Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Курсовая работа
по курсу «Операционные системы»

Реализация lock-free очередь. Сравнение с многопоточными версиями очереди, реализованными с помощью стандартного примитива блокировки мьютекс.

Студент:	Марков А.Н.
Группа:	М80-208Б-18
Преподаватель:	Миронов Е.С.
Оценка:	
Дата:	

Москва
2020

Содержание

Постановка задачи

Реализовать lock-free очередь. Сравнить с многопоточными версиями очереди, реализованными с помощью стандартного примитива блокировки мьютекс.

Общие сведения о программе

Программа состоит из 6 файлов:

- main.cpp — основной исполняемый файл.
- lock_free_queue.hpp — файл с реализацией lock-free очереди.
- lock_queue.hpp — файл с реализацией очереди с mutex.
- timer.hpp — вспомогательный файл с реализацией таймера для замера времени работы программы.
- sort.hpp — вспомогательный файл с реализацией пирамидальной сортировки, необходимой для lock-free очереди.
- binary_search.hpp — вспомогательный файл с реализацией бинарного поиска, необходимого для lock-free очереди.

Общий метод и алгоритм решения

Неблокирующая синхронизация — это подход в параллельном программировании, в котором принят отказ от традиционных примитивов блокировки, таких, как семафоры, мьютексы.

Причина появления подобных алгоритмов — традиционный подход позволяет предоставить последовательный доступ при помощи такого механизма синхронизации, как блокировки, при таком подходе попытка одного из потоков получить блокировку, которая уже занята другим потоком, приводит к приостановке выполнения первого потока до момента освобождения блокировки во втором потоке, такая схема может привести к приостановке потоков на неопределенное время. Неблокирующие алгоритмы гарантируют, что такие остановки одного потока не приведут к простою остальных.

За основу реализации lock-free очереди была взята схема Майкла-Скотта с помощью Hazard Pointer. Использование hazard pointer обусловлено появлением ABA проблемы, которая возникает, когда ячейка памяти читается дважды, оба раза прочитано одинаковое значение, и признак «значение одинаковое» трактуется как «ничего не менялось». Однако, другой поток может выполниться между этими двумя чтениями, поменять значение, сделать что-нибудь еще и восстановить старое значение. Таким образом, первый поток обманется, считая, что не поменялось ничего, хотя второй поток уже разрушил это предположение.

Суть схемы hazard pointer заключается в обязанности объявлять указатель на lock-free контейнера как hazard внутри операции lock-free структуры данных, то есть прежде чем работать с указателем на элемент, мы обязаны поместить его в массив hazard pointer`ов текущего потока. Такой массив является приватным для потока: пишет в него только поток-владелец, читать могут все потоки. При удалении элемента lock-free контейнера поток помещает элемент в локальный список отложенных элементов. Как только размер списка достигает некоторого значения, вызывается процедура Scan(), которая и занимается удалением отложенных элементов.

Процедура Scan() состоит из четырех стадий:

- Сначала готовится массив plist текущих hazard pointers, в который включаются все отличные от nullptr hazard pointer всех потоков.
- Стадия 2 сортирует массив plist, чтобы оптимизировать последующий поиск.
- Стадия 3. Просматриваются все элементы массива dlist текущего потока. Если элемент dlist[i] находится в plist (то есть какой-то поток работает с этим указателем, объявив его как hazard pointer), его удалять нельзя и он остается в dlist (переносится в new_dlist). Иначе элемент dlist[i] может быть удален — ни один поток не работает с ним.
- Стадия 4 копирует неудаленные элементы из new_dlist в dlist.

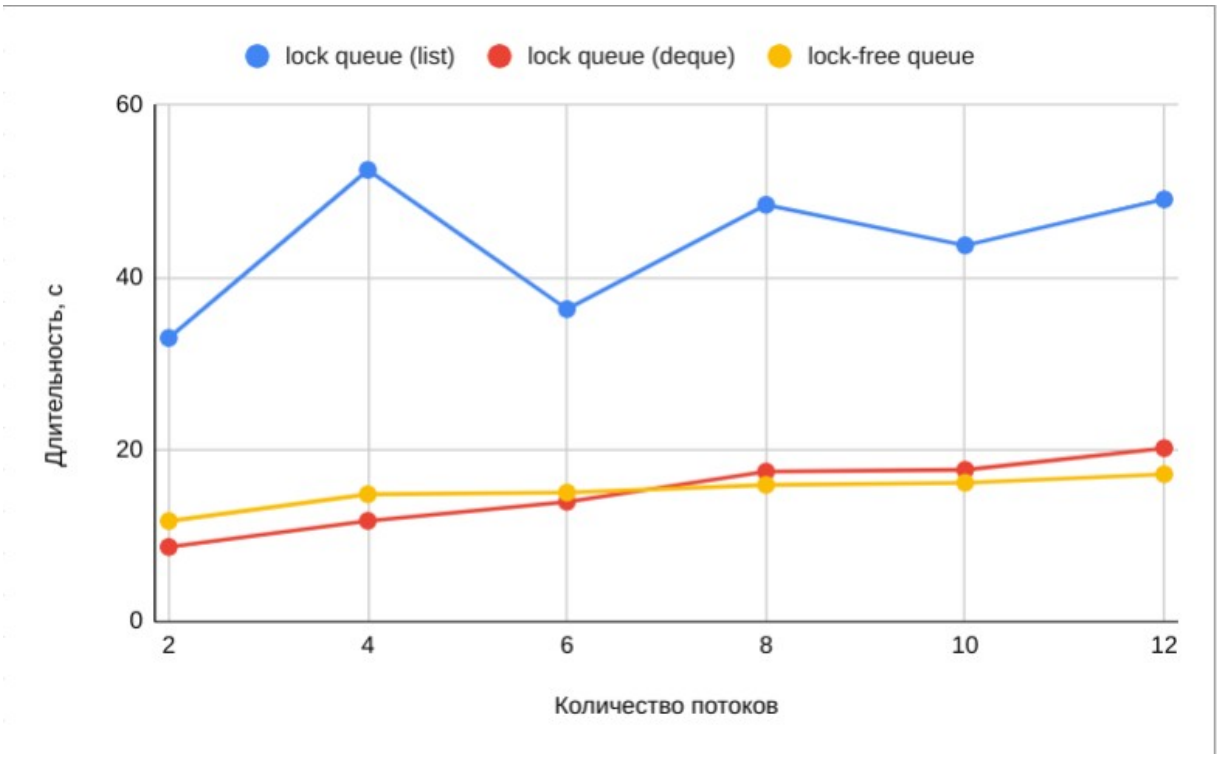
Демонстрация работы программы

```
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 2
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 11.6852 s
LOCK QUEUE DEQUE Time 8.67076 s
LOCK QUEUE LIST Time 32.9613 s
alex@pc:~/Documents/OS/kp/src$ g++ -o main main.cpp -lpthread
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 4
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 14.7997 s
LOCK QUEUE DEQUE Time 11.7077 s
LOCK QUEUE LIST Time 52.4895 s
alex@pc:~/Documents/OS/kp/src$ g++ -o main main.cpp -lpthread
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 6
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 14.9949 s
LOCK QUEUE DEQUE Time 13.9265 s
LOCK QUEUE LIST Time 36.3071 s
alex@pc:~/Documents/OS/kp/src$ g++ -o main main.cpp -lpthread
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 8
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 15.8839 s
LOCK QUEUE DEQUE Time 17.4339 s
LOCK QUEUE LIST Time 48.4348 s
alex@pc:~/Documents/OS/kp/src$ g++ -o main main.cpp -lpthread
```

```
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 10
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 16.1358 s
LOCK QUEUE DEQUE Time 17.6354 s
LOCK QUEUE LIST Time 43.7285 s
alex@pc:~/Documents/OS/kp/src$ g++ -o main main.cpp -lpthread
alex@pc:~/Documents/OS/kp/src$ ./main
CNT OF THREADS: 12
CNT PUSH: 50000000
CNT POP: 12500000
LOCK FREE QUEUE Time 17.1324 s
LOCK QUEUE DEQUE Time 20.1541 s
LOCK QUEUE LIST Time 49.0715 s
```

Сравнение времени работы lock-free очереди и очереди, реализованного с помощью мьютекса

Тест состоит из 50.000.000 вставок в очередь и 12.500.000 удалений.



Из графика видно, что lock-free очередь быстрее, чем очередь, реализованная на списке, с мьютексами и примерно одинаково работает по скорости с очередью, основанной на деке, с мьютексами.

Выводы

Была реализована классическая версия lock-free очереди Майкла-Скотта, представляющая из себя список элементов. По результатам теста можно сделать вывод, что lock-free очередь, которая реализована на списке, быстрее, чем

очередь с блокировками. Примерное равенство по скорости выполнения lock-free очереди и очереди с блокировками, основанной на деке, обуславливается тем, что сама по себе структура дек быстрее выполняет вставку и удаление, чем список