Государственное образовательное учреждение высшего образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №5

Конвейер

Студент группы ИУ7-54Б,
Котов Никита

2019 г.

Содержание

Введение

При обработке данных могут возникать ситуации, когда необходимо обработать множество данных последовательно несколькими алгоритмами. В этом случае удобно использовать конвейерную обработку данных. Это может быть полезно при следующих задачах:

- шифровании данных;
- сортировки и фильтрации данных;
- и др.

Цель данной работы: получить навык организации асинхронного взаимодействия потоков на примере конвейерной обработки данных.

В рамках выполнения работы необходимо решить следующие задачи:

- рассмотреть и изучить конвейерную обработку данных;
- реализовать конвейер с количеством лент не меньше трех в многопоточной среде;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

1.1 Описание конвейерной обработки данных

При конвейерной обработке данных каждая лента имеет свою очередь с некоторыми задачами, ожидающими обработки. Лента берет данные из своей очереди с входными данными, проводит с ними необходимые операции и передает в очередь следующей ленты или, в случае последней ленты, в пул обработанных задач.

2 Конструкторская часть

2.1 Разработка конвейерной обработки данных

Принцип работы стадийной обработки представлен на рис. 1.

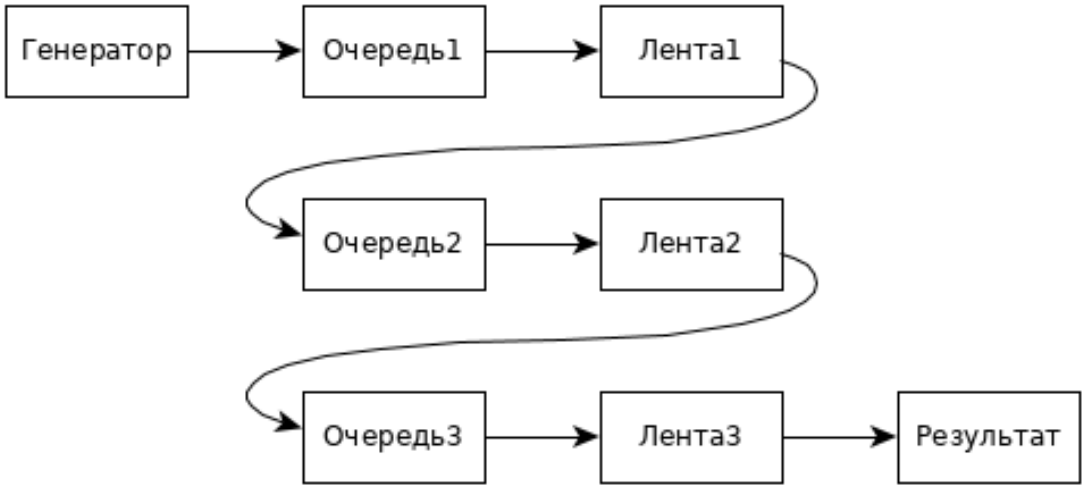

Рис. 1. Конвейерная обработка данных

3 Технологическая часть

3.1 Требования к программному обеспечению

В программе должны быть реализованы как минимум три ленты обработки данных и обеспечен вывод времени входа и выхода каждой задачи из каждой ленты.

3.2 Средства реализации

В качестве языка программирования был выбран Golang, так как он предоставляет широкие возможности и крайне удобный интерфейс для эффективной реализации асинхронной, параллельной обработки данных [1].

Для измерения времени использовалась стандартная библиотека time. Так как основное время работы составляет ожидание sleep, то достаточно замерить время работы один раз.

3.3 Листинг кода

В листинге 1 представлена реализация конвейерная обработка данных.

Листинг 1. Конвейерная обработка данных

```
package main

import (
        "fmt"
        "math/rand"
        "sync"
        "time"
)

const (
        TasksNumber = 3
)

type job func(in, out chan interface{})

type Task struct {
        Value       int
        Result      int
        StartTimes  []time.Time
        EndTimes    []time.Time
}

func startJob(function job, in, out chan interface{}, wg *sync.
   WaitGroup) {
        defer wg.Done()
        defer close(out)
        function(in, out)
}

func ExecutePipeline(jobs ...job) {
        wg := &sync.WaitGroup{}
        inChan := make(chan interface{}, TasksNumber)

        for _, jobItem := range jobs {
                outChan := make(chan interface{}, TasksNumber)
                wg.Add(1)

                go startJob(jobItem, inChan, outChan, wg)
                inChan = outChan
        }

        wg.Wait()
}

func dataGenerator(_, out chan interface{}) {
        s := rand.NewSource(time.Now().UnixNano())
        r := rand.New(s)
        for i := 0; i < TasksNumber; i++ {
```

```
                val := r.Intn(100)
                task := Task{Value: val, Result: val}
                out <- task
        }
}

func makeHandler(sleepDuration int, multiplier int) func(chan
   interface{}, chan interface{}) {
        return func(in, out chan interface{}) {
                for rawTask := range in {
                        task := rawTask.(Task)
                        task.StartTimes = append(task.StartTimes,
                           time.Now())

                        task.Result *= multiplier
                        time.Sleep(time.Millisecond * time.Duration
                           (sleepDuration))

                        task.EndTimes = append(task.EndTimes, time.
                           Now())
                        out <- task
                }
        }
}

func logHandler(in, _ chan interface{}) {
        for rawTask := range in {
                task := rawTask.(Task)
                fmt.Printf("Task's start value = %d, result = %d\n"
                   , task.Value, task.Result)
                fmt.Printf("Start 1st: %s, end 1st: %s\n", task.
                   StartTimes[0].Local(), task.EndTimes[0].Local())
                fmt.Printf("Start 2nd: %s, end 2nd: %s\n", task.
                   StartTimes[1].Local(), task.EndTimes[1].Local())
                fmt.Printf("Start 3d: %s, end 3d: %s\n", task.
                   StartTimes[2].Local(), task.EndTimes[2].Local())
                fmt.Println()
        }
}

func main() {
        jobs := []job{
                job(dataGenerator),
                job(makeHandler(100, 2)),
                job(makeHandler(200, 3)),
                job(makeHandler(300, 2)),
                job(logHandler),
        }

        ExecutePipeline(jobs...)
}
```

3.4 Тестирование фунций

Для тестирования были реализованы функции, представленные на листинге 2. Результаты тестирования представоены в таблице 1. Видно, что тестирование пройдено успешно.

Листинг 2. Тестовые задачи

```
job(func(in, out chan interface{}) {
        out <- uint32(1)
        out <- uint32(3)
        out <- uint32(4)
}),
job(func(in, out chan interface{}) {
        for val := range in {
                out <- val.(uint32) * 3
        }
}),
job(func(in, out chan interface{}) {
        for val := range in {
                fmt.Println("collected", val)
                atomic.AddUint32(&recieved, val.(
                   uint32))
        }
}),
```

Таблица 1. Тестирование конвейерной обработки

Входные данные	Ожидаемый результат	Результат
1,3,4	24	24

4 Экспериментальная часть

4.1 Замеры времени работы функций

Реализованная контейнерная обработка данных работает за 1,2с. Последовательная реализация потребовала бы $(0,1 + 0,2 + 0,3) * 3 = 1,8$с, что на 50% медленней.

Заключение

В рамках лабораторной работы была рассмотрена и изучена конвейерная обработка данных. Благодаря ней возможна крайне удобная реализация задач, тербующих поэтапной обработки некоторого набора данных, а также в некоторых случаях позволяет значительно ускорить выполнение программы (в реализованном синтетическом примере выигрыш составил 50%).

Список литературы

[1] Dan Gorby. Multi-thread For Loops Easily and Safely in Go [Электронный ресурс]// Medium. 2016. 8 февраля. URL:https://medium.com/@greenraccoon23/multi-thread-for-loops-easily-and-safely-in-go-a2e915302f8b (дата обращения: 28.10.2019).