Lab 05

Problem Pięciu Filozofów

Adrian Madej 30.10.2023

1. Treść zadań

Problem pięciu filozofów:

- Każdy filozof zajmuje się głównie myśleniem
- Od czasu do czasu potrzebuje zjeść
- Do jedzenie potrzebne mu są oba widelce po jego prawej i lewej stronie
- Jedzenie trwa skończona (ale nieokreślona z góry) ilość czasu, po czym filozof widelce odkłada i wraca do myślenia
- Cykl powtarza się od początku

Zadania:

1. Zaimplementować trywialne rozwiązanie z symetrycznymi filozofami. Zaobserwować problem blokady.
2. Zaimplementować rozwiązanie z widelcami podnoszonymi jednocześnie. Jaki problem może tutaj wystąpić ?
3. Zaimplementować rozwiązanie z lokajem.

Wykonać pomiary dla każdego rozwiązania i wywnioskować co ma wpływ na wydajność każdego rozwiązania.

2. Rozwiązanie zadań

Zadanie 1

Implementacja trywialnego rozwiązania z symetrycznymi filozofami. Problem blokady.

Koncepcja

Każdy filozof zajmuje się myśleniem i jedzeniem. Poszczególne czasy trwania tych czynności zostały zdefiniowane w klasie Philosopher. W momencie w którym filozof będzie chciał jeść, musi podnieść dwa widelce, sięgając najpierw po lewy, a następnie po prawy. Każdy filozof musi zjeść po 50 posiłków. Czas myślenia stanowi dwukrotność czasu jedzenia.

Zmierzymy czas oczekiwania każdego z filozofów na dostęp do widelców. Przeprowadzimy 50 powtórzeń, a wynikiem będą uśrednione czasy. W przypadku wystąpienia zakleszczenia, wątki zostaną automatycznie zakończone; wówczas nie uwzględnimy takiego przypadku w średniej.

Implementacja

Klasa Fork

```
public class Fork {
    private final Semaphore semaphore = new Semaphore(1);

    public void acquire() throws InterruptedException {
        semaphore.acquire();
    }

    public void release(){
        semaphore.release();
    }
}
```

Klasa Philosopher

```
public class Philosopher extends Thread{
    private static final int EATING_TIME = 15;

    private static final int THINKING_TIME = 30;
    private static final int ITERATIONS = 50;

    private final int id;
    private final Fork leftFork;
    private final Fork rightFork;

    long starvingTime = 0;

    Philosopher(int id, Fork leftFork, Fork rightFork){
        this.id = id;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }
```

```
    private void think() throws InterruptedException {
        //System.out.println("Philosopher " + id + " started
thinking");
        sleep(THINKING_TIME);
        //System.out.println("Philosopher " + id + " stopped
thinking");
    }

    private void eat() throws InterruptedException {
        long start = System.currentTimeMillis();
        leftFork.acquire();
        rightFork.acquire();

        starvingTime += System.currentTimeMillis() - start;
        //System.out.println("Philosopher " + id + " started
eating");
        sleep(EATING_TIME);
        //System.out.println("Philosopher " + id + " stopped
eating");

        leftFork.release();
        rightFork.release();
    }


    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            try {
                think();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            try {
                eat();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public long getStarvingTime(){
        return this.starvingTime;
    }
}
```

Klasa Main

```
class Main {
    public static Integer PHILOSOPHERS = 5;
```

```
    public static void main(String[] args) {
        long[] times = testCaseWrapper();
        StringBuilder stringBuilder = new StringBuilder();
        for (long t : times) {
            stringBuilder.append(t);
            stringBuilder.append(",");
        }

        System.out.println(stringBuilder.toString());

        Path out = Paths.get("philosophers_deadlock.csv");

        try {
            Files.writeString(out, stringBuilder.toString(),
Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static long[] testCaseWrapper() {
        LinkedList<long[]> times = new LinkedList<long[]>();

        IntStream.range(0, 50).forEach(i -> {
            Optional<long[]> results = testCase();
            results.ifPresent(times::add);
        });

        return Arrays.stream(
                        times
                                .stream()
                                .reduce(new
long[PHILOSOPHERS], (sums, r) -> {
                                    IntStream.range(0,
PHILOSOPHERS).forEach(i -> {
                                        sums[i] += r[i];
                                    });
                                    return sums;
                                }))
                .map(v -> v / times.size())
                .toArray();
    }

    public static Optional<long[]> testCase() {
        var philosophers = new Philosopher[PHILOSOPHERS];
        var forks = new Fork[philosophers.length];

        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            forks[i] = new Fork();
        });
```

```
        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            philosophers[i] = new Philosopher(i, forks[i],
forks[(i + 1) % forks.length]);
        });

        var executor = Executors.newFixedThreadPool(5);
        Arrays.stream(philosophers).forEach(executor::submit);
        executor.shutdown();
        boolean finishedNormally = false;
        try {
            finishedNormally = executor.awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {
        }

        if (finishedNormally) {
            long[] results = new long[PHILOSOPHERS];
            IntStream.range(0, PHILOSOPHERS).forEach(i -> {
                results[i] =
philosophers[i].getStarvingTime();
            });

            return Optional.of(results);
        }

        System.out.println("timeout");
        executor.shutdownNow();
        return Optional.empty();
    }
}
```

Wyniki

W wyniku uruchomienia program dostajemy następujące wyniki:

timeout

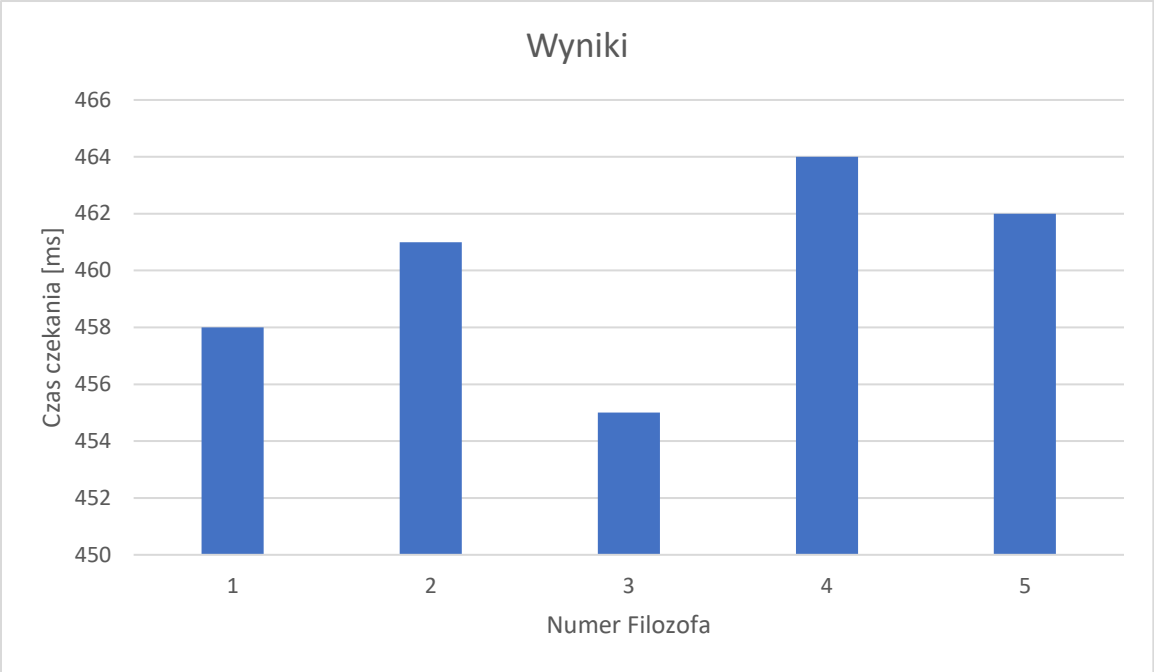
timeout

timeout

timeout

timeout

458,461,455,464,462,



Jak widzimy doszło 5-krotnie do zakleszczenia (timeout). Czasy oczekiwania na widelce są do siebie zbliżone, co wynika z faktu iż sztućce są podnoszone w sposób niedeterministyczny. Blokada następuje w momencie, gdy każdy z filozofów podniesie po jednym widelcu; wówczas każdy czeka aż zwolni się drugi widelec, co nigdy to nie następuje.

Zadanie 2

Implementacja rozwiązania z widelcami podnoszonymi jednocześnie.

Koncepcja

Podobnie jak poprzednio filozof zajmuje się myśleniem i jedzeniem, jednakże w tym wariancie filozof będzie podnosił dwa widelce na raz. Jeśli jeden z widelców będzie zajęty, to filozof zrezygnuje z jedzenia i spróbuje ponownie za jakiś czas.

Implementacja

Klasa Fork

```
public class Fork {
    private final Semaphore semaphore = new Semaphore(1);

    public boolean tryAcquire() {
        return semaphore.tryAcquire();
```

```
    }

    public void release(){
        semaphore.release();
    }
}
```

Klasa Philosopher

```
public class Philosopher extends Thread{
    private static final int EATING_TIME = 15;

    private static final int THINKING_TIME = 30;
    private static final int ITERATIONS = 50;

    private final int id;
    private final Fork leftFork;
    private final Fork rightFork;

    long starvingTime = 0;

    Philosopher(int id, Fork leftFork, Fork rightFork){
        this.id = id;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    private void think() throws InterruptedException {
        //System.out.println("Philosopher " + id + " started
thinking");
        sleep(THINKING_TIME);
        //System.out.println("Philosopher " + id + " stopped
thinking");
    }

    private void eat() throws InterruptedException {
        long start = System.currentTimeMillis();
        boolean full = false;

        while (!full) {
            if (leftFork.tryAcquire()) {
                if (rightFork.tryAcquire()){
                    starvingTime += System.currentTimeMillis()
- start;
                    sleep(EATING_TIME);
                    full = true;
                    rightFork.release();
                }
                leftFork.release();
            }
        }
```

```
    }


    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            try {
                think();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            try {
                eat();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public long getStarvingTime(){
        return this.starvingTime;
    }
}
```

Klasa Main

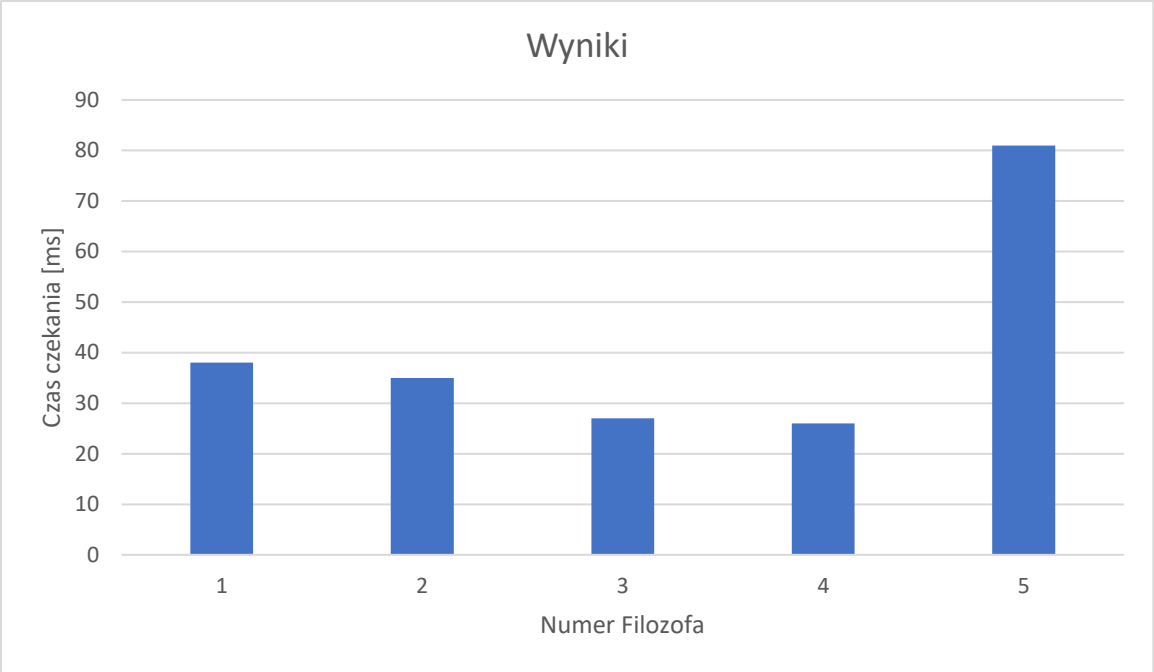
```
class Main {
    public static Integer PHILOSOPHERS = 5;

    public static void main(String[] args) {
        long[] times = testCase();
        StringBuilder stringBuilder = new StringBuilder();
        for (long t : times) {
            stringBuilder.append(t);
            stringBuilder.append(",");
        }

        System.out.println(stringBuilder);

        Path out = Paths.get("philosophers_both_forks.csv");
        try {
            Files.writeString(out, stringBuilder.toString(),
Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static long[] testCase() {
        var philosophers = new Philosopher[PHILOSOPHERS];
        var forks = new Fork[philosophers.length];
```

```
        IntStream.range(0, PHILOSOPHERS).forEach(i -> forks[i]
= new Fork());

        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            philosophers[i] = new Philosopher(i, forks[i],
forks[(i + 1) % forks.length]);
        });

        var executor = Executors.newFixedThreadPool(5);
        Arrays.stream(philosophers).forEach(executor::submit);
        executor.shutdown();

        try {
            executor.awaitTermination(Long.MAX_VALUE,
TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {
        }

        var results = new long[PHILOSOPHERS];
        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            results[i] = philosophers[i].getStarvingTime();
        });

        return results;
    }
}
```

Wyniki

W wyniku wykonania program dostajemy:

38,35,27,26,81,



Filozofowie 1-4 czekali podobną ilość czasu, jednakże filozof nr 5 czekał ponad dwukrotnie więcej. Najdłuższy czas oczekiwania stanowi trzykrotność najkrótszego czasu. Zaletą takiej implementacji jest brak występującego zakleszczenia; jednakże w tym wariancie może wystąpić zagłodzenie, czyli wątek będzie oczekiwał zbyt długo otrzymanie dostępu do zasobu; w naszym wypadku zagłodzeniu uległ filozof nr 5.

Zadanie 3

Implementacja rozwiązania z lokajem.

Koncepcja

Nad symulacją czuwa lokaj. Zarządza on dostępem do sztućców. Będzie on reprezentowany przez semafor zliczający do 4. Jeśli naraz wszyscy filozofowie będą chcieli jeść, powstrzyma on jednego z nich do czasu, gdy któryś skończy jeść.

Implementacja

Klasa Fork

```
public class Fork {
    private final Semaphore semaphore = new Semaphore(1);

    public void acquire() throws InterruptedException {
```

```
        semaphore.acquire();
    }

    public void release(){
        semaphore.release();
    }
}
```

Klasa Arbiter (lokaj)

```
public class Arbiter {
    private final Semaphore semaphore = new
Semaphore(Main.PHILOSOPHERS - 1);

    public void acquire() throws InterruptedException {
        semaphore.acquire();
    }

    public void release(){
        semaphore.release();
    }
}
```

Klasa Philosopher

```
package org.example;

public class Philosopher extends Thread{
    private static final int EATING_TIME = 15;

    private static final int THINKING_TIME = 30;
    private static final int ITERATIONS = 50;

    private final int id;
    private final Fork leftFork;
    private final Fork rightFork;
    private final Arbiter arbiter;

    long starvingTime = 0;

    Philosopher(int id, Arbiter arbiter, Fork leftFork, Fork
rightFork){
        this.id = id;
        this.arbiter = arbiter;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    private void think() throws InterruptedException {
        //System.out.println("Philosopher " + id + " started
```

```
thinking");
        sleep(THINKING_TIME);
        //System.out.println("Philosopher " + id + " stopped
thinking");
    }

    private void eat() throws InterruptedException {
        long start = System.currentTimeMillis();
        arbiter.acquire();
        leftFork.acquire();
        rightFork.acquire();

        starvingTime += System.currentTimeMillis() - start;
        //System.out.println("Philosopher " + id + " started
eating");
        sleep(EATING_TIME);
        //System.out.println("Philosopher " + id + " stopped
eating");

        leftFork.release();
        rightFork.release();
        arbiter.release();
    }


    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            try {
                think();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            try {
                eat();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public long getStarvingTime(){
        return this.starvingTime;
    }
}
```

Klasa Main

```
class Main {
    public static Integer PHILOSOPHERS = 5;

    public static void main(String[] args) {
```

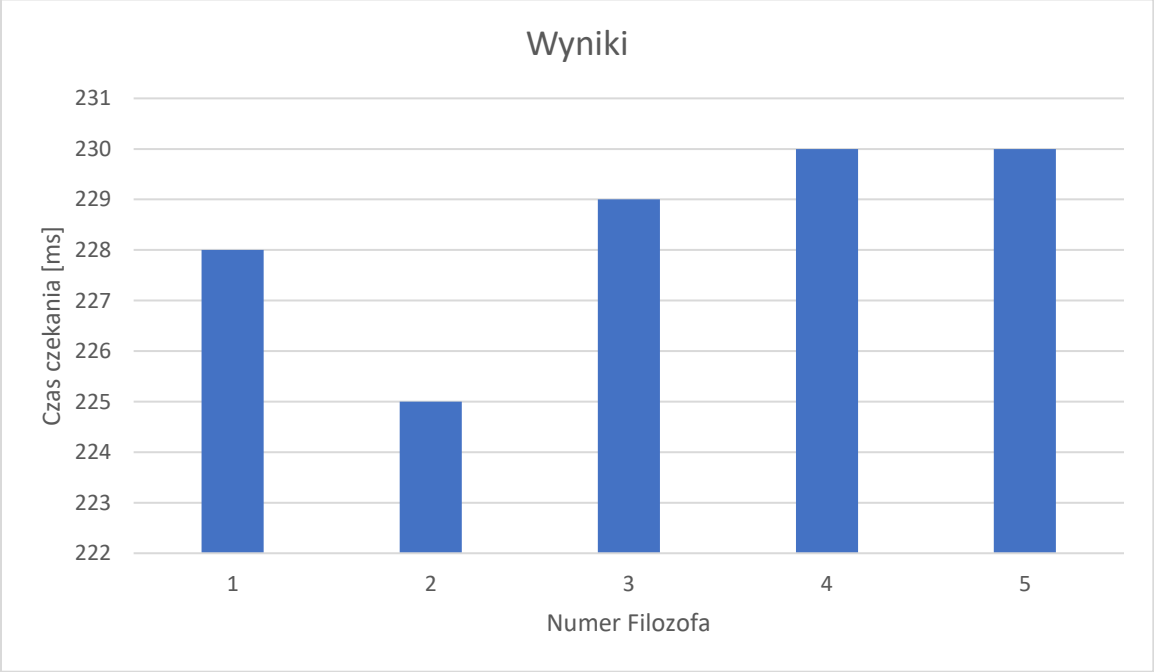
```
        long[] times = testCaseWrapper();
        StringBuilder stringBuilder = new StringBuilder();
        for (long t : times) {
            stringBuilder.append(t);
            stringBuilder.append(",");
        }

        System.out.println(stringBuilder);

        Path out = Paths.get("philosophers_arbiter.csv");
        try {
            Files.writeString(out, stringBuilder.toString(),
Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static long[] testCaseWrapper() {
        var times = new LinkedList<long[]>();

        IntStream.range(0, 50).forEach(i -> {
            long[] results = testCase();
            times.add(results);
        });

        return Arrays.stream(
                        times
                                .stream()
                                .reduce(new
long[PHILOSOPHERS], (sums, r) -> {
                                    IntStream.range(0,
PHILOSOPHERS).forEach(i -> {
                                        sums[i] += r[i];
                                    });
                                    return sums;
                                }))
                .map(v -> v / times.size())
                .toArray();
    }

    public static long[] testCase() {
        var philosophers = new Philosopher[PHILOSOPHERS];
        var forks = new Fork[philosophers.length];
        var arbiter = new Arbiter();

        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            forks[i] = new Fork();
        });

        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
```

```
            philosophers[i] = new Philosopher(i, arbiter,
forks[i], forks[(i + 1) % forks.length]);
        });

        var executor = Executors.newFixedThreadPool(5);
        Arrays.stream(philosophers).forEach(executor::submit);
        executor.shutdown();
        try {
            executor.awaitTermination(30, TimeUnit.SECONDS);
        } catch (InterruptedException ignored) {
        }

        var results = new long[PHILOSOPHERS];
        IntStream.range(0, PHILOSOPHERS).forEach(i -> {
            results[i] = philosophers[i].getStarvingTime();
        });
        return results;
    }
}
```

Wyniki

W wyniku wywołania programu otrzymujemy:

228,225,229,230,230,



Jak widać wyniki na wykresie są zbliżone do siebie, co świadczy o tym, że rozdział widelców był sprawiedliwy. W podanym rozwiązaniu nie dochodzi do zakleszczeń, gdyż mamy lokaja gwarantującego, że zawsze przynajmniej jeden filozof może użyć dwóch widelców. Dane rozwiązanie eliminuje problemy występujące w poprzednich zadaniach.

3. Wnioski

- W trywialnym rozwiązaniu z symetrycznymi filozofami, filozofowie próbują podnieść oba widelce jednocześnie. Problemem, który tu występuje, jest możliwość utknięcia w stanie blokady, zwanej "blokadą wzajemną". W sytuacji, gdy wszyscy filozofowie jednocześnie próbują podnieść widelce, mogą się wzajemnie zablokować i nie będą w stanie kontynuować jedzenia.
- W rozwiązaniu, w którym filozofowie próbują podnosić oba widelce jednocześnie, problemem może być zagrożenie zagłodzenia, czyli wątek będzie oczekiwał zbyt długo na otrzymanie dostępu do zasobu.
- Rozwiązanie z lokajem polega na wprowadzeniu trzeciej osoby, lokaja, który zarządza dostępem filozofów do widelców. Lokaj jest odpowiedzialny za zapewnienie, że filozofowie podnoszą widelce w sposób bezpieczny, tzn. nie prowadzący do zagłodzenia, czy też zakleszczenia programu.

4. Bibliografia

- Materiały do laboratorium
- Problem pięciu filozofów
- The Dining Philosophers Problem
- Dokumentacja Javy