Teoria Współbieżności

Laboratorium 6 - Problem ucztujących filozofów.

Magdalena Pastuła

Data laboratorium: 10.11.2020

1. Zadanie do wykonania.

Na laboratorium należało zaimplementować problem pięciu filozofów w trzech wariantach:

- rozwiązanie symetryczne
- rozwiązanie, w którym dwa widelce są podnoszone jednocześnie
- rozwiązanie przy pomocy kelnera

2. Koncepcja rozwiązania.

2.1 Symetryczni filozofowie.

W celu rozwiązania problemu filozofów symetrycznie utworzoną dodatkową klasę `EatersCounter` oprócz klas `Fork` i `Philosophers`, która zlicza jedzących filozofów. Przed podniesieniem widelców filozowofie zwiększają licznik, który jest zabezpieczony semaforem. Filozofowie mogą podnieść swoje widelce tylko wtedy, gdy liczba jedzących jest mniejsza niż połowa wszystkich filozofów, w przeciwnym wypadku czekają, aż ten warunek nie zostanie spełniony.

2.2 Jednoczesne podnoszenie dwóch widelców.

Podnoszenie dwóch widelców jednocześnie zostało zaimplementowanie dodatkowego, głównego semafora. Aby filozof mógł podnieść swoje widelce, musi najpierw zająć ten semafor. Następnie sprawdza, czy oba jego widelce są wolne, jeżeli tak, to je podnosi i zwalnia główny semafor, w przeciwnym wypadku zajmuje i oddaje główny semafor dopóki widelce nie będą wolne. Sprawdzanie, czy widelec jest zajęty, zaimplementowano poprzez dodanie pola `isPickedUp` do klasy `Fork`.

2.3 Kelner.

W rozwiązaniu przy użyciu kelnera zaimplementowano dodatkową klasę `Butler`. Filozof, w momencie gdy chce zjeść, najpierw pyta kelnera czy może. Jeżeli tak, podnosi widelce, w przeciwnym wypadku czeka na pozwolenie. Po zjedzeniu filozof informuje kelnera, że widelce zostały zwolnione.
Sama klasa `Butler` zawiera w sobie tablicę booleanów, gdzie przechowywana jest informacja, które widelce są zajęte, a także semafor na dostęp do tej tablicy.

3. Implementacja i wyniki.

Wszystkie pliki źródłowe znajdują się w załączonym archiwum.

3.1 Symetryczni filozofowie.

Implementacja klasy `EatersCounter`:

```
public class EatersCounter {
    private int eating_no = 0;
    private Semaphore _sem = new Semaphore(1);
    private int _philosophers_no;

    public EatersCounter(int philosophers_no) {
        _philosophers_no = philosophers_no;
    }

    void increase()
    {
        try
        {
            _sem.acquire();
            eating_no++;
            _sem.release();
        } catch (Exception e) {}
    }

    void decrease()
    {
        try
        {
            _sem.acquire();
            eating_no--;
            _sem.release();
        } catch (Exception e) {}
    }

    boolean canIEat()
    {
        boolean toReturn = false;
        try
        {
            _sem.acquire();
            toReturn = (eating_no < _philosophers_no/2.0f);
            _sem.release();
        } catch (Exception e) {}

        return toReturn;
    }
}
```

Dodatkowo, poniżej znajduje się początek logów z wykonania programu. Dla ułatwienia każdy filozof wypisuje logi w innym kolorze.

```
> Task :Fil5mon.main()
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Philosopher Thread[Thread-1,5,main]: I am starting to eat.
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-1,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-0,5,main]: I am starting to eat.
Philosopher Thread[Thread-4,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I am starting to eat.
Philosopher Thread[Thread-3,5,main]: Waiting for forks.
Philosopher Thread[Thread-0,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-4,5,main]: I am starting to eat.
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-2,5,main]: Waiting for forks.
Philosopher Thread[Thread-4,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-3,5,main]: I am starting to eat.
Philosopher Thread[Thread-0,5,main]: I am starting to eat.
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Philosopher Thread[Thread-4,5,main]: Waiting for forks.
Philosopher Thread[Thread-0,5,main]: I have eaten 2 time(s).
Philosopher Thread[Thread-3,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-4,5,main]: I am starting to eat.
Philosopher Thread[Thread-2,5,main]: I am starting to eat.
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I have eaten 2 time(s).
Philosopher Thread[Thread-1,5,main]: I am starting to eat.
Philosopher Thread[Thread-1,5,main]: I have eaten 2 time(s).
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Philosopher Thread[Thread-1,5,main]: I am starting to eat.
Philosopher Thread[Thread-1,5,main]: I have eaten 3 time(s).
Philosopher Thread[Thread-4,5,main]: I have eaten 2 time(s).
```

Zmierzony czas wykonania programu to 1.130 s przy założeniu, że każdy filozof próbuje zjeść dokładnie 10 razy.

3.2 Jednoczesne podnoszenie dwóch widelców.

Implementacja klasy `Philosopher`:

```
public class Philosopher extends Thread {
    private final Random gen = new Random();
    private Semaphore _mainSemaphore;
    private Fork _left;
    private Fork _right;
    private String _logsColor;

    public static final String ANSI_RESET = "\u001B[0m";

    public Philosopher(Fork left, Fork right, Semaphore mainSemaphore, String
logsColor)
    {
        _left = left;
        _right = right;
        _mainSemaphore = mainSemaphore;
        _logsColor = logsColor;
    }

    public void run()
    {
        for (int i=0;i<10;i++) {
            //think
            try {
                sleep(gen.nextInt(100));
            } catch (Exception e) {;}

            System.out.println(_logsColor + "Philosopher " +
Thread.currentThread() +
                    ": Waiting for forks." + ANSI_RESET);
            try {
                _mainSemaphore.acquire();
                while (_left.isPickedUp() || _right.isPickedUp())
                {
                    _mainSemaphore.release();
                    sleep(gen.nextInt(3));
                    _mainSemaphore.acquire();
                }
                _left.pick_up();
                _right.pick_up();
                _mainSemaphore.release();
            } catch (Exception e) {;}
            // eat
            System.out.println(_logsColor + "Philosopher " +
Thread.currentThread() +
                                ": I am starting to eat." + ANSI_RESET);
            try {
                sleep(gen.nextInt(50));
            } catch (Exception e) {;}

            // koniec jedzenia
            try {
                _mainSemaphore.acquire();
                _left.put_down();
```

```
                _right.put_down();
                _mainSemaphore.release();
            } catch (Exception e) {;}
            System.out.println(_logsColor + "Philosopher " +
Thread.currentThread() +
                    ": I have eaten " + (i+1) + " time(s)." + ANSI_RESET);
        }
    }
}
```

Dodatkowo, jak dla poprzedniego przykładu, poniżej znajduje się wycinek wypisywanych komunikatów.

```
> Task :Fil5mon.main()
Philosopher Thread[Thread-2,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I am starting to eat.
Philosopher Thread[Thread-3,5,main]: Waiting for forks.
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-0,5,main]: I am starting to eat.
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Philosopher Thread[Thread-4,5,main]: Waiting for forks.
Philosopher Thread[Thread-3,5,main]: I am starting to eat.
Philosopher Thread[Thread-2,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-3,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-2,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I am starting to eat.
Philosopher Thread[Thread-0,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-4,5,main]: I am starting to eat.
Philosopher Thread[Thread-4,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-2,5,main]: I have eaten 2 time(s).
Philosopher Thread[Thread-1,5,main]: I am starting to eat.
Philosopher Thread[Thread-3,5,main]: Waiting for forks.
Philosopher Thread[Thread-3,5,main]: I am starting to eat.
Philosopher Thread[Thread-3,5,main]: I have eaten 2 time(s).
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-1,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-0,5,main]: I am starting to eat.
Philosopher Thread[Thread-4,5,main]: Waiting for forks.
Philosopher Thread[Thread-0,5,main]: I have eaten 2 time(s).
```

Zmierzony czas wywołania programu to 0.938 ms. Jak poprzednio, każdy filozof jadł dokładnie 10 razy.

3.3 Kelner.

Implementacja klasy `Butler`:

```
public class Butler {
    private int _forks_no;
    private Fork [] _forks;
    private boolean [] _pickedUpForks;
    private Semaphore _semaphore = new Semaphore(1);

    public Butler(int forks_no, Fork [] forks)
    {
        _forks_no = forks_no;
        _forks = forks;
        _pickedUpForks = new boolean[forks_no];

        for (int i=0;i<forks_no;i++)
        {
            _pickedUpForks[i] = false;
        }
    }

    public boolean canIEat(int philosopherID)
    {
        boolean toReturn = false;
        try {
            while (!_semaphore.tryAcquire());

            if (!_pickedUpForks[philosopherID] &&
!_pickedUpForks[(philosopherID+1)%_forks_no])
            {
                _pickedUpForks[philosopherID] = true;
                _pickedUpForks[(philosopherID+1)%_forks_no] = true;
                toReturn = true;
                System.out.println("Butler: Philosopher with id " + philosopherID
+ " will start to eat.");
            }
            else
            {
                toReturn = false;
            }
            _semaphore.release();
        } catch (Exception e) {;}

        return toReturn;
    }

    public void returnForks(int philosopherID)
    {
        try {
            _semaphore.acquire();

            _pickedUpForks[philosopherID] = false;
            _pickedUpForks[(philosopherID+1)%_forks_no] = false;
```

```
            _semaphore.release();
            System.out.println("Butler: Philosopher with id " + philosopherID + "
returned forks.");
        } catch (Exception e) {;}
    }
}
```

Ponownie poniżej znajduje się częśc komunikatów wypisanych podczas wykonywania programu. Dodatkowo, w tym podpunkcie również kelner wypisuje komunikaty, którym filozofow dał lub zabrał widelce.

```
> Task :Fil5mon.main()
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Butler: Philosopher with id 1 will start to eat.
Philosopher Thread[Thread-1,5,main]: I am starting to eat.
Philosopher Thread[Thread-4,5,main]: Waiting for forks.
Butler: Philosopher with id 4 will start to eat.
Philosopher Thread[Thread-4,5,main]: I am starting to eat.
Philosopher Thread[Thread-2,5,main]: Waiting for forks.
Philosopher Thread[Thread-3,5,main]: Waiting for forks.
Butler: Philosopher with id 1 returned forks.
Philosopher Thread[Thread-1,5,main]: I have eaten 1 time(s).
Butler: Philosopher with id 2 will start to eat.
Philosopher Thread[Thread-0,5,main]: Waiting for forks.
Philosopher Thread[Thread-2,5,main]: I am starting to eat.
Butler: Philosopher with id 4 returned forks.
Philosopher Thread[Thread-4,5,main]: I have eaten 1 time(s).
Butler: Philosopher with id 0 will start to eat.
Philosopher Thread[Thread-0,5,main]: I am starting to eat.
Butler: Philosopher with id 2 returned forks.
Philosopher Thread[Thread-2,5,main]: I have eaten 1 time(s).
Butler: Philosopher with id 3 will start to eat.
Philosopher Thread[Thread-3,5,main]: I am starting to eat.
Butler: Philosopher with id 3 returned forks.
Philosopher Thread[Thread-3,5,main]: I have eaten 1 time(s).
Philosopher Thread[Thread-1,5,main]: Waiting for forks.
Butler: Philosopher with id 0 returned forks.
Butler: Philosopher with id 1 will start to eat.
Philosopher Thread[Thread-0,5,main]: I have eaten 1 time(s).
```

Zmierzony czas wywołania programu to 0.943. Analogicznie, jak w poprzednich podpunktach, każdy filozof jadł dokładnie 10 razy.

4. Wnioski z ćwiczenia.

4.1 Symetryczni filozofowie.

Mechanizm symetrycznych filozofów nie broni całkowicie przed zakleszczeniem. Może się zdarzyć sytuacja, że dwóch filozofów obok siebie będzie chciało się posilić i przy takim rozwiązniu problemu jeden z nich będzie czekał aż drugi odda widelec. Przykładem tutaj może być filozof numer 3 (niebieski) i 4 (fioletowy) w wypisanych komunikatach. Widać, że filozof numer 3 czeka aż to jego sąsiad odda widelec, a nie jakikolwiek jedzący.
Pomimo to, ten mechanizm broni przed sytuacją, gdy każdy z filozofów, poza jedzącym, czeka na sąsiada. Zatem maksymalna liczba czekających to dopuszczona liczba filozofów przez `EatersCounter` pomniejszona o jeden.

Na zwiększenie czasu wykonania tego programu wpływają głównie sytuacje, gdy dopuszczeni do jedzenia zostają sąsiedzi. Wtedy, mimo ograniczenia na liczbę filozofów, jeden z nich czeka, podczas gdy jakiś inny niedopuszczony przez licznik mógłby zjeść.

4.2 Jednoczesne podnoszenie dwóch widelców.

Podobnie, jak w poprzednim podpunkcie, w przypadku jednoczesnego podnoszenia widelców może zaistnieć sytuacja, że filozof czeka na jego zwolnienie przez sąsiada. Sytuację tą widać już w drugiej linijce w komunikatach: filozof numer cztery czeka na zwolnenie zasobu przez filozofa numer 0.

Dodatkowo, w trakcie rozwiązywania tego problemu, natknęłam się na sytuację zakleszczenia. W trakcie, gdy jeden z filozofów jadł, drugi również zdecydował sie posilić, zatem zajął główny semafor. Jednakże musiał czekać na zwolnienie widelca przez swojego sąsiada, a ten natomiast, po zjedzeniu czekał na zwolnienie głównego semafora. Dodanie do klasy `Fork` pola `isPickedUp`, do którego jest dostęp po zajęciu głównego semafora, pozwoliło na rozwiązanie tego problemu.

Analogicznie, jak poprzednio, głównym czynnikiem zwiększającym czas wykonania programu jest sytuacja, gdy sąsiadujący ze sobą filozofowie zdecydują się zjeść. Jednakże, w przypadku, gdy jakiś trzeci filozof również zdecyduje się zjeść, to o ile nie jest drugim sąsiadem pierwszego, to będzie mógł zjeść. Prawdopodobnie dlatego czas wykonania tego programu jest szybszy niż poprzedniego.

4.3 Kelner.

Analogicznie, jak w poprzednich podpunktach, w przypadku rozwiazania z kelnerem również mamy sytuację, że filozof może czekać na zwolnienie widelca przez sąsiada. Natomiast niemożliwa jest sytuacja zakleszczenia.

Czas wykonania tego wariantu zależy od tych samych czynników, co poprzedniego. Tutaj również, jezeli jeden z filozofów czeka na zwolnienie widelca, to nie blokuje drugiego, zatem widelce są blokowane tylko przez faktycznie jedzących. Dlatego również dla tego programu czas wykonania jest mniejszy niż w wariancie z symetrycznymi filozofami.

5. Bibliografia.

Opis problemu filozofów