МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Параллельные алгоритмы»

Тема: Реализация структур данных без блокировок

Студент гр. 0304 _____ Люлин Д.В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург

2023

Цель работы.

Изучить принципы построения потокобезопасных структур данных без блокировок.

Здание.

Выполняется на основе работы 2.

Реализовать очередь, удовлетворяющую lock-free гарантии прогресса. Протестировать доступ к реализованной структуре данных в случае нескольких потоков производителей и потребителей.

В отчёте: Сравнить производительность с реализациями структур данных из работы 2. В отчёте сформулировать инвариант структуры данных.

Выполнение работы.

В работе была реализована lock-free очередь Майкла-Скотта, удовлетворяющая гарантии прогресса. Для очереди был написан шаблонный класс *LockFreeQueue*. Исходный код приведён в приложении А.

Для безопасного освобождения памяти была использована схема указателей опасности (hazard pointers).

Принцип работы очереди.

Очередь построена на односвязном списке, в котором элементы добавляются в конец и удаляются из начала. В методах добавления и удаления (*push()* и *pop()*, соответственно) существуют бесконечные циклы с условием *while(true)*. В циклах происходят попытки атомарной замены указателей на элементы списка с помощью операции CAS, которая в C++ реализована функциями *compare_exchange_weak* и *compare_exchange_strong*. Если сторонний поток помешал текущему потоку осуществить свою функцию, то цикл повторяется до тех пор, пока операция вставки или удаления не будет выполнена.

В начале очереди всегда находится фиктивный элемент, чтобы указатели на начало и конец списка не были нелувыми.

Если в цикле только проверять выполнимость операции, то такой алгоритм не будет удовлетворять гарантии lock-free прогресса, потому что операции будут зависеть от успеха операций других потоков. Эти потоки могут быть приостановлены планировщиком на неопределённое время. Таким образом, гарантия будет нарушена. Чтобы она соблюдалась, нужно заканчивать операции за другими потоками, если обнаружено промежуточное состояние очереди.

Например, если во время операции *push()* будет установлено, что указатель *tail_* на конец очереди содержит следующий элемент (*tail_->next != nullptr*), то в том же потоке будет попытка передвинуть указатель *tail_* (то есть, *tail_ = tail_->next*, но с помощью CAS), не дожидаясь, когда это сделает другой поток. Таким образом, даже если другие потоки будут приостановлены, то текущий поток сможет беспрепятственно продолжить свою работу.

Аналогично настраивается указатель *head_*: если замечено, что он изменился и другой поток произвёл удаление, то операция повторяется с корректным значением *head_*.

Можно утверждать, что гарантия lock-free прогресса соблюдается. Также размеры всех атомарных переменных (указателей) не превышают размеров машинного слова, поэтому операции с ними свободны от блокировок (метод *is_lock_free* всегда возвращает *true*).

Учитывая приведённые факты, в некоторые промежутки времени указатели *head_* и *tail_* могут указывать не на фактические начало и конец очереди. Инвариант можно сформулировать так: указатель на начало очереди *head_* всегда указывает на элемент, стоящий не после элемента под указателем на конец очереди *tail_*. При этом, они могут указывать на один и тот же (фиктивный) элемент, например, когда очередь пуста или добавление первого элемента не завершилось.

Безопасное освобождение памяти (SMR).

Если удалять элементы сразу после извлечения из очереди, может произойти повреждение памяти и другие виды неопределённого поведения. Это

связано с тем, что другие потоки в это время используют освобождаемую память, разыменовываясь по своей локальной копии указателя. Поэтому память должна освобождаться в момент времени, когда ни один поток не владеет указателем.

Существует множество алгоритмов безопасного освобождения памяти: tagged pointers, epoch-based reclamation, hazard pointers, read-copy-update и другие. Для данной работы был выбран алгоритм hazard pointers, потому что является одним из наиболее распространённых алгоритмов.

Суть алгоритма заключается в том, что для каждого потока выделяется H ячеек под указатели опасности. В эти ячейки потоки записывают указатели, которыми пользуются в данный момент. Число H выбирается в зависимости от структуры данных, для очереди достаточно $H = 2$. Число потоков T должно быть известно заранее.

Каждый поток также хранит массив указателей для отложенного удаления. При извлечении элементов из очереди потоки добавляют извлечённые элементы в свои массивы. Когда массив заполняется, вызывается процедура освобождения памяти. При этом, освобождаются только те указатели, которые не объявлены указателями опасности ни в одном из потоков. Размер такого массива для каждого потока должен превышать общее количество всех указателей опасности $H*T$, чтобы каждое освобождение памяти удаляло хотя бы один указатель. В реализации очереди размер был принят равным $2*H*T$.

Данный метод освобождает память только при заполнении массива, поэтому вся память также освобождается в деструкторе во избежание утечек.

Сравнение реализаций многопоточной очереди.

В табл. 1-3 приведено сравнение многопоточных очередей с «грубыми» и «тонкими» блокировками, а также без блокировок. При измерениях очередь обрабатывала 600 задач по умножению матриц 10x10. Во всех случаях бралось среднее значение времени работы из 1000 запусков программы.

Таблица 1. Сравнение очередей при 5 потребителях и 5 производителях.

Очередь	User time, мс	System time, мс	Real time, мс
«Грубые» блокировки	34.081	9.264	27.745
«Тонкие» блокировки	34.040	9.314	28.254
Lock-free	53.407	10.118	33.799

В табл. 2 приведено сравнение очередей при 8 потребителях и 2 производителях.

Таблица 2. Сравнение очередей при 8 потребителях и 2 производителях.

Очередь	User time, мс	System time, мс	Real time, мс
«Грубые» блокировки	31.782	8.827	27.232
«Тонкие» блокировки	31.572	7.854	28.255
Lock-free	73.625	6.199	38.664

В табл. 3 приведено сравнение очередей при 2 потребителях и 8 производителях.

Таблица 3. Сравнение очередей при 2 потребителях и 8 производителях.

Очередь	User time, мс	System time, мс	Real time, мс
«Грубые» блокировки	33.266	10.973	27.310
«Тонкие» блокировки	33.069	10.305	27.460
Lock-free	42.299	10.721	30.054

Видно, что операции с lock-free очередью занимают больше времени, чем операции с очередями с блокировками, особенно, когда потребителей больше,

чем производителей (табл. 2) и потребители простаивают в цикле без блокировок. User time превышает аналоги более, чем в 2 раза, потому что не делаются блокировки, и большая часть программы выполняется в пространстве пользователя. Однако system time не уменьшилось, из-за чего real time превышает время аналогов.

Причина падения производительности заключается в частых выделениях и освобождениях динамической памяти. В lock-free очереди выделений памяти происходит больше, чем в очередях с блокировками, потому что выделяется память не только для элементов очереди, но и для работы алгоритма указателей опасности.

Выделение и освобождение памяти в системе происходит с помощью захвата мьютекса, поэтому такие операции приводят к существенной потере производительности. Использование атомарных операций влечёт за собой сброс кэша в оперативную память, что замедляет работу программы. Также большая часть user time была потрачена на ожидание поступления элементов в очередь, в отличие от очередей с блокировками, где ожидание выполнялось в условной переменной.

Выводы.

Были изучены принципы построения потокобезопасной очереди без блокировок. Был сформулирован инвариант lock-free очереди: указатель на начало очереди *head_* всегда указывает на элемент, стоящий не после элемента под указателем на конец очереди *tail_*.

Было установлено, что lock-free очередь позволяет выполнять больше параллельных операций, чем очереди с блокировками, но это преимущество не даёт эффекта. Очередь показывает худшую производительность из-за большего, чем у очередей с блокировками, количества выделений и освобождений динамической памяти, а также частой синхронизации кэша с памятью из-за атомарных операций. Параллельные потоки в данном случае замедляли друг друга.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Исходный код программы доступен в репозитории https://github.com/Astana-Mirza/parallel_algo/tree/master.