Университет ИТМО
Кафедра ИПМ

Лабораторная работа №2
По предмету:
«ТЕОРИЯ РАСПРЕДЕЛЁННЫХ И ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ»
Вариант: 1

Работу выполнила:
Гулямова С. И.
Студентка группы:
P4115

Санкт-Петербург
2018 г

Задание:

Разработать консольное приложение для обработки большого потока данных из внешнего источника (файла) с применением алгоритма распределения данных. В рамках данной работы необходимо реализовать три алгоритма: **Round-Robin**, **Least Loaded**, **Predictive**. Для алгоритма «*Predictive*» необходимо самостоятельно выбрать функцию оценки сложности данных и предоставить обоснование выбора. В качестве аргументов запуска приложения должны передаваться следующие параметры: номер способа распределения данных, путь до файла с исходными данными. После запуска программа должна выполнить свою работу и вывести на экран следующие показатели:

1. Общее время обработки данных;
2. Полезное время работы потока (среднее значение в %);
3. Время ожидания потока (среднее значение в %);
4. Диаграмму распределения данных по потокам (в %).

Формат входного файла и алгоритм обработки информации определяется вариантов из первой лабораторной работы. Сравнить полученные результаты и сделать вывод. Каждый запуск необходимо выполнять на максимальном кол-во потоков.

Выполнение:
Для выполнения работы была реализована структура данных: «Цикличная очередь». Структура используется для упрощенной реализации алгоритма Round Robin.

```
struct CircularQueue<T: Equatable> {
    fileprivate var items = [T]()
    fileprivate var pointer = 0

    var itemsArray: [T] {
        return items
    }

    var size: Int {
        return items.count
    }
    var isEmpty: Bool {
        return items.isEmpty
    }
    var currentPosition: Int {
        return pointer
    }

    mutating func enque(value: T) {
        items.append(value)
    }

    mutating func peek() -> T? {
        if !items.isEmpty {
            if pointer == items.count {
                pointer = 0
            }
            pointer += 1
            return items[pointer - 1]
        } else {
            return nil
        }
    }

    func first() -> T? {
        return items.first
    }

    func last() -> T? {
        return items.last
    }

    func get(at position: Int) -> T? {
        if position < items.count && position >= 0 {
            return items[position]
        } else {
```

```
            return nil
        }
    }

    func current() -> T? {
        return items[pointer - 1]
    }

    func contains(_ item: T) -> Bool {
        return items.contains(item)
    }

    mutating func removeAll() {
        items.removeAll()
    }

    mutating func changePointerPosition(with newPosition: Int) {
        if newPosition >= 0 && newPosition < items.count {
            pointer = newPosition
        }
    }

    mutating func remove(_ item: T) {
        if let removeItemIndex = items.index(of: item) {
            items.remove(at: removeItemIndex)
        }
    }

    mutating func movePointerBack() {
        if pointer == 0 {
            // минус 2 т.к. после каждой выдачи элемента указатель сразу
смещается на следующий
            //            элемент и для получения предыдущего нам нужно сделать
шаг назад
            //            относительно используемого в данный момент
            //            то есть два шага назад от того, на который указывает
указатель
            pointer = size - 2
        } else if pointer == 1 {
            pointer = size - 1
        } else {
            pointer -= 2
        }
    }
}
```

Для хранения времени ожидания и работы операций, был реализован класс «накопитель»:

```
func addWorkTime(value: Double) {
        queue.async(flags: .barrier) { [weak self] in
            self?.usefulTimeAcumulator+=value
        }
    }

    func addWaitTime(value: Double) {
        queue.async(flags: .barrier) { [weak self] in
            self?.uselessTimeAcumulator+=value
        }
    }

    func readTotalWorkTime() -> Double {
        var result = 0.0
        queue.sync {
            result = usefulTimeAcumulator
        }
        return result
    }

    func readTotalWaitTime() -> Double {
        var result = 0.0
        queue.sync {
            result = uselessTimeAcumulator
        }
        return result
    }
```

На основе описанных выше структур, был реализован алгоритм **Round Robin**:

```
   func roundRobinCalculating(for figures: [Figure],
                              completionHandler: @escaping() -> Void) {
        var cores = getCores()
        let finalTask = BlockOperation {
           print("RR result = \
(AcumulatorService.shared.readTotalWorkTime()*100/
(AcumulatorService.shared.readTotalWaitTime() +
AcumulatorService.shared.readTotalWorkTime()))")
            completionHandler()
        }
        for figure in figures {
            let operation = CalculateOperation(figure: figure)
            finalTask.addDependency(operation)
            cores.peek()?.addOperation(operation)
        }
        cores.peek()?.addOperation(finalTask)
    }
```

Least Loaded:
Для реализации алгоритма Least Loaded, использовался обычный массив потоков, затем при каждом добавление новой задачи, мы сортируем массив по количеству задач в очереди. В элемент с 0 индексом, наименьшим числом задач в очереди, добавляется текущая задача.

```
    func leastLoaded(for figures: [Figure],
                     completionHandler: @escaping() -> Void) {
        var cores = getCores().itemsArray
        let finalTask = BlockOperation {
            print("LL result = \
(AcumulatorService.shared.readTotalWorkTime()*100/
(AcumulatorService.shared.readTotalWaitTime() +
AcumulatorService.shared.readTotalWorkTime()))")
            completionHandler()
        }

        for figure in figures {
            let operation = CalculateOperation(figure: figure)
            finalTask.addDependency(operation)
            cores.sorted{ $0.operationCount >
$1.operationCount }.first?.addOperation(operation)
        }
        cores[0].addOperation(finalTask)
    }
```

Predictive:
Для реализации алгоритма предиктивов, был модифицирован enum с перечислением вариантов фигур. Каждой из фигур был присвоен индекс сложности вычисления.

```
enum Figures: UInt32, Codable {
    case circle = 3
    case box = 1
    case triangle = 2
}
```

К структуре очереди операций было добавлено расширение, возвращающее общую сложность очереди. Мы берем все операции в очереди, затем для каждой из них получаем сложность вычисления площади фигуры. Возвращаем общую сложность задач в очереди:

```
extension OperationQueue<Figure> {
    func complexiety() -> Int {
        return operations.map{ $0.figure.type }
    }
}
```

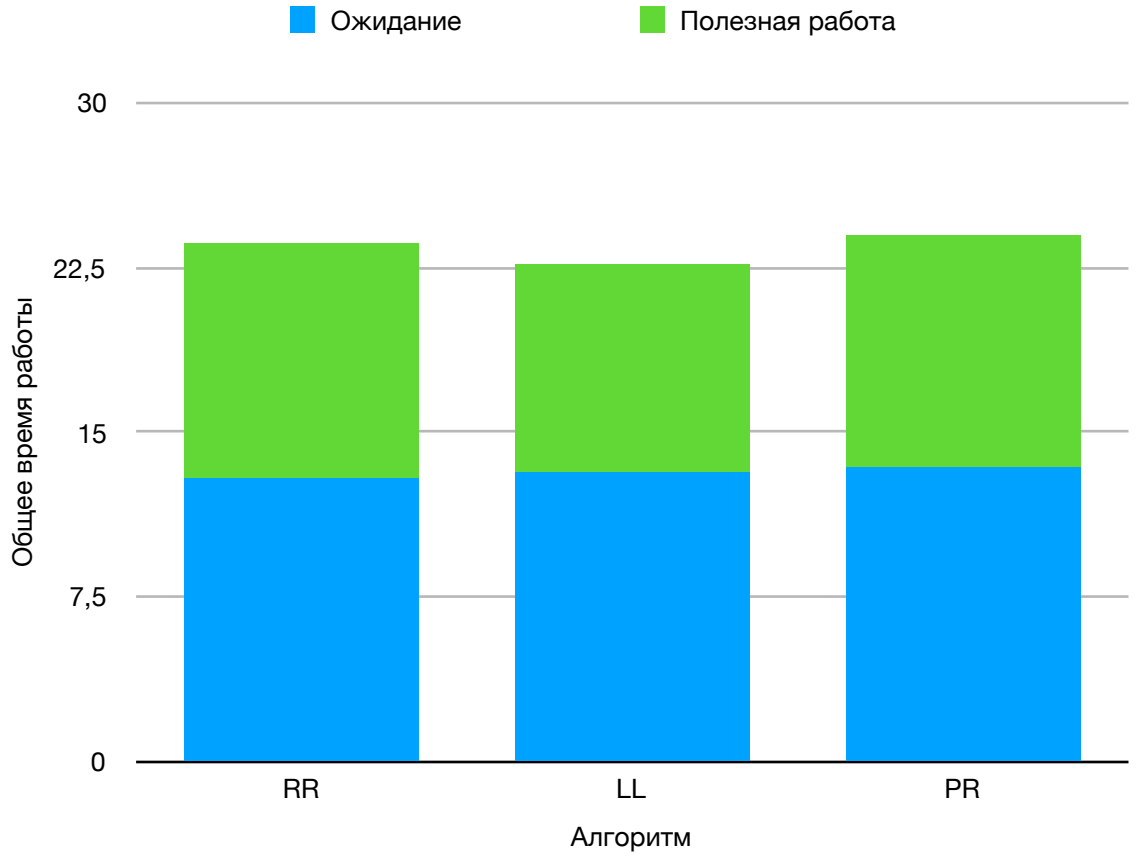
Реализация алгоритма:
В отличии от предыдущего алгоритма, в данном случае сортировка идет не по числу запросов в очереди, а по их суммарной сложности.

```
func predictive(for figures: [Figure],
                    completionHandler: @escaping() -> Void) {
        var cores = getCores().itemsArray
        let finalTask = BlockOperation {
            print("PB result = \
(AcumulatorService.shared.readTotalWorkTime()*100/
(AcumulatorService.shared.readTotalWaitTime() +
AcumulatorService.shared.readTotalWorkTime()))")
            completionHandler()
        }

        for figure in figures {
            let operation = CalculateOperation(figure: figure)
            finalTask.addDependency(operation)
            cores.sorted{ $0.complexiety() >
$1.complexiety() }.first?.addOperation(operation)
        }
        cores[0].addOperation(finalTask)
    }
```

Результат:

	Round Robin	Least Loaded	Predictive
Кол-во фигур	500 000		
Кол-во потоков	20		
Общее время работы	23.6201280000003	22.6595406000006	24.0219483999999
Полезное время работы	10.6620429999999	9.48544269999999	10.6326985
Время ожидания	12.9580850000003	13.1740979000006	13.3892498999999



Вывод:
Наиболее быстрым оказался алгоритм Least Loaded, однако он уступил Round Robin по показателю полезной нагрузки. Наиболее неэффективный — предикаты. К результатам стоит относиться скептически так как вычисление площади очень «легкая» операция, а разница в сложности между кругом и квадратом незначительная, так что использования алгоритма предикатов в этом случае не является правильным выбором. Накладные расходы на сложную сортировку очереди не могут сравниться с разницей сложности операций. Least loaded наиболее простой и самый эффективный алгоритм, его легко реализовать и его результаты превзошли два других алгоритма в конкретном примере.