Programowanie Systemowe i Współbieżne Repetytorium

Wróg publiczny nr 1
Wróg publiczny nr 2
anonim
anonim
anonim

8 godzin przed PSiW

Spis treści

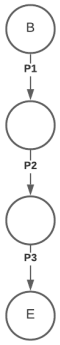
1 Wstęp

Poniższe 45 stron wiedzy jest wynikiem wielu godzin prawdziwej współpracy. Materiały powstały w trakcie przygotowywania się do egzaminu z PSiW, który odbył się dnia 01.02.2021 roku. Praca nad materiałami zakończyła się tego samego dnia, 8 godzin przed egzaminem, a w pliku brakuje niektórych informacji podanych na wykładach. Dogłębne zapoznanie się z całością dokumentu powinno być wystarczające do zaliczenia egzaminu w pierwszym terminie, niemniej jednak autorzy nie podejmują żadnej odpowiedzialności za możliwe ewentualne błędy zarówno w części teoretycznej jak i praktycznej. Powodzenia w nauce!

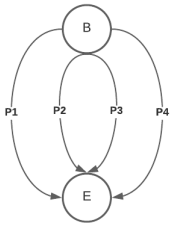
2 Modelowanie współbieżności

2.1 Graf przepływu procesów

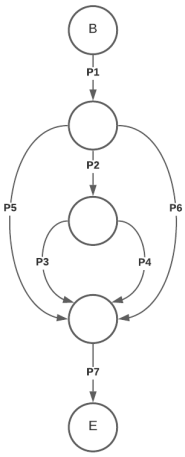
Grafy przepływu procesów przedstawiają zależności czasowe wykonywania procesów. Wierzchołki tych grafów reprezentują chwile czasu, natomiast krawędzie zorientowane - procesy. Dwa wierzchołki są połączone krawędzią zorientowaną (łukiem) jeżeli istnieje proces, którego moment rozpoczęcia odpowiada pierwszemu wierzchołkowi, a moment zakończenia - drugiemu.



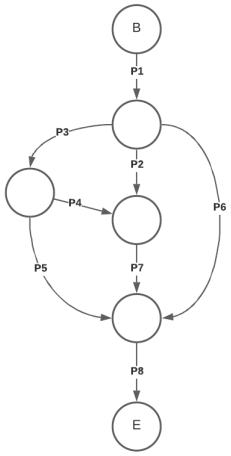
Rysunek 2.1: Graf procesów wykonywanych szeregowo (B - Begin, E - End).



Rysunek 2.2: Graf procesów wykonywanych równolegle (B - Begin, E - End).

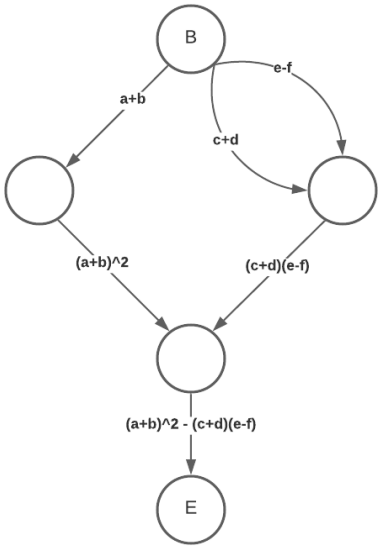


Rysunek 2.3: Graf procesów wykonywanych szeregowo-równolegle (B - Begin, E - End).



Rysunek 2.4: Graf procesów mieszanych (B - Begin, E - End).

Graf przepływu procesów jest **dobrze zagnieżdżony**, jeżeli może być opisany przez funkcje $P(a, b)$ i $S(a, b)$ lub ich złożenie, gdzie $P(a, b)$ i $S(a, b)$ oznaczają odpowiednio wykonanie równoległe i szeregowe procesów a i b.

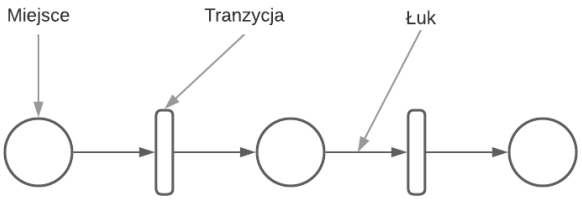


Rysunek 2.5: Graf dobrze zagnieżdżony dla wyrażenia $y := (a+b)^2 + (c+d)(e-f)$ (B - Begin, E - End).

2.2 Sieci Petriego

Elementarna sieć Petriego (*EPN*, sieć Petriego, Petri Net, Place/Transition net, elementary Petri net) to uporządkowana trójka *EPN = (P,T,A)*, gdzie:

1. P jest niepustym zbiorem miejsc (places),
2. T jest niepustym zbiorem przejść (tranzycji, transitions), takim, że $P \cap T = 0$,
3. $A \subseteq (P \times T, T \times P)$ jest zbiorem łuków sieci (arcs).



Rysunek 2.6: Przykład sieci.

2.3 Znakowana sieć Petriego

Znakowana sieć Petriego (*MPN*, marked Petri net) to uporządkowana piątka

$$MPN = (P, T, A, M_0)$$

gdzie:

1. *(P,T,A)* jest elementarną siecią Petriego,
2. $M : P \to N$ jest funkcją określoną na zbiorze miejsc zwaną **znakowaniem**, a M_0 jest **znakowaniem początkowym** sieci *MPN*.

Znakowaniem sieci *MPN=(P,T,A,M)* nazywamy dowolną funkcję $M : P \to N$.

Przejście $t \in T$ jest **aktywne** przy znakowaniu M (jest M-aktywne), jeżeli każde z jego miejsc wejściowych zawiera co najmniej jeden znacznik. Nowe M' otrzymane w wyniku wykonania przejścia t ze znakowaniem M jest wyznaczane za pomocą:

$$M'(p) = \begin{cases} M(p) - 1, & gdy\ p \in In(t) - Out(t) \\ M(p) + 1, & gdy\ p \in Out(t) - In(t) \\ M(p) & w\ pozostaych\ przypadkach \end{cases}$$

Gdzie *In(t)* jest zbiorem miejsc wejściowych tranzycji t, a *Out(t)* jest zbiorem miejsc wyjściowych tranzycji t.

2.4 Wielozbiory

Wielozbiór jest modyfikacją koncepcji zbioru, w której elementy zbioru mogą występować wielokrotnie.

Formalnie, przez **wielozbiór** nad zbiorem S rozumiemy parę *(S,ms)*, gdzie S jest niepustym zbiorem, a ms jest dowolną funkcją $ms : S \to N$.

Przez wielozbiór nad niepustym zbiorem S rozumie się też czasami wprost funkcję ms, oznaczoną S^*, $ms : S \to N$.

Niech S będzie dowolnym zbiorem. Symbolem 2^{S*} będziemy oznaczać rodzinę wszystkich wielozbiorów zbudowanych nad zbiorem S.

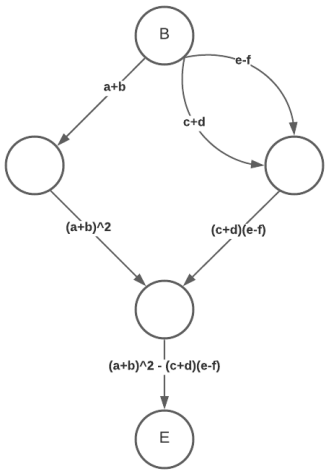
2.5 Notacje

2.5.1 Notacja "and" (Wirth)

Współbieżnie wykonanie może być specyfikowane za pomocą operatora AND, który łączy dwa wyrażenia wykonywane współbieżnie: a AND b.

Przykład:

```
begin
    begin
        x1:=a+b;
        x2:=x1*x1;
    end
   and
    begin
        x3:=c+d;
       and
        x4:=e-f;
    x5:=x3*x4;
    end;
   x6:=x2-x5;
end.
```



Rysunek 2.7: Graf dla którego zaimplementowano notację AND.

2.5.2 Notacja "parbegin - parend" ("cobegin - coend", Dijsktra)

Wszystkie wyrażenia ujęte w nawiasy PARBEGIN - PAREND są wykonywane współbieżnie.

Przykład notacji PARBEGIN - PAREND dla wyrażenia $y := (a+b)^2 + (c+d)(e-f)$:

```
begin
    parbegin
        begin
            x1:=a+b;
            x2:=x1*x1;
        end;
        begin
            parbegin
                x3:=c+d;
                x4:=e-f;
            parend;
            x5:=x3*x4;
        end;
    parend;
    x6:=x2-x5;
end.
```

2.5.3 Notacja "fork, join, quit" (Conway)

Znaczenie instrukcji:

- Instrukcja QUIT powoduje zakończenie procesu.
- Instrukcja FORK ω oznacza, że proces w którym wystąpiła ta instrukcja będzie dalej wykonywany współbieżnie z procesem identyfikowanym przez etykietę ω.
- Instrukcja JOIN $t,\ \omega$ ma dwa argumenty, zktórych t jest licznikiem, a ω jest etykietą.

  Wykonanie instrukcji JOIN, $t,\ \omega$ oznacza:

  ```
  t:=t-1;
  if t=0 then go to w;
  ```

  przy czym sekwencja tych dwóch instrukcji jest wykonywana atomowo, tzn. że jest niepodzielna (instrukcja jest wykonana w całości albo w cale).

Przykład notacji FORK, JOIN, QUIT:

```
begin
    t1:=2;
    t2:=2;
    fork w1;
    fork w2;
    fork w3;
    quit;
w1:
    x1:=a+b;
    x2:=x1*x1;
    join t1, w5;
    quit;
w2:
    x3:=c+d;
    join t2, w4;
    quit;
w3:
    x4:=e-f;
    join t2, w4;
    quit;
w4:
    x5:=x3*x4;
    join t1, w5;
    quit;
w5:
    x6:=x2-x5;
    quit;
```

3 Kolorowane sieci Petriego

3.1 Znakowana sieć Petriego z wagami

Znakowana sieć Petriego z wagami *(MWPN*, marked and weighted Petri net) to uporządkowana piątka

$$MWPN = (P, T, A, W, M_0)$$

gdzie:

1. (P, T, A, M_0) jest znakowaną siecią Petriego.
2. $W : A \to N$ jest funkcją określoną na zabiorze łuków przypisujących łukom wagi (weights).

3.2 Etykietowana sieć Petriego

Etykietowaną siecią Petriego *(LPN,* labeled Petri net) nazywamy uporządkowaną piątkę

$$LPN = (P, T, A, \sigma, M_0)$$

gdzie:

1. (P, T, A, M_0) jest znakowaną siecią Petriego.
2. $\sigma : T \to E$ jest funkcją etykietowaną, gdzie E to skończony zbiór wszystkich ciągów znaków.

3.3 Kolorowana sieć Petriego

Kolorwana sieć Petriego(CPN, Colored Petri Net) nazywamy uporządkowaną strukturę

$$CPN = (\Sigma, P, T, A, \gamma, C, G, E, M_0)$$

gdzie:

1. Σ : Niepusty skończony zbiór typów(kolorów), z których każdy jest niepusty.
2. P: Niepusty skończony zbiór miejsc
3. T: Niepusty skończony zbiór przejść
4. A: Niepusty skończony zbiór łuków, gdzie $P \bigcap T = P \bigcap A = T \bigcap A = \emptyset$ (Brak łuków, które idą do jakiegokolwiek $t \in T$ i jednocześnie $p \in P$
5. $\gamma : A \to (P \times T) \cup (T \times P)$: Funkcja zaczepienia przypisująca każdemu łukowi uporządkowaną parę węzłów.
6. $C : P \to \Sigma$: Funkcja typów(kolorów) określająca jakiego typu znaczniki każde z miejsc może zawierać.
7. G: Funkcja zastrzeżeń(dozorów, guards) przypisuje każdemu z przejść wyrażenie takie, że $\forall t \in T : \mathcal{T}(G(t)) \subseteq Bool \wedge \mathcal{T}(\mathcal{V}(G(t))) \subseteq \Sigma$. Co znaczy, że wyrażenie, które może zawierać zmienne typów należących do Σ i którego dowolne wartościowanie daje w wyniku wartość logiczną.
8. E: Funkcja wag łuków, przypisująca każdemu łukowi takie wyrażenie, że: $\forall a \in A : \mathcal{T}(E(a)) \subseteq 2^{C(P(a))*} \wedge \mathcal{T}(\mathcal{V}(E(a))) \subseteq \Sigma$. Co znaczy, że wyrażenie, które może zawierać zmienne typów należących do Σ i którego dowolne wartościowanie daje w wyniku wielozbiór nad typem przypisanym do miejsca $P(a)$.
9. M_0: Jest znakowaniem początkowym, takim że dla dowolnego miejsca $p \in P, M_0(p) \in 2^{C(p)*}$. Co znaczy, że M_0 to funkcja, która przyporządkowuje każdemu miejscu wielozbiór nad typem przypisanym do tego miejsca.
10. $\mathcal{V}(t)$: Zbiór zmiennych, które występują w zastrzeżeniu przejścia t lub w wyrażeniach przypisanych łukom ze zbioru $A(t)$
11. $A(t)$: Zbiór łuków otaczających węzeł t
12. : Wiązaniem(binding) przejścia $t \in T$ nazywany odwzorowanie b określone na zbiorze $\mathcal{V}(t)$, takie że każdej zmiennej przypisano wartość należącą do typu tej zmiennej i jednocześnie jest spełnione zastrzeżenie przejścia t. (Wynikiem wartościowania zastrzeżenia jest wartość *true*)
13. $\mathcal{B}(t)$: Zbiór wszystkich wiązań oznaczanych parą $(t, b), t \in T, b \in \mathcal{B}(t)$

3.3.1 Znakowanie sieci CPN

Znakowanie sieci CPN nazywamy dowolną funkcję M określoną na zbiorze miejsc sieci, taką że:

$$\forall p \in P : M(p) \in 2^{C(p)*}$$

1. $x, y \in V$: węzły sieci
2. E(x,y),(x:początek,y:koniec): wyrażenie powstałe przez zsumowanie wyrażeń na łukach
3. $E(x,y)_b$: wynik wartościowania wyrażenia $E(x,y)$ przy ustalonym wiązaniu b
4. $E(p,t)_b$: Wielozbiór znaczników, które są usuwane z miejsca p, jeżeli przejście t jest wykonane przy wiązaniu b
5. $E(t,p)_b$: Wielozbiór znaczników, które są dodawane do miejsca p, jeżeli przejście t jest wykonane przy wiązaniu b.
6. Przejście $t \in T$ jest aktywne przy znakowaniu M i wiązaniu $b \in \mathcal{B}(t)$(Jest M-aktywne przy wiązaniu b) jeśli jest spełniony warunek
   $$\forall p \in In(t) : E(p,t)_b \leqslant M(p)$$
   co oznacza: Jeśli każde miejsce wejściowe przejścia t zawiera przy wiązaniu b odpowiednią liczbę i odpowiednie wartości znaczników.
7. Przejście t jest aktywne przy znakowaniu M, jeżeli jest aktywne przy znakowaniu M i pewnym wiązaniu $b \in \mathcal{B}(t)$
8. Jeżeli przejście t jest aktywne przy znakowaniu M i wiązaniu b to nowe znakowanie M', które uzyskano w wyniku wykonania przejścia t, jest określone w sposób następujący:
   $$M'(p) \begin{cases} M(p) - E(p,t)_b & p \in In(t) - Out(t) \\ M(p) + E(t,p)_b & p \in Out(t) - In(t) \\ M(p) - E(p,t)_b + E(t,P) & p \in In(t) \cap Out(t) \\ M(p) & p \in else \end{cases}$$
   Jeżeli przejście t jest aktywne przy znakowaniu M i wiązaniu b, oraz w w wyniku jego wykonania otrzymujemy znakowanie M', to fakt ten zapisujemy jako:
   $$M \xrightarrow{(t,b)} M'$$

4 Synchronizacja

4.1 Sformułowanie formalne problemu wzajemnego wykluczania

4.1.1 Definicja

Dany jest zbiór procesów sekwencyjnych komunikujących się przez wspólną pamięć. Każdy z procesów zawiera sekcję krytyczną, w której następuje dostęp do wspólnej pamięci. Procesy te są cyklicznymi, o dodatkowych założeniach

1. Zapis i odczyt wspólnych danych jest operacją niepodzielną, a próba jednoczesnych zapisów lub odczytów realizowana jest sekwencyjnie w nieznanej kolejności.
2. Sekcje krytyczne nie mają priorytetu.
3. Względne prędkości wykonywania procesów są nieznane.
4. Proces może zostać zawieszony poza sekcją krytyczną.
5. Warunek bezpieczeństwa (Słaby warunek postępu): Procesy realizujące instrukcje poza sekcją krytyczną nie mogą uniemożliwiać innym procesom wejścia do sekcji krytycznej.
6. Warunek postępu: Procesy powinny uzyskać dostęp do sekcji krytycznej w skończonym czasie.

Przy tych założeniach należy zagwarantować, że w każdej chwili czasu co najwyżej jeden proces jest w swej sekcji krytycznej.

4.2 Rozwiązanie programowe problemu wzajemnego wykluczania

4.2.1 Wersja 1: (Nie działający)

```
program VersionOne:
var processNumber: INTEGER;

procedure ProcessOne;
begin
  while True do
    begin
      // Pusty While czekający na dostęp
      while processNumber=2 do;

      criticalSectionOne;
      processNumber:=2;
      // Other
    end;
end;
```

```
procedure ProcessTwo;
begin
  while True do;
    begin
      // Pusty While czekający na dostęp
      while processNumber=1 do;

      criticalSectionTwo;
      processNumber:=1;
      // Other
    end;
end;
begin
  processNumber:=1;
  parbegin
    ProcessOne;
    ProcessTwo;
  parend;
end.
```

Wyjaśnienie

Proces rozpoczyna współbieżnie 2 pod-procesy, które mają 2 różne sekcje krytyczne. Zmienna processNumber zezwala i pełni rolę wskaźnika, który proces ma prawo wejścia do sekcji krytycznej. Po wykonaniu jednej sekcji krytycznej, jest nadawany dostęp do drugiej, ale zamykany do pierwszej sekcji.

Te rozwiązanie spełnia warunek bezpieczeństwa na podstawie tego, że processNumber(wskaźnik) może przybierać wartość 1 albo 2. Nie jest za to spełniony warunek postępu, ponieważ, gdy zostanie zatrzymany jeden z procesów to drugi będzie bezowocnie oczekiwał na ponowne prawo do wejścia do sekcji krytycznej przez co nigdy się nie zakończy.

4.2.2 Wersja 2: (Nie działający)

```
program VersionTwo:
var P1inside, P2inside: BOOLEAN;

procedure ProcessOne;
begin
  while True do
    begin
      // Pusty While czekający na dostęp
      while P2inside do;

      P1inside:=True;
      criticalSectionOne;
      P1inside:=False;
      // Other
    end;
end;
```

```
procedure ProcessTwo;
begin
  while True do;
    begin
      // Pusty While czekający na dostęp
      while P1inside do;

      P2inside:=True;
      criticalSectionTwo;
      P2inside:=False;
      // Other
    end;
end;
begin
  P1inside:=False;
  P2inside:=False;
  parbegin
    ProcessOne;
    ProcessTwo;
  parend;
end.
```

Wyjaśnienie

Proces rozpoczyna współbieżnie 2 pod-procesy, które mają 2 różne sekcje krytyczne. Zmienne P1/2inside zaczynające z False decydują o tym, który proces ma prawo wejścia do sekcji krytycznej i pełnią rolę wskaźników. Przed wejściem do sekcji krytycznej zostaje zapisana informacja o tym, że proces się odbywa przez co inny proces wie o tym i sam nie wejdzie do własnej sekcji krytycznej.

Te rozwiązanie nie spełnia warunku bezpieczeństwa na podstawie tego, że P1/2inside na start mają False więc obie na raz mogą zacząc pracę, nie spełniając właśnie warunku bezpieczeństwa. Jest spełniony warunek postępu, ponieważ, gdy zostanie zatrzymany jeden z procesów to drugi będzie już zawsze miał dostęp do zasobów i sam się zakończy.

4.2.3 Wersja 3: (Nie działający)

```
program VersionThree:
var P1WantsToEnter: BOOLEAN;
var P2WantsToEnter: BOOLEAN;

procedure ProcessOne;
begin
  while True do
    begin
      // Pusty While czekający na dostęp
      P1WantsToEnter:=True;
      while P2WantsToEnter do;
      criticalSectionOne;
      P1WantsToEnter:=False;
      // Other
    end;
end;
```

```
procedure ProcessTwo;
begin
  while True do;
    begin
      // Pusty While czekający na dostęp
      P2WantsToEnter:=True;
      while P1WantsToEnter do;

      criticalSectionTwo;
      P2WantsToEnter:=False;
      // Other
    end;
end;
begin
  P1WantsToEnter:=False;
  P1WantsToEnter:=False;
  parbegin
    ProcessOne;
    ProcessTwo;
  parend;
end.
```

Wyjaśnienie

Proces rozpoczyna współbieżnie 2 pod-procesy, które mają 2 różne sekcje krytyczne. Zmienne P1/2WantsToEnter zaczynające z False decydują o tym, który proces ma chęć wejścia do sekcji krytycznej i pełnią rolę wskaźników. Przed wejściem do pętli czekającej na dostęp nadawana jest chęć wejścia do sekcji krytycznej.

Te rozwiązanie spełnia warunku bezpieczeństwa na podstawie tego, że na start mają False więc obie nie mogą na raz zacząć pracy i wejść do obu sekcji krytycznych w tym samym czasie. Nie jest spełniony warunek postępu, ponieważ, gdy obie zmienne P1/P2WantsToEnter przyjmą wartość True w tym samym czasie to oba procesy będą się pętliły do końca ich życia nigdy nie kończąc swoich sekcji krytycznych przez co procesy nigdy się nie zakończą i dlatego nie jest właśnie spełniony warunek postępu.

4.2.4 Wersja 4: (Nie działający, ale używalny)

```
program VersionFour:
var P1WantsToEnter: BOOLEAN;
var P2WantsToEnter: BOOLEAN;

procedure ProcessOne;
begin
  while True do
    begin
      // pętla While realizująca wskazanie
      P1WantsToEnter:=True;
      while P2WantsToEnter do
        begin
          P1WantsToEnter:=False;
          delay(random,freecycles);
          P1WantsToEnter:=True;
        end;

      criticalSectionOne;
      P1WantsToEnter:=False;
      // Other
    end;
end;
```

```
procedure ProcessTwo;
begin
  while True do;
    begin
      // pętla While realizująca wskazanie
      P2WantsToEnter:=True;
      while P1WantsToEnter do
        begin
          P2WantsToEnter:=False;
          delay(random,freecycles);
          P2WantsToEnter:=True;
        end;

      criticalSectionTwo;
      P2WantsToEnter:=False;
      // Other
    end;
end;
begin
  P1WantsToEnter:=False;
  P1WantsToEnter:=False;
  parbegin
    ProcessOne;
    ProcessTwo;
  parend;
end.
```

Wyjaśnienie

Proces rozpoczyna współbieżnie 2 pod-procesy, które mają 2 różne sekcje krytyczne. Zmienne P1/2WantsToEnter zaczynające z False decydują o tym, który proces ma chęć wejścia do sekcji krytycznej i pełnią rolę wskaźników. Przed wejściem do pętli czekającej na dostęp nadawana jest chęć wejścia do sekcji krytycznej, które jest rozbudowane o wycofanie się z decyzji o chęci wejścia do sekcji na podstawie funkcji random. W odróżnieniu od poprzedniego rozwiązania wystąpienie konfliktu wartości P1/2WantsToEnter rezultuje w wycofaniu się od konkurencji i pozwolenie jednemu z procesów na wejście do swojej sekcji krytycznej.

Te rozwiązanie spełnia warunek bezpieczeństwa na podstawie sytuacji wystąpienia konfliktu, te losują między sobą, która dostanie dostęp do sekcji krytycznej. Nie jest spełniony warunek postępu, ponieważ, gdy obie zmienne P1/P2WantsToEnter popadną w konflikt to istnieje szansa, że przy losowaniu kto odejdzie od rywalizacji, obie zmienne odejdą wymuszając ponowne losowanie, które może odbywać się w nieskończoność, nie gwarantując deterministycznej skończoności programu.

Mimo to, ta niedoskonałość w praktyce jest ignorowalna i można ją używać.

4.3 Algorytm Dekkera

Algorytm Dekkera - algorytm ten pozwala dwóm wątkom na bezkonfliktową pracę na danych pochodzących z jednego źródła przy użyciu do komunikacji między nimi jedynie pamięci dzielonej.

4.3.1 Implementacja algorytmu Dekkera w języku Pascal

```
program DekkerAlgorithm;
var favoredProcess; enum (First, Second);
var P1WantsToEnter, P2WantToEnter: BOOLEAN;
```

```
  procedure ProcessOne;
  begin
    while True do
      begin
        P1WantsToEnter:=True;
        while P2WantsToEnter do
          if favoredProcess=Second then
            begin
              P1WantsToEnter:=False
              while favoredProcess=Second do;
              P1WantsToEnter:=True;
            end;
        criticalSectionOne;
        favoredProcess:=Second;
        P1WantsToEnter:=False;
        otherStuffOne;
      end;
  end;
```

```
procedure ProcessTwo;
begin
  while True do
    begin
      P2WantsToEnter:=True;
      while P1WantsToEnter do
        if favoredProcess=First then
          begin
            P2WantsToEnter:=False;
            while favoredProcess=First do;
            P2WantsToEnter:=True;
          end;
      criticalSectionTwo;
      favoredProcess:=First;
      P2WantsToEnter:=False;
      otherStuffTwo;
    end;
end;
```

```
begin
  P1WantsToEnter:=False;
  P2WantsToEnter:=False
  favoredProcess:=First;
  parbegin
    ProcessOne;
    ProcessTwo;
  parend;
end.
```

Wyjaśnienie

Proces główny inicjalizuje trzy zmienne: P1/2WantsToEnter, którym przypisuje wartość False oraz favoredProcess z wartością First. Obydwa procesy posiadają podobną strukturę. Opiszemy wyłącznie działanie procesu pierwszego, ponieważ proces drugi działa analogicznie względem pierwszego.

Procedura ProcessOne rozpoczyna nieskończoną pętlę while, w której opisany jest algorytm przydziału sekcji krytycznej. Na początku zmienna P1WantsToEnter przyjmuje wartość True, co oznacza, że proces chce uzyskać dostęp do sekcji krytycznej. Następnie pętla while, rozpocznie iteracje, gdy warunek P2WantsToEnter jest prawdziwy, gdyby tak się nie stało, proces pierwszy uzyskuje dostęp do sekcji krytycznej. Po wykonaniu swojej pracy w sekcji krytycznej nadaje pierwszeństwo procesu drugiego i traci chęć dostępu do sekcji krytycznej.

Rozważmy teraz wariant konfliktu procesów. Występuje on, gdy oba procesy chcą uzyskać dostęp do sekcji krytycznej (P1/2WantsToEnter:=True). Instrukcja warunkowa sprawdza, który z procesów ma wyższy priorytet i na tej podstawie zmienia chęć dostępu do sekcji krytycznej.

Warunek bezpieczeństwa jest spełniony na podstawie tego, że gdy występuje konflikt P1/2WantsToEnter to jest on rozwiązany poprzez istniejący priorytet. Warunek postępu jest spełniony na podstawie tego, że nigdy nie dojdzie do scenariusza, że oba procesy są w sekcji krytycznej, ponieważ, gdy dochodzi do konfliktu, jest on rozwiązywany.

Rysunek 4.1: Pomóż mi to zrozumieć.

4.4 Algorytm Dijkstry

4.4.1 Implementacja algorytmu Dijkstry w języku Pascal

```
program DijkstraAlgorithm;
begin
  shared
    flag[1..n]: 0..2;
    turn      : 1..n;
  local
    test  : 0..2;
    k, other, temp : 1..n;
  while True do
    begin
      L: flag[i]:=1;
      other:=turn;
      // Sekcja próby
      while other!=i do
        begin
          test:=flag[other];
          if test=0 then
            turn:=i;
          other:=turn;
        end;
        flag[i]:=2;
        for k:=1 to n do
          if k!=i then
            begin
              test:=flag[k];
              if test=2 then
                goto L;
            end;
        criticalSection;
        flag[i]:=0;
        // Other;
    end;
end.
```

Wyjaśnienie

Na początek inicjalizowane są dwie zmienne wspólne, flag oraz turn i cztery zmienne lokalne (dla każdego procesu): test, k, other oraz temp. W tablicy flag możemy dostać się do zmiennych reprezentujących chęć dostania się do sekcji krytycznej(0 - brak chęci, 1 - pierwszy etap, 2 - drugi etap). Zmienna turn, to indeks wątku, który obecnie znajduje się w sekcji krytycznej.

Przetwarzanie algorytmu Dijkstry rozpoczynamy od przypisania obecnemu procesowi flagi 1, a następnie zmiennej other przypiszemy indeks procesu uprzywilejowanego. W sekcji próby sprawdzamy, czy indeks obecnego procesu jest różny od indeksu procesu z pierwszeństwem. Jeżeli indeks naszego procesu pokrywa się z indeksem procesu uprzywilejowanego, to oflagujemy nasz proces cyfrą 2 i przejdziemy do następnego etapu. W razie niepowodzenia wchodzimy w pętlę while, gdzie do zmiennej test przypisujemy flagę procesu uprzywilejowanego. Pozostajemy w pętli do momentu, aż proces okupujący sekcję krytyczną z niej wyjdzie, po czym przypiszemy sobie pierwszeństwo i wyjdziemy z pętli.

W drugim etapie pozyskiwania dostępu do sekcji krytycznej iterujemy po indeksach procesów, jeżeli nie proces nie jest procesem uprzywilejowanym, to jest on odsyłany do etykiety L. Proces, który przejdzie przez etap drugi rozpoczyna okupowanie sekcji krytycznej. Po jej opuszczeniu zmienia flagę na 0 i wraca do sekcji początkowej.

Warunek bezpieczeństwa jest spełniony co wynika z dwóch etapów sekcji próby, gdzie pierwszy sprawdza czy może przypisać sobie pierwszeństwo, a w drugiej, wybierany jest tylko jeden z tych procesów. Warunek postępu nie jest spełniony, ponieważ można nie wyjść z sekcji próby w deterministycznym czasie, procesy mogą nie w nieskończoność utknąć na sekcji próby. W praktyce jest to ignorowalne, ponieważ wyjdzie w z sekcji próby z prawdopodobieństwem dążącym do 1

4.5 Algorytm Petersona

Algorytm Petersona – algorytm przetwarzania współbieżnego, zapewniający wzajemne wykluczenie, umożliwiające dwóm procesom lub wątkom bezkonfliktowy dostęp do współdzielonego zasobu (sekcji krytycznej).

Algorytm Petersona zapewnia wzajemne wykluczanie tylko dla dwóch procesów, istnieje jednak uogólniona postać algorytmu do zastosowania z wieloma procesami.

4.5.1 Implementacja algorytmu Petersona dla dwóch procesów w języku Pascal

```
program PetersonAlgorithm
begin
  shared
    flag[0..1]: BOOLEAN;
    turn: INTEGER;
  local
    other: BOOLEAN;
    whose: INTEGER;
    while True do
      begin
        flag[i]:=True;
        turn:=1-i;
        repeat
          whose:=turn;
          other:=flag[1-i];
        until (whose=i or not other);

        criticalSection;
        flag[i]:=False;
        // Other;
      end;
end.
```

Wyjaśnienie

Na początek są inicjowane 2 zmienne wspólne, flag oraz turn, i po 2 zmienne lokalne dla procesów, other oraz whose, gdzie flag symbolizuje PWantsToEnter z poprzednich rozwiązań, a turn oznacza to samo co Favoured. W sekcji próby proces wskazuje chęć wykonania i mówi, który proces ma być faworyzowany, następnie odczytuje wspólną zmienną turn(priorytet) do whose oraz chęć wejścia do sekcji krytycznej do other do momentu spełnienia warunku wyjściowego, którym jest suma logiczna wyrażenia whose(priorytet)=i or not other(czyjaś inna tura), po wyjściu z sekcji próby jest realizowane wejście do sekcji krytycznej oraz po wyjściu z tej sekcji fladze odpowiadającej procesowi jest nadawany brak chęci wejścia do sekcji krytycznej.

Spełniony jest warunek bezpieczeństwa, ponieważ przy wejściu do sekcji próby rozwiązywany jest problem konfliktu poprzez nadanie priorytetów dla poszczególnych procesów, co powoduje, że oba procesy nie mogą w tym samym czasie wejść do sekcji krytycznej. Również jest spełniony warunek postępu, ponieważ sekcja próby jest realizowana w czasie deterministycznym, co gwarantuje dojście procesu do swojej sekcji krytycznej.

4.5.2 Implementacja algorytmu Petersona dla n procesów w języku Pascal

```
program PetersonAlgorithm_N
begin
  shared
    flag[1..n]: INTEGER;
    turn[1..n-1]: INTEGER;
  local
    k, l, other, whose : INTEGER;
  while True do
    begin
      for k:=1 to n-1 do
```

```
        begin
          flag[i]:=k;
          turn[k]:=i;
          repeat
            whose:=trun[k]; other:=0;
            if whose!=i then break;
              for l:=1 to n do
                begin
                  if l!=i then
                    other:=flag[l];
                  if other>=k then break;
                end;
          until other <k;
        end;
      criticalSection;
      flag[i]:=0;
      // Other;
    end;
end.
```

Wyjaśnienie

Proces rozpoczyna się inicjalizacją dwóch zmiennych wspólnych. Zmienna tablicowej flag, reprezentuje etap, na którym i-ty proces jest jeżeli chodzi o ubieganie się do sekcji krytycznej. 0 oznacza, że proces nie ubiega się o dostęp do sekcji krytycznej. Druga zmienna tablicowa turn, składa się z n-1 elementów, ponieważ jest n-1 etapów ubiegania się o dostęp do sekcji krytycznej, określa proces, na każdym i-tym miejscu przechowuję informację, który proces jest uprzywilejowany na danym etapie. Przechodzi do pętli while, gdzie znajduje się sekcja próby, w której pętla for przechodzi przez wszystkie etapy ubiegania się o dostęp do sekcji krytycznej, gdzie jego ciało mówi o tym, że jeśli proces i-ty zaczął trial to mówi o tym innym i nadaje sobie turn(pierwszeństwo), później powtarza wewnętrzna próba dostania się do kolejnego etapu i kolejnego. Wewnętrzna próba dostania się do kolejnego etapu polega na odczytaniu swojego priorytetu, który mógł być w nadpisany przez inne procesy, i jeżeli priorytet został nadpisany to wychodzi z repeat bezwarunkowo, w innym wypadku przechodzi przez wszystkie l-te procesy poza sobą, sprawdzając czy nadpisany priorytet jest niższy, jeśli nie, to wyłamuje się i powtarza znowu sprawdzenie, jeżeli za to pętla zakończy się, i każdy l-ty proces ma niższy priorytet, to proces przechodzi do kolejnego etapu próby. jeżeli przejdzie wszystkie etapy, może dojść do sekcji krytycznej.

Warunek bezpieczeństwa jest spełniony ponieważ tylko jeden proces może dojść do sekcji krytycznej w tej samym czasie, co wynika z faktu, że każdy nie jest możliwe przejście przez pętle repeat przez 2 procesy na raz. Warunek postępu jest spełniony, ponieważ w deterministycznym czasie zakończy się działanie każdego procesu, a to wynika z tego, że żaden proces nie jest w stanie wyprzedzić innego procesu, gdy ten jest wyższej w hierarchii, co powoduje, że na pewno ten dojdzie do swojej sekcji krytycznej i ustawi własną flagę na 0, pozwalając innym też dojść do swoich.

4.6 Algorytm Lamporta

Algorytm Lamporta zwany inaczej **algorytmem piekarnianym** – algorytm *Leslie Lamporta* rozwiązujący wykluczanie się w sekcji krytycznej dla dowolnej N liczby procesów. Algorytm działa na podobnej zasadzie jak automaty do wydawania numerków w bankach i urzędach. Proces o najwyższym indeksie wykona swoją sekcję krytyczną najpóźniej.

4.6.1 Implementacja algorytmu Lamporta dla n procesów w języku Pascal

```
program LamportAlgorithm;
begin
  shared
    choosing[1..n]: 0..1;
    num[1..n]: INTEGER;
  local
    test: 0..1;
    k, mine: INTEGER
```

```
    other, temp: INTEGER;
  while True do
    begin
      choosing[i]:=1;
      mine:=0;
      for k:=1 to n do
        if k!=i then
          begin
            temp:=num[k];
            mine:=max(mine,temp);
          end;
    mine:=mine+1;
    num[i]:=mine;
    choosing[i]:=0;

    for k:=1 to n do
      if k!=i then
        begin
          repeat
            test:=choosing[k]
          until test=0;
          repeat
            other:=num[k];
          until other=0 or (mine,i)<(other,k);
        end;
      //??
      criticalSection;
      num[i]:=0;
      // Other;
    end;
end;
```

Problem Piekarni

Mamy wiele osób ubiegających się o wyłączny dostęp do osoby sprzedającej chleb, problem jest taki, że jeżeli będzie sytuacja wiele osób jednocześnie może wejść do kolejki to powstaje problem jak uporządkować te osoby, a żeby w efekcie końcowym tylko jedna osoba z tych ubiegających się o dostęp uzyskała dostęp. Podchodzi osoba, dostaje numerek, i problem taki, że musimy zagwarantować wyłączny dostęp do rejestracji i podążanie tą drogą doprowadzałoby nas do sytuacji, gdzie byłaby rejestracja do rejestracji i tak w nieskończoność

Rozwiązanie problemu piekarni

Każdy wchodząc pyta się w sposób uczciwy kto jaki ma numer i na tej podstawie stwierdza jaki największy numer został dotychczas wybrany i przypisuje ten numer_max + 1, i dalej te numerki będą później podstawą kierunkowania dostępu, problem jest, gdy wchodzi wiele osób może się zdarzyć, że jeżeli te osoby pytałyby o inne numerki to nadałby wszystkie sobie numer_max + 1, W takim przypadku o kolejności decyduje dla tej grupy osób będzie decydował kolejny element, to jest identyfikator. Osoba o najmniejszym numerze może kupić chleb w pierwszej kolejności, a wychodząc mówi, że nie chce już chleba.

Wyjaśnienie

Na początku inicjalizujemy dwie zmienne wspólne choosing oraz num, następnie dla procesów inicjalizujemy zmienne lokalne: test, k, mine, other oraz temp. Zmienna choosing przyjmuje wartość 1, w momencie, w którym wybieramy zmienną num dla naszego procesu, wynosi ona 0, gdy mamy już wybrany numer.

Aby proces mógł zacząć ubiegać się o sekcję krytyczną, musi on wybrać swój numer(analogicznie do kolejki u lekarz - wybiera numer o jeden większy od ostatniego). Po tym jak zasygnalizuje wybieranie numeru, przypisuje on zmiennej mine wartość zero (numer początkowy), po czym iteruje po numerach pozostałych wątków i aktualizuje zmienną mine wybierając wartość maksymalną numeru. Po przeiterowaniu całej tablicy num[], następuje inkrementacja zmiennej mine i zasygnalizowanie choosing:=0.

Następnie przechodzimy do sekcji próby, gdzie sprawdzamy, czy każdy proces ma wybrany swój numer, jeżeli tak to poprzez sumę logiczną: żaden proces nie ubiegałby się o sekcję krytyczną lub numer mojego procesu jest mniejszy od wszystkich innych (w przypadku, gdyby numery były takie same porównujemy identyfikatory procesów i zwracamy prawdę, jeżeli nasze id jest mniejsze). Po wyjściu z pętli for przechodzimy do sekcji krytycznej, po czym ustawimy nasz numer na 0 i wrócimy do sekcji pierwszej.

Warunek bezpieczeństwa jest spełniony na podstawie tego, że nadajemy procesom indywidualne numery, na podstawie których dwa procesy nie będą mogły wejść do sekcji krytycznej w tym samym momencie, ponieważ wejść może tylko ten o najmniejszym numerze, a taki jest tylko i wyłącznie jeden. Warunek postępu jest spełniony na podstawie tego, że nigdy nie dojdzie do scenariusza, że oba procesy znajdą się w sekcji krytycznej, ponieważ, gdy dochodzi do konfliktu priorytetu, jest on rozwiązywany na podstawie wyboru id, a id z definicji są różne wzajemnie.

W tej wersji algorytmu istnieje problem, że może być sytuacja, że numery będą rosły w nieskończoność, w przypadku, gdy kolejka procesów oczekujących, które nadały sobie numer nigdy nie będzie pusta, przez co te numery będą ciągłe rosły, przez co może dojść do przepełnienia co oznaczało by numer 0, który spowoduje, że dwa procesy przejdą do sekcji krytycznej.

4.7 Instrukcja testandset

Załóżmy, że w systemie dostępna jest instrukcja typu *testandset(a,b)*, która w sposób atomowy (niepodzielny) dokonuje odczytu zmiennej **b**, zapamiętania wartości tej zmiennej w zmiennej **a** oraz przypisania zmiennej b wartości True.

testandset(a,b) jest równa

```
a:=b;
b:=True;
```

nottestandset(a,b) jest równa

```
a:=b;
b:=False;
```

4.7.1 Przykład testandset w języku Pascal

```
program TestAndSet_Example;
var active: BOOLEAN;

procedure ProcessOne;
var oneCannotEnter: BOOLEAN;
begin
  while True do
    begin
      oneCannotEnter:=True;
      while oneCannotEnter do
        testandset(oneCannotEnter, active);

      criticalSectionOne;
      active:=False;
      // Other;
    end;
end;
```

```
procedure ProcessTwo;
var twoCannotEnter: BOOLEAN;
begin
  while True do
    begin
      twoCannotEnter:=True;
      while twoCannotEnter do
        testandset(twoCannotEnter, active);

      criticalSectionTwo;
      active:=False;
      // Other;
    end;
endl;

begin
  active:=False;
  parbegin
    ProcessOne;
    ProvessTwo;
  parend;
end.
```

Wyjaśnienie

Atomowo jest aktualizowana zmienna Active, na podstawie, której zawsze jeden proces może uzyskać dostęp do sekcji krytycznej.

Spełniony jest silny warunek bezpieczeństwa, ponieważ zawsze tylko jeden proces będzie w sekcji krytycznej.

Nie jest spełniony warunek postępu, ponieważ może dojść do sytuacji, gdy będą się ciągle pojawiały nowe procesy, to jeden z procesów będzie ciągle przegrywał loterię dostępu, przez co może dojść do sytuacji, że nigdy nie dojdzie do swojej sekcji krytycznej nie spełniając właśnie warunku postępu.

4.8 Semafory

Semaforem nazywamy zmienną chronioną, na ogól będącą nieujemną zmienną typu INTEGER, do której dostęp (zapis i odczyt) możliwy jest tylko poprzez wywołanie specjalnych funkcji (operacji) dostępu i inicjacji.

Wyróżnia się semafory:

- **Binarne** - przyjmują tylko wartość 0 lub 1.
- **Ogólne** (licznikowe) - mogą przyjąć nieujemną wartość całkowitoliczbową.

4.8.1 Operacja P i V (Dijkstra)

Oznaczenie:

- **P** - pochodzi od holenderskiego *proben* (testuj) (ang. wait) (pot. podnieś).
- **V** - pochodzi od holenderskiego *verhogen* (inkrementuj) (ang. signal) (pot. upuść).

Operacja **P(S)** na semaforze **S** działa w sposób następujący:

```
if S > 0 then S:= S-1
else (wait on S)
```

Operacja **V(S)** na semaforze **S** działa następujaco:

```
if (oneOrMoreProcessesAreWaitingOnS) then (letOneOfTheseProcessesProceed)
else S:= S+1;
```

4.8.2 Przykład

```
program SemaphoreExample;
var active: SEMAPHORE;

procedure ProcessOne;
begin
  while True do
    begin
      P(active);
      criticalSectionOne;
      V(active);
      otherStuffOne;
    end;
end;

begin
  semaphore_initialize(active,1);
  parbegin
    ProcessOne;
    ...
    ProcessNth;
  parend;
end.
```

Wyjaśnienie

Spełnia warunek bezpieczeństwa z definicji semaforu, czyli dokonywania operacji na semaforze atomowo.

Nie spełnia warunku postępu, ponieważ mogą znaleźć się procesy nie będące wstanie dostrzec zmiany zmiennej active na False.

4.9 Problem producenta-konsumenta

```
program ProducentConsumer;
var emptyBuffers, fullBuffers, active: SEMAPHORE;

            procedure Producer;
            begin
                while True do
                    begin
                        produceNextRecord;
                        P(emptyBuffers);
                        P(active);
                        addToBuffer;
                        V(active);
                        V(fullBuffers);
                    end
            end;

procedure Consumer;
begin
    while True do
        begin
            P(fullBuffers);
            P(active);
            takeFromBuffer;
            V(active);
            V(emptyBuffers);
            processNextRecord;
        end
end;

begin
    semaphore_initialize(active, 1);
    semaphore_initialize(emptyBuffers, N);
    semaphore_initialize(fullBuffers, 0);
    parbegin
        Producer;
        Consumer;
    parend;
end.
```

Wyjaśnienie

Spełnia warunek bezpieczeństwa. Spełnia słaby warunek postępu, ale nie spełnia silnego warunku postępu, ponieważ mogą znaleźć się procesy nie będące wstanie dostrzec zmiany zmiennej active na False.

4.10 Semafory Binarne

Semafory binarne Sb mogą przyjmować tylko dwie wartości 0 i 1.
Przez **Pb** i **Vb** oznaczone są operacje na semaforach binarnych odpowiadające operacją **P** i **V**.
Definicja **Pb** jest taka sama jak **P**, natomiast definicja **Vb** różni się od **V** tylko tym, że **Vb** nie zmienia wartości semafora binarnego jeśli miał on wartość 1.

4.10.1 Operacje Pb(Sb) i Vb(Sb)

Operacja **Pb(Sb)** działa w sposób następujący:

```
repeat
    nottestandset(pActive, Sb) // Sb:=False
until (pActive)
```

Operacja **Vb(Sb)** działa następujaco:

```
Sb:=True;
```

4.10.2 Specyfikacja implementacji z aktywnym czekaniem (Busy wait)

```
procedure P_S;
begin
    while S =< 0 do;
    S:=S-1;
end;

procedure V_S;
begin
    S:=S+1;
end;
```

Wyjaśnienie

Ta implementacja nie działa ze względu na brak gwarancji atomowości operacji. Nie jest bezpieczna.

4.11 Implementacja z aktywnym czekaniem (Niepoprawna)

```
program PV_implementation;
var active, delay : BOOLEAN;
var NS : INTEGER

procedure PIMPLEMENTATION;
var pActive, pDelay: BOOLEAN;
begin
  pActive := True;
  while pActive do
    testandset(pActive, active);
  // Początek Sekcji Krytycznej
  NS:=NS-1;
  if NS >= 0 then
    begin
      S:=S-1;
      active:=False;
    end;
  else
    begin
      active := False;
      pDelay := True;
      while pDelay do
        testandset(pDelay, delay)
    end;
end;

procedure VIMPLEMENTATION;
var vActive : BOOLEAN;
begin
  vActive := True;
  while vActive do
    testandset(vActive, active);
  NS:=NS+1;
  if NS > 0 then
    begin
      S:=S+1;
    end;
  else
    active:=False;
  delay:=False;
end;

begin
  active:=False;
  delay:=True;
end.
```

Wyjaśnienie

Wprowadzono dodatkową zmienną NS, która jest INTEGER i może przyjmować wartości ujemne (w przeciwieństwie do zmeinnej semaforowej S), która odwzorowuje liczbę dostępnych zasobów. Gdy Wartość ta jest ujemna, to jej bezwzględna wartość symbolizuję liczbę zawieszonych procesów, natomiast gdy dodatnia, to obrazuje wartość zmiennej semaforowej S.

Operacja P zaczyna się od ustawienia zmiennej pActive na True i następnie wykonanie pętli testandset(pActive, active) wykonane do uzyskania sygnału False. Gdy dojdzie do takiego sygnału to wiemy, że reszta programu jest wykonywana w trybie wzajemnego wykluczania. Następnie wechodząc do sekcji krytycznej w trybie bezpiecznym

dekrementujemy wartość NS po czym sprawdzamy, czy liczba NS jest nieujemne (co jest równe sprawdzenia dosępności zasobów), gdy jest dostępny zasób to zmniejszamy liczbę dostępnych zasobów na zmiennej semaforowej S i sygnalizujemy koniec procesu poprzez przypisanie zmiennej współdzielonej active wartości False, co mówi innym procesom o dostępności sekcji krytycznej. W przeciwnym wypadku, zwaniamy dostęp dostęp do sekcji krytycznej active False i jednocześnie aktywujemy oczekiwanie na nowy zasób poprzez oczekiwanie na sygnał delay od innego procesu.

Operacja V zaczyna się od ustawienia oczekiwania na zasygnalizowanie wartości False dla zmiennej vActive w pętli, co powoduje wzajemne wykluczenie procesu i pozwala na bezpieczne przejście do sekcji krytycznej procedury. W sekcji krytycznej jest dokonywana inkremenetacja zmiennej NS i jeżeli NS jest dodatnie to również jest inkremenetowana wartość zmiennej semaforowej S, a w przeciwnym wypadku (gdy wartość NS jest niedodatnia) ustawiony jest delay na False. na koniec jest sygnalizowane zakończenie aktywności.

Główną wadą tej implementacji jest aktywne czekanie, proces zawieszany, jest zawieszany przez pętle (busy-waiting) niepotrzebnie konsumując czas procesora.

Przykład błędnego działania

Sytuacja gdy zmienna semaforowa S ma wartość 0 i chcą wejść 3 kolejne procesy P, NS zostanie 3 razy zdekrementowane, i za każdym razem zostaną wrzucone na oczekiwanie sygnału False od delay, ale przed tym Bardzo Wolno wykonują linię 19 tak wolno, że są wykonywane 3 operacje V, wchodzą do sekcji krytycznej sekwencyjnie inkrementują NS przez co wysyłają delay = False i zwalniają sekcje krytyczne, i nareszcie pierwszej operacji P udaje się dokonać pDelay:=True; i wchodzi do pętli, widzi, że delay jest false, więc wychodzi z pętli i synchronicznie ustawia delay na True, Ale teraz drugiemu procesowi P udaje się wejść do pętli oczekiwania na sygnał delay False, i okazuje się, że jest True pomimo wykonania 3 operacji V, co rezulutuje w 2 procesach oczekujących na zamknięcie. Co oczywiście jest błędne

4.12 Implementacja z aktywnym czekaniem (Poprawna)

```
program PV_implementation;
var active, delay : BOOLEAN;
var NS : INTEGER

procedure PIMPLEMENTATION;
var pActive, pDelay: BOOLEAN;
begin
  pActive := True;
  while pActive do
    testandset(pActive, active);
  // Początek Sekcji Krytycznej
  NS:=NS-1;
  if NS >= 0 then
      S:=S-1;
  else
    begin
      active:=False;
      pDelay:=True;
      while pDelay do
        testandset(pDelay, delay)
    end;
  active:=False;
end;

procedure VIMPLEMENTATION;
var vActive : BOOLEAN;
begin
  vActive:=True;
  while vActive do
    testandset(vActive, active);
```

```
  NS:=NS+1;
  if NS > 0 then
    begin
      S:=S+1;
      active:=False;
    end;
  else
    active:=False;
end;

begin
  active:=False;
  delay:=True;
end.
```

Wyjaśnienie

Ponownie wprowadzoną dodatkową zmienną NS.

Operacja P ponownie zaczyna się od ustawienia zmiennej pActive na True i następnie wykonanie pętli testandset(pActive, active) wykonane do uzyskania sygnału False. Następnie wchodząc do sekcji krytycznej w trybie bezpiecznym dekrementujemy wartość NS po czym sprawdzamy, czy liczba NS jest nieujemne, gdy jest dostępny zasób to zmniejszamy liczbę dostępnych zasobów na zmiennej semaforowej S i sygnalizujemy, ale Nie sygnalizujemy końca procesu. W przeciwnym wypadku, jak w pierwszym wariancie zwalniamy dostęp dostęp do sekcji krytycznej active False i jednocześnie aktywujemy oczekiwanie na nowy zasób poprzez oczekiwanie na sygnał delay od innego procesu, a na końcu przypisana jest wartość False dla active.

Operacja V zaczyna się od ustawienia oczekiwania na zasygnalizowanie wartości False dla zmiennej vActive w pętli. W sekcji krytycznej jest dokonywana ponownie inkremenetacja zmiennej NS i jeżeli NS jest dodatnie to również jest inkremenetowana wartość zmiennej semaforowej S i wysyłana informacja o zakończeniu aktywności sekcji krytycznej, a w przeciwnym wypadku ustawiony jest delay na False. na koniec jest sygnalizowane zakończenie aktywności.

Ponowną wadą tej implementacji jest aktywne czekanie.

Rozróżnienie

W odróżnieniu do poprzedniej, Błędnej, implementacji teraz, gdy dojdzie do sytuacji, że pDelay jest bardzo powolne, to w sposób implementacji operacji V pozwala wstrzymać kolejną operację V przed wykonaniem do momentu, gdy sygnał delay od V nie zostanie uwzględniony przez operację P, która kończąc swoje czekanie na sygnał ponownie sygnalizuje o dostępności sekcji krytycznej. (Ale to Cholernie Mądre)

4.13 Implementacja operacji P i V (z wspomaganiem operacji systemowych)

```
program PV_IMPLEMENTATION;
var active, delay : BOOLEAN;
var NS: INTEGER;

procedure PIMPLEMENTATION;
var pActive : BOOLEAN;
begin
  Disable interrupts;
  pActive := True;
  while pActive do
    testandset(pActive, active);
  NS := NS - 1;
  if NS >= 0 then
    begin
      S:=S-1;
      active := False;
      Enable interrupts;
```

```
      end;
    else
      begin
        Block process invoking P(S);
        p := Remove from RL;
        active := False;
        Transfer control to p with Enable interrupts;
      end;
end;

procedure VIMPLEMENTATION;
var vActive : BOOLEAN;
begin
  Disable interrupts;
  vActive := True;
  while vActive do
    testandset(vActive, active);
  NS := NS + 1;
  if NS > 0 then
    S:=S+1;
  else
    begin
      p:=remove from LS;
      add p to RL;
    end;
  active := False;
  Enable interrupts;
end;

// Niebezpieczne! (użytkownik zwykle nie ma dostepu do systemu przerwań) +
// nieskuteczne dla systemów wieloprocesorowych.
```

Legenda:

LS - List associated with **S**.

RL - Ready List.

4.14 Implementacja operacji wait i signal

```
type Semaphore = record
  value: INTEGER;
  L: list of processes;
end;
```

```
procedure WAIT(S);
begin
  S.value := S.value - 1;
  if S.value < 0 then
    begin
      add this process ID to S.L;
      block this process;
    end;
end;
```

```
procedure SIGNAL(S);
begin
  S.value := S.value + 1;
    if S.value =< 0 then
      begin
        remove a process P from S.L;
        wakeup(P);
      end;
end;
```

Wyjaśnienie

Przy tych deklaracjach musimy założyć, że całe operacje wait i signal są atomowe.

4.15 Inne operacje semaforowe

```
lock w:
  L : if w = 1 then go to L
  else w := 1;

unlock w:
  w := 0;

ENQ(r):
  if inuse[r] then                //resource r is used
    begin
      insert p on r-queue;        //queue associated with r-queue
      Block p;
    end
  else
    inuse[r] := True;

DEQ(r):
  p := Remove from r-queue
  if p <> Omega                   // <> == !=
    then Activate p               // p = Omega means that queue was emptyBuffers
  else
    inuse[r] := False;

WAIT(e):                          // oczekiwanie procesu na zajście zdarzenia
if not posted[e] then             // założenie : only one process can wait for event e
  begin
    wait[e] := True;
    process[e] := p;
    Block p;
  end
else
  posted[e] := False;

POST(e):
if not posted[e] then
  begin
  posted[e] := True;
  if wait[e] then
    begin
      wait[e] := False;
      posted[e] := False;
      Activate process[e];
   end;
end;

Block(i):
if not wws[i]                     // wait for Wakeup flag associated with process i
  then Block process i
  else wws[i] := False;

Wakeup(i):
if ready(i)                       // process is ready
  then wws[i] := True
  else Activate process i;
```

Jest to poprawne, jeśli zagwarantujmy, że wszystkie operacje są atomowe.

4.16 Liczniki zdarzeń (Event counters)

Definiujemy trzy operacje na zmiennej event counter E:

- **read(E)** - zwraca obecną wartość zmiennej E.
- **advance(E)** - inkrementacja zmiennej E o 1.
- **await(E,v)** - czeka do momentu, aż E będzie miało wartość v lub większą.

4.17 Producent-Konsument z użyciem liczników zdarzeń

```
#include "prototypes.h"
#define N 100                       //number of slots in the buffer
typedef INT EVENT_COUNTER;          //even coutners are a special kind of int
EVENT_COUNTER in = 0;               //counts items inserted into buffer
EVENT_COUNTER out = 0;              //counts items removed from buffer

void PRODUCER(void) {
  INT item, sequence = 0;
  while(True) {                     //infinite loop
    produce_item(&item);            //generate something to put in buffer
    sequence = sequence + 1;        //counts items produced so far
    await(out, sequence - N);       //wait until there is room in buffer
    enter_item(item);               //put item in slot(sequence - 1) % N
    advance(&in);                   //let consumer know about another item
  }
}

void CONSUMER(void) {
  INT item, sequence = 0;
  while(True){                      //infinite loop
    sequence = sequence + 1;        //number of item to remove from buffer
    await(in, sequence);            //wait until required item is present
    remove_item(&item);                    //take item from slot(sequence - 1) & N
    advance(&out);                  //let producer know that item is gone
    consume_item(item);             //do something with the item
  }
}
```

Budowa

Bufor zbudowany jest z dwóch zmiennych EVENT_COUNTER (są to zmienne współdzielone) oraz zmiennej N równej 100 oznaczającej wielkość bufora.

Producent oraz konsument posiadają zmienne lokalne typu INT o nazwach item oraz sequence.

Wyjaśnienie

W tej implementacji jest założona atomowość operacji await i advance.

Zacznij od omówienia działania PRODUCENTA. W nieskończonej pętli while produkujemy przedmiot, następie zmienna *sequence* ulega inkrementacji. Następnym działaniem producenta będzie sprawdzenie, czy jest miejsce w buforze, jeśli jest to dodamy przedmiot do bufora, natomiast jeżeli nie ma miejsca to proces zostanie wstrzymany.

Analogicznie działa KONSUMENT. Jeżeli w buforze nie będzie przedmiotu do pobrania, to proces zostanie wstrzymany.

Nie jest spełniony warunek postępu, ponieważ może dojść do sytuacji, że w wyniku atomowości operacji await i advance, jeden z procesów będzie ciągle przegrywał loterię dostępu, przez co może dojść do sytuacji, że nigdy nie wejdzie do swojej sekcji krytycznej nie spełniając właśnie warunku postępu.

4.18 Regiony krytyczne

4.18.1 Definicja

Niech następująca deklaracja zmiennej v typu T określa zmienną dzieloną przez wiele procesów.

```
var v: shared T;
```

Zmienna v będzie dostępna tylko w obrębie instrukcji REGION o następującej postaci:

```
region v: do S;
```

4.18.2 Implementacja

Dla każdej deklaracji

```
var v: shared T;
```

Kompilator generuje semafor *v-mutex* z wartością początkową 1.
Dla każdej instrukcji

```
region v: do S;
```

Kompilator generuje następujący kod:

```
wait(v-mutex);
S;
signal(v-mutex);
```

4.19 Warunkowy region krytyczny

4.19.1 Definicja

Następująca instrukcja jest instrukcją warunkowego regionu krytycznego

```
region v when B do S ;
```

w której B jest wyrażeniem boolowskim. Jak poprzednio, regiony odwołujące się do tych samych zmiennych dzielonych wykluczają się wzajemnie w czasie. Obecnie jednak, kiedy proces wchodzi do regionu sekcji krytycznej, wtedy następuje obliczenie wyrażenia boolowskiego B. Jeśli wyrażenie jest prawdziwe, to instrukcja S będzie wykonana. Jeśli jest fałszywe, to proces nie ubiega się o wyłączny dostęp i ulega opóźnianiu do czasu, aż wyrażenie B stanie się prawdziwe oraz żaden inny proces nie będzie przebywał w regionie związanym ze zmienną v.

4.19.2 Producent konsument - warunkowy region krytyczny

```
var buffer: shared record
  pool: array[0..n-1] of ITEM;
  count, in, out: INTEGER;
end;
```

Proces produkujący umieszcza nową jednostkę NEXTP w buforze dzielonym wykonując instrukcję:

```
region buffer when count < n
  do begin
    pool[in] := nextp;
    in := (in + 1) mod n;
    count := count + 1;
  end;
```

Proces konsumujący usuwa jednostkę z bufora dzielonego i zapamiętuje ją w NEXTK za pomocą instrukcji:

```
region buffer when count > 0
  do begin
    nextk := pool[out]
    out := (out + 1) mod n;
    count := count         - 1;
  end;
```

4.19.3 Warunkowy region krytyczny - implementacja

```
region v when B do S;

var xMutex, xDelay : SEMAPHORE;    //xCount - the number of processes
  xCount, xTemp : INTEGER;         //waiting for xDelay

wait(xMutex);                      //xTemp - the number of processes that have been
if not B then                      //allowed to test their Boolean condition during
  begin                            //one trace
    xCount := xCount + 1;
    signal(xMutex);
    wait(xDelay);
    while not B do
      begin
        xTemp := xTemp + 1;
        if xTemp < xCount then
          signal(xDelay)
          else
            signal(xMutex);
          wait(xDelay);
      end;
    xCount := xCount + 1;
  end;
S;
if xCount > 0 then
  begin
    xTemp := 0;
    signal(xDelay);
  end;
else
  signal(xMutex);
```

Budowa

Region jest zbudowany wokół zmiennej współdzielonej v, przy warunku Booleańskim na ciele S (funkcji)
W tej implementacji jest założona atomowość operacji wait i signal.

Wyjaśnienie

Warunkowe regiony polegają na tym, że na początku są inicjowane dwóch zmienne semaforowe xMutex, xDelay i xCount, który mówi o liczbie procesów oczekujących, a xTemp to liczba procesów, które mogą testować ich warunki booleańskie podczas jednego przejścia.

Następnie oczekujemy operacją Wait oczekująca na zasób xMutex, co zapewniającą wzajemne wykluczenie. Gdy wchodzimy do sekcji krytycznej sprawdzamy warunek B, jeśli jest spełniony zwalniamy to wykonujemy operację S i jeżeli liczba procesów oczekujących na xDelay jest dodatnia to zerujemy xTemp i sygnalizujemy wyjście z semafora xDelay.

W przeciwnym wypadku sygnalizujemy wyjście z xMutex. Natomiast gdy warunek B nie jest spełniony inkrementujemy liczbę procesów oczekujących i zwalniamy dostęp do semafora xMutex, ale sygnalizujemy potrzebę do semafora xDelay oczekując na zmianę warunku B na prawdziwy w pętli while, gdzie najpierw inkrementujemy liczbę procesów, które mogą testować warunki a następnie jeżeli liczba procesów oczekujących jest większa od liczby procesów mogących testować warunki to przesyłamy sygnał xDelay, w przeciwnym wypadku xMutex po czym oczekujemy na sygnał od xDelay.

Gdy warunek B zostanie w końcu spełniony odbywa się proces dekrementacji liczby procesów oczekujących, a potem jest wykonywana operacja S i jeżeli liczba procesów oczekujących na xDelay jest dodatnia to zerujemy xTemp i sygnalizujemy wyjście z semafora xDelay. w przeciwnym wypadku sygnalizujemy wyjście z xMutex.

Co oznacza, że dopóki warunek B jest niespełniony to procesy będą wrzucane na kolejkę oczekującą na spełnienie warunku, ale będą go sprawdzać dopiero, gdy jakichś kolejny proces przejdzie przez ciało S pozytywnie, co spowoduje,

że wszystkie oczekujące procesy ponownie sprawdzą warunek B i ewentualnie wyjdą z pętli sygnalizując to następnie innym.

4.20 Problem pisarzy i czytelników

4.20.1 Rozwiązanie problemu pisarzy i czytelników z użyciem semaforów

```
shared var
  nReaders : INTEGER;
  mutex, wmutex, srmutex : SEMAPHORE;

procedure READER;
begin
  P(mutex);
  if nReaders = 0 then
    begin
      nReaders := nReaders + 1;
      P(wmutex);
    end
  else
    nReaders := nReaders + 1;
  V(mutex);
  read(f);
  P(mutex);
  nReaders := nReaders - 1;
  if nReaders = 0 then
    V(wmutex);
  V(mutex);
end;

procedure WRITER;
begin
  P(srmutex);
  P(wmutex);
  write(f,d);
  V(wmutex);
  V(srmutex);
end;

begin
  mutex := wmutex := srmutex := 1;
  nReaders := 0;
end.
```

Uwagi

Rozwiązanie to preferuje czytelników, co oznacza NIEBEZPIECZEŃSTWO ZAGŁODZENIA PISARZY!

Wyjaśnienie

Na początku deklarujemy zmienne współdzielone, którymi są liczba czytelników (nReaders) oraz trzy semafory: mutex (semafor czytelnika), wmutex (semafor pisarza) i srmutex (semafor obecnie pracującego pisarza). Wszystkim semaforom przypisujemy wartość 1 oraz ustawiamy zmienną nReaders na 0.

Zaczniemy od opisu procesu czytelnika. Czytelnik zaczyna swoją pracę od sprawdzenia ilu jest obecnie aktywnych czytelników, jeśli nie ma żadnego czytelnika to zmienna nReaders ulega inkrementacji, a semafor wmutex jest blokowany przez nasz proces, co uniemożliwia pisarzom rozpoczęcia swojej pracy. Następnie zwalniamy semafor mutex i rozpoczynamy operację odczytu pliku f. Po skończeniu odczytu dokonujemy operacji P(mutex), zmniejszamy liczbę czytelników o 1 i jeśli jest to możliwe zwalniamy semafor pisarzy umożliwiając im rozpoczęcie swojej pracy.

Proces pisarza rozpoczyna się od próby uzyskania dostępu do semafora srmutex oraz wmutex. Jeśli obie dyrektywy zakończą się pomyślnie proces pisarza może zacząć swoją pracę w pliku f. Po wykonaniu pracy zwalniamy poprzednio zajęte semafory.

Rozwiązanie powyższego problemu nie spełnia silnego warunku postępu, z racji uprzywilejowania procesów czytelników. Mianowicie może dojść do sytuacji, gdzie procesy pisarzy będą blokowane przez pojawiające się nowe procesy czytelników.

4.20.2 Rozwiązanie problemu pisarzy i czytelników z użyciem regionów krytycznych

```
var v: shared record
  nReaders, nWriters: INTEGER;
  busy: BOOLEAN
end;

//Proces czytelnika
region v do
begin
  await(nWriters=0);
  nReaders := nReaders + 1;
end;
...
read file
...
region v do
begin
  nReaders := nReaders - 1;
end;

//Proces pisarza
region v do
begin
  nWriters = nWriters + 1;
  await((not busy) and (nReaders = 0);
  busy := True;
end;
...
write file
...
region v do
begin
  nWriters := nWriters - 1;
  busy := False;
end;
```

Wyjaśnienie

Ta implementacja jest oparta o regiony krytyczne, które zapewniają wzajemne wykluczenie, gdzie w pierwszym regionie zmienna v, która jest rekordem zawierającym liczbę czytelników nReaders i zmienną booleańską busy mówiącą o tym, czy jest aktualnie plik w użyciu przez pisarza.

Pierwszy region dla procesu czytelnika podnosi zmienną v i oczekuje w niej 0 nWriters(brak czytelników), jeżeli jest 0 to przechodzi dalej inkrementując liczbę czytelników, odczytuje plik i w kolejnym regionie wzajemnego wykluczenia dekrementuje liczbę czytelników.

Proces odpowiedzialny za pisarza w pierwszym regionie inkrementuje liczbę pisarzy i oczekuje momentu, gdy nie będzie plik w użyciu pisarza oraz liczba czytelników wyniesie 0, jeżeli do tego dojdzie mówi, że plik jest w użyciu poprzez zmienną busy = True, zapisuje do pliku i w drugim regionie pisarza, dekrementuje liczbę pisarzy oraz mówi, że plik nie jest już w użyciu przez pisarza przez busy = False

4.21 Monitor

Monitor to taka struktura, która charakteryzuje się przez predefiniowany zasób operacji, której stan jest definiowany przez jej wartość.

Uwaga: Monitor gwarantuje wzajemne wykluczanie

Przykładowe ciało monitora

```
type MONITOR_NAME = monitor
  variable declaration
  procedure entry P1 (...)
    begin /* Ciało */ end;
  ...
  procedure entry Pn (...)
    begin /* Ciało */ end;
  begin
    // Kod inicjalizujący
  end;
```

4.21.1 Operacje wait i signal

Uwaga: Semantyka tych wyrażeń jest inna niż operacji semaforowych

Programista, który chce zapisać przykrojony na miarę własnych potrzeb schemat synchronizacji, może zdefiniować jedną lub kilka zmiennych typu warunek:

```
var x, y: CONDITION;
```

Jedynymi operacjami, które mogą dotyczyć warunku, są operacje:

- **x.wait** - oznacza, że proces ją wywołujący zostaje zawieszony do czasu, aż inny proces wywoła operację x.signal.
- **x.signa**l - wznawia dokładnie jeden z zawieszonych procesów. Jeśli żaden proces nie jest zawieszony, to operacja ta nie ma żadnych skutków, tzn. stan zmiennej x jest taki, jak gdyby operacji tej nie wykonano wcale.

Implementacja operacji

```
Procedure x.wait;
xCount:=xCount+1;
if nextCount > 0
  then singal(next);
  else signal(mutex);
wait(xSem);
xCount:=xCount-1;
```

```
procedure x.signal;
if xCount > 0
  then
    begin
      nextCount:=nextCount+1;
      wait(xSem);
      wait(next);
      nextCount:=nextCount-1;
    end.
```

Oznaczenia

- mutex: Semafor gwarantujący wzajemne wykluczenie
- nextCount: Liczba procesów, które wywołały x.signal i zostały zawieszone
- next: Semafor umożliwiający zawieszenie procesu wywołującego x.signal
- xCount: Liczba procesów oczekujących na x.signal
- xSem: Semafor umożliwiający zawieszenie procesu wywołującego x.wait

4.21.2 Problem producenta-konsumenta rozwiązany z użyciem monitorów

```
type PRODUCER_CONSUMER = monitor
var full, empty : CONDITION;
count : INTEGER;

procedure entry ENTER;
begin
if count = N then full.wait;
  enter_item;
  count := count + 1;
  if count = 1 then empty.singal;   //tak naprawdę można to zrobić bez ifa
end;

procedure entry REMOVE;
begin
  if count = 0 then empty.wait;
    remove_item;
    count := count - 1;
    if count = N - 1 then full.signal;
end;

begin
  count := 0;
end monitor;

procedure PRODUCER;
begin
  while True do
    begin
      produce_item;
      PRODUCER_CONSUMER.ENTER;
    end
end;

procedure CONSUMER;
begin
  while True do
    begin
      PRODUCER_CONSUMER.REMOVE;
      consume_item;
    end;
end.
```

Wyjaśnienie

W powyższym algorytmie przedstawiono jedynie procedury opisujące działanie producenta i konsumenta, a nie przedstawiono całego działania kodu.

Procedura ENTER opisuje dodanie itemu do bufora. Zaczynamy od sprawdzenia, czy bufor jest pełny, jeżeli jest on pełny to wstrzymujemy producenta do momentu zwolnienia się miejsca w buforze. Natomiast jeżeli mamy w nim miejsce, to zwiększamy zmienną count o 1, a jeżeli przed dodaniem itemu do bufora był on pusty (tzn. że teraz count = 1), to wykonujemy operację empty.signal, która obudzi niedeterministycznie proces konsumenta który oczekiwał na wyprodukowanie przedmiotu.

Procedura REMOVE opisuje usunięcie przedmiotu z bufora. Zaczynamy od sprawdzenia, czy bufor jest pusty, jeżeli nie ma w nim (buforze) nic to wstrzymujemy pobieranie danych. Jeśli uda nam się wejść do instrukcji warunkowej następuje pobranie danych i zmniejszenie zmiennej **count** o 1. Na sam koniec sprawdzamy, czy po operacji bufor jest niepusty, jeśli tak to wysyłamy sygnał o możliwości dodania danych.

Warunek bezpieczeństwa jest spełniony, ponieważ nie zdarzy się sytuacja, w której dwa procesy będą w tym samym momencie w sekcji krytycznej.

4.21.3 Alokacja zasobów z wykorzystaniem monitora

```
type RESOURCE_ALLOCATION = monitor
  var busy: BOOLEAN;
      x: INTEGER;

  procedure Acquire(time: Integer);
    begin
      if busy then x.wait(time);
      busy:=True;
    end;

  procedure Release;
    begin
      busy:=False;
      x.singal;
    end;

  begin // Inicjalizacja
    busy:=False;
  end.
```

Wyjaśnienie

Monitor polegający na dwóch operacjach Acquire i Release z dwoma zmiennymi, jedna booleańska busy mówiąca o tym, czy aktualnie zasób jest pobierany, i x, opisująca dany oczekiwany zasób.

Acquire polega na tym, że wywołana komenda będzie umieszczała proces w kolejce skojarzonej ze zmienną warunkową x będzie umieszczała w miejscu takim, a żeby oczekujące procesy były uporządkowane według rosnących czasów, czyli pierwszy na początku będzie o najkrótszym czasie i potem dłuższy itd. Release informuje o tym, że zasób został pobrany, nie jest zajęty (busy=False) i sygnalizowane jest, że ponownie można skorzystać z zasobu.

4.21.4 Rozwiązanie problemu czytelników i pisarzy z wykorzystaniem monitorów

```
type READERS_WRITERS = monitor;
var readerCount : INTEGER;
busy : BOOLEAN;
OKtoRead, OKtoWrite : CONDITION;

procedure entry STARTREAD;
begin
  if busy then OKtoRead.wait;
    readerCount := readerCount + 1;
    OKtoRead.signal;                //Once one reader can start, they all can
end;

procedure entry ENDREAD;
begin
  readerCount := readerCount - 1;
  if readerCount = 0 then OKtoWrite.signal;
end;

procedure entry STARTWRITE;
begin
  if busy or readerCount != 0 then OKtoWrite.wait;
  busy := True;
```

```
end;

procedure entry ENDWRITE;
begin
  busy := False;
  if OKtoRead.queue then OKtoRead.signal;
  else OKtoWrite.signal;
end;

begin                                 // initialization
readerCount:=0;
busy :=False;
end;
```

Wyjaśnienie

Brak

4.21.5 Monitor - implementacja

```
wait(mutex);          //mutex - semafor gwarantujący wzajemne wykluczanie
...                   //nextCount - liczba procesów, które wywołały sygnał x.signal i zostały zawieszone
tresc procedury F;    //next - semafor umożliwiający zawieszenie procesu wywołującego x.signal
...                   //xCount - liczba procesów czekających na x.signal
if nextCount > 0 then signal(next)    //xSem - semafor umożliwiający zawieszenie
else signal(mutex);                   //procesu wywołującego x.wait


x.wait:
  xCount := xCount + 1;
  if nextCount > 0 then signal(next)
  else singal(mutex);
  wait(xSem);
  xCount := xCount - 1;

x.signal:
  if xCount > 0 then
    begin
      nextCount := nextCount + 1;
      singal(xSem);
      wait(next);
      nextCount := nextCount + 1;
    end.
```

4.22 Problem jedzących filozofów

4.22.1 Opis problemu

Pięciu filozofów siedzi sobie przy okrągłym stole, każdy z nich ma przed sobą talerz spaghetti. Żeby zjeść spaghetti filozof potrzebuje dwóch widelców, jednego w prawej ręce, jednego w lewej. Problem polega na tym że na stole leży tylko 5 widelców. Każdy z filozofów ma dwie fazy:

- **myśli** - filozof myśli sobie przez skończony okres czasu i zaczyna być głodny.
- **je** - filozof chce zjeść spaghetti, żeby to zrobić musi podnieść dwa widelce, ale nie może podnieść obu jednocześnie.

Kiedy wreszcie uda mu się podnieść dwa widelce, filozof je przez skończony okres czasu, odkłada widelce i zaczyna myśleć Zauważmy że nigdy dwóch siedzących obok siebie filozofów nie będzie jadło jednocześnie. W każdym rozwiązaniu tego problemu, podnoszenie widelca musi być sekcją krytyczną.

4.22.2 Rozwiązanie problemu jedzących filozofów z wykorzystaniem monitorów

```
type DINNING_PHILOSOPHERS = monitor
var state : array[0..4] of (Thinking, Hungry, Eating);
var self : array[0..4] of CONDITION;

procedure entry PICKUP(i: 0..4);
begin
  state[i] := Hungry;
  test (i);
  if state[i] != eating then self[i].wait;  //CZEKAMY AŻ ZWOLNIĄ SIĘ SZTUĆCE (*)
end;

procedure entry PUTDOWN(i: 0..4);
begin
  state[i] := Thinking;
  test(i + 4 mod 5);                //testujemy
  test(i + 1 mod 5);                //sąsiadów
end;

procedure entry TEST(k: 0..4);
begin
  if state[k + 4 mod 5] != Eating and state[k] = Hungry and state[k + 1 mod 5] != Eating
   then                             //sprawdzenie czy sąsiedzi jedzą
    begin
      state[k] := Eating;           //ten co czekał (*)
      self[k].signal;               //signalujemy temu co czekał
    end;
end;

begin
  for i := 0 to 4 do
    state[i] := Thinking;
end;
```

Uwaga!

Rozwiązanie problemu jedzących filozofów z wykorzystaniem monitorów (PROBLEM DEADLOCK'U : co jeżeli sąsiedzi ciągle będą jeść? biedny filozof będzie głodny).

Wyjaśnienie

Wszyscy filozofowie zaczynają od myślenia. Mają trzy możliwe opcje, TEST, polegające na sprawdzeniu czy sąsiedzi nie jedzą i czy filozof jest głodny jeśli to prawda, to filozof zaczyna jeść i sygnalizuje to czekającemu, PUTDOWN polega na odłożeniu sztućca przez i-tego i przez to dwóch kolejnych może spróbować jeść robiąc TEST, Puckup mówi, że filozof i-ty jest głodny i sprawdza czy może zjeść teraz, jeśli nie może to mówi, że czeka na jedzenie, co jest później sprawdzane, gdy ktoś inny zje, to zasyganlizuje w TEST, że zjadł i ten co chciał nie musi już czekać, a może np sprawdzić ponownie, czy może zjeść.

4.23 Łącza

Wymiana komunikatów (ang. message passing) realizowana jest z użyciem dwóch podstawowych operacji komunikacyjnych:

- send(P, m)
- recieve(Q, m)
- m: przesyłany komunikat (ang. message)

- P: odbiorca komunikatu(P jak odbiorca)
- Q: nadawca komunikatu(Q jak nadawca)

Wariant symetryczny W komunikacji bezpośredniej każdy proces, który chce nadać lub odebrać komunikat musi jawnie nazwać odbiorcę lub nadawcę uczestniczącego w tej wymianie informacji. W tym wypadku operacje send i receive są zdefiniowane następująco:

- send(P, m): nadaj komunikat m do procesu P
- receive(Q, m): odbierz komunikat od procesu Q
- Cechy:
  - ustawiane są automatycznie między parą procesów, które mają komunikować się
  - dotyczą dokładnie dwóch procesów
  - są dwukierunkowe

Wariant asymetryczny, w którym nadawca nazywa odbiorcę, a od odbiorcy nie wymaga się znajomości nadawcy, w tym wypadku:

- send(P, m): nadaj komunikat m do procesu P
- receive(id, m): odbierz komunikat od dowolnego procesu; pod id zostanie podstawiona nazwa procesu, od którego nadszedł komunikat.

Łącze komunikacyjne jest elementem umożliwiającym transmisję informacji między interfejsami odległych węzłów. Wyróżnia się łącza jedno i dwukierunkowe. Wyposażone są one w bufory o określonej pojemności (ang. links capacity).

Jeżeli łącze nie posiada buforów (jego pojemność jest równa zero), to mówimy o łączu niebuforowanym, w przeciwnym razie – o buforowanym.

Zwykle kolejność odbierania komunikatów wysyłanych z danego węzła jest zgodna z kolejnością ich wysłania, wówczas łącze nazywamy łączem FIFO, w przeciwnym razie – nonFIFO.

Łącza mogą gwarantować również, w sposób niewidoczny dla użytkownika, że żadna wiadomość nie jest tracona, duplikowana lub zmieniana - są to tzw. łącza niezawodne (ang. reliable, lossless, duplicate free, error free, uncorrupted, no spurious).

4.23.1 Czas transmisji w łączu niezawodnym

Czas transmisji w łączu niezawodnym (ang. transmission delay, in-transit time) może być ograniczony lub jedynie określony jako skończony lecz nieprzewidywalny. W pierwszym przypadku mówimy o transmisji synchronicznej lub z czasem deterministycznie ograniczonym (w szczególności równym zero), a w drugim – o transmisji asynchronicznej lub z czasem niedeterministycznym.

4.23.2 Komunikacja pośrednia

W komunikacji pośredniej komunikaty są nadawane i odbierane poprzez skrzynki pocztowe (nazywane też portami, ang. mailbox)

Abstrakcyjna skrzynka pocztowa jest obiektem, w którym procesy mogą umieszczać komunikaty, i z którego komunikaty mogą być pobierane. Każda skrzynka pocztowa ma jednoznaczną identyfikację. Proces może komunikować się z innymi procesami za pomocą różnych skrzynek pocztowych. W tym wypadku operacje send i receive są zdefiniowane następująco:

- send(A, m) – nadaj komunikat m do skrzynki A
- receive(A, m)– odbierz komunikat ze skrzynki A
- Cechy:
  - ustawiane są między procesami tylko wówczas, gdy procesy te dzielą jakąś skrzynkę pocztową
  - mogą wiązać więcej niż dwa procesy
  - każda para procesów może mieć kilka różnych łączy
  - mogą być jednokierunkowe lub dwukierunkowe.

4.23.3 Skrzynka Pocztowa

Skrzynka może być własnością procesu lub systemu. Jeżeli skrzynka należy do procesu (tzn. jest przypisana lub zdefiniowana jako cześć procesu), to rozróżnia się jej właściciela (który za jej pośrednictwem może tylko odbierać komunikaty) i użytkownika (który może tylko nadawać komunikaty do danej skrzynki).

W wielu przypadkach, proces ma możliwość zadeklarowania zmiennej typu skrzynka_pocztowa. Proces deklarujący skrzynkę pocztową staje się jej właścicielem. Każdy inny proces, który zna nazwę tej skrzynki, może zostać jej użytkownikiem.

Skrzynka pocztowa należąca do systemu istnieje bez inicjatywy procesu i dlatego jest niezależna od jakiegokolwiek procesu. System operacyjny dostarcza mechanizmów pozwalających na:

- tworzenie nowej skrzynki
- nadawanie i odbieranie komunikatów za pośrednictwem skrzynki
- likwidowanie skrzynki

Proces, na którego zamówienie jest tworzona skrzynka, staje się domyślnie jej właścicielem. Przywilej własności jak i odbierania komunikatów może jednak zostać przekazany innym procesom za pomocą odpowiednich funkcji systemowych.

4.23.4 Synchroniczność i Asynchroniczność

Kanały o niezerowej pojemności umożliwiają realizację następujących operacji komunikacji:

- Nieblokowanych (asynchronicznych): proces nadający przekazuje komunikat do kanału (bufora) i natychmiast kontynuuje swe działanie, a proces odbierający odczytuje stan kanału wejściowego, lecz nawet gdy kanał jest pusty, proces kontynuuje działanie
- W komunikacji synchronicznej, nadawca i odbiorca są blokowani aż odpowiedni odbiorca odczyta przesłaną do niego wiadomość (ang. rendez-vous)
- Blokowanych (synchronicznych): nadawca jest wstrzymywany do momentu, gdy wiadomość zostanie odebrana przez adresata, natomiast odbiorca - do momentu, gdy oczekiwana wiadomość pojawi się w jego buforze wejściowym
- W przypadku komunikacji asynchronicznej, nadawca lub odbiorca komunikuje się w sposób nieblokowany.

4.23.5 Rozwiązanie problemu producenta-konsumenta przy użyciu komunikacji między procesami

```
program PRODUCERCONSUMER_MESSAGETRANSMISSION;
var bufferPool: array [0..x] of BUFFER;

procedure PRODUCER;
begin
  while True do
    begin
      produceNextMessage;
      receive(producer, empty);          //odbiór blokowany
      addMessageToCommonBuffer;
      send(consumer, empty);             //wysyłanie asynchroniczne
    end;
end;

procedure CONSUMER;
begin
  while True do
    begin
      receive(consumer, empty);
      takeMessageFromCommonBuffer;
```

```
      send(producer, empty);
      processMessage;
   end;
end;

// Inicjalizacja, służąca wysłaniu
// początkowych wiadomości producentom
begin
  I:=N;
  while I > 0 do
    begin
          send(producer, empty);
          I := I - 1;
        end;
  parbegin
    PRODUCER;
        CONSUMER;
  parend;
end.
```

Wyjaśnienie

Wymiana wiadomości jest tutaj wykorzystywana do synchronizacji, do wstrzymywania procesów jeżeli nie są spełnione warunki kontynuacji.

W programie mamy producenta, który coś w kółko produkuje starając się umieścić ten obiekt w buforze i konsumenta, który coś zżre w kółko. Informacje przesyłane są nieważne w tym wypadku ze względu na to, że tylko chcemy poinformować konsumenta lub producenta o zaistnieniu potrzeby wyprodukowania, zeżarcia czegoś

5 Zakleszczenie

5.1 Zakleszczenie

Rozważmy system składający się z n procesów (zadań) P1, P2, ... Pn współdzielących s zasobów nieprzywłaszczalnych, tzn. zasobów, których zwolnienie może nastąpić jedynie z inicjatywy zadania dysponującego zasobem. Każdy zasób składa się z m_k jednostek dla k = 1,2,...,s. Jednostki zasobów tego samego typu są równoważne. Każda jednostka w każdej chwili może być przydzielona tylko do jednego zadania, czyli dostęp do nich jest wyłączny.

5.2 Charakterystyka zadania P_j w każdej chwili

- wektor maksymalnych żądań (ang. claims)

$$C(P_j) = [C_1(P_j), C_2(P_j), ..., C_s(P_j)]^T$$

  oznaczający maksymalne żądanie zasobowe zadania P_j w dowolnej chwili czasu.

- wektor aktualnego przydziału (ang. current allocations)

$$A(P_j) = [A_1(P_j), A_2(P_j), ..., A_s(P_j)]^T$$

- wektor rang zdefiniowany jako różnica pomiędzy wektorami C i A

$$H(P_j) = C(P_j) - A(P_j)$$

5.3 Wektor wolnych zasobów f

Zakładamy, że jeżeli żądania zadania przydziału zasobów są spełnione w skończonym czasie, to zadanie to zakończy się w skończonym czasie i zwolni wszystkie przydzielone mu zasoby. Na podstawie liczby zasobów w systemie oraz wektorów aktualnego przydziału można wyznaczyć wektor zasobów wolnych f, gdzie:

$$f = [f_1, f_2, ..., f_s]^T$$

gdzie,

$$f_k = m_k - \sum_{j=1}^{n} = A_k(P_j) \qquad k = 1, 2, ..., s$$

(różnica liczby zasobów w systemie, a tymi które zostały przydzielone poszczególnym procesom)

5.4 Typy żądań

Wyróżniamy dwa typy żądań, które mogą być wygenerowane przez każde zadanie P_j

- żądanie przydziału dodatkowych zasobów (ang. request for resource allocation):

  $$\rho^a(P_j) = [\rho_1^a(P_j), \rho_2^a(P_j), ..., \rho_s^a(P_j)]^T$$

  gdzie, $\rho_k^a(P_j)$ jest liczbą jednostek zasobu R_k żądanych dodatkowo przez P_j.Żądane zasoby mogą nie być wolne, wtedy proces jest wstrzymywany do czasu spełnienia żądania)

- żądanie zwolnienia zasobów (ang. request for resource release):

  $$\rho^r(P_j) = [\rho_1^r(P_j), \rho_2^r(P_j), ..., \rho_s^r(P_j)]^T$$

  gdzie ρ_k^r jest liczbą jednostek zasobu R_k zwalnianych przez P_j.Zwolnienie zasobów zawsze się powiedzie.

5.5 Zadania przebywające w systemie

- Łatwo wykazać, że:
  - Żaden proces nie może żądać więcej niż wynosi jego ranga, czyli nie może żądać więcej niż zadeklarował, że maksymalnie zażąda po pomniejszeniu o liczbie zasobów które są przydzielone

    $$\underset{k}{\forall}\underset{j}{\forall} \rho_k^a(P_j) \leqslant H_k(P_j)$$

  - Każdy proces zwalnia co najwyżej tyle zasobów ile sam posiada

    $$\underset{k}{\forall}\underset{j}{\forall} \rho_k^r \leqslant A_k(P_j)$$

  - Oczywiście żądanie przydziału dodatkowego zasobu może być spełnione tylko wówczas gdy:

    $$\underset{k}{\forall} \rho_k^a(P_j) \leqslant f_k \qquad j = 1, 2, ..., s$$

- Przez zadanie przebywające w systemie rozumiemy zadanie, któremu przydzielono co najmniej jedną jednostkę zasobu. Stan systemu jest zdefiniowany przez stan przydziału, zasobu wszystkim zadaniom. Mówimy, że tan jest realizowalny, jeżeli spełniona jest następująca zależność:

  $$\underset{k}{\forall} \rho_k^a(P_j) \leqslant f_k$$

- Stan systemu nazywamy stanem bezpiecznym (ang. safe) ze względu na zakleszczenie, jeżeli istnieje sekwencja wykonywania zadań przebywających w systemie oznaczona $(P^1, P^2, ..., P^n)$ i nazywana sekwencją bezpieczną spełnia zależność:

  $$H_k(P_j) \leqslant f_k + \sum_{i=1}^{j-1} A_k(P^i) \qquad k = 1, 2, ..., s \qquad j = 1, 2, ..., n$$

  W przeciwnym razie, tzn. jeżeli sekwencja taka nie istnieje, stan jest nazywany stanem niebezpiecznym. Innymi słowy, stan jest bezpieczny jeżeli istnieje takie uporządkowanie wykonywania zadań, że wszystkie zadania przebywające w systemie zostaną zakończone. Powiemy że tranzycja stanu systemu wynikająca z alokacji zasobów jest bezpieczna, jeżeli stan końcowy jest stanem bezpiecznym.

"Czyli innymi słowy pierwszy proces w sekwencji może zgłosić co najwyżej żądanie, w którym żądania w odniesieniu do poszczególnych zasobów będą mniejsze lub równe liczbie zasobów wolnych aktualnie w systemie [...] dla drugiego procesu co to oznacza? Ranga drugiego procesu w sekwencji musi być mniejsza lub równa (czyli maksymalne dodatkowe żądanie drugiego procesu w sekwencji) liczby zasobów aktualnie wolnych w systemie powiększoną o liczbę zasobów które aktualnie przydzielone są pierwszemu procesowi, z kolei dla trzeciego procesu tldr: ranga =¡ liczba aktualnie wolnych zasobów w systemie + aktualnie przydzielone zasoby 1 i 2 procesu z sekwencji itd." J. Brzeziński

5.6 Zakleszczenie - definicja

- Przez zakleszczenie (ang. deadlock) rozumieć będziemy formalnie stan systemu, w którym spełniany jest następujący warunek:
$$\underset{\Omega\neq\emptyset}{\exists}\ \underset{j\in\Omega}{\forall}\ \underset{k}{\exists}\ \rho_k^a(P_j) > f_k + \sum_{i\notin\Omega} A_k(P_i)$$
gdzie, Ω jest zbiorem indeksów (lub zbiorem zadań).
- Mówimy że system jest w stanie zakleszczenia (w systemie wystąpił stan zakleszczenia), jeżeli istnieje niepusty zbiór Ω zadań, które żądają przydziału dodatkowych zasobów nieprzywłaszczalnych będących aktualnie w dyspozycji innych zadań tego zbioru.
- Innymi słowy system jest w stanie zakleszczenia, jeżeli istnieje niepusty zbiór Ω zadań, których żądania przydziału dodatkowych zasobów nieprzywłaszczalnych nie mogą być spełnione nawet jeśli wszystkie zadania nie należące do Ω zwolnią wszystkie zajmowane zasoby.
- Jeżeli $\Omega \neq \emptyset$ to zbiór ten nazywamy zbiorem zadań zakleszczonych.

5.7 Warunki konieczne wystąpienia zakleszczenia:

1. **Wzajemne wykluczanie (ang. mutual exlcusion condition):**
W każdej chwili zasób może być przydzielony co najwyżej jednemu zadaniu.
2. **Zachowywanie zasobu (ang. wait for condition):**
Proces oczekujący na przydzielenie dodatkowych zasobów nie zwalnia zasobów będących aktualnie w jego dyspozycji.
3. **Nieprzywłaszczalność (ang. non preemption condition):**
Zasoby są nieprzywłaszczalne tzn. ich zwolnienie może być zainicjowane jedynie przez proces dysponujący w danej chwili zasobem.
4. **Istnienie cyklu oczekiwania (ang. circular wait condiditon):** Występuje pewien cykl procesów z których każdy ubiega się o przydział dodatkowych zasobów będących w dyspozycji kolejnego procesu w cyklu.

5.8 Przeciwdziałanie zakleszczeniom:

- **Konstrukcja systemów wolnych od zakleszczeń (ang. construction of deadlock free systems)**
Podejście to polega w ogólności na wyposażeniu systemu w taką liczbę zasobów, aby wszystkie możliwe żądania zasobowe były możliwe do zrealizowania. Przykładowo, uzyskuje się to, gdy liczba zasobów każdego rodzaju jest nie mniejsza od sumy wszystkich maksymalnych i możliwych jednocześnie żądań.' "Bardzo kosztowne"
- **Detekcja zakleszczenia i odtwarzanie stanu wolnego od zakleszczenia (ang. detection and recover)**
W podejsciu detekcji i odtwarzania, stan systemu jest periodycznie sprawdzany i jeśli wykryty zostanie stan zakleszczenia, system podejmuje specjalne akcje w celu odtworzenia stanu wolnego od zakleszczenia. Często stosowany"
- **Unikanie zakleszczenia (ang. avoidance)**
W podejściu tym zakłada się znajomość maksymalnych żądań zasobowych. Każda potencjalna tranzycja stanu jest sprawdzana i jeśli jej wykonywanie prowadziłoby do stanu niebezpiecznego, to żądanie zasobowe nie jest w danej chwili realizowane "Bardzo ważne"

- **Zapobieganie zakleszczeniu (ang. prevention)**
  w ogólności podejście to polega na wyeliminowaniu możliwości zajścia jednego z warunków koniecznych zakleszczenia

5.9 Detekcja zakleszczenia - Algorytm Habermana $O(n^2)$

1. Zainicjuj D := 1,2,...,n i f;
2. Szukaj zadania o indeksie $j \in D$ takiego, że: $p^{\alpha}(P_j) \leqslant f$ (wektor żądań jest mniejszy od wektora zasobów).
3. Jeżeli zadanie takie nie istnieje, to zbiór odpowiadający zbiorowi D jest zbiorem zadań zakleszczeniowych. **Zakończ wykonywanie algorytmu**
4. W przeciwnym razie, podstaw:
$$D := D - j;$$
$$f := f + A(P_j)$$
5. Jeżeli $D = \emptyset$, to zakończ wykonywanie algorytmu. W przeciwnym razie przejdź do kroku 2.

5.10 Odtwarzanie stanu - Algorytm Holta $O(n \log n)$

Spośród zadań zakleszczonych wybierz zadanie(zadania), którego usunięcie spowoduje osiągnięcie stanu wolnego od zakleszczenia najmniejszym kosztem.

```
begin
  initialize: I_k = 1, k = 1,2,...,s;
              c_i = s, i = 1,2,...,n; c_0 = n;
  LS: Y:=False;
  for k = 1 step 1 until s do
    begin
      while E_{1,k,Ik} =< f_k and I_k =< n do
        begin
          c_{E2,k,I_k} := c_{E2,k,I_k} - 1;
          I_k := I_k - 1;
          if c_{E2,k,I_k} = 0 then
            begin
              c_0 := c_0 - 1;
              Y := True;
              for i = 1 step 1 until s do
                f_i := f_i + A_i(P_{E_2,k,I_k});
            end;
        end;
    end;
  if Y = True c_0 > 0 then go to LS;
  if Y = True then answer "no"
  else answer "yes"
end.
```

5.10.1 Wady i zalety podejścia detekcji do odtwarzania stanu:

- Wady:
  - Narzut wynikający z opóźnionego wykrycia stanu zakleszczenia
  - Narzut czasowy algorytmu detekcji i odtwarzania stanu
  - Utrata efektów dotychczasowego przetwarzania odrzuconego zadania
- Zalety:
  - Brak ograniczeń na współbieżność wykonywania zadań

- Wysoki stopień wykorzystywania zasobów
- Podejście unikania

5.11 Algorytm unikania:

1. Za każdym razem, gdy wystąpi żądanie przydziału dodatkowego zasobu, sprawdź bezpieczeństwo tranzycji stanu odpowiadającej realizacji tego żądania. Jeśli tranzycja ta jest bezpieczna, to przydziel żądany zasób i kontynuuj wykonywanie zadania. W przeciwnym razie zawieś wykonywanie zadania.

2. Za każdym razem, gdy wystąpi żądanie zwolnienia zasobu, zrealizuj to żądanie i przejrzyj zbiór zadań zawieszonych w celu znalezienia zadania, którego tranzycja z nowego stanu odpowiadałaby tranzycji bezpiecznej. Jeśli takie zadanie istnieje, zrealizuj jego żądanie przydziału zasobów.

5.11.1 Wady i zalety podejścia unikania:

- Wady:
  - Duży narzut czasowy wynikający z konieczności wykonywania algorytmu unikania przy każdym żądaniu przydziału dodatkowego zasobu i przy każdym żądaniu zwolnienia zasobu.
  - Mało realistyczne założenie o znajomości maksymalnych żądań zasobów
  - Założenie, że liczba zasobów w systemie nie może maleć.
- Zalety:
  - Potencjalnie wyższy stopień wykorzystania zasobów niż w podejściu zapobiegania.

5.12 Podejście zapobiegania

Rozwiązania wykluczające możliwość wystąpienia cyklu żądań.

5.12.1 Algorytm wstępnego przydziału

1. Przydziel w chwili początkowej wszystkie wymagane do realizacji zadania zasoby lub nie przydzielaj żadnego z nich

5.12.2 Algorytm przydziału zasobów uporządkowanych:

1. Uporządkuj jednoznacznie zbiór zasobów,

2. Narzuć zadaniom ograniczenie na żądania przydziału zasobów, polegające na możliwości żądania zasobów tylko zgodnie z uporządkowaniem zasobów.

5.13 Algorytm Wait-Die (rozwiązanie negujące zachowywanie zasobów (wait for condition):

1. Uporządkuj jednoznacznie zbiór zadań według etykiet czasowych.

2. Jeżeli zadanie P_1, będące w konflikcie z zadaniem P_2, jest starsze (ma mniejszą etykietę czasową), toP_1 czeka (wait) na zwolnienie zasobu przez P_2. W przeciwnym razie zadani P_1 jest w całości odrzucane (abort) i zwalnia wszystkie posiadane zasoby.

5.14 Algorytm Wound-Wait (rozwiązanie dopuszczające przywłaszczalność)

1. Uporządkuj jednoznacznie zbiór zadań według etykiet czasowych.

2. Jeżeli zadanie P_1, będące w konflikcie z zadaniem P_2, jest starsze (ma mniejszą etykietę czasową), to zadanie P_2 jest odrzucane (abort) i zwalnia wszystkie posiadane zasoby. W przeciwnym razie P_1 czeka (wait) na zwolnienie zasobu przez P_2.

5.14.1 Wady i zalety podejścia zapobiegania:

- Wady
  - Ograniczony stopień wykorzystania zasobów
- Zalety:
  - Prostota i mały narzut czasowy

6 Zadania

6.1 Zadanie 1: algorytm Habermanna

Sprawdź, czy w podanym stanie występuje zakleszczenie stosując algorytm Habermana.

Stan systemu:

$$A = \begin{matrix} P_1 & P_2 & P_3 & P_4 & P_5 \end{matrix} \begin{bmatrix} 1 & 2 & 1 & 0 & 2 \\ 2 & 3 & 1 & 0 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 2 & 3 & 1 & 1 & 1 \end{bmatrix} \qquad m = \begin{bmatrix} 8 \\ 8 \\ 8 \\ 11 \end{bmatrix} \qquad \rho^a = \begin{bmatrix} 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix}$$

A - Macierz obecnie pobranych zasobów.

m - Transponowany wektor zasobów (wszystkie zasoby w sumie, te zajęte i wolne).

ρ^a - Macierz zapotrzebowań procesów na określone zasoby.

Na podstawie macierzy A i wektora m obliczamy wektor f (Transponowany wektor wolnych zasobów), który jest różnicą danego wiersza m i sumy wiersza A.

$$f = \begin{bmatrix} (8-6) = 2 \\ (8-7) = 1 \\ (8-8) = 0 \\ (11-8) = 3 \end{bmatrix}$$

Wyjaśnienie

Na początek sprawdzamy czy możemy dokończyć jakiś proces z obecnymi zasobami. Jeśli każdy element kolumny jest mniejszy bądź równy elementowi z macierzy *f*, to proces ten wywłaszcza sobie potrzebne zasoby, po czym kończy pracę i oddaje wszystkie swoje zasoby do macierzy *f*.

Jeżeli żaden proces nie może zostać wykonany wywłaszczając sobie zasoby macierzy f to obserwujemy zjawisko ZAKLESZCZENIA się procesów.

Natomiast jeżeli wszystkie procesy wykonały się po sobie dzięki zwalniającym się po sobie zasobom, to nie ma zakleszczeń.

Rozwiązanie

1. $D = \{1, 2, 3, 4, 5\} \wedge f$
2. $j = 4$
3. $D = \{1, 2, 3, 5\} \wedge f' = [2, 1, 2, 4]^T$
4. $j = 3$
5. $D = \{1, 2, 3, 5\} \wedge f'' = [3, 2, 4, 5]^T$
6. itd. aż do $D = \emptyset$

6.2 Zadanie 2: algorytm Holt'a

Stan systemu:

$$A = \begin{matrix} P_1 & P_2 & P_3 & P_4 & P_5 \end{matrix} \begin{bmatrix} 1 & 2 & 1 & 0 & 2 \\ 2 & 3 & 1 & 0 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 2 & 3 & 1 & 1 & 1 \end{bmatrix} \qquad m = \begin{bmatrix} 8 \\ 8 \\ 8 \\ 11 \end{bmatrix} \qquad \rho^a = \begin{bmatrix} 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix}$$

A - Macierz obecnie pobranych zasobów.

m - Transponowany wektor zasobów (wszystkie zasoby w sumie, te zajęte i wolne).

ρ^a - Macierz zapotrzebowań procesów na określone zasoby.

Na podstawie macierzy A i wektora m obliczamy: wektor f który jest różnicą danego wiersza m i sumy wiersza A, macierz trzy-wymiarową E (gdzie 3-ci wymiar to indeksy procesu) która jest posortowaną macierzą ρ^a, wektor I to indeksy na każdym wierszu na którym się zatrzymujemy, a c to licznik procesów i ich zapotrzebowań.

$$f = \begin{bmatrix} (8-6)=2 \\ (8-7)=1 \\ (8-8)=0 \\ (11-8)=3 \end{bmatrix} \qquad E = \begin{bmatrix} 1_2 & 1_4 & 1_5 & 2_1 & 2_3^{\downarrow} \\ 1_3 & 1_4^{\downarrow\Downarrow} & 2_1 & 2_2 & 2_5 \\ 0_4^{\downarrow} & 1_1 & 1_2 & 1_3 & 1_5^{\Downarrow} \\ 0_1 & 0_3 & 1_2 & 2_4 & 2_5^{\downarrow} \end{bmatrix} \qquad I = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$c = \begin{matrix} \text{Liczba wątków} & P_1 & P_2 & P_3 & P_4 & P_5 \\ [5 & 4 & 4 & 4 & 4 & 4] \end{matrix}$$

Algorytm:

1. Zaczynając od numeru pozycji w wektorze I dla danego wiersza macierzy E sprawdzamy czy kolejne pozycje są mniejsze niż ta w wektorze f.
2. Jeżeli tak to inkrementujemy pozycje w I i zmniejszamy c dla danego procesu
3. Jeżeli nie przerywamy i przechodzimy do następnego wiersza.
4. Jeżeli po przejściu przez wszystkie wiersze macierzy E, któraś pozycja poza pierwszą w wektorze c jest równa 0, to zmniejszamy pierwszą pozycje w c dodajemy wartości z odpowiedniej kolumny macierzy A do f i wracamy do pierwszego kroku.
5. Jeżeli pierwsza pozycja w wektorze c jest równa 0 to kończymy brak zakleszczeń.
6. Jeżeli nie możemy dalej wykonać kroków to występuje zakleszczenie.

Rozwiązanie

1. $f = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 3 \end{bmatrix}$ $I = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ $c = \begin{bmatrix} 5 & 4 & 4 & 4 & 4 & 4 \end{bmatrix}$
2. $f' = \begin{bmatrix} 2 \\ 1 \\ 2 \\ 4 \end{bmatrix}$ $I' = \begin{bmatrix} 6 \\ 3 \\ 2 \\ 6 \end{bmatrix}$ $c' = \begin{bmatrix} 4 & 2 & 2 & 1 & 0 & 2 \end{bmatrix}$
3. $f'' = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 5 \end{bmatrix}$ $I'' = \begin{bmatrix} 6 \\ 3 \\ 6 \\ 6 \end{bmatrix}$ $c'' = \begin{bmatrix} 3 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$
4. W następnej iteracji algorytmu przeszlibyśmy drugi wiesz macierzy do końca, rysując 'potrójną' strzałkę na ostatnim elemencie w drugim wierszu macierzy E. Każdy element wektora c zostałby wyzerowany - brak zakleszczeń.

6.3 Zadanie 3: sprawdzanie bezpieczeństwa

Stan systemu:

$$A = \begin{matrix} P_1 & P_2 & P_3 & P_4 & P_5 \end{matrix} \begin{bmatrix} 1 & 2 & 1 & 0 & 2 \\ 2 & 3 & 1 & 0 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 2 & 3 & 1 & 1 & 1 \end{bmatrix} \qquad m = \begin{bmatrix} 8 \\ 8 \\ 8 \\ 11 \end{bmatrix} \qquad C = \begin{bmatrix} 3 & 3 & 3 & 1 & 3 \\ 4 & 5 & 2 & 1 & 3 \\ 2 & 3 & 3 & 2 & 2 \\ 2 & 4 & 1 & 3 & 3 \end{bmatrix}$$

A - Macierz obecnie pobranych zasobów.

m - Transponowany wektor zasobów (wszystkie zasoby w sumie, te zajęte i wolne).

ρ^a - Macierz zapotrzebowań procesów na określone zasoby.

Na podstawie macierzy A i wektora m obliczamy: wektor f który jest różnicą danego wiersza m i sumy wiersza A, macierz H która jest różnicą C i A.

$$f = \begin{bmatrix} (8-6) = 2 \\ (8-7) = 1 \\ (8-8) = 0 \\ (11-8) = 3 \end{bmatrix} \qquad H = \begin{bmatrix} 2 & 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 1 & 2 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix}$$

Sprawdzamy tą samą metodą co w zadaniu 2 tylko że zamiast ρ^a jest $H($ zakładamy że procesy żądają maksymalnej możliwej wartości zasobów, aka. do limitu $C)$. Sprawdzanie dla kolejnych chwil polega na dodaniu do macierzy A podanego wektora $\rho^a_{t_n}(P_x)$ (A co za tym idzie zmieni się macierz H i wektor f). Jeżeli system nadal jest bezpieczny to uznajemy utrzymane A za A, a jeżeli nie to odrzucany nowe A i wracamy do ostatniego bezpiecznego.