Laboratorium V

Problem pięciu filozofów

Dominik Marek

4 listopada 2024

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1.Zadania

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

a) Zaimplementować trywialne rozwiązanie z symetrycznymi filozofami. Zaobserwować problem blokady.
b) Zaimplementować rozwiązanie z widelcami podnoszonymi jednocześnie. Jaki problem może tutaj wystąpić ?
c) Zaimplementować rozwiązanie z lokajem.
d) Wykonać pomiary dla każdego rozwiązania i wywnioskować co ma wpływ na wydajność każdego rozwiązania
e) Dodatkowe zadanie: Zaproponować autorskie rozwiązanie, inne niż w/w.

2.Rozwiązania

a)

W podanym kodzie mamy rozwiązanie klasycznego problemu "Pięciu Filozofów". Pięciu filozofów siedzi przy okrągłym stole i na przemian myśli oraz je. Każdy filozof potrzebuje dwóch widelców – jednego po lewej i jednego po prawej stronie – aby móc jeść.

Klasa *Fork* reprezentuje pojedynczy widelec. Każdy widelec ma stan isAvailable, który jest ustawiany na true, gdy widelec jest wolny, i na false, gdy jest używany przez jednego z filozofów. Klasa ta posiada trzy metody:

- take() – Jest to metoda synchronizowana, która blokuje widelec, gdy zostanie podjęta przez filozofa. Metoda sprawdza, czy widelec jest dostępny (isAvailable == true). Jeśli widelec jest zajęty, filozof (wątek) czeka, aż widelec zostanie zwolniony (wait()). Gdy widelec staje się dostępny, ustawiany jest na niedostępny (isAvailable = false).
- putDown() – Również jest to metoda synchronizowana. Zwalnia widelec, ustawiając isAvailable na true, a następnie wybudza wszystkie czekające wątki za pomocą notifyAll(). Dzięki temu inne wątki (filozofowie) mogą spróbować podnieść widelec.
- isAvailable() – Zwraca aktualny stan dostępności widelca (czy jest dostępny).

Synchronizacja metod take i putDown jest kluczowa, aby zapobiec sytuacji, w której dwóch filozofów jednocześnie próbuje podnieść ten sam widelec, co prowadziłoby do błędów w dostępie do zasobów.

Klasa *Philosopher* dziedziczy po klasie Thread, co oznacza, że każdy filozof jest realizowany jako osobny wątek w programie. Każdy filozof ma dwa widelce: lewy i prawy (leftFork oraz rightFork). Klasa ta posiada następujące elementy:

- *counter* – Licznik, który śledzi, ile razy dany filozof udał się jeść (podniósł oba widelce i zjadł).
- *run()* – Metoda, która jest wykonywana w pętli nieskończonej (while (true)). W każdej iteracji filozof próbuje podnieść dwa widelce (najpierw lewy, potem prawy), aby zjeść.

Kiedy filozofowi uda się podnieść oba widelce, zwiększa licznik counter, odkłada widelce i przechodzi do krótkiej przerwy (reprezentowanej przez Thread.sleep()).

- Podnoszenie widelców: Najpierw filozof podnosi lewy widelec (leftFork.take()), a następnie prawy (rightFork.take()).
- Jedzenie: Filozof zwiększa licznik counter, co reprezentuje udane jedzenie.
- Odkładanie widelców: Po jedzeniu filozof odkłada najpierw prawy, a potem lewy widelec (rightFork.putDown() i leftFork.putDown()).
- Przerwa: Po jedzeniu filozof przechodzi na chwilę do "myślenia", czyli do pauzy (Thread.sleep((int) (Math.random() * 100))). Długość przerwy jest losowa, co symuluje różne czasy myślenia.

Co 100 posiłków filozof wypisuje komunikat, informując, ile razy udało mu się zjeść.

- setLeftFork i setRightFork – Ustawiają referencje do lewego i prawego widelca. Dzięki temu każdy filozof ma przypisane dwa widelce (po lewej i prawej stronie).

Klasa *Fil5mon* to klasa główna, która inicjalizuje i uruchamia symulację:

- Początkowo tworzymy tablicę forks pięciu obiektów klasy Fork, co oznacza, że przy stole znajduje się pięć widelców, każdy z nich jest współdzielony przez dwóch sąsiadujących filozofów. Kolejno tworzona jest tablica philosophers pięciu obiektów klasy Philosopher. Każdemu filozofowi przypisywane są dwa widelce: lewy to forks[i], a prawy to forks[(i+1)%5]. Dzięki temu filozofowie dzielą widelce w taki sposób, że sąsiedni filozofowie muszą korzystać z tego samego widelca. Każdy filozof (wątek) jest uruchamiany za pomocą start(), co powoduje rozpoczęcie ich cyklicznej aktywności. Finalnie program czeka na zakończenie każdego wątku filozofa join().

Niestety, to rozwiązanie ujawnia problem znany jako "zakleszczenie" (ang. deadlock). Zakleszczenie może wystąpić, ponieważ filozofowie podnoszą widelce w sposób symetryczny i sekwencyjny. Jeżeli każdy filozof jednocześnie podniesie widelec po swojej lewej stronie, wszyscy będą czekać na widelec po prawej stronie, który jest zajęty przez sąsiada. W ten sposób żaden z filozofów nie będzie mógł kontynuować jedzenia, co powoduje zablokowanie programu. To proste rozwiązanie ukazuje problem zakleszczenia przy symetrycznym pobieraniu zasobów w programowaniu współbieżnym, który wymaga zastosowania dodatkowych technik, aby go uniknąć.

Poniżej przedstawiono wcześniej opisany kod programu zaimplementowany w języku Java.

```
class Fork {
    private boolean isAvailable = true;


    public synchronized void take() {
        while(!isAvailable){
            try {
                wait();
```

```
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        isAvailable = false;

    }
    public synchronized void putDown() {
        isAvailable = true;
        notifyAll();

    }

    public boolean isAvailable() {
        return isAvailable;
    }
}
```

```
public class Philosopher extends Thread {
    private int counter = 0;
    private Fork leftFork;
    private Fork rightFork;

    public void run() {
        while (true) {

            // jedzenie

            leftFork.take();
            rightFork.take();

            ++counter;

            rightFork.putDown();
            leftFork.putDown();

            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

            if (counter % 100 == 0) {
                System.out.println("Filozof: " + Thread.currentThread() +
                        "jadlem " + counter + " razy");
            }


        }
    }


    public void setLeftFork(Fork leftFork) {
        this.leftFork = leftFork;
    }

    public void setRightFork(Fork rightFork) {
        this.rightFork = rightFork;
```

```
    }


}
```

```
public class Fil5mon {
    public static void main(String[] args) {
        Fork[] forks = new Fork[5];
        for (int i = 0; i < 5; i++) {
            forks[i] = new Fork();
        }


        Philosopher[] philosophers = new Philosopher[5];
        for (int i = 0; i < 5; i++) {
            philosophers[i] = new Philosopher();
            philosophers[i].setLeftFork(forks[i]);
            philosophers[i].setRightFork(forks[(i+1)%5]);
        }

        for (int i = 0; i < 5; i++) {
            philosophers[i].start();
        }

        for(int i = 0; i < 5; i++){
            try {
                philosophers[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

b)

W implementacji kodu względem podpunktu a) wprowadzono kilka zmian. Kluczowa zmiana polega na wprowadzeniu obiektu lock w klasie Philosopher, co umożliwia synchronizację podnoszenia widelców po obu stronach filozofa jednocześnie, w obrębie jednego bloku synchronized. Oznacza to, że gdy filozof próbuje jednocześnie podnieść lewy i prawy widelec, blokada lock zapewnia, że oba widelce są zabezpieczone w tej samej sekcji krytycznej, zanim inny filozof będzie mógł próbować je podnieść. Takie podejście ma na celu zmniejszenie ryzyka deadlocku poprzez wyeliminowanie sekwencyjnego blokowania zasobów.

Choć zmiana ta redukuje prawdopodobieństwo deadlocku, nadal istnieje problem głodzenia (ang. starvation). Filozofowie mogą blokować dostęp do widelców na dłuższy czas, szczególnie gdy jednemu filozofowi udaje się często zablokować oba widelce jednocześnie, zanim inni filozofowie zdążą je podnieść. W takim przypadku filozofowie po przeciwnych stronach mogą czekać dłużej na dostęp do zasobów, co prowadzi do sytuacji, w której niektórym filozofom udaje się jeść rzadziej niż innym.

Poniżej znajduje się kod realizujący logikę wyżej opisanego problemu:

```
class Fork {
    private boolean isAvailable = true;


    public synchronized void take() {
        while(!isAvailable){
            try {
                wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

        isAvailable = false;

    }
    public synchronized void putDown() {
        isAvailable = true;
        notifyAll();

    }

    public boolean isAvailable() {
        return isAvailable;
    }
}
```

```
public class Philosopher extends Thread {
    private int counter = 0;
    private Fork leftFork;
    private Fork rightFork;
    private final Object lock;
    private final int iterationsNumber;


    public Philosopher(Object objectLock, int iterationsNumber) {
        this.lock = objectLock;
        this.iterationsNumber = iterationsNumber;
    }

    public void run() {
        long startTime = System.currentTimeMillis();
        while (true) {

            // jedzenie

            synchronized (this.lock) {
                leftFork.take();
                rightFork.take();
            }

            ++counter;

            rightFork.putDown();
            leftFork.putDown();
            if (counter == iterationsNumber) {
                long endTime   = System.currentTimeMillis();
                long elapsedTime = endTime - startTime;
```

```
                System.out.println("Filozof: " + Thread.currentThread() +
"jadlem " + counter + " razy" + " czas:" +elapsedTime);
                break;

            }



        }
    }


    public void setLeftFork(Fork leftFork) {
        this.leftFork = leftFork;
    }

    public void setRightFork(Fork rightFork) {
        this.rightFork = rightFork;
    }


}
```

```
public class Fil5mon {
    public static void main(String[] args) {
        int iterationsNumber = 10_000_000;
        Object lockObject = new Object();
        Fork[] forks = new Fork[5];
        for (int i = 0; i < 5; i++) {
            forks[i] = new Fork();
        }


        Philosopher[] philosophers = new Philosopher[5];
        for (int i = 0; i < 5; i++) {
            philosophers[i] = new Philosopher(lockObject,
iterationsNumber);
            philosophers[i].setLeftFork(forks[i]);
            philosophers[i].setRightFork(forks[(i+1)%5]);


        }

        for (int i = 0; i < 5; i++) {
            philosophers[i].start();
        }

        for(int i = 0; i < 5; i++){
            try {
                philosophers[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

c)

W implementacji zastosowano rozwiązanie problemu "Pięciu Filozofów" z użyciem "lokaja" (ang. Butler), który zarządza dostępem do widelców. Lokaj pełni rolę centralnego nadzorcy, który pozwala filozofom podnieść oba widelce tylko wtedy, gdy nie prowadzi to do sytuacji deadlocku (zakleszczenia). Dzięki lokajowi unikamy sytuacji, w której wszyscy filozofowie mogliby jednocześnie próbować podnieść widelce, co mogłoby spowodować zakleszczenie.

Klasa Butler działa jako nadzorca i wprowadza ograniczenie liczby filozofów, którzy mogą jednocześnie podnieść oba widelce. W tej implementacji:

- *takenForkNumber* – Zmienna przechowująca liczbę widelców aktualnie używanych przez filozofów. Kluczowe jest ograniczenie tej liczby do 4, aby zawsze przynajmniej jeden widelec był dostępny, co zapobiega zakleszczeniu.
- Metoda *passForks(Fork leftFork, Fork rightFork)* – Synchronizowana metoda sprawdzająca dostępność widelców i liczbę aktualnie zajętych widelców. Przebieg tej metody:
  - Jeśli jeden z widelców jest niedostępny, filozof czeka (wywołanie wait()), aż widelec zostanie zwolniony.
  - Jeśli liczba zajętych widelców wynosi 4, oznacza to, że czterech filozofów już podniosło widelce, więc filozof czeka, aby uniknąć ryzyka zakleszczenia.
  - Po przejściu obu warunków metoda zwiększa takenForkNumber o 2 (symbolizując zajęcie obu widelców) i pozwala filozofowi podnieść widelce (leftFork.take() i rightFork.take()).
- Metoda *takeForks(Fork leftFork, Fork rightFork)* – Synchronizowana metoda, w której filozof odkłada oba widelce po zakończeniu jedzenia:
  - Zmniejsza takenForkNumber o 2.
  - Wywołuje notifyAll(), aby obudzić wszystkich filozofów czekających na widelce, dzięki czemu mogą ponownie sprawdzić, czy mają dostęp do zasobów.

Klasa Fork pozostała niemal bez zmian w porównaniu do wcześniejszych wersji. Zawiera metody *take()* i *putDown(),* które synchronizują dostęp do widelców. Jednak ponieważ zarządzanie dostępem do widelców odbywa się teraz przez lokaja, metody wait() i notify() nie są już wymagana w klasie Fork. Każdy filozof musi najpierw uzyskać zgodę od lokaja, aby móc skorzystać z widelców.

Klasa Philosopher reprezentuje filozofa jako osobny wątek. W tej wersji:

- Pole butler – Każdy filozof otrzymuje referencję do instancji klasy Butler, która będzie odpowiadała za przydział widelców.
- Metoda run() – Metoda główna wykonująca cykl życia filozofa:
  - Filozof prosi lokaja o pozwolenie na podniesienie obu widelców, wywołując *butler.passForks(leftFork, rightFork).*

- Jeśli lokaj wyrazi zgodę (warunki w metodzie *passForks* zostaną spełnione), filozof je, zwiększa licznik counter.
- Następnie filozof odkłada oba widelce, wywołując *butler.takeForks(leftFork, rightFork).*

Klasa Fil5mon to główna klasa inicjalizująca i uruchamiająca symulację:

Tworzymy instancję lokaja oraz tablicę pięciu widelców i tablicę pięciu filozofów. Każdy filozof otrzymuje referencję do butler, liczbę iteracji, oraz przypisane widelce (lewy i prawy).Następnie wątki filozofów są uruchamiane, a program oczekuje na ich zakończenie.

Dzięki wprowadzeniu lokaja rozwiązanie to zapobiega zakleszczeniu, ponieważ nigdy więcej niż czterech filozofów nie może jednocześnie mieć podniesionych widelców. Lokaj kontroluje dostęp i czeka, gdy zajętych jest 4 widelce, co gwarantuje, że zawsze pozostanie jeden wolny widelec, eliminując możliwość deadlocku.

Poniżej znajduje się implementacja rozwiązania problemu pięciu filozofów z zastosowaniem lokaja:

```
public class Butler {
    private int takenForkNumber = 0;



    public synchronized void passForks(Fork leftFork, Fork rightFork){
        while(!leftFork.isAvailable() || !rightFork.isAvailable()){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        while(takenForkNumber == 4){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        takenForkNumber += 2;
        leftFork.take();
        rightFork.take();
    }

    public synchronized void takeForks(Fork leftFork, Fork rightFork){
        takenForkNumber -= 2;
        leftFork.putDown();
        rightFork.putDown();
        notifyAll();
    }


}
```

```
public class Fork {
    private boolean isAvailable = true;


    public synchronized void take() {
        isAvailable = false;

    }
    public synchronized void putDown() {
        isAvailable = true;

    }

    public boolean isAvailable() {
        return isAvailable;
    }
}
```

```
public class Philosopher extends Thread {
    private int counter = 0;
    private Fork leftFork;
    private Fork rightFork;
    private Butler butler;
    private int iterations_number;


    public Philosopher(Butler butler, int iterations_number) {
        this.butler = butler;
        this.iterations_number = iterations_number;
    }

    public void run() {
        long startTime = System.currentTimeMillis();
        while (true) {

            // jedzenie

            this.butler.passForks(this.leftFork, this.rightFork);

            ++counter;

            this.butler.takeForks(this.leftFork, this.rightFork);

            if (counter == iterations_number) {
                long endTime   = System.currentTimeMillis();
                long elapsedTime = endTime - startTime;
                System.out.println("Filozof: " + Thread.currentThread() +
"jadlem " + counter + " razy" + " czas:" +elapsedTime);
                break;

            }

        }
    }

    public void setLeftFork(Fork leftFork) {
        this.leftFork = leftFork;
    }

    public void setRightFork(Fork rightFork) {
```

```
        this.rightFork = rightFork;
    }
}
```

```
public class Fil5mon {

    public static void main(String[] args) {
        int iterationsNumber = 10_000_000;
        Butler butler =  new Butler();
        Fork[] forks = new Fork[5];
        for (int i = 0; i < 5; i++) {
            forks[i] = new Fork();
        }


        Philosopher[] philosophers = new Philosopher[5];
        for (int i = 0; i < 5; i++) {
            philosophers[i] = new Philosopher(butler, iterationsNumber);
            philosophers[i].setLeftFork(forks[i]);
            philosophers[i].setRightFork(forks[(i+1)%5]);


        }

        for (int i = 0; i < 5; i++) {
            philosophers[i].start();
        }

        for(int i = 0; i < 5; i++){
            try {
                philosophers[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

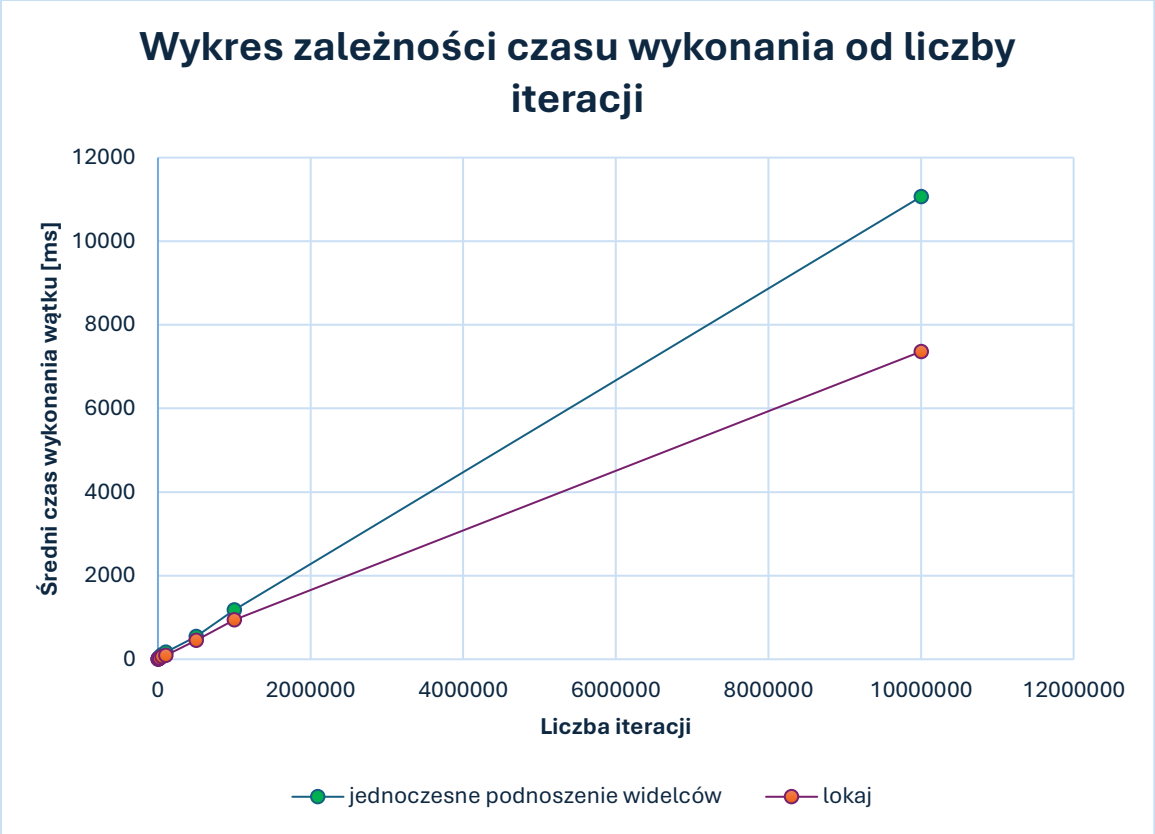
d)

Przeprowadzono eksperyment, w którym zmierzono czas w wykonania programu dla poszczególnych wątków przy zadanej liczbie iteracji. Wyniki oraz ich reprezentacja graficzna zostały przedstawione poniżej.

Czas ewaluacji dla poszczególnych wątków						*Uśredniony czas ewaluacji*
Wątek 1	*Wątek 2*	*Wątek 3*	*Wątek 4*	*Wątek 5*	*liczba iteracji*	*Implementacja z jedoczesnym podnoszeniem widelców*
4	4	7	1	4	1000	4
14	5	17	18	9	5000	12,6
18	19	19	15	6	10000	15,4
59	60	59	45	55	20000	55,6

112	99	90	123	123	50000	109,4
168	171	164	160	164	100000	165,4
492	546	552	555	563	500000	541,6
1087	1186	1200	1211	1217	1000000	1180,2
10886	11002	11094	11156	11181	10000000	11063,8

Czas ewaluacji dla poszczególnych wątków						*Uśredniony czas ewaluacji*
Wątek 1	*Wątek 2*	*Wątek 3*	*Wątek 4*	*Wątek 5*	*liczba iteracji*	*Implementacja z zastosowaniem lokaja*
4	3	3	1	4	1000	3
10	9	10	6	10	5000	9
16	12	14	16	13	10000	14,2
23	20	30	17	29	20000	23,8
62	50	52	67	68	50000	59,8
98	92	82	95	108	100000	95
380	425	474	480	487	500000	449,2
875	924	970	975	979	1000000	944,6
5644	7400	7621	7982	8152	10000000	7359,8



Analiza wykresu

1. **Implementacja z jednoczesnym podnoszeniem widelców**:
    - Linie dla tej implementacji rosną szybciej, co oznacza, że czas wykonania wątku rośnie wykładniczo wraz ze wzrostem liczby iteracji.
    - Przy bardzo dużych liczbach iteracji (np. 10 milionów) wykres pokazuje, że czas wykonania jest znacznie dłuższy w porównaniu z implementacją z lokajem.
    - Taki wzrost sugeruje, że jednoczesne podnoszenie widelców powoduje znaczne obciążenie przy dużej liczbie iteracji, co prowadzi do wyższej konkurencji między wątkami i większego wpływu na czas przetwarzania.
2. **Implementacja z lokajem**:
    - Wzrost czasu wykonania w tej implementacji jest bardziej liniowy, co wskazuje na lepszą skalowalność.
    - Przy większych liczbach iteracji czas wykonania rośnie wolniej niż w przypadku jednoczesnego podnoszenia widelców, co sugeruje, że implementacja z lokajem lepiej zarządza zasobami.
    - Niższy czas wykonania dla dużych wartości iteracji pokazuje, że rozwiązanie z lokajem jest bardziej efektywne przy długotrwałych zadaniach, dzięki ograniczeniu liczby jednocześnie jedzących filozofów. W ten sposób unika się zbyt dużej konkurencji o zasoby.

Wnioski

- Jednoczesne podnoszenie widelców jest efektywne przy mniejszych liczbach iteracji, ale przy większych liczbach powoduje gwałtowny wzrost czasu przetwarzania ze względu na problemy związane z synchronizacją między filozofami.
- Implementacja z lokajem jest bardziej wydajna niż implementacja z równoczesnym podnoszeniem widelców , co staje się szczególnie odczuwalne przy większych liczbach iteracji, ponieważ lepiej kontroluje liczbę jednocześnie korzystających z widelców wątków, co zmniejsza ryzyko blokady i pozwala na bardziej równomierny dostęp do zasobów.

Podsumowując, rozwiązanie z lokajem jest bardziej wydajne i stabilne przy większej liczbie iteracji, ponieważ ogranicza liczbę aktywnych filozofów, redukując czas oczekiwania i minimalizując ryzyko głodzenia. W przypadku krótkotrwałych zadań lub

mniejszej liczby iteracji rozwiązanie z jednoczesnym podnoszeniem widelców może być wystarczająco efektywne.

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://docs.oracle.com/en/java/javase/19/index.html
- https://en.wikipedia.org/wiki/Dining_philosophers_problem
- https://nscpolteksby.ac.id/ebook/files/Ebook/Computer%20Engineering/Using%20Concurrency%20and%20Parallelism%20Effectively-I%20(2014)/13.%20Chapter%2012%20-%20Dining%20Philosophers%20A%20Classic%20Problem.pdf