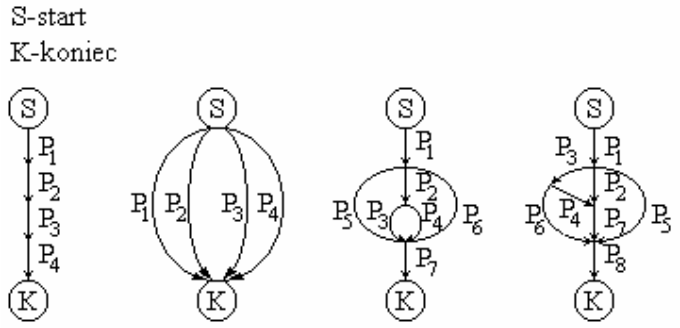
Programowanie współbieżne.

Grafy przepływu procesów.

Grafy przepływu procesów przedstawiają zależności czasowe wykonywania procesów. Węzły tych grafów reprezentują chwilę czasu, natomiast łuki - procesy. Dwa węzły są połączone łukiem jeżeli istnieje proces, którego moment rozpoczęcia odpowiada pierwszemu węzłowi czasu, a moment zakończenia - drugiemu.

Przykład :

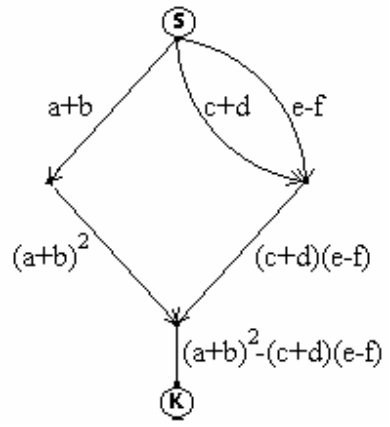


szeregowe wykonywanie procesów — równoległe wykonywanie procesów — graf szeregowo-równoległy — graf mieszany

Powyższe grafy są nazywane grafami skierowanymi, **DAG** (ang. **D**irected **A**cyclic **G**raph)

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

Przykład.
Przedstawić graf przepływu procesów odpowiadających wyznaczeniu wyrażenia $y=(a+b)^2-(c+d)(e-f)$.

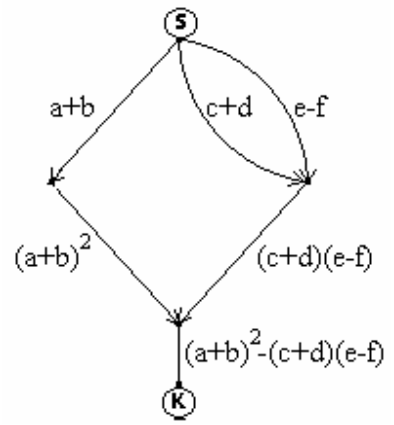


Notacja "and" (Wirth).

Współbieżne wykonanie może być specyfikowane za pomocą operatora **and**, który łączy dwa wyrażenia np.

```
   ...
begin
  x1:= x1 + 2;
  y1:= x1 + y1
end
and
x2:= 2 x2 + y2;
y2:= x1 + x2 + y1 + y2;
   ...
```

Zastosujemy notację **and** do implementacji programu wyznaczającego wartość wyrażenia $(a+b)^2-(c+d)(e-f)$.

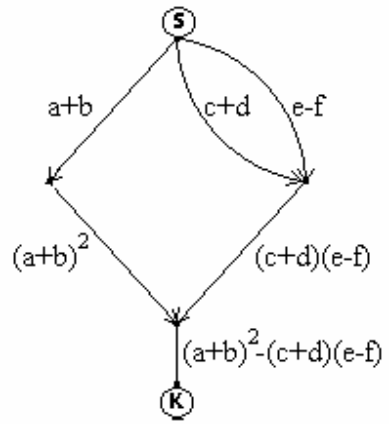



```
begin
  begin
    x1:=a+b;
    x2:=x1* x1
  end
  and
  begin
     x3:=c+d
     and
     x4:=e-f;
     x5:=x3* x4
  end;
  x6:=x2-x5
end.
```

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

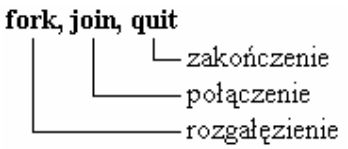
Wszystkie wyrażenia ujęte w nawiasy **parbegin**, **parend** są wykonywane współbieżnie.

Zastosujemy notację **parbegin** i **parend.**




```
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).



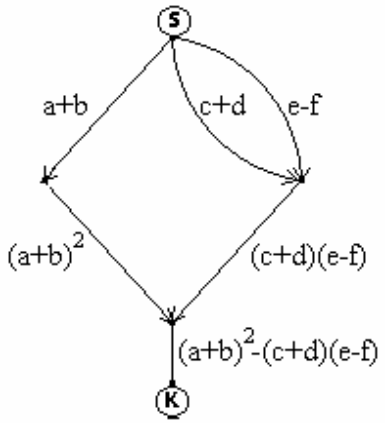
Instrukcja **quit** powoduje zakończenie procesu.
Instrukcja **fork w** oznacza, że proces w którym wystąpiła ta instrukcja będzie dalej wykonywany współbieżnie z procesem identyfikowanym przez etykietę **w**, np. zapis.
Składnia **join**: **join** t,w
t - licznik
w - etykieta
Wykonanie instrukcji **join** t,w oznacza:

...
t:= t-1;
if t= 0 **then goto** w;
...

przy czym sekwencja tych dwóch instrukcji jest wykonywana **atomowo**, tzn. że jest **niepodzielna**. Instrukcja jest zatem wykonana w całości albo wcale.
Przykład :

```
begin
 t1:=2;  // tyle , ile procesów
 t2:=2;  // „ schodzi się ″ w węzłach
 fork w1; // rozgałęzienia
 fork w2;
 fork w3;
 quit;
w1:  x1:= a + b;
      x2:= x1 * x1;
     join t1,w6;
     quit;
w2:  x3:= c + d;
     join t2,w5;
     quit;
w3:  x4:= e - f;
     join t2,w5;
     quit;
w5:  x5:= x3 * x4;
     join t1,w6;
     quit;
w6:  x6:= x2 - x5;
     quit;

end.
```



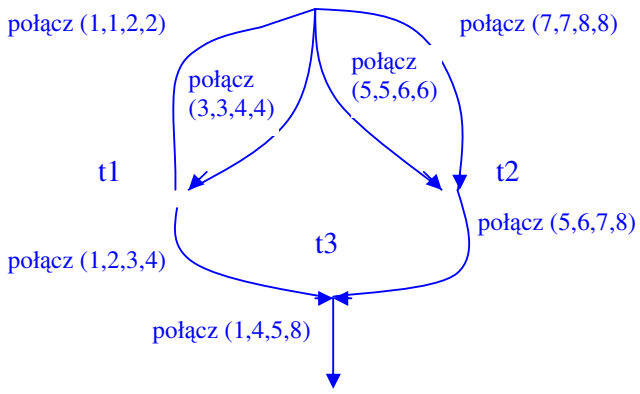
Inny przykład – sortowanie przez scalanie

połącz(x1,x2,y1,y2) – łączy 2 ciągi uporządkowane – x i y

Jeśli sortujemy np. 8 elementów – procedura połącz jest wywoływana w następujący sposób :

połącz(1,1,2,2)
połącz(3,3,4,4)
połącz(5,5,6,6)
połącz(7,7,8,8)
połącz(1,2,3,4)
połącz(5,6,7,8)
połącz(1,4,5,8)

Graf przepływu procesów ma następującą postać :



Zapis w notacji **and** :

```
begin

  begin
    połącz(1,1,2,2)
    and
    połącz(3,3,4,4)
    połącz(1,2,3,4)
  end
  and
  begin
    połącz(5,5,6,6)
    and
    połącz(7,7,8,8)
    połącz(5,6,7,8)
  end
  połącz (1,4,5,8)

end.
```

Zapis w notacji **parbegin** :

```
begin

  parbegin
    begin
      parbegin
        połącz(1,1,2,2)
        połącz(3,3,4,4)
      parend
      połącz(1,2,3,4)
    end
    begin
      parbegin
        połącz(5,5,6,6)
        połącz(7,7,8,8)
      parend
      połącz(5,6,7,8)
    end
  parend
  połącz (1,4,5,8)

end.
```

Zapis w notacji **fork – join – quit** :

```
begin
  t1=t2=t3=2;
  fork a1;
  fork a2;
  fork a3;
  fork a4;
  quit;
a1: połącz(1,1,2,2);
    join t1,a5;
    quit;
a2: połącz(3,3,4,4);
    join t1,a5;
    quit;
a3: połącz(5,5,6,6);
    join t2,a6;
    quit;
a4: połącz(7,7,8,8);
    join t2,a6;
    quit;
a5: połącz(1,2,3,4);
    join t3,a7;
    quit;
a6: połącz(5,6,7,8);
    join t3,a7;
    quit;
a7: połącz(1,4,5,8);
    quit;

end.
```

Problem wzajemnego wykluczania.

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. Przyjmijmy ponadto, że procesy te mają dostęp do wspólnych zasobów. Rozważmy dla przykładu dwa procesy:

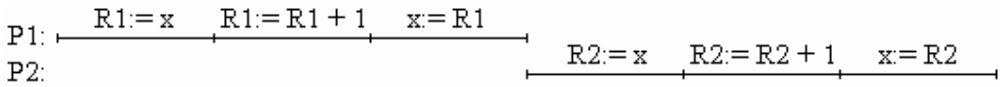
P1: x:= x + 1;
P2: x:= x + 1;

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

1° R:= x ; /* pobranie wartości zmiennej x do rejestru wewnętrznego procesora */
2° R:= R + 1; /* inkrementacja zawartości rejestru wewnętrznego procesora */
3° x:= R; /* zapisanie zawartości rejestru do zmiennej x */

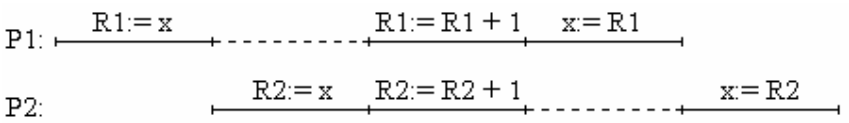
Rozważmy możliwe sekwencje wykonywania takich współbieżnych procesów.

a)



OK. – otrzymujemy x = x + 2

b)



Źle – otrzymujemy x = x + 1

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 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° 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.

Rozwiązanie programowe problemu wzajemnego wykluczania.

```
program versionone;
var processnumber: integer;

procedure processone;
begin
  while true do
  begin
    while processnumber = 2 do;
    criticalsectionone;
    processnumber:= 2;
    otherstuffone
  end
 end;

procedure processtwo;
begin
  while true do
  begin
    while processnumber = 1 do;
    criticalsectiontwo;
    processnumber:= 1;
    otherstufftwo
  end
end;

begin
  processnumber:= 1;
  parbegin
    processone;
    processtwo
  parend
end.
```

Problemy :

- rozwiązanie gwarantuje wzajemne wykluczanie , ale procesy mogą wykonywać swoje sekcje krytyczne tylko naprzemiennie i jeśli np. processnumber = 1 , to proces 2 nie będzie mógł wejść do sekcji krytycznej , mimo , że w tym czasie proces 1 wcale nie musi być w sekcji tylko np. wykonywać otherstuffa – proces 2 mimo to będzie czekać
- jeśli proces 1 wyjdzie z sekcji krytycznej i przydzieli prawo wejścia procesowi 2 , a proces 2 się skończy nie przydzieliwszy go z powrotem procesowi 1 (nie wejdzie już do sekcji krytycznej po przydzieleniu mu prawa – będzie wykonywał otherstuff) , to jeżeli proces 1 będzie chciał wejść do sekcji krytycznej , będzie czekał w nieskończoność na otrzymanie prawa wejścia od procesu 2

```
program versiontwo;
var P1inside, P2inside: boolean;

procedure processone;
begin
  while true do
  begin
    while P2inside do;
    P1inside:= true;
    criticalsectionone;
    P1inside:= false;
    otherstuffone
  end
end;

procedure processtwo;
begin
  while true do
  begin
    while P1inside do;
    P2inside:= true;
    criticalsectiontwo;
    P2inside:= false;
   otherstufftwo
  end
end;

begin
  P1inside:= false;
  P2inside:= false;
  parbegin
    processone;
    processtwo
  parend
end.
```

Problemy :

- oba procesy mogą naraz wejść do sekcji krytycznej (choćby na samym początku – obie flagi ustawione są na false i jeżeli np. zdarzy się następująca sekwencja instrukcji :
  - P1 rozpocznie pętlę **while**
  - P2 rozpocznie pętlę **while**

oba naraz wejdą do sekcji krytycznej) . Rozwiązanie uzależnione od dokładnych przebiegów czasowych obu procesów .

```
program versionthree;
var P1wantstoenter, P2wantstoenter: boolean;

procedure processone;
begin
  while true do
  begin
    P1wantstoenter:= true; // sygnalizuje gotowość do wejścia
    while P2wantstoenter do; // sprawdzanie , czy drugi też nie jest gotowy do wejścia
    criticalsectionone;
    P1wantstoenter:= false; // teraz – jeśli drugi czeka – to może wejść
    otherstuffone
  end
end;

procedure processtwo;
begin
  while true do
  begin
    P2wantstoenter:= true;
    while P1wantstoenter do;
    criticalsectiontwo;
    P2wantstoenter:= false;
    otherstufftwo
  end
end;

begin
  P1wantstoenter:= false;
  P2wantstoenter:= false;
  parbegin
    processone;
    processtwo
  parend
end.
```

Właściwie inna wersja rozwiązania poprzedniego – różni się tylko zamianą kolejności występowania instrukcji sprawdzania wartości zmiennej oraz instrukcji przypisania . Również zależy od przebiegów czasowych – jeżeli zdarzy się następująca sekwencja instrukcji :

- P1 ustala P1wantstoenter na true
- P2 ustala P2wantstoenter na true

W takim wypadku pętla **while** w obu procesach będzie

```
program versionfour;
var P1wantstoenter, P2wantstoenter: boolean;

procedure processone;
begin
  while true do
  begin
    P1wantstoenter:= true;   // pierwszy chce wejśc
    while P2wantstoenter do // jeśli także drugi chce wejść – czekanie w pętli
    begin
      P1wantstoenter:=  false;     // w takim razie pierwszy ustępuje
      delay (random, freecycles); // pierwszy odczekuje jakiś czas
      P1wantstoenter:= true       // i znowu chce wejść , będzie mógł wejśc , jeśli drugi w
    end;                          // tym czasie wyjdzie z sekcji krytycznej
    criticalsectionone;
    P1wantstoenter:= false; // teraz drugi może wejść
    otherstuffone
  end
end;

procedure processtwo;
begin
  while true do
  begin
    P2wantstoenter:= true;
    while P1wantstoenter do
    begin
      P2wantstoenter:=  false;
      delay (random, freecycles);
      P2wantstoenter:= true
    end;
    criticalsectiontwo;
    P2wantstoenter:= false;
    otherstufftwo
  end
end;

begin
  P1wantstoenter:= false;
  P2wantstoenter:= false;
  parbegin
    processone;
    processtwo
  parend
end.
```

Algorytm zasadniczo poprawny , może się jednak zdarzyć taka (mało prawdopodobna) sytuacja , że P1 będzie czekał na wejście w pętli while (P2 będzie w sekcji krytycznej) , wykona delay , w tym czasie P2 wyjdzie z sekcji krytycznej , ale znowu zdąży wejść (bo P1 wantstoenter = false w czasie , gdy P1 czeka) , zanim P1 odczeka i P1 się nie doczeka .

Ponadto , jeśli dojdzie do tego , że P1wantstoenter = true i jednocześnie P2wantstoenter = true , to jeżeli oba odczekają w delay odpowiednią ilość czasu , może dojść do tego , że znowu P1wantstoenter = true i jednocześnie P2wantstoenter = true itd. co jest oczywiście również b. mało prawdopodobne

```
program dekkersalgorithm;
var favoredprocess (first, second);
var P1wantstoenter, P2wantstoenter: boolean;
procedure processone;
begin
  while true do
  begin
    P1wantstoenter:= true;  // P1 chce wejśc
     while P2wantstoenter do if favoredprocess = second then
     begin  // pętla tak długo , jak P2 chce wejść i P2 jest uprzywilejowany
       P1wantstoenter:=  false;  // P1 rezygnuje
       while favoredprocess = second do;  // czeka tak długo , aż P2 jest faworyzowany
       P1wantstoenter:= true  // jeśli już nie jest ( wyszedł z sekcji ) – P1 znów chce wejść
     end;
     criticalsectionone;
     favoredprocess := second;  // teraz rezygnuje
     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.
```

Algorytm poprawny – ani się nie zapętli , ani oba procesy nie będą naraz w sekcji kryt. Z warunku pętli wynika , że P_i może wejść wtedy , jeśli favoredprocess=i albo Pjwantstoenter=false . Tak więc , aby oba procesy weszły naraz do sekcji , musiałoby zajść jednocześnie :

1. favoredprocess=i=j – niemożliwe jednocześnie lub :
2. Piwantstoenter ,Pjwantstoenter=false – ale jeśli proces ma być w sekcji , to jego wantstoenter jest true
3. favoredprocess=i , ale Piwantstoenter=false . Ale jeśli Piwantstoenter=false , to oznacza że favoredprocess=j i Piwantstoenter zmieni się na true dopiero jeśli favoredprocess zmieni się na i.

4. favoredprocess=j i Pjwantstoenter=false. Analogicznie

Ponadto , jeśli jeden z procesów np. P_i jest w sekcji , to Piwantstoenter=true i favoredprocess =i , więc P_j nie wejdzie .

Jedyne miejsce , gdzie może się zapętlić – pętla while trwa np.dla P_i tak długo , aż nie zajdzie favoredprocess=i lub też nie będzie Pjwantstoenter=false . Jeśli P_j nie jest gotowy , to favoredprocess=i oraz Pjwantstoenter=false i P_i wejdzie . Jeśli oba gotowe i czekają na wejście , to oba wantstoenter=true , ale favoredprocess przyjmuje 1 wartość i któryś wejdzie . Jeżeli jakiś proces np. P_i jest w sekcji , to po wyjściu zmieni Piwantstoenter na false i favoredprocess na j , więc P_j będzie mógł wejść .

Algorytm Petersona dla 2 procesów

```
program Peterson_algorithm;
begin
shared
    flag[0..1]: Boolean ; /* initially false */
    turn: integer ; /* initially 0 or 1 */
local
    other_i: Boolean;
    whose_i: integer;

while true do
begin
 flag[i] := true;  // P1 gotowy do wejścia
 turn := 1–i; // zakłada , że P2 też chce
 repeat
   whose_i := turn;      // spr. czy P2 chce
   other_i := flag[1–i]; // spr. czy P2 gotowy
 until (whose_i = i or not other_i); // aż P2 nie zmieni turn na i lub też flag[1-i] na false

 critical section;

 flag[i] := false  ;  /* exit section */

 remainder section;

end

end.
```

Algorytm poprawny - nie spowoduje ani zapętlenia ani równoczesnego wejścia do sekcji dwóch procesów . Jedyne miejsce , gdzie mogłoby się zapętlić – instrukcja **repeat** – wykonywana dla procesu P_i tak długo , aż nie zajdzie *turn* = *i* lub *flag*[*j*] = *false* . Jeśli P_j nie jest gotowy do wejścia do sekcji , to *flag*[*j*] = *false* i do sekcji może wejść P_i . Jeśli P_j spowodował , że *flag*[*j*] = *true* oraz też wykonuje pętlę , to jeśli *turn* = *i* , to P_i wejdzie do sekcji , a jeśli *turn* = *j* , to P_j wejdzie . Jednak , kiedy P_j wyjdzie , to zmieni *flag*[*j*] na *false* i P_i będzie mógł wejść . Jeśli P_j zmieni *flag*[*j*] na *true* , to musi także zmienić turn na i ,a w tej sytuacji P_i który oczekując na wejście w pętli nie zmienia wartości turn wejdzie do sekcji .

Każdy proces P_i wchodzi do sekcji krytycznej tylko wtedy , gdy albo *flag*[*j*] = *false* , albo *turn* = *i* . Poza tym , gdyby oba procesy miały jednocześnie być w sekcji , to spełnione byłoby *flag*[0] = *flag*[1] = *true* - każdy P_i przypisuje *flag*[*i*] = *true* przed swoim wejściem do sekcji . W takim razie , aby oba były jednocześnie w sekcji , musiałoby jednocześnie zachodzić *turn* = *i* = j , co nie jest możliwe . Ponadto podczas gdy P_j jest w sekcji , to *flag*[*j*] = *true* i jeśli P_i będzie chciał wejść do sekcji , to zanim wykona pętlę **repeat** , przypisze *turn* = *j* i będzie musiał czekać w pętli (bo $whose_i$ = j i $other_i$ = true) aż P_j wyjdzie . Tak więc zawsze tylko 1 będzie w sekcji krytycznej .

Algorytm Petersona dla *n* procesów

```
program Peterson _n_algorithm;
begin
shared
   flag[1..n]: integer; /* initially 0 */ // do którego cyklu pętli for doszedł każdy z procesów
   turn[1..n–1]: integer; /* initially arbitrary */ // kto ostatni doszedł do danego cyklu pętli for
local
   k, l, other_i, whose_i: integer;
while true do
begin
 for k := 1 to n–1 do
 begin
  flag[i] := k; // proces i-ty jest w k-tym przebiegu pętli for
  turn[k] := i; // do k-tego przebiegu wszedł jako ostatni proces i-ty
  repeat
    whose_i := turn[k] ; // jeśli jakiś proces wszedł po naszym – można przejść dalej
    if whose_i ≠ i then break /* continue the for loop for the next value of k */
    for l := 1 to n do
    begin
      if l ≠ i then  // sprawdzanie flag wszystkich innych procesów
      other_i := flag[l];
      if other_i ≥ k then break // jeśli jakiś zaszedł dalej – repeat od nowa
    end ;
  until other_i < k;
  /* the repeat-until loop continues till turn[k]≠i or ∀l=1..n,l≠i: flag[l]<k */
  czyli do czasu , aż albo inny proces wszedł do tego samego cyklu po P_i albo wszystkie są
we
  wcześniejszych cyklach niż P_i

   critical section;

   flag[i] := 0 ;   /* exit section */

   remainder section;

end

end.
```

Jeśli proces jest sam na jakimś etapie pętli for , to przejdzie dalej , jeśli wszystkie inne są na wcześniejszych . Jeśli jest ich więcej , powiedzmy x , to dalej przejdzie ich x-1 i zostanie ten który jako ostatni wszedł do tego etapu (niezależnie od tego , czy jakieś procesy są na dalszych etapach , czy nie) . Tak więc jeżeli w 1 etapie (czyli dla k = 1)mamy n procesów , to do
(n-1)-tego dojdzie max. 2 , a do sekcji max. 1.
„Najbardziej wysunięty” proces (o max. fladze) będzie zawsze szedł dalej , ponadto jeżeli na jakimś etapie jest więcej niż 1 proces , to wszystkie one z wyjątkiem 1 przejdą zawsze dalej więc algorytm się nigdy nie zapętli .

Algorytm Dijkstry dla *n* procesów

```
program Dijkstra_algorithm;

begin
shared
   flag[1..n]: 0..2 ; /* initially 0 */ // 0 – nie chce wejść , 1 – chce , 2 – został wybrany
   turn: 1..n ; /* initially arbitrary */
local
   test_i: 0..2;
   k, other_i, temp_i: 1..n;

while true do
begin
   L: flag[i]):=1; // chce zostać wybrany
   other_i:= turn;
   while other_i ≠ i do // tak długo jak P_i nie jest wybrany
   begin
     test_i:= flag[other_i] ; // sprawdzanie flagi wybranego
     if test_i = 0 then turn := i;  // jeśli wybrany nie chce – bo wyszedł z sekcji to nasz P_i staje
     other_i := turn                // się wybrany - ale może to zrobić jednocześniewiele procesów
   end ;
   flag[i] := 2;  // P_i ustawia się na wybranego
   for k := 1 to n do if k ≠ i then  // spr. czy inne procesy nie są wybrane
   begin
     test_i := flag[k] ;
     if test_i = 2 then  goto L  // jeśli tak to P_i rezygnuje
   end;

   critical section;

   flag[i] := 0; /*  exit section  */  // teraz już nie chce

   remainder section;
end

end.
```

1 część – czekanie aż proces nie zostanie wybrany

2 część – spr. czy inne proc. też nie zostały wybr. -mogło dojść do tego równocz. W czasie wyk. 2 cz. inny proc. nie może już być wybr. bo flag[turn] = 2

Nigdy 2 procesy nie wejdą do sekcji naraz – jeśli miałoby się tak stać , dla 2 proc. naraz musiałoby być flag = 2 , co jest niemożliwe ,bo nawet jeśli w części 1 zostanie wybrany więcej niż 1 proces , to tylko 1 z nich przejdzie przez część 2 .

Ponadto jeśli jeden proces jest w sekcji , to jego flaga = 2 a więc inne zatrzymają się w cz. 1 lub 2

Jedyne miejsce , gdzie mogłoby się zapętlić – pętla while , trwa ona tak długo , aż wreszcie któryś proces odkryje , że ten , który był w sekcji , wyszedł i ustawił swoją flagę na 0 . Tak więc jeżeli jakiś wyjdzie , to inny przestanie czekać (choć może dojść również do tego , że proces , który wyjdzie z sekcji , zmieni sobie flagę na 0 , wykona resztę i zdąży znów zmienić flagę na 1 zanim inne to zauważą – ale wtedy on wejdzie do sekcji , bo
jest nadal wybranym – czyli i tak nie dojdzie do zapętlenia) .

Za pierwszym razem wchodzi ten , na który

Algorytm Lamporta dla *n* procesów

```
program Lamport_algoprithm;
begin
shared
    choosing[1..n]: 0..1 ; /* initially 0 */
    num[1..n]: integer ; /* initially 0 or 1 */
local
    test_i: 0..1
    k, mine_i, other_i, temp_i: integer

while true do  // proces P_i
begin
  choosing[i] :=1; // w trakcie wybierania swojego numerka
  for k := 1 to n do if k ≠ i then /* mine_i := max(num[k] | k ≠ i) */
  begin
    temp_i := num[k] ;
    mine_i := max(mine_i, temp_i)
  end;
  mine_i := mine_i + 1;
  num[i] := mine_i; // przyznaje sobie najwyższy numerek z wszystkich
  choosing[i] := 0; // skończył wybieranie
  for k := 1 to n do if k ≠ i then // dla wszystkich innych procesów
  begin
    repeat
      test_i := choosing[k] // tak długo aż jakiś proces nie wybrał
    until test_i = 0;
    repeat  // jeżeli już wybrał
      other_i := num[k] // tak długo aż nasz numerek nie jest mniejszy albo ktoś wyszedł z sekcji
    until other_i = 0 or (mine_i , i) < (other_i , k); // mniejszy w tym sensie , że (mine_i < other_i) lub
  end;                                                 // (mine_i = other_i ) i (i < k)

  critical section;

  num[i] := 0 ;          /*  exit section  */

  remainder section;

end

end.
```

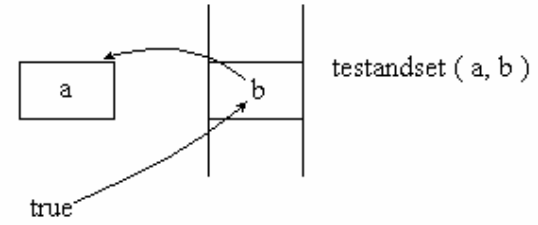
Algorytm ”piekarski” – każdy proces przed wejściem dostaje numerek . Obsługę rozpoczyna się od klienta z najmniejszym numerem . Jeżeli P_i i P_j mają ten sam numer pierwszy będzie obsłużony ten o wcześniejszej nazwie – nazwy procesów są jednoznaczne i całkowicie uporządkowane . Jeżeli proces P_i jest w sekcji , to $\forall k \neq i$: jeśli P_k ma już wybrany swój numer to (num[i],i)<(num[k],k) – wynika to z warunku drugiej pętli repeat – przepuści ona do sekcji proces o najmniejszym num[i]
z istniejących , a kolejne zgłaszające żadanie wejścia do sekcji będą miały coraz wyższe numery .

Algorytm zapewnia wzajemne wykluczanie – jeśli proces P_i jest w sekcji , a inny np. P_k będzie chciał wejść , to odkryje , że num[i]≠0 oraz (num[i],i) < (num[k],k) . Będzie zatem czekał w pętli repeat na wyjście P_i z sekcji .
Nie zapętli się , bo zawsze jakiś proces ma min. numerek i on wejdzie do sekcji .Potem wejdzie kolejny , który ma najmniejszy
numerek z pozostałych itd .

Rozwiązanie problemu wzajemnego wykluczania z użyciem mechanizmów sprzętowych.

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**.



Rozwiązanie skalowalne (dla n procesów) , ale problem w systemach wieloprocesorowych – tylko dla 1 procesorowych testandset jest atomowa

```
program testandsetexample;
var active: boolean;

procedure processone;
var onecannotenter: boolean;
begin
  while true do
  begin
    onecannotenter:= true; // zakładamy na początku , że nie może wejść
     while onecannotenter do testandset (onecannotenter, active); // jeśli onecannotenter
     prawdziwe tak długo , jak active = true – czekanie na prawo do wejścia
     criticalsectionone;
     active:= false; // teraz inny proces może wejść
     otherstuffone
   end
end;

procedure processtwo;
var twocannotenter: boolean;
begin
   while true do
   begin
      twocannotenter:= true;
      while twocannotenter do testandset (twocannotenter, active);
      criticalsectiontwo;
      active:= false;
      otherstufftwo
   end
end;

begin
  active:= false;
  parbegin
    processone;
    processtwo
  parend
end.
```

Systemowe rozwiązania problemu wzajemnego wykluczania.

Semafory.

Semaforem nazywamy zmienną chronioną, na ogół 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:

a) binarne - przyjmujące tylko wartość 0 lub 1,
b) ogólne (licznikowe) - mogą przyjąć nieujemną wartość całkowitoliczbową.

Operacje P i V (Dijkstra)

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

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

if S > 0 then S:= S – 1 else (wait on S) Stosowane w trial section ; proces , który wywołał tą instrukcję jest zawieszany (włączany do zbioru zadań skojarzonych z tym semaforem)

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 // jeśli zbiór procesów semafora S jest pusty – zwiększanie S

Z semaforem skojarzony jest zbiór procesów , które mogą być do tego zbioru dołączane / odłączane .
Rozwiązanie w pełni skalowalne dla n procesów .

Zastosowanie operacji P i V do wzajemnego wykluczania. Rozwiązanie semaforowe problemu sekcji krytycznej.

```
program semaphoreexample;
var active: semaphore;

procedure processone;
begin
  while true do
  begin
    P(active);
    criticalsectionone;
    V(active);
    otherstuffone
  end
end;
  .
  .
  .


begin
  semaphoreinitialize(active, 1);
  parbegin
    processone;
    .
    .
    .
    process n
  parend
end.
```

Na początku ustawiane jest active=1 . Pierwszy proces przed wejściem do sekcji krytycznej dokonuje P(active) , co powoduje wyzerowanie active - tak więc następne procesy będą czekały . Proces ten wchodzi do sekcji krytycznej i wychodząc wykonuje V(active) – jeśli jakieś procesy czekają to jeden z nich wejdzie do sekcji krytycznej (i active będzie nadal wynosić 0 , co uniemożliwi wejście innym procesom) , w innym wypadku active jest zwiększane o 1 (teraz wynosi więc 1) i jeżeli jakiś proces będzie chciał wejść do sekcji krytycznej , to wyzeruje active itd.

Problem producenta-konsumenta.
(przy założeniu , że zapis i odczyt z bufora nie mogą być równoczesne –
konieczna jest sekcja krytyczna)

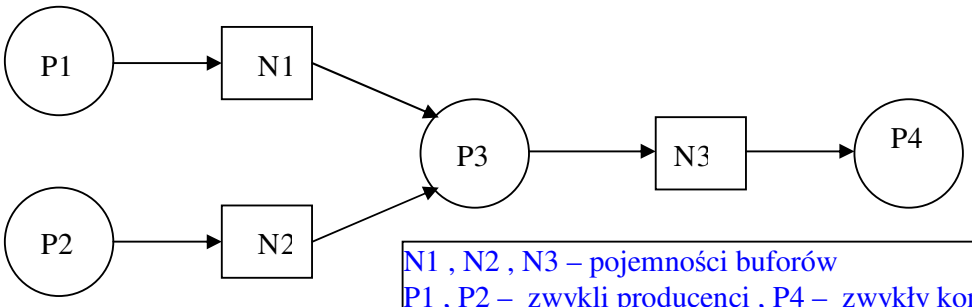
```
program producerconsumer;
var emptybuffers, fullbuffers, active: semaphore;

procedure producer;
begin
  while true do
  begin
    producenextrecord;
    P(emptybuffers);  // jeśli nie ma pustych – niech czeka ; jeśli są – zmniejszenie ich liczby
o 1
    P(active);          //  żeby w tym samym czasie było tylko dodawanie lub pobieranie
    addtobuffer;        // z bufora a nie oba naraz
    V(active);
    V(fullbuffers)      // jeśli konsument czeka na zapełnienie – może już pobrać i działać
dalej
  end                   //  w innym wypadku – zwiększenie liczby zapełnionych o 1
end;

procedure consumer;
begin
  while true do
  begin
    P(fullbuffers);     // jeśli nie ma pełnych – niech czeka ; jeśli są – zmniejszenie ich liczby o
1
    P(active);
    takefrombuffer;
    V(active);
    V(emptybuffers);   // jeśli producent czeka na opróżnienie – może już dodać i działać dalej
    processnextrecord // w innym wypadku – zwiększenie liczby pustych o 1
  end
end;

begin
  semaphoreinitialize(active, 1);
  semaphoreinitialize(emptybuffers, N);  // początkowo N pustych buforów
  semaphoreinitialize(fullbuffers, 0);       // i 0 pełnych
  parbegin
    producer;
    consumer
  parend
end.
```

Inny przykład problemu p-k



N1 , N2 , N3 – pojemności buforów
P1 , P2 – zwykli producenci , P4 – zwykły konsument .
P3 – zarówno producent , jak i konsument . Aby produkować coś dla P4 , musi otrzymać coś zarówno od P1 jak i P2 .

Implementacja P3 :

```
procedure P3 ;
begin
  while true do
  begin
    parbegin // rónoległe pobieranie z buforów 1 i 2 – jeśli jeden pusty , to nie blokuje
      begin    // pobierania z drugiego
        P(fb1);
        P(active1);          //  osobna sekcja krytyczna dla każdego bufora (2 różne semafory )
        pobierz_z_buf1;  //  – operacje na jednym nie blokują operacji na innych
        V(active1);
        V(eb1);
      end;
      begin
        P(fb2);
        P(active2);
        pobierz_z_buf2;
        V(active2);
        V(eb2);
      end;
    parend;
    P(eb3);       // jeśli już pobrane po 1 elemencie z buforów 1 i 2 , to można
    P(active3);  // wysłać element do bufora 3
    wyślij_do_buf3;
    V(active3);
    V(fb3);
  end;
end;
```

Na początku programu ustawiane są : active1,2,3 = 1 ; fb1,2,3 = 0 ; eb1,2,3 = N1,2,3 .
Następnie równolegle uruchamiane są P1,2,3,4 .

Semafory binarne

Mogą przyjmować wartości 0 lub 1 . Na semaforze binarnym Sb można wykonać następujące operacje :

Pb(sem) – działa analogicznie jak P dla zwykłych semaforów

Vb(sem) – jak V ,ale nie zmienia wartości semafora binarnego jeśli miał on wartość 1

Implementacja operacji semaforowych na semaforach binarnych

Pb(Sb) :

```
repeat
 not testandset (Pactive,Sb) // instrukcja nottestandset ma działanie :
                                  // { Pactive = Sb;  Sb = false; }
until (Pactive);
```

Czyli tak długo , jak Pactive = 0 , Pactive jest przypisywane Sb , które jest następnie zerowane – tak więc jedynym powodem wyjścia z pętli może być zmiana Sb poprzez **Vb**(Sb)

Vb(Sb) :

Sb:= **true**;

Implementacja ogólnych operacji semaforowych

```
procedure P(S)
begin
  while S≤0 do ;  // czekanie , aż wartość semafora będzie większa niż 0
  S:= S-1;           // zmniejszenie wartości S o 1 ( po wyjściu z pętli )
end;

procedure V(S)
begin
  S:= S+1;
end;
```

Rozwiązanie ma 3 wady : – jest nieatomowe – wiele procesów może modyfikować S naraz , nie ma zabezpieczenia przed przyjęciem ujemnej wartości przez zmienną semaforową – np. gdyby 2 procesy jednocześnie czekały aż S>0 , S w wyniku wykonania V(S) przez jakiś inny proces wyniosłoby 1 i jednocześnie dowiedziały się o tym i wykonały S:= S-1 , to wtedy S przyjęłoby wartość –1 , ponadto oba procesy przeszłyby dalej (choć powinien tylko 1 z nich) . Poza tym w rozwiązaniu tym zastosowano aktywne czekanie – proces czekający , mimo , iż właściwie nic nie może robić , korzysta z czasu procesora

Implementacja operacji semaforowych.

```
program PVimplementation; // z aktywnym czekaniem
var active, delay: boolean;
var NS:  integer

procedure Pimplementation;
var Pactive, Pdelay: boolean;
begin
  Pactive:= true;
  while Pactive do testandset(Pactive, active); // konieczne jest wzajemne wykluczanie – tylko
1 proces może wykonywać operację na semaforze. Zapewnia je powyższa pętla – proces
czeka aż active przyjmie wartość false . Instrukcja testandset zapewnia atomwość operacji .
  NS:= NS - 1;
  if NS ≥ 0 then // równoważne  if  S > 0 ( bo teraz NS = S – 1 ) – zm. semaforowa nieujemna
  begin
     S:=S-1; // ponieważ tylko 1 proces w sekcji krytycznej – można bezpiecznie zmniejszyć S
     active:=false; // wyjście z sekcji krytycznej ; teraz inny proces może wejść
  end else
  begin
     active:= false; // wyjście z sekcji krytycznej ; teraz inny proces może wejść
     Pdelay:= true; // proces ma czekać
     while Pdelay do  testandset(Pdelay, delay) // czeka tak długo , aż V zmieni delay na  false
  end
end;

procedure Vimplementation;
var Vactive, Vdelay: boolean;
begin
  Vactive:= true;
   while Vactive do testandset(Vactive, active);
   NS:= NS + 1;
   if  NS > 0 then S:= S + 1 // jeśli nic nie czeka  ( NS  > 0 czyli S > 0 ) - zwiększenie S
   else delay:= false;          // inaczej – niech któryś z procesów przestanie czekać
   active:= false
end;

begin
  active:= false; // 1 z procesów może wejść do sekcji krytycznej
  delay:= true
end.
```

Rozwiązanie poprawne - atomowe , zmienna semaforowa nieujemna , ale ma wadę – jeśli S = 0 i proces zostaje zawieszony , to mimo , że nic nie robi , musi być wyonywana pętla czyli korzysta z czasu procesora – aktywne czekanie

```
program PVimplementation; // implementacja operacji semaforowych bez aktyw. czekania
var active, delay: boolean;
var NS:  integer

procedure Pimplementation;
var Pactive, Pdelay: boolean;
begin
  Disable Interrupts;  // wyłączenie przerwań żeby było szybciej
  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);  // zablokowanie procesu który wywołał operację
    p := RemovefromRL; (* RL – Ready List *)  // i usunięcie go z listy gotowych do działania
    active:= false;
    Transfer control to p with Interrupts Enable; // włączenie przerwań i uruchomienie innego
  end
end;

procedure Vimplementation;
var Vactive, Vdelay: 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; (* LS – List associated with S *) // usunięcie procesu czekającego z
    listy procesów czekających na S
    Add p to RL  (* RL – Ready List *) // i dodanie do listy gotowych
  end ;
  active:= false;
  Enable Interrupts;
end;
```

Rozw. poprawne , bez aktywnego czekania – jeśli S = 0 , zamiast czekać w pętli , wywoływana jest procedura systemu operacyjnego powodująca zawieszenie procesu wywołującego P

Inna implementacja operacji semaforowych

```
Type semaphore = record
  Value: integer;
  L: list of process;
end;
```

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

```
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;
```

Implementacja operacji *signal*(S) = V(S);

```
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;
```

Rozwiązanie nie spełnia atomowości – nie ma wzajemnego wykluczania i może być jednocześnie wykonywane kilka operacji na 1 semaforze .

Inne operacje semaforowe ???

lock *w* : *L*: **if** *w* = 1 **then goto** *L* **else** *w* : = 1;
unlock *w*: *w* : = 0;

Nie jest to implementacja , tylko znaczenie tych operacji ; nie są one atomowe

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

DEQ(*r*):
p : = *Removefromr-queue* ;
if $p \neq \Omega$ **then** *Activate p* (* p= Ω means that queue was empty *)
else *inuse*[*r*] : = **false** ;

ENQ(r) – jeśli zasób r jest zajęty – dołączenie procesu p do kolejki z nim skojarzonej , jeśli zasób wolny – p może na nim operować , ale inne już nie .
DEQ(r) – zdejmuje p z kolejki , i jeśli kolejka nie jest pusta uaktywnia go , w przeciwnym wypadku można już operować na zasobie .

WAIT(*e*):
if ¬ *posted*[*e*] **then** (* only one process can wait for event e *)
begin
wait[*e*] := **true** ; *process*[*e*] := *p* ; *Block p*
end
else *posted*[*e*] := **false**

W każdej chwili tylko 1 proces może czekać na nadejście jakiegoś zdarzenia .
WAIT(e) – jeśli nikt nie czeka na e , to procesem czekającym staje się p , który zostaje zablokowany

POST(*e*):
if ¬ *posted*[*e*] **then**
begin
posted [*e*] := **true** ;
if *wait*[*e*] **then**
begin
wait[*e*] := **false**; *Activate process*[*e*]
end
end ;

Block(*i*):
if *ready*(*i*) **then** *Block process i* (* process is ready or running *
else wws[*i*] := **false** ; (* flag associated with process i *)
// wws – false jeśli blokowanie zablokowanego

Wakeup(*i*):
if *ready*(*i*) **then** wws[*i*] : = **true else** *Activate process i* ;
// wws – true jeśli pobudka obudzonego

BLOCK(i) – jeśli i gotowy – zostaje zablokowany , w innym wypadku flaga wws dla i zostaje ustawiona na false .
WAKEUP(i) – jeśli i gotowy – flaga wws ustawiana na true , jeśli i nie jest gotowy – uaktywnienie procesu i .

Event counters

Three operations are defined on a event counter *E*:

1. Read(*E*): Return the current value of *E*.
2. Advance(*E*): Automatically increment *E* by 1.
3. Await(*E*,*v*): Wait until *E* has a value of *v* or more.

Producer-consumer problem using event counters.

```
#include  "prototypes.h"
#define N 100                          /* number of slots in the buffer */
typedef int event_counter ;            /* event_counters 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 ;    /* count items produced so far */
   await (out, sequence - N) ;  /* wait until there is room in buffer */ aż out >= sequence– N
   enter_item (item) ;          /* put item in slot (sequence -1) % N */
   advance (&in) ;              /* let consumer know about another item */ in ++
  }
 }

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 */ aż in >=  sequence
   remove_item (&item) ;         /* take item from slot (sequence -1) % N */
   advance(&out) ;               /* let producer know that item is gone */ out ++
   consume_item (item) ;         /* do something with the item */
 }
}
```

Producent zmienia licznik in , konsument – licznik out .
Producent czeka aż out >= sequence – N , czyli liczba wyjętych z bufora jest >= niż (liczba wyprodukowanych – pojemność bufora) , czyli (liczba wyjętych + poj. bufora) >= liczba wyprodukowanych – oznacza to , że można produkować
Konsument czeka , aż in >= sequence , czyli liczba wyprodukowanych >= liczba skonsumowanych .

Programowe mechanizmy synchronizacji

Regiony krytyczne - definicja i implementacja

A variable *v* of type *T*, which is to be shared among many processes, can be declared:

var *v*: **shared** *T* ;

The variable *v* can be accessed only inside a region statement of the following form:

region *v* **do** *S* ;

For each declaration

var *v*: **shared** *T* ;

the compiler generates a semaphore *v-mutex* initialized to 1.
For each statement

region *v* **do** *S* ;

the compiler generates the following code:

wait(*v-mutex*) ; // rejony krytyczne gwarantują wzajemne wykluczanie operacji na
S ; // zmiennych dzielonych
signal(*v-mutex*) ;

Conditional critical regions.

Conditional critical region has the form

region *v* **when** *B* **do** *S* ;

where *B* is a Boolean expression. As before, regions referring to the same shared variable exclude each other in time. Now, however, when a process enters the critical-section region, the Boolean expression *B* is evaluated. If the expression is *true*, statement *S* is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until *B* becomes true and no other process is in the region associated with *v*.

Producer- consumer problem using critical regions

```
var buffer:
shared record
  pool: array [0..n – 1] of item ;  // bufor właściwy
  count, in, out: integer ;          // liczniki
end ;
```

The producer process inserts a new item *nextp* into the shared buffer by executing

```
region buffer when count < n do  // jeśli bufor nie jest pełny
begin
  pool[in] : = nextp ;  // dodanie do bufora
  in : = (in + 1) mod n ;  // które miejsce w buforze dla następnego
  count : = count + 1 ;  // zwiększenie licznika elementów w buforze
end ;
```

The consumer process removes an item from the shared buffer and puts it in *nextc* by executing

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

Implementation of the conditional region construct

region *v* **when** *B* **do** *S* ;

```
var x-mutex, x-delay : semaphore; // początkowo ustawione na : x-mutex : = 1 ; x-delay := 0
semafory binarne ; x-delay – do czekania na B = true ; x-mutex – do czekania na wejście do S
    x-count, x-temp : integer; // obie zmienne początkowo mają wartośc 0
(*x-count – the number of processes waiting for x-delay *)
(*x-temp – the number of processes that have been allowed to test their Boolean condition
during one trace *)

wait(x-mutex) ;  // dla wzajemnego wykluczania
if not  B then
begin
  x-count : = x-count + 1 ;  // zwiększenie liczby czekających na spełnienie B
  signal(x-mutex) ; // jakiś inny może wejść do sprawdzania B ( i jeśli dla niego B = true – do
S)
  wait(x-delay) ;  // teraz czekanie na x-delay aż jakiś wychodzący pozwoli testować B
  while not B do
  begin
    x-temp : = x-temp + 1 ;  // teraz testuje o 1 więcej
    if x-temp < x-count  then signal(x-delay) // jeśli wciąż mniej testowanych niż czekających

    przepuszczenie kolejnego czekającego na spełnienie B do testowania
    else signal(x-mutex) ; // x-temp = x-count oznacza , że to już ostatni proces testujący -
   pozwala on więc jakiemuś procesowi czekającemu na wejście do regionu na wejście do
niego
    wait(x-delay) ; // czekanie na przepuszczenie przez wychodzącego z regionu – i jeśli wtedy
    B będzie spełnione , proces przejdzie do S , w przeciwnym przypadku – cała pętla od nowa
  end ;
  x-count : = x-count – 1 ; // jeśli B =true dla jakiegoś – już nie czeka i zmniejsza liczbę czek.
end ;
S ;
if x- count > 0 then  // jeśli jakieś czekają
begin
  x-temp : = 0 ; // żeby testowanie B przez czekające na B mogło przebiegać poprawnie
  signal(x-delay);  // przepuszczenie pierwszego procesu czekającego na B do testowania B
end                 //  proces ten przepuści następne do testowania B
else signal(x-mutex) ;  // nikt nie czeka na spełnienie B – ktoś może zacząć wykonywać S
```

Jeśli teraz jakiś proces wychodząc z S wykona signal (delay) - czekający przejdzie dalej .

W pętli **while** jest wykonywane testowanie B dla kolejnych procesów

Za każdym razem , kiedy jakiś proces opuści region , następuje dla wszystkich procesów czekających na B sprawdzanie wartości ich B (która mogła się zmienić w czasie gdy jakiś proces wykonywał S) . Jeśli ten proces po raz pierwszy czeka na B – to albo wejdzie do pętli , albo ją przeskoczy . Procesy czekające w pętli będą w niej czekały tak długo , aż B nie będzie dla nich spełnione . Każdy z tych procesów zwiększa liczbę czekających , przepuszcza kolejnego (ostatni – jakiegoś czekającego na zewnątrz regionu) , po czym może wyjść z pętli lub czeka na x-delay . Jeśli jakiś proces wyjdzie z pętli w trakcie

testowania , to nie podniesie on x-delay w tej pętli i zrobi to dopiero jak wykona S .

Implementacja warunkowego regionu krytycznego jeśli warunki synchronizacji zlokalizowane są wewnątrz tego regionu :

```
region v
do begin
  S1 ;
  await (B) ;  // czekanie aż B = true
  S2 ;
end ;
```

The Readers-Writers Problem

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.

Priorytet czytelnika – żaden czytelnik ne powinien czekać , chyba , że pisarrz pisze czyli nie powinien czekać na zakończenie pracy innych czytelników tylko dlatego , że czeka na to też jakiś pisarz
Priorytet pisarza – jeśli pisarz jest gotowy , to to rozpocznie wykonanie swojej pracy tak wcześnie jak to tylko możliwe – jeśli jakiś pisarz czeka , to żaden nowy czytelnik nie rozpocznie czytania

Readers-Writers problem using semaphores // z priorytetem czytelnika

```
shared var
nreaders : integer ;  // ilu czytelników czyta w danej chwili
mutex, wmyutex, srmutex : semaphore ;

procedure reader ;
begin
  P(mutex) ;  // żeby tylko 1 czytelnik mógł naraz zmieniać liczbę czytelników
  if nreaders = 0  then  // pierwszy czytelnik dopuszczony do czytania
  begin
    nreaders : = nreaders + 1 ;  // zwiększa aktualną liczbę czytelników
    P(wmutex)                    // opuszcza  semafor dla pisarzy  ( lub czeka jeśli ktoś pisze )
  end
  else nreaders : = nreaders + 1;  // kolejni – tylko zwiększają liczbę , bo semafor już opuszcz.
  V(mutex) // żeby wielu mogło równolegle czytać jeśli zakończą pierwszą sekcję krytyczną
  read(f,d) ;
  P(mutex) // wejście do drugiej sekcji krytycznej
  nreaders := nreaders – 1 ;
  i f nreaders = 0 then V(wmutex);  // podniesienie pisarzom jeśli nikt nie czyta
  V(mutex) ;  // wyjście z sekcji krytycznej – można podnieść semafor
end ;

procedure writer(d: data) ;
begin
  P(srmutex) ;
  P(wmutex) ;  // żeby naraz tylko 1 pisarz mógł pisać
  write(f, d) ;
  V(wmutex) ;  // teraz inny pisarz może pisać
  V(srmutex) ;
end ;

begin (* initialization* )
  mutex = wmutex = srmutex = 1 ; // wszystkie semafory – binarne
  nreaders : = 0 ;
end.
```

Semafor srmutex zapewnia priorytet czytelnikom – jeśli np. pisarz pisze , a na prawo dostępu do pliku oczekują zarówno pisarze , jak i czytelnicy , to oczekują oni na wmutex . Jeśli jednak pisarz skończy , to podniesie wmutex – i wtedy mogą już wejść czytelnicy , bo czekają tylko na wmutex , a pisarze muszą jeszcze wpierw mieć podniesiony srmutex co nastąpi później .

Readers - Writers problem using critical regions // z priorytetem pisarza

```
var v: shared record
   nreaders, nwriters: integer ;
   busy: boolean ;  // czy jakiś pisarz pisze
end ;
```

Reader`s process

```
region v do
begin
  await (nwriters = 0) ;  // czekanie aż żaden pisarz nie czeka – a więc priorytet pisarza
  nreaders : = nreaders + 1 ;  // zwiększenie liczby czytających
end ;
...
read file
...
region v do
begin
   nreaders : = nreaders – 1
end ;
```

Writer process

```
region v do
begin
  nwriters: = nwriters + 1 ;
  await((not busy) and (nreaders = 0)) ; // czekanie aż nikt nie pisze i nikt nie czyta
  busy: = true ;
end ;
...
write file
...
region v do
begin
  nwriters : = nwriters – 1 ;
  busy: = false ;
end ;
```

Monitory: definicja i implementacja

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 ;
```

A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type *condition*:

var *x*, *y*: *condition* ;

The only operations that can be invoked on a condition variable are *wait* and *signal*. The operation

x.wait ;

means that the process invoking this operation is suspended until another process invokes

x.signal ;

The *x.signal* operations resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect; that is, the state of *x* is as though the operation was never executed.
Jeśli proces odwoła się do zajętego monitora – proces odwołujący się zostaje wstrzymany i umieszczony w kolejce związanej z monitorem . Natomiast procesy które wywołały np. *x.wait* są zawieszane w kolejce związanej ze zmienną warunkową , poza monitorem – nie blokując tym samym innym procesom wejścia do monitora – inaczej byłby deadlock .

Producer – consumer problem using monitor

```
type ProducerConsumer = 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.signal ;
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 ;
    ProducerConsumer.enter ;
  end
end ;

procedure consumer ;
begin
  while true do
  begin
    ProcedureConsumer.remove ;
    consume_item
  end ;
end.
```

Resource allocation using monitor

```
type resource-allocation = monitor
var busy: boolean ;
x: condition ;

procedure entry acquire (time : integer);
begin
  if busy then x.wait(time); (* process priority *)
  busy : = true ;
end ;

procedure entry release ;
begin
  busy : = false ;
  x.signal ;
end ;

begin
  busy : = false ;
end.
```

Readers – Writers problem using monitor // z priorytetem czytelnika

```
type readers-writers : monitor ;
var     redaercount : integer ;
busy : boolean ;  // czy ktoś pisze
OKtoread, OKtowrite : condition ;

procedure entry startread ;
begin
  if busy then OKtoread.wait ;  // ktoś pisze – czekanie aż zmieni  OKtoread
  readercount : = readercount + 1 ;
  OKtoread.signal ;  (* Once one reader can start, they all can *)  // no właśnie
end startread ;

procedure entry endread ;
begin
  readercount : = readercount – 1 ;
  if readercount  = 0 then OKtowrite.signal ; // priorytet czytelnika – przepuszczenie pisarza
end endread ;                                  // dopiero jeśli nikt nie czyta

procedure entry startwrite ;
begin
  if busy OR readercount ≠ 0 then OKtowrite.wait ;
  busy : = true ;
end startwrite ;

procedure entry endwrite ;
begin
  busy : = false ;
  if OKtoread.queue then OKtoread.signal // prior. czytelnika – przepuszczenie czytelników
  else OKtowrite.signal ;                // a pisarzy tylko jeśli żaden czytelnik nie czeka
end endwrite ;


begin (* initialization *)
  readercount : = 0 ;
  busy : = false ;
end ;
```

Monitor implementation.

Dla każdego monitora - semafor *mutex* z wartością początkową 1, przed wejściem do monitora proces musi wykonać wait(mutex) , przy wyjściu signal (mutex) Ponieważ proces sygnalizujący musi czekać na wyjście lub rozpoczęcie czekania przez proces wznowiony - bo w przeciwnym wypadku zarówno sygnalizujący , jak i wznowiony działałyby naraz w monitorze - wprowadza się dodatkowy semafor *next* z wartością początk. 0 , za którego pomocą procesy sygnalizujące mogą same wstrzymywać swoje wykonanie .

wait(*mutex*) ; // najpierw sami czekamy
(**mutex* – semaphore to guarantee mutual exclusion
...
body of *F* ;
...
if *next-count* > 0 // po wyjściu można kogoś przepuścić
(**next-count* – the number of processes invoking *x.signal* and suspended *)
then *signal*(*next*) // na przykład czekającego na next
(**next* – semaphore to suspend a process invoking *x.signal* *)
else *signal*(*mutex*); // a jeśli tam nikt nie czeka , to może na mutex ?

x.wait :

x-count : = *x-count* + 1 ;
(**x-count* – the number of processes waiting for *x.signal**)
if *next-count* > 0 **then** *signal*(*next*) // teraz czekamy i jakiś inny może wejść do monitora – jeśli
else *signal*(*mutex*) ; // jakiś czeka na next , lub jeśli nie na next , to może na mutex
wait(*x-sem*) ;
(* semaphore to suspend a process invoking *x.wait* *)
x-count : = *x-count* – 1 ; // wykonane po odczekaniu i x-signal

x.signal :

if *x-count* > 0 **then** // jeśli jakieś wykonały x.wait i czekają na x.signal – trzeba im ustąpić
begin
next-count : = *next-count* + 1 ; // teraz więcej czeka na next
signal(*x.sem*) ; // przepuszczamy jakiegoś czekającego na x.signal
wait(*next*) ; // i sami czekamy na signal(next)
next-count : = *next-count* – 1 ; // po odczekaniu czeka o jeden mniej
end .

The Dining Philosophers Problem

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.

Dining-Philosophers problem using monitor

```
type dinning-philosophers = monitor
var state : array [0..4] of  (thinking, hungry eating) ;
var self : array [0..4] of condition ; // za pomocą tablicy self głodni będą czekać na widelce

procedure entry pickup (i: 0..4) ;
begin
  state[i] := hungry ;
  test (i) ;  // sprawdzenie czy  może jeść
  if state[i] ≠ eating then self[i].wait ; // jeśli nie może jeść – niech  czeka
end ;

procedure entry putdown (i: 0..4) ;
begin
  state[i] : = thinking ;
  test (i + 4 mod 5) ;  // sprawdzenie czy  sąsiedzi mogą jeść
  test (i + 1 mod 5) ;
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
  begin // obaj sąsiedzi nie mogą jeść i  sam musi być głodny
    state[k] : = eating ;  // wtedy k-ty może rozpocząć jedzenie
    self[k].signal ;  // i może przestać czekać
  end ;
end ;

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

procedure philosoph_i;
begin
  while true do
  begin
    pickup (i);
    jedzenie ;
    putdown (i);
  end;
end;
```

Rozwiązanie zapewnia , że dwaj sąsiedzi nigdy nie będą jedli równocześnie ; nie dojdzie też do blokady – ale istniej możliwość zagłodzenia filozofa

Operacje wymiany komunikatów (ang. message passing)

Wymiana komunikatów realizowana jest z użyciem dwóch podstawowych *operacji komunikacyjnych:*

- *send(P, m)* i
- *receive(Q, m)*,

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

Łącza

Jeżeli procesy *P* i *Q* chcą komunikować się ze sobą, to musi istnieć między nimi *łącze komunikacyjne (kanał)*. Można wyróżnić następujące łącza:

- *jednokierunkowe* i *dwukierunkowe*;
- *niebuforowane* i *buforowane* (o określonej, niezerowej *pojemności*);
- *zachowujące uporządkowanie* wiadomości (*FIFO*) i *niezachowujące uporządkowania* wiadomości (*non-FIFO*);
- *synchroniczne* (o określonym czasie transmisji) lub *asynchroniczne* (o nieokreślonym czasie transmisji);
- *niezawodne* (gwarantujące, że żadna wiadomość nie jest tracona, duplikowana lub zmieniana - ang. reliable, lossless, duplicate free, error free, uncorrupted, no spurious), lub *zawodne* (wiadomość może być utracona, zduplikowana lub zmieniona).

Określanie nadawców i odbiorców

Procesy mogą komunikować się *bezpośrednio* lub *pośrednio*. 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*.

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*. 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:

- *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.

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 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.

Operacje synchroniczne i asynchroniczne

Istnienie buforów umożliwia realizację a*synchronicznych* (*nieblokowanych) operacji komunikacji,* w których węzeł nadający przekazuje komunikat do bufora i natychmiast kontynuuje swe działanie. Z kolei węzeł odbiorczy próbuje w tym przypadku jedynie odczytać stan bufora wejściowego łącza, lecz nawet gdy bufor jest pusty węzeł kontynuuje działanie.

W wypadku *synchronicznych (blokowanych) operacji komunikacji,* nadawca jest wstrzymywany do momenty, gdy wiadomość została przesłana lub nawet odebrana i

potwierdzona, natomiast odbiorca - do momentu, gdy oczekiwana wiadomość zostaje pobrana z jego bufora wejściowego.

W tym kontekście, wyróżnia się komunikację synchroniczną i asynchroniczną. W *komunikacji synchronicznej*, nadawca i odbiorca są blokowani aż odpowiedni odbiorca odczyta przesłaną do niego wiadomość (rendezvous). W przypadku komunikacji asynchronicznej, nadawca lub odbiorca komunikuje się w sposób nieblokowany.

```
program producer_consumer_message_transmission;
var buffer_pool: array[0..x] of buffer;

procedure producer;
begin
  while true do
  begin
    produce_next_message;
    receive(producer, empty);            /* odbiór blokowany */
    add_message_to_common_buffer;
    send(consumer, empty)                /* wysłanie asynchroniczne */
  end
end;

procedure consumer;
begin
  while true do
  begin
    receive(consumer, empty);            /* odbiór blokowany */
    take_message_from_common_buffer;
    send(producer, empty);               /* wysłanie asynchroniczne */
    process_message
  end
end;
```

```
begin
  I:= N;
  while I>0 do
  begin
    send(producer, empty);
    I:= I - 1
  end;
  parbegin
    producer;
    consumer
  parend
end.
```

Najpierw program wysyła tyle sygnałów do producenta , ile jest miejsc w buforze . Producent zawsze przed zapisaniem do bufora czeka na sygnał – dopiero gdy go otrzyma przechodzi dalej i sam wysyła sygnał do konsumenta , nie czekając aż tamten odbierze ten sygnał .

Problem P-K za pomocą wymiany wiadomości przy założeniu , że jedynym mechanizmem komunikacji i synchronizacji jest wymiana wiadomości – bez operacji add i take – rekordy są przesyłane za pomocą send i receive

```
procedure producer
begin
  while true do
  begin
    produce;
    receive(producer,empty); // odbiór synchroniczny – czeka na empty
    send(consumer,record);     // zapis asynchroniczny – wysyła i przechodzi dalej
  end;
end;

procedure consumer
begin
  while true do
  begin
    receive(consumer,record);    // czeka na rekord
    send(producer,empty);        // pobrał – producent może dalej działać ( jeśli czekał na to )
    consume;
  end;
end;


begin
  I:= N;
  while I>0 do
  begin
    send(producer, empty);
    I:= I - 1
  end;
  parbegin
    producer;
    consumer
  parend
end.
```

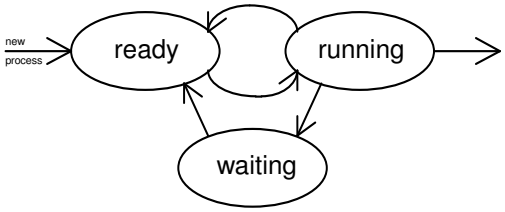
Zarządzanie procesami w systemach operacyjnych.

Alen Shaw: "Proces sekwencyjny (zwany niekiedy zadaniem) jest działalnością wynikłą z wykonywania programu wraz z jego danymi przez procesor sekwencyjny".

Abraham Silberschatz: "Proces jest jednostką pracy systemu. Proces jest czymś więcej niż kodem programu z określoną bieżącą czynnością. W ogólności proces obejmuje również stos zawierający dane tymczasowe (takie jak parametry procedur, adresy powrotów, zmienne tymczasowe) sekcje danych zawierające zmienne globalne oraz zestaw informacji pomocniczych".

Andrew Tanenbaum: "Proces jest wykonywanym programem wraz z bieżącymi wartościami licznika rozkazów, rejestrów i zmiennych."

Proces jest najmniejszą jednostką aktywności, która może ubiegać się samodzielnie o przydział zasobów systemu komputerowego. Proces obejmuje wykonywany program wraz ze zmiennymi określającymi stan przydzielonych zasobów: procesora, pamięci operacyjnej, urządzeń wejścia/wyjścia, plików, systemowych struktur danych itp.



Process state diagram

PROCESS CONTROL BLOCK - PBC

POINTER	PROCESS STATE
PROCESS NUMBER	
PROGRAM COUNTER	
REGISTERS ⋮	
MEMORY MANAGEMENT INFORMATION	
CPU SCHEDULING INFORMATION	
ACCOUNTING INFORMATION	
I/O STATUS INFORMATION - LIST OF OPEN FILES ...	

Accounting inf. – inf. o kosztach przetwarzania - czasami chcemy preferować procesy o mniejszym zapotrzebowaniu na zasoby systemu i wtedy konieczna jest inf. o tym

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to *bss* segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective *uid*
Time when process started	Proces *id*	Effective *gid*
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real *uid*	
Message queue pointer	Effective *uid*	
Pending signal bits	Real *gid*	
Process *id*	Effective *gid*	
Various flag bits	Bit maps for signals	
	Various flag bits	

Process state - opisuje stan wykonywania procesu; **process number** - jest unikalnym identyfikatorem procesu; **program counter** i **registers** - opisują stan procesora; **memory menagement information -** to informacja opisująca obszary przydzielonej procesowi pamięci operacyjnej; **accounting information.** - to informacja opisująca rozliczenia; **CPU scheduling information -** to informacja określająca priorytet procesu; **I/O status information** to informacja dotycząca oczekiwanych zdarzeń, otwartych plików itp.

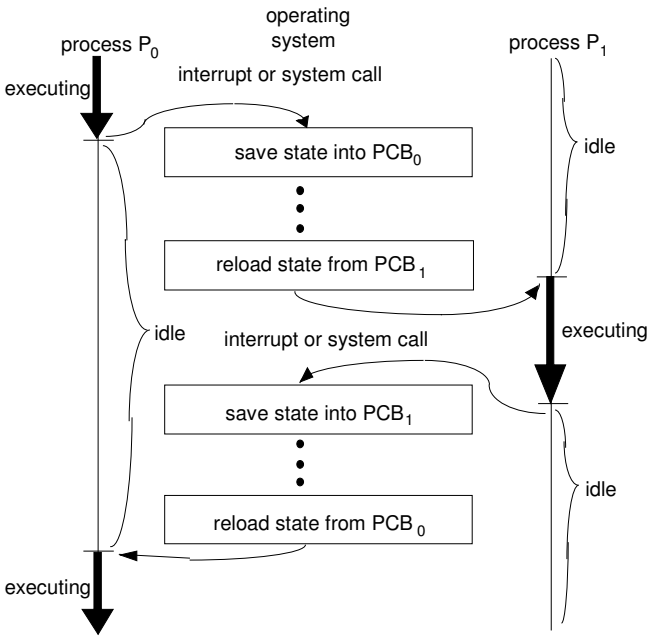
Przełączanie kontekstu

Przełączanie kontekstu polega na zmianie przydziały procesora. Realizowane jest ono z wykorzystaniem systemu przerwań. Załóżmy, że wykonywany jest pewien proces użytkowy i pojawiło się przerwanie "dyskowe". Wówczas:

1. Licznik rozkazów, słowo stanu programu i podstawowe rejestry są składowane na stosie systemowym przez sprzęt obsługujący przerwanie.
2. Wykonywany jest skok do adresu wskazanego przez wektor przerwań.

Kolejne kroki są wykonywane programowo - przez odpowiednie moduły systemu operacyjnego.

3. Procedura obsługi przerwania rozpoczyna się od zapamiętania wszystkich pozostałych rejestrów w tablicy procesów.
4. Numer indentyfikacyjny (*Id*) bieżącego procesu i wskaźnik do tablicy procesów są zapamiętywane pod odpowiednimi zmiennymi systemowymi.
5. Identyfikowany jest proces, który zainicjował wykonywanie operacji dyskowej spośród procesów znajdujących się w stanie zawieszenia.
6. Zidentyfikowany proces przechodzi w stan **gotowości.**
7. Wywołany zostaje moduł szeregujący (*proces scheduler*) w celu zdecydowania, któremu z procesów znajdujących się w stanie **gotowości** ma być przydzielony procesor.



Przełączanie kontekstu

Tworzenie nowych procesów

Aby procesy w systemie mogły być wykonywane współbieżnie, musi istnieć mechanizm tworzenia i usuwania procesów. Proces może tworzyć nowe procesy za pomocą funkcji systemowej **utwórz** (ang. create) proces. Proces tworzący nowe procesy nazywa się **procesem macierzystym**, utworzone zaś przez niego procesy są nazywane **procesami potomnymi**.

Do realizacji swych zadań proces potrzebuje na ogół pewnych zasobów (czasu procesora, pamięci, plików). Gdy proces tworzy podproces, ten ostatni może otrzymać zasoby od:
- systemu operacyjnego, lub
- procesu macierzystego, którego zasoby własne zostają wówczas uszczuplone.

Proces macierzysty może rozdzielać własne zasoby między procesy potomne lub powodować, że niektóre zasoby będą przez potomków współdzielone.Gdy proces tworzy nowy proces, wtedy w odniesieniu do jego działania praktykuje się dwojakie postępowanie:
- proces macierzysty kontynuuje działanie wspólnie ze swoimi potomkami,
- proces macierzysty oczekuje (funkcja systemowa ***wait***), dopóki wszystkie jego procesy potomne nie zakończą pracy.

Do wypełniania swych zadań proces potrzebuje pewnych zasobów dodatkowych. Zasoby te są przydzielane bądź w chwili tworzenia procesu bądź też dynamicznie w odpowiedzi na żądanie przydziału.

Relacje między procesami.

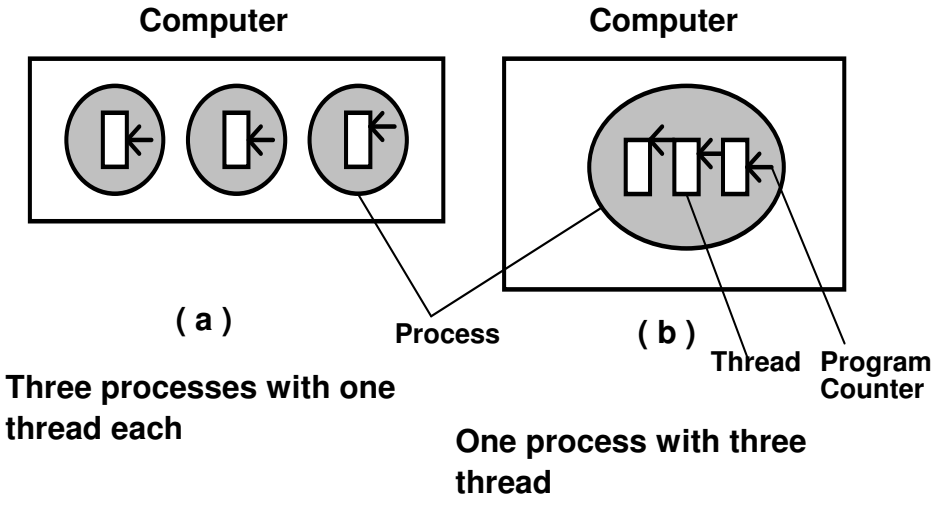
Wyróżnia się:
- procesy niezależne,
- procesy zależne (inaczej - współpracujące).

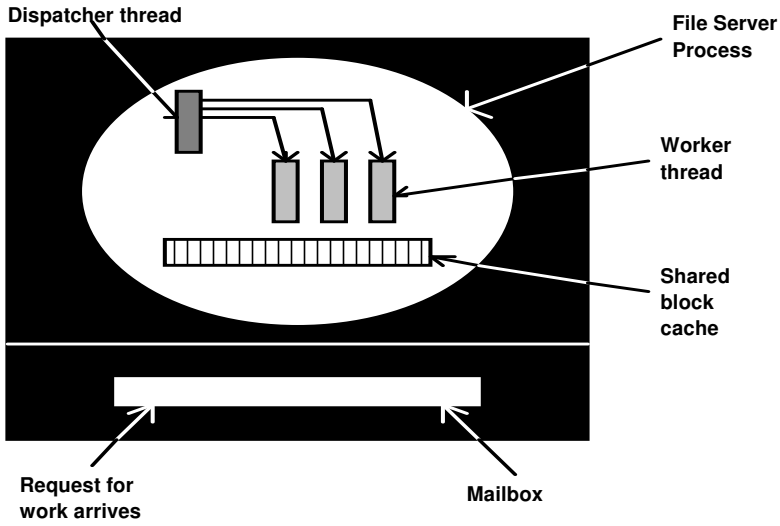
Proces niezależny ma następujące własności:
- na jego stan nie wpływa żaden inny proces,
- jego działanie jest deterministyczne, tzn. wynik jego pracy jest zależny wyłącznie od jego stanu wejściowego (chyba, że zawiera instrukcje wyboru niedeterministycznego),
- jego działanie jest powtarzalne, tzn. wynik pracy procesu niezależnego jest zawsze taki sam przy takich samych danych (chyba, że zawiera instrukcje wyboru niedeterministycznego),
- jego wykonywanie może być wstrzymywane i wznawiane bez żadnych szkodliwych skutków.

Proces współpracujący ma następujące własności:
- jego stan jest współdzielony z innymi procesami,
- współpracuje on z innymi procesami w celu realizacji wspólnego celu przetwarzania,
- jego wykonywanie może być uzależnione od stanu wykonania innych procesów,
- wynik działania procesu może zależeć on od względnej kolejności wykonywania wszystkich współpracujących procesów.

Wątki

W większości tradycyjnych systemów operacyjnych każdy proces ma własną przestrzeń adresową i pojedynczy wątek sterowania. Występują jednak sytuacje, w których pożądane byłoby posiadanie wielu wątków sterowania. Rozważmy przykład serwera dyskowego, który od czasu do czasu musi się blokować w oczekiwaniu na zakończenie operacji dyskowej. Jeśli taki serwer miałby wiele wątków sterowania, to drugi wątek mógłby być wykonywany podczas, gdy pierwszy znajduje się w stanie zawieszenia. Można by w ten sposób osiągnąć większą sprawność sieci i lepszą przepustowość. Nie można tego celu osiągnąć tworząc dwa niezależne procesy serwerów, ponieważ musiałyby one dzielić wspólną pamięć buforową.



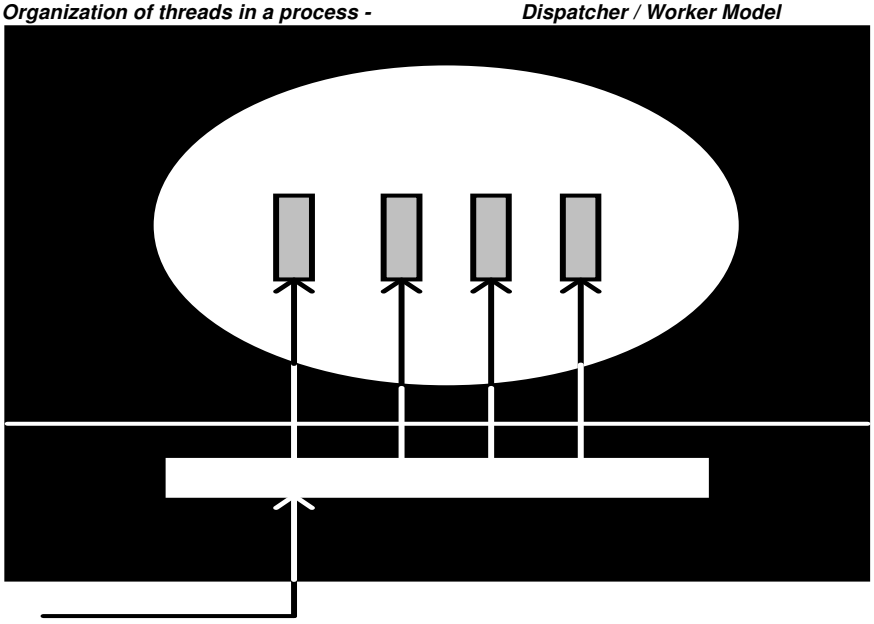


Team model - równorzędne wątki

Model hierarchiczny :

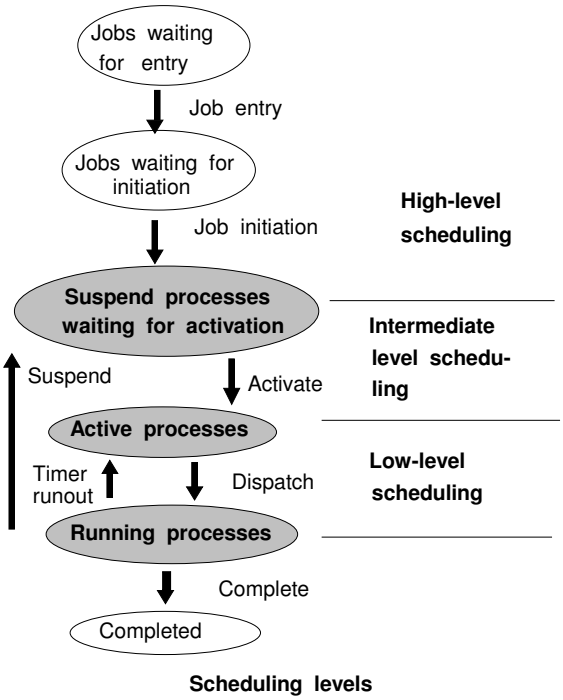
Dispatcher – zarządca
Worker – uaktywniany na żądanie zarządcy

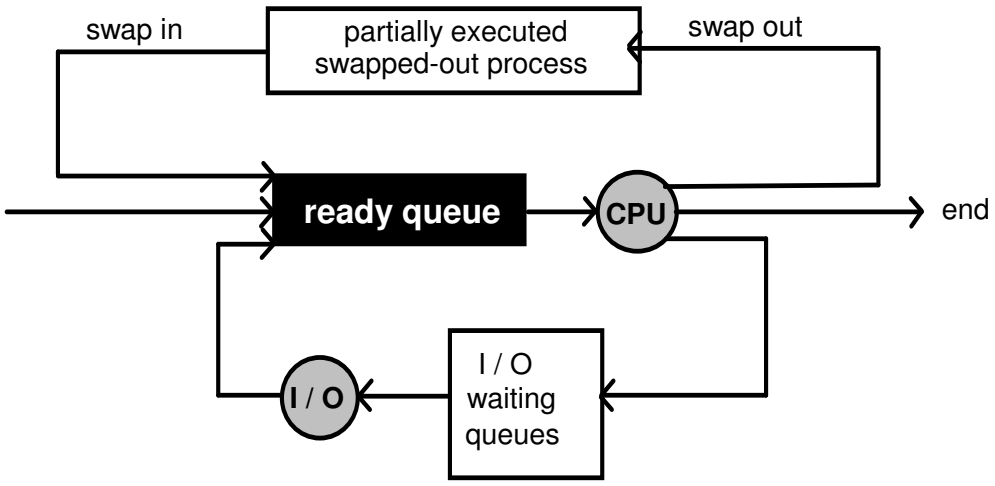
Organization of threads in a process - Dispatcher / Worker Model



Organization of threads in a process - Team Model

Algorytmy szeregowania





Kryteria oceny algorytmów szeregowania:

- **długość uszeregowania** C_{max} - czas wykonania zbioru procesów,
- **przepustowość** - liczba procesów wykonana w ciągu jednostki czasu,
- **średni czas przebywania procesu w systemie** - średni czas odpowiedzi,
- **stopień wykorzystania procesora.**

średni czas – kryterium użytkownika , pozostałe – kryteria właściciela systemu

Algorytmy przydziału procesora dzielą się na dwie podstawowe grupy:

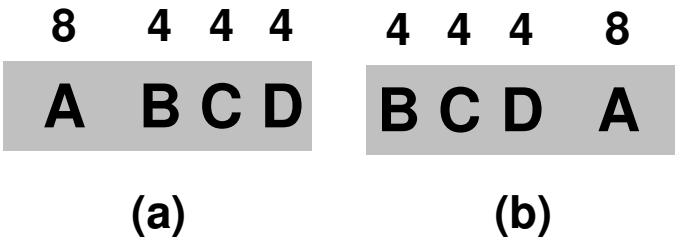
- **z wywłaszczaniem/podzielne** (ang. preemptive scheduling) - procesor może być odebrany procesowi, a zawieszony proces może być kontynuowany na innym procesorze,
- **bez wywłaszczania/niepodzielne** (ang. nonpreemptive scheduling) - proces utrzymuje procesor aż do zakończenia pracy.

Algorytm FCFS (ang**.** First Come First Served)
Algorytm FCFS szereguje zadania zgodnie z porządkiem ich przybywania.

Charakterystyka algorytmu FCFS:
- algorytm bez wywłaszczania;
- implementacja - kolejka FIFO;
- nie preferuje żadnych zadań;
- nieprzydatny w systemach interakcyjnych z podziałem czasu;
- rzadko używany jako schemat podstawowy - często jako schemat wewnętrzny innych metod.

Algorytm SJF (ang. Shortest Job First)
Algorytm SJF szereguje zadania zgodnie z porządkiem określonym przez czasy ich wykonywania - najpierw wykonywane jest zadanie najkrótsze.

Charakterystyka algorytmu SJF
- algorytm bez wywłaszczania;
- procesor jest przydzielany procesowi, który ma najkrótszy przewidywany czas wykonania, a więc algorytm ten faworyzuje zadania krótkie,;
- udowodniono, że jest to algorytm optymalny ze względu na średni czas przebywania procesów w systemie.



Example of shortest job first scheduling

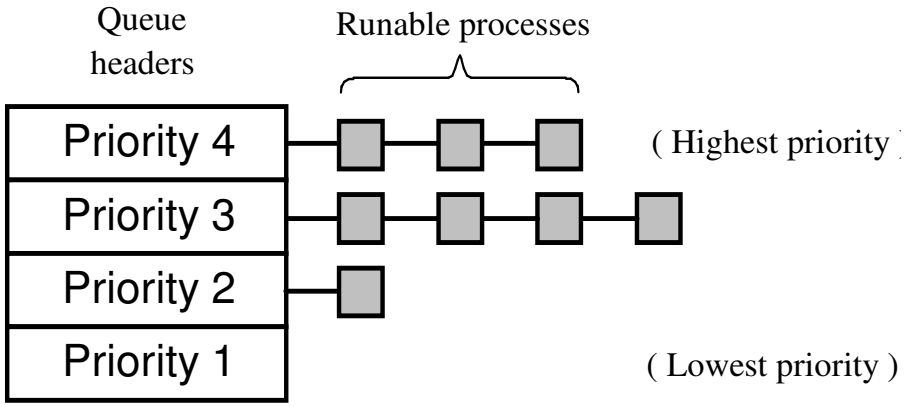
Problemy :
- skąd wiadomo , które zadanie będzie miało jaki czas ???
  (można estymować i takie inne pierdoły np. sprawdzać ile dane zadanie wykonywało się ostatnio , ale szkoda czasu i za duzo roboty , poza tym i tak nie zawsze się da)
- może być livelock dla zadań długich

Algorytmy priorytetowe

W algorytmach priorytetowych, każdemu procesowi przydziela się pewien priorytet, po czym procesor przydziela się temu procesowi, którego priorytet jest najwyższy.

Charakterystyka algorytmów priorytetowych:
- procesy o równych priorytetach są porządkowane na ogól według algorytmu FCFS;
- priorytety mogą być definiowane w sposób statyczny lub dynamiczny;
- priorytety mogą być przydzielane dynamicznie po to, aby osiągnąć określone cele systemowe, np. jeśli specjalny proces zażąda przydziału procesora, powinien go otrzymać natychmiast;
- planowanie priorytetowe może być wywłaszczające lub niewywłaszczające.

Podstawowym problemem w planowaniu priorytetowym jest **stałe blokowanie** (ang. *indefinite blocking, starvation, livelock*).



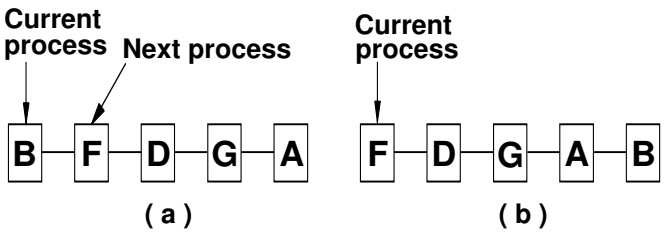
A scheduling algorithm with four priority classes.

Algorytm rotacyjny (cykliczny, karuzelowy) (ang. *Round Robin - RR)*

W algorytmie rotacyjnym procesor jest przydzielany zadaniom kolejno na określony odcinek czasu (kwant).

Charakterystyka algorytmu RR:
- kwant czasu przydziału procesora jest najczęściej rzędu 10 do 100 msek;
- kolejka procesów gotowych jest traktowana jak kolejka cykliczna - nowe procesy są dołączane na koniec kolejki procesów gotowych;
- jeśli proces ma fazę procesora krótszą niż przydzielony kwant czasu, to wówczas z własnej inicjatywy zwalnia procesor;
- jeśli faza procesora procesu jest dłuższa niż przydzielony kwant czasu, to nastąpi przerwanie zegarowe i przełączenie kontekstu, a proces przerwany trafia na koniec kolejki.

Podstawowym problemem przy konstrukcji algorytmu RR jest określenie długości kwantu czasu. (jeśli kwant czasu jest bardzo mały to algorytm *RR* nazywa się **dzieleniem procesora**).



Round Robin scheduling. (a) The list of runnable processes.

(b) The list of runnable processes after B's quantum runs out.

Jeśli b. długi kwant czasu (*timeslice*) , to upodabnia się do FIFO (dla kwantu nieskończonego każde zadanie się w nim zakończy czyli będzie zwykła kolejka)

Jeśli b. krótki – upodabnia się do SJF (im krótsze zadanie , tym większe P , że wyliczy się ono od razu w 1 kwancie , a długie będą ciągle przerywane i odsyłane na koniec)

Kwant czasu nie może być za krótki – im krótszy , tym stosunkowo więcej czasu traci się na przełączanie m. zadaniami

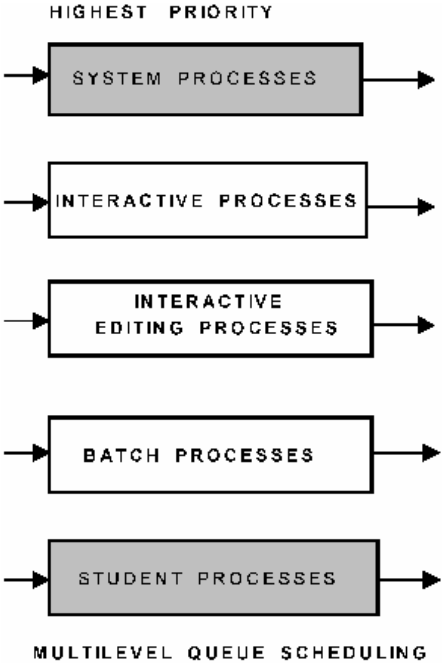
Algorytmy wielopoziomowe

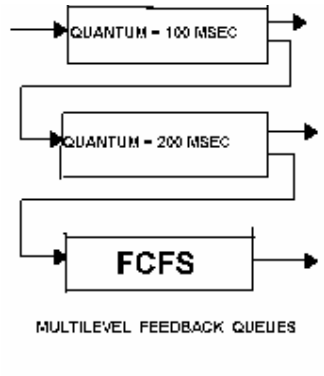
W algorytmach wielopoziomowych, zbiór procesów gotowych jest rozdzielany na wiele kolejek, a szeregowane w każdej kolejce realizowane jest według określonego algorytmu.

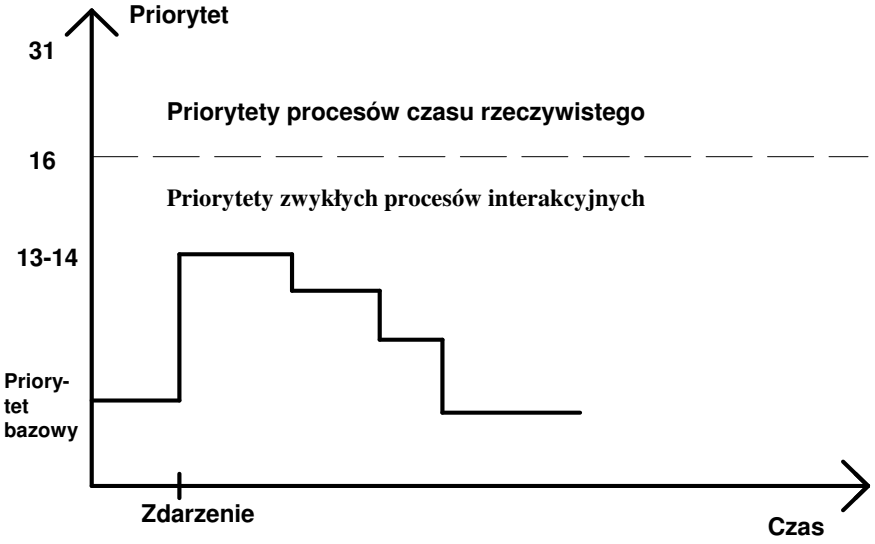
Charakterystyka algorytmów wielopoziomowych:
- procesy mogą lecz nie muszą być na stałe przypisywane do określonej kolejeki;
- każda kolejka może mieć własny algorytm szeregowania;
- musi istnieć plan przechodzenia między kolejkami.

W ramach planowania przechodzenia między kolejkami, wyróżnia się stałopriorytetowe planowanie wywłaszczające oraz planowanie ze sprzężeniem zwrotnym,

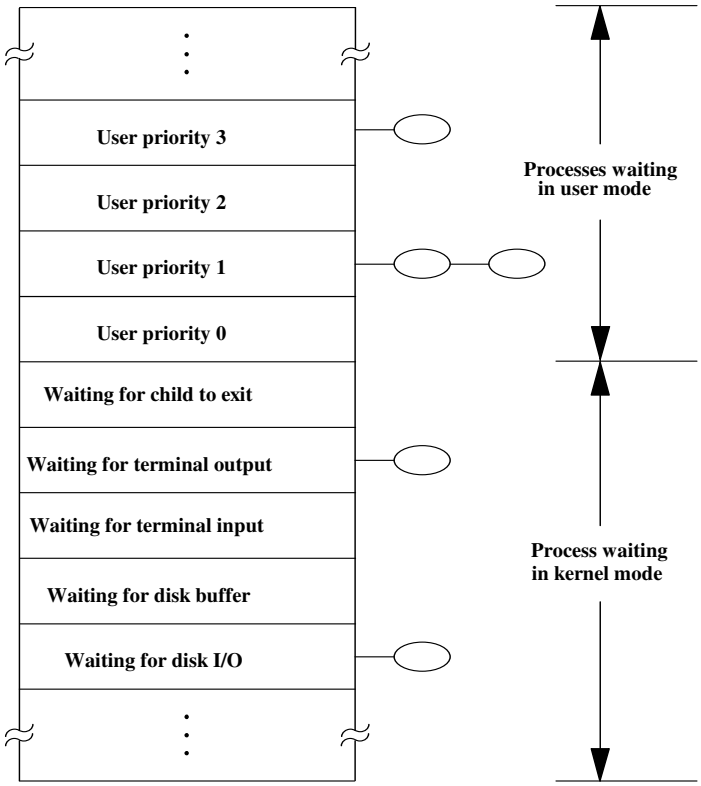
W **stałopriorytetowym planowaniu wywłaszczającym** każda kolejka ma bezwzględne pierwszeństwo przed kolejkami o niższych priorytetach,
W **planowaniu ze sprzężeniem zwrotnym** możliwe jest przemieszczanie procesów między kolejkami.


HIGHEST PRIORITY
SYSTEM PROCESSES
INTERACTIVE PROCESSES
INTERACTIVE EDITING PROCESSES
BATCH PROCESSES
STUDENT PROCESSES
MULTILEVEL QUEUE SCHEDULING


QUANTUM = 100 MSEC
QUANTUM = 200 MSEC
FCFS
MULTILEVEL FEEDBACK QUEUES


Priorytet
31
Priorytety procesów czasu rzeczywistego
16
Priorytety zwykłych procesów interakcyjnych
13-14
Priory-
tet
bazowy
Zdarzenie
Czas

Szeregowanie procesów w systemie UNIX



The UNIX scheduler is based on a multilevel queue structure.

Im więcej czasu procesora dany proces zużywa, tym niższy staje się jego priorytet, to znaczy, że w algorytmie szeregowania procesów jest wykorzystywane negatywne sprzężenie zwrotne. Co sekundę system przelicza na nowo wartości priorytetów, według następującej reguły:

nowy priorytet = wartość bazowa + wykorzystanie procesora.

W ramach określonej kolejki jest wykorzystywany algorytm *Round Robin*. Współpracuje on z systemową procedurą *timeout*.

Zakleszczenie (deadlock)

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

$$C(P_j) = [C_1(P_j), C_2(P_j), \ldots, C_s(P_j)]^{\mathrm{T}}$$

oznaczający maksymalne żądanie zasobowe zadania P_j w dowolnej chwili czasu

- *wektor aktualnego przydziału* (current allocations),

$$A(P_j) = [A_1(P_j), A_2(P_j), \ldots, A_s(P_j)]^{\mathrm{T}}$$

- *wektor rang* zdefiniowany jako różnica między wektorami $\boldsymbol{C}$ i $\boldsymbol{A}$,

$$H(P_j) = C(P_j) - A(P_j)$$ // ile zadanie może jeszcze max. zażądać

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 $\boldsymbol{f}$, gdzie

$$f = [f_1, f_2, \ldots, f_s]^{\mathrm{T}}$$

gdzie

$$f_k = m_k - \sum_{j=1}^{n} A_k(P_j) \qquad k = 1,2,\ldots,s$$ wolne = istniejące – przydzielone

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

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

$$\rho^a(P_j) = [\rho_1^a(P_j), \rho_2^a(P_j), \ldots, \rho_s^a(P_j)]^{\mathrm{T}}$$

gdzie

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

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

$$\rho^r(P_j) = [\rho_1^r(P_j), \rho_2^r(P_j), \ldots, \rho_s^r(P_j)]^{\mathrm{T}}$$

gdzie

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

Łatwo wykazać:

$$\bigvee_k \bigvee_j \rho_k^a(P_j) \le H_k(P_j)$$ nie można zażądać więcej niż wynosi wartość wektora rang

$$\bigvee_k \bigvee_j \rho_k^r(P_j) \le A_k(P_j)$$ nie można zwolnić więcej niż się ma przydzielone

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

$$\bigvee_k \rho_k^a(P_j) \le f_k \qquad j = 1,2,\ldots,s$$ nie można zażądać więcej niż jest wolne

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:

$$\sum_{j=1}^{n} A_k(P_j) \le m_k \qquad k = 1,2,\ldots,s$$ nie można mieć więcej zasobów niż ich istnieje

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

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

Żeby stan był bezpieczny wystarczy , aby istniała nawet tylko 1 sekwencja bezp. !!!
Sekwencja bezpieczna – jeśli wektor rang dla P^j nie większy niż (liczba wolnych + liczba posiadanych przez P^1 ... P^{j-1})

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.

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

$$\exists_{\Omega \ne \Phi} \bigvee_{j \in \Omega} \exists_k \rho_k^a > f_k + \sum_{i \ne \Omega} A_k(P_i)$$

Istnieje niepusty zbiór taki , że dla każdego procesu z tego zbioru istnieje jakieś żądanie przez ten proces dodatkowych zasobów / zasobu (nieprzywłaszczal.) , które przekracza (liczbę wolnych + liczbę zasobów w dyspozycji procesów niezakleszczonych)

gdzie Ω jest zbiórem indeksów (lub zbiórem zadań)

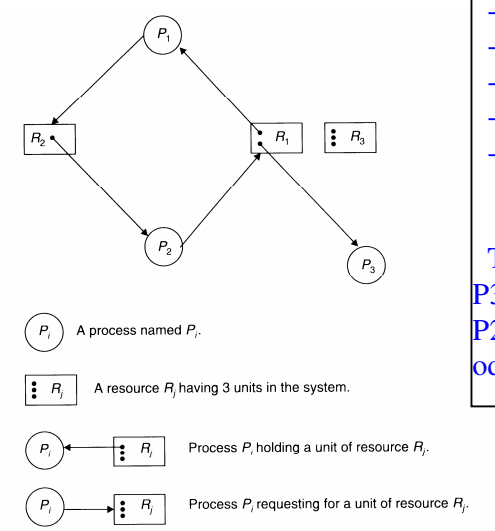
Mówimy, że system jest w *stanie zakleszczenia* (w systemie wystąpił *stan zakleszczenia*), jeżeli istnieje niepusty zbiór Ω zadań, które żądają przydziału dodatkowych zasobów nieprzywłaszczalnych będących aktualnie w dyspozycji innych zadań tego zbioru.

Innymi słowy, system jest w *stanie zakleszczenia*, jeżeli istnieje niepusty zbiór Ω zadań, których żądania przydziału dodatkowych zasobów nieprzywłaszczalnych nie mogą być spełnione nawet jesli wszystkie zadania nie należące do Ω zwolnią wszystkie zajmowane zasoby.

Jeżeli $\Omega \neq \Phi$, to zbiór ten nazywamy *zbiorem zadań zakleszczonych*.

Modele grafowe zaklaszczenia

Graf alokacji zasobów :

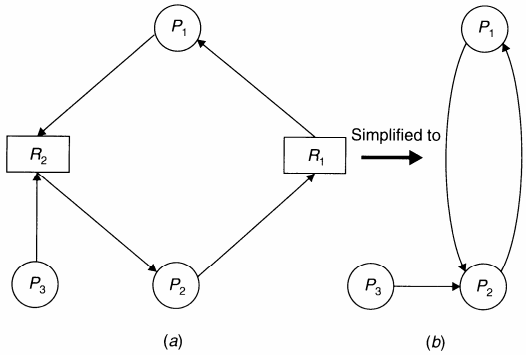


- P1 żąda jedynej jednostki R2
- P2 posiada jedyną jednostkę R2
- P2 żąda jednostki R1
- P1 posiada jedną z 2 jednostek R1
- P3 posiada drugą z 2 jednostek R1
- R3 ma 3 jednostki nieużywane i nikt go nie chce

Tutaj nie ma deadlocku (mimo cyklu w grafie) – P3 w końcu kiedyś odda 1 jednostkę R1 procesowi P2 i ten będzie się mógł wykonywać dalej i kiedyś odda R2 procesowi P1

Grafy oczekiwania (Wait-for Graph - WFG)

Z grafu aloakcji zasobów można uzyskać graf uproszczony przez usuniecie 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 (***wait-for-graph)*.**



Tutaj występuje deadlock – P3 czeka na R2 , który posiada P2 . P2 natomiast musi dostać R1 , który jest w posiadaniu P1 . P1 jednak czeka na otrzymanie R2 ....

A conversion from a resource allocation graph (a) to a WFG (b)

Warunki konieczne wystąpienia zakleszczenia

Warunkami koniecznymi wystąpienia zakleszczenia są:

1. ***Wzajemne wykluczanie*** (mutual exclusion condition),
   W każdej chwili zasób może być przydzielony co najwyżej jednemu zadaniu. Dostęp do zasobów musi być wyłączny (zwielokrotnienie zasobu – wtedy w.w. dla każdej jednostki)
2. ***Zachowywanie zasobu*** (wait for condition),
   Proces oczekujący na przydzielenie dodatkowych zasobów nie zwalnia zasobów będących aktualnie w jego dyspozycji.
3. ***Nieprzywłaszczalność*** (non preemption condition),
   Zasoby są nieprzywłaszczalne tzn. ich zwolnienie może być zainicjowane jedynie przez proces dysponujący w danej chwili zasobem. Nie ma np. timeoutu
4. ***Istnienie cyklu oczekiwań*** (circular wait condition),
   Występuje pewien cykl procesów z których każdy ubiega się o przydział dodatkowych zasobów będących w dyspozycji kolejnego procesu w cyklu. (Cykl w Wait-for-Graph)

Rozwiązania problemu zakleszczenia. Przeciwdziałanie zakleszczeniom.

1. ***Konstrukcje systemów immanentnie wolnych od zakleszczenia*** (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. Ograniczenie z góry liczby procesów wyk.
   równocześnie – jeśli mamy np. 5 zasobów to max. 4 procesy – zawsze „rezerwa" zasobu , który może być przydzielony w razie zażądania
2. ***Detekcja zakleszczenia i odtwarzanie stanu wolnego od zakleszczenia*** (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. Pełna swoboda w przydziale zasobów . Problemy – jak wykryć / odtworzyć ??? . W razie deadlocku usuwanie procesów (trzeba minimalizować koszty usunięcia)
3. ***Unikanie*** zakleszczenia (avoidance).
   W podejściu tym zakłada sie znajomość maksymalnych żądań zasobowych. Każda potencjalna tranzycja stanu jest sprawdzana i jeśli jej wykonanie prowadziłoby do stanu niebezpiecznego, to żądanie zasobowe nie jest w danej chwili realizowane. Spradwzanie każdego zadania , czy jego realizacja nie doprowadzi do stanu niebezppiecznego i jeśli tak
   to zawieszenie procesu . Każde zwolnienie zasobu – analiza proc. zawieszonych .
4. ***Zapobieganie*** zakleszczeniu (prevention)
   W ogólności podejście to polega na wyeliminowaniu możliwości zajścia jednego z warunków koniecznych zakleszczenia.

Detekcja zakleszczenia

Algorytm Habermana

1. Zainicjuj $\boldsymbol{D} := \{1,2,...,n\}$ i $\boldsymbol{f}$; D – zbiór procesów (na początku wszystkie mogą być potencjalnie zakleszczone)

2. Szukaj zadania o indeksie $j \in \boldsymbol{D}$ takiego, że
   $\rho^a(P_j) \le f$ czyli takiego , którego żądanie może być spełnione

3. Jeżeli zadanie takie nie istnieje, to zbiór zadań odpowiadający zbiorowi $\boldsymbol{D}$ jest zbiorem zadań zakleszczonych. Zakończ wykonywanie algorytmu.

4. W przeciwnym razie, podstaw:
   $D := D - \{j\}$; $f := f + A(P_j)$ założenie , że zadanie się skończy i zwolni zasoby

5. Jeżeli $D = \varnothing$, to zakończ wykonywanie algorytmu. W przeciwnym razie przejdź do kroku 2.

Algorytm mało efektywny – w najgorszym przypadku za każdym razem przeglądanie całej listy procesów (n razy n – $O(n^2)$)

Odtwarzanie stanu (recovery)

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

Wady podejścia detekcji i odtwarzania stanu.

1. Narzut wynikający z opóźnionego wykrycia stanu zakleszczenia - Algorytm sprawdzania może być uruchamiany tylko co jakis czas żeby inne procesy mogły się wykonywać , ale oznacza to , że w czasie m. dwoma kolejnymi uruchomieniami może dojść do deadlocku , i jeśli procesy zakleszczone dostaną CPU , to będą one zakleszczone aż do czasu kolejnego uruchomienia algorytmu w stystemie .
2. Narzut czasowy algorytmu detekcji i odtwarzania stanu
3. Utrata efektów dotychczasowego przetwarzania odrzuconego zadania. – zakleszczone procesy trzeba usunąć

Zalety podejścia detekcji o odtwarzania stanu.

1. Brak ograniczeń na współbieżność wykonywania zadań

Wysoki stopień wykorzystania zasobów – wszystkie zasoby wolne mogą być przydzielone jeśli procesy tego zażądają .

Algorytm Holt'a // c.d. poprzedniego podejścia

1 begin
2 initialize: $I_k = 1$, $k = 1,2,...,s$;
$c_i = s$, $i = 1,2,...,n$; $c_0 = n$;
3 LS: Y := false;
4 for $k = 1$ step 1 until s do
5 begin
6 while $E_{1,k,Ik} \le f_k \wedge I_k \le n$ 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 := 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 = true $\wedge$ $c_0 > 0$ then go to LS;
20 if Y = true then answer "no"
21 else answer "yes";
22 end.

Idea algorytmu – E – macierz trójwymiarowa zawierająca uporządkowaną tablicę żądań przydziału dodatkowych zasobów ρ – składa się z 2 2-wymiarowych tablic s·n „ jedna za drugą ” :

- w 1 tablicy – żądania dodatkowych przydziałów określonego zasobu posort. rosnąco
- w 2 tablicy – nr. procesu generującego żądanie w 1 tablicy

Przeglądanie tablicy i sprawdzanie – jeśli żądanie może być spełnione , to zmniejszenie licznika dodatkowych zasobów , które żąda proces – tak długo aż napotkamy na żądanie , które nie może być spełnione i wtedy kończymy dla danego zasobu . Jeśli przejdziemy do końca , to sprawdzamy , czy jakiś licznik = 0 (żądania skojarzone z licznikiem mogły być spełnione) i wtedy zmieniamy stan wektora f tak , jak w alg. Habermana . Jeśli liczniki dla wszystkich procesów mają wartość 0 , to nie dojdzie do zakleszczenia .

W algorytmie tym nie trzeba analizować tych procesów , które już były analizowane – złożoność O(n) , jeśli macierz uporządkowana , w przec. razie O(nlogn) – trzeba posortować

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 zawieś wykonywanie zadania. Sprawdzenie , czy wystąpi zakleszczenie , jeśli $\rho = H$
   (najgorszy możliwy przypadek) . Jeśli nie – żaden przydział nie doprowadzi do deadlocku
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 podejścia unikania.

1. Duży narzut czasowy wynikający z konieczności wykonywania algorytmu unikania przy każdym żądaniu przydziału dodatkowego zasobu i przy każdym żądaniu zwolnienia zasobu.
2. Mało realistyczne założenie o znajomości maksymalnych żądań zasobów.
3. Założenie, że liczba zasobów w systemie nie może maleć.

Zalety podejścia unikania.

1. Potencjalnie wyższy stopień wykorzystania zasobów niż w podejściu zapobiegania.
   (nie zawsze zasoby wolne mogą być przydzielone)

Podejście zapobiegania

Rozwiązania 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. Jeśli wszystkie zasoby przydzielone – proces nie może już nic zażądać i mamy go z głowy .

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 przyklad żądanie przydział zasobów 2 i 3. Trochę większa współbieżność niż w algorytmie wstępnego przydziału . Nigdy nie dojdzie do cyklu – żaden proces nie może zażądać zasobu już przydzielonego innemu procesowi

Rozwiązanie negujące zachowywanie zasobów (wait for condition) :

Algorytm Wait-Die

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 całosci odrzucane (abort) i zwalnie wszystkie posiadane zasoby.

Rozwiązanie dopuszczające przywłaszczalność . Nie wystąpi cykl żądań , bo tylko starszy czeka na młodszy (młodszy się wycofuje) .

Algorytm Wound-Wait

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.

Najstarsze zadanie nigdy nie jest wstrzymywane . Tutaj też nigdy nie będzie cyklu i występuje przywłaszczanie zasobów .

Wady podejścia zapobiegania.

1. Ograniczony stopień wykorzystania zasobów.

Zalety podejścia zapobiegania.

1. Prostota i mały narzut czasowy.