Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 5

по курсу «Синхронизация потоков»

Студент: Пишикина М.В.

Группа: ИУ9-51Б

Преподаватель: Царев А.С.

Москва 2024

Содержание

1 Практическая реализация задачи 1

```
package main

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

const (
	rows     = 5 // Размеры матрицы
	cols     = 5
	steps    = 5 // Количество шагов моделирования
	nThreads = 16 // Количество потоков
)

type Barrier struct {
	sync.Mutex
	cond   *sync.Cond
	count  int
	limit  int
}

// Инициализация барьера
func NewBarrier(limit int) *Barrier {
	b := &Barrier{limit: limit}
	b.cond = sync.NewCond(&b.Mutex)
	return b
}

// Ожидание барьера
```

```
func (b *Barrier) Wait() {
	b.Lock()
	defer b.Unlock()
	b.count++
	if b.count == b.limit {
		b.count = 0
		b.cond.Broadcast()
	} else {
		b.cond.Wait()
	}
}

// Функция для создания случайной матрицы
func createRandomMatrix(rows, cols int) [][]int {
	matrix := make([][]int, rows)
	rand.Seed(time.Now().UnixNano())
	for i := range matrix {
		matrix[i] = make([]int, cols)
		for j := range matrix[i] {
			matrix[i][j] = rand.Intn(2) // Случайные значения 0 или 1
		}
	}
	return matrix
}

// Функция для копирования матрицы
func copyMatrix(source [][]int) [][]int {
	rows := len(source)
	cols := len(source[0])
	copy := make([][]int, rows)
	for i := range source {
		copy[i] = make([]int, cols)
		for j := range source[i] {
```

```
                copy[i][j] = source[i][j]
            }
        }
        return copy
}

// Функция для подсчета живых (и нет) соседей с учетом тороидальных
 ↪  границ
func countLiveNeighbors(grid [][]int, x, y int) int {
        dirs := [][2]int{
            {-1, -1}, {-1, 0}, {-1, 1},
            {0, -1},          {0, 1},
            {1, -1}, {1, 0}, {1, 1},
        }
        count := 0
        for _, dir := range dirs {
            nx, ny := (x+dir[0]+rows)%rows, (y+dir[1]+cols)%cols
            count += grid[nx][ny]
        }
        return count
}

func evolve(matrix, newMatrix [][]int, startRow, endRow int, b *Barrier,
 ↪  rowCount, colCount int) {
        for x := startRow; x <= endRow; x++ {
            for y := 0; y < colCount; y++ {
                // Заменили вызов countNeighbors на countLiveNeighbors
                liveNeighbors := countLiveNeighbors(matrix, x, y)
                if matrix[x][y] == 1 {
                    if liveNeighbors < 2 || liveNeighbors > 3 {
                        newMatrix[x][y] = 0
                    } else {
                        newMatrix[x][y] = 1
```

```
                        }
                    } else {
                        if liveNeighbors == 3 {
                            newMatrix[x][y] = 1
                        }
                    }
                }
            }
            b.Wait() // Ждем, пока все потоки завершат обновление
        }

        func main() {
            // Инициализация матрицы
            matrix := [][]int{
                {1, 0, 1, 0, 1},
                {0, 0, 0, 1, 0},
                {1, 0, 1, 1, 0},
                {0, 1, 0, 0, 0},
                {1, 1, 0, 1, 0},
            }

            // Генерация случайной матрицы
            //matrix := createRandomMatrix(rows, cols)
            fmt.Println("Исходная матрица:")
            printGrid(matrix)

            // Замер времени выполнения симуляции
            start := time.Now()

            // Инициализация барьера
            b := NewBarrier(nThreads)

            var totalStepTime time.Duration
```

```
for step := 0; step < steps; step++ {
        startStep := time.Now() // Начало шага

        fmt.Printf("\nШаг %d:\n", step+1)

        rows := len(matrix)
        cols := len(matrix[0])

        newMatrix := make([][]int, rows)
        for i := range newMatrix {
                newMatrix[i] = make([]int, cols)
        }

        var wg sync.WaitGroup
        rowsPerThread := rows / nThreads

        for i := 0; i < nThreads; i++ {
                startRow := i * rowsPerThread
                endRow := startRow + rowsPerThread - 1

                if i == nThreads-1 {
                        endRow = rows - 1
                }

                wg.Add(1)
                go func(startRow, endRow int) {
                        defer wg.Done()
                        evolve(matrix, newMatrix, startRow, endRow, b, rows,
                        ↪  cols)
                }(startRow, endRow)
        }
```

```
            wg.Wait() // Ждем завершения всех горутин

            matrix = copyMatrix(newMatrix)

            printGrid(matrix)

            stepDuration := time.Since(startStep) // Время одного шага
            totalStepTime += stepDuration
        }

        // Вычисление среднего времени одного шага
        avgStepTime := totalStepTime / time.Duration(steps)
        fmt.Printf("\nСреднее время выполнения одного шага: %v\n",
        ↪  avgStepTime)

        elapsed := time.Since(start)
        fmt.Printf("\nВремя выполнения симуляции: %v\n", elapsed)
}

func printGrid(grid [][]int) {
        for _, row := range grid {
            for _, cell := range row {
                fmt.Printf("%d ", cell)
            }
            fmt.Println()
        }
        fmt.Println()
}
```

2 Практическая реализация

```
import threading
import random

class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
        self.read_lock = threading.Lock()  # Блокировка для операций чтения
        self.write_lock = threading.Lock() # Блокировка для операций записи

    def contains(self):     # проверка на дубликаты: True - найдены дубликаты
        with self.read_lock:
            seen = set()    # seen нужен для хранения уже встреченных значений
            current = self.head
            while current:
                if current.value in seen:
                    return True
                seen.add(current.value)
                current = current.next
            return False

    def append(self, value):    # добавление уникальных значений в конец
    ↪   списка
        with self.write_lock:       # захват блокировки
            current = self.head
            while current:
                if current.value == value:
```

```
                return
            current = current.next

        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

        # print(f"Число {value} добавлено в список.")

def generate_numbers(linked_list, num_values, thread_id):
    for _ in range(num_values):
        value = random.randint(0, 1000)
        linked_list.append(value)

def main():
    num_threads = 4
    num_values_per_thread = 10

    linked_list = LinkedList()
    threads = []

    for i in range(num_threads):
        thread = threading.Thread(target=generate_numbers, args=(linked_list,
        ↪  num_values_per_thread, i))
        threads.append(thread)
        thread.start()

    for thread in threads:
```

```
        thread.join()

    # Вывод всех элементов списка
    print("\nВсе уникальные числа в списке:")
    with linked_list.read_lock:
        current = linked_list.head
        while current:
            print(current.value, end="\n")
            current = current.next
    print()

    if linked_list.contains():
        print("Дубликаты найдены в списке!")
    else:
        print("Проверка завершена, повторяющихся чисел нет.")

if __name__ == "__main__":
    main()
```

3 Характеристика устройства

Устройство: MacBook Pro 2020

Операционная система: macOS Sonoma

Процессор: Intel Core i5

Характеристика процессора: 4-ядерный чип, частота 2 ГГц

Оперативная память: 16GB LPDDR4X

4 Работа программы 1

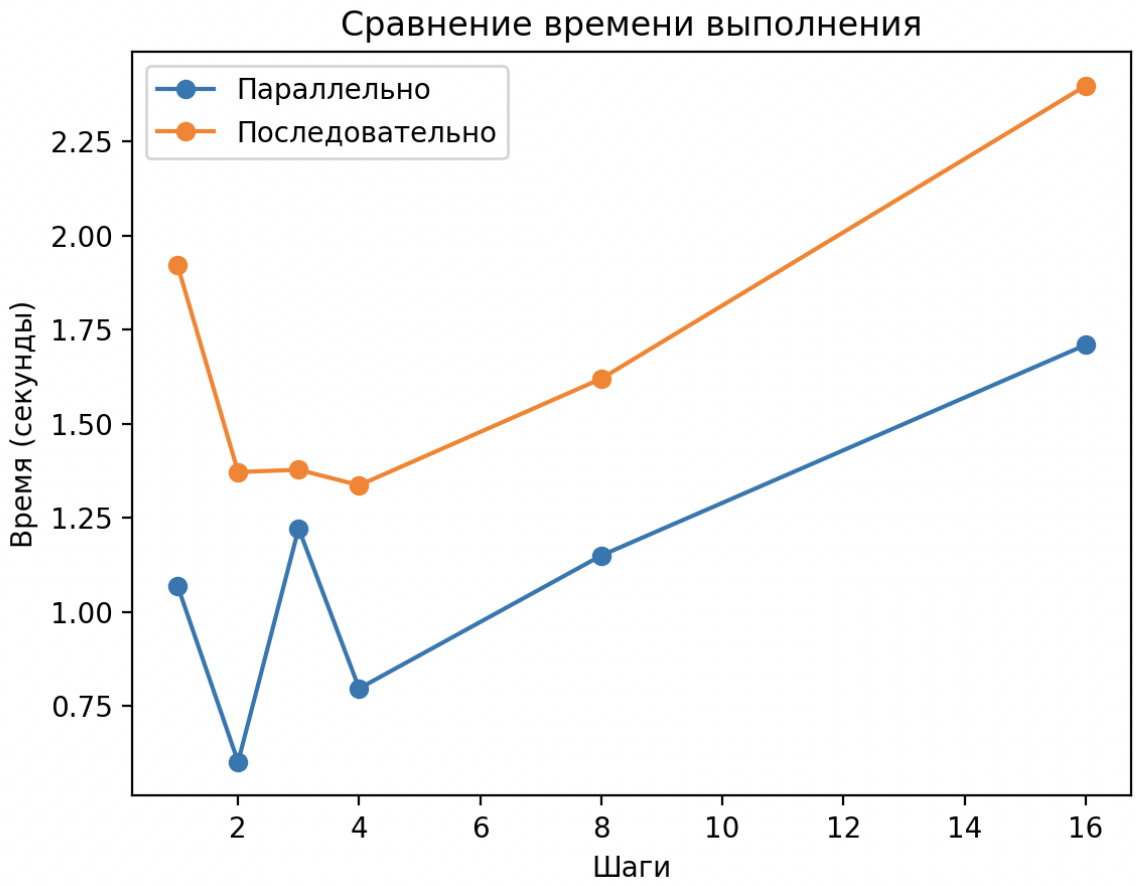

Рис. 1 — График: Зависимость времени выполнения от количества процессоров