Sprawozdanie z zadania nr 1
Problem pięciu filozofów

Mateusz Nowak

Kraków, 9 listopada 2025.

1 Opis ćwiczenia

Problem pięciu filozofów to klasyczny problem z zakresu programowania współbieżnego. Polega on na tym, że pięciu (a w ogólności N) filozofów siedzi przy okrągłym stole, i pomiędzy każdymi dwoma leży widelec. Działanie filozofa to cztery powtarzające się kroki: myślenie, próba podniesienia widelców, jedzenie, odłożenie widelców. Filozof może jeść, kiedy podniesie oba "swoje" widelce.
Problemem w tym zadaniu jest oczywiście technika przydzielania współdzielonych zasobów (widelców) procesom (filozofom). Poniżej w tym sprawozdaniu opiszę 6 metod, które udało mi się zaimplementować, korzystających z różnych mechanizmów Javy - synchronized, Reentrant Lock, semafory. Każdą z metod przetestowałem dla N = {5, 10, 15, 20} filozofów.

2 Struktura projektu i opisy poszczególnych rozwiązań

Na początku stworzyłem klasy **Fork** (widelca) i **Philosopher** (filozofa). Klasa **Fork** zaimplementowana jest zarówno dla wariantu synchronized z metodami *wait()*, *notifyAll()*, jak i dla wariantu z Reentrant Lockiem. O tym, z którego mechanizmu korzystamy decyduje pole *boolean withLock*.
Klasa **Philosopher** jest klasą abstrakcyjną, zawierającą zachowanie i ogólne metody działania filozofa - *think()*, *eat()* - te dwie metody, w mojej implementacji, zajmują filozofowi losowo od 0 do 1000 milisekund, *pickForks()* (metoda abstrakcyjna - ponieważ dla każdego wariantu będzie inna technika podnoszenia widelców, jej ciało jest w klasach dziedziczących), *putForks()*. Zawiera również pola *timeWaited* i *eatenCounter*, które będą przydatne do sporządzania statystyk symulacji.
Po klasie **Philosopher** dziedziczą klasy poszczególnych typów filozofów, które poniżej omówię.

2.1 Rozwiązanie naiwne

Każdy filozof czeka, aż wolny będzie lewy widelec, a następnie go podnosi (zajmuje), następnie podobnie postępuje z prawym widelcem.
W tym przypadku skorzystałem z mechanizmu synchronized - filozof podnosi widelec, kiedy jest on wolny lub kiedy dostanie *notifyAll()* powiadamiające, że się właśnie zwalnia. Dla ustalonego (nielosowego) czasu jedzenia i myślenia, następował deadlock, ponieważ każdy z filozofów podniósł jeden widelec i czekał w nieskończoność.

2.2 Rozwiązanie z możliwością zagłodzenia

Każdy filozof sprawdza czy oba sąsiednie widelce są wolne i dopiero wtedy zajmuje je jednocześnie. Rozwiązanie to jest wolne od blokady, jednak w przypadku, gdy zawsze któryś z sąsiadów będzie zajęty jedzeniem, nastąpi zagłodzenie, gdyż oba widelce nigdy nie będą wolne.
Do tego przykładu użyłem mutexa (Reentrant Locka). Małym problemem był fakt, że "zajęcie obu widelców jednocześnie" nie jest operacją atomową, więc w mojej implementacji filozof zajmował po kolei mutex lewego i prawego widelca ale w przypadku, gdy prawy był zajęty, oddawał mutex lewego i rozpoczynał od nowa.

2.3 Rozwiązanie asymetryczne

Filozofowie są ponumerowani. Filozof z parzystym numerem najpierw podnosi prawy widelec, filozof z nieparzystym numerem najpierw podnosi lewy widelec.
Podobnie, jak w pierwszym przykładzie, użyłem synchronized. Jest to ideowo podobny wariant, lecz działający lepiej dzięki wprowadzeniu asymetrii.

2.4 Rozwiązanie stochastyczne

Każdy filozof rzuca monetą tuż przed podniesieniem widelców i w ten sposób decyduje, który najpierw podnieść - lewy czy prawy.
W tym przypadku skorzystałem z Reentrant Locka, a do losowania użyłem metody *Random.nextInt(2)*.

2.5 Rozwiązanie z arbitrem

Zewnętrzny arbiter (lokaj, kelner) pilnuje, aby jednocześnie co najwyżej czterech (w ogólnym przypadku N-1) filozofów konkurowało o widelce. Każdy podnosi najpierw lewy a potem prawy widelec. Jeśli naraz wszyscy filozofowie będą chcieli jeść, arbiter powstrzymuje jednego z nich aż do czasu, gdy któryś z filozofów skończy jeść.
W tym przypadku użyłem semafora o rozmiarze *N - 1*. W metodzie *pickForks()*, przed próbą podniesienia widelców, przydzielany jest zasób semafora poprzez *semaphore.acquire()*. W przypadku, kiedy już nie ma miejsca, filozof czeka aż się zasób zwolni, czyli czeka, aż inny filozof wywoła *semaphore.release()* po swoim posiłku.

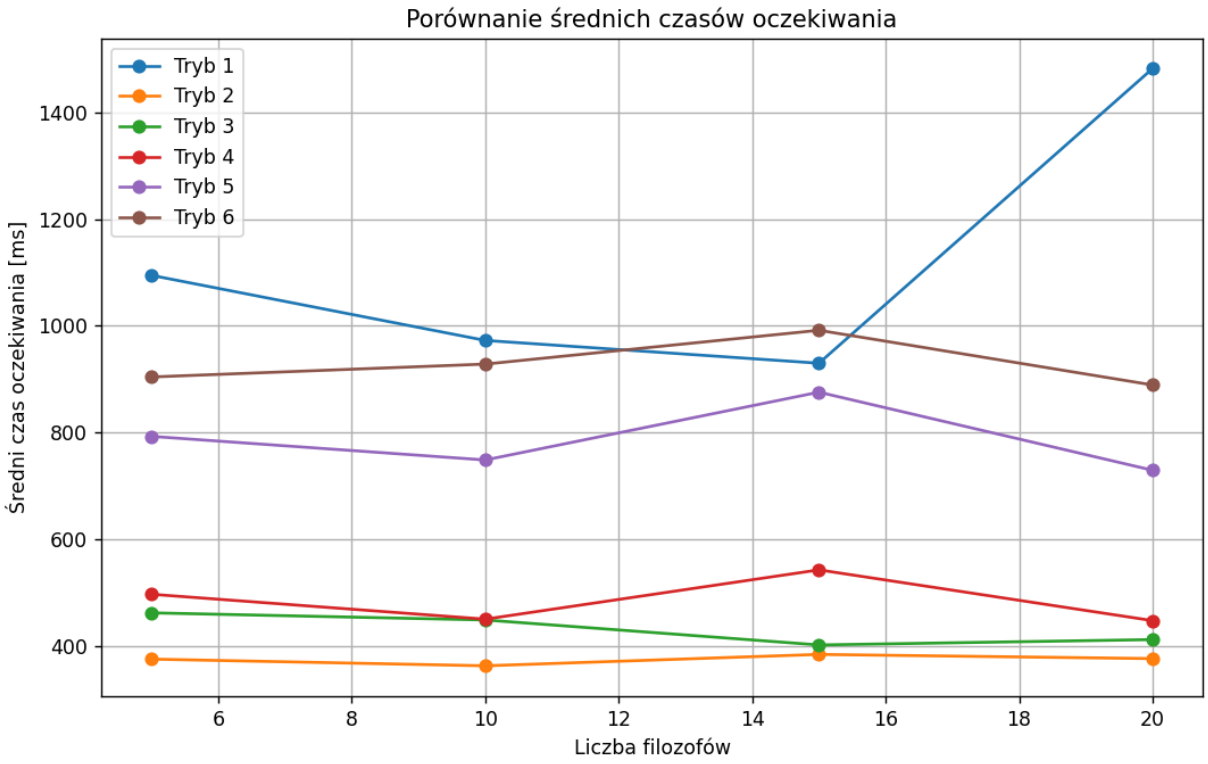
2.6 Rozwiązanie z jadalnią

Rozwiązanie jest modyfikacją wersji z arbitrem. Filozof, który nie zmieści się w jadalni (czyli arbiter nie pozwolił mu jeść) je „na korytarzu" podnosząc jednorazowo widelce w odwrotnej kolejności (do reszty filozofów w jadalni).
Rozwiązanie podobne do poprzedniego, w tym także użyłem semafora o rozmiarze *N - 1*. W tym przypadku jednak, filozof wywołuje *semaphore.tryAcquire()*, co jeśli zwróci prawdę, oznacza, że ma miejsce przy stole i próbuje wziąć najpierw lewy, a potem prawy widelec, natomiast, jeśli zwróci fałsz, to próbuje wziąć widelce w odwrotnej kolejności.

2.7 Klasa Simulation i Main

W klasie **Main** wywołałem symulację dla każdego wariantu i dla czterech różnych wartości N - 5, 10, 15 i 20. Klasa **Simulation** odpowiadała za całą logikę symulacji - tworzenie filozofów, widelców odpowiednio z mutexem lub bez, semaforów do dwóch ostatnich wariantów, uruchomienie wątków filozofów i, na koniec, zapis otrzymanych statystyk do pliku csv.

3 Wyniki



Rysunek 1: Wykresy

W powyższych pomiarach nie dochodziło do zagłodzenia, ani, dzięki losowości czasu myślenia i jedzenia, zakleszczenia. Najkrótszy średni czas oczekiwania na dostęp do zasobu oferowały tryby 2, 3 i 4 - czyli rozwiązania z możliwością zagłodzenia, asymetryczne i stochastyczne. Rozwiązania z arbitrem, czyli semaforem pozwalały na mniejszą liczbę zjedzonych posiłków, czyli dłuższy średni czas oczekiwania. Wyróżnia się także tryb 1, rozwiązanie naiwne z najgorszym średnim czasem, gdzie w niektórych wywołaniach (jak na rysunku dla N = 20) ten czas bardzo rósł.
Dla niektórych metod, głównie drugiej, odnotowałem również zjawisko zagłodzenia, charakteryzujące się niskim, odstającym od reszty, lub nawet zerowym wynikiem pola *eatenCounter* u któregoś filozofa. Wyniki bywały również inne przy ustalonym czasie myślenia i jedzenia - wtedy na przykład w pierwszym wariancie od razu dochodziło do deadlocka - wszyscy podnieśli lewy widelec i się wzajemnie zablokowali.

```
> Task :org.example.Main.main()
Philosopher 5 is thinking
Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 3 picked up the left.
Philosopher 5 picked up the left.
Philosopher 1 picked up the left.
Philosopher 4 picked up the left.
Philosopher 2 picked up the left.
```

Rysunek 2: Logi z trybu 1

4 Wnioski

Wszystkie zaimplementwane rozwiązania w pewien sposób działają, jednak mają swoje wady - naiwne prowadzi do zakleszczenia przy ustalonym czasie, drugie może prowadzić do zagłodzenia. Asymetryczne i stochastyczne działa, lecz dawało często nierówne wyniki jeśli chodzi o liczbę zjedzonych posiłków. Wykorzystanie arbitra w dwóch ostatnich rozwiązaniach zapobiegło tym problemom, dawało stabilne i równomierne (sprawiedliwe) wyniki, chociaż pod kątem średniego czasu oczekiwania trochę odstawało.