Teoria współbieżności - problem producenta-konsumenta

Paweł Podedworny, grupa nr 11

12.11.2024

1 Opis omawianego zagadnienia

Problem producenta-konsumenta to klasyczny problem synchronizacji wątków, często spotykany w programach wielowątkowych. Dotyczy on dwóch głównych działań: produkcji zasobów przez wątek producenta oraz ich konsumpcji przez wątek konsumenta, z wykorzystaniem wspólnego bufora. Kluczowy problem polega na zapewnieniu poprawnego zapisywania i odczytywania danych z bufora – błędna obsługa tych operacji może prowadzić do straty danych i nieprawidłowego działania programu. Istnieje kilka podejść do rozwiązania tego problemu, lecz wiele z nich może prowadzić do zagłodzenia lub zakleszczenia wątków. Dlatego porównnamy ze sobą trzy rozwiązania: z wykorzystaniem dwóch zmiennych warunkowych (rozwiązanie zagładzające), czterech zmiennych warunkowych oraz trzech lock'ów. Celem jest pokazanie różnic między prostszym rozwiązaniem, które jednak zagłodzenie wywołuje, a bardziej skomplikowanymi, które powinny działać bez zakłóceń.

2 Opis implementacji rozwiązań niezagładzających

Sposób implementacji z wykorzystaniem czterech zmiennych warunkowych polega na użyciu jednego warunku dla pierwszego oczekującego wątku, drugiego warunku dla pozostałych wątków i jednej zmiennej typu boolean w celu sprawdzania braku oczekującego na swoją kolej pierwszego wątku dla operacji konsumpcji i produkcji. Dzięki temu upewniamy się, że wątek, który jako pierwszy wszedł do monitora i nie mógł wykonać w danym momencie swojej operacji, w przyszłości będzie pierwszy obsłużony. Wykorzystujemy tutaj zmienną boolean, a nie metodę hasWaiters(), ponieważ po analizie jej działania wykazaliśmy, że w odpowiednim momencie zwraca wartość niepożądaną, dla naszego zamysłu implementacji. Po dalszej analizie doszliśmy do kolejnego wniosku, że nawet tak rozbudowane rozwiązanie dalej może wykazywać elementy zagładzania, ponieważ przez nieuporządkowane wchodzenie wątków do monitora, zasoby może dostać wątek nie ten, który w teorii powinien. W takim wypadku, aby całkowicie pozbyć się opisanej możliwości, powinniśmy użyć po dwie zmienne typu boolean na daną operację, jednakże w dalszej części testów korzystam z pierwszego rozwiązania. Dodatkowo podczas tworzenia instancji Lock'a podaję flagę fair, ale jest to dokładniej wyjaśnione w punkcie 4.
Sposób implementacji rozwizania z wykorzystaniem 3 lock'ów polega za zagnieżdzeniu jednego monitora wewnątrz dwóch pozostałych. Producenci i konsumenci mają po monitorze wejściowym, z którego następnie wchodzą już do monitora głównego, jeżeli będzie akurat zwolniony. W nim dopiero mogą dane pobierać lub dokładać. Jak okaże się, że nie mogą wykonać operacji na buforze, zwalniają monitor główny, zawieszając się na warunku, jednakże wejściowy przez cały czas oczekiwania jest zajęty. Takie podejście sprawia, że nawet jeżeli wątek nie mógł w danym momencie wykonać swojej operacji, na pewno wykona ją pierwszy jak tylko to będzie możliwe.

3 Dane techniczne

Komputer z systemem Windows 10 x64
Procesor: AMD Ryzen 5 3600 3.60GHz - 6 rdzeni fizycznych
Pamięć RAM: 32GB 3000MHz
Środowisko: IntelliJ IDEA 2024.2.4
Język: Java 17.0.9

4 Wyniki z użyciem 4 zmiennych warunkowych

Podczas implementacji rozwiązania korzystającego łącznie z czterech zmiennych warunkowych oraz dwóch zmiennych typu boolean, zauważyłem, że otrzymane wyniki działania wątków różnią się od dwóch pozostałych badanych rozwiązań. Do pokazania dziwnego zjawiska posłużę się różnicą pomiędzy minimalną a maksymanlną liczbą operacji w monitorze oraz średnim najmniejszym czasem a największym spędzonym na oczekiwaniu danego wątku. Test został wykonany na buforze o rozmiarze 100, 3 wątkach producentów i 3 wątkach konsumentów oraz losowym wybiorze liczby danych z ziarnem generowania "42" w czasie 60 sekund.

Sposób	Liczba operacji			Średni czas oczekiwania [ns]		
	Min.	Max.	Różnica	Min.	Max.	Różnica
4 zmienne warunkowe	1.430.556	1.850.882	420.326	32.370	41.895	9.525
2 zmienne warunkowe	2.512.761	2.529.692	16.931	23.675	23.834	159
3 lock'i	4.268.857	4.358.842	89.985	13.708	13.997	289

Tabela 1: Wyniki pomiarów dla 3 konsumentów i 3 producentów dla wszystkich omawianych sposobów synchronizacji wykonane w czasie 60 sekund

Aby lepiej zobrazować występującą rozbieżność wyników, wykonałem dodatkowy test dla dziesięciu konsumentów oraz dziesięciu producentów:

Sposób	Liczba operacji			Średni czas oczekiwania [ns]		
	Min.	Max.	Różnica	Min.	Max.	Różnica
4 zmienne warunkowe	110.784	700.712	589.928	85.605	541.689	456.084
2 zmienne warunkowe	775.806	786.874	11.068	76.204	77.292	1.088
3 lock'i	1.206.400	1.395.586	189.186	42.950	49.690	6.740

Tabela 2: Wyniki pomiarów dla 10 konsumentów i 10 producentów dla wszystkich omawianych sposobów synchronizacji wykonane w czasie 60 sekund

Jak możemy odczytać z tabeli 1 oraz tabeli 2 różnice pomiedzy liczbą operacji oraz średnim czasem oczekiwania dla rozwiązania wykorzystującego cztery zmienne warunkowe, uzyskują bardzo duże wartości różnicy bezwględnej. Jeszcze bardziej stan ten pogarsza fakt, że różnica ta jest o wiele większa w stosunku do mniejszej liczby wykonanych operacji, jaką np. wykazuje rozwiązanie na 3 lock'ach. Po analizie problemu, stwierdziłem, że w kolejnych testach posłużę się flagą *fair* przekazywaną do konstruktora Lock'a w tym sposobie. Rozwiązanie to sprawia, że program będzie wybierał zawsze wątki czekające najdłużej. Nie doszedłem do powodu, dlaczego akurat w tej metodzie, nieskorzystanie z flagi powoduje aż tak duże rozbieżności. Wydaje się, że wprowadzanie dodatkowych metod synchronizacji, takie jak 4 zmienne warunkowe oraz zmienne boolean, w połączeniu z losowym doborem wątków, może sprawiać, że opisane w punkcie 2 możliwe wystąpienie zagłodzenia dosadnie się uwypukla.

Liczba wątków	Liczba operacji			Średni czas oczekiwania [ns]		
	Min.	Max.	Różnica	Min.	Max.	Różnica
3-3	1.120.859	1.121.283	424	53.442	53.463	21
10-10	236.935	239.373	2.438	250.616	253.196	2.580

Tabela 3: Wyniki pomiarów dla 3 konsumentów i 3 producentów oraz 10 konsumentów i 10 producentów dla metody z czterema zmiennymi warunkowymi z flagą *fair* wykonane w czasie 60 sekund

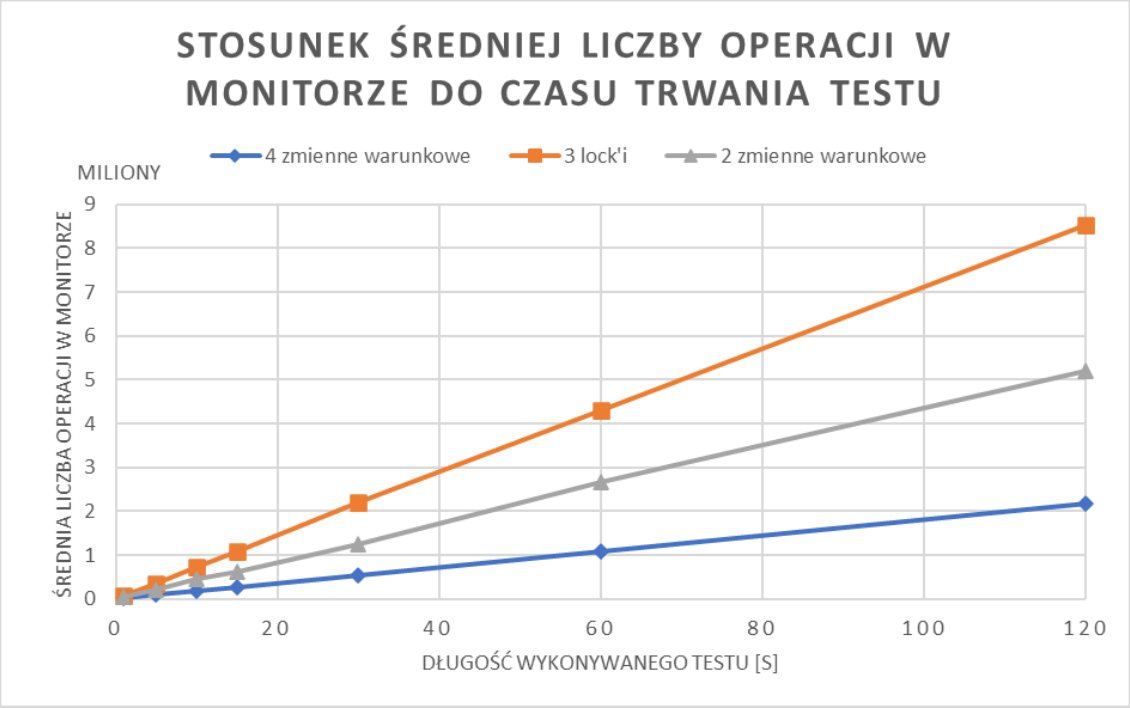
Patrząc na tabele 3 widzimy, że po zastosowaniu flagi *fair* otrzymane wyniki mają bardzo małą rozbieżność. Koszt jednak tej operacji to zauważalny wzrost czasu oczekiwania na wykonanie swojej operacji. Dlatego w dalszych porównaniach te ustawnienie będę stosować tylko dla rozwiązania na czterech zmiennych warunkowych.

5 Przeprowadzanie testów

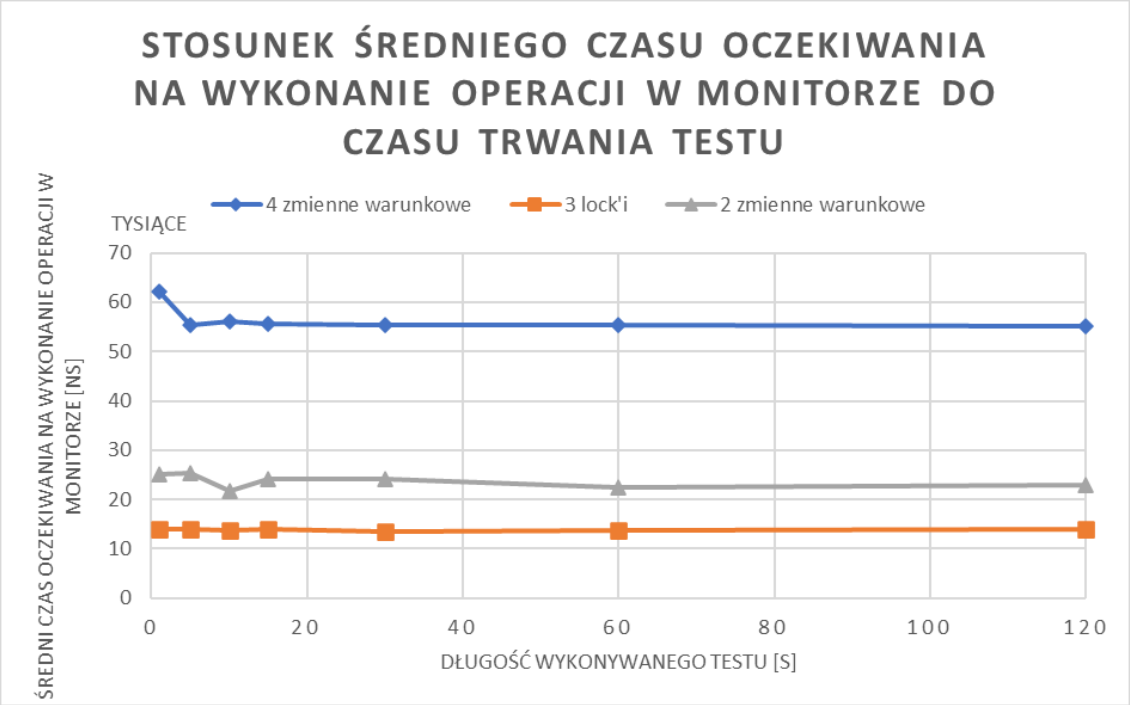
W celu przetestowania wymienionych już metod, bazowo stworzyłem po 3 wątki producentów i konsumentów, w związku z 6 rdzeniami fizycznymi procesora, na których były wykonywane obliczenia, z rozmiarem bufora 100 oraz czasem wykonywania testu 60 sekund. Przekazana im została ta sama instancja obiektu Random z ustawionym ziarnem "42". Następnie w zależności od danego testu modyfikowałem wartości poszczególnych zmiennych w celu sprawdzenia zmian na różne czynniki. Dodatkowo każdy pojedynczy test odbywał się poprzez osobne włączenie programu i zapisanie wyników testów do pliku xlsx, w celu uzyskania jak najbardziej rzetelnych i niezależnych od siebie rezultatów.

5.1 Czas wykonywania testu

Na samym początku sprwadziłem, czy długość wykonywania testów ma znaczenie. Patrząc teoretycznie na omawiane zagadnienie powinniśmy uzyskać analogiczne wyniki dla wszystkich jednostek pomiarowych. Celem tego pomiaru jest sprawdzenie, czy wszystkie implementacje działają poprawnie i nie występują żadne niepożądane anomalie.



Wykres 1: Stosunek średniej liczby operacji w monitorze do czasu trwania testu

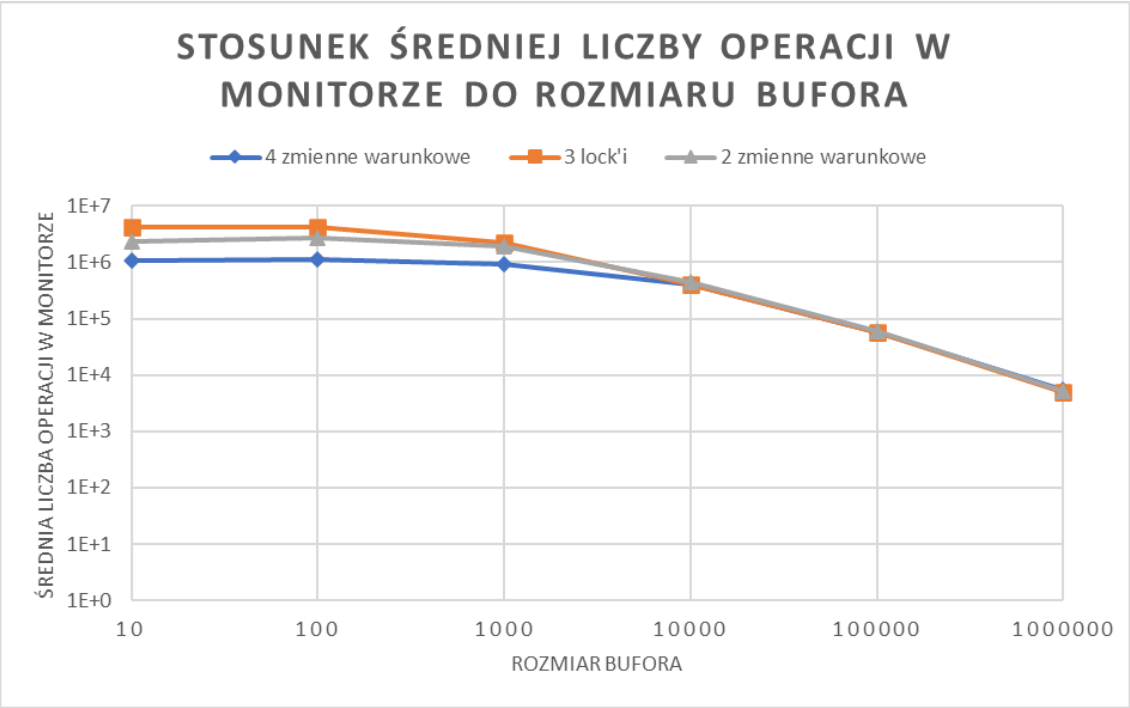


Wykres 2: Stosunek średniego czasu oczekiwania na wykonanie operacji w monitorze do czasu trwania testu

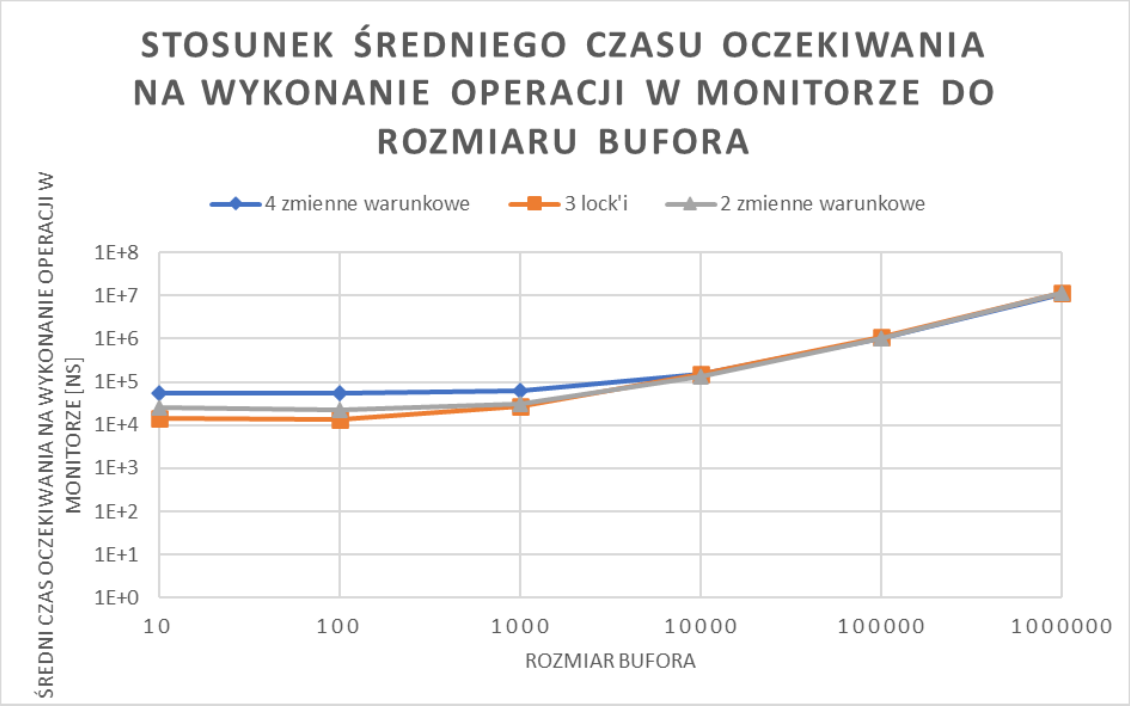
Odczytując rezultaty testów z wykresu 1, możemy śmiało stwierdzić, że utrzymaliśmy wyniki zgodne z początkową intuicją. Zebrane dane kształtują się liniowo, co ma sens, ponieważ im dłużej będziemy wykonywać test, tym więcej operacji zdołamy wykonać.
W przypadku wykresu 2 możemy dojść do podobnego wniosku. Odczytane dane pokazują nam, że długość wykonywanego testu nie wpływa na średni czas oczekiwania na wykonanie swojej operacji w monitorze. Jedyną większą anomalię możemy zobaczyć dla rozwizania na czterech zmiennych warunkowych. Widzimy, że średni czas oczekiwania jest trochę większy od pozostałych. Możliwe, że jest to wynik losowych opóźnień lub interferencji z innymi procesami działającymi w tle. Mimo wszystko wydaję mi się, że mieszczą się w granicy błędu pomiarów.
Dodatkowo z powyższych wykresów możemy zauważyć, że w stosunku do zgładzającego rozwiązania na dwóch zmiennych warunkowych, rozwiązanie wykorzystujące trzy lock'i sprawdza się lepiej. W takim samym czasie wątki wykonują więcej operacji, przy tym czekają mniej czasu na ich wykonanie. Gorzej za to wypada rozwiązanie wykorzystujące cztery zmienne warunkowe. Wpływ na to ma zastosowanie flagi *fair*, ale również aspekt bardziej skomplikowanego podejścia do problemu.
Z wyników tego testu może również stwierdzić, że wszystkie implementacje działają poprawnie.

5.2 Wielkość bufora

Kolejnym czynnikiem, który sprawdzę to wpływ rozmiaru bufora na średni czas oraz liczbę operacji. Z racji, że dane dla wątków są generowane z zakresu od 1 do połowy jego rozmiaru, może uda nam się zauważyć wpływ tego założenia na liczbę oraz czas wykonywania operacji.



Wykres 3: Stosunek średniej liczby operacji w monitorze do rozmiaru bufora przedstawiony w skali logarytmicznej



Wykres 4: Stosunek średniego czasu oczekiwania na wykonanie operacji w monitorze do rozmiaru bufora przedstawiony w skali logarytmicznej

Tak jak przypuszczałem, z wykresów 3 i 4 wynika, że wraz ze zwiększeniem rozmiaru bufora, liczba operacji malała odwrotnie proporcjonalnie do średniego czasu oczekiwania na wykonania operacji przez monitor. Co ciekawe, dla naszych trzech implementacji optymalne wyniki dostawaliśmy dla buforów z zakresu 10-1.000. Testy dla większych rozmiarów wykazywały już nieoptymalne, ale prawie identyczne wyniki dla wszystkich badanych sposobów. Dlatego w tym wypadku tylko dla rozmiarów mniejszych niż 1.000, rozwiązanie zagładzające wyszło gorzej od sposobu na 3 lock'ach, ale ponownie lepiej niż implementacja wykorzystująca 4 zmienne warunkowe. Dodatkowo, wraz ze zwiększaniem bufora, liczba operacji malała ze względu na szeroki zakres możliwych losowych wartości. W takim wypadku, można by założyć, że wątki producentów będą częściej wchodzić do monitora i generować dane dla konsumentów. Spróbuję zatem porównać minimalne i maksymalne liczby operacji wątków z podziałem na konsumentów i producentów przy rozmiarze bufora 1.000.000:

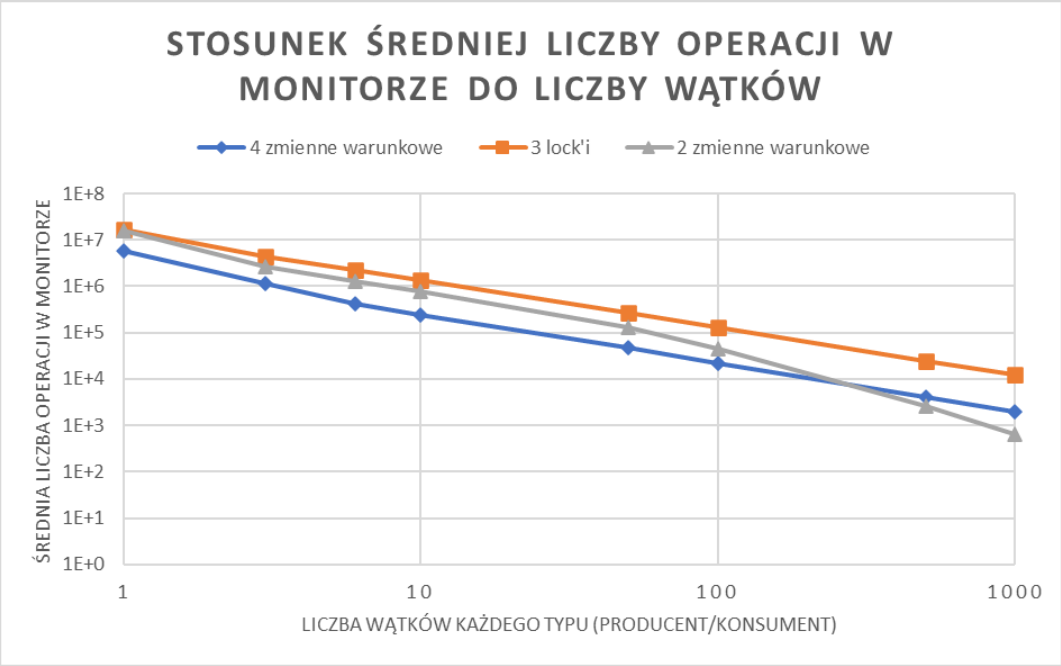
Sposób	Liczba operacji			
	Producenci		Konsumenci	
	Min.	Max.	Min.	Max.
4 zmienne warunkowe	5.369	5.420	5.412	5.444
3 lock'i	4.952	5.423	4.994	5.535
2 zmienne warunkowe	5.124	5.367	5.157	5.429

Tabela 4: Minimalne i maksymalne liczby operacji wykonywanych przez producentów i konsumentów przy różnych metodach synchronizacji dla rozmiaru bufora 1.000.000

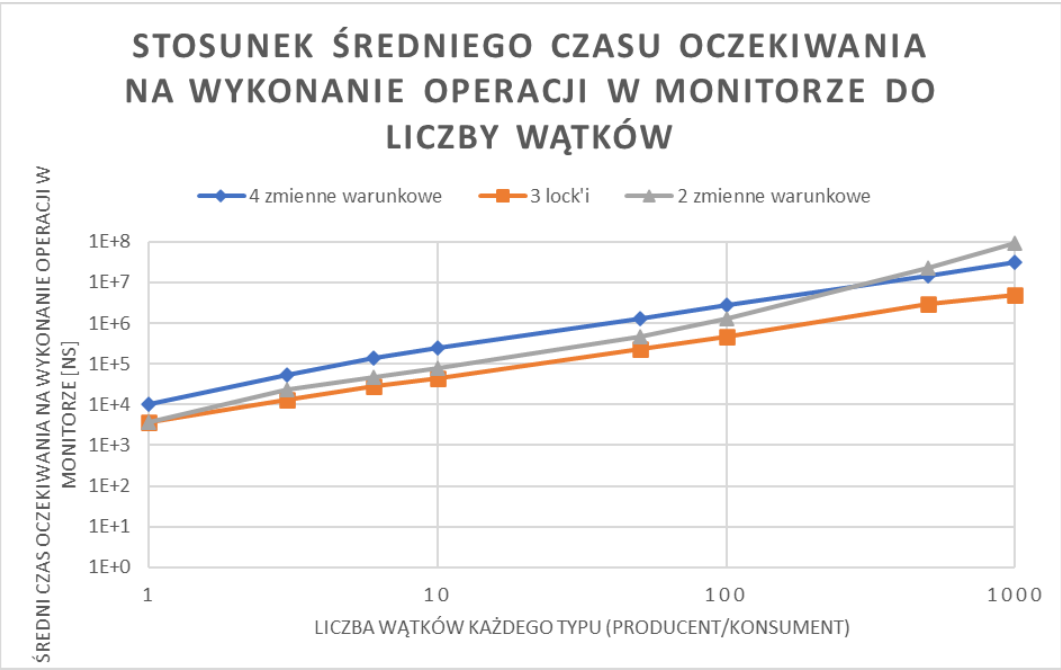
Pomimo wcześniejszej intuicji, z tabeli 4 wynika, że dla tak dużego rozmiaru bufora jak 1.000.000 minimalne i maksymalne liczby wykonanych operacji wątków producentów i konsumentów, dla wszystkich metod, są bardzo do siebie zbliżone. Możemy z tego wywnioskować, że mimo tak dużego bufora i jakby mogło się wydawać, większego zapotrzebowania na generowanie danych, wątki i tak starają się wchodzić i budzić w miarę równy sposób. Przez takie zachowanie uzyskujemy bardzo małą liczbe ogólnych operacji.

5.3 Liczba wątków

Od początku prawie wszystkie testy przeprowadzane były na łącznie 6 wątkach. Wynikało to z tego, że procesor, na którym były wykonywane testy, ma 6 wątków fizycznych. Takie ograniczenie umożliwiało maksymalne wykorzystanie zasobów sprzętowych bez narzutu wynikającego z przełączania kontekstua a testy były przeprowadzane z minimalnym opóźnieniem. Teraz sprawdzimy, czy takie założenie faktycznie miało sens, testując omawiane implementacje na innych liczbach wątków.



Wykres 5: Stosunek średniej liczby operacji w monitorze do liczby wątków przedstawiony w skali logarytmicznej



Wykres 6: Stosunek średniego czasu oczekiwania na wykonanie operacji w monitorze do liczby wątków przedstawiony w skali logarytmicznej

Po wykonanych testach, z wykresu 5 wynika, że wraz ze zwiększeniem liczby wątków w programie, wydajność spada. Okazuje się, że najwięcej operacji możemy wykonać mając po jednym konsumencie i producencie. Analogicznie z tabeli 6 widzimy, że wraz ze spadkiem liczby operacji wzrasta czas oczekiwania na wykonanie działania przez wątek. Co ciekawe, po raz pierwszy możemy zauważyć, że rozwiązanie zagładzające wychodzi najmniej optymalnie ze wszystkich. Zaczyna tak się dziać dopiero przy 1.000 wątkach łącznie. Dla mniejszej liczby w dalszym ciągu zachowana jest zależność: rozwizanie zagładzające działa wydajniej od tego na czterech zmiennych warunkowych, ale gorzej od zagnieżdżonych lock'ów.

6 Test zagładzania

W punkcie 5 sprawdzałem przygotowane implementacje używając wartości losowych z dozwolonego rozmiaru. Korzystając z takich danych rzadko można doprowadzić do wykazania jawnego przypadku zagłodzenia. Z tego też powodu możemy na szybko wywnioskować, że rozwiązanie w teorii zagładzające jest prostsze i zarazem wydajniejsze od tego z wykorzystaniem czterech zmiennych warunkowych. Ale co w wypadku, kiedy nasze dane są o określonej wielkości. Czy faktycznie zagładzanie, do którego może dojść, jest na tyle inwazyjne, że zasłania efektywność rozwiązania na dwóch zmiennych warunkowych? Do sprawdzenia tego ustawimy jednego producenta, który będzie generował dane rozmiaru 1, pięciu konsumentów, którzy będą pobierać dane takiego samego rozmiaru oraz jednego konsumenta, który będzie konsumował dane rozmiaru 10. Bufor zostanie ustawiony na rozmiar 100, a testy będą przeprowadzane w czasie 60 sekund.

	Liczba operacji			Średni czas oczekiwania [ns]		
	2 zm. warunkowe	4 zm. warunkowe	3 lock'i	2 zm. warunkowe	4 zm. warunkowe	3 lock'i
Konsument 1	7.525.875	93.129	20.659.492	7.942	644.364	2.872
Konsument 2	7.319.811	117.055	20.565.155	8.167	512.644	2.885
Konsument 3	7.440.122	106.979	21.134.809	8.034	560.935	2.807
Konsument 4	7.746.617	106.343	20.655.158	7.715	564.279	2.872
Konsument 5	7.509.432	112.699	21.318.719	7.960	532.455	2.782
Konsument 6	4.692.565	521.355	16.613.444	12.756	115.065	3.579
Różnica	3.054.052	428.226	4.705.275	5.041	529.299	797

Tabela 5: Porównanie liczby operacji i średniego czasu oczekiwania wątków dla wszystkich badanych metod przy sztucznym ustawieniu wielkości konsumowanych danych

Z tabeli 5 łatwo możemy zauważyć, że wątkiem, który miał inne ustawienia rozmiaru pobieranych danych, był konsument 6. Metodą, która zachowała się w najbardziej przewidywalny sposób, są 3 lock'i. Z uzyskanych danych widzimy, że liczba wykonanych operacji szóstego konsumenta bardzo nie odstaje od pozostałych, tym bardziej jak pod uwage weźmiemy, że on i tak potrzebował dziesięc razy więcej danych niż pozostali. Tak samo czas oczekiwania nieznacząco się różni.
Patrząc na implementacje w teorii zagładzającą, widzimy, że nie uzyskuje ona aż tak złych wyników. Szósty konsument wykonywał swoje operacje ok. 2 razy mniej razy od pozostałych. Wydaję mi się, że nie jest to tak ogromna różnica, biorąc pod uwagę, że w tym czasie, tak jak poprzednia metoda, pobiera więcej zasobów niż pozostali. Wypada mimo wszystko gorzej.
Największe zaskoczenie wykazuje metoda korzystająca z 4 zmiennych warunkowych. U niej możemy zaobserwować, że ostatni konsument zagładza pozostałych. Po analizie nie potrafiłem dojść do wniosku dlaczego tak się dzieje. Jedyną możliwością jest omawiana na początku flaga *fair*. Po otrzymanym rezultacie testu sprawdziłem jeszcze wyniki bez niej, ale okazało się że w takim przypadku szósty konsument zagładzał pozostałych jeszcze bardziej. Wykonywał operacje ok. 200 razy częściej. Tak jak w poprzednich testach uważam, że sposób implementacji tego rozwiązania w Javie, może być przez nią w niepożądany interpretowany i dlatego dostajemy takie nieoptymalne wyniki.

7 Wnioski

Problem producenta-konsumenta ma pewną pulę rozwiązań, z której możemy czerpać w zależności od naszych potrzeb. Omówione w tym sprawozdaniu metody z wykorzystaniem różnej liczby zmiennych warunkowych lub nawet większej liczby lock'ów mają swoje zalety i wady, które postarałem się wykazać.
Na podstawie testów z czasem wykonywania widzimy, że niezależnie od użytej metody dostawaliśmy w obrębie każdej bardzo podobne wyniki. Daje nam to do zrozumienia, że w długofalowym działaniu programu wielowątkowego, każde rozwiązanie powinno sobie poradzić, bez wprowadzania większych nieprawidłowości w naszym programie.
Podczas testowania różnej wielkości bufora mogliśmy zauważyć, że przynajmniej w przypadku danych losowych, nie warto z jego rozmiarem przesadzać. Według pomiarów najlepiej trzymać sie stosunkowo małego zakresu. Wtedy też wątki działają najbardziej optymalnie i nie blokują się w oczekiwaniu na inne.
Badanie liczby wątków, które optymalnie ze sobą działały wykazało, że najlepiej jest stosować liczbę niewiększą niż liczba wątków fizycznych procesora, na którym program wielowątkowy jest wykonywany. Z reguły jeżeli chcemy nasze rozwiązanie udostępnić na większą liczbę maszyn, nie będziemy wiedzieli jaką mocą dysponują. Dlatego ważne jest, aby po prostu starać się rozsądnie dysponować liczbą wątków.
Po wszystkich tych testach możemy zauważyć, że najbardziej optymalną metodą we wszystkich wykonanych testach była ta wykorzystująca 3 lock'i. Wykazywała się największą liczbą wykonanych operacji przez wątki oraz najmniejszym czasem oczekiwania. Metoda zagładzająca wyszła również w bardzo przyzwoity sposób. Nie odstawała tak bardzo od zagnieżdzonych lock'ów i była zarazem lepsza od czterech zmiennych warunkowych. Te zaś okazały się najgorszym rozwiązaniem. Jedynym większym odstępstwem od tych rezultatów jest bardzo duży bufor. W takim wypadku nie ma większej różnicy jaką metodę wybierzemy, bo każda będzię działać nieoptymalnie.
Powyższe wnioski potwierdził test zagładzania, z którego uzyskaliśmy najdokładniejsze wyniki dla zagnieżdżonych lock'ów. Dwie pozostałe metody wykazywały zagłodzenie, jednakże w dwie inne strony.