AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



Projekt zaliczeniowy

pt.

„Wykrywanie zakleszczeń (deadlock) w systemach rozproszonych”

Zespół w składzie: Łukasz Basiura

Michał Czarnik

Przemysław Duda

Adrian Sołtys

Kierunek studiów: Informatyka Stosowana

Spis treści

[1. Analiza zagadnienia: 3](#_Toc453918866)

[2. **Algorytmy rozwiązujące problem zakleszczenia:** 3](#_Toc453918867)

[2.1 Algorytm bankiera: 4](#_Toc453918898)

[2.2 Algorytm grafu przydziału zasobów: 4](#_Toc453918900)

[2.3 Algorytmy oparte o metodę One-Shot 5](#_Toc453918902)

[2.4 Algorytmy oparte o metodę Hierarchiczną 7](#_Toc453918903)

[2.5 Wait-Die algorithm 8](#_Toc453918904)

[2.6 Wound-Wait algorithm 8](#_Toc453918908)

[2.7 Chandy-Misra-Haas Algorithm 9](#_Toc453918909)

[2.8 Algorytm Ho-Ramamoorthy 11](#_Toc453918921)

[**3.** **Analiza algorytmów:** 11](#_Toc453918924)

[3.1.Rodzaje algorytmów do zakleszczeń: 11](#_Toc453918925)

[3.2 Złożoność obliczeniowa: 12](#_Toc453918926)

[3.3 Pseudokody algorytmów: 13](#_Toc453918958)

[3.4 Zakres Stosowalności: 16](#_Toc453918976)

[3.5 Porównanie algorytmów wraz z ze znaną realizacją wdrożeń : 18](#_Toc453918981)

[3.6 Trudności implementacyjne: 18](#_Toc453918989)

[**4.** **Użyte technologie rozproszone:** 21](#_Toc453918996)

**5. Analiza porównawcza implementacji:** ………………………………………………..21

[Bibliografia: 25](#_Toc453918997)

1. Analiza zagadnienia:

Jeżeli grupa procesów oczekuje na zdarzenie, które może być spowodowane jedynie przez któryś z tych procesów, to taka sytuacje nazywamy *zakleszczeniem* lub *impasem (ang. deadlock)*. Przez grupę rozumiemy dwa lub większa liczbę procesów. Do zjawiska zakleszczenia może zazwyczaj dojść przy próbie dostępu do zasobów współdzielonych. Istnieją również inne zjawiska, które maja skutki podobne do zakleszczenia, ale nie spełniają definicji zakleszczeń. Przykładem może być *autozakleszczenie*, do którego może dojść, jeśli proces na skutek błędu logicznego będzie próbował zając zasób, który wcześniej już zajął.

Rozważmy system, w którym są procesor i drukarka. Żeby coś wydrukować, proces musi otrzymać oba te zasoby. Przypuśćmy, że dwa procesy chcą drukować. Jednemu z nich system przydzielił drukarkę a drugiemu procesor. Pierwszy trzyma więc procesor i oczekuje na przydział drukarki, który nigdy nie nastąpi, ponieważ tę trzyma drugi proces oczekujący na przydział procesora.[1]

Zasadniczo można wyróżnić dwa rodzaje podejść do rozwiązania problemu zakleszczenia. Jedno polega na niedopuszczeniu do powstania zakleszczenia, drugie dopuszcza zakleszczenie, ale jego istotą jest wykrywanie (detekcja, ang. deadlock detection) i usuwanie tego stanu. Niedopuszczenie do zakleszczenia sprowadza się do zapobiegania (ang. deadlock prevention) lub unikania (ang.deadlock avoidance).

Zapobieganie jest metodą dość zachowawczą, polegającą na przeciwdziałaniu zajściu jednego z warunków koniecznych wystąpienia zakleszczenia. Unikanie jest z kolei metodą pośrednią pomiędzy zapobieganiem a detekcją. Jej istotą jest przewidywanie przyszłych zdarzeń w systemie i sprawdzanie, czy w osiągalnych stanach występuje zakleszczenie. Stosuje się przy tym takie same metody, jak w przypadku wykrywania. Jeśli stan zakleszczenia jest osiągalny, to mamy do czynienie ze stanem niebezpiecznym (stanem zagrożenia), którego należy unikać, realizując żądania procesów.

1. Algorytmy rozwiązujące problem zakleszczenia:

|  |  |  |
| --- | --- | --- |
| Nazwa algorytmu | Autor | Rok powstania |
| Algorytm bankiera | [Edsger Dijkstra](https://pl.wikipedia.org/wiki/Edsger_Dijkstra) | 1965 |
| Algorytm przydziału zasobów | Haberman | 1953 |
| One-shot algorithm | Howard J. Hamilton | - |
| Hierarchical Algorithm | Howard J. Hamilton | - |
| Wait-Die algorithm | Rosenkrantz | 1978 |
| Wound-Wait algorithm | Rosenkrantz | 1978 |
| Chandy-Misra-Haas Algorithm | K. Mani Chandy,  Jayadev Misra,  Laura M Haas | 1983 |
| Algorytm  Ho-Ramamoorthy | S. F. Ho  C. V Ramamoorthy | 1985 |

* 1. Algorytm bankiera:

Algorytm bankiera jest analogią do banku, który nie zainwestuje gotówki tak, aby uniemożliwić zaspokojenie wymagań wszystkich klientów. Stosowany jest przez system operacyjny kiedy jakikolwiek proces zażąda zasobu. Zapobiega on wystąpieniu zakleszczeń przez odmówienie lub zawieszenie dostępu do zasobu w przypadku, gdyby dostęp do tego zasobu mógł spowodować wejście

systemu w stan niebezpieczny. Każdy typ zasobu ma wiele egzemplarzy. Ponadto proces musi zadeklarować maksymalną liczbę egzemplarzy zasobów, które będą mu potrzebne. Liczba ta nie może przekraczać ogólnej liczby w systemie. W sytuacji gdy proces żąda zasobu, którego przydział spowodowałby wejście systemu w stan zagrożenia, proces musi poczekać.

Aby algorytm bankiera działał potrzebuje wiedzieć trzy rzeczy:

1. Jaką część każdego zasobu każdy proces może zażądać

2. Jaką część każdego zasobu każdy proces aktualnie używa

3. Jaką część każdego zasobu system ma aktualnie dostępną.

**Struktury danych algorytmu bankiera**:

* Dostępne: Wektor o długości m, określający liczbę dostępnych zasobów każdego typu.

Dostępne[j] = k - oznacza, że jest dostępnych k egzemplarzy zasobu typu Zj.

* Maksymalne: Macierz o wymiarach n x m, definiująca maksymalne żądania każdego

procesu. Jeśli Maksymalne[i,j]=k to proces Pi może zamówić co najwyżej k egzemplarzy zasobu typu Zj.

* Przydzielone: Macierz o wymiarach n x m, definiująca liczbę zasobów poszczególnych

typów, przydzielonych do każdego z procesów. Gdy Przydzielone[i, j] = k, wówczas

proces Pi ma przydzielonych k egzemplarzy zasobu typu Zj.

* Potrzebne: Macierz o wymiarach n x m, przechowująca pozostałe do spełnienia

zamówienia każdego z procesów. Element Potrzebne[i,j]=k oznacza, że do

zakończenia swojej pracy proces Pi może jeszcze potrzebować k dodatkowych

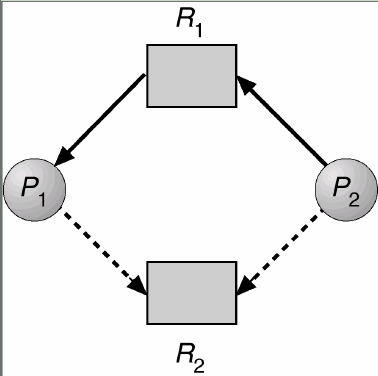
egzemplarzy zasobu typu Zj. Zauważmy, że

Potrzebne[i,j] = Maksymalne[i,j] - Przydzielone[i,j]

* 1. Algorytm grafu przydziału zasobów:

Graf przydziału zasobów jest wygodną formą graficzną reprezentacji stanu sytemu na potrzeby analizy zjawisk związanych z zakleszczeniem. Poza tym można wykazać na podstawie pewnych własności tego grafu, że wystąpił pewien szczególny stan, np. właśnie stan zakleszczenia lub zagrożenia, co w połączeniu z powszechnie znanymi algorytmami analizy grafów ułatwia wykrywanie takich stanów.

Graf przydziału zasobów jest to skierowany graf dwudzielny, w którym jedną grupę wierzchołków tworzą procesy, a drugą zasoby. Krawędzie skierowane od procesów do zasobów reprezentują zamówienia, a krawędzie skierowane od zasobów (ich jednostek) do procesów reprezentują przydział.



Rysunek 1 Przykład grafu przydziału zasobów.

W grafie tym występują dwa rodzaje wierzchołków:

1. Pierwszy rodzaj odpowiada procesom P.

2. Drugi rodzaj wierzchołków odpowiada zasobom R.

Krawędź skierowana od procesu *Pi* do zasobu *Rj* (*Pi -> Rj* ) nazywamy *krawędzią zamówienia*. Krawędź skierowana odwrotnie (*Pi<- Rj* ) nazywamy *krawędzią przydziału*. *Warunkiem koniecznym* do wystąpienia zakleszczenia w systemie jest to, aby w grafie przydziału zasobów powstał cykl. Nie jest to jednak *warunek wystarczający*.

* 1. Algorytmy oparte o metodę One-Shot

W celu uniknięcia zakleszczenia trzeba się upewnić żeby kolejkowanie się nigdy nadmiernie nie zwiększało. Jedną z możliwości, aby temu zapobiec jest podział wszystkich procesów na dwie grupy:

* Pierwsza grupa: Procesy posiadające wszystkie zasoby, których kiedykolwiek będą potrzebować.
* Druga grupa: Procesy nieposiadające zasobów.

Procesy należące do pierwszej grupy nigdy nie będą blokowane, natomiast procesy należące do drugiej grupy będą blokowane w trakcie oczekiwania na zwolnienie zasobów przez grupę pierwszą. Nazywa się to alokacją *one-shot*, i implikuje to, że każdy proces w momencie dostępu do zasobów pobiera wszystkie zasoby mu potrzebne. Procesy próbujące dostać zasoby aktualnie używane przez inny proces są blokowane aż do momentu ich zwolnienia przez używający ich aktualnie proces. Każda następna próba dostępu do zasobów po ich poprzednim otrzymaniu skutkuje błędem.

W swojej czystej formie, metoda *one-shot* wymaga, aby każdy proces pobrał wymagane przez siebie zasoby tylko raz, i nigdy więcej nie próbował uzyskać dostępu do nich ponownie.

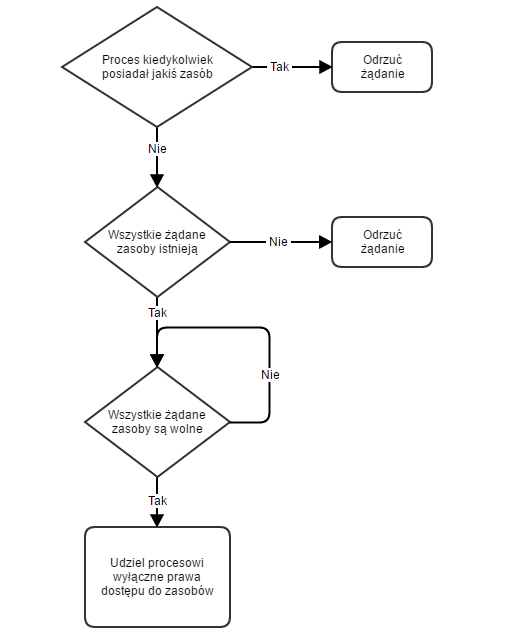
W skrócie metoda *one-shot* podlega jednej bardzo ważnej zasadzie:

***Proces może żądać zasoby tylko, jeżeli nie posiada żadnych.***

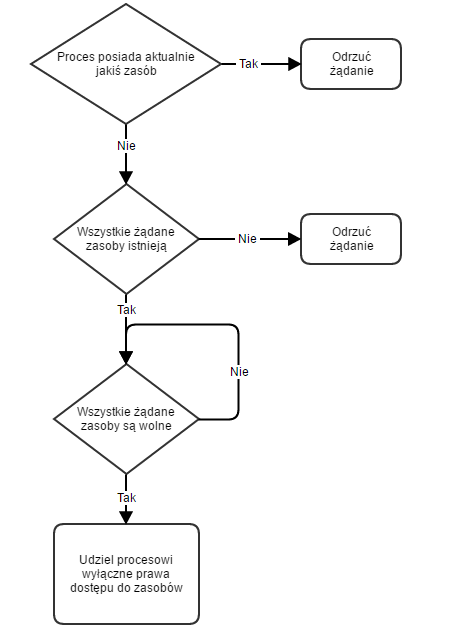
Metoda ta jest bardzo konserwatywna, przez co mało praktyczna. Dla przykładu, jeżeli jakiś proces będzie potrzebował bardzo dużej ilości zasobów, będzie musiał pobrać je wszystkie na raz, tym samym blokując dostęp do nich dla innych procesów, które w międzyczasie mogłyby go używać. Ponadto proces może nie wiedzieć, jakich zasobów może w przyszłości potrzebować.

Na postawie tej metody powstały dwa algorytmy:

* Algorytm One-Shot – algorytm ściśle przestrzegający metody *one-shot*. Posiada złożoność obliczeniową *0(****nt****).* Jedną ze znanych implementacji tego algorytmu jest funkcja WaitForMultipleObjects dostępna od Windowsa 2000. *Jego* działanie jest zaprezentowane na Schemat Blokowy 1.
* Algorytm Multishot – algorytm działający wedle metody *one-shot* z jedną małą różnicą. Pozwala on na wielokrotne wysyłanie żądań przez ten sam wątek. Również posiada złożoność obliczeniową 0(**nt**). Jego działanie jest zaprezentowane na Schemat Blokowy 2.



Schemat Blokowy 1 Algorytm One-Shot



Schemat Blokowy 2 Algorytm Multishot

* 1. Algorytmy oparte o metodę Hierarchiczną

Aby uniknąć zakleszczenia i nadmiernego wzrostu kolejkowania procesów w celu dostępu do danych używa się metody Hierarchicznej. Polega ona na przydzielaniu procesom numerów rosnąco, oraz asocjacji procesów z najwyższym priorytetem posiadanych zasobów. Metoda hierarchiczna podlega następującej zasadzie:

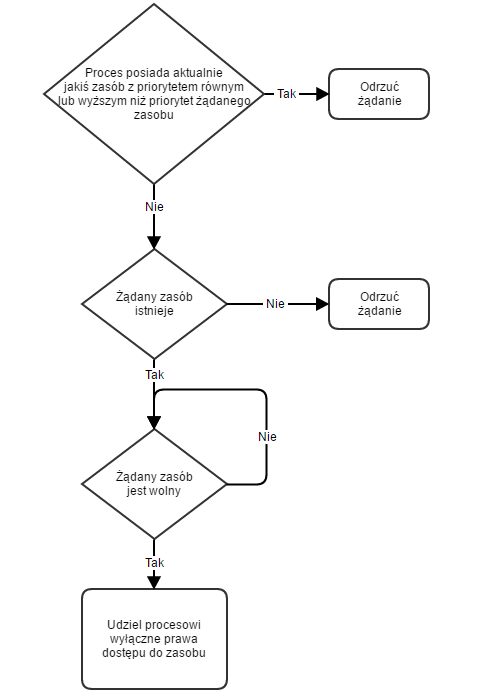
***Proces może żądać zasobów tylko posiadających wyższe priorytety niż***

***aktualnie posiadane zasoby***

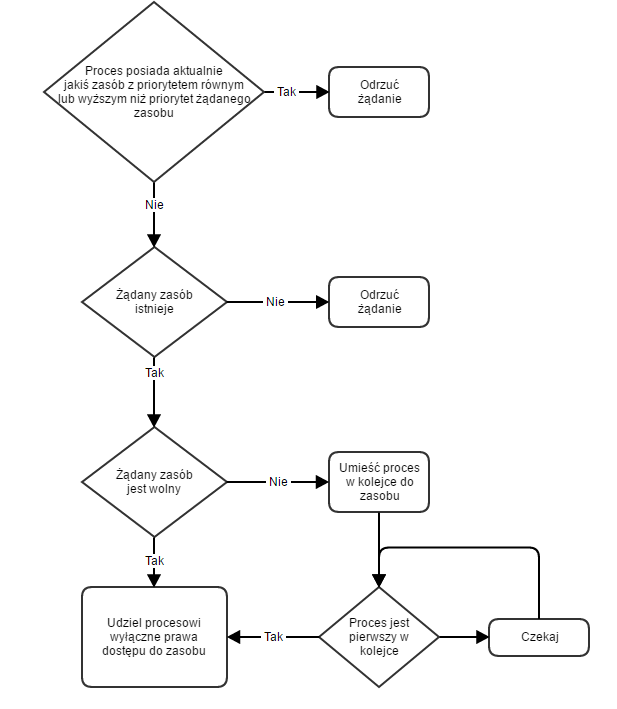
Metoda hierarchicznej alokacji zasobów rozwiązuje problem zakleszczenia. Niestety jest to algorytm dalej konserwatywny, ponieważ proces musi żądać zasobów z niskimi priorytetami dużo wcześniej niż będzie wysyłał żądania o zasoby wysoko priorytetowe. Po dostaniu zasobów wysokopriorytetowych algorytm nie będzie miał możliwości pozyskania zasobów z niskimi priorytetami do momentu aż nie odda wszystkich zasobów posiadających wyższe priorytety. Implikuje to powstanie sztucznego cyklu żądania zasobów z niższymi priorytetami, aby proces nie musiał oddawać zasobów wysokopriorytetowych w celu ich pozyskania. Procesy pozyskujące w ten sposób blokują zasoby nisko priorytetowe, pomimo że ich jeszcze nie potrzebują, przez co zabierają dostęp do nich innym procesom, które mogą ich potrzebować wcześniej. Przy używaniu tej metody rzadkie zasoby powinny dostawać wysokie priorytety, aby żądania one były wykonywane tylko wtedy, gdy są potrzebne.

Na podstawie tej metody powstały dwa algorytmy:

* Algorytm Hierarchiczny – Algorytm ten przestrzega metody Hierarchicznej. Posiada on złożoność obliczeniową 0(**nt**). Jego działanie jest zaprezentowane na Schemat Blokowy 3
* Algorytm Hierarchiczny z kolejką - Algorytm ten przestrzega metody Hierarchicznej, z taką różnicą, że procesy próbujące uzyskać dostęp do danego zasobu zostają zakolejkowane. Posiada on złożoność obliczeniową 0(**nt**). Jego działanie jest zaprezentowane na Schemat Blokowy 4



Schemat Blokowy 3 Algorytm Hierarchiczny



Schemat Blokowy 4 Algorytm Hierarchiczny z kolejką

* 1. Wait-Die algorithm

**Algorytm “czekaj albo giń”** (angielskie *wait-die algorithm*), algorytm zapobiegania blokadzie, w którym proces z dłuższą historią oczekuje na zasób przetrzymywany przez nowy proces, natomiast nowy proces zostaje zaniechany tylekroć, ilekroć spróbuje pozyskać zasób użytkowany przez proces stary. Algorytm powstał przez Rosenkrantz’a w 1978. Możemy zapobiegać zakleszczeniom poprzez dawanie transakcjom priorytetów, takim priorytetem może być czas stworzenia takiego procesu zaraz przy jego tworzeniu i na podstawie tego czasu potem decydować czy starszy czy młodszy powinien czekać w zależności, który algorytm, w wypadku Wait and die algorytm działa następująco.

**Algorytm polega na tym, że gdy transakcja Ti prosi o dane z Tj, Ti może czekać na te dane tylko wtedy gdy czas stworzenia jest mniejszy niż Tj, czyli Ti jest starszy niż Tj, w innym przypadku Ti umiera, jest później restartowany ale z randomowym opóźnieniem i tym samym czasem stworzenia. Ten algorytm jest mniej wydajny niż Wound-Wait algorytm.**

**Algorytm jest często używany w bazach danych. Tak jak wound- wait.**

* 1. Wound-Wait algorithm

Algorytm stworzony przez tego samego człowieka co stworzył Wait-Die algorytm. Z tą różnicą, że ten algorytm działa na odwrót do tamtego. Jeżeli Ti ma wyższy priorytet niż Tj, anuluj Tj w innym wypadku Ti czeka. To znaczy, że gdy Ti jest młodszy niż Tj to może czekać na zwolnienie zasobów. Czyli oznacza to, że ten algorytm daje dostęp do zasobów najpierw starszy proces, jeżeli proces jest młodszy to czeka w kolejce. W poprzednim algorytmie było na odwrót, proces starszy czekał na dane. Mądrzejszym wyjściem jest jednak właśnie starszym procesom dać pierwszeństwo.

W zakleszczeniu pojawia się też problem zagłodzenia procesów, oba te algorytmy unikają tego zjawiska poprzez ciągłe zwiększanie priorytetów powoduje, że sytuacja nie powtarza się , tzn. te same procesy nie są dwa razy z rzędu odwoływane co nie powoduje ich zagłodzenia. Pytanie , kiedy powinniśmy używać wykrywania zakleszczenia a kiedy algorytmu do jego unikania? Odpowiedź jest prosta, kiedy transakcja jest krótka, posiada mało elementów do wymiany bądź po prostu jest bardzo szybka możemy użyć wykrywania zakleszczenia, natomiast jeżeli transakcja jest duża, zachodzi w niej wymiana dużej ilości danych najlepiej zastosować zapobieganie zakleszczeniu.

* 1. Chandy-Misra-Haas Algorithm

Algorytm ten został opublikowany w 1983 przez jego autorów - K. ManChandy, Jayadev Misra oraz Laurę M Haas. Pozwala on wykrywać zakleszczenia w systemach rozproszonych.

Zdefiniujmy najpierw zależności lokalne. Rozważmy n procesów P1, P2 … Pn które wykonywane są w jednym systemie. P1 jest lokalnie zależne od Pn, jeśli P1 zależy od P2, P2 od P3 itd. Zatem P1 jest lokalnie zależne od siebie jeśli jest lokalnie zależne od Pn,, a Pn jest zależne od P1.

Algorytm używa wiadomości nazywanych próbkami(i,j,k) aby przesyłać wiadomości z kontrolera procesu Pj do kontrolera procesu Pk. Wiadomość jest wysyłana z procesu Pi w celu sprawdzenia czy zakleszczenie wystąpi. Każdy proces Pj przetwarza tablicę binarną zawierającą zależności od których jest on zależny. Na samym początku każda wartość tej tablicy ustawiona jest na "fałsz".

Przed wysyłaniem, proces sprawdza czy Pj jest lokalnie zależne od niego samego. Jeśli tak to występuje zakleszczenie. W przeciwnym wypadku sprawdza czy Pj i Pk są w różnych kontrolerach, czy są lokalnie zależne i czy Pj czeka na zasoby zablokowane przez Pk. Jeśli wszystkie te warunki są spełnione wysyła zapytanie.

Po stronie odbierającej, kontroler sprawdza czy Pk wykonuje zadanie. Jeśli tak to porzuca sprawdzanie. Jeśli nie, sprawdza odpowiedzi wysłane przez Pk do Pj i dependentk(i) jest nie spełniony. Po weryfikacji ustawia wartość pozytywną w tablicy dependentk(i). Jeśli w tym momencie k jest równe i to zakleszczenie występuje, jeśli nie zapytanie jest wysyłane do następnego zależnego procesu.

Pseudokod kontrolera wysyłającego zapytanie:

**jeśli** *Pj* jest lokalnie zależne od siebie

**to zadeklaruj zakleszczenie**

**inaczej** dla każdego *Pj,Pk* takiego, że:

(i) *Pi* jest lokalnie zależne od *Pj*,

(ii) *Pj* oczekuje na *'Pk*,

(iii) *Pj*, *Pk* są na różnych kontrolerach

**wyślij zapytanie(i, j, k)**

Pseudokod kontrolera obsługującego zapytanie:

**jeśli**

(i)*Pk* nie jest zajęty,

(ii) *dependentk*(i) = fałsz,

(iii)odpowiedź na zapytanie z *Pk* do *Pj*

**wykonaj**

dependentsk(i) = prawda;

**jeśli** k == i

**zadeklaruj** *Pi* jako zakleszczone

**inaczej** dla wszystkich *Pa*,*Pb* takich, że:

(i) *Pk* jest lokalnie zależne od *Pa*,

(ii) *Pa* oczekuje na '*Pb*,

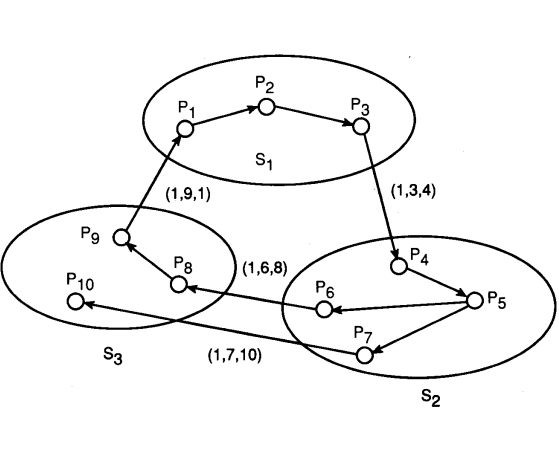
(iii) *Pa*, *Pb* są na różnych kontrolerach

**wyślij zapytanie(i, j, k)**

Przykładowo P1 inicjuje detekcję zakleszczenia. C1 wysyła zapytanie mówiące, że P2 zależy od P3. Kiedy zapytanie zostanie odebrane przez C2, C2 sprawdzi czy P3

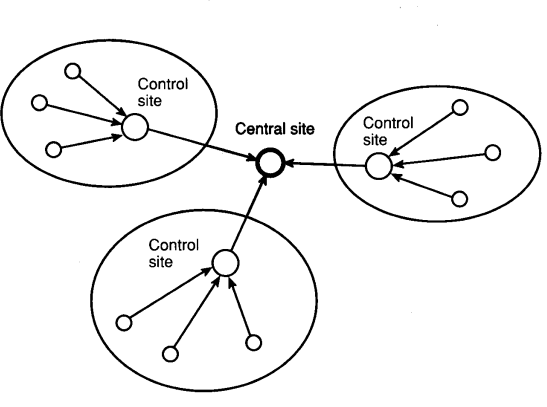
jest w stanie spoczynku. P3 jest w stanie spoczynku ponieważ jest lokalnie zależny od P4 i zaktualizuje tablicę zależności D3(2) na wartość 1. Tak jak wyżej C2 wysyła próbkę do C3 i C3 wysyła do C1. W C1 proces P1 jest w stanie spoczynku więc zaktualizuje tablicę D1(1) na wartość 1. W tym momencie zostaje zadeklarowane zakleszczenie.

Jeśli chodzi o złożoność, załóżmy że jest m kontrolerów oraz p procesów do przetworzenia. Aby zadeklarować czy zakleszczenie wystąpi musimy "odwiedzić" wszystkie kontrolery i procesy. Dlatego rozwiązaniem będzie O(m+p), a złożoność czasowa obliczeń to O(n).



* 1. Algorytm Ho-Ramamoorthy

Algorytm Ho-Ramamoorthy jest algorytmem hierarchicznym. Klienci są zgrupowani w wiele osobnych klastrów. Cyklicznie jeden z klientów jest wybierany jak główny klient zarządzający danym klastrem. Centralny system żąda od każdego z klientów zarządzających w danym momencie statusu transakcji wewnątrz klastra oraz zależności oczekiwania na. W ten sposób wybrany klient zarządzający tworzy tablicę z danych o klientach wewnątrz klastra i uruchamia na niej jedno fazowy algorytm detekcji zakleszczeń ograniczony tylko do transakcji wewnątrz klastra. Po zakończeniu wysyła informacje o statusach transakcji i relacjach oczekiwania. Centralny system na podstawie otrzymanych danych tworzy graf stanów w którym wyszukuje cykle. W ten sposób lokalnie wybrany klient zarządzający wykrywa wszystkie zakleszczenia wewnątrz klastra, a centralny system wykrywa zakleszczenia w żądaniach pomiędzy klastrami.



1. **Analiza algorytmów:**

Niniejszy rozdział prezentuje szczegółową analizę porównawczą wybranych algorytmów rozwiązujących problem uzgadniania.

## 3.1.Rodzaje algorytmów do zakleszczeń:

|  |  |  |
| --- | --- | --- |
|  | | |
| Deadlock prevencion | **Deadlock detection** | **Deadlock avoidance** |
| Algorytmy z tej dziedziny skupiają się na tym aby podczas wymiany danych, podczas działania programu nie doszło do zakleszczenia. Jest to o tyle dobra technika, że system nie traci na wydajności poprzez opóźnianie wykonania zadania przez brak wymiany danych pomiędzy procesami. Nie trzeba wykrywać zakleszczeń ani próbować ich unikać, jest to najlepsza metoda. Wadą jest to, że algorytmy te muszą posiadać prealokacje zasobów co redukuje potencjalną współbieżność.  Znane metody : Circular Wait, hold and wait. | Algorytmy z tej dziedziny skupiają się na tym, że szukają w programie czy wystąpiło zakleszczenie , jeśli wystąpiło algorytm natychmiast próbuje przywrócić wszystko na właściwe tory. Wadą tej taktyki jest to, że zamiast unikać zakleszczeń od razu zakładamy że wystąpią i skupiamy się na powstrzymywaniu skutków a nie reperowaniu problemu u źródła.  Znane algorytmy to : hierarchiczne, Chandy-Misra-Haas, Ho-Ramamoorthy. | Algorytmy z tej dziedziny skupiają się na tym, że sprawdzają czy w programie może dojść do zakleszczenia, jeśli tak to starają się uprzedzić program i tak zarządzać danymi by uniknąć tego zakleszczenia które ma nastąpić(rozpoznaje czy transakcja jest bezpieczna). Najczęściej stosowane algorytmy w bazodanowych systemach, są w tej dziedzinie najbardziej popularne gdyż takie systemy już mają umiejętność anulowania transakcji.  Znane algorytmy to : Wound-Wait, Wait-Die,bankiera. |

## 3.2 Złożoność obliczeniowa:

|  |  |
| --- | --- |
| Nazwa algorytmu | Złożoność obliczeniowa |
| Algorytm Bankiera | Gdzie n liczba procesów, d – liczba zasobów |
| Algorytm grafu przydziału zasobów |  |
| Algorytm One-Shot | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Algorytm Multishot | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Algorytm Hierarchiczny | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Algorytm Hierarchiczny z kolejką | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Wait-Die | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Wound-Wait | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Chandy-Misra-Haas Algorithm | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |
| Algorytm Ho-Ramamoorthy | Gdzie n liczba procesów, d – czas dostępu do zasobu przez proces |

## 3.3 Pseudokody algorytmów:

|  |  |  |  |
| --- | --- | --- | --- |
| Nazwa algorytmu | Pseudokod |  |  |
| Algorytm Bankiera | work[1..m] = available /\* how many resources are available \*/  finish[1..n] = false (for all i) /\* none finished yet \*/  **Step 1:**  Find an i such that finish[i]=false and need[i] <= work /\* find a proc that can complete its request  now \*/  If no such i exists, go to step 3 /\* we’re done \*/  **Step 2:** Found an i:  finish [i] = true /\* done with this process \*/  work = work + allocation [i]  /\* assume this process were to finish, and its allocation back to the  available list \*/  go to step 1  **Step 3:** If finish[i] = true for all i, the system is safe. Else Not |  |  |
| Algorytm grafu przydziału zasobów | Request = wektor zamówień dla procesu Pi.  Requesti [j] = k - proces Pi potrzebuje k egzemplarzy zasobu typu Rj.  1. If Requesti £ Needi then go to step 2  else error /proces przekroczył deklarowane max/  2. If Requesti £ Available then go to step 3  elsePi musi czekać /zasoby nie są dostepne/  3. Zmiany stanu:  Available = Available - Requesti;  Allocationi = Allocationi + Requesti;  Needi = Needi – Requesti;  • If stan bezpieczny ⇒ zasoby przydzielone procesowi Pi.  • If stan zagrożenia ⇒ Pi musi czekać; przywrócenie  poprzedniego stanu przydziału zasobów |  |  |
| Algorytm One-Shot |  |  |  |
| Algorytm Multishot |  |  |  |
| Algorytm Hierarchiczny |  |  |  |
| Algorytm Hierarchiczny z kolejką |  |  |  |
| Wait-Die | https://1.bp.blogspot.com/-tthXQSIZd6Y/U1yeI6sxJ4I/AAAAAAAAARs/ybKMcBf42e0/s1600/wait-die.JPG |  |  |
| Wound-Wait |  |  |  |
| Chandy-Misra-Haas Algorithm | **jeśli** *Pj* jest lokalnie zależne od siebie  **to zadeklaruj zakleszczenie**  **inaczej** dla każdego *Pj,Pk* takiego, że:  (i) *Pi* jest lokalnie zależne od *Pj*,  (ii) *Pj* oczekuje na *'Pk*,  (iii) *Pj*, *Pk* są na różnych kontrolerach  **wyślij zapytanie(i, j, k)**  **jeśli**  (i)*Pk* nie jest zajęty,  (ii) *dependentk*(i) = fałsz,  (iii)odpowiedź na zapytanie z *Pk* do *Pj*  **wykonaj**  dependentsk(i) = prawda;  **jeśli** k == i  **zadeklaruj** *Pi* jako zakleszczone  **inaczej** dla wszystkich *Pa*,*Pb* takich, że:  (i) *Pk* jest lokalnie zależne od *Pa*,  (ii) *Pa* oczekuje na '*Pb*,  (iii) *Pa*, *Pb* są na różnych kontrolerach  **wyślij zapytanie(i, j, k)** |  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## 3.4 Zakres Stosowalności:

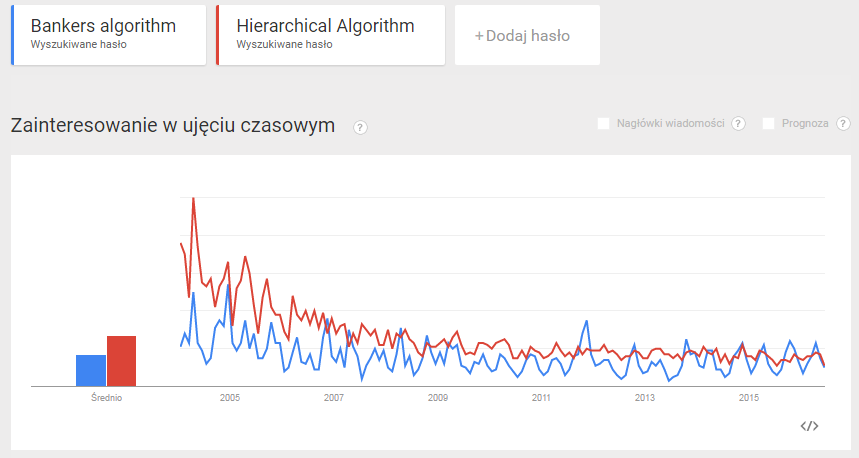
|  |  |
| --- | --- |
| Algorytm Bankiera | Dostępne: Wektor o długości m, określający liczbę dostępnych zasobów każdego typu.  Dostępne[j] = k - oznacza, że jest dostępnych k egzemplarzy zasobu typu Zj.  Maksymalne: Macierz o wymiarach n x m, definiująca maksymalne żądania każdego  procesu. Jeśli Maksymalne[i,j]=k to proces Pi może zamówić co najwyżej k egzemplarzy zasobu typu Zj.  Przydzielone: Macierz o wymiarach n x m, definiująca liczbę zasobów poszczególnych  typów, przydzielonych do każdego z procesów. Gdy Przydzielone[i, j] = k, wówczas  proces Pi ma przydzielonych k egzemplarzy zasobu typu Zj.  Potrzebne: Macierz o wymiarach n x m, przechowująca pozostałe do spełnienia  zamówienia każdego z procesów. Element Potrzebne[i,j]=k oznacza, że do  zakończenia swojej pracy proces Pi może jeszcze potrzebować k dodatkowych egzemplarzy zasobu typu Zj. |
| Algorytm grafu przydziału zasobów | Graf przydziału zasobów jest wygodną formą graficzną reprezentacji stanu sytemu na potrzeby analizy zjawisk związanych z zakleszczeniem. Poza tym można wykazać na podstawie pewnych własności tego grafu, że wystąpił pewien szczególny stan, np. właśnie stan zakleszczenia lub zagrożenia, co w połączeniu z powszechnie znanymi algorytmami analizy grafów ułatwia wykrywanie takich stanów. |
| Algorytm One-Shot | Procesy należące do pierwszej grupy nigdy nie będą blokowane, natomiast procesy należące do drugiej grupy będą blokowane w trakcie oczekiwania na zwolnienie zasobów przez grupę pierwszą. Nazywa się to alokacją *one-shot*, i implikuje to, że każdy proces w momencie dostępu do zasobów pobiera wszystkie zasoby mu potrzebne. Procesy próbujące dostać zasoby aktualnie używane przez inny proces są blokowane aż do momentu ich zwolnienia przez używający ich aktualnie proces. Każda następna próba dostępu do zasobów po ich poprzednim otrzymaniu skutkuje błędem. |
| Algorytm Multishot | Algorytm Multishot – algorytm działający wedle metody *one-shot* z jedną małą różnicą. Pozwala on na wielokrotne wysyłanie żądań przez ten sam wątek. Również posiada złożoność obliczeniową 0(**n**). Jego działanie jest zaprezentowane na Schemat Blokowy 2. |
| Algorytm Hierarchiczny | Po dostaniu zasobów wysokopriorytetowych algorytm nie będzie miał możliwości pozyskania zasobów z niskimi priorytetami do momentu aż nie odda wszystkich zasobów posiadających wyższe priorytety. Implikuje to powstanie sztucznego cyklu żądania zasobów z niższymi priorytetami, aby proces nie musiał oddawać zasobów wysokopriorytetowych w celu ich pozyskania. Procesy pozyskujące w ten sposób blokują zasoby nisko priorytetowe, pomimo że ich jeszcze nie potrzebują, przez co zabierają dostęp do nich innym procesom, które mogą ich potrzebować wcześniej. Przy używaniu tej metody rzadkie zasoby powinny dostawać wysokie priorytety, aby żądania one były wykonywane tylko wtedy, gdy są potrzebne. |
| Algorytm Hierarchiczny z kolejką | Algorytm Hierarchiczny z kolejką - Algorytm ten przestrzega metody Hierarchicznej, z taką różnicą, że procesy próbujące uzyskać dostęp do danego zasobu zostają zakolejkowane. |
| Wait-Die | **Algorytm polega na tym, że gdy transakcja Ti prosi o dane z Tj, Ti może czekać na te dane tylko wtedy gdy czas stworzenia jest mniejszy niż Tj, czyli Ti jest starszy niż Tj, w innym przypadku Ti umiera, jest później restartowany ale z randomowym opóźnieniem i tym samym czasem stworzenia. Ten algorytm jest mniej wydajny niż Wound-Wait algorytm.** |
| Wound-Wait | Algorytm stworzony przez tego samego człowieka co stworzył Wait-Die algorytm. Z tą różnicą, że ten algorytm działa na odwrót do tamtego. Jeżeli Ti ma wyższy priorytet niż Tj, anuluj Tj w innym wypadku Ti czeka. To znaczy, że gdy Ti jest młodszy niż Tj to może czekać na zwolnienie zasobów. Czyli oznacza to, że ten algorytm daje dostęp do zasobów najpierw starszy proces, jeżeli proces jest młodszy to czeka w kolejce. W poprzednim algorytmie było na odwrót, proces starszy czekał na dane. Mądrzejszym wyjściem jest jednak właśnie starszym procesom dać pierwszeństwo. |
| Chandy-Misra-Haas Algorithm | Algorytm używa wiadomości nazywanych próbkami(i,j,k) aby przesyłać wiadomości z kontrolera procesu Pj do kontrolera procesu Pk. Wiadomość jest wysyłana z procesu Pi w celu sprawdzenia czy zakleszczenie wystąpi. Każdy proces Pj przetwarza tablicę binarną zawierającą zależności od których jest on zależny. Na samym początku każda wartość tej tablicy ustawiona jest na "fałsz".  Przed wysyłaniem, proces sprawdza czy Pj jest lokalnie zależne od niego samego. Jeśli tak to występuje zakleszczenie. W przeciwnym wypadku sprawdza czy Pj i Pk są w różnych kontrolerach, czy są lokalnie zależne i czy Pj czeka na zasoby zablokowane przez Pk. Jeśli wszystkie te warunki są spełnione wysyła zapytanie.  Po stronie odbierającej, kontroler sprawdza czy Pk wykonuje zadanie. Jeśli tak to porzuca sprawdzanie. Jeśli nie, sprawdza odpowiedzi wysłane przez Pk do Pj i dependentk(i) jest nie spełniony. Po weryfikacji ustawia wartość pozytywną w tablicy dependentk(i). Jeśli w tym momencie k jest równe i to zakleszczenie występuje, jeśli nie zapytanie jest wysyłane do następnego zależnego procesu. |
| Algorytm Ho-Ramamoorthy | Klienci są zgrupowani w wiele osobnych klastrów. Cyklicznie jeden z klientów jest wybierany jak główny klient zarządzający danym klastrem. Centralny system żąda od każdego z klientów zarządzających w danym momencie statusu transakcji wewnątrz klastra oraz zależności oczekiwania na. W ten sposób wybrany klient zarządzający tworzy tablicę z danych o klientach wewnątrz klastra i uruchamia na niej jedno fazowy algorytm detekcji zakleszczeń ograniczony tylko do transakcji wewnątrz klastra. Po zakończeniu wysyła informacje o statusach transakcji i relacjach oczekiwania. Centralny system na podstawie otrzymanych danych tworzy graf stanów w którym wyszukuje cykle. |

## 3.5 Porównanie algorytmów wraz z ze znaną realizacją wdrożeń :

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Nazwa  algorytmu | Algorytm Bankiera | Algorytm grafu przydziału zasobów | Algorytm One-Shot | Algorytm Multishot | Algorytm Hierarchiczny | Algorytm Hierarchiczny z kolejką | Wait-Die | Wound-Wait | Chandy-Misra-Haas Algorithm | Algorytm Ho-Ramamoorthy |
| Złożoność Obliczeniowa |  |  |  |  |  |  |  | |  |  |
| Typ algorytmu | avoidance | avoidance | Detection | Detection | Detection | Detection | Avoidance | | Detection | Detection |
| Zastosowanie | Bankowość | Prowadzenie Firm, Systemy w medycynie | Brak danych gdzie używany | | | Stosowany w Windows | Systemy bazodanowe | | Brak danych | Brak danych |
| Dostępność źródeł | Duża | Mała | Mała | Mała | Średnia | Mała | Średnia | | Średnia | Mała |
| Popularność | Duża(często stosowany ) | Średnia | Mała | mała | Mała | Średnia | Duża(często w bazodanowych systemach. Wspominany przy okazji zakleszczeń) | | średnia | Mała |

## 3.6 Trudności implementacyjne:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Algorytm Bankiera | Algorytm grafu przydziału zasobów | Algorytm One-Shot | Algorytm Multishot | Algorytm Hierarchiczny | Algorytm Hierarchiczny z kolejką | Wait-Die | Wound-Wait | Chandy-Misra-Haas Algorithm | Algorytm Ho-Ramamoorthy |
| Ilość linii | 7 | 6 | 6 | 6 | 6 | 9 | 5 | | 16 | Brak pseudokodu |
| Ocena trudności implementacji(łatwo:1-5:Trudno) | 3 | 3 | 4 | 4 | 4 | 4 | 2 | | 5 | 5 |
| Opis trudności | Dużo linii kodu, pomimo że 7 kroków to jednak każda funkcja musi być zaimplementowana z użyciem większej liczby linii do zaimplementowania. Dodatkowo dochodzą trudności implementacyjne w różnych technologiach. | Ze względu na średnią popularność mało źródeł pomocnych przy implementacji. Dodatkowo dochodzą trudności implementacyjne w różnych technologiach. | Algorytmy bardzo mało znane przez co liczba źródeł odnośnie tego algorytmu mocno ograniczona. | Bardziej niż trudności co do tego algorytmu jest pewna niechęć do implementacji ze względu na fakt, że jest to niezbyt popularny algorytm ponadto sam algorytm kiepsko się sprawuje w swoim zadaniu. | Algorytmy hierarchiczne nie dość że mają dużą ilość kroków to jeszcze podczas poszukiwań źródeł na Internecie nie znaleźliśmy wystarczająco dużo informacji odnośnie implementacji. Dodatkowo dochodzą trudności implementacyjne w różnych technologiach. | | Trudności w zasadzie nie ma, bardzo prosty kod i bardzo prosta zasada działania. | | Duża liczba kodu do napisania, bardzo mało książek i źródeł dostępnych odnośnie tego algorytmu. | Pomimo ubogiej liczby źródeł to jeszcze brak pseudokodu odnośnie tego algorytmu. |



Tylko tyle algorytmów było przewidzianych w google Trend, reszta nie miała w ogóle opcji aby je wyszukać. Na podstawie tego jestem w stanie stwierdzić, że najprawdopodobniej są to bardzo popularne algorytmy. Jak możemy zauważyć z czasem algorytm bankiera stał się bardziej popularny niż hierarchiczny.

WYBRANY ALGORYTM

|  |  |
| --- | --- |
| NAZWA | Algorytm Bankiera |
| Dlaczego wybrany | Algorytm wybrany na podstawie tego, iż jest najbardziej popularnym z przeglądanych przez nas algorytmów. Cechuje się najlepszą jakością zwalczania deadlock’a. Z podanych informacji jest to algorytm typu avoidance, jest wykorzystywany w systemach bankowych dlatego uznaliśmy, że jest to bardzo solidny algorytm ze względu na systemy w których jest wykorzystywany. Pomimo dość dużej złożoności obliczeniowej względem innych, zdecydowaliśmy się na ten algorytm ze względu na skuteczność unikania zakleszczeń. |

1. **Użyte technologie rozproszone:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Socket | Corba | RMI | RPC |
| Rok powstania | 1983 | 1991 | 1997 | 1984 |
| Obsługiwane języki | C, C++, Java, C#, Visual Basic, Python, Fortran | C, C++, Java, COBOL, Ada, Lisp, Python | Java, C# | C |
| Platformy | Windows, Unix | Windows, Unix | Windows, Unix | Unix |
| Język opisu interfejsu | API | CORBA IDL | Java | XDR |
| Protokół transportowy | TCP//IP  UDP/IP | IIOP | JRMP | TCP//IP  UDP/IP |
| Format komunikatów | binarny | binarny | binarny | binarny |
| Obsługa wyjątków | częściowo | tak | tak | tak |
| Popularność | duża | niska | umiarkowana | umiarkowana |

Tabela przedstawia zestawienie porównawcze cech i możliwości wybranych technologii rozproszonych. Analizując powyższe zestawienie można stwierdzić, iż najbardziej elastycznymi mechanizmami pod względem platformy sprzętowej sa gniazda oraz RMI. W kwestii obsługiwanych języków programowania przodują Socety natomiast najmniejsza liczbę posiada RPC. Wszystkie technologie posiadają binarna formę komunikatów. Pod względem popularności na pierwszym miejscu są Socety, a DCOM jest technologią o niskiej popularności.

|  |  |  |
| --- | --- | --- |
| Technologia | Zalety | Wady |
| Gniazda | - proste w implementacji  - wbudowany system obsługi w systemie Linux  - duża popularność  - możliwość implementacji w wielu językach programowania | - nieprzenośność implementacji pomiędzy systemem operacyjnym Linux a Windows |
| RMI | - proste do wykonania  - wykorzystuje model bezpieczeństwa Javy  - dynamiczne tworzenie interfejsu | - technologia może działać wolniej niż inne alternatywne  - kod RMI jest wydajniejszy w języku Java |
| RPC | - niezależny serwer  - prosty rozwój systemów rozproszonych  - procedura połączenia zachowuje logikę biznesu  - umożliwia korzystanie z aplikacji używanych w środowisku rozproszonym | - przełączanie kontekstu zwiększa koszty planowania  - brak standaryzacji  - nie rozwiązuje większości problemów związanych z tworzeniem dystrybucji |
| Cobra | - technologia szybsza od RMI  - pozwala na używanie różnych platform systemowych  - wszechstronna | - skomplikowana w implementacji  - nieprzyjazny dla użytkownika |

1. **Analiza porównawcza implementacji:**

Wykres 1 Trudność Implementacji - Im mniej tym lepiej

Wykres 2 Czas implementacji w godzinach - Im mniej tym lepiej

Wykres 3 Czas wykonania w milisekundach - Im mniej tym lepiej

Wykres 4 Ilość linii kodu

Wykres 5 Rozmiar w kilobajtach - im mniej tym lepiej

Wykres 6 Ilość komputerów, na której działa implementacja - Im więcej tym lepiej

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Imię i Nazwisko | Michał  Czarnik | Przemysław  Duda | Łukasz  Basiura | Adrian  Sołtys |
| Technologia | Sockety | Corba | RMI | RPC |
| Język | Python | C++ | JAVA | C |
| Ilość komputerów, na której działa implementacja | 6 | 1 | 3 | 4 |
| Dokumentacja | TAK | NIE | NIE | TAK |
| Trudność (1-5) 1-Najłatwiejsze 5-Najtrudniejsze | 2 | 5 | 4 | 5 |
| Czas implementacji w godzinach | 3 | 6 | 5 | 3 |
| Czas wykonania w milisekundach | 93.7 | 20.4 | 194 | 400 |
| Ilość linii kodu | 187 | 113 | 143 | 611 |
| Rozmiar w kilobajtach | 5 | 4 | 188 | 54 |

Tabela 1 Analiza tabelaryczna implementacji

Bibliografia:

1. Zakleszczenia, M.Kubica; K.Stencla, [online], [dostęp 03.06.2016], dostępny w Internecie: http://edu.pjwstk.edu.pl/wyklady/sop/scb/wyklad6/wyklad.html
2. Przeciwdziałania zakleszczeniu, J. Brzeziński, D. Wawrzyniak, [online], [dostęp 02.06.2016], dostępny w Internecie: http://wazniak.mimuw.edu.pl/images/e/ef/Sop\_16\_wyk\_1.1.pdf
3. K. Elleithy, „*Advances and Innovations in Systems, Computing Sciences and Software Engineering*”, Springer, 2007, ISBN 978-1-4020-6264-3
4. Corman, T. H, Leiserson, C. E., Rivest, R. L., “*Wprowadzenie do algorytmów”,* Wydawnictwo Naukowo- Techniczne, Warszawa 2001, ISBN:83-204-2665-0
5. Deadlock Prevention algorithms in Database, [online], [dostęp 12.06.2016], dostępny w Internecie: https://exploredatabase.blogspot.com/2014/04/deadlock-prevention-algorithms-in.html
6. How to Handling a Deadlocks?, [online], [dostęp 12.6.2016], dostępny w Internecie: http://ecomputernotes.com/database-system/rdbms/handling-a-deadlocks
7. R. Chhanda, „*Distributed Database Systems, Pearson Education India”*, Indie 2009, rozdział IX, [dostęp 12.06.2016], dostępna w Internecie: https://books.google.pl/books?id=0i9uJJ78LtAC&pg=PA145&lpg=PA145&dq=wound+and+wait+deadlock+prevention&source=bl&ots=wW37\_wrT2b&sig=qCF8PKekCdM5fVdwtL9jhaZySkQ&hl=pl&sa=X&ved=0ahUKEwia-LryvKPNAhXEJJoKHeijCKw4FBDoAQgbMAA#v=onepage&q=wound%20and%20wait%20deadlock%20prevention&f=false
8. Deadlocks in Transactions, [online], [dostęp 12.06.2016], dostępny w Internecie: http://www.edugrabs.com/deadlock-in-transaction/
9. S. K. Rahimi; F. S. Haug, “*Distributed Database Management Systems”*, A John Wiley & Sons Inc, [dostęp 12.06.2016], dostępny w Internecie: http://learngroup.org/uploads/2015-10-02/Distributed\_Database\_Management\_Systems\_Leangroup\_org.pdf
10. Ahmed K. Elmagarmid, „*A survey of distributed deadlock detection algorithms”*, nr 15, s. 37-45, [dostęp 13.06.2016], dostępny w Internecie: http://www.ics.uci.edu/~cs237/reading/p37-elmagarmid.pdf
11. Chrobot A, Systemy Operacyjne — Zakleszczenia, [online],[dostęp 16.06.2016], dostępny w Internecie: Achilles.tu.kielce.pl/Members/achrobot/archiwum/i-semestr/pdf/so/os\_lect\_6ee.pdf
12. Singhal M., Deadlock Detection in Distributed Systems, “Computer”, 1989, nr 11, s. 37-48, [dostęp 15.06.2016], dostępny w Internecie: http://www.kiv.zcu.cz/~ledvina/ds/~singhaldeadsurvey.pdf
13. A. Kshemkalyani; M. Singhal, Deadlock Detection in Distributed Systems, [online], [dostęp 14.06.2016], dostępny w Internecie: https://www.cs.uic.edu/~ajayk/Chapter10.pdf
14. Kaveh N. Wolfgang E., “*Deadlock Detection in Distributed Object Systems”*, [online], [dostęp 14.06.2016], dostępny w Internecie: http://www0.cs.ucl.ac.uk/staff/w.emmerich/publications/ESEC01/ModelChecking/esec.pdf
15. Coffman, E. G. Jr.; M.J. Elphick; A. Shoshani, (1971). System Deadlocks, “Computing Surveys “ s. 67–78, nr 2/3, [dostęp 15.06.2016], dostępny w Internecie: https://people.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman\_deadlocks/coffman\_deadlocks.pdf.
16. Finkel, R. A. (1998). "*Operating Systems Vade-mecum"* (Tom 2nd Revised edition). Upper Saddle River, New Jersey: Prentice Hall International Paperback Editions.
17. Gonzalez, T., Diaz-Herrera, J. i Tucker, A., "*Computing Handbook; Third Edition: Computer Science and Software Engineering",* Chapman and Hall/CRC, Londyn 2014, ISBN: 1439898529
18. Hamilton, H. J., "*Deadlock Prevention Algorithms"*, [online], [dostęp 16.06.2016], dostępny w Internecie: http://www2.cs.uregina.ca/~hamilton/courses/330/notes/synchro/deadlock-ex.html

19. Karbowski, A., Niewiadomska-Szynkiewicz, E., "*Programowanie równoległe i rozproszone"*, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2009, ISBN:978-83- 7207-803-2

20. McHale C., CORBA Explained Simply,[online],[dostep 15.06.2016], dostepny w Internecie: www.ciaranmchale.com/download/**corba**-explained-simply.pdf

21. Henning M., "*CORBA/C++ Programming with ORBacus*", IONA Technologies, Waltham 2001

22. Python 3.6.0a1 documentation,[online],[dostep:15.06.2016], dostępny w Internecie:

https://docs.python.org/3.6/

23. Downey, Allen. Think Python,[online],[dostep:15.06.2016],dostepny w Internecie:

<http://www.greenteapress.com/thinkpython/thinkpython.pdf>