

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«КРЫМСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ им. В. И. ВЕРНАДСКОГО»
ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
Кафедра компьютерной инженерии и моделирования

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3
«МНОГОПОТОЧНАЯ СИСТЕМА С ВОЗМОЖНЫМИ СОСТОЯНИЯМИ
ВЗАИМНОЙ БЛОКИРОВКИ И РЕСУРСНОГО ГОЛОДАНИЯ»

Лабораторная работа
по дисциплине
«Параллельные и распределенные вычисления»
студента 4 курса группы ПИ-б-о-182(2)
Змитрович Никита Сергеевич
направления подготовки 09.03.04 «Программная инженерия»

Научный руководитель
старший преподаватель кафедры
компьютерной инженерии и моделирования

(оценка)

(подпись, дата)

Чабанов В.В.

Симферополь, 2021

Цель:

1. Изучить многопоточную систему с возможными состояниями взаимной блокировки и ресурсного голодания;
2. Научится разрабатывать алгоритмы не допускающие состояния взаимной блокировки;
3. Научится разрабатывать алгоритмы не допускающие состояния ресурсного голодания;

Постановка задачи:

n [2..10] безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов. Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок). Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются отдельными действиями, которые должны выполняться одно за другим. Реализуйте алгоритм работы описанной системы, при этом предполагается, что время в течении которого философы размышляют T_{think} и едят T_{eat} задано и одинаково для каждого философа.

Выполнение работы

Задание 1.

Таблица 1 — Результаты измерения для 2 философов

	0/1	0.5/1	1/1	2/1
1	1	0.500	0.500	0.500
2	0	0.499	0.499	0.499

Таблица 2 — Результаты измерения для 5 философов

	0/1	0.5/1	1/1	2/1
1	0.280	0.201	0.199	0.200
2	0.183	0.199	0.206	0.199
3	0.175	0.198	0.196	0.199
4	0.105	0.200	0.194	0.199
5	0.255	0.199	0.204	0.199

Таблица 3 — Результаты измерения для 10 философов

	0/1	0.5/1	1/1	2/1
1	0.096	0.099	0.099	0.100
2	0.157	0.099	0.099	0.099
3	0.021	0.100	0.100	0.100
4	0.292	0.099	0.100	0.099
5	0.023	0.100	0.099	0.099
6	0.092	0.099	0.099	0.100
7	0.051	0.099	0.099	0.99
8	0.084	0.100	0.099	0.100
9	0.094	0.099	0.099	0.100
10	0.086	0.100	0.100	0.100

Вывод:

Изучили многопоточную систему с возможными состояниями взаимной блокировки и ресурсного голодания на платформе JVM. Научились разрабатывать алгоритмы не допускающие состояния взаимной блокировки. Научились разрабатывать алгоритмы не допускающие состояния ресурсного голодания.

Приложение

```
public class Main {

    static final int NUMBER_OF_PHILOSOPHERS = 10;

    static Chopstick[] chopSticks = new Chopstick[NUMBER_OF_PHILOSOPHERS];
    static Philosopher[] philosophers = new Philosopher[NUMBER_OF_PHILOSOPHERS];

    static double T = 0;

    public static void main(String[] args) throws InterruptedException {

        for (int i = 0; i < chopSticks.length; i++) {
            chopSticks[i] = new Chopstick(i);
        }

        for (int i = 0; i < philosophers.length; i++) {
            philosophers[i] = new Philosopher(chopSticks[i % chopSticks.length], chopSticks[(i + 1) %
chopSticks.length]);
            philosophers[i].start();
        }

        Thread.sleep(5000);

        for (var philosopher : philosophers) {
            philosopher.interrupt();
            T += philosopher.eatingCounter;
        }

        for (int i = 0; i < philosophers.length; i++) {
            System.out.println("Philosopher " + i + " " + philosophers[i].eatingCounter / T);
        }
    }
}
```

```

public class Philosopher extends Thread {

    final long THINK_TIME = 4;
    final long EAT_TIME = 2;

    final Chopstick leftChopstick;
    final Chopstick rightChopstick;
    long eatingCounter = 0;

    public Philosopher(Chopstick leftChopstick, Chopstick rightChopstick) {
        this.leftChopstick = leftChopstick;
        this.rightChopstick = rightChopstick;
    }

    @Override
    public void run() {
        try {
            while (true) {
                if (leftChopstick.getState() != StateOfChopstick.BUSY && rightChopstick.getState() !=
StateOfChopstick.BUSY) {
                    doAction();
                }
            }
        } catch (InterruptedException e) {
            this.interrupt();
            e.printStackTrace();
        }
    }

    private void doAction() throws InterruptedException {
        synchronized (leftChopstick) {
            leftChopstick.setState(StateOfChopstick.BUSY);
            //      System.out.println("Philosopher " + getName() + " take " + leftChopstick.numberOfChopstick + "
chopstick");
            synchronized (rightChopstick) {
                rightChopstick.setState(StateOfChopstick.BUSY);
                //      System.out.println("Philosopher " + getName() + " take " + rightChopstick.numberOfChopstick + "
chopstick");
                System.out.println("Philosopher " + getName() + " eating...");
                eatingCounter += EAT_TIME;
                sleep(EAT_TIME);
            }
            System.out.println("Philosopher " + getName() + " thinking...");
            //      System.out.println("Philosopher " + getName() + " leave " + rightChopstick.numberOfChopstick + "
chopstick");
            rightChopstick.setState(StateOfChopstick.FREE);
        }
        //      System.out.println("Philosopher " + getName() + " leave " + leftChopstick.numberOfChopstick + "
chopstick");
        leftChopstick.setState(StateOfChopstick.FREE);
        sleep(THINK_TIME);
    }
}

```

}
}

```
public class Chopstick {

    Chopstick(int numberOfChopstick) {
        this.numberOfChopstick = numberOfChopstick;
    }

    final int numberOfChopstick;
    StateOfChopstick state = StateOfChopstick.FREE;

    synchronized StateOfChopstick getState() {
        return state;
    }

    synchronized void setState(StateOfChopstick newState) {
        state = newState;
    }

    @Override
    public String toString() {
        return "Chopstick{" +
            "numberOfChopstick=" + numberOfChopstick +
            '}';
    }
}

public enum StateOfChopstick {
    FREE,
    BUSY
}
```