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. 0303 _____ Морозов А.Ю.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург

2023

Цель работы.

Изучение способов реализации структур данных без блокировок.

Задание.

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, сформулировать инвариант структуры данных.

Выполнение работы.

В ходе выполнения лабораторной работы был реализован шаблонный классы *LockFreeQueue,* представляющий собой потокобезопасную очередь без блокировок.

Очередь имеет структуру односвязного списка, в котором добавление элемента происходит в конец, а изъятие производится из начала. Она состоит из узлов типа *Node* и имеет один узел по умолчанию, для предотвращения пустоты списка.

Очередь без блокировок имеет методы добавления элемента *add(T element)* и извлечения элемента *bool get(T &element).*

Операция извлечения элемента из очереди представлена бесконечным циклом, в котором происходит чтение текущей головы и хвоста очереди, а также элемента, следующего за головой узла. Следующим шагом в цикле проверяется возможность пустоты очереди: если указатели на голову и хвост совпадают, возможны два исхода:

1) Следующего за головой элемента нет – очередь пуста, а, следовательно, извлечение элемента невозможно.
2) Следующий за головой элемент есть, а значит поток, выполняющий добавление элемента в очередь, не успел переставить указатель хвоста.

Во втором случае необходимо помочь ему и передвинуть этот указатель самому. Если указатели на голову и хвост не совпадают, это гарантирует наличие элементов в очереди. Производится попытка сместить указатель головы, в случае если она не изменилась. При успехе можно доставать значение узла.

Операция добавления элемента в очередь тоже представлена бесконечным циклом. Производится создание нового узла с нужным значением, а затем, в цикле производится чтение хвоста списка, и попытка записать новый узел в конец списка. Если элемента за текущим хвостом нет, спокойно вставляем его в конец списка. Далее пытаемся сместить указатель на хвост. При успехе мы сместили его сами, при неудаче поможет другой поток.

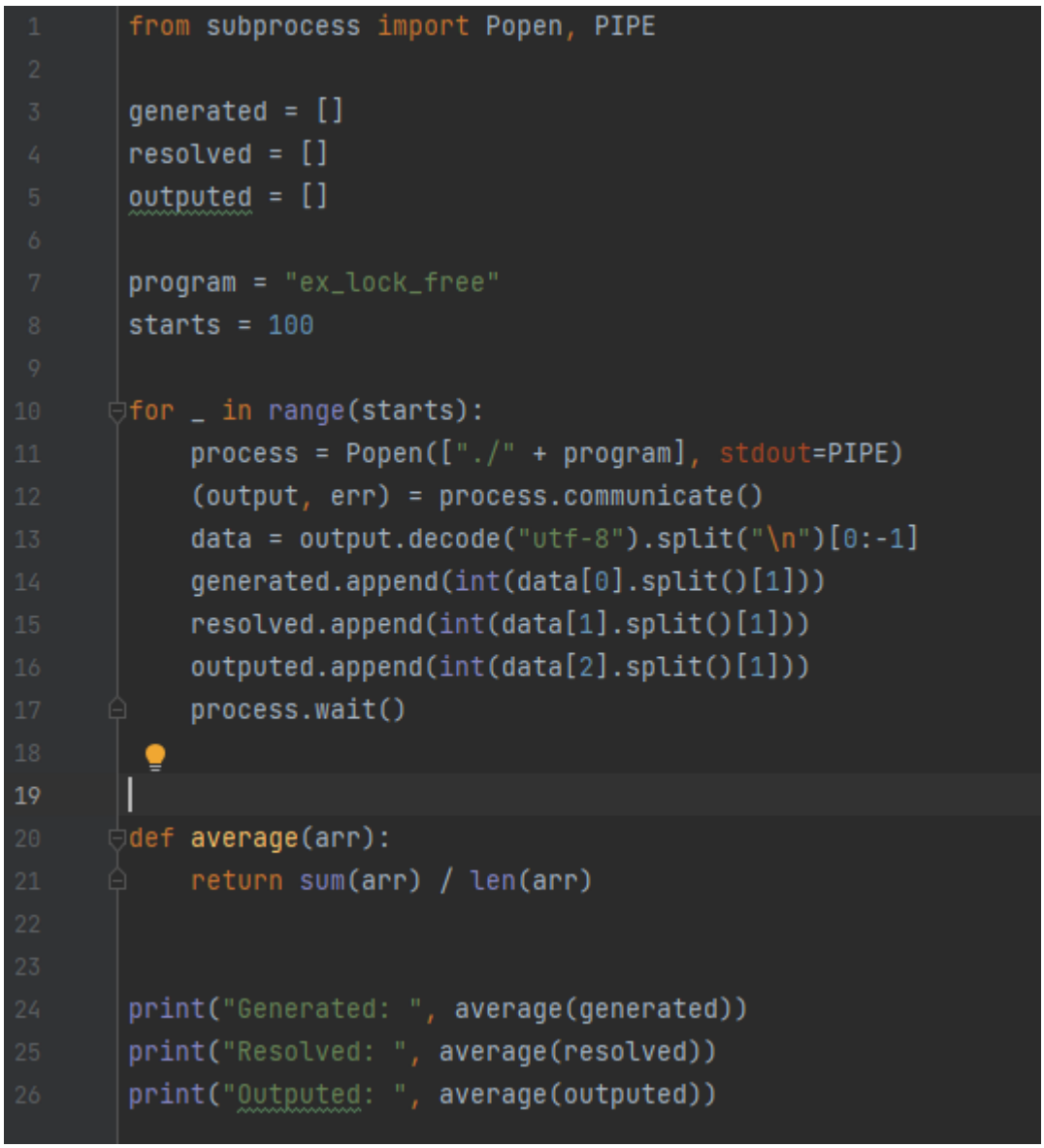
Если элемент за текущим хвостом существует, пытаемся помочь другим потокам, передвигая указатель на хвост при возможности.

Все попытки изменения списка производятся при помощи псевдо-функции CAS (Compare and Swap), реализованной в языке C++ функцией *compare_exchange_weak*, вызываемую у атомарной переменной, принимающей в себя ожидаемое значение в ней и новое значение. Логика проста: если значение в атомарной переменной совпадает с ожидаемым, в нее записывается новое значение и возвращается *true* иначе – *false*.

В качестве инварианта очереди без блокировок может служить утверждение, что указатель на *head* всегда находится ближе или на одном расстоянии с началом списка, чем указатель на *tail*.

Исследование.

Исследование производительности в зависимости от количества производителей и потребителей производилось при помощи запуска программ в течение 100 миллисекунд на матрицах 20 * 20 при разном количестве потоков-производителей и потоков-потребителей. Количество выполненных умножений считалось как среднее среди 100 вызовов программ. Вызовы и подсчет производились скриптом, написанным на языке Python (см. рис. 1).



```python
from subprocess import Popen, PIPE

generated = []
resolved = []
outputed = []

program = "ex_lock_free"
starts = 100

for _ in range(starts):
    process = Popen(["./" + program], stdout=PIPE)
    (output, err) = process.communicate()
    data = output.decode("utf-8").split("\n")[0:-1]
    generated.append(int(data[0].split()[1]))
    resolved.append(int(data[1].split()[1]))
    outputed.append(int(data[2].split()[1]))
    process.wait()


def average(arr):
    return sum(arr) / len(arr)


print("Generated: ", average(generated))
print("Resolved: ", average(resolved))
print("Outputed: ", average(outputed))
```

Рисунок 1 – Скрипт для подсчета количества проведенных умножений.

Результаты для всех очередей представлены в таблице (см. табл. 1).

Таблица 1 – Результаты для потокобезопасных очередей.

Производители	Потребители	Умножения		
		ThreadSafe	FineGrained	LockFree
1	1	723	722	660
3	3	788	1056	802
10	10	854	1191	771
2	5	999	1232	714
5	2	595	795	665

Из проведенного исследования можно сделать вывод, что потокобезопасная очередь без блокировок проигрывает по производительности очередям с «грубой» и «тонкой» блокировками. Это связано с тем, что lock-free подход написания кода не ограничивает потоки при конкуренции за ресурсы. В случае реализации очередей с «грубой» и «тонкой» блокировками у потоков был «план» войны за операции очереди, а в очереди без блокировок потоки не дают друг другу быстро выполнить свою задачу.

Выводы.

В ходе лабораторной работы были изучены способы реализации потокобезопасных структур данных без блокировок. Была реализована шаблонная очередь, поддерживающая безопасную работу с несколькими потоками и не использующая блокировки. Также было проведено исследование производительности очередей в зависимости от количества потоков-производителей и потоков-потребителей. Был сделан вывод о низкой эффективности очереди, не использующей блокировок.