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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Параллельные алгоритмы»
Тема: Реализация потокобезопасных структур данных с блокировками

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2024

Цель работы.

Изучить принцип построения потокобезопасных структур данных с блокировками.

Задание.

Реализовать итерационное (потенциально бесконечное) выполнение подготовки, обработки и вывода данных по шаблону “производитель-потребитель” (на основе лаб. 1 (части 1.2.1 и 1.2.2)).

Количество производителей и потребителей должно быть изменяемым.

Обеспечить параллельное выполнение потоков обработки готовой порции данных, подготовки следующей порции данных и вывода предыдущих полученных результатов.

Использовать механизм “условных переменных”.

2.1

Использовать очередь с “грубой” блокировкой.

2.2

Использовать очередь с “тонкой” блокировкой

Выполнение работы.

В основных файла прописана работа типа производитель-потребитель: производитель производит задачу и добавляет ее в очередь. Потребитель извлекает из очереди задачу и выполняет ее. В данной работе в качестве задачи производителя выступает генерация двух матриц, в качестве задачи потребителя — их умножение.

Очередь с “Грубой” блокировкой.

Для очереди с «грубой» блокировкой был написан шаблонный класс Queue, с методами push, pop и setEnd, которые добавляют и удаляют элемент из очереди соответственно, а также метод, устанавливающий флаг о том, что очередь пуста и больше элементов в нее добавлено не будет.

Для выполнения операции добавления и извлечения мьютекс блокируется. Операция добавления push сигнализирует условной переменной о том, что очередь не пуста. Операция !queue.empty() проверяет, пуста ли очередь. Если очередь пуста, то выполнение блокируется до момента сигнализирования условной переменной. Когда очередь не пуста, элемент извлекается из очереди.

```
1. class Queue {
2. private:
3.     std::queue<T> queue;
4.     std::mutex mutex;
5.     std::condition_variable in;
6.  std::condition_variable out;
7.
8. public:
9.     void push(const T& data) {
10.        std::unique_lock<std::mutex> guard(mutex);
11.        in.wait(guard, [&] {
12.            return queue.size() < QUEUE_SIZE;
13.        });
14.        queue.push(data);
15.        out.notify_one();
16.    }
17.
18.    bool pop(T& data) {
19.        std::unique_lock<std::mutex> guard(mutex);
20.        out.wait(guard, [this](){
21.            return !queue.empty();
22.        });
23.        if (!queue.empty()) {
24.            data = queue.front();
25.            queue.pop();
26.   in.notify_one();
27.            return true;
28.        }
29.        return false;
30.    }
31.};
```

Очередь с “Тонкой” блокировкой.

Для реализации очереди с «тонкой» блокировкой был написан класс QueueFine, который представляет собой односвязный список. В отличии от предыдущего класса заключается в том, что здесь есть два мьютекса, один на начало (блокируется при удалении), другой на конец очереди (при добавлении).

Выполнение операции добавления элемента требует блокировки мьютекса хвоста и изменения значения текущего хвостового узла и привязывания нового хвостового узла, после чего следует сигнализации условной переменной. Выполнение операции извлечения требует больших усилий: для работы условной переменной блокируется мьютекс головы очереди, после чего ожидается условная переменная, в предикате которой проверяется пустота списка (данная операция требует блокировки еще и мьютекса хвоста). Когда очередь не пуста, данные извлекаются, причем мьютекс головы до конца извлечения остается заблокированным.

```
1. class List {
2. private:
3.     struct Node {
4.         T data;
5.         std::unique_ptr<Node> next;
6.     };
7.
8.     std::unique_ptr<Node> head;
9.     Node *tail;
10.    std::mutex mutexHead;
11.    std::mutex mutexTail;
12.    std::condition_variable out;
13.    std::condition_variable in;
14.    int size = 0;
15.
16.    Node *getLockedTail() {
17.        std::unique_lock<std::mutex> guard(mutexTail);
18.        return tail;
19.    }
20.
21.public:
22.    List(): head{std::make_unique<Node>()}, tail{head.get()} {}
23.
24.    void push(const T& data) {
25.        std::unique_lock<std::mutex> guard(mutexTail);
26.        in.wait(guard, [this]() {
27.            return size < QUEUE_SIZE;
28.        });
29.
30.        tail->data = data;
31.        tail->next = std::make_unique<Node>();
32.        tail = tail->next.get();
33.        size++;
34.
35.        out.notify_one();
36.    }
```

```
37.
38.     bool pop(T& data) {
39.         std::unique_lock<std::mutex> guard(mutexHead);
40.         out.wait(guard, [this](){
41.             return head.get() != getLockedTail();
42.         });
43.         if (head.get() != getLockedTail()) {
44.             data = head->data;
45.             head = std::move(head->next);
46.             size--;
47.             in.notify_one();
48.             return true;
49.         }
50.         return false;
51.     }
52. };
```

Сравнение потокобезопасных очередей с блокировками.

Сравнение очередей осуществлялось при помощи измерений обработки очереди 300 задач по умножению матриц 100x100, очередь 10.

Время выполнения программы с «грубыми» блокировками.

Производители	**Потребители**	**Real Time, сек**	**Sys Time, сек**
2	2	5.384	0.189
12	12	2.661	0.478
16	1	10.089	0.732
1	16	2.565	0.159
500	500	3.788	9.666
500	10	3.221	9.686
10	500	2.845	1.278

Время выполнения программы с «тонкими» блокировками.

Производители	**Потребители**	**Real Time, сек**	**Sys Time, сек**
2	2	5.541	0.101
12	12	2.678	0.388
16	1	10.491	0.546
1	16	2.573	0.101
500	500	3.630	8.423
500	10	3.076	7.859

10	500	2.997	0.981

По таблицам видно, что в ряде случаев реализация с тонкой блокировкой по времени почти совпадает с грубой.

При малом количестве потоков, когда потоки меньше конкурируют за мьютекс, реализация с грубой блокировкой оказывается быстрей.

При большем количестве потоков борьба за мьютекс возрастает, но она все ещё невелика, поэтому реализации с грубыми и тонкими блокировками работают примерно за равное время, но тонкая быстрее.

Также можно заметить, что при обеих блокировках случай, когда производителей большей, чем потребителей, работает дольше, чем наоборот. Это объясняется тем, что данные быстрее поступают в очередь, чем могут быть обработаны и извлечены потребителями.

Также по результатам, представленным в таблицах, видно, что реализация с «тонкими» блокировками использует меньше системного времени (Sys. Time), что означает, что время «сна» потоков и ожидания разблокировок мьютексов в ней меньше.

Выводы.

В ходе работы были изучены потокобезопасные очереди с блокировками. Для этого были использовано два вида блокировок - «грубая» и «тонкая».

Было выявлено, что «тонкая» и «грубая» блокировка затрачивают примерно одинаковое время, но первая опережает в случае увеличения потоков.

Также каждая из блокировок плохо работает в случае, когда производителей больше, чем потребителей.

Системное время реализации с «тонкими» блокировками было меньше системного времени реализации с «грубыми» блокировками, что свидетельствует о меньшем количестве блокировок в реализации с «тонкими» блокировками.