

Jerzy Brzeziński

Programowanie współbiezne



Literatura

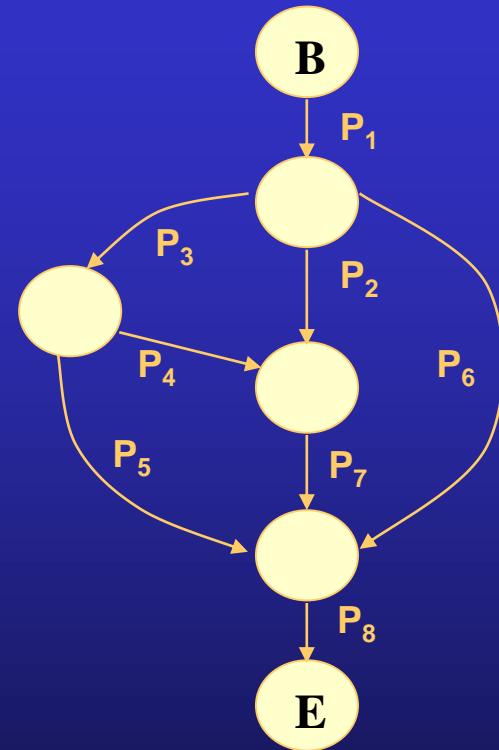
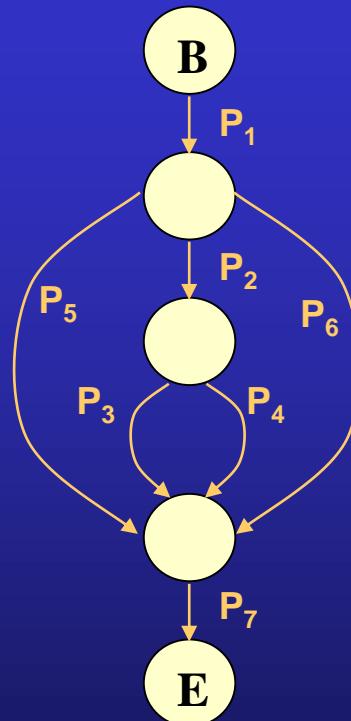
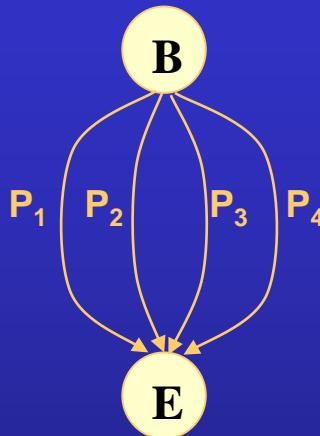
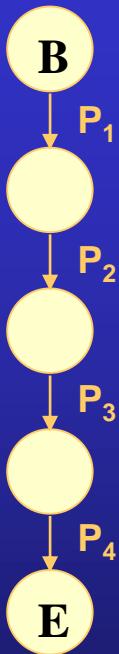
1. **M. Raynal:** *Concurrent Programming: Algorithms, Principles, and Foundations*, Springer-Verlag, Berlin Heidelberg, 2013
2. **A. Silberschatz, P.B. Galvin:** *Podstawy systemów operacyjnych*, WNT, Warszawa, 2000.
3. **Z. Weiss, T. Gruźlewski:** *Programowanie współbieżne i rozproszone*, WNT, Warszawa, 1993
4. **C.A.R. Hoare:** *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985
5. **M. J. Rochkind:** *Programowanie w systemie Unix dla zaawansowanych*, WNT, Warszawa, 1993.
6. **R. W. Stevens:** *Programowanie zastosowań sieciowych w systemie Unix*, WNT, Warszawa, 1995.
7. **M. Gabassi, B. Dupouy:** *Przetwarzanie rozproszone w systemie Unix*, LUPUS, Warszawa, 1995.
8. .

Grafy 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.



Przykład grafów



Legenda:

B-begin

E-end



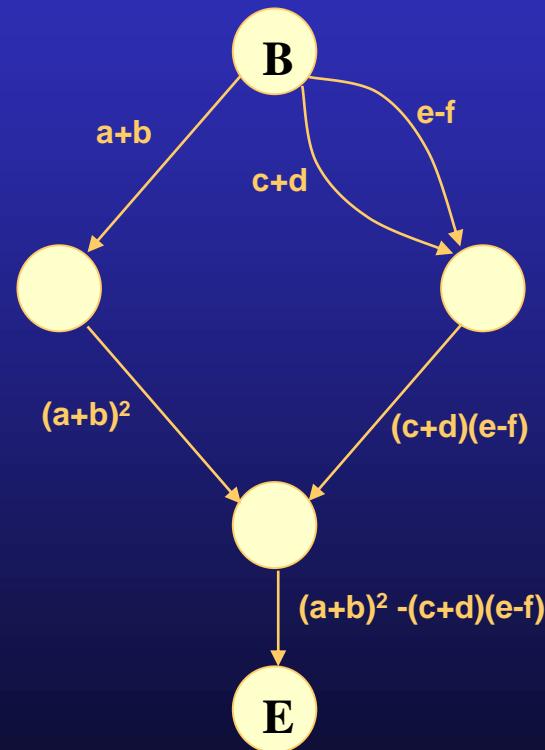
Dobrze zagnieżdzony graf przepływu procesów

Graf przepływu procesów jest **dobrze zagnieżdzony**, 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.

Przykład

Przedstawić graf przepływu procesów odpowiadający wyznaczeniu wyrażenia

$$y := (a + b)^2 - (c + d)(e - f)$$



Notacja "and" (Wirth).

Współbieżne wykonanie może być specyfikowane za pomocą operatora **and**, który łączy dwa wyrażenia wykonywane współbieżnie: *a and b*;;.

Przykład:

...

begin

x1:=x1+2;

y1:=x1+y1;

end

and

*x2:=2*x2+y2;*

y2:=x1+x2+y1+y2;

...

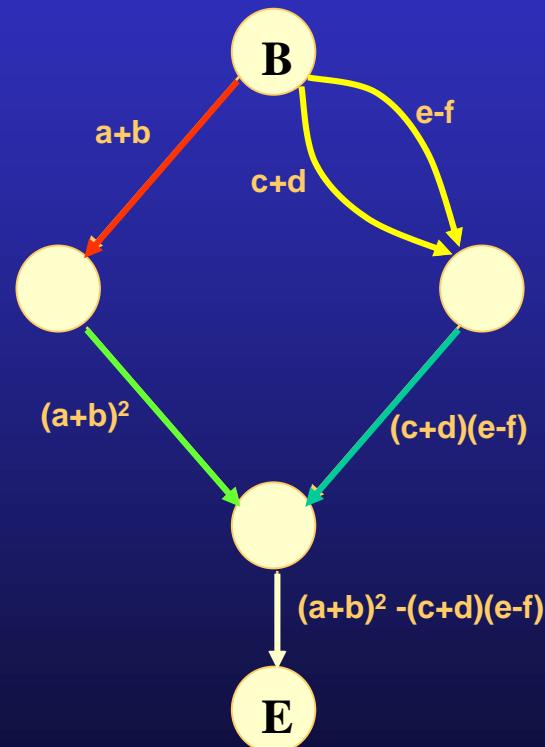


Przykład notacji „and”

Zastosowanie notacji and do implementacji programu wyznaczającego wartość wyrażenia:

$$(a+b)^2 - (c+d)(e-f)$$

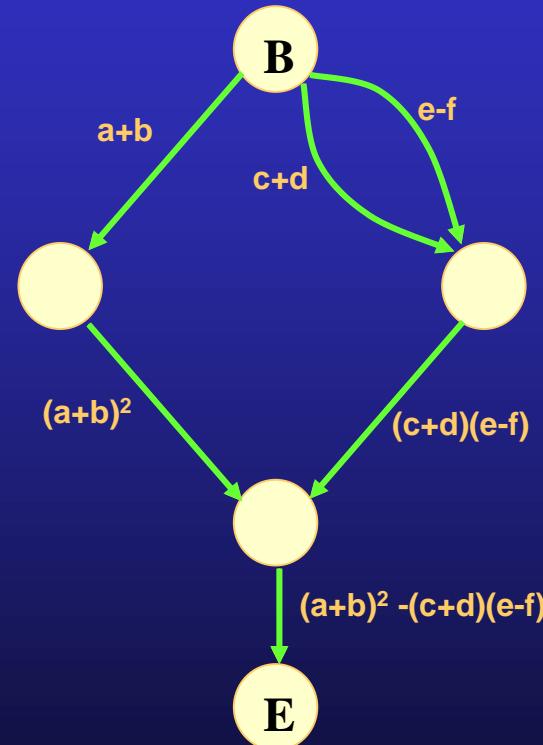
```
begin
  begin
    x1:=a+b;
    x2:=x1*x1;
  end
  and
  begin
    x3:=c+d
    and
    x4:=e-f;
    x5:=x3*x4;
  end;
  x6:=x2-x5;
end.
```



Notacja "parbegin - parend" ("cobegin - coend", Dijkstra)

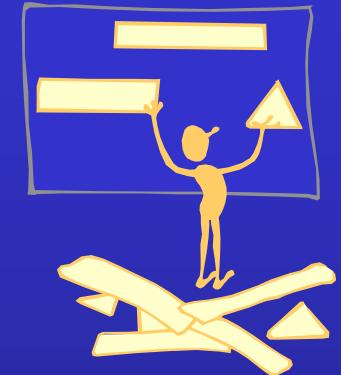
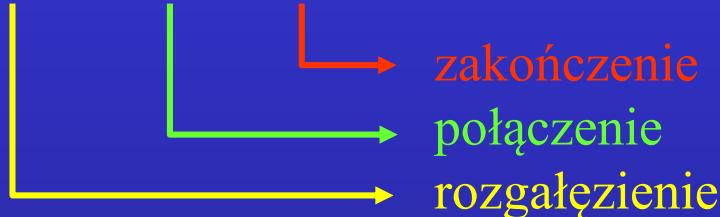
Wszystkie wyrażenia ujęte w nawiasy parbegin – parend; są wykonywane współbieżnie

```
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.
```



Notacja "fork, join, quit" (Conway)

fork join quit



- Instrukcja **quit** powoduje zakończenie procesu.
- Instrukcja **fork w** oznacza, że proces w którym wystąpiła ta instrukcja będzie dalej wykonywany współbieżnie z procesem identyfikowanym przez etykietę *w*.
- Instrukcja **join t, w** ma dwa argumenty, z których *t* jest licznikiem a *w* - etykietą

Wykonanie instrukcji *join t, w* oznacza:

$t := t - 1;$

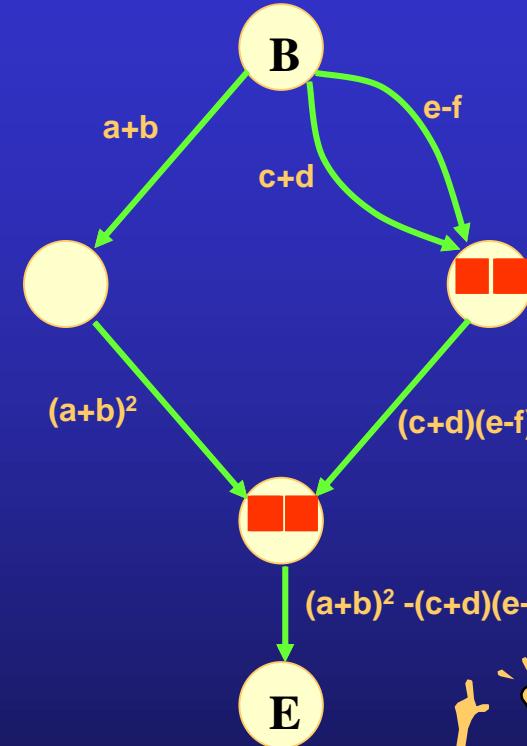
if $t = 0$ **then go to** *w*;

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

Przykład notacji „fork, join, quit”

begin

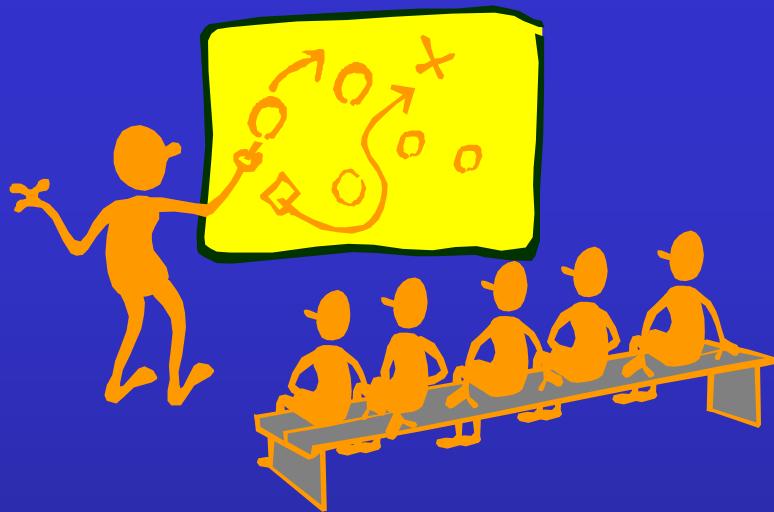
```
t1:=2;           w1:   x1:= a + b;  
t2:=2;           x2:= x1 * x1;  
fork w1;         join t1, w5;  
fork w2;         quit;  
fork w3;         x3:= c + d;  
quit;            join t2, w4;  
w3:              quit;  
x4:= e - f;  
join t2, w4;  
quit;  
w4:              x5:= x3 * x4;  
join t1, w5;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



Koniec

*... kolejne spotkanie,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*





Jerzy Brzeziński

Synchronizacja cz. I

Problem wzajemnego wykluczania (1)

Rozważamy system, w którym współbieżnie wykonywane są procesy od 1 do n .

Zakładamy, że nie są znane względne prędkości wykonywania tych procesów, tzn. że liczba instrukcji wykonywanych przez poszczególne procesory w jednostce czasu może być dowolna. Przyjmujemy ponadto, że procesy te mają dostęp do wspólnych zasobów.

Rozważmy dla przykładu dwa procesy:

P1: $x := x + 1;$

P2: $x := x + 1;$



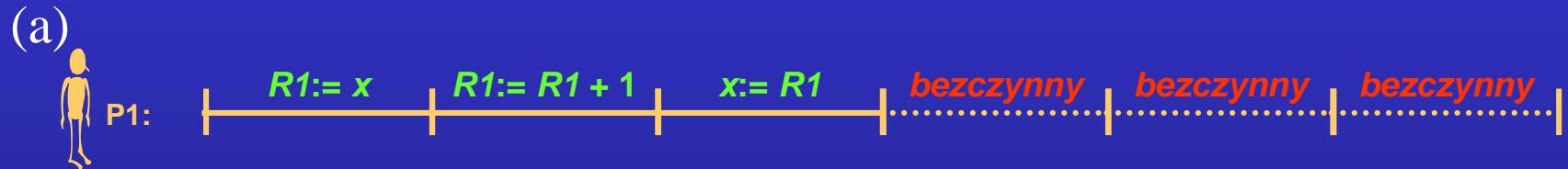
Problem wzajemnego wykluczania ⁽²⁾

Założymy w tym przypadku, że każde uaktualnienie składa się z trzech faz:

- $R := x;$
pobranie wartości zmiennej x do rejestru wewnętrznego procesora
- $R := R + 1$
inkrementacja zawartości rejestru wewnętrznego procesora
- $x := R$
zapisanie zawartości rejestru do zmiennej x

Sekwencja wykonania współbieżnych procesów

Rozważmy możliwe sekwencje wykonywania takich współbieżnych procesów.



Sformułowanie formalne problemu wzajemnego wykluczania

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 procesami cyklicznymi

Zakłada się ponadto:

1. Zapis i odczyt wspólnych danych jest operacją niepodzielnią, a próba jednocześnie 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. Procesy realizujące instrukcje poza sekcją krytyczną nie mogą uniemożliwiać innym procesom wejścia do sekcji krytycznej.
6. 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.

Rozwiązywanie programowe problemu wzajemnego wykluczania (1)

```
1. program VERSIONONE;
2. var processNumber: INTEGER;
3. procedure PROCESSONE;
4. begin
5.   while True do
6.     begin
7.       while processNumber=2 do;
8.       criticalSectionOne;
9.       processNumber:=2;
10.      otherStuffOne;
11.    end;
12.  end;
```

```
13. procedure PROCESSTWO;
14. begin
15.   while True do
16.     begin
17.       while processNumber=1 do;
18.       criticalSectionTwo;
19.       processNumber:=1;
20.       otherStuffTwo;
21.     end;
22.   end;
```

```
23. begin
24.   processNumber:= 1;
25.   parbegin
26.     PROCESSONE;
27.     PROCESSTWO;
28.   parend;
29. end.
```



Rozwiązywanie programowe problemu wzajemnego wykluczania⁽²⁾

```
1. program VERSIONTWO;
2. var P1inside, P2inside: BOOLEAN;

3. procedure PROCESSONE;
4. begin
5.   while True do
6.     begin
7.       while P2inside do;
8.       P1inside:=True;
9.       criticalSectionOne;
10.      P1inside:=False;
11.      otherStaffOne;
12.    end;
13. end;                                14. procedure PROCESSTWO;
15. begin
16.   while True do
17.     begin
18.       while P1inside do;
19.       P2inside:=True;
20.       criticalSectionTwo;
21.       P2inside:=False;
22.       otherStaffTwo;
23.     end;
24. end;
25. begin
26.   P1inside:=False;
27.   P2inside:=False;
28.   parbegin
29.     PROCESSONE;
30.     PROCESSTWO;
31.   parend;
32. end.
```



Rozwiązanie programowe problemu wzajemnego wykluczania (3)

```
1. program VERSIONTHREE;
2. var P1WantsToEnter: BOOLEAN;
3. var P2WantsToEnter: BOOLEAN;

4. procedure PROCESSONE;
5. begin
6.   while True do
7.     begin
8.       P1WantsToEnter:=True;
9.       while P2WantsToEnter do;
10.      criticalSectionOne;
11.      P1WantsToEnter:=False;
12.      otherStuffOne;
13.    end;
14. end;                                15. procedure PROCESSTWO;
16. begin
17.   while True do
18.     begin
19.       P2WantsToEnter:=True;
20.       while P1WantsToEnter do;
21.       criticalSectionTwo;
22.       P2WantsToEnter:=False;
23.       otherStuffTwo;
24.     end;
25. end;
26. begin
27.   P1WantsToEnter:=False;
28.   P2WantsToEnter:=False;
29.   parbegin
30.     PROCESSONE;
31.     PROCESSTWO;
32.   parend;
33. end.
```



Rozwiązywanie programowe problemu wzajemnego wykluczania (4)

```
1. program VERSIONFOUR;
2. var P1WantsToEnter: BOOLEAN;
3. var P2WantsToEnter: BOOLEAN;
4. procedure PROCESSONE;
5. begin
6.   while True do
7.     begin
8.       P1WantsToEnter:=True;
9.       while P2WantsToEnter do
10.         begin
11.           P1WantsToEnter:=False;
12.           delay(random, freecycles);
13.           P1WantsToEnter:=True;
14.         end;
15.         criticalSectionOne;
16.         P1WantsToEnter:=False;
17.         otherStuffOne;
18.       end;
19.     end;
```



```
20.   procedure PROCESSTWO;
21.   begin
22.     while True do
23.       begin
24.         P2WantsToEnter:=True;
25.         while P1WantsToEnter do
26.           begin
27.             P2WantsToEnter:=False;
28.             delay(random, freecycles);
29.             P2WantsToEnter:=True;
30.           end;
31.           criticalSectionTwo;
32.           P2WantsToEnter:=False;
33.           otherStuffTwo;
34.         end;
35.       end;
```

```
20.   begin
21.     P1WantsToEnter:=False;
22.     P2WantsToEnter:=False;
23.     parbegin
24.       PROCESSONE;
25.       PROCESSTWO;
26.     parend;
27.   end.
```

Algorytm Dekkera (1)

```
1. program DEKKERALGORITHM;
2. var favoredProcess: enum (First, Second);
3. var P1WantsToEnter, P2WantsToEnter: BOOLEAN;
4.
5. procedure PROCESSONE;
6. begin
7.   while True do
8.     begin
9.       P1WantsToEnter:=True;
10.      while P2WantsToEnter do
11.        if favoredProcess=Second then
12.          begin
13.            P1WantsToEnter:=False;
14.            while favoredProcess=Second do;
15.            P1WantsToEnter:=True;
16.          end;
17.          criticalSectionOne;
18.          favoredProcess:=Second;
19.          P1WantsToEnter:=False;
20.          otherStuffOne;
21.        end;
22.      end;
```



Algorytm Dekkera (2)



```
22. procedure PROCESSTwo;
23. begin
24.   while True do
25.     begin
26.       P2WantsToEnter:= True;
27.       while P1WantsToEnter do
28.         if favoredProcess=First then
29.           begin
30.             P2WantsToEnter:=False;
31.             while favoredProcess=First do;
32.             P2WantsToEnter:= True;
33.           end;
34.           criticalSectionTwo;
35.           favoredProcess:=First;
36.           P2WantsToEnter:=False;
37.           otherStuffTwo;
38.         end;
39.     end;
40.   begin
41.     P1WantsToEnter:= False;
42.     P2WantsToEnter:= False;
43.     favoredProcess:= First;
44.     parbegin
45.       PROCESSONE;
46.       PROCESSTwo;
47.     parend;
48.   end.
```

Algorytm Dijkstry

```
1. program DIJKSTRAALGORITHM;
2. begin
3.   shared
4.     flag[1..n]: 0..2;
5.     turn      : 1..n;
6.   local
7.     testi : 0..2;
8.     k, otheri, tempi: 1..n;
9.   while True do
10.    begin
11.      L: flag[i]:=1;
12.      otheri:=turn;
13.      while otheri≠i do
14.        begin
15.          testi:=flag[otheri];
16.          if testi=0 then
17.            turn:=i;
18.            otheri:=turn;
19.          end;
20.          flag[i]:=2;
21.        for k:=1 to n do
22.          if k≠i then
23.            begin
24.              testi:=flag[k];
25.              if testi=2 then
26.                goto L;
27.            end;
28.            criticalSection;
29.            flag[i]:=0;
30.            reminderSection;
31.          end;
32.        end.
```



Algorytm Petersona dla 2 procesów

```
1. program PATERSONALGORITHM;
2. begin
3.   shared
4.     flag[0..1]: BOOLEAN;
5.     turn: INTEGER;
6.   local
7.     otheri: BOOLEAN;
8.     whosei: INTEGER;
9.   while True do
10.    begin
11.      flag[i]:= True;
12.      turn := 1-i;
13.      repeat
14.        whosei:=turn;
15.        otheri := flag[1-i];
16.      until (whosei=i or not otheri);
17.      criticalSection;
18.      flag[i] := False;
19.      reminderSection;
20.    end;
21. end.
```



Algorytm Petersona dla n procesów

```
1. program PATERSONALGORITHM_N;
2. begin
3.   shared
4.     flag[1..n]: INTEGER;
5.     turn[1..n-1]: INTEGER;
6.   local
7.     k, l, otheri, whosei: INTEGER;
8.   while True do
9.     begin
10.    for k:=1 to n-1 do
11.      begin
12.        flag[i]:=k;
13.        turn[k]:=i;
14.        repeat
15.          whosei:=turn[k];
16.          if whosei≠i then break;
17.          for l:=1 to n do
18.            begin
19.              if l≠i then
20.                otheri:=flag[l];
21.                if otheri≥k then break;
22.              end;
23.            until otheri<k;
24.          end;
25.          criticalSection;
26.          flag[i]:=0;
27.          reminderSection;
28.        end;
29.      end.
```



Algorytm Lamporta dla n procesów

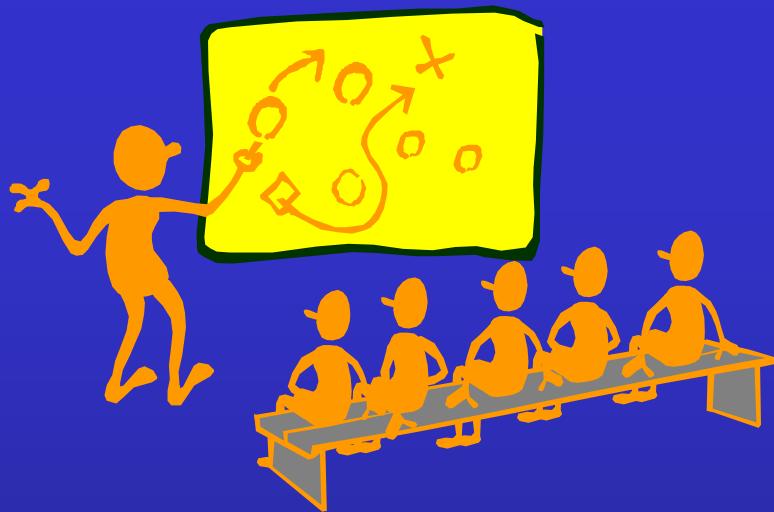
```
1. program LAMPORTALGORITHM;
2. begin
3.   shared
4.     choosing[1..n]: 0..1;
5.     num[1..n]: INTEGER;
6.   local
7.     testi: 0..1;
8.     k, minei : INTEGER;
9.     otheri, tempi: INTEGER;
10.  while True do
11.    begin
12.      choosing[i]:=1;
13.      minei:= 0;
14.      for k:=1 to n do
15.        if k≠i then
16.          begin
17.            tempi:=num[k] ;
18.            minei:=max(minei, tempi);
19.          end;
20.          minei:= minei+1;
21.          num[i]:= minei;
22.          choosing[i]:=0;
23.        for k:=1 to n do
24.          if k≠i then
25.            begin
26.              repeat
27.                testi:=choosing[k];
28.              until testi=0;
29.              repeat
30.                otheri:=num[k];
31.              until otheri=0 or
32.                (minei, i)<(otheri, k);
33.              end;
34.              criticalSection;
35.              num[i]:=0 ;
36.              reminderSection;
37.            end;
38.          end.
```



Koniec

*... kolejne spotkanie,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*





Synchronizacja

cz. II

Instrukcja testandset

Założmy, że w systemie dostępna jest instrukcja typu $\text{testandset}(a, b)$, która w sposób atomowy (niepodzielny) dokonuje odczytu zmiennej b , zapamiętania wartości tej zmiennej w zmiennej a oraz przypisania zmiennej b wartości true.

$\text{testandset}(a, b)$ is equivalent to :

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

$\text{nottestandset}(a, b)$ is equivalent to :

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



Przykład - testandset

```
1. program TESTANDSET_EXAMPLE;
2. var active: BOOLEAN;

3. procedure PROCESSONE;
4. var oneCannotEnter: BOOLEAN;
5. begin
6.   while True do
7.     begin
8.       oneCannotEnter:=True;
9.       while oneCannotEnter do
10.         testandset
11.           (oneCannotEnter, active);
12.           criticalSectionOne;
13.           active:=False;
14.           otherStuffOne;
15.     end;
16.   procedure PROCESSTWO;
17.   var twoCannotEnter: BOOLEAN;
18.   begin
19.     while True do
20.       begin
21.         twoCannotEnter:=True;
22.         while twoCannotEnter do
23.           testandset
24.             (twoCannotEnter, active);
25.             criticalSectionTwo;
26.             active:=False;
27.             otherStuffTwo;
28.         end;
29.       begin
30.         active:=False;
31.         parbegin
32.           PROCESSONE;
33.           PROCESSTWO;
34.         parend
35.       end.
```



Semafony

Semaforem nazywamy zmienną chronioną, na ogólnie 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
przyjmujące tylko wartość 0 lub 1,
- ogólne (licznikowe)
mogą przyjąć nieujemną wartość całkowitoliczbową.

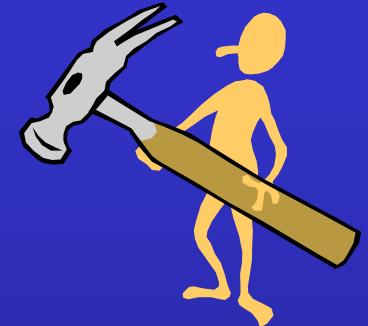


Operacje P(S) i V(S)

Operacje P i V (Dijkstra)

Oznaczenie:

- **P** – pochodzi od holenderskiego proberen (testuj),
- **V** – pochodzi od holenderskiego od **verhogen** (inkrementuj)



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 następująco:

```
if (one or more processes are waiting on S)  
    then (let one of these processes proceed)  
    else S := S + 1
```

Przykład - semafory

```
1. program SEMAPHOREEXAMPLE;
2. var active: SEMAPHORE;

3. procedure PROCESSONE;
4. begin
5.   while True do
6.     begin
7.       P(active);
8.       criticalSectionOne;
9.       V(active);
10.      otherStuffOne;
11.    end
12.  end;

13. begin
14.   semaphore_initialize(active,1);
15.   parbegin
16.     PROCESSONE;
17.     ...
18.     PROCESSNTH;
19.   parend
20. end.
```



Problem producenta-konsumenta

```
1. program PRODUCERCONSUMER;
2. var emptyBuffers, fullBuffers, active: SEMAPHORE;
3. procedure PRODUCER;
4. begin
5.   while True do
6.     begin
7.       produceNextRecord;
8.       P(emptyBuffers);
9.       P(active);
10.      addToBuffer;
11.      V(active);
12.      V(fullBuffers);
13.    end
14.  end;
15. procedure CONSUMER;
16. begin
17.   while True do
18.     begin
19.       P(fullBuffers);
20.       P(active);
21.       takeFromBuffer;
22.       V(active);
23.       V(emptyBuffers);
24.       processNextRecord;
25.     end
26.   end;
27. begin
28.   semaphore_initialize(active, 1);
29.   semaphore_initialize(emptyBuffers, N);
30.   semaphore_initialize(fullBuffers, 0);
31.   parbegin
32.     PRODUCER;
33.     CONSUMER;
34.   parend
35. end.
```



Semafony binarne

Semafony binarne ***Sb*** mogą przyjmować tylko dwie wartości: 0 i 1.

Przez Pb i Vb oznaczone są operacje na semaforach binarnych odpowiadające operacjom 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.

Implementacja binarnych operacji semaforowych z aktywnym czekaniem

Pb(Sb) is equivalent to:

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

Vb(Sb) is equivalent to :

```
Sb := True ;
```



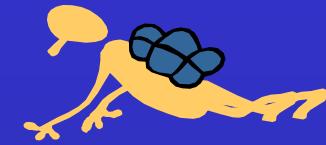
Specyfikacja implementacji z aktywnym czekaniem (ang.busy wait)

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

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

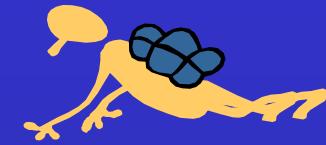


Implementacja z aktywnym czekaniem



```
1. program PV_IMPLEMENTATION;
2. var active, delay: BOOLEAN;
3. var NS: INTEGER;
4. procedure PIMPLEMENTATION;
5. var pActive, pDelay: BOOLEAN;
6. begin
7.   pActive:=True;
8.   while pActive do
9.     testandset(pActive, active);
10.    NS:=NS-1;
11.    if NS ≥ 0 then
12.      begin
13.        S:=S-1;
14.        active:=False;
15.      end
16.    else
17.      begin
18.        active:=False;
19.        pDelay:=True;
20.        while pDelay do
21.          testandset(pDelay, delay)
22.        end
23.      end;
24. procedure VIMPLEMENTATION;
25. var vActive: BOOLEAN;
26. begin
27.   vActive:=True;
28.   while vActive do
29.     testandset(vActive, active);
30.     NS:=NS+1;
31.     if NS > 0 then
32.       S:=S+1
33.     else
34.       delay:=False;
35.       active:=False
36.     end;
37.   begin
38.     active:=False;
39.     delay:=True;
40.   end.
```

Implementacja z aktywnym czekaniem

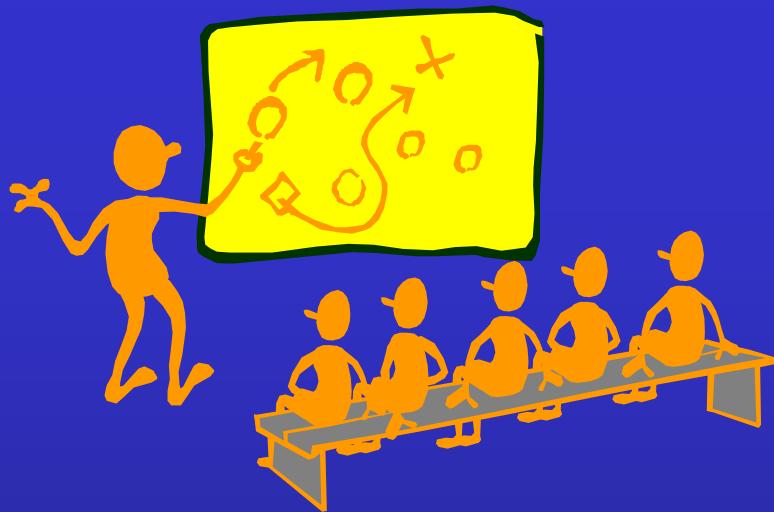


```
1. program PV_IMPLEMENTATION;
2. var active, delay: BOOLEAN;
3. var NS: INTEGER;
4. procedure PIMPLEMENTATION;
5. var pActive, pDelay: BOOLEAN;
6. begin
7.   pActive:=True;
8.   while pActive do
9.     testandset(pActive, active);
10.    NS:=NS-1;
11.    if NS ≥ 0 then
12.      S:=S-1
13.    else
14.      begin
15.        active:=False;
16.        pDelay:=True;
17.        while pDelay do
18.          testandset(pDelay, delay)
19.        end;
20.        active:=False;
21.      end;
24. procedure VIMPLEMENTATION;
25. var vActive: BOOLEAN;
26. begin
27.   vActive:=True;
28.   while vActive do
29.     testandset(vActive, active);
30.     NS:=NS+1;
31.     if NS > 0 then
32.       begin
33.         S:=S+1;
34.         active:=False
35.       end
36.     else
37.       delay:=False
38.     end;
39.   begin
40.     active:=False;
41.     delay:=True;
42.   end.
```

Koniec

*... za tydzień,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*





Synchronizacja

cz. III

Implementacja operacji P i V

```
1. program PV_IMPLEMENTATION;
2.   var active, delay: BOOLEAN;
3.   var NS: INTEGER;
4.
5. procedure PIMPLEMENTATION;
6.   var pActive : BOOLEAN;
7.   begin
8.     Disable interrupts;
9.     pActive:=True;
10.    while pActive do
11.      testandset(pActive,active);
12.      NS:=NS-1;
13.      if NS ≥ 0 then
14.        begin
15.          S:=S-1;
16.          active:=False;
17.          Enable interrupts;
18.        end
19.      else
20.        begin
21.          Block process invoking P(S);
22.          p:= Remove from RL;
23.          active:=False;
24.          Transfer control to p with
25.          Enable interrupts;
26.        end
27.    end;
```

```
1. procedure VIMPLEMENTATION;
2.   var vActive : BOOLEAN;
3.   begin
4.     Disable interrupts;
5.     vActive:=True;
6.     while vActive do
7.       testandset(vActive,active);
8.       NS:=NS+1;
9.       if NS > 0 then
10.         S:=S+1;
11.       else
12.         begin
13.           p:=remove from LS;
14.           Add p to RL;
15.         end;
16.       active:=False;
17.     end;
```



Legenda:

- LS – List associated with S
- RL – Ready List

Implementacja operacji wait i signal

```
1. type SEMAPHORE = record
2.           value: INTEGER;
3.           L: list of process;
4.       end;
```

Implementacja operacji $wait(S) = P(S)$:

```
5. procedure WAIT(S);
6. begin
7.   S.value:=S.value - 1;
8.   if S.value < 0 then
9.     begin
10.      add this process ID to S.L;
11.      block this process;
12.    end;
13. end;
```

Implementacja operacji $signal(S) = V(S)$:

```
14. procedure SIGNAL(S);
15. begin
16.   S.value:=S.value + 1;
17.   if S.value ≤ 0 then
18.     begin
19.       remove a process P from S.L;
20.       wakeup(P);
21.     end;
22. end;
```



Inne operacje semaforowe (1)

```
1. lock w:  
2.   L: if w = 1 then go to L  
3.   else w := 1;  
  
4. unlock w:  
5.   w := 0;  
  
6. ENQ(r):  
7.   if inuse[r] then // resource r is used  
8.     begin  
9.       Insert p on r-queue;  
10.      Block p  
11.    end // queue associated with r  
12.   else  
13.     inuse[r] := True ;  
  
14. DEQ(r):  
15.   p:=Remove from r-queue  
16.   if p ≠ Ω  
17.     then Activate p // p = Ω means that queue was empty  
18.   else inuse[r] := False ;
```



Inne operacje semaforowe (2)

```
1. WAIT (e) :
2.     if  $\neg$  posted[e] then // only one process can wait for event e
3.         begin
4.             wait[e] := True;
5.             process[e] := p;
6.             Block p;
7.         end
8.     else posted[e] := False;

9. POST (e) :
10.    if  $\neg$  posted[e] then
11.        begin
12.            posted[e] := True;
13.            if wait[e] then
14.                begin
15.                    wait[e] := False;
16.                    posted[e] := False;
17.                    Activate process[e];
18.                end;
19.        end;
```



Inne operacje semaforowe ⁽³⁾

```
1. Block(i) :  
2.   if  $\neg$  wws[i]    // wait for Wakeup flag associated with process i  
3.     then Block process i  
4.   else wws[i] := False;  
  
5. Wakeup(i) :  
6.   if ready(i)    // process is ready  
7.     then wws[i] := True  
8.   else Activate process i;
```



Three operations are defined on a event counter E :

- ❖ $\text{read}(E)$ – return the current value of E .
- ❖ $\text{advance}(E)$ – automatically increment E by 1.
- ❖ $\text{await}(E, v)$ – wait until E has a value of v or more.

Producer-consumer problem using event counters

```
1. #include "prototypes.h"
2. #define N 100                                // number of slots in the buffer
3. typedef INT EVENT_COUNTER;      // event counters are a special kind of int
4. EVENT_COUNTER in=0;                  // counts items inserted into buffer
5. EVENT_COUNTER out=0;                 // counts items removed from buffer

6. void PRODUCER(void) {
7.     INT item, sequence =0;
8.     while(True) {                         // infinite loop
9.         produce_item(&item);           // generate something to put in buffer
10.        sequence=sequence+1;          // count items produced so far
11.        await(out, sequence-N);       // wait until there is room in buffer
12.        enter_item(item);           // put item in slot (sequence -1) % N
13.        advance(&in);                // let consumer know about another item
14.    }
15. }

16. void CONSUMER(void) {
17.     INT item, sequence=0;
18.     while(True) {                         // infinite loop
19.         sequence=sequence+1;           // number of item to remove from buffer
20.         await(in, sequence);          // wait until required item is present
21.         remove_item(&item);           // take item from slot (sequence-1)%N
22.         advance(&out);              // let producer know that item is gone
23.         consume_item(item);          // do something with the item
24.    }
25. }
```



Regiony krytyczne - 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 ;



Regiony krytyczne - 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);
```



Warunkowy region krytyczny (1)



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 .

Warunkowy region krytyczny (2)

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

Proces produkujący umieszcza nową jednostkę *nextp* w buforze dzielonym wykonując instrukcje:

```
5. region buffer when count < n  
6. do begin  
7.   pool[in]:=nextp;  
8.   in:=(in + 1) mod n;  
9.   count:=count + 1;  
10.  end;
```

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

```
?  
11. region buffer when count > 0  
12. do begin  
13.   nextk:=pool[out];  
14.   out:=(out + 1) mod n;  
15.   count:=count - 1;  
16.  end;
```



Warunkowe regiony krytyczne - implementacja

```
1. region v when B do S;  
2. var xMutex, xDelay : SEMAPHORE;  
3. xCount, xTemp : INTEGER;  
                                // xCount - the number of processes waiting for xDelay  
                                // xTemp - the number of processes that have been allowed  
                                to test their Boolean condition during one trace  
4. wait(xMutex);  
5. if not B then  
6.   begin  
7.     xCount:=xCount + 1;  
8.     signal(xMutex);  
9.     wait(xDelay);  
10.    while not B do  
11.      begin  
12.        xTemp:=xTemp + 1;  
13.        if xTemp < xCount then  
14.          signal(xDelay)  
15.        else  
16.          signal(xMutex);  
17.        wait(xDelay);  
18.      end;  
19.      xCount:=xCount - 1;  
20.  end;  
21. S;  
22. if xCount > 0 then  
23.   begin  
24.     xTemp:=0;  
25.     signal(xDelay);  
26.   end;  
27. else  
28.   signal(xMutex);
```



```
region v do
```

```
begin
```

```
    S1;
```

```
    await (B) ;
```

```
    S2;
```

```
end;
```





Problem pisarzy i czytelników

In the readers-writers problem, the shared resource is a file that is accessed by both the reader and writer process. Reader processes simply read the information in the file without changing its content. Writer processes may change the information in the file. The basic synchronization constraint is that any number of readers should be able to concurrently access the file, but only one writer can access the file at a given time. Moreover, readers and writers must always exclude each other.



Rozwiążanie problemu pisarzy i czytelników z użyciem semaforów

```
1. shared var
2.     nReaders : INTEGER;
3.     mutex, wmutex, srmutex : SEMAPHORE;

4. procedure READER;
5. begin
6.     P(mutex);
7.     if nReaders=0 then
8.         begin
9.             nReaders:=nReaders + 1;
10.            P(wmutex);
11.        end
12.    else
13.        nReaders:=nReaders + 1;
14.        V(mutex);
15.        read(f);
16.        P(mutex);
17.        nReaders:=nReaders - 1;
18.        if nReaders = 0 then
19.            V(wmutex);
20.            V(mutex);
21.        end ;

22. procedure WRITER(d: data);
23. begin
24.     P(srmutex);
25.     P(wmutex);
26.     write(f, d);
27.     V(wmutex);
28.     V(srmutex);
29. end;

30. begin // initialization
31.     mutex:=wmutex:=srmutex:=1;
32.     nReaders:=0;
33. end.
```



Rozwiązywanie problemu pisarzy i czytelników z użyciem regionów krytycznych

```
1. var v: shared record  
2.           nReaders, nWriters: INTEGER;  
3.           busy: BOOLEAN;  
4. end;
```



Proces czytelnika

```
5. region v do  
6. begin  
7.   await(nWriters=0);  
8.   nReaders:=nReaders + 1;  
9. end;  
10. ...  
11. read file  
12. ...  
13. region v do  
14. begin  
15.   nReaders:=nReaders - 1;  
16. end;
```

Proces pisarza

```
17. region v do  
18. begin  
19.   nWriters:=nWriters + 1;  
20.   await((not busy) and (nReaders=0));  
21.   busy:=True;  
22. end;  
23. ...  
24. write file  
25. ...  
26. region v do  
27. begin  
28.   nWriters:=nWriters- 1;  
29.   busy:=False;  
30. end;
```

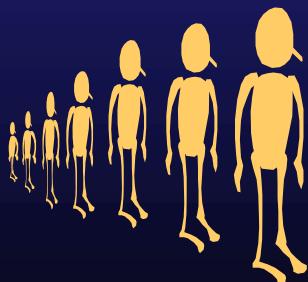
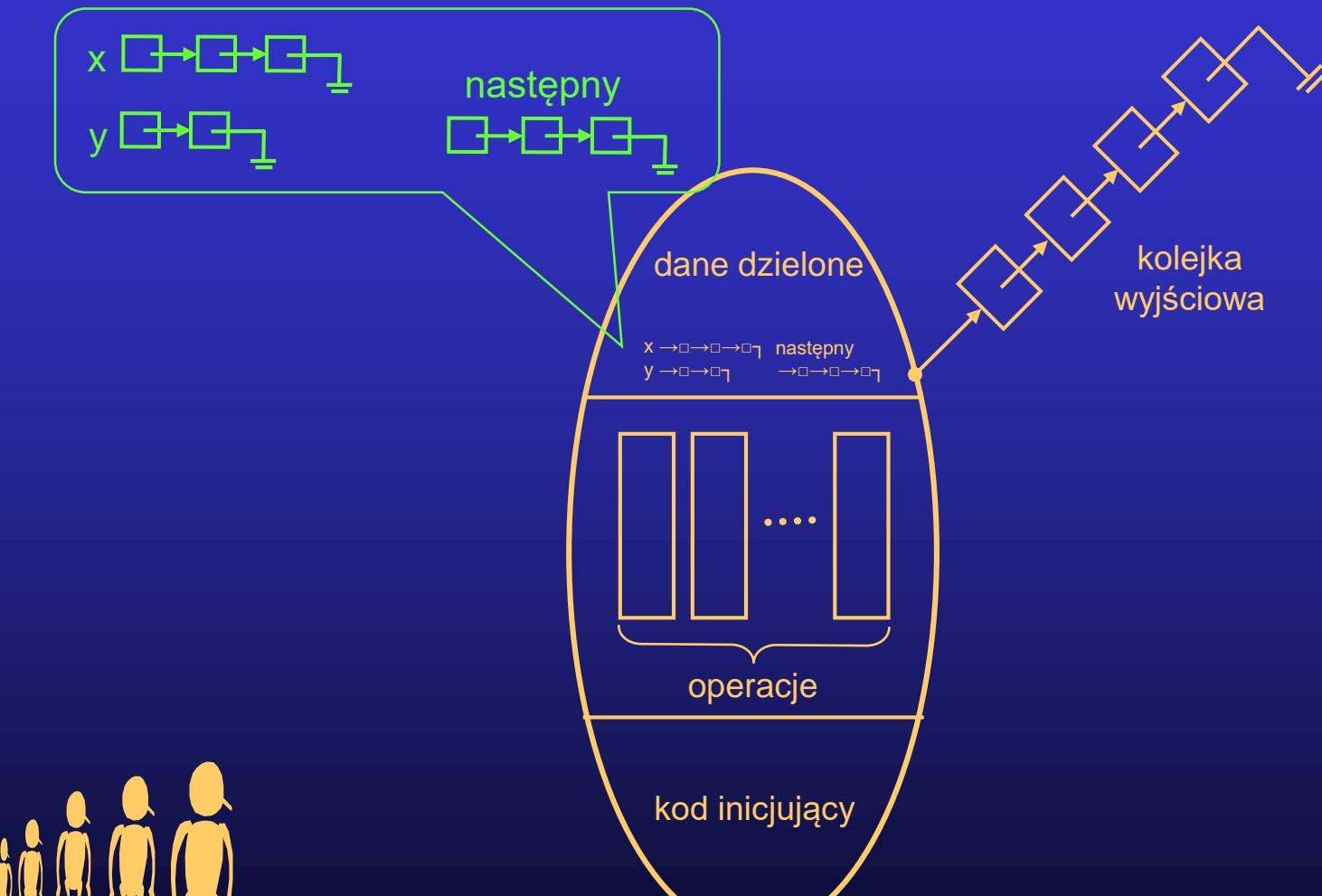
Monitory – definicja

A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is

```
type MONITOR_NAME = monitor  
variable declarations  
  
procedure entry P1 (...);  
begin ... end;  
  
procedure entry P2 (...);  
begin ... end;  
  
:  
  
procedure entry Pn (...);  
begin ... end ;  
  
begin  
    initialization code;  
end ;
```



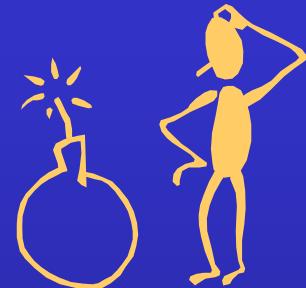
Schemat monitora



Operacje wait i signal

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.signal*

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.

Rozwiązanie problemu producenta – konsumenta z wykorzystaniem monitorów

```
1. type PRODUCER_CONSUMER = monitor
2. var full, empty : CONDITION;
3. count : INTEGER;
4. procedure entry ENTER;
5. begin
6.   if count = N then full.wait;
7.   enter_item;
8.   count:=count + 1;
9.   if count = 1 then empty.signal;
10. end;
11. procedure entry REMOVE;
12. begin
13.   if count = 0 then empty.wait;
14.   remove_item;
15.   count:=count - 1;
16.   if count=N - 1 then full.signal;
17. end;
18. begin
19.   count:=0;
20. end monitor;
21. procedure PRODUCER;
22. begin
23.   while True do
24.     begin
25.       produce_item;
26.       PRODUCER_CONSUMER.ENTER;
27.     end
28.   end;
29. procedure CONSUMER;
30. begin
31.   while True do
32.     begin
33.       PRODUCER_CONSUMER.REMOVE;
34.       consume_item
35.     end;
36.   end.
```

Alokacja zasobów z wykorzystaniem monitora

```
1. type RESOURCE_ALLOCATION = monitor
2. var busy: BOOLEAN;
3.      x: INTEGER;

4. procedure entry ACQUIRE(time : INTEGER);
5. begin
6.     if busy then x.wait(time);    //process priority
7.     busy := True;
8. end;

9. procedure entry RELEASE;
10. begin
11.     busy := False;
12.     x.signal;
13. end;

14. begin
15.     busy := False;
16. end.
```



Rozwiążanie problemu czytelników i pisarzy z wykorzystaniem monitorów

```
1. type READERS_WRITERS = monitor;
2. var      readerCount : INTEGER;
3. busy : BOOLEAN;
4. OKtoRead, OKtoWrite : CONDITION;

5. procedure entry STARTREAD;
6. begin
7.   if busy
8.     then OKtoRead.wait ;
9.   readerCount:=readerCount+1;
10.  OKtoRead.signal;
11. // Once one reader can start, they all can
12. end;

13. procedure entry ENDREAD;
14. begin
15.   readerCount:=readerCount-1;
16.   if readerCount = 0
17.     then OKtoWrite.signal;
18. end;

19. procedure entry STARTWRITE;
20. begin
21.   if busy or readerCount ≠ 0
22.     then OKtoWrite.wait;
23.   busy:=True;
24. end;

25. procedure entry ENDWRITE;
26. begin
27.   busy:=False;
28.   if OKtoRead.queue
29.     then OKtoRead.signal
30.   else OKtoWrite.signal;
31. end;

32. begin // initialization
33.   readerCount:=0;
34.   busy :=False;
35. end ;
```



Monitor – implementacja

```
1. wait(mutex);  
2. ...  
3. treść procedury F;  
4. ...  
5. if nextCount > 0  
6.   then signal(next)  
7. else signal(mutex);
```

- *mutex* – semafor gwarantujący wzajemne wykluczanie
- *nextCount* – liczba procesów, które wywołały *x.signal* i zostały zawieszone
- *next* – semafor umożliwiający zwieszenie procesu wywołującego *x.signal*
- *xCount* – liczba procesów czekających na *x.signal*
- *xSem* – semafor umożliwiający zwieszenie procesu wywołującego *x.wait*

x.wait:

```
8. xCount:=xCount+1;  
9. if nextCount > 0  
10.  then signal(next)  
11.  else signal(mutex);  
12. wait(xSem);  
13. xCount:=xCount-1;
```

x.signal:

```
14. if xCount > 0  
15.  then  
16.    begin  
17.      nextCount:=nextCount+1;  
18.      signal(xSem);  
19.      wait(next);  
20.      nextCount:=nextCount-1;  
21.    end.
```



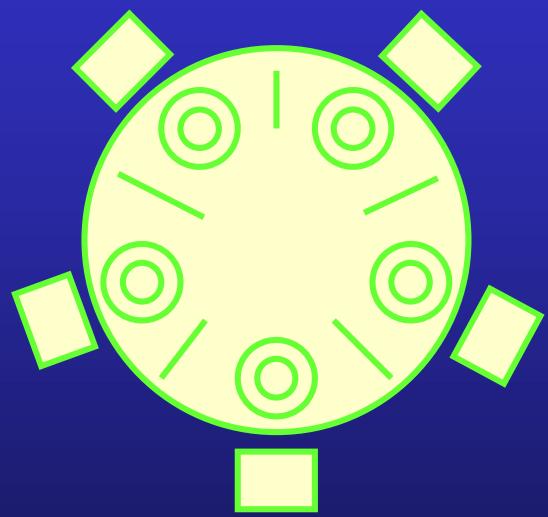
Problem jedzących filozofów

The dining philosophers problem is a classic problem that has formed the basis for a large class of synchronization problems. In one version of this problem five philosophers are sitting in a circle, attempting to eat spaghetti with the help of forks. Each philosopher has a bowl of spaghetti but there are only five forks (with one fork placed on the left and one to the right of each philosopher) to share among them. This creates a dilemma, as both forks (to the left and right) are needed by each philosopher to consume the spaghetti.

A philosopher alternates between two phases: thinking and eating. In the *thinking* mode, the philosopher does not hold a fork. However, when hungry (after staying in the thinking mode for a finite time), a philosopher attempts to pick up both forks on the left and right sides. (At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously). A philosopher can start eating only after obtaining both forks. Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes (which lasts for finite time), both forks are put back in their original position and the philosopher reenters the thinking phase.

Note that no two neighboring philosophers, can eat simultaneously. In any solution to this problem, the act of picking up a fork by a philosopher must be a critical section. Devising a deadlock-free solution to this problem, in which no philosopher starves, is nontrivial.

Schemat filozofów



Rozwiązywanie problemu jedzących filozofów z wykorzystaniem monitorów

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

4. procedure entry PICKUP(i: 0..4);
5. begin
6.   state[i]:=Hungry;
7.   test (i);
8.   if state[i] ≠ eating
9.     then self[i].wait;
10. end ;

11. procedure entry PUTDOWN(i: 0..4);
12. begin
13.   state[i]:=Thinking;
14.   test(i + 4 mod 5);
15.   test(i + 1 mod 5);
16. end;

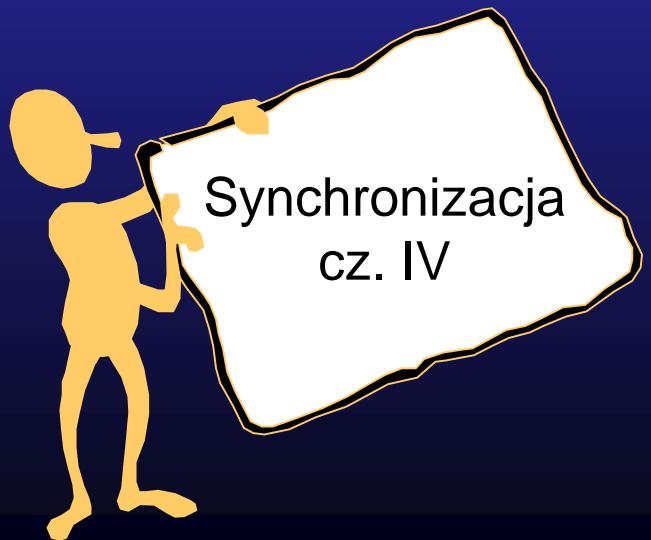
17. procedure TEST(k: 0..4);
18. begin
19.   if state[k+4 mod 5] ≠ Eating
20.   and state[k] = Hungry
21.   and state[k+1 mod 5] ≠ Eating
22.   then
23.     begin
24.       state[k]:=Eating;
25.       self[k].signal;
26.     end;
27. end;

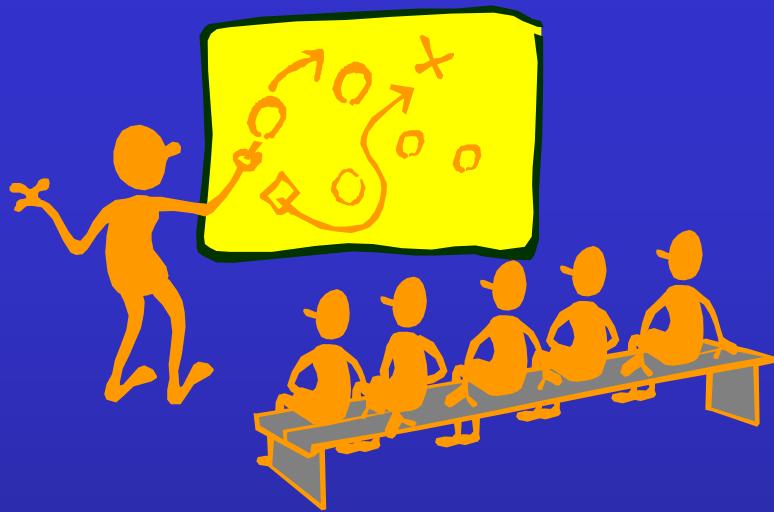
28. begin
29.   for i:=0 to 4 do
30.     state[i]:=Thinking;
31. end ;
```



Koniec

*... kolejne spotkanie,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*





Synchronizacja

cz. IV

Operacje wymiany komunikatów

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

- ❖ $\text{send}(P, m)$
- ❖ $\text{receive}(Q, m)$

gdzie: m jest przesyłanym *komunikatem* (wiadomością, ang. message),
 P - jest *odbiorcą* komunikatu,
 Q - jest *nadawcą* komunikatu.



Łącza⁽¹⁾

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. **łączą niezawodne** (ang. reliable, lossless, duplicate free, error free, uncorrupted, no spurious).

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**.

Określanie nadawców i odbiorców

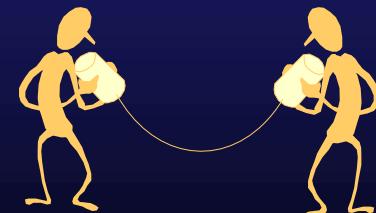
komunikacja bezpośrednią

Procesy mogą komunikować się bezpośrednio lub pośrednio. W komunikacji bezpośrednią 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:

- $\text{send}(P, m)$ – nadaj komunikat m do procesu P
- $\text{receive}(Q, m)$ – odbierz komunikat od procesu Q

Łącza komunikacyjne mają tu następujące własności:

- ❖ ustawiane są automatycznie między parą procesów, które mają komunikować się;
- ❖ dotyczą dokładnie dwóch procesów;
- ❖ są dwukierunkowe.



Przedstawiony schemat charakteryzuje się symetrią adresowania

Asymetria adresowania

Istnieje też asymetryczny wariant adresowania, w którym nadawca nazywa odbiorcę, a od odbiorcy nie wymaga się znajomości nadawcy. W tym wypadku operacje *send* i *receive* są zdefiniowane następująco:

- ❖ $\text{send}(P, m)$ – nadaj komunikat m do procesu P ;
- ❖ $\text{receive}(id, m)$ – odbierz komunikat od dowolnego procesu; pod id zostanie podstawiona nazwa procesu, od którego nadszedł komunikat.



Określanie nadawców i odbiorców komunikacja pośrednia

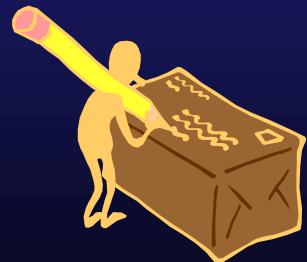
W komunikacji pośredniej komunikaty są nadawane i odbierane poprzez skrzyni pocztowe (nazywane też portami, ang. mailbox).

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*

Łącza komunikacyjne mają tu następujące własności:

- ❖ 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.



Skrzynka pocztowa

Skrzynka może być własnością procesu lub systemu. Jeżeli skrzynka należy do procesu (tzn. jest przypisana lub zdefiniowana jako część 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_poczta*. 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.



Operacje synchroniczne i asynchroniczne

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;

- ❖ **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 komunikacji *synchronicznej*, nadawca i odbiorca są blokowani aż odpowiedni odbiorca odczyta przeslaną do niego wiadomość (ang. rendez-vous). W przypadku komunikacji *asynchronicznej*, nadawca lub odbiorca komunikuje się w sposób nieblokowany.



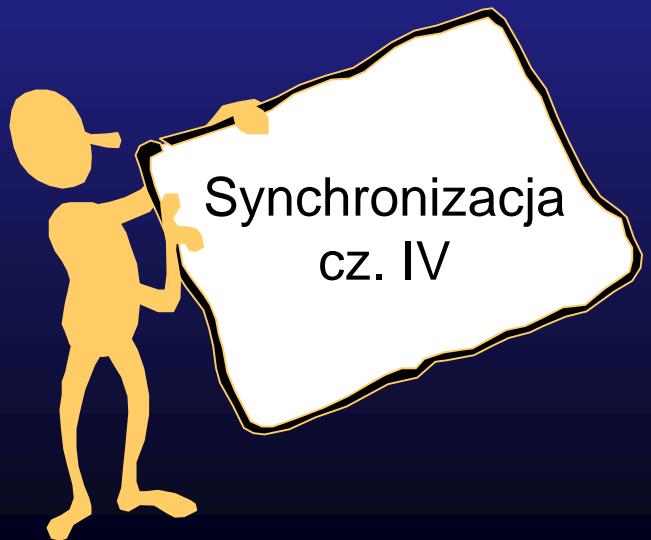
Producent – Konsument

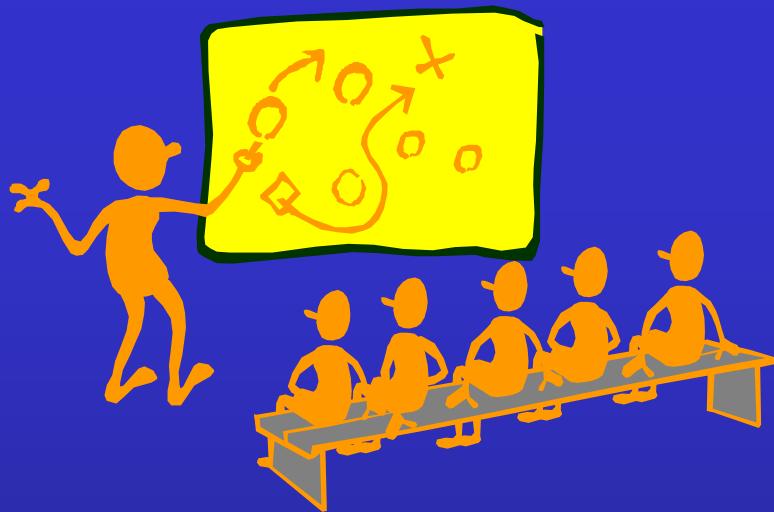
```
1. program PRODUCERCONSUMER_MESSAGETRANSMISSION;
2. var bufferPool: array[0..x] of BUFFER;
3. procedure PRODUCER;
4. begin
5.   while True do
6.     begin
7.       produceNextMessage;
8.       receive(producer, empty); /* odbiór blokowany */
9.       addMessageToCommonBuffer;
10.      send(consumer, empty); /* wysłanie asynchroniczne */
11.      end;
12.    end;
13. procedure CONSUMER;
14. begin
15.   while True do
16.     begin
17.       receive(consumer, empty);
18.       takeMessageFromCommonBuffer;
19.       send(producer, empty);
20.       processMessage;
21.     end;
22.   end;
23.   begin
24.     I:=N;
25.     while I>0 do
26.       begin
27.         send(producer, empty);
28.         I:=I-1;
29.       end;
30.     parbegin
31.       PRODUCER;
32.       CONSUMER;
33.     parend
34.   end.
```



Koniec

*... kolejne spotkanie,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*





Zakleszczenie

Zakleszczenie

Rozważmy system składający się z n procesów (zadań) P_1, P_2, \dots, P_n współdzielący s zasobów **nieprzywłasczalnych** tzn. zasobów, których zwolnienie może nastąpić jedynie z inicjatywy zadania dysponującego zasobem. Każdy zasób k składa się z m_k jednostek dla $k = 1, 2, \dots, s$. 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.

W każdej chwili zadanie P_j jest scharakteryzowane przez:

- **wektor maksymalnych żądań** (ang. claims),

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

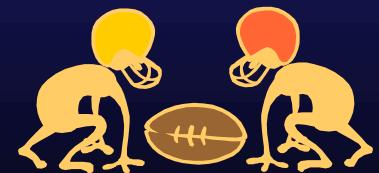
oznaczający maksymalne żądanie zasobowe zadania P_j w dowolnej chwili czasu

- **wektor aktualnego przydziału** (ang. current allocations),

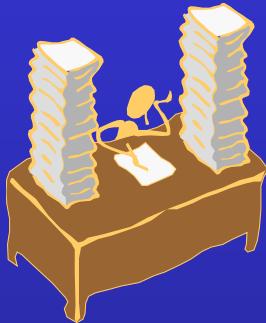
$$A(P_j) = [A_1(P_j), A_2(P_j), \dots, A_s(P_j)]^T$$

- **wektor rang** zdefiniowany jako różnica między wektorami C i A ,

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



Zakleszczenie (2)



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

$$f = [f_1, f_2, \dots, f_s]^T$$

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

Typy żądań

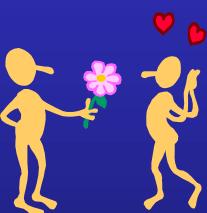
Wyróżniamy dwa typy żądań, które mogą być wygenerowane przez każde zadanie P_j

- ❖ **żądanie przydziału dodatkowych zasobów** (ang. request for resource allocation),

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

gdzie

$\rho_k^a(P_j)$ jest liczbą jednostek zasobu R_k żądanych dodatkowo przez P_j



- ❖ **żądanie zwolnienia zasobu** (ang. request for resource release),

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

gdzie

$\rho_k^r(P_j)$ jest liczbą jednostek zasobu R_k zwalnianych przez P_j



Zadanie przebywające w systemie

Łatwo wykazać:

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

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

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

$$\forall \underset{k}{\forall} \rho_k^a(P_j) \leq f_k \quad j = 1, 2, \dots, s$$

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 stan jest **realizowalny** jeżeli jest spełniona następująca zależność:

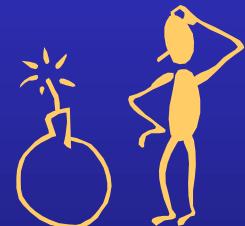
$$\sum_{j=1}^n A_k(P_j) \leq m_k \quad k = 1, 2, \dots, s$$



Stan bezpieczny

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

$$H_k(P^j) \leq f_k + \sum_{i=1}^{j-1} A_k(P^i) \quad k = 1, 2, \dots, s \\ j = 1, 2, \dots, n$$

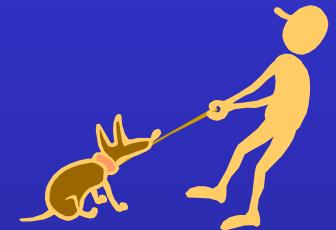


W przeciwnym razie, tzn. jeżeli sekwencja taką nie istnieje, stan jest nazywany ***stanem niebezpiecznym***. Innymi słowy, stan jest bezpieczny jeżeli istnieje takie uporządkowanie wykonywania zadań, że wszystkie zadania przebywające w systemie zostaną zakończone. Powiemy, że ***tranzycja*** stanu systemu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

Zakleszczenie – definicja

Przez **zakleszczenie** (ang. deadlock) rozumieć będziemy formalnie stan systemu, w którym spełniany jest następujący warunek:

$$\exists_{\Omega \neq \emptyset} \forall_{j \in \Omega} \exists_k \rho_k^a(P_j) > f_k + \sum_{i \notin \Omega} A_k(P_i)$$



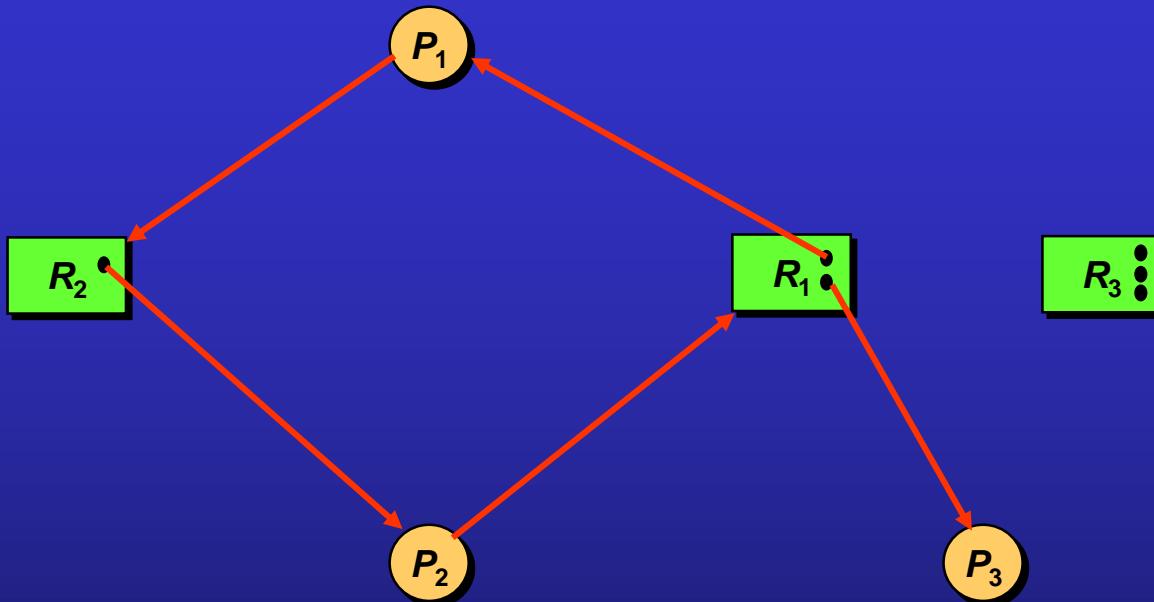
gdzie Ω jest zbiorem indeksów (lub zbiorem zadań)

Mówimy, że system jest w **stanie zakleszczenia** (w systemie wystąpił **stan zakleszczenia**), jeżeli istnieje niepusty zbiór Ω zadań, które żądają przydziału dodatkowych zasobów nieprzywłaszcjalnych 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 nieprzywłaszcjalnych 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**.

Graf alokacji zasobów



Legenda:



proces P_i



zasób R_j posiadający 3 jednostki w systemie



proces P_i posiadający jednostkę zasobu R_j

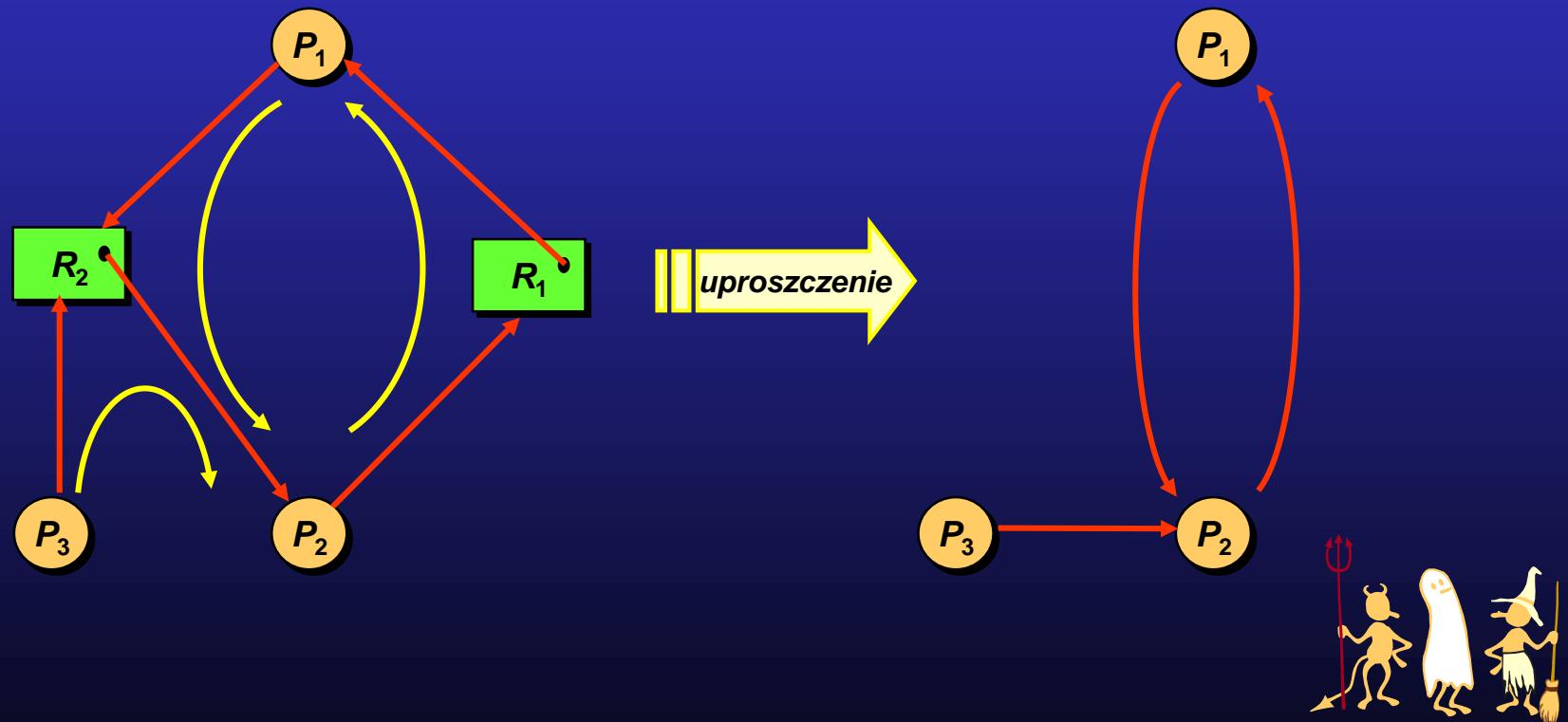


proces P_i żądający jednostki zasobu R_j



Grafy oczekiwania

Z grafu alokacji zasobów można uzyskać graf uproszczony przez usunięcie węzłów zasobowych i złączenie odpowiednich krawędzi. To uproszczenie wynika z obserwacji, że zasób może być jednoznacznie identyfikowany przez bieżącego właściciela. Ten uproszczony graf jest nazywany **grafem oczekiwania** (ang. **wait-for-graph**).



Warunki konieczne wystąpienia zakleszczenia

Warunkami koniecznymi wystąpienia zakleszczenia są:

1. **Wzajemne wykluczanie** (ang. mutual exclusion 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. **Nieprzywłaszcjalność** (ang. non preemption condition),

Zasoby są nieprzywłaszcjalne tzn. ich zwolnienie może być zainicjowane jedynie przez proces dysponujący w danej chwili zasobem.

4. **Istnienie cyklu oczekiwania** (ang. circular wait condition),

Występuje pewien cykl procesów z których każdy ubiega się o przydział dodatkowych zasobów będących w dyspozycji kolejnego procesu w cyklu.





Przeciwdziałanie zakleszczeniom

✓ *Konstrukcje systemów wolnych od zakleszczenia*

(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.

✓ *Detekcja zakleszczenia i odtwarzanie stanu wolnego od zakleszczenia* (ang. detection and recovery).

W podejściu detekcji i odtwarzania, stan systemu jest periodycznie sprawdzany i jeśli wykryty zostanie stan zakleszczenia, system podejmuje specjalne akcje w celu odtworzenia stanu wolnego do zakleszczenia.

✓ *Unikanie zakleszczenia* (ang. avoidance).

W podejściu tym zakłada się znajomość maksymalnych żądań zasobowych. Każda potencjalna transakcja stanu jest sprawdzana i jeśli jej wykonanie prowadziłoby do stanu niebezpiecznego, to żądanie zasobowe nie jest w danej chwili realizowane.

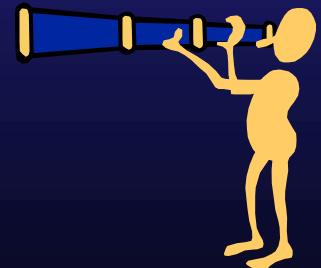
✓ *Zapobieganie zakleszczeniu* (ang. prevention)

W ogólności podejście to polega na wyeliminowaniu możliwości zajścia jednego z warunków koniecznych zakleszczenia

Detekcja zakleszczenia

Algorytm Habermana

1. Zainicjuj $D := \{1, 2, \dots, n\}$ i $f := 0$;
2. Szukaj zadania o indeksie $j \in D$ takiego, że
 $\rho^\alpha(P_j) \leq f$
3. Jeżeli zadanie takie nie istnieje, to zbiór zadań odpowiadający zbiorowi D jest zbiorem zadań zakleszczonych. Zakończ wykonywanie algorytmu.
4. W przeciwnym razie, podstaw:
$$D := D - \{j\}; \quad f := f + A(P_j)$$
5. Jeżeli $D = \emptyset$, to zakończ wykonywanie algorytmu. W przeciwnym razie przejdź do kroku 2.



Odtwarzanie stanu

Algorytm Holt'a:

```
1. begin
    2. initialize:  $I_k = 1, k=1, 2, \dots, s$ ;  $c_i = s, i=1, 2, \dots, n; c_0 = n;$ 
3. LS:  $Y := \text{False};$ 
4. for  $k = 1$  step 1 until  $s$  do
5.     begin
6.         while  $E_{1,k,Ik} \leq f_k \wedge I_k \leq n$  do
7.             begin
8.                  $C_{E2,k,Ik} := C_{E2,k,Ik} - 1;$ 
9.                  $I_k := I_k - 1;$ 
10.                if  $C_{E2,k,Ik} = 0$  then
11.                    begin
12.                         $C_0 := C_0 - 1;$ 
13.                         $Y := \text{True};$ 
14.                        for  $i = 1$  step 1 until  $s$  do
15.                             $f_i := f_i + A_i(P_{E2,k,Ik});$ 
16.                        end;
17.                    end;
18.                end;
19.                if  $Y = \text{true}$   $c_0 > 0$  then go to LS;
20.                if  $Y = \text{true}$  then answer "no"
21.                else answer "yes";
22.            end.
```

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



Wady i zalety podejścia detekcji do odtwarzania stanu

- 
- Narzut wynikający z opóźnionego wykrycia stanu zakleszczenia
 - Narzut czasowy algorytmu detekcji i odtwarzania stanu
 - Utrata efektów dotychczasowego przetwarzania odrzuconego zadania.

- 
- Brak ograniczeń na współbieżność wykonywania zadań
 - Wysoki stopień wykorzystania zasobów
 - Podejście unikania



Algorytm podejścia 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 zawiesz 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

Wady i zalety podejścia unikania



- 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



- Potencjalnie wyższy stopień wykorzystania zasobów niż w podejściu zapobiegania.



Podejście zapobiegania

Rozwiązań wykluczające możliwość wystąpienia cyklu żądań.

Algorytm wstępnego przydziału

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

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



Przykładowo, proces może żądać kolejno zasobów 1, 2, 3, 6, ... , natomiast nie może żądać zasobu 3 a później 2. Jeśli więc z kontekstu programu wynika kolejność żądań inna niż narzucony porządek, to proces musi zażądać wstępnej alokacji zasobów, generując na przykład żądanie przydział zasobów 2 i 3.

Alogrytmy Wait-Die i Wound-Wait

Algorytm Wait-Die

Rozwiążanie negujące zachowywanie zasobów (ang. 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), to P_1 czeka (*wait*) na zwolnienie zasobu przez P_2 . W przeciwnym razie zadania P_1 jest w całości odrzucane (*abort*) i zwalnia wszystkie posiadane zasoby.

Algorytm Wound-Wait

Rozwiążanie dopuszczające przywłaszczałość



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 odrzucane (*abort*) i zwalnia wszystkie posiadane zasoby. W przeciwnym razie P_1 czeka (*wait*) na zwolnienie zasobu przez P_2 .

Wady i zalety podejścia zapobiegania



- Ograniczony stopień wykorzystania zasobów.



- Prostota i mały narzut czasowy.



Koniec

*... kolejne spotkanie,
w tym samym miejscu,
o tej samej porze ...
Przyjdź i zobacz !!!*

