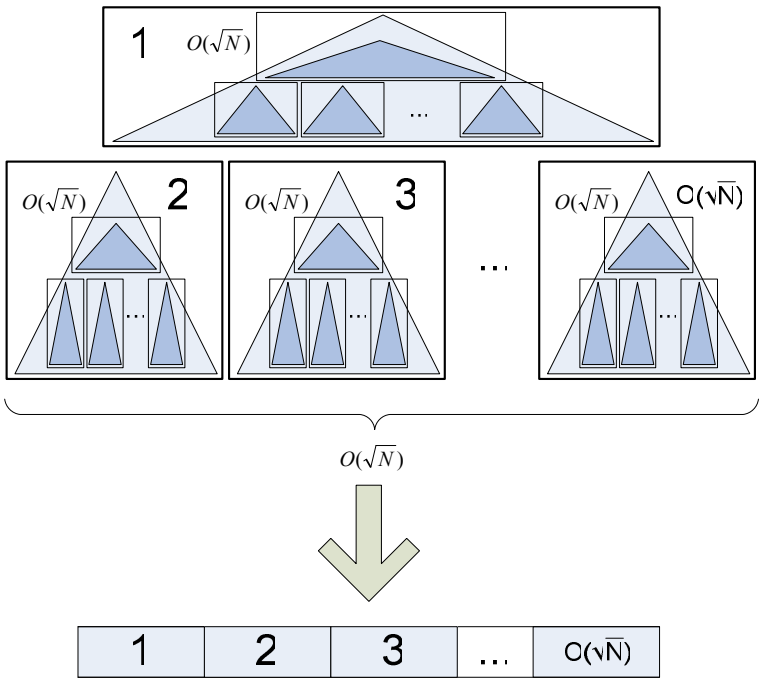

Рис. 1.4. Укладка ван Эмде Боаса

Она строится рекурсивно. Сначала дерево размера $O(N)$ разделяется на *верхушку* размера $O(\sqrt{N})$, и поддеревья, которые свисают с ее листьев, количество и размеры которых также составляют $O(\sqrt{N})$. В память сначала рекурсивно укладывается верхушка, а затем – все листья по порядку. В работе [7] доказывается, что при поиске в таком дереве число операций составляет $O(\log N)$, где N – размер дерева, а число используемых переносов

блоков памяти – $O(\log_L N)$, где L – размер блока памяти. Эта оценка соответствует оценке для Б-дерева с размером узла порядка L и является оптимальной.

Для того, чтобы сделать на основе этой идеи динамическую поисковую структуру данных, дерево поиска дополняется до полного пустыми ячейками, которые используются для вставки новых элементов. При этом, когда зарезервированные пустые ячейки заканчиваются, происходит перебалансировка участка, в котором это произошло.

Подобная укладка дерева играет важную роль в разработке динамических структур данных, нечувствительных к размеру кеша, так как за счет своей рекурсивной структуры обеспечивает близость соседних областей дерева в запакованном массиве на всех масштабах одновременно.

1.2. Параллельные алгоритмы

В настоящее время параллельные системы играют очень большую роль, и со временем их значимость только увеличивается.

Знаменитый закон Мура, сформулированный еще в 1965 году и гласящий, что вычислительная мощь компьютеров будет увеличиваться вдвое каждые полтора-два года, успешно выполнялся до недавнего времени. Это достигалось путем увеличения количества транзисторов в микросхемах процессоров и увеличения их тактовой частоты.

Однако в последние десятилетия темп роста производительности стал заметно снижаться. Это было связано с постепенным насыщением существующих технологий и приближением к теоретическому максимуму их возможностей. Плотность размещения различных электронных элементов на маленьких кристаллах процессоров стала такой большой и требовала такой точности, что дальнейшее ее увеличение с трудом могло достигаться существующими технологиями. Аналогичная ситуация сложилась и с тактовой частотой.

Ввиду вышеописанных причин, производители вычислительных систем обратились к новому подходу: вместо того, чтобы ускорять частоту и усложнять чипы процессоров они стали делать процессоры с несколькими ядрами и машины с несколькими процессорами, которые выполняют инструкции параллельно и одновременно. В настоящее время даже на подавляющем большинстве персональных компьютеров в той или иной степени используются многоядерные процессоры. Следовательно, можно действительно утверждать, что ближайшее будущее развития сферы вычислительных устройств стоит за параллельными системами.

Далее в этом разделе будут рассмотрены основы принципы параллельного программирования, различные теоретические конструкции, которые применяются при проектировании параллельных алгоритмов, а также будет дан обзор существующих параллельных алгоритмов и структур данных.

1.2.1. Параллельные системы

Модель параллельной системы можно схематически изобразить, как несколько независимых процессоров, которые параллельно работают с одной *разделяемой* памятью. Это, так называемая, *симметричная мультипроцессорность* (*SMP – Symmetric Multiprocessing*), так как все процессоры имеют одинаковые роли, и каждый из них может работать со всей доступной памятью. Существуют и другие, альтернативные подходы, такие как, например архитектура *несимметричного доступа к памяти* (*NUMA – Non Uniform Memory Access*), в которой за каждым процессором закрепляется отдельный участок основной памяти, а их взаимодействие производится через небольшой, но быстрый участок разделяемой памяти. В противопоставление ней архитектуру *SMP* также еще называют *UMA* (*Uniform Memory Access*). Другим вариантом архитектуры параллельной системы является *асимметрическая многопроцессорность* (*ASMP*), при которой различные процессоры имеют разное строение и выполняют различные функции (например, в системах обработки трехмерной графики).

Далее мы не будем вдаваться в технические детали различных вычислительных систем, и абстрагируемся моделью, состоящей из симметричных процессоров, каждый из которых имеет свою локальную память, а также имеет доступ к общей разделяемой памяти (рис. 1.5).

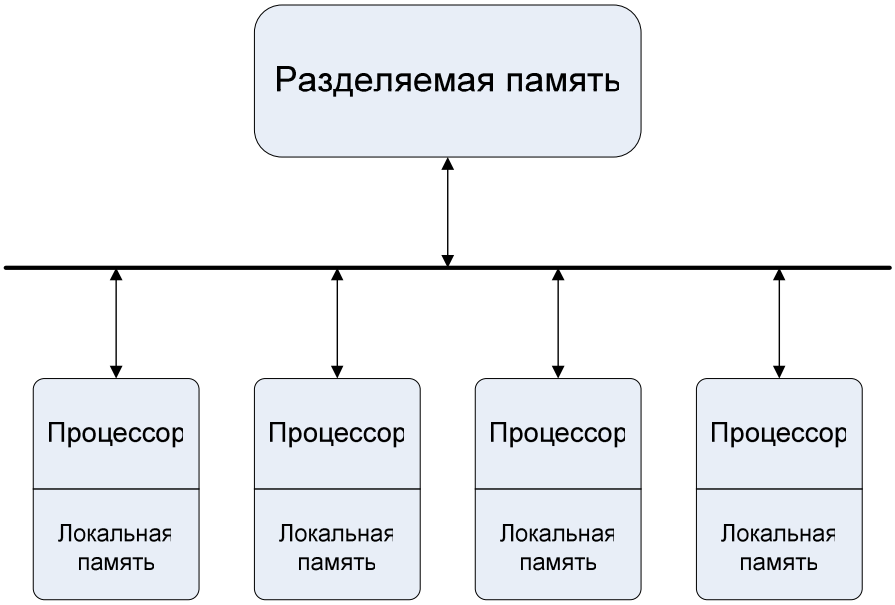

Рис. 1.5. Схематическое изображение многопроцессорной системы

Для того чтобы выполнять параллельные программы, необходимо, чтобы кроме физической реализации многопроцессорности система давала доступ к многопроцессорным примитивам на уровне операционной системы. Аналогами процессоров в операционных системах являются *процессы* и *потоки*. Они представляют собой отдельные программы, которые выполняются системой, и, по сути, являются виртуальными процессорами. Различие между процессами и потоками заключается в том, что процессы – это группы, состоящие из нескольких потоков, которые работают с одним общим участком памяти, но имеют раздельные стеки вызовов и регистры. Современные операционные системы могут эмулировать многопроцессорность и на одном физическом процессоре, поочередно используя его для работы различных потоков выполнения.

Далее будем использовать абстрактную модель, изображенную на рис. 1.5. При этом термины *процесс* и *процессор* считаются синонимами.

Более подробный обзор параллельных систем можно найти в первой главе работы [8].

1.2.2. Сложности параллельного программирования

Составление программ для *параллельных* систем сопряжено со многими трудностями и проблемами, которые не были известны при изучении классических, *последовательных* систем.

Рассмотрим классический пример: упрощенный вариант системы банковских счетов. Пусть все счета представлены в виде ячеек массива, а различные потоки выполняют операции над ними.

Операция перевода некоторой суммы денег с одного счета на другой может выглядеть примерно так:

```
transfer(i, j, x) {
    v[i] = v[i] – x;
    v[j] = v[j] + x;
}
```

Этот код на языке высокого уровня будет скомпилирован в примерно следующий код на ассемблере:

```
1   mov x, R1
2   mov v[i], R2
3   sub R2, R1
4   mov R2, v[i]
5   mov v[j], R2
6   add R2, R1
7   mov R2, v[j]
```

Пусть на счету номер *X* имеется 350$, а на счету *Y* – 0$, и два потока (*А* и *Б*) выполнения хотят перевести по 100$ каждый со счета *X* на счет *Y*. Так как они будут выполняться параллельно, то возможная последовательность их действий может выглядеть следующим образом:

1. Процесс *A* выполняет три первые ассемблерные инструкции и получает в регистре *R2* новое значение счета *X* – 250 (350$ - 100$).

2. Процесс *Б* делает то же самое и получает в своем регистре *R2* значение 250.
3. Процесс *А* выполняет оставшиеся инструкции, и записывает в *v[X]* значение 250, а в *v[Y]* значение 100.
4. Процесс *Б* завершает выполнение своих оставшихся инструкций. При этом в ячейке *v[Y]* он прочтет уже новое значение 100 и увеличит его еще на 100, записав туда в итоге 200. Но перед этим, в строке 4 он присвоит ячейке *v[X]* значение своего регистра *R2*, а именно – 250.

Таким образом, после параллельного выполнения двух перекрывающихся операций `transfer` значения счетов *X* и *Y* будут соответственно 250$ и 200$, что в сумме дает на 100$ больше, чем было до выполнения операций.

Этот пример иллюстрирует так называемую проблему *состояния гонки* (*race condition*), когда конечный результат последовательности действий напрямую зависит от их очередности их выполнения. Нетрудно также придумать пример последовательности действий, ведущей к тому, что суммарное значение балансов счетов будет на 100$ меньше исходного.

Для того чтобы избежать возникновения подобной проблемы, необходимо придумать некоторые механизмы, которые позволяли бы обеспечить выполнение условия, что операции, выполняемые одним процессом, не могут повлиять на результат операций, выполняемых другим процессом параллельно с ним. Это так называемая *задача синхронизации потоков исполнения*.

Простейшим подходом к решению этой проблемы является обеспечение того, чтобы некоторые участки кода просто не могли выполняться параллельно разными потоками. Такие участки кода принято называть *критическими секциями*. В нашем примере выше критической секцией является блок ассемблерных команд со второй по четвертую,

которые соответствуют чтению, изменению и записи нового значения одной ячейки памяти.

Задача избежания выполнения критической секции несколькими процессами одновременно называется *задачей взаимного исключения* (*mutual exclusion problem*). Как ни странно, даже решение, казалось бы, такой простой задачи не является тривиальным. Первым сформулировал эту задачу и предложил алгоритм ее решения Дейкстра в 1965 году [9], а позднее Лампортом [10] и Петерсоном [11] были предложены другие, более простые и практические решения.

Кроме программных алгоритмов, позволяющих получить достигнуть взаимного исключения, интерес представляют реализованные на аппаратном уровне *операции с повышенной атомарностью*. Операция называется атомарной, если она всегда выполняется так, что никакой другой поток исполнения не мог повлиять на ее промежуточный результат. В примере с банком операция изменения баланса счета на заданную величину не является атомарной, так как параллельное выполнение двух таких операций привело к ошибке. Однако если бы было возможно выполнить эту операцию «зараз», используя не три ассемблерные инструкции, как в приведенном примере, а одну инструкцию, которая сразу же читала, изменяла и записывала бы значение в ячейку памяти, то проблема, описанная выше, не возникала бы. Подобные более сложные, чем чтение и запись, атомарные инструкции будут рассмотрены в следующем параграфе.

1.2.3. Синхронизация без блокировок

Решением задачи взаимного исключения является синхронизационный примитив, называемый *мьютекс* (*mutex – mutual exclusion*; в английской терминологии также обычно используется термин *lock – «замок»*). Он представляет собой объект, содержащий два метода: `lock()` и `unlock()`. Реализация этих методов зависит от используемого алгоритма решения задачи взаимного исключения (см. предыдущий

параграф), но схема его использования всегда одинакова. Перед тем, как получить доступ к критической секции, поток исполнения вызывает метод `lock()`, затем выполняется собственно код критической секции, а затем вызывается метод `unlock()`:

```
lock();
<код критической секции>
Unlock();
```

Все другие потоки действуют точно так же. При этом корректная реализация методов `lock()` и `unlock()` должна обеспечивать то, что ни в какой момент времени никакие два потока не будут одновременно находиться внутри критической секции (свойство *взаимного исключения*). После того, как один из потоков выполнит метод `lock()`, все другие будут бесконечно зацикливаться внутри этого метода, до тех пор, пока первый поток не выполнит метод `unlock()`. Если же несколько потоков начнут выполнение метода `lock()` одновременно, то гарантируется, что лишь один из них выйдет из этого метода, не зациклившись.

Мьютекс является не единственным примитивом, с помощью которого можно решать задачу взаимного исключения. Имеется ряд более сложных конструкций, называемых *синхронизационными примитивами*, таких как *мониторы* и *семафоры*, которые подробно описываются в третьей главе работы [8].

Подобный подход к задаче синхронизации потоков исполнения называется *блокирующим* (*blocking, lock-based*), так как поток, который выполняет код критической секции, блокирует работу всех остальных потоков, которые хотят ее использовать, до тех пор, пока не выйдет из нее сам.

У такого подхода существует множество недостатков. Если какой-то поток в процессе выполнения кода критической секции сгенерирует ошибку и некорректно завершится, то все остальные потоки никогда уже не смогут войти в критическую секцию (так как первый не выполнит в этом случае

вызов метода `unlock()`), и вся параллельная система попросту зависнет. Даже если поток, выполняющий критическую секцию, никогда не сломается, он может быть очень медленным, и заставлять подолгу ждать все остальные потоки, сводя на нет все преимущества параллельной системы.

Другой опасностью данного подхода является тот факт, что если критических секций и соответствующих им мьютексов несколько, то это может привести к *взаимной блокировке* или *тупику* (*dead-lock*). Эта такая ситуация, когда несколько потоков заблокировали несколько ресурсов (критических секций) но каждый из них хочет получить доступ к некоторым другим ресурсам, которые, в свою очередь, заблокированы другими потоками. Эту ситуацию иллюстрирует рис. 1.6.

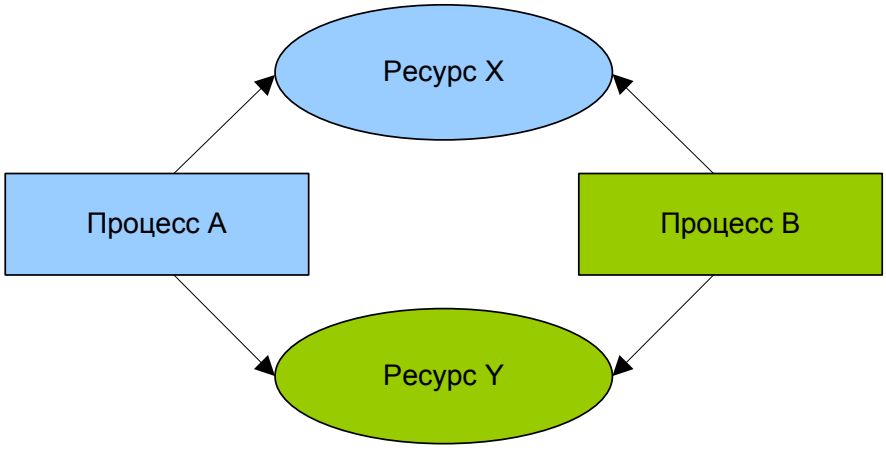

Рис. 1.6. Взаимная блокировка

Из-за указанных недостатков использование блокирующих алгоритмов в задачах, от которых требуется повышенная отказоустойчивость и в системах реального времени оказывается затруднено.

Существует класс параллельных алгоритмов, которые используют другой подход. Алгоритмы, в которых для синхронизации не используется механизм, называются *не блокирующими алгоритмами* (*non-blocking algorithms*). Они в свою очередь также делятся на разные категории, в зависимости от того, какими свойствами они обладают. Рассмотрим их подробнее.

Алгоритм с отсутствием блокировок (*lock-free algorithm*) – это такой алгоритм, который гарантирует, что вне зависимости от текущего состояния

системы и от того, в какой очередности будут выполняться команды разных процессов, какой либо процесс гарантированно завершит свою операцию за конечное число шагов. В зависимости от очередности выполнения команд это могут быть разные процессы, однако в любом случае такой всегда должен найтись. Из этого определения следует, что никакой блокирующий алгоритм не может являться алгоритмом с отсутствием блокировок, так как любой процесс, заблокировавший доступ к критической секции, может работать очень медленно, или быть «заморожен» системой на любой, сколь угодно большой промежуток времени, и в таком случае, если все остальные алгоритмы заинтересованы в доступе к критической секции, то за этот промежуток ни один из них не завершится.

Однако не каждый не блокирующий алгоритм является алгоритмом с отсутствием блокировок. Кроме опасности попасть во взаимную блокировку существует еще также опасность *живой блокировки* (*livelock*). В случае живой блокировки ни один из процессов в явном виде не блокирует никакие ресурсы, однако при этом процессы одновременным выполнением своих операций мешают друг другу, и могут выполняться бесконечно долго.

Свойство отсутствия блокировок автоматически гарантирует, что даже если какой-то процесс будет некорректно прерван посреди выполнения своей операции, это не повлияет на способность других процессов выполнять свою работу.

Другим важным свойством, которым могут обладать не блокирующие алгоритмы, является свойство *отсутствия ожидания* (*wait-free*). Алгоритмы с отсутствием ожидания гарантируют, что независимо от состояния системы про любой процесс известно, что он завершит свое выполнение за некоторое конечное, заранее определенное время, независимо от действий других процессов. Это свойство сильнее предыдущего, то есть из отсутствия ожидания автоматически следует отсутствие блокировок. По сути, отсутствие ожидания означает, что ни один процесс не может повлиять на

остальные и задержать их выполнение более чем на некоторое фиксированное количество операций.

Разработка алгоритмов с отсутствием ожидания достаточно сложна, и в настоящее время их существует не так много. Такие алгоритмы обычно бывают наиболее эффективными, так как лучше всего поддаются масштабированию, путем увеличения работающих числа процессов.

Еще один класс алгоритмов – алгоритмы *с отсутствием препятствий* (*obstruction-free*). Это свойство слабее двух предыдущих. Считается, что в алгоритме отсутствуют препятствия, если для любого допустимого состояния системы верно, что если взять любой процесс и заморозить выполнение всех остальных процессов, то он завершится за конечное число шагов. Таким образом, это свойство не требует отсутствия живой блокировки, однако блокирующие алгоритмы ему, тем не менее, не удовлетворяют.

Классической задачей, связанной синхронизацией без ожидания, является *задача о консенсусе* [13]. Она формулируется следующим образом: имеется несколько потоков исполнения, каждому из которых на вход дано некоторое число. Необходимо построить алгоритм на основе заданного синхронизационного примитива, который позволит процессом синхронизироваться без ожидания таким образом, что все они после завершения своей работы вернут в качестве результата одно из множества чисел, поданных им на вход. При этом некоторые процессы могут быть остановлены в произвольный момент времени.

Эта задача является модельной при анализе различных свойств синхронизационных примитивов и при оценке возможности их теоретического использования для построения алгоритмов без ожидания. В частности известно, что задачу о консенсусе нельзя решить, имея лишь возможность атомарно записывать и читать данные из одной ячейки памяти, даже для двух процессов [13]. Поэтому в алгоритмах без ожидания используются более сложные атомарные примитивы, которые

поддерживаются современными системами на аппаратном уровне. Две наиболее распространенные из них *CAS* (*Compare And Swap – сравнение с обменом*) и *LL/SC* (*Load-Linked / Store-Conditional*).

Примитив *CAS* поддерживает две операции:

- Загрузка значения из определенной ячейки памяти.
- `CAS(x, old_value, new_value)` – сравнение значения в определенной ячейке памяти `x` с заданным ожидаемым значением `old_value` и замена его на новое заданное значение `new_value` в случае равенства (собственно, операция сравнения с обменом). Эта операция возвращает логическое значение, которое истинно, если равенство выполнялось, и новое значение было успешно записано.

Примитив *LL/SC* также поддерживает две операции:

- Загрузку значения из ячейки памяти (*load-linked*).
- Запись нового значения в ячейку памяти, при условии, что эта ячейка не была изменена другим процессом с момента последней загрузки (*store-conditional*). Так же, как и в примитиве *CAS* эта операция возвращает логическое значение «истина», если ячейка памяти не менялась до этой записи, и новое значение было записано, и «ложь» в противном случае.

Эти два примитива очень похожи, однако у операции *CAS* существует один недостаток, называемый *проблемой ABA* (*ABA problem*). Суть ее заключается в следующем. Пусть значение ячейки памяти `x` было равно `A`, затем она была загружена некоторым процессом, и затем успешно перезаписана при помощи операции `CAS(x, A, C)`. Однако это вовсе не означает, что между выполнением этих двух операций значение ячейки x не менялось. Другой процесс мог в это время два раза изменить значение ячейки `x` с `A` на `B`, а затем обратно с `B` на `A`. Такая проблема отсутствует у примитива

LL/SC, а в случае *CAS* обычно решается использованием дополнительных *маркеров версии*.

Подробный теоретический обзор различных не блокирующих алгоритмов можно найти в работах [12 – 14], а также в работе [8].

1.2.4. Обзор существующих параллельных структуры данных

Существует множество различных интересных алгоритмов для параллельных структур данных. В работе [15] предлагается алгоритм с отсутствием препятствий для реализации дека (двусторонней очереди). В работах [16–18] рассмотрен ряд блокировочных и безблокировочных алгоритмов для поисковых структур данных на основе двоичных деревьев поиска и *Б*-деревьев.

Что касается параллельных алгоритмов, нечувствительных к размеру кеша, то здесь наиболее интересной является работа [19]. В ней рассматривается подход, позволяющий модифицировать *COB*-дерево (упоминавшееся в разд. 1.1.4) таким образом, чтобы иметь возможность параллельно производить над ним операции. Один из алгоритмов, рассмотренных в данной работе, является алгоритмом с отсутствием блокировок, однако все перечисленные там модификации *COB*-дерева имеют весьма сложную структуру и представляют скорее чисто теоретический интерес.

Существуют различные параллельные реализации структуры данных очереди без блокировок. Большинство из них построено на основе связного списка. Одной из наиболее интересных с практической точки зрения является реализация очереди Майкла и Скотта [20]. Эта реализация довольно проста, и в то же время обладает свойством отсутствия блокировок и эффективна на практике. Алгоритм очереди Майкла и Скотта был реализован фирмой *Sun* и включен в состав стандартной библиотеки алгоритмов в языке *Java*, начиная с версии *Java SDK 1.5*. Обзор других алгоритмов параллельной реализации *FIFO* очереди и существующих у них недостатков можно также найти в

работе [20]. Есть так же несколько новых работ, которые описывают, как в той или иной мере можно улучшить алгоритм Майкла и Скотта [21, 22].

ВЫВОДЫ ПО ГЛАВЕ 1

В данной главе были рассмотрены различные теоретические модели вычислений, в частности различные модели доступа к памяти, а так же модели параллельных вычислений, и связанные с ними теоретические понятия и примитивы. Кроме того, был дан обзор различных существующих параллельных алгоритмов и структур данных.

ГЛАВА 2. АЛГОРИТМ ОЧЕРЕДИ С ОТСУТСТВИЕМ БЛОКИРОВОК НА ОСНОВЕ РАЗВЕРНУТОГО СВЯЗНОГО СПИСКА

В этой главе будет представлен алгоритм очереди с отсутствием блокировок на основе развернутого связного списка. Также будут изложены идеи, которые послужили для него основой, и будет приведен теоретический анализ некоторых свойств этого алгоритма.

2.1. Основные идеи алгоритма

В данном разделе будут рассмотрены три основных идеи, на основе которых спроектирован данный алгоритм.

2.1.1. Алгоритм очереди без блокировок Майкла и Скотта

Основой для приведенного в данной работе алгоритма очереди на основе развернутого связного списка послужил алгоритм очереди Майкла и Скотта [20]. Рассмотрим более подробно его суть.

Очередь Майкла и Скотта строится на основе обычного односвязного списка (рис. 2.1).

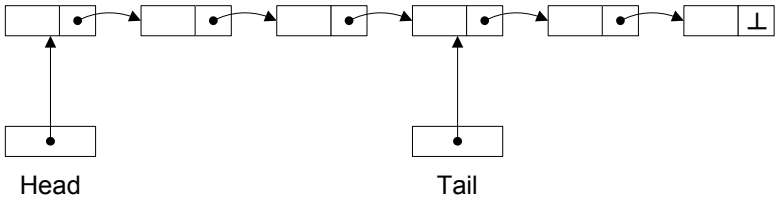

Рис. 2.1. Очередь Майкла и Скотта

Голова и хвост списка соответствуют голове и хвосту очереди, а элементы очереди хранятся в ячейках списка по одному. Важной особенностью очереди Майкла и Скотта является тот факт, что головной элемент списка не содержит значения первого элемента очереди, а является вспомогательным «лишним» узлом (*dummy-node*). Этот нехитрый прием позволяет не опасаться случая, когда очередь становится пуста, и корректно его обрабатывать.

Для того чтобы атомарно изменять указатели на узлы списка, а так же на голову и хвост очереди, в данном алгоритме используется атомарная инструкция *CAS*. Корректность алгоритма следует из поддержания в каждый момент времени пяти простых условий:

1. Список всегда остается связным.
2. Новые узлы всегда вставляются только после последнего узла списка.
3. Узлы удаляются только из головы списка.
4. Указатель *головы* всегда указывает на первый элемент списка.
5. Указатель *хвоста* всегда указывает на некоторый узел списка.

Алгоритм очереди Майкла и Скотта не использует блокирующих механизмов. Для того чтобы получить согласованное состояние нескольких указателей, используется принцип нахождения атомарного снимка без блокировок: сначала значения всех указателей читаются по очереди, а затем проверяется, что ни один из них не успел поменяться за время чтения. В противном случае указатели приходится читать повторно, и это может повторяться бесконечное количество раз, поэтому данный алгоритм не является алгоритмом без ожидания. Однако так как каждое изменение указателей производится только в процессе успешного выполнения операций над очередью, то если какому-то потоку приходится терять время, заново читая указатели, то это автоматически означает, что какой-то другой поток успешно выполнил свою операцию. Такой подход обеспечивает данному алгоритму свойство отсутствия блокировок.

2.1.2. Развернутый связный список

Развернутый связный список [23] (*unrolled linked list*) представляет собой модификацию обыкновенного связного списка, в узлах которого хранится не один элемент, а сразу несколько (рис. 2.2).

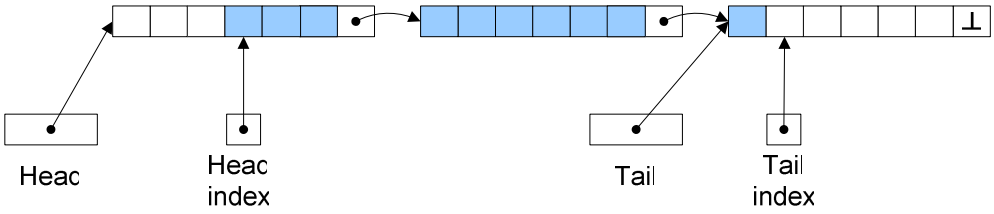

Рис. 2.2. Развернутый связный список

Эта структура обладает рядом важных преимуществ над обычным связным списком. Во-первых, она позволяет значительно экономить память, так как на `K` указателей на данные здесь приходится всего один служебный указатель на следующий элемент списка, а не `K`, как в случае обычного списка. Во-вторых, в массиве все элементы располагаются в памяти последовательно, что играет важную роль для эффективной работы с кешом.

Идея развернутого связного списка довольно проста, и реализовать его в рамках обычной последовательной модели не представляет большого труда. В разд. 2.2 будет показано, как применить подобную идею для реализации параллельной структуры данных.

2.1.3. Простая очередь на массиве с отсутствием блокировок

В работе [21] приводится идея, позволяющая реализовать простой алгоритм очереди на массиве с отсутствием блокировок. Чтобы понять суть этой идеи рассмотрим сначала классическую структуру обычной, последовательной очереди на массиве [24, глава 2] и проблемы, возникающие при простом, наивном подходе к организации на ее основе параллельной очереди без блокировок с использованием примитива *CAS*.

Очередь на массиве состоит из собственно массива `a` размера `N`, хранящего значения элементов и указателей `head` и `tail` на ее начало и конец (рис. 2.3). Указатель `head` показывает на первый элемент очереди, а `tail` – на первую пустую ячейку сразу же после конца очереди (изначально оба указателя равны 0).

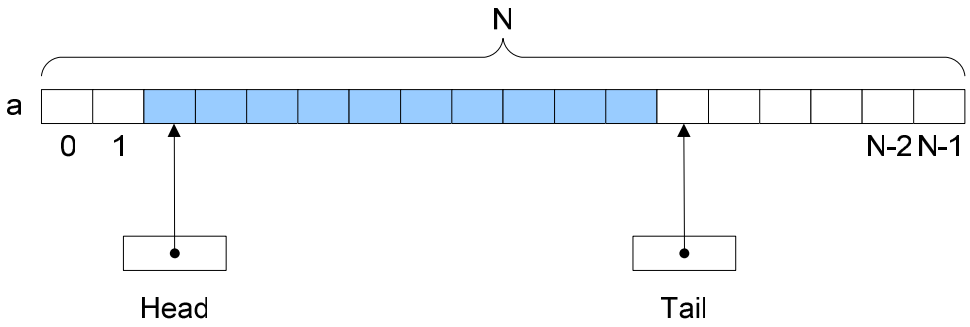

Рис. 2.3. Классическая очередь на массиве

В последовательном алгоритме нас не интересуют значения, которые хранятся в ячейках вне очереди. Так как мы всегда смотрим на указатели `head` и `tail` и никогда не читаем значения ячеек вне очереди, то там может храниться произвольный мусор. Алгоритмы добавления (`enqueue(x)`) и извлечения (`dequeue()`) элементов очереди будут примерно следующими:

```
1.   enqueue(x) {
2.         if tail = N {
3.               error // очередь дошла до конца
4.         }
5.         a[tail] = x
6.         tail = tail + 1
7.   }
```

```
1.   dequeue() {
2.         if head = tail {
3.               error // очередь пуста
4.         }
5.         result = a[head]
6.         head = head + 1
7.         return result
8.   }
```

При реализации параллельного доступа к очереди мы не можем опираться лишь на значения указателей `head` и `tail`. При параллельном выполнении двух операций `enqueue` они могут записать новые значения в одну и ту же ячейку памяти, и затем сдвинуть указатель `tail`, и при этом одно значение попросту потеряется. Аналогичная ситуация будет и с операциями `dequeue`. Так как мы не можем одновременно атомарно менять ячейки массива `a` и указатели `head` и `tail`, то последние теперь могут играть лишь вспомогательную роль подсказок, так как могут отставать от реальных позиций соответствующих ячеек. Естественный путь для

избежания этой проблемы – использование специальных значений NULL для пустых ячеек, и изменение всех ячеек памяти при помощи операции *CAS*:

```
1.  enqueue(x) {
2.        loop {
3.              t = tail
4.              if t = N {
5.                    error // очередь дошла до конца
6.              }
7.              if a[t] != NULL {
8.                    CAS(tail, t, t + 1) // в tail было
9.                    continue            // старое значение
10.             }
11.             if CAS(a[t], NULL, x) {
12.                   CAS(tail, t, t + 1)
13.                   return // успешно записали, иначе повтор
14.             }
15.       }
16. }
```

```
1.  dequeue() {
2.        loop {
3.              h = head
4.              t = tail
5.              if h = t {
6.                    error // очередь пуста
7.              }
8.              result = a[h]
9.              if result == NULL {
10.                   CAS(head, h, h + 1) // в head было
11.                   continue            // старое значение
12.             }
13.             if CAS(a[h], result, NULL) {
14.                   CAS(head, h, h + 1)
15.                   return result // успешно считали
16.             }
17.       }
18. }
```

Подобный подход, однако, содержит один тонкий момент. Он не учитывает случая, когда очередь почти пуста. Рассмотрим такой вариант развития событий, изображенный на рис. 2.4: пусть некоторый поток хочет записать новый элемент *x* в конец очереди. Он считывает значение указателя head в переменную h (1). Сразу же после этого другие потоки выполняют множество операций над очередью, так что очередь успевает сдвинуться по массиву далеко направо (2). После этого первый поток «успешно» заменяет при помощи *CAS* значение в ячейке t на x (3). При этом он убеждается, что

до этого в этой ячейке было значение `NULL`, и считает, что его никто не опередил. Это типичный случай проблемы *ABA*.

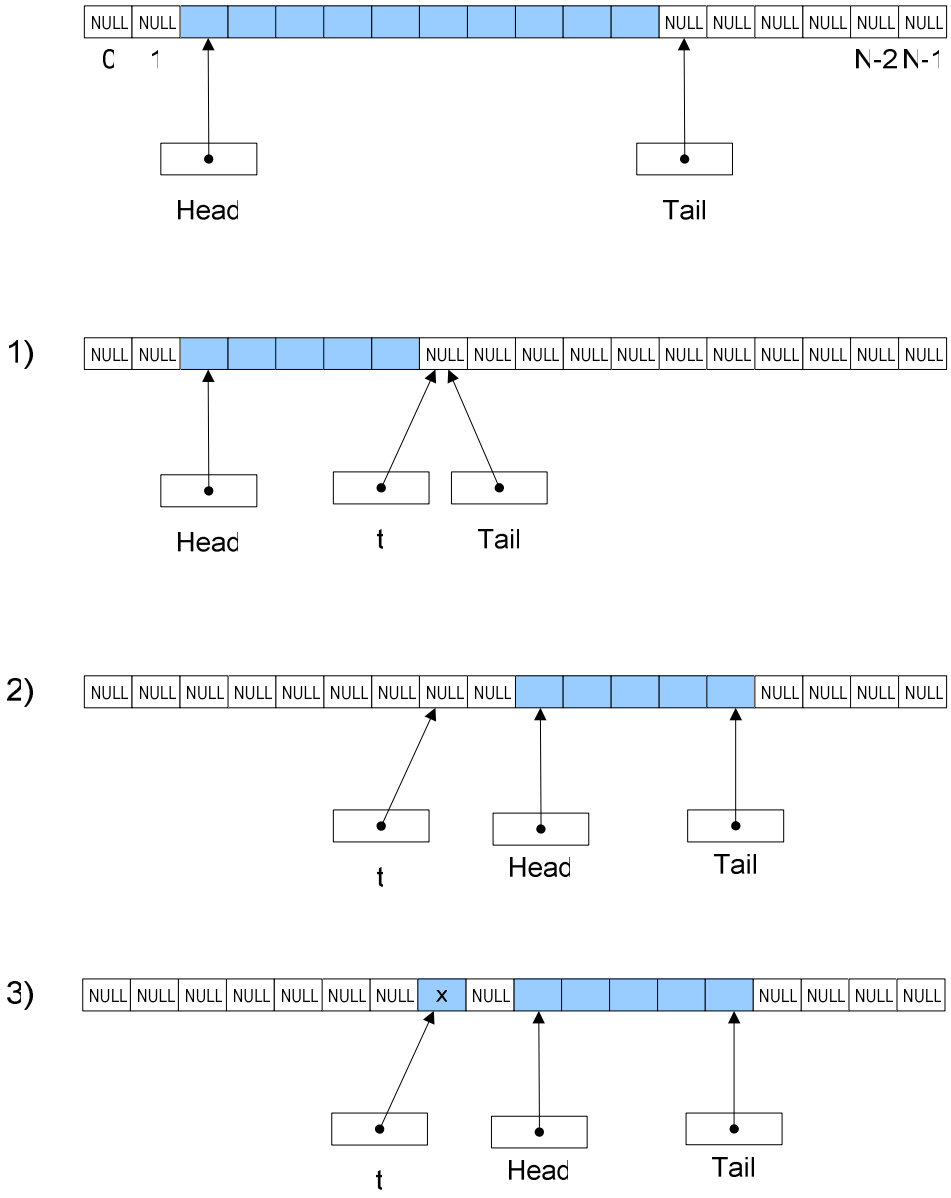

Рис. 2.4. Второй подход и его проблема

В работе [21] предлагается такой подход к решению этой проблемы: нужно завести два различных специальных значения `NULL(0)` и `NULL(1)` для ячеек, которые находятся правее и левее текущего положения очереди соответственно (рис. 2.5). Тогда *CAS*, замещающий элемент не сможет перепутать эти два типа ячеек, и алгоритм станет работать корректно. Подобная реализация будет приведена в разд. 2.2.2, и будет использована для построения нового алгоритма.

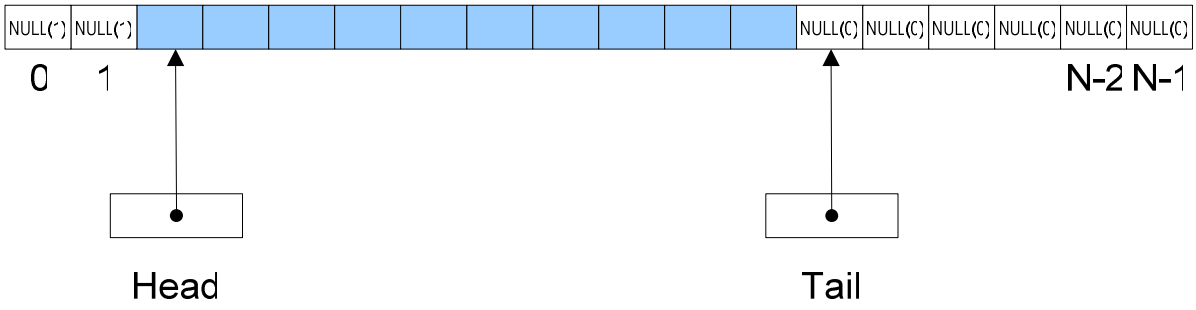

Рис. 2.5. Корректная реализация параллельной очереди на массиве

Очередь на обычном массиве сама по себе представляет малый интерес, так как она рано или поздно заканчивается, доползая до края массива, и ее нельзя использовать бесконечно долго. В работе [21] такой подход применяется для построения *циклической* очереди на массиве [24, глава 2]. В работе применяются специальные механизмы, позволяющие корректно обрабатывать случай, когда очередь почти полна и указатель `tail` догоняет `head`. Для того чтобы реализовать зацикливание, вводится дополнительная переменная, которая указывает, какое из значений `NULL(0)` и `NULL(1)` в данный момент используется для затирания значений при извлечении элементов из очереди (значения `NULL(0)` и `NULL(1)` по мере циклического продвижения очереди поочередно меняются ролями).

Подобный подход позволяет избежать возникновения проблемы *ABA*, однако он все же теоретически не исключает вероятности возникновения *проблемы ABA'B'A*. Теоретически, пока один процесс пытается записать или извлечь элемент очереди, она может совершить один полный оборот по массиву (то есть остальные процессы успеют выполнить порядка N операций) и тогда возникнет проблема, аналогичная проблеме предыдущего рассмотренного примера. Авторы работы [21] предлагают обобщить идею двух значений `NULL` на K значений `NULL(0)`, `NULL(1)`, … `NULL(K - 1)`, тем самым, сделав вероятность возникновения проблемы *ABA'B'A* сколь угодно малой. Возможно также применение дополнительных *маркеров версий* (разд. 1.2.3), которые, по сути, моделируют *LL/SC* через *CAS*. Однако это сведет на нет все преимущества того, что структура реализована на массиве (так как вместо одной ячейки массива на каждый элемент очереди

потребуется выделять в памяти отдельную запись с двумя полями: значением и маркером версии).

2.2. Реализация методов

В данном разделе будет приведена реализация операций над очередью на развернутом связном списке.

2.2.1. Общая структура очереди на основе развернутого связного списка

Общая схема структуры данных, рассматриваемой в данной работе, изображена на рис. 2.6.

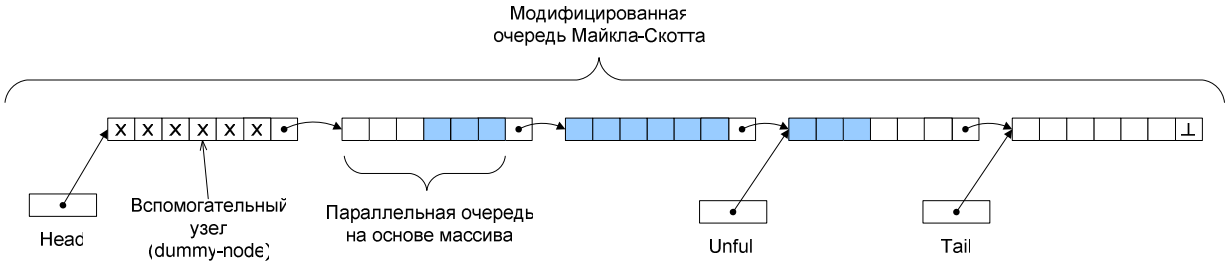

Рис. 2.6. Очередь на основе развернутого связного списка

Она является, по сути, модифицированной очередью Майкла и Скотта, которая в качестве элементов в узле хранит не сами значения, а целые очереди на основе массивов. Таким образом, выполнение операций извлечения и вставки над этой структурой данных распадается на два шага:

1. Нахождение во «внешнем» связном списке нужного узла, в котором будет выполнена необходимая операция.
2. Выполнение операции над очередью на массиве. При невозможности этого сделать, возможно, производится изменение «внешней» очереди на связном списке и повторяется шаг 1.

Отличием данной структуры от классической очереди Майкла и Скотта является наличие третьего указателя `unfull`, помимо старых `head` и `tail`. Из-за параллельного выполнения операций может возникнуть такая

ситуация, когда в конце внешней связной очереди содержится сразу несколько пустых очередей на массиве. Переменная `unfull` указывает на тот промежуточный узел очереди, который является первым из тех, в которых еще есть место.

К пяти условиям, постоянно поддерживаемым в оригинальной очереди Майкла и Скотта, сформулированным в разд. 2.1.1, добавляется несколько дополнительных:

6. Указатель `unfull` всегда указывает на элемент списка, причем этот элемент всегда лежит не левее элемента, на который указывает `head` (но может совпадать с ним). Относительное положение элементов `unfull` и `tail` в принципе может быть любым.
7. Левее элемента, на который указывает `unfull`, никогда нет других элементов, в которых еще есть место для вставки.

Полученная структура данных сложнее, чем исходная очередь Майкла и Скотта, и, по сути, содержит ее как часть, однако, у нее есть ряд преимуществ, которые будут рассмотрены далее.

2.2.2. Реализация очереди на массиве

Для начала рассмотрим реализацию безблокировочной очереди на массиве, идея которой была изложена в разд. 2.1.3. Код ее методов приведен в листинге 2.1.

Листинг 2.1. Реализация очереди на массиве

```
const N: integer;
a: array[0..N-1] of E;
head, tail: integer;
const NULL_UNPROCESSED, NULL_PROCESSED: E;


initialize()
      a = new integer[0..N-1];
      for i = 1..N
            a[i] = NULL_UNPROCESSED;
      head = tail = 0;


enqueue(e: E): Boolean
```

```
e1:         loop
e2:               t = tail;
e3:               if (t = N)
e4:                     return false;
e5:               while (a[t] != NULL_UNPROCESSED)
e6:                     t = t + 1;
e7:                     tail = t;
e8:                     if (t = N)
e9:                           return false;
e10:              if (CAS(a[t], NULL_UNPROCESSED, e))
e11:                    tail = t + 1;
e12:                    return true;

      dequeue(): E
d1:         h = head;
d2:         if (h = N)
d3:               return NULL;
d4:         loop
d5:               v = a[h];
d6:               if (v = NULL_PROCESSED)
d7:                     h = h + 1;
d8:                     head = h;
d9:                     if (h = N)
d10:                          return NULL;
d11:              else if (v = NULL_UNPROCESSED)
d12:                    return NULL;
d13:              else
d14:                    if (CAS(a[h], v, NULL_PROCESSED))
d15:                          head = h + 1;
d17:                          return v;

      peek(): E
p1:         h = head;
p2:         if (h = N)
p3:               return NULL;
p4:         loop
p5:               v = a[h];
p6:               if (v = NULL_PROCESSED)
p7:                     h = h + 1;
p8:                     head = h;
p9:                     if (h = N)
p10:                          return NULL
p11:              else if (v = NULL_UNPROCESSED)
p12:                    return NULL;
p13:              else
p14:                    return v;

      isExhausted(): boolean
            return (a[N - 1] = NULL_PROCESSED);
```

Рассмотрим реализацию операции помещения элемента в очередь `enqueue`. Первым делом атомарно читается указатель на хвост очереди (e2). Затем производится проверка того, что очередь еще не дошла до конца (e3–e4). Далее производится проверка того, что нас никто не опередил и место `t` действительно свободно (e5–e9). В случае если это так, то делается попытка заменить с помощью операции *CAS* пустое значение в ячейке `t` на `v` (e10–

e12). В случае успеха указатель `tail` обновляется, и процесс прерывается, иначе идет повторное выполнение цикла (e1).

Операция извлечения из очереди `dequeue` действует похожим образом. Сначала читается указатель на голову, и проверяется, что еще не достигли конца (d1–d3). Далее в цикле делается попытка прочитать значение, соответствующее концу очереди. Здесь возможно три варианта:

- (d6–d10) Текущий процесс опередили, и это значение пустое (`NULL_PROCESSED`), значит его забрал кто-то другой. Продолжаем дальше.
- (d11–d12) Текущий процесс дошел до еще не использованной ячейки (`NULL_UNPROCESSED`), и, следовательно, очередь пуста.
- (d13–d17) В ячейке записано нормальное значение. В этом случае процесс пытается его заменить при помощи *CAS*.

Операция `peek`, которая возвращает значение головы очереди, но не удаляет его, действует аналогично операции `dequeue`.

Вспомогательная функция `isExhausted()` возвращает значение `true` тогда и только тогда, когда очередь уже была полностью использована и дошла до конца массива. Она понадобится нам в дальнейшем.

2.2.3. Реализация основной структуры

Реализация операций над итоговой параллельной очередью без блокировок на основе развернутого связного списка приведена в листинге 2.2.

Листинг 2.2. Реализация очереди на основе развернутого связного списка

```
const NODE_SIZE: integer;
structure Node {item: ArrayConcurrentQueue, Node next}

head, tail, unfull: Node;

initialize()
      head = tail = unfull = new Node(NULL, NULL);
```

```
     enqueue(e: E)
e1:        loop
e2:              loop
e3:                    u = unfull;
e4:                    unfullArray = u.getItem();
e5:                    if (unfullArray != NULL)
e6:                          if (unfullArray.enqueue(e))
e7:                                return true;
e8:                    Node next = u.getNext();
e9:                    if (next = null)
e10:                         break;
e11:                   CAS(unfull, u, next);
                 // Failed to insert into existing nodes.
                 // Need to create new one.
e12:             newArray = ArrayConcurrentQueue(NODE_SIZE);
e13:             Node n = new Node(newArray, null);
e14:             loop
e15:                   Node t = tail;
e16:                   Node s = t.getNext();
e17:                   if (t = tail)
e18:                         if (s = null)
e19:                               if (CAS(t.next, s, n))
e20:                                     CAS(tail, t, n);
e21:                                     break;
e22:                         else
e23:                               CAS(tail, t, n);


     dequeue(): E
d1:        loop
d2:              h = head;
d3:              t = tail;
d4:              first = h.getNext();
d5:              if (h = head)
d6:                    if (h = t)
d7:                          if (first = NULL)
d8:                                return NULL;
d9:                          else
d10:                               CAS(tail, t, first);
d11:                   else
d12:                         a = first.getItem();
d13:                         if (a = null)
d14:                               CAS(unfull, h, first);
d15:                               CAS(head, h, first);
d16:                         else
d17:                               e = a.dequeue();
d18:                               if (e != NULL)
d19:                                     return e;
d20:                               else
d21:                                     if (a.isExhausted())
d22:                                           CAS(unfull, h, first);
d23:                                           CAS(head, h, first);
d24:                                     else
d25:                                           return NULL;


     peek(): E
p1:        loop
p2:              h = head;
p3:              t = tail;
p4:              first = h.getNext();
p5:              if (h = head)
p6:                    if (h = t)
p7:                          if (first = NULL)
```

```
p8:                                 return NULL;
p9:                           else
p10:                                CAS(tail, t, first);
p11:                    else
p12:                          a = first.getItem();
p13:                          if (a = null)
p14:                                CAS(unfull, h, first);
p15:                                CAS(head, h, first);
p16:                          else
p17:                                e = a.peek();
p18:                                if (e != NULL)
p19:                                      return e;
p20:                                else
p21:                                      if (a.isExhausted())
p22:                                            CAS(unfull, h, first);
p23:                                            CAS(head, h, first);
p24:                                      else
p25:                                            return NULL;
```

Рассмотрим подробнее операцию `enqueue`. Она состоит из внешнего цикла, внутри которого выполняется две фазы. Первая фаза (e3–e11) – это попытка записать добавляемое значение в уже имеющиеся узлы очереди. Алгоритм считывает указатель `unfull` и пытается записать значение в тот массив, который хранится в соответствующей ячейке. Если ему это не удается сделать, то это значит, что его опередили, и ячейка, на которую указывал указатель `unfull` уже полна. В этом случае он сдвигает указатель и начинает заново. Рано или поздно этот алгоритм либо все-таки вставит новый элемент в очередь, либо дойдет до конца связного списка. В последнем случае он переходит ко второй фазе. Вторая фаза (e12–e23) заключается в добавлении в связный список нового элемента, содержащего новую, пустую очередь на массиве. Она в точности соответствует операции `enqueue` в обычной очереди Майкла-Скотта.

Операции `dequeue` и `peek` находят первый еще не полностью использованный элемент связного списка выполняют соответствующую операцию над очередью на массиве, записанной в этом элементе. Эти два метода полностью идентичны, так как они отличаются лишь той операцией, которую делегируют очереди на массиве (строки d18 и p18 соответственно). По этой причине мы рассмотрим лишь первый из них.

Здесь первый этап, как и в предыдущем случае, очень похож на соответствующую операцию над классической очередью Майкла и Скотта. Алгоритм считывает значения указателей `head`, `tail`, а также значение указателя `head.next` (d2–d4), после чего убеждается в их согласованности между собой (d5). Затем он отдельно обрабатывает случай, когда считанные значения указателей `head` и `tail` совпадают (d6–d10) что возможно лишь в случае, если очередь пуста, или в случае, если значение указателя `tail` устарело. Далее алгоритм (уже в отличие от оригинального алгоритма Майкла и Скотта) смотрит на очередь-массив, записанную в ячейке, на которую указывал `head`, и пытается извлечь из нее элемент и вернуть его в качестве результата (d12–d27). При этом если эта очередь-массив отсутствует или была полностью использована, то он сдвигает указатели `head` и `unfull`. Именно здесь, чтобы отличить случай, когда очередь на массиве просто пуста, и когда она уже использована окончательно, пригождается упомянутый в разд. 2.2.2 метод `isExhausted()`. Стоит заметить, что всегда, когда при помощи операции *CAS* сдвигается указатель `head`, перед этим производится попытка сдвинуть указатель `unfull` (d14–d15 и d23–d24). Это делается для того, чтобы обеспечить выполнение шестого свойства из разд. 2.2.1.

2.3. Анализ свойств алгоритма

В данном разделе будут установлены некоторые свойства описанного алгоритма очереди на развернутом связном списке.

2.3.1. Корректность и отсутствие блокировок

Корректность приведенного алгоритма обеспечивается выполнением семи свойств, описанных в разд. 2.1.1, 2.2.1. Доказательство справедливости первых пяти из них ничем не отличаются от доказательства, рассматриваемого в работе [20].

Шестое свойство также всегда выполняется. Это следует из двух наблюдений. Во-первых, так как всякий раз, когда мы сдвигаем указатель `unfull` вправо, то его новое значение – это не `NULL`, а действительный элемент списка, следовательно, он не может вылезти направо за пределы очереди. Во-вторых, всякий раз, когда мы сдвигаем вправо указатель `head`, мы перед этим делаем попытку также сдвинуть и указатель `unfull`, тем самым исключая возможность того, что `head` обгонит `unfull` и станет правее его.

Седьмое свойство выполняется в силу того, что указатель `unfull` сдвигается вправо лишь в том случае, когда убеждаемся, что очередь того элемента, на который он указывал, заполнена.

Из этих семи свойств следует то, что всякий раз при добавлении или удалении элемента мы обращаемся к нужным ячейкам связного списка.

Корректность реализации очереди на одном массиве едва ли оставляет сомнения, в силу относительной простоты ее структуры. Так как она является упрощенным вариантом очереди на циклическом массиве, описанной в работе [21], то желающие изучить более строгие идеи по поводу ее корректности отсылаются к соответствующей статье.

Для того чтобы убедиться в отсутствии блокировок в данном алгоритме, необходимо внимательно рассмотреть каждый цикл `loop` в листингах 2.1 и 2.2. Легко убедиться в том факте, что если некоторый цикл выполняется более одного раза, то это означает один из двух возможных вариантов:

- значения указателей, которые были прочитаны при выполнении этого цикла, были после этого параллельно изменены другим процессом (о чем можно узнать при сравнении их значений или по неудачному результату выполнения операции *CAS*);
- значения каких-то указателей были старыми и не соответствовали действительности.

Первый случай означает, что, несмотря на то, что нашему процессу придется выполнять часть операций заново, какой-то другой процесс, выполняемый параллельно, уже успел совершить какую-то операцию.

Во втором случае наш алгоритм всякий раз начинает заново выполнять цикл лишь после того, как обновит встретившиеся ему устаревшие значения указателей.

2.3.2. Эффективная работа с памятью

Главным преимуществом описанного алгоритма, по отношению, к алгоритму неблокирующей очереди Майкла и Скотта, является тот факт, что он эффективнее работает с памятью за счет использования на нижнем уровне массивов вместо указателей. Ниже будет показано, как при помощи моделей, описанных в первой главе, формально объяснить это преимущество.

В этом разделе речь пойдет об эффективности использования памяти, поэтому далее алгоритмы будут рассматриваться вне контекста параллельных систем, то есть при наличии всего одного процесса.

Нетрудно убедиться, что при последовательном выполнении время работы операций добавления и удаления одного элемента, составляет $O(1)$ как для очереди Майкла-Скотта, так и для очереди на основе развернутого связного списка. Очевидно, что за это время можно успеть обратиться только к $O(1)$ блокам памяти, а, следовательно, число переносов блоков из памяти в кеш при любых параметрах кеша также не может превышать $O(1)$. Таким образом, для любой одной конкретной операции $q(N, Z, L) = O(1)$. С такой точки зрения оба алгоритма оказываются оптимальными и нечувствительными к размеру кеша, и придумать ничего лучше невозможно.

Однако такой анализ был бы неверным. Процессы, работающие с очередью, обычно обращаются к ней многократно, поэтому хочется как-то оценить *суммарное* число переносов блоков памяти для нескольких операций, выполняющихся подряд.

Обозначим параметр, соответствующий количеству элементов очереди, хранящихся в одном узле списка в новом алгоритме очереди, как K. Рассмотрим ситуацию, когда над очередью выполняется в некотором порядке M операций, из которых M_1 операций добавления элемента и M_2 операций удаления элемента. В случае очереди Майкла и Скотта, каждая из операций добавления выделит новый кусок динамической памяти, размера $O(1)$, для того, чтобы добавить новый узел в связный список, а каждая операция удаления просмотрит один такой кусок памяти. Так как нет никаких дополнительных гарантий по поводу того, как должны располагаться узлы списка в реальной физической памяти, то они могут все оказаться в разных ее местах, и для того, чтобы обойти их все, понадобится

$$O(M_1) + O(M_2) + O(1) = O(M)$$

переносов блоков памяти.

В случае если используется очередь на основе развернутого связного списка с размером блока K, то на каждые M_1 последовательных операций добавления и на каждые M_2 операций удаления нам потребуется посетить, целиком M_1/K и M_2/K отдельных последовательных массивов длины K соответственно.

В случае если $L \leq O(K)$, для последовательного посещения всех ячеек одного такого массива потребуется $O(K/L+1)$ переносов блоков памяти. Это в сумме даст $(M_1/K+M_2/K)\cdot O(K/L+1)+O(1) = M/K\cdot O(K/L+1) = O(M/L+M/K+1)= = O(M/L+1)$ переносов блоков памяти. Очевидно, что данная оценка является оптимальной, так как каждая из M операций обращается к новой ячейке памяти, а, следовательно, каждые $O(L)$ операций не избежать очередного переноса блока из памяти в кеш.

К сожалению, в случае, если $L >> K$, такой оценки достичь не удастся. Таким образом, данный алгоритм не является в полной мере нечувствительным к размеру кеша.

Однако, подобное ограничение сверху на порядок размера одного блока кеша все же лучше, чем строгая оценка порядка размера блока кеша.

Так, например, для алгоритма *Б*-дерева, классического для модели идеального кеша, важно, чтобы порядок размера одного узла дерева K совпадал с порядком параметра L, используемого в кеше. Если взять $K << L$, то различные узлы дерева будут слишком малы, и могут быть слишком разбросаны по памяти, а глубина дерева, равная $O(\log_K N)$ будет слишком большой, так что поиск будет занимать слишком много переносов блоков. Если же взять $K >> L$, то узлы будут слишком большими, и их изменение будет уже занимать больше, чем $O(1)$ операций переноса.

Поэтому, в частности, асимптотический выигрыш от использования *Б*-деревьев теряется, если в системе используется несколько уровней кеш-памяти. В описанном же алгоритме очереди для того, чтобы добиться оптимальной работы с памятью важно знать лишь оценку сверху на возможный размер одного блока памяти в используемой системе.

Описанное выше свойство предложенного алгоритма очереди на основе развернутого связного списка является главной причиной того, что он показывает хорошие результаты на практике.

Выводы по главе 2

В данной главе была приведена реализация нового параллельного алгоритма очереди на основе развернутого связанного списка. Также была доказана его корректность, и были установлены и исследованы некоторые его свойства в модели параллельных вычислений, в частности отсутствие блокировок. Кроме того, был произведен теоретический анализ эффективности работы с памятью данного алгоритма, и был установлен ряд свойств, указывающих, что алгоритм должен показывать хорошую производительность на реальных системах с кешом.

ГЛАВА 3. ПРАКТИЧЕСКОЕ ИЗМЕРЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ПРЕДЛОЖЕННОГО АЛГОРИТМА

3.1. МЕТОДИКА АНАЛИЗА ПРОИЗВОДИТЕЛЬНОСТИ

Реализация описанного во второй главе алгоритма на языке *Java* доступна в приложении.

Анализ производительности алгоритма проводился на персональном компьютере *Sony Vaio* с процессором *Intel Core 2 Duo* 1,60 ГГц в рамках виртуальной машины *Java HotSpot Client VM (Java 2 Runtime Environment 1.6)*, а так же на сервере *Sun Microsystems UltraSPARC-T1* с 32 ядрами, каждое из которых имеет частоту 1 ГГц, в рамках виртуальной машины *Java HotSpot(TM) Server VM (Java 2 Runtime Environment 1.5)*.

Для того чтобы исследовать скорость работы алгоритма был реализован тестирующий модуль. Различные алгоритмы, реализующие очередь, которые подавались ему на вход, последовательно прогонялись на наборе stress-тестов, каждый из которых заключался в том, что несколько потоков параллельно производили над структурой данных различные случайные операции, после чего замерялось среднее время их выполнения. Различные параметры тестирования, такие как количество одновременно работающих потоков, время и число повторений одного теста, а так же средний размер очереди на протяжении тестирования, варьировались, а результаты измерений заносились в таблицы, по которым в дальнейшем строились графики.

Операции, которые выполнялись над тестируемыми реализациями очереди, генерировались случайно. Это делалось заранее для того, чтобы максимально уменьшить число вспомогательных действий, выполняемых потоками непосредственно в процессе тестирования и тем самым повысить точность замеров времени и чистоту экспериментов.

Специальный режим тестирования был реализован для проверки правильности работы алгоритмов. В этом режиме через очередь несколькими потоками прогонялся набор различных уникальных чисел. При этом запоминался порядок, в котором каждый поток добавлял свои значения в очередь, а так же номера и порядок чисел, извлекаемых каждым потоком из очереди. Затем проверялось, что никакие элементы из тех, которые прогонялись через очередь, не были утеряны, или продублированы, а также на основе полученных разными процессами отношениями частичного порядка между элементами в очереди проверялось отсутствие нарушений условия *FIFO*. Этот режим позволял тестировать алгоритмы во время написания на предмет ошибок.

3.2. Графики тестирования производительности

На рис. 3.1, 3.2 изображены графики зависимости числа выполняемых операций над очередью в секунду от числа работающих параллельно процессов для очереди Майкла и Скотта и новой, предложенной в работе очереди на основе развернутого связного списка. Использованный размер ячейки *K* при тестировании – 128Kb, что соответствует размеру кеша второго уровня на 32-ядерном сервере *UltraSPARC-T1*.

Первый график (рис. 3.1) получен на платформе с двуядерным процессором *Intel Core 2 Duo*, а второй (рис. 3.2) – на платформе *Sun Microsystems UltraSPARC-T1* с 32 ядрами.

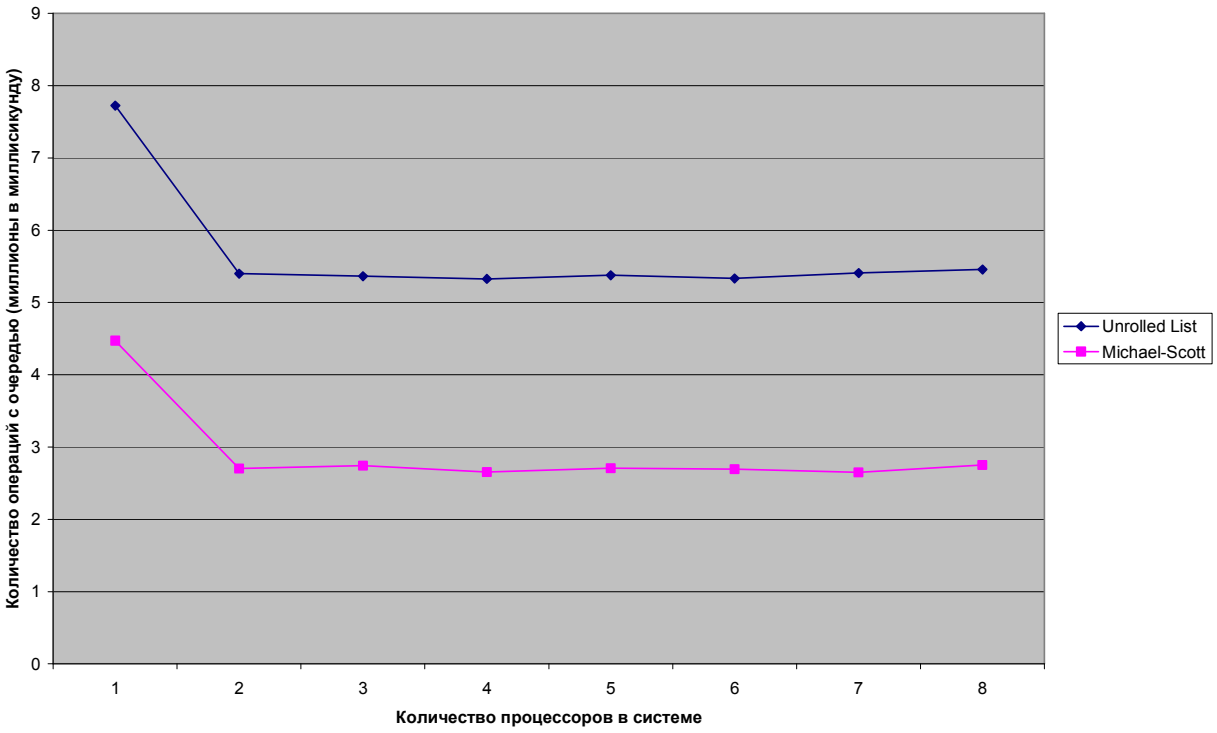

Рис.3.1. График зависимости числа операций производимых в единицу времени над очередью от числа процессов в системе с процессором *Intel Core 2 Duo*

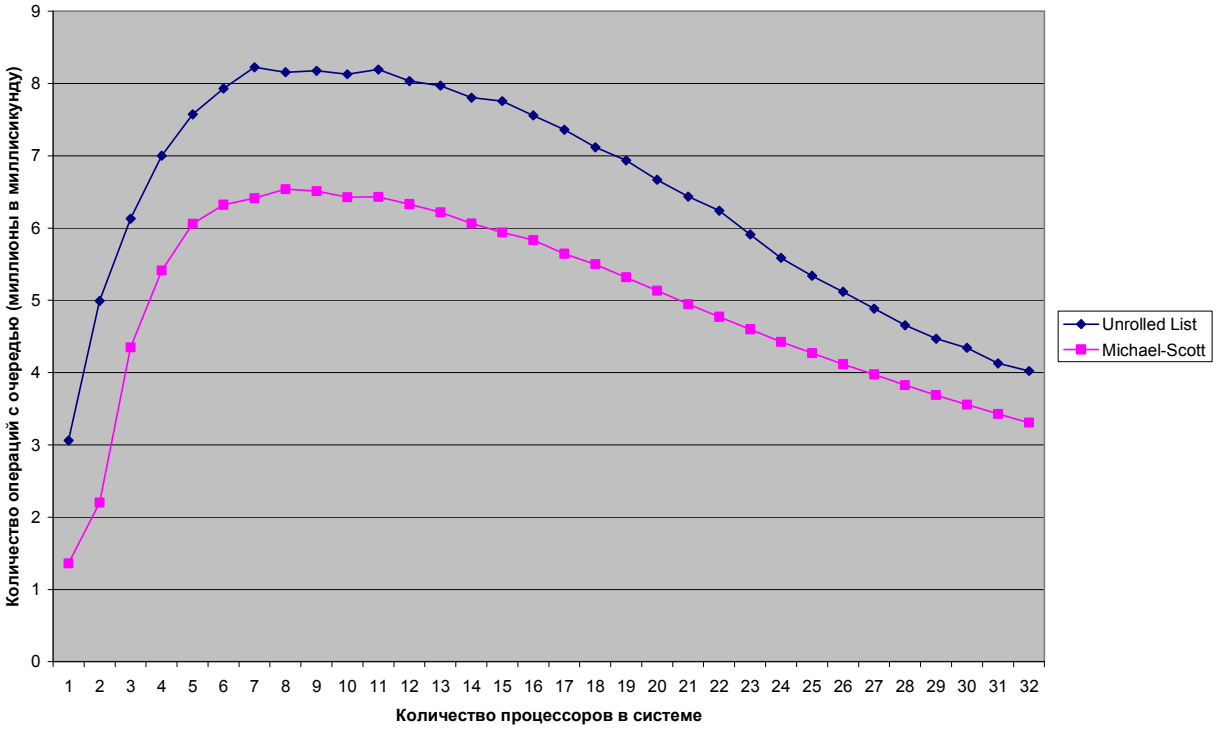

Рис. 3.2. График зависимости числа операций производимых в единицу времени над очередью от числа процессов в системе *Sun Microsystems UltraSPARC-T1*

Как видно из графиков, предложенный алгоритм превосходит по производительности алгоритм очереди Майкла и Скотта в обоих случаях. Самый сильный выигрыш наблюдается в случае, когда параллельная нагрузка слабая или отсутствует (один – два процесса), но и при большей нагрузке новый алгоритм имеет преимущество.

График на рис. 3.2. показывает, что разработанный алгоритм не уступает алгоритму Майкла и Скотта в масштабируемости. Так, при малом числе процессоров производительность возрастает почти линейно по мере увеличения их числа.

Выводы по главе 3

В данной главе был произведен практический анализ производительности описанного алгоритма в сравнении с предыдущими. Была описана применявшаяся методика измерения производительности и сравнения разных параллельных алгоритмов, реализующих очередь. Также были приведены графики производительности, полученные при тестировании на различных вычислительных системах, позволяющие говорить о хорошей масштабируемости и эффективности полученного в данной работе алгоритма.

ЗАКЛЮЧЕНИЕ

В данной работе был произведен обзор различных моделей вычислений, которые в разной степени соответствуют практическим системам, были рассмотрены основные достоинства и недостатки различных из них. Были приведены различные алгоритмы, реализующие поисковые структуры данных и очереди, исследуемые в рамках этих моделей. На основе уже имеющихся алгоритмов был построен новый алгоритм, эффективно реализующий очередь без блокировок на основе структуры, под названием развернутый связный список.

Предложенный алгоритм был также проанализирован в рамках изложенных моделей, что позволило доказать некоторые его полезные свойства. В частности, при его реализации удалось в полной мере добиться наличия у него свойств, связанных с параллельностью (в работе было показано, предложенный алгоритм является алгоритмом без блокировок). Однако при его анализе в рамках модели доступа к памяти, не чувствительной к размеру кеша, было выявлено, что он лишь частично соответствует свойствам алгоритма, не чувствительного к размеру кеша.

Алгоритм, разработанный в рамках данной работы, был протестирован на различных физических системах. Он показал хорошие результаты производительности и, как следствие, имеет широкий спектр задач для практического применения.

ИСТОЧНИКИ

1. *Melzak Z. A.* An informal Arithmetical Approach to Computability and Computation //Canadian Mathematical Bulletin. 1961. Vol. 4, № 3, pp. 279–293.
2. *Belady L. A.* A study of replacement algorithms for virtual storage computers //IBM Systems Journal. 1966. 5(2), pp.78–101.
3. *Sleator D. D., Tarjan R. E.* Amortized efficiency of list update and paging rules //Communications of the ACM. 1985. 28(2), pp. 202–208.
4. *Frigo M., Leiserson C. E., Prokop H., Ramachandran S. Cache-Oblivious* algorithms /In Proc. 40th Annual IEEE Symposium on Foundations of Computer Science. 1999, pp. 285–298.
5. *Aggarwal A., Vitter. J. S.* The input/output complexity of sorting and related problems //Communications of the ACM. 1988. 31(9), pp. 1116–1127.
6. *Bayer R., McCreight E. M.* Organization and maintenance of large ordered indexes //Acta Informatica. 1972. 1(3), pp. 173–189.
7. *Bender M. A., Demaine E., Farach-Colton M. Cache-Oblivious* B-trees. /In FOCS'2000. 2000, pp. 399–409.
8. *Gark V. K.* Concurrent and Distributed Computing in Java //IEEE Press. Wiley Interscience. 2004.
9. *Dijkstra E. W.* Solution of a problem in concurrent programming control //Commun. of the ACM. 1965. 8(9), p. 569.
10. *Lamport L.* A new solution of Dijkstra's concurrent programming program //Commun. of the ACM. 1974. 17(8), pp. 125–137.
11. *Peterson G. L.* Myths about the mutual exclusion problem //Information Processing Letters. 1981. 12(3), pp. 115–116.
12. *Lamport L.* On interprocess communication, part II: algorithms //Distributed computing. 1986. № 1, pp. 86–101.
13. *Herlihy M.* Impossibility and universality results for wait-free synchronization //Technical Report TR-CS-8, Carnegie-Mellon University (Pittsburg PA), May 1988.

14. *Herlihy M.* Wait-Free Synchronization //ACM Transactions on Programming Languages and Systems, 13(1):124–149, 1991.
15. *Herlihy M., Luchangco V., Moir M.* Obstruction-free synchronization: double-ended queues as an example / Proceedings of 23 International Conference on Distributed Computing Systems, 2003.
16. *Kung H. T., Lehman P. L.* Concurrent manipulation of binary search trees // ACM Transactions on Database Systems. 5(3):354–382, 1980.
17. *Lehman P. L., Yao B.* Efficient locking for concurrent operations on B-trees //ACM Transactions on Database Systems. 6(4):650-670, 1981.
18. *Kornacker M., Mophan C., Hellerstein J. M.* Concurrency and recovery in generalized search trees /ACM SIGMOD Record. 26(2):62–72, 1997.
19. *Bender M.* A., Fineman J. T., Gilbert S., Kuszmaul B. C. Concurrent *Cache-Oblivious* B-trees /ACM Symposium on Parallel Algorithms and Architectures, Las-Vegas, Nevada, USA, 2005.
20. *Michael M., Scott M.* Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms /Annual ACM Symposium on Principles of Distributed Computing, 1996.
21. *Tsigas P., Zhang Y.* A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems /ACM Symposium on Parallel Algorithms and Architectures. 2001, pp 134–143.
22. *Ladan-Mozes E., Shavit N.* An optimistic approach to lock-free FIFO queues. /In Proc. of the 18th International Conference on Distributed Computing, 2004, pp. 117–131.
23. *Shao Z., Reppy J. H., Appel A. W.* Unrolling lists /ACM SIGPLAN Lisp Pointers, Volume VII(3). 1994, pp 185–195.
24. *Aho A. V. Hopcroft J. E., Ullman J. D.* Data Structures and Algorithms. Addison-Wesley, 1983.

ПРИЛОЖЕНИЕ. ИСХОДНЫЕ КОДЫ НА ЯЗЫКЕ *JAVA*

ArrayConcurrentQueue.java

```
package ru.ifmo.arraylockfreequeue;

import java.util.*;
import java.util.concurrent.atomic.AtomicReferenceArray;

/**
 * @author Iskander Akishev
 */
public class ArrayConcurrentQueue<E> extends AbstractQueue<E> {

    private final int N;
    private final AtomicReferenceArray a;
    private int head, tail;

    private final static Object NULL_UNPROCESSED = new Object();
    private final static Object NULL_PROCESSED = new Object();

    public ArrayConcurrentQueue(int length) {
        this.N = length;
        this.a = new AtomicReferenceArray(length);
        for (int i = 0; i < N; i++) {
            a.set(i, NULL_UNPROCESSED);
        }
        head = tail = 0;
    }

    public int size() {
        while (true) {
            int h = head;
            if (h == N) {
                return 0;
            }
            Object hv = a.get(h);
            if (hv == NULL_PROCESSED) {
                head = ++h;
                continue;
            }

            int t = tail;
            Object tv;
            if (t == N) {
                tv = NULL_UNPROCESSED;
            } else {
                tv = a.get(t);
            }
            if (tv != NULL_UNPROCESSED) {
                tail = t + 1;
                continue;
            }
            return t - h;
        }
    }

    public boolean offer(E e) {
        if (e == null) {
```

```
            throw new NullPointerException();
        }
        while (true) {
            int t = tail;
            if (t == N) {
                return false;
            }
            while (a.get(t) != NULL_UNPROCESSED) {
                t++;
                if (t == N) {
                    tail = N;
                    return false;
                }
            }
            if (a.compareAndSet(t, NULL_UNPROCESSED, e)) {
                tail = t + 1;
                return true;
            }
        }
    }

    public E poll() {
        int h = head;
        if (h == N) {
            return null;
        }
        while (true) {
            Object v = a.get(h);
            if (v == NULL_PROCESSED) {
                head = ++h;
                if (h == N) {
                    return null;
                }
            } else if (v == NULL_UNPROCESSED) {
                return null;
            } else {
                if (a.compareAndSet(h, v, NULL_PROCESSED)) {
                    head = h + 1;
                    return (E) v;
                }
            }
        }
    }

    public E peek() {
        int h = head;
        if (h == N) {
            return null;
        }
        while (true) {
            Object v = a.get(h);
            if (v == NULL_PROCESSED) {
                head = ++h;
                if (h == N) {
                    return null;
                }
            } else if (v == NULL_UNPROCESSED) {
                return null;
            } else {
                return (E) v;
            }
        }
    }
```

```
    public boolean isExhausted() {
        return (a.get(N - 1) == NULL_PROCESSED);
    }

    public Iterator<E> iterator() {
        throw new UnsupportedOperationException("Iterator is not supported");
    }

}
```

Node.java

```
package ru.ifmo.arraylockfreequeue;

import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 * @author Iskander Akishev
 */
class Node<E> {
    private volatile E item;
    private volatile Node<E> next;

    private static final
    AtomicReferenceFieldUpdater<Node, Node>
        nextUpdater =
        AtomicReferenceFieldUpdater.newUpdater
        (Node.class, Node.class, "next");
    private static final
        AtomicReferenceFieldUpdater<Node, Object>
        itemUpdater =
        AtomicReferenceFieldUpdater.newUpdater
        (Node.class, Object.class, "item");

    Node(E x) { item = x; }

    Node(E x, Node<E> n) { item = x; next = n; }

    E getItem() {
        return item;
    }

    boolean casItem(E cmp, E val) {
        return itemUpdater.compareAndSet(this, cmp, val);
    }

    void setItem(E val) {
        itemUpdater.set(this, val);
    }

    Node<E> getNext() {
        return next;
    }

    boolean casNext(Node<E> cmp, Node<E> val) {
        return nextUpdater.compareAndSet(this, cmp, val);
    }

    void setNext(Node<E> val) {
        nextUpdater.set(this, val);
    }
```

```
}
```

UnrolledListConcurrentQueue.java

```
package ru.ifmo.arraylockfreequeue;

import java.util.*;
import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 * Lock-free concurrent queue implementation, based on unrolled linked list.
 *
 * @author Iskander Akishev
 */
public class UnrolledListConcurrentQueue<E> extends AbstractQueue<E>
implements Queue<E> {

    private final int NODE_SIZE;

    public UnrolledListConcurrentQueue(int nodeSize) {
        this.NODE_SIZE = nodeSize;
    }

    private static final
        AtomicReferenceFieldUpdater<UnrolledListConcurrentQueue, Node>
        tailUpdater =
        AtomicReferenceFieldUpdater.newUpdater
        (UnrolledListConcurrentQueue.class, Node.class, "tail");
    private static final
        AtomicReferenceFieldUpdater<UnrolledListConcurrentQueue, Node>
        headUpdater =
        AtomicReferenceFieldUpdater.newUpdater
        (UnrolledListConcurrentQueue.class,  Node.class, "head");
    private static final
        AtomicReferenceFieldUpdater<UnrolledListConcurrentQueue, Node>
        unfullUpdater =
        AtomicReferenceFieldUpdater.newUpdater
        (UnrolledListConcurrentQueue.class,  Node.class, "unfull");

    private boolean casTail(Node<ArrayConcurrentQueue<E>> cmp,
Node<ArrayConcurrentQueue<E>> val) {
        return tailUpdater.compareAndSet(this, cmp, val);
    }
    private boolean casHead(Node<ArrayConcurrentQueue<E>> cmp,
Node<ArrayConcurrentQueue<E>> val) {
        return headUpdater.compareAndSet(this, cmp, val);
    }
    private boolean casUnfull(Node<ArrayConcurrentQueue<E>> cmp,
Node<ArrayConcurrentQueue<E>> val) {
        return unfullUpdater.compareAndSet(this, cmp, val);
    }

    private volatile Node<ArrayConcurrentQueue<E>> head = new
Node<ArrayConcurrentQueue<E>>(null, null);
    private volatile Node<ArrayConcurrentQueue<E>> tail = head;
    private volatile Node<ArrayConcurrentQueue<E>> unfull = head;


    public int size() {
        int count = 0;
```

```
        for (Node<ArrayConcurrentQueue<E>> p = getFirst(); p != null; p =
p.getNext()) {
            ArrayConcurrentQueue<E> a = p.getItem();
            if (a != null) {
                count += a.size();
                if (count < 0) {
                    return Integer.MAX_VALUE;
                }
            }
        }
        return count;
    }

    public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        while (true) {
            while (true) {
                Node<ArrayConcurrentQueue<E>> u = unfull;
                ArrayConcurrentQueue<E> unfullArray = u.getItem();
                if (unfullArray != null) {
                    if (unfullArray.offer(e)) {
                        return true;
                    }
                }
                Node<ArrayConcurrentQueue<E>> next = u.getNext();
                if (next == null) {
                    break;
                }
                casUnfull(u, next);
            }

            // Failed to insert into existing nodes. Need to create new one.
            Node<ArrayConcurrentQueue<E>> n = new
Node<ArrayConcurrentQueue<E>>(new ArrayConcurrentQueue<E>(NODE_SIZE), null);
            while (true) {
                Node<ArrayConcurrentQueue<E>> t = tail;
                Node<ArrayConcurrentQueue<E>> s = t.getNext();
                if (t == tail) {
                    if (s == null) {
                        if (t.casNext(s, n)) {
                            casTail(t, n);
                            break;
                        }
                    } else {
                        casTail(t, s);
                    }
                }
            }
        }
    }

    Node<ArrayConcurrentQueue<E>> getFirst() {
        while (true) {
            Node<ArrayConcurrentQueue<E>> h = head;
            Node<ArrayConcurrentQueue<E>> t = tail;
            Node<ArrayConcurrentQueue<E>> first = h.getNext();
            if (h == head) {
                if (h == t) {
                    if (first == null)
                        return null;
                    else
                        casTail(t, first);
                } else {
                    if (first.getItem() != null)
```

```
                    return first;
                else // remove deleted node and continue
                    casUnfull(h, first);
                    casHead(h, first);
            }
        }
    }
}

public E poll() {
    while (true) {
        Node<ArrayConcurrentQueue<E>> h = head;
        Node<ArrayConcurrentQueue<E>> t = tail;
        Node<ArrayConcurrentQueue<E>> first = h.getNext();
        if (h == head) {
            if (h == t) {
                if (first == null)
                    return null;
                else
                    casTail(t, first);
            } else {
                ArrayConcurrentQueue<E> a = first.getItem();
                if (a == null) {
                    casUnfull(h, first);
                    casHead(h, first);
                } else {
                    E item = a.poll();
                    if (item != null) {
                        return item;
                    } else {
                        if (a.isExhausted()) {
                            casUnfull(h, first);
                            casHead(h, first);
                        } else {
                            return null;
                        }
                    }
                }
            }
        }
    }
}

public E peek() {
    while (true) {
        Node<ArrayConcurrentQueue<E>> h = head;
        Node<ArrayConcurrentQueue<E>> t = tail;
        Node<ArrayConcurrentQueue<E>> first = h.getNext();
        if (h == head) {
            if (h == t) {
                if (first == null)
                    return null;
                else
                    casTail(t, first);
            } else {
                ArrayConcurrentQueue<E> a = first.getItem();
                if (a == null) {
                    casUnfull(h, first);
                    casHead(h, first);
                } else {
                    E item = a.peek();
                    if (item != null) {
                        return item;
                    } else {
```

```
                                if (a.isExhausted()) {
                                    casUnfull(h, first);
                                    casHead(h, first);
                                } else {
                                    return null;
                                }
                            }
                        }
                    }
                }
            }
        }

        public void clear() {
            head = tail = unfull = new Node<ArrayConcurrentQueue<E>>(null, null);
        }

        public Iterator<E> iterator() {
            throw new UnsupportedOperationException("Iterator is not supported");
        }

    }
```