Wzajemne wykluczanie i zakleszczenie

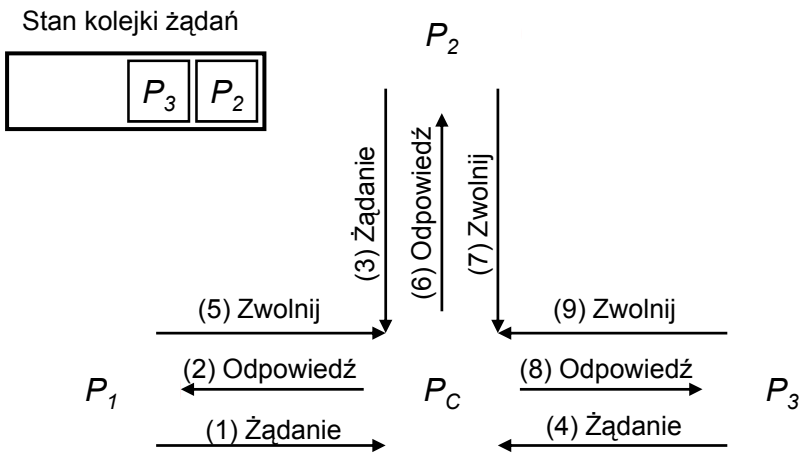
Wzajemne wykluczanie

- Wzajemne wykluczenie zapewnia procesom ochronę przy dostępie do zasobów, daje im np. gwarancję, że jako jedyne będą mogły z nich korzystać
- Typy algorytmów:
  - Podejście scentralizowane
  - Algorytmy rozproszone
    - algorytm Lamporta
  - Algorytmy bazujące na żetonie
    - algorytm Suzuki-Kasami

Podejście scentralizowane

1. Jeden proces jest wybierany jako koordynator
2. Proces *P*, który chce wejść do sekcji krytycznej wysyła wiadomość do koordynatora
3. Jeżeli inny proces nie przebywa aktualnie w danej sekcji krytycznej, odsyła pozwolenie do *P*
4. Po otrzymaniu pozwolenia *P* wchodzi do sekcji krytycznej
5. Jeżeli w tym samym czasie do tej samej sekcji krytycznej chce się dostać inny proces, jego prośba jest rozpatrywana później lub otrzymuje wiadomość odmowną

Podejście scentralizowane – przykład



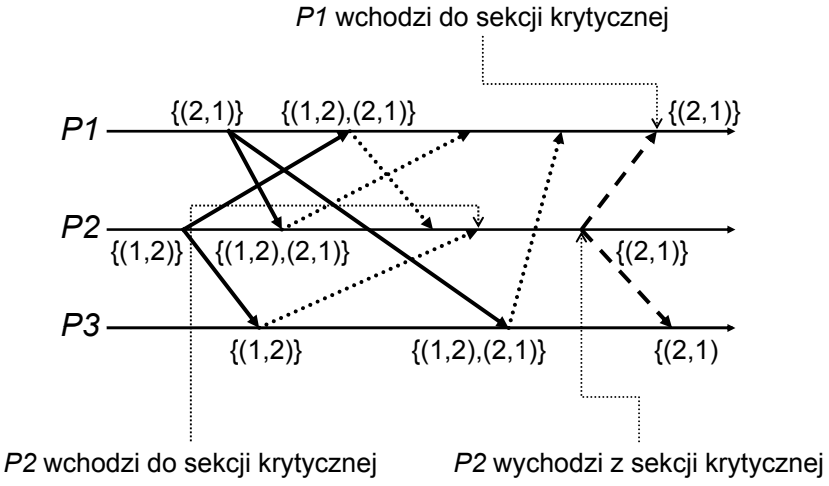
Algorytm Lamporta – wprowadzenie

- Wykorzystuje mechanizm synchronizacji zegarów Lamporta
- **Zbiór żądań** – zbiór procesów, od których wymagane są pozwolenia na wejście do sekcji krytycznej
- Zbiór żądań w algorytmie Lamporta jest zbiorem wszystkich procesów
- Każdy proces przechowuje kolejkę żądań sekcji krytycznej uszeregowanych według znaczników czasowych

Algorytm Lamporta

1) Żądanie sekcji krytycznej w procesie *Pi*
   - wysłanie żądania ze znacznikiem czasu (*ts*(*i*),*i*) do wszystkich procesów ze zbioru *Ri*
   - dodanie żądania do kolejki (również lokalnie)
   - odesłanie odpowiedzi ze znacznikami czasu
2) Wejście do sekcji krytycznej, gdy:
   - odpowiedzi od wszystkich procesów *Pj* mają znaczniki czasowe (*ts(j)*,*i*) większe od znacznika żądania
   - żądanie to jest na początku kolejki procesu żądającego
3) Zwalnianie sekcji krytycznej
   - wysłanie wiadomości *ZWOLNIJ* do innych procesów
   - usunięcie żądania z początku kolejki

Algorytm Lamporta – przykład



Algorytm Suzuki-Kasami – wprowadzenie (1)

- W algorytmie Suzuki-Kasami wykorzystywany jest żeton, o który ubiegają się procesy chcące wejść do sekcji krytycznej
- Proces, który posiada żeton może wchodzić do sekcji krytycznej do czasu, gdy nie poprosi o niego inny proces
- Pojawiają się problemy, co zrobić ze:
  - starymi (przedawnionymi) żądaniami
  - zaległymi żądaniami

Algorytm Suzuki-Kasami – wprowadzenie (2)

- Problem przedawnionych żądań
  - Żądania mają postać *ŻĄDANIE(j, n)*, gdzie *n* oznacza, że proces *Pj* żąda *n*-tego wykonania sekcji krytycznej
  - *RNi[1..N]* jest tablicą przechowywaną przez proces *Pi*; *RNi[j]* oznacza największą liczbę porządkową otrzymaną w żądaniu od procesu *Pj*
  - *ŻĄDANIE(j, n)* otrzymane przez *Pi* jest przedawnione jeżeli *RNi[j]>n*
  - Po otrzymaniu przez *Pi* wiadomości *ŻĄDANIE(j, n), RNi[j] = max(RNi[j], n)*

Algorytm Suzuki-Kasami – wprowadzenie (3)

- Problem zaległych żądań
  - Zaległe żądania określane są przy użyciu zawartości żetonu, który składa się z **kolejki żetonu** *Q* i **tablicy żetonu** *LN*
  - *Q* kolejką procesów żądających
  - *LN* jest tablicą o rozmiarze *N*, a *LN[j]* oznacza liczbę porządkową ostatnio wykonanego żądania przez proces *Pj*
  - Po wykonaniu sekcji krytycznej proces *Pi* aktualizuje tablicę *LN: LN[i]* := *RNi[i]*
  - Na podstawie tablicy *LN* można stwierdzić, czy jakiś proces ma zaległe żądanie

Algorytm Suzuki-Kasami

1) Żądanie sekcji krytycznej
   - wysłanie przez proces żądania o żeton, jeżeli go nie ma
   - odesłanie wolnego żetonu
2) Wejście do sekcji po otrzymaniu żetonu
3) Zwalnianie sekcji krytycznej
   - aktualizacja tablicy oraz kolejki żetonu
   - wysłanie żetonu do następnego procesu w kolejce

Algorytm Suzuki-Kasami

1) Żądanie sekcji krytycznej
   - Jeśli proces *Pi* nie ma żetonu, to inkrementuje wartość *RNi*[*i*] i wysyła *ŻĄDANIE(i, sn*), gdzie *sn=RNi[i]*
   - Kiedy proces *Pj* otrzymuje ŻĄDANIE, to uaktualnia RNj[i]=max(RNj[i], sn)
   - Jeżeli Pj ma nieużywany token to wysyła go do procesu Pi, jeśli RNj[i]=LN[i] + 1
2) Wejście do sekcji po otrzymaniu żetonu
   - Proces Pi wchodzi do sekcji krytycznej po otrzymaniu żetonu

Algorytm Suzuki-Kasami

3) Wyjście procesu *Pi* z sekcji krytycznej
   - LN[i] = RNi[i]
   - Dla każdego procesu *Pj*, którego identyfikatora nie ma w kolejce żetonu, jego identyfikator dodawany jest do kolejki *Q*, jeśli spełniony jest warunek *RNi[j]=LN[j]+1*.
   - Jeżeli kolejka *Q* nie jest pusta, *Pi* usuwa identyfikator z początku kolejki i wysyła żeton do procesu oznaczonego tym identyfikatorem.

Zakleszczenie - wprowadzenie

Procesy tworzące przetwarzanie rozproszone komunikują się ze sobą za pomocą mechanizmu wymiany wiadomości, realizując wspólny cel przetwarzania. Jedne procesy wysyłają komunikaty zawierające żądania przydziału pewnych zasobów, inne – w odpowiedzi – przesyłają ewentualnie komunikaty potwierdzające przydział żądanych zasobów.

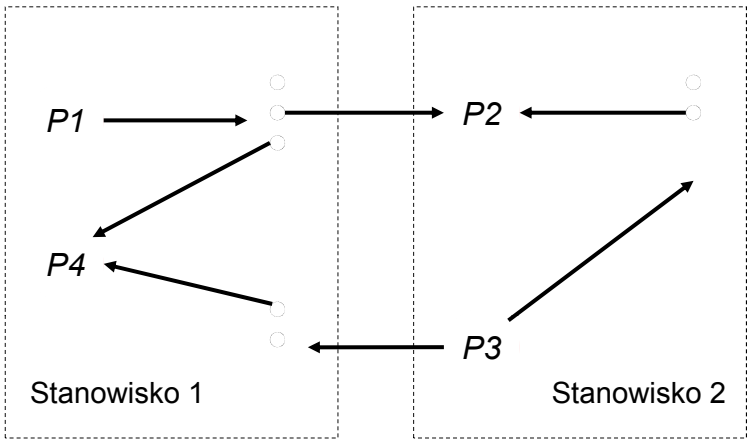
Podejście rozproszone - nieformalna definicja problemu

W przetwarzaniu rozproszonym może w ogólności wystąpić sytuacja, w której wszystkie procesy pewnego niepustego zbioru procesów oczekują na wiadomości (potwierdzające na przykład przydział zasobów) od innych procesów tego właśnie zbioru. Stan taki nazywany jest **zakleszczeniem rozproszonym** (ang. distributed deadlock).

Warunki konieczne zakleszczenia

1. Wzajemne wykluczanie
2. Istnienie procesu, który blokuje zasób, a jednocześnie sam czeka na zasób blokowany przez inny proces
3. Brak wywłaszczania zasobów
4. Czekanie cykliczne

Graf przydziału zasobów



Strategie postępowania z zakleszczeniami

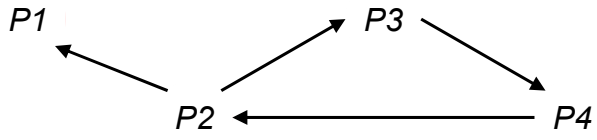
- Sposoby postępowania z zakleszczeniami
  - niedopuszczanie do zakleszczeń
  - dopuszczanie do zakleszczeń i późniejsze ich usuwania
  - ignorowanie zakleszczeń
- Metody niedopuszczania do zakleszczeń
  - zapobieganie zakleszczeniom
  - unikanie zakleszczeń

Zapobieganie zakleszczeniom w systemach rozproszonych

- Przydział priorytetów dla procesów przy dostępie do zasobów
- Zastosowanie znaczników czasowych – możliwość pozbycia się problemu zagłodzenia procesów o niskich priorytetach. Metody:
  1) *Czekanie albo śmierć* (ang. *wait-die*),
  2) *Zranienie albo czekanie* (ang. *wound-wait*)
- Wadą powyższych algorytmów jest występowanie niepotrzebnych wywłaszczeń

Wykrywanie zakleszczeń

- Do wykrywania zakleszczeń używa się *grafu oczekiwania* (WFG), który reprezentuje stan przydziału zasobów
- Jeżeli stan przedstawiany przez graf dotyczy całego systemu rozproszonego mówimy o *globalnym grafie oczekiwania*
- Jeżeli stan, który reprezentuje graf dotyczy tylko danego stanowiska, to jest to *lokalny graf oczekiwania*



Procesy aktywne i pasywne

W każdej chwili proces może być w jednym z dwóch stanów: **aktywnym** albo **pasywnym**.

Proces aktywny może realizować przetwarzanie wykonując operacje odpowiadające zajściu zdarzeń wewnętrznych i komunikacyjnych.

Procesy aktywne i pasywne

W stanie **pasywnym** procesu P_i ($passive_i$=$True$) dopuszczalne są natomiast co najwyżej zdarzenia odbioru.

Zmiana stanu procesu z pasywnego na aktywny uwarunkowana jest osiągnięciem gotowości przez choćby jedno z dopuszczalnych zdarzeń odbioru, czyli spełnieniem tak zwanego **warunku uaktywnienia**.

Warunek uaktywnienia

Warunek uaktywnienia (ang. *activation condition*) procesu P_i związany jest ze zbiorem warunkującym $\mathcal{D}_i$, zbiorem $\mathcal{P}_i^A$, oraz predykatem $activate_i(\mathcal{X})$.

Zbiór warunkujący

Zbiór warunkujący (ang. *dependent set*), jest sumą mnogościową zbiorów $\mathcal{P}_i^S$ wszystkich zdarzeń odbioru dopuszczalnych w danej chwili.

Warunek uaktywnienia

Warunek uaktywnienia jest wyrażony przez predykat:

$$ready_i(\mathcal{X}) \equiv (\mathcal{P}_i^A \supseteq \mathcal{X}) \wedge activate_i(\mathcal{X})$$

Gdy proces jest uaktywniany, to wiadomości, których dostarczanie doprowadziło do spełnienia warunku uaktywnienia, są atomowo pobierane z buforów wejściowych i dalej przetwarzane.

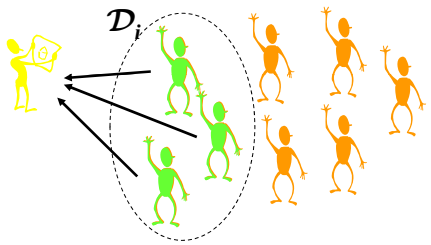
Definicja problemu

Przez $deadlock(\mathcal{B})$ oznaczamy predykat stwierdzający, że w danej chwili τ, niepusty zbiór procesów $\mathcal{B}$ jest zbiorem procesów zakleszczonych.

Model *AND*

W modelu *AND* proces pasywny staje się aktywnym, jeżeli dotarły wiadomości od wszystkich procesów tworzących zbiór warunkujący.

Model ten nazywany jest również **modelem zasobowym**.



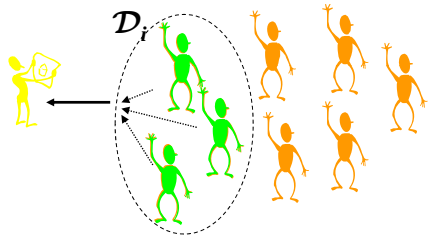
Zakleszczenie w modelu AND

$$deadlock(\mathcal{B}) \equiv$$
$$(\ \mathcal{B} \subseteq \mathcal{P}\ ) \wedge (\ \mathcal{B} \neq 0) \wedge$$
$$(\forall P_i :: P_i \in \mathcal{B} :: (\ passive_i \wedge$$
$$(\exists\ P_j :: P_j \in \mathcal{D}_i \cap \mathcal{B} :: (\neg\ in\text{-}transit_i[j] \wedge \neg\ available_i[j]\ ))))$$

Model *OR*

W modelu *OR* do uaktywnienia procesu wystarczy jedna wiadomość od któregokolwiek z procesów ze zbioru warunkującego.

Model ten nazywany jest również modelem *komunikacyjnym*.

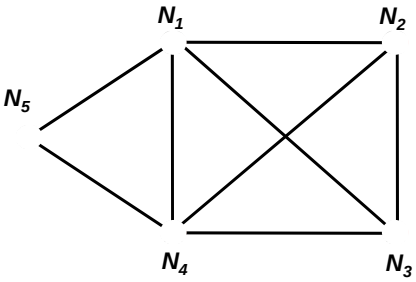


Zakleszczenie w modelu OR

$$
\begin{aligned}
&deadlock(\mathcal{B}) \equiv \\
&\quad (\ \mathcal{B} \subseteq \mathcal{P}\ ) \wedge (\ \mathcal{B} \neq 0) \wedge \\
&\quad (\forall P_i :: P_i \in \mathcal{B} :: (\ passive_i \wedge \\
&\qquad \mathcal{D}_i \subseteq \mathcal{B} \ \wedge \\
&\qquad (\forall\ P_j :: P_j \in \mathcal{D}_i :: (\neg\ in\text{-}transit_i[j] \wedge \neg\ available_i[j]\ ))))
\end{aligned}
$$

Przykłady zakleszczeń

***Wait-For Graph* (*WFG*):**



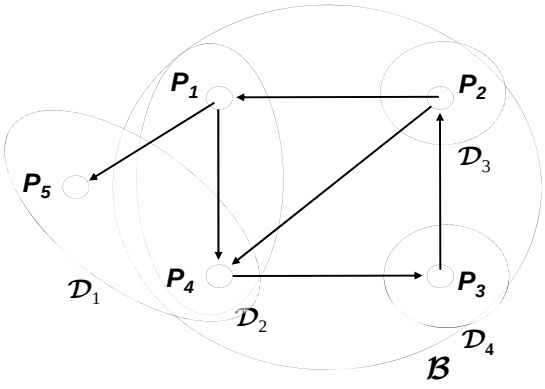
N_1, N_2, N_3, N_4, N_5 – węzły środowiska przetwarzania

P_1, P_2, P_3, P_4, P_5 – procesy wykonywane w odpowiednich węzłach

$P_i \longrightarrow P_j$ – łuk $\langle P_i, P_j \rangle$ reprezentujący fakt, że proces P_i, oczekuje na wiadomość od procesu P_j

- Każdy proces w grafie z łukiem wychodzącym jest pasywny.
- Procesy bez łuków wychodzących są aktywne.
- Zakładamy, że wszystkie kanały są puste.

Przykład – model AND

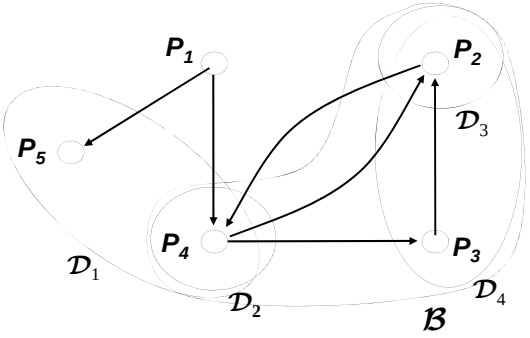


Zbiory warunkujące:

$\mathcal{D}_1 = \{P_4, P_5\}$,
$\mathcal{D}_2 = \{P_1, P_4\}$,
$\mathcal{D}_3 = \{P_2\}$,
$\mathcal{D}_4 = \{P_3\}$.

deadlock($\mathcal{B}$) zachodzi dla $\mathcal{B} = \{P_1, P_2, P_3, P_4\}$

Przykład – model OR



Zbiory warunkujące:

$\mathcal{D}_1 = \{P_4, P_5\}$,
$\mathcal{D}_2 = \{P_4\}$,
$\mathcal{D}_3 = \{P_2\}$,
$\mathcal{D}_4 = \{P_2, P_3\}$.

deadlock($\mathcal{B}$) zachodzi dla $\mathcal{B} = \{P_2, P_3, P_4\}$

Klasyfikacja problemów detekcji zakleszczenia

- problem detekcji wystąpienia zakleszczenia
- problem detekcji zakleszczenia procesu
- problem detekcji zakleszczenia zbioru procesów
- problem detekcji maksymalnego zbioru procesów zakleszczonych

Detekcja wystąpienia zakleszczenia

Czy istnieje w pewnej chwili zbiór $\mathcal{B}$, dla którego predykat *deadlock*($\mathcal{B}$) jest prawdziwy ?

Odpowiedź na to pytanie określa wartość predykatu:

$$dE \equiv (\exists \mathcal{B} :: deadlock(\mathcal{B})) \qquad (5.9)$$

Detekcja wystąpienia zakleszczenia procesu

Detekcja zakleszczenia procesu P_i sprowadza się do sprawdzenia czy prawdziwy jest predykat:

$$dP_i \equiv (\exists \mathcal{B} :: deadlock(\mathcal{B})) \wedge P_i \in \mathcal{B}) \qquad (5.10)$$

Detekcja wystąpienia zakleszczenia zbioru procesów

Detekcja zakleszczenia zbioru procesów polega na znalezienia zbioru $\mathcal{B}^*$, dla którego prawdziwy jest następujący predykat:

$$deadlock(\mathcal{B}^*) \vee ((\mathcal{B}^* = \emptyset) \wedge (\nexists \mathcal{B} :: deadlock(\mathcal{B}))) \quad (5.11)$$

Detekcja maksymalnego zbioru zakleszczonego

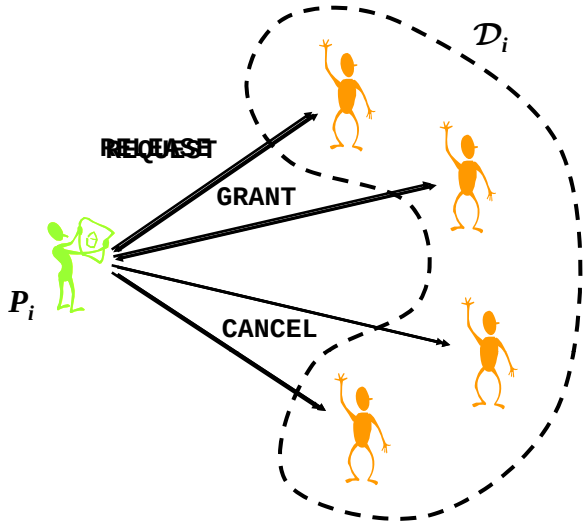
Detekcja maksymalnego zbioru zakleszczonego, sprowadza się do znalezienia takiego zbioru $\mathcal{B}^*$, dla którego spełniony jest warunek:

$$(deadlock(\mathcal{B}^*) \vee \mathcal{B}^* = \emptyset) \wedge (maxdead(\mathcal{B}^*)), \quad (5.12)$$

gdzie

$$maxdead(\mathcal{B}^*) \equiv (\forall \mathcal{B} :: deadlock(\mathcal{B}) \Rightarrow (\mathcal{B} \subseteq \mathcal{B}^*)) \quad (5.13)$$

Model aplikacyjnego przetwarzania rozproszonego



Alg. Chandy, Misra, Hass dla modelu AND (1)

```
type PROBE extends FRAME is
record of
   initIndex : INTEGER
end record

probeOut    : PROBE
granted_i   : array [1..n] of BOOLEAN := False
D_i         : set of PROCESS_ID
recvProbe_i : array [1..n] of BOOLEAN := False
α_i         : INTEGER
k           : INTEGER
deadlockDetected_i  : BOOLEAN := False
```

Alg. Chandy, Misra, Hass dla modelu AND (2)

```
1.  when e_start(Q_α , DeadlockDetection) do
2.   if passive_α
3.     then
4.        for all Q_k :: P_k ∈ 𝒟_α do
5.        probeOut.initIndex := α
6.        send(Q_α, Q_k, probeOut)
7.    end for
8.    end if
9.  end when
```

Alg. Chandy, Misra, Hass dla modelu AND (3)

```
10. when e_activate(P_i) do
11.       for all k ∈ {1, 2, ... , n} do
12.          recvProbe_i[k]:= False
13.       end for
14. end when
```

Alg. Chandy, Misra, Hass dla modelu AND (4)

```
15.  when e_receive(Q_j, Q_i, probeIn: PROBE) do
16.   α_i := probeIn.InitIndex
17.   if passive_i ∧ (¬recvProbe_i[α_i]) ∧ (¬granted_i[j])
18.   then
19.      recvProbe_i[α_i] := True
20.      if α_i = i
21.      then
22.         deadlockDetected_i := True
23.         decide(deadlockDetected_i)
24.      else
25.         probeOut.initIndex := α_i
26.         for all Q_k :: P_k ∈ D_i do
27.            send(Q_i , Q_k , probeOut)
28.         end for
29.      end if
30.   end if
31.   end when
```

Detekcja zakleszczenia dla modelu *OR* (1)

Algorytm detekcji zakleszczenia dla modelu *OR* opiera się na **przetwarzaniu dyfuzyjnym** (ang. *query computation*).

Detekcja zakleszczenia dla modelu *OR* (2)

Twierdzenie 5.2
Jeżeli inicjator Q_α rozpoczyna detekcję w chwili, gdy jego proces aplikacyjny P_α jest zakleszczony, to Q_α stwierdzi zakleszczenie procesu P_α w skończonym czasie (algorytm detekcji zakończy się).

Detekcja zakleszczenia dla modelu *OR* (3)

Twierdzenie 5.2
Jeżeli inicjator Q_α deklaruje, że jego proces aplikacyjny P_α jest zakleszczony, to P_α należy do pewnego zbioru procesów zakleszczonych w chwili zakończenia algorytmu.

Alg. Chandy, Misra, Hass dla modelu OR (1)

```
type CONTROL extends FRAME is
   record of
    initIndex : INTEGER
    queryNo   : INTEGER
   end record

type QUERY extends CONTROL

type REPLY extends CONTROL
```

Alg. Chandy, Misra, Hass dla modelu OR (1)

- zablokowany proces inicjuje proces wykrywania zakleszczenia wysyłając wiadomość QUERY do wszystkich procesów ze swojego zbioru warunkującego
- Kiedy aktywny proces otrzymuje wiadomość QUERY lub REPLY unieważnia ją
- Kiedy zablokowany proces Pk otrzymuje wiadomość QUERY(i,j,k) podejmuje następujące działania:

Alg. Chandy, Misra, Hass dla modelu OR (2)

1. Jeśli to pierwsza wiadomość QUERY otrzymana przez Pk, i zainicjowana przez Pi (engaging query), Pk propaguje QUERY do wszystkich procesów ze swojego zbioru warunkującego, i nadaje lokalnej wartości numk(i) wartość równą liczbie wysłanych wiadomości

2. Jeśli otrzymana wiadomość nie jest pierwsza i Pk był jest zablokowany od otrzymania pierwszej wiadomości QUERY of Pi, to Pk zwraca REPLY. W przeciwnym razie ignoruje otrzymaną wiadomośći

Alg. Chandy, Misra, Hass dla modelu OR (3)

- Proces Pk utrzymuje zmienną logiczną waitk(i) oznaczającą fakt, że jest zablokowany od otrzymania odtatniej wiadomości QUERY.
- Kiedy zablokowany proces otrzymuje wiadomość REPLY(i,j,k), zmniejsza wartość numk(i) jeśli zachodzi waitk(i)
- Proces odsyła odpowiedź na wiadomość QUERY, dopiero po uzyskaniu wiadomości REPLY na każdą wiadomość QUERY, którą on sam wysłał.
- Inicjator wykrywa zakleszczenie po otrzymaniu wiadomości REPLY na wszystkie wiadomości QUERY, które wysłał.

Alg. Chandy, Misra, Hass dla modelu OR (4)

Initiate a diffusion computation for a blocked process *Pi:*

send *query(i, i, j) to all processes Pj in the dependent set DSi of Pi;*

numi(i) = DSi;
waiti(i) =*true;*

Alg. Chandy, Misra, Hass dla modelu OR (5)

When a blocked process Pk receives a query(i, j, k):
if this is the engaging query for process Pi then
send query(i, k, m) to all Pm in its dependent set DSk;
numk(i) = DSk;
waitk(i) =true
else
if waitk(i) then send a reply(i,k,j) to Pj .

Alg. Chandy, Misra, Hass dla modelu OR (6)

When a process Pk receives a *reply(i,j,k):*

```
    if waitk(i) then
        numk(i) = numk(i)−1;
        if numk(i) = 0 then
            if i = k then declare a
                        deadlock
        else
            send reply(i, k, m) to the
            process Pm
            which sent the engaging
            query.
.
```