Programowanie Współbieżne – opracowanie

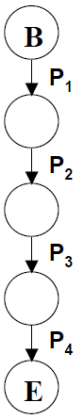
Opracował i przygotował Rolex 😊

Spis treści

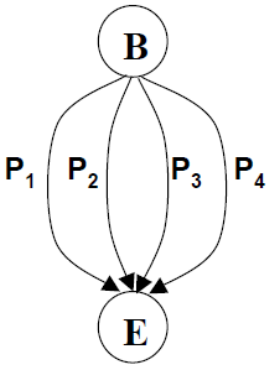
Wykład I:

1. Graf przepływu procesów:

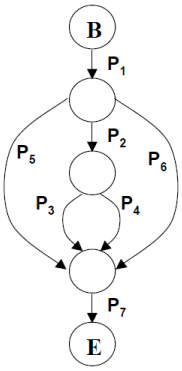
Graf przepływu procesów przedstawia zależności czasowe wykonywania procesów. Węzły tych grafów reprezentują dany moment, łuki – procesy. Dwa węzły połączone są jednym łukiem jeżeli istnieje proces, którego moment rozpoczęcia odpowiada pierwszemu węzłowi czasu, a moment zakończenia drugiemu. W praktyce oznacze to że między początkiem, a końcem następują kolejne istrukcje (procedury, procesy), które doprowadzaj stan początkowy do stanu końcowego, istnieją różne zależności między procesami:



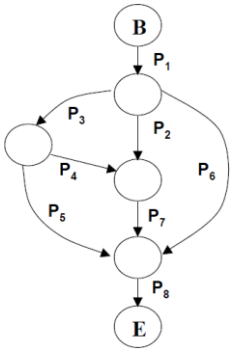
Szeregowe wykonywanie procesów



Równoległe wykonywanie procesów(GDZ)



Przykładowy graf przepływu procesów(GDZ)



Przykładowy graf nie jest GDZ--mieszany

Powyższe grafy są grafami skierowanymi DAG(Directed Acycle Graph).

Graf przepływu procesów jest (i powinien być) **dobrze zagnieżdżony (GDZ)**, 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 wykoanie równoległe i szeregowe procesów a i b.
P(a,b) := równległe
S(a,b) := szeregowe, sekwencyjne

Dla grafu z równoległym wykonywaniem procesów, procesy P1-P4 zaczynają się w tym samym czasie, natomiast E oznacza zakończenie ostatniego z procesów.

Dla przykłądowego grafu przepływu procesów procesy P2(a następnie P3 i P4), P5 i P6 wykonywane są równolegle, aż do punktu który je synchronizuje czyli wierzchołka przed procesem P7. Ten graf jest dobrze zagnieżdzony nie jesteśmy w stanie bardzije zminimalizować wysokości grafu.

Dla kolejnego grafu zauważyć możemy iż nie jest on dobrze zagnieżdżonym grafem mieszanym. Gdyż nie posiada jedynie procesów wykonywanych równolegle i szeregowo.

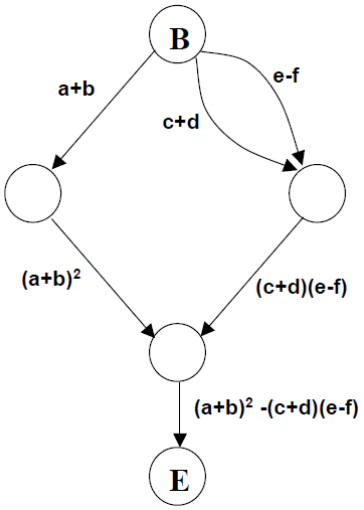
(Graf dobrze zagnieżdżony ma postać najbardziej zminimalizowanej funkcji w praktyce oznacza to, że ma najmniej procesów wykonywanych potrzebnych do przejścia momentu początkowego w końcowy, a zarazem ma najniższą wysokość – najmniej łuków między B, a E. Do dobrego zagnieżdżenia grafu niezbędne jest wykorzystanie równoległego wykonywania procesów -jeżeli

tylko jest taka możliwość. Taki graf posiada procesy wykonywane jedynie szeregowo lub równlogle, lub oba typy.)

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

3 warunki dobrze sporządzonego grafu:

- minimalna wysokośc dlanego procesu
- jak najmniejsza ilosć kroków
- poprawny wynik



2. Notacja „and” (Wirth):

Współbieżne wykonanie może być specyfikowane za pomocą operatora and, który łączy dwa wyrażenia wykonywane współbieżnie.

```
begin
        begin
                x1 := a + b;
                y1 := x1 * x1;
        end
        and
        begin
                        x3 := c + d ;
                and
                        x4 := e – f;

                y2:=x3 * x4;
        end
        z1 := y1 - y2;

end
```

Proces y2 zaczyna się dopiero gdy zakończy się proces x3 **ORAZ(AND)** x4.

3. Notacja „parbegin, parend” (Dijsktra):
   Wszystkie wyrażenia ujęte w nawiasy **parbegin, parend wykonywane są współbieżnie.**
   Instruckja po parend wykonuje się dopiero po wykonaniu wszystkich instrukcji wewnętrz klauzul.

```
begin
        parbegin
                begin
                        x1 := a + b;
                        y1 := x1 * x1;
                end
                begin
                        parbegin
                                x2 := c + d;
                                x3 := e – f;
                        parend
                        y2 := x2 * x3;
                end
        parend
        z1:= y1 – y2;
end
```

4. Notacja „join, fork, quit” (Conway):
   Fork – rozgałęzienie
   Join – połączenie
   quit – zakończenie procesu

   fork w – oznacza że proces w którym wystąpiła ta instrukcja będzie dalej wykonywany współbieżnie z procesem identyfikowanym przez etykietę **w**.

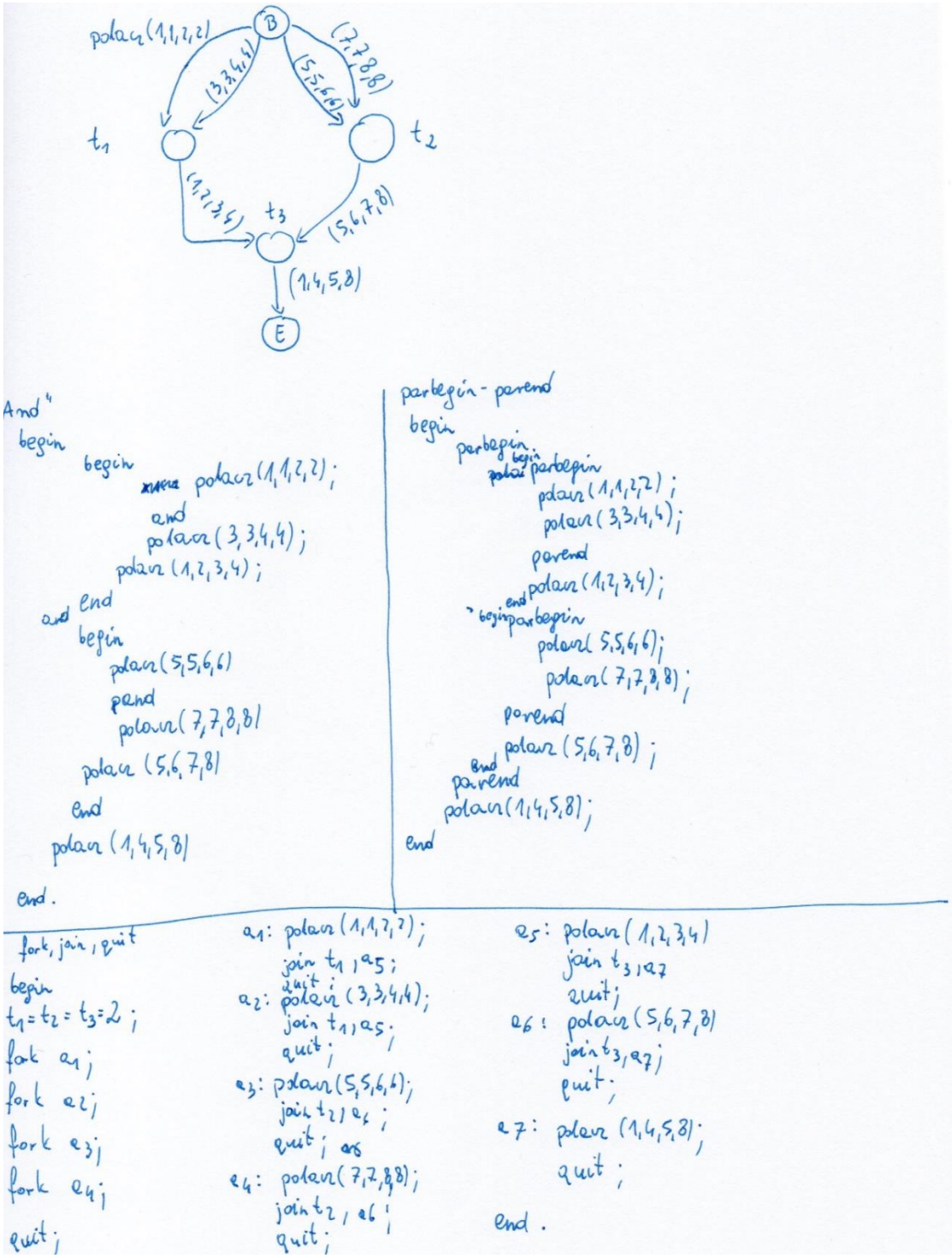
   join t,w – t – licznik ; w – etykieta:
   t := t -1;
   if t = 0 then goto w

   Sekwencja powyższych instrukcji wykonywana jest **atomow** tzn. że jest niepodzielna, zatem instrukcja wykonywana jest w całości albo wcale.

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

Licznik ustawiamy na wartość równą ilości procesów (łuków) wykonywanych przez jedną z współbieżnych gałęzi, aż do punktu synchronizacji danych współbieżnych procesów. Np. wartość t2 := 2, gdyż wierzchołek synchronizujący: c + d i e – f może rozpocząć dalsze instrukcje dopiero po wykonaniu tych **DWÓCH** instrukcji.

5. Przykład notacji i grafu- sortowanie przez scalanie:
   Procedura połącz(x1,x2,y1,y2) – łączy dwa uporządkowane ciagi x i y:
   dla:
   -polacz(1,1,2,2)
   -polacz(3,3,4,4)
   -polacz(5,5,6,6)
   -polacz(7,7,8,8)
   -polacz(1,2,3,4)
   -polacz(5,6,7,8)
   -polacz(1,4,5,8)

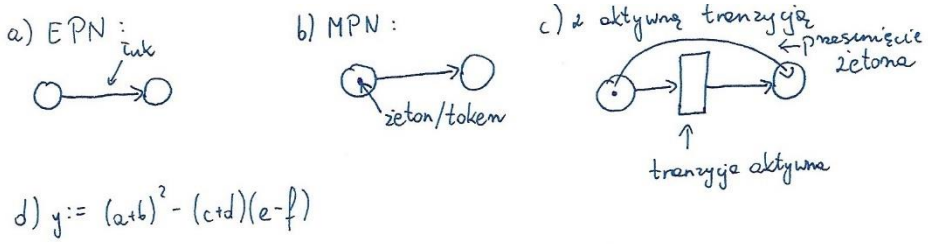


Wykład II: Sieci Petriego:

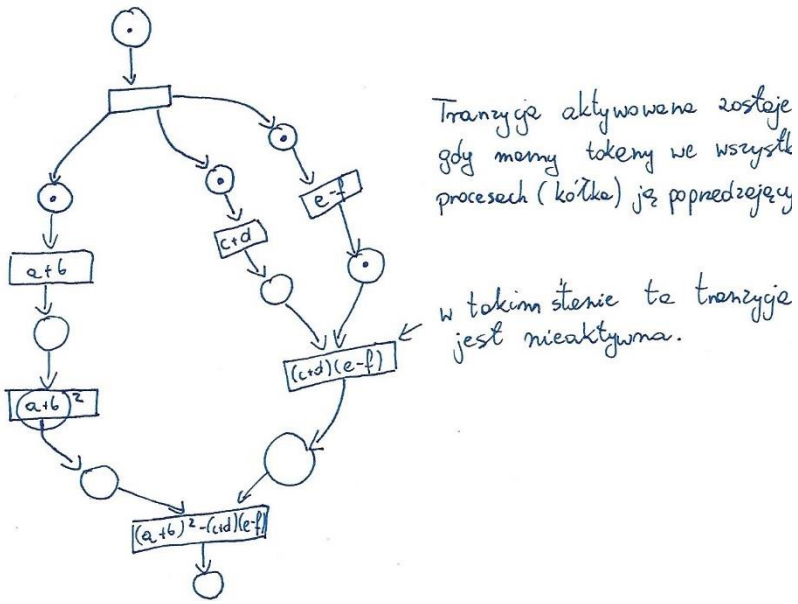
1. Elementarna sieć Petriego -> uporządkowana trójka:
   EPN – (P, T, A)
   P-> niepusty zbiór miejsc
   T -> niepusty zbiór przejść P∩T = Ø
2. Znakowana sieć Petriego -> uporządkowana trójka:
   MPN = (P, T, A, M_o)
   EPN = (P,T,A)
   M_o: P ->Z_+ (znakowanie początkowe)
   Znakowanie aktywne: każde z miejsc wejściowych zawiera co najmniej jednego znacznika, Nowe znakowanie:

$$M'(p) = \begin{cases} M(p) - 1 & p \in In(t) \backslash Out(t) \\ M(p) + 1 & p \in Out(t) \backslash In(t) \\ M(p) & pozostałe \end{cases}$$

   Znakowana sieć petriego z aktywną tranzycją, kolejny proces wykonywany jest dopiero po dotarciu sygnału do tranzycji aktywnej.
3. Etykietowana sieć Petriego:
   LPN = (P,T,A,l, M_o)
   l: T ->E jest etykietą, a E skończonym zbiorem etykiet.
4. Inne typy sieci Petriego to:
   a. Uogolnione
   b. Priorytetowe
   c. Czasowe
   d. Kolorowane
   e. Czasu Rzeczywistego
5. Przykłady sieci oraz rozwiązanie zadania y := $(a + b)^2 - (c + d)(e - f)$:



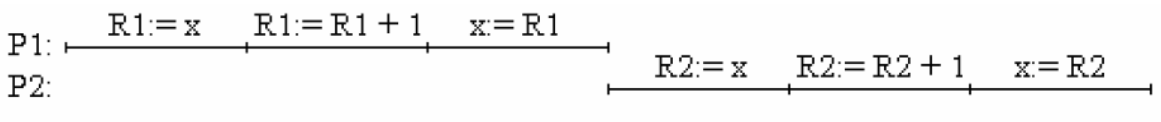
d) $y := (a+b)^2 - (c+d)(e-f)$



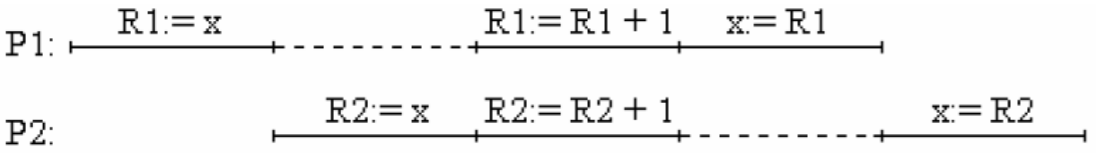
Wykład III: Synchronizacja cz. I:

1. Problem wzajemnego wykluczania:
   Rozważamy system w którym wzajemnie wykonywane są procesy od 1 do n.
   Zakładamy że nie są znane względne prędkości wykonywania tych procesów, tzn. że liczba instrukcji wykonywanych przez poszczególne procesory w jednostce czasu może być dowolna.
   Przyjmujemy ponadto, że procesy te mają dostęp do wspólnych zasobów.
   Rozważmy dla przykładu dwa procesy:
   P1: x := x + 1;
   P2: x := x + 1;
   Załóżmy że każde uaktualnienie składa się z trzech faz:
   - R := x; // pobranie wartości zmiennej x do rejestru wewnętrznego procesora
   - R := R + 1; //inkrementacja zawartości wewnętrznej rejestru procesora
   - x := R; // zapisanie wartości rejestru do zmiennej x

   Rozważmy możliwe sekwencje wykonania takich współbieżnych procesów:

   a.
   

   OK. -> w rezultacie otrzymujemy: x := x + 2;

   b.
   

   ŹLE. -> w rezultacie otrzymamy: x := x +1, gdyż w do różnych rejestrów w fazie początkowej zapisane zastały początkowe wartosci x a następnie wykonane + 1.

2. Sformułowanie 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.
   Założenia:
   a. Zapis i odczyt wspólnych danych jest operacją niepodzielną, a próba jednoczenych zapisów lub odczytów realizowana jest sekwencyjnie w nieznanej kolejności.
   b. Sekcje krytyczne nie mają priorytetu.
   c. Względne prędkości wykonywania procesów są nieznane.
   d. Proces może zostać zawieszony poza sekcją krytyczną.
   e. Proccesy realizujące instrukcje poza sekcją krytyczną nie mogą uniemożliwiać innym procesom wejścia do sekcji krytycznej.
   f. Procesy powinny uzyskać dostęp do sekcji krytycznej w skończonym czasie.

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

   Sekcja krytyczna jest zasobem współdzielonym przez wiele procesów ubiegajacych się o dostęp do tego wspóldzielonego medium.

Pierwsza próba rozwiązania problemu wzajemnego wykluczania:

```
1.  program VERSIONONE;
2.  var processNumber: INTEGER;

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

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

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

Problemy powyższego algorytmu:

- Rozwiązanie gwarantuje wzajemne wykluczanie, ale procesy mogą wykonywać swoje sekcje krytyczne tylko naprzemiennie. Wykonanie sekcji krytycznej pierwszego procesu powoduje przypisanie processNumber := 2; co zawiesza proces pierwszy i w przypadku przełączenia kontekstów zezwala na dostęp do sekcji krytycznej procesu 2.
- Jeśli 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. wykonyważ otherstuff – 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 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. Do processNumber prrzypisana została wartosć 2 co sprawiło zawieszenie procesu 1, a skończenie procesu 2.

Silny warunek postępu:

Warunek postępu oznacza, że jeśli nie ma żadnego procesu w sekcji krytycznej, a są procesy w sekcji wejściowej, to jeden z nich w skończonym czasie (po zajściu skończonej liczby zdarzeń w systemie) wejdzie do sekcji krytycznej. Warunek postępu nie gwarantuje, że konkretny proces wejdzie do sekcji krytycznej. Może się zdarzyć tak, że w momencie przejścia do sekcji wyjściowej, proces opuszczający sekcję krytyczną do sygnał do wejścia procesom oczekującym w sekcji wejściowej, w wyniku którego jakiś proces wejdzie do sekcji krytycznej, a inny (przy zachowaniu warunku bezpieczeństwa) oczywiście nie wejdzie. Przy kolejnym sygnale ze strony procesu wychodzącego z sekcji krytycznej pominięty poprzednio proces ponownie może nie uzyskać prawa wejścia, podczas gdy inny proces wykonujący swoją sekcję wejściową prawo takie dostanie. Sytuacja może się powtarzać w nieskończoność. Postęp jest zachowany bo jakiś proces wchodzi do sekcji krytycznej, ale istnieje proces permanentnie pomijany.

Druga próba rozwiązania problemu wzajemnego wykluczania:

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

3.  procedure PROCESSONE;
4.  begin
5.    while True do
6.      begin
7.        while P2inside do;
8.        P1inside:=True;
9.        criticalSectionOne;
10.       P1inside:=False;
11.       otherStaffOne;
12.     end;
13. end;
```

```
14. procedure PROCESSTWO;
15. begin
16.   while True do
17.     begin
18.       while P1inside do;
19.       P2inside:=True;
20.       criticalSectionTwo;
21.       P2inside:=False;
22.       otherStaffTwo;
23.     end;
24. end;

25. begin
26.   P1inside:=False;
27.   P2inside:=False;
28.   parbegin
29.       PROCESSONE;
30.       PROCESSTWO;
31.   parend;
32. end.
```

Problemy powyższego algorytmu:

- Sekcja krytyczna nie jest zabezpieczona, oba procesy mają do niej dostęp, oba procesy na raz mogą wejść do sekcji krytycznej (chociażby na samym początku gdy obie flagi ustawione zostały na false) może to nastąpić w następującej sekwencji instukcji i nastąpi przełączenie kontekstów:
    - ProcessOne rozpocznie pętlę **while**
    - ProcessTwo rozpoczni pętlę **while**
- To rozwiązanie uzależnione jest od przebiegów czasowych obu procesów, gdyż może dojśc do sytuacji zagłodzenia procesu.

Trzecia próba rozwiązania problemu wzajemnego wykluczania:

```
1.  program VERSIONTHREE;
2.  var P1WantsToEnter: BOOLEAN;
3.  var P2WantsToEnter: BOOLEAN;
4.  procedure PROCESSONE;
5.  begin
6.    while True do
7.      begin
8.        P1WantsToEnter:=True;
9.        while P2WantsToEnter do;
10.       criticalSectionOne;
11.       P1WantsToEnter:=False;
12.       otherStuffOne;
13.     end;
14. end;
```

```
15. procedure PROCESSTWO;
16. begin
17.   while True do
18.     begin
19.       P2WantsToEnter:=True;
20.       while P1WantsToEnter do;
21.       criticalSectionTwo;
22.       P2WantsToEnter:=False;
23.       otherStuffTwo;
24.     end;
25. end;
26. begin
27.   P1WantsToEnter:=False;
28.   P2WantsToEnter:=False;
29.   parbegin
30.       PROCESSONE;
31.       PROCESSTWO;
32.   parend;
33. end.
```

8. sygnalizuje gotowość do wejścia
9. Sprawdzanie czy drugi też nie jest gotowy do wejścia
11. Po sekcji krytycznej, jeśli drugi czeka, to ma zezwolenie na dostęp do sekcji krytycznej

Problemy powyższego algorytmu:

- Delikatnie zmieniona wersja poprzedniego rozwiazania- różni się zmianą kolejności występowania instrukcji sprawdzania wartości zmiennej oraz instrukcji przypisania. Również zależy od przebiegów czasowych.
- Jeśli zdarzy się następująca sekwensja instrukcji przez przełączenie kontekstów:
    - ProcessOne ustala P1WantsToEnter na True
    - ProcessTwo ustala P2WantsToEnter na True

  W takim przypadku pętla **while** w obu przypadkach będzie nieskończona, a procesy się zawieszą.

Czwarta próba rozwiązania problemu wzajemnego wykluczania:

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

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

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

8. ProcessOne chce wejść
9. Jeśli także drugi chce wejść to następują instrukcje czekania z pętli
11. ProcessOne ustępuje dostępu do sekcji krytycznej
12. Pierwszy oczekuje(zawiesza się) przez jakiś czas (random)
13. ProcessOne kolejny raz chce wejść, będzie mógł wejść jeżeli w tym czasie ProcessTwo wyszedł z sekcji krytyczznej i ustawił P2WantsToEnter na False, dzięki czemu ProcessOne wyjdzie z pętli i będize miał bezpośredni dostęp do sekcji krytycznej. Jeżeli jednak w tym czasie ProcessTwo nie wyjdzie z sekcji krytycznej, pętla wykona się kolejny raz

Problemy powyższego algorytmu:

- Algorytm zasadniczo poprawny, może się jednak zdarzyć taka (mało prawdopodobna) sytuacja, że ProcessOne będzie czekał na wejście do sekcji krytycznej w pętlę while (ProcessTwo w tym czasie będzie w sekcji krytycznej), wykona delay(P1), a w tym czasie P2 wyjdzie z sekcji krytycznej , ale znowu zdąży do niej wejść (bo P1wantstoenter = false w czasie gdy P1 czeka), ten rzadko występujący przypadek może jednak doprowadzać do zagłodzenia jednego z procesów w dostępie do procesora.
- Ponadto, jeśli dojdzie do tego że P1WantsToEnter = true i jednocześnie P2wantsToEnter = true, to jeżeli oba odczekają w delay odpowiednią ilosć czasu, może dojść do tego że znowu P1WantsToEnter = true i P2WantsToEnter = true, a to również prowadzi do niedopuszczenia procesów do procesora i jest również małoprawdopodobne.

Algorytm ten jest używany w duzej liczbie routerów i sieci.

Algorytm Dekkera:

```
1.  program DekkerAlgorithm;
2.  var favoredProcess: enum (First, Second);
3.  var P1WantsToEnter, P2WantsToEnter: BOOLEAN;

4.  procedure ProcessOne;
5.  begin
6.    while True do
7.      begin
8.        P1WantsToEnter:=True;
9.        while P2WantsToEnter do
10.         if favoredProcess=Second then
11.           begin
12.             P1WantsToEnter:=False;
13.             while favoredProcess=Second do;
14.             P1WantsToEnter:=True;
15.           end;
16.       criticalSectionOne;
17.       favoredProcess:=Second;
18.       P1WantsToEnter:=False;
19.       otherStuffOne;
20.     end;
21. end;
```

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

8.P1 chce wejść do sekcji krytycznej
10. Pętla wykonuje się tak długo jak P2 chce wejść i jest uprzywilejowany
11. P1 rezygnuje z dostępu do sekcji krytycznej
12. P1 czeka tak długo jak drugi(P2) jest faworyzowany

13. Jeśli P2 nie jest już fawryzowany (wyszedł z sekcji krytycznej) to pętla while kończy się, a P1 znowu ubiega się o dostęp do sekcji krytycznej.
17. P1 rezygnuje z faworyzacji.

Opis działania powyższego algorytmu:

- Algorytm poprwny- nie zapętla się, oba procesy nie mogą mieć dostępu do sekcji krytyczznej jednocześnie. Z warunku pętli wynika, że P_i może wejść do sekcji krytycznej wtedy gdy favoredProcess = i, lub PjwantsToEnter = false; P1 ma dotsęp gdy favoredProcess = First lub P2WantsToEnter = false; A więc aby oba procesy weszły do sekcji krytyczznej musiałoby nastapić:
  - favoredProcess = i = j -> niemożliwe do zajścia jednocześnie
  - PiWantsToEnter , PjWantsToEnter = false, natomiast gdy proces ma być w sekwencji jego WantsToEnter = true; (patrz. 8 i 26)
  - favoredProcess = i ale PiWantToEnter = false. Ale jeśli PiWantsToEnter=false, to oznacza że favoredProcess = j, a PiWantsToEnter zmieni się na true dopiero jeśli favoredProcess zmieni się na i.
  - favoredProcess = j oraz PjWantsToEnter =false; jw.
  - Ponadto jeśli jeden z procesów np. P_i jest w sekcji krytycznej, to PiWantsToEnter = true i favoredProcess = i, wiec Pj nie wejdzie do sekcji krytycznej.
  - Jedyne miejsce gdzie może powstać pętla nieskończona-> pętla while trwa np. dla Pi tak długo, aż nie zajdzie favoredProcess = i lub też nei będzie PjWantsToEnter = false . Jeżeli

Pj nie jest gotowy na wejście do sekcji krytycznej, to favoredProcess = i oraz PjWantsToEner = false i Pi wejdzie do sekcji krytycznej. Jeśli oba są gotowe i czekają na wejście, to oba wantsToEnter = true, ale favoredProcess przyjmuje jedną wartosć i któryś z procesów i/j dostanie dostęp do sekcji krytycznej.
Jeżeli jakiś proces np. Pi jest w sekcji krytycznej to po wyjściu z niej zmieni PiWantsToEnter na false i favoredProcess na j, więc Pj będzie mógł wejść do sekcji krytycznej.

Algorytm Dijkstry dla n procesów:

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

4. 0 – nie chce wejśćc; 1 – chce wejść; 2 – jest obsługiwany/został wybrany
11-19 -> 1 część algorytmu, czekanie aż proces zostanie wybrany
11. Pi chce zostać wybrany = 1;
13. Pętla while działa tak dlugo aż Pi nie jest wybrany
15. Sprawdzanie flagi wybranego procesu
16. Jeśli flaga Pi (wybranego procesu) jest 0 (nie chce) – bo Pi wyszedł z sekcji krytycznej to Pi staje się wybrany – może to jednocześnie zrobić wiele procesów więc neikoniecznie on będzie pod zmienną turn
21-27 -> 2 część algorytmu, sprawdzenie czy inne procesy nie zostały również wybrane, gdyż mogło do tego dojść równocześnie, w czasie wykonywania 2 części algorytmu inny proces nie może być już wybrany bo flag[turn] = 2; (turn -> wybrany proces)
20. Pi ustawia się jako 2 czyli wybrany proces
25. Jeśli inne procesy są wybrane to skaczemy do flagi L z lini 11.

Opis działania algorytmu:

- Nigdy 2 procesy nie wejdą na raz do sekcji krytycznej- jeśli miałoby się tak stać, dla 2 procesów na raz musiałaby istnieć flaga: flag = 2, co jest niemożliwe, bo nawet gdy w sekcji pierwszej zostanie wybrany więcej niż jeden proces to tylko jeden z nich przejdzie do sekcji drugiej.
- Ponadto jeśli jeden proces jest w sekcji, to jego flag = 2, a więc inne zatrzymają się w cz. 1 lub 2.
- Jedyne miejsce gdzie mogłaby powstać pętla nieskończona- pętla while, trwa ona tak długo, aż wreszcie któryś z procesów odkryje, że ten który był w sekcji wyszedł i ustawił swoją flage na 0. Tak więc jeżeli jakiś wyjdzie z sekcji krytycznej, to inny przestanie czekać (choć może również dojść do tego, że proces, który wyjdzie z sekcji krytycznej, zmieni sobie flagę na 0, wykona resztę i zdąży znów zmienić flagę na 1 zanim inne to zauważą – wtedy on wejdzie do sekcji gdyż nadal jest procesem wybranym, **czyli i tak nie dojdzie do zapętlenia**).
- Za pierwszym razem wchodzi do sekcji krytycznej ten, na który wskazuje początkowa wartość turn.

Algorytm Petersona dla 2 procesów:

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

4. inicjalizując false
5. 0 lub 1

11.P1 gotowy do wejścia
12.zakłada że P2 też chce wejść do sekcji krytycznej
14. Sprawdza czy P2 chce wejść do sekcji
15. Sprawdza gotowość P2
16.pętla wykonuje się aż P2 nie zmieni turn na i lub też flag[1 – i] na false

Opis działania algorytmu:

- Algorytm poprawny nie spowoduje ani zapętlenia ani równoczesnego wejścia do sekcji krytycznej dwóch procesów.
- Jedynym miejscem o podwyższonym ryzykunzapętlenia jest instrukcja **repeat-** wykonywana dla procesu Pi tak długo, aż nie zajdzie turn = i lub flag[j] = false. Jeśli Pj nie jest gotowy do wejścia do sekcji, to flag[j] = false i do sekcji krytycznej może wejść Pi.
  Jeśli Pj spowodował, że flaga[j] = true oraz też wykonuje pętlę, to jeśli turn = i, to Pi wejdzie do sekcji krytycznej, a jeśli turn = j, to Pj wejdzie do sekcji.
  Jednak, kiedy Pj wyjdzie to zmini flag[j] na false i Pi będzie mógł wejść. Jeśli Pj zmieni flag[j] na true, to musi także zmienić turn na i , a w tej sytuacji Pi, który oczekując w pętli na wejscie nie zmienia wartości turn, wejdzie do sekcji krytycznej.
- Każdy proces Pi wchodzi do sekcji krytycznej tylko wtedy, gdy albo flag[j] = false, albo turn = i. Poza tym, gdy oba procesy miały jednocześnie być w sekcji, to spełnione byłoby flag[0] = flag[1] = true – każdy Pi przypiuje flag[i] = true przed swoim wejściem do sekcji krytycznej. W takim razie oba byłyby jednocześnie w sekcji krytycznej, musiałoby jednocześnie zachodzić turn = i = j, co nie jest możliwe. Ponadto podczas gdy Pj jest w sekcji, to flag[j] = true i jeśli Pi 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 Pj wyjdzie z sekcji. Tak więc zawsze tylko jeden proces będzie znajdował się w sekcji krytycznej.

Algorytm Petersona dla n procesów:

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

4. Początkowo przypisywana jest wartość 0,
5. Zapisuje kto ostatni doszedł do danego cyklu pętli.

12.Proces i-ty jest w k-tym przebiegu pętli for
13. Do k-tego przebiegu wszedł jako ostatni proces i-ty.
15. Jeśli jakiś proces wszedł po naszym – można przejść dalej
16. Sprawdzenie flag wszystkich innych procesów
21. Jeśli jakiś proces zaszedł dalej to repeat od nowa

23. Pętla repeat until działa dopólki turn[k]≠i albo dla każdego l = 1..n, l≠i: flag[l] < k, czyli do czasu aż albo inny proces wszedł do samego cyklu po Pi albo wszystkie są we wcześniejszych cyklach niż Pi

Opis działania powyższego algorytmu:

- Jeśli proces jest sam na jakimś etapie pętli for, to przejdzie dalej, jeśli wszystkie inne są na wcześniejszych etapach pętli. Jeśli jest ich wiecej 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 krytycznej max 1.
  „Najbardziej wysunięty" proces (o maxymalnej fladze[flag]) będzie zawsze szedł dalej, ponadto jeżeli na jakimś etapie jest więcej niż 1 proces, to wszystkie one z wyjątkiem tego 1 przejdą zawsze dalej więc algorytm nigdy się nei zapętli.

Działanie pętli repeat-unitil, pętla się przerywa jeżeli warunek w until jest PRAWDZIWY(TRUE).

Algorytm Lamporta dla n procesów:

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

10. dla procesu Pi

12. W trakcie wybierania swojego numeru

20. Przyznaje sobie największy numer ze wszystkich

21. Skończył wybieranie

21. Dla wszystkich innych procesów niż i wykonuje pętle

25. Tak długo aż jakiś k-ty proces nie wybrał

27. Zaczyna się jeżeli już wybrał

28. Instrukcja wykonuje się tak długo aż numerek nie jest mniejszy (jest większy lub równy) albo ktoś wyszedł z sekcji krytycznej.

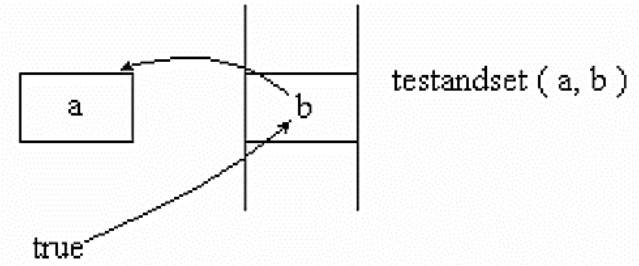
29. **$mine_i < other_i$ lub $mine_i = other_i$ i i < k**

Opis działania algorytmu:

- Każdy proces przed wejściem dostaje swój numer. Obsługę rozpoczyna się od klienta z najmniejszym numerkiem. Jeżeli Pi i Pj mają ten sam numerek pierwszy będzie obsłużony ten o wcześniejszej nazwie- nazwy procesów sa jednoznaczne i całkowicie uporządkowane. Jeżeli proces Pi jest w sekcji, to dla każdego k≠i : jeśli Pk ma już wybrany swój numer to (num[i],i)<(num[k],k) wynika to z warunku drugiego pętli repeat – przepuści ona do sekcji krytycznej proces o niższym numerze z istniejących, a kolejne zgłaszające żadania dostępu do sekcji krytycznej będą miały wyższe numery.
- Algorytm ten zapewnia wzajemne wykluczanie jeśli proces Pi jest w sekcji, a inny np. Pk będzie chciał wejśc to odkryje że num[i]≠0 oraz (num[i],i)<(num[k],k). Będzie zatem czekał w pętli repeat na wyjście Pi z sekcji krytycznej.
  Nie zapętli się, bo zawsze jakiś proces ma min. Numer i on wejdzie do sekcji krytycznej pierwszy, potem wejdzie kolejny z najmniejszym numerem z pozostałych procesów.

Wykład IV: Synchronizacja cz. II:

1. Instrukcja **testandset**:
   Załóożmy że w systemie jest instrukcja **testandset(a , b);** która w sposób atomowy (czyli wykonuje się w całości albo wcale - niepodzielnie) dokonuje odczytu wartości zmiennej **b,** zapamiętania wartości tej zmiennej w zmiennej **a** oraz przypisania zmiennej **b** wartości **True.**




```
testandset(a , b){
        a := b;
        b := True;
}
nottestandset(a,b){
        a := b;
        b := False;
}
```

Użycie powyższej instrukcji jest skalowalne (dla n procesów), ale problem powstaje w systemach wieloprocesowrowych – **tylko w systemach 1 procesorowych instrukcja testandset jest atomowa.** (w pozostałych systemach nie jest)

Przykład:

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

3.  procedure ProcessOne;
4.  var oneCannotEnter: BOOLEAN;
5.  begin
6.    while True do
7.     begin
8.       oneCannotEnter:=True;
9.       while oneCannotEnter do
10.       testandset
            (oneCannotEnter,active);
11.      criticalSectionOne;
12.      active:=False;
13.      otherStuffOne;
14.    end
15. end;
```

```
16. procedure ProcessTwo;
17. var twoCannotEnter: BOOLEAN;
18. begin
19.   while True do
20.    begin
21.      twoCannotEnter:=True;
22.      while twoCannotEnter do
23.       testandset
            (twoCannotEnter, active);
24.      criticalSectionTwo;
25.      active:=False;
26.      otherStuffTwo;
27.    end
28. end;

29. begin
30.   active:=False;
31.   parbegin
32.       ProcessOne;
33.       ProcessTwo;
34.   parend
35. end.
```

8. na starcie zakładamy że nie może wejśc do sekcji krytycznej proces pierwszy(21. Drugi)
9. Pętla while trwa tak długo dopóty, dopóki active nie będzie false, a po instrukcji testandset onecannotenter również nie będzie false, co spowoduje przerwanie pętli i wejście do sekcji krytycznej (pętla działa tak długo jak aktive = true, a więc tak długo jak proces oczekuje na prawo wejścia do sekcji krytycznej)
12. Active ustawiamy na false a więc inny proces może ubiegać się o dostęp do sekcji krytycznej.

2. Semafory:
   Semaforem nazywamy zmienną chronioną, na ogół będącą nieujemną zmienną typu integer, do której dostęp (zapis i odczyt) możliwy jest poprzez wywołanie specjalnych funkcji (operacji semaforowych) dostępu i inicjacji. Wyróżniamy semafory:
   - Binarne – przyjmują wartośći 0 lub 1. (mutex)
   - Ogólne (licznikowe) mogące przyjąć nieujemną wartość całkowito liczbową.

3. Operacje P(S) i V(S):
   Operacje P i V pierwszy raz opisał Dijkstra:
   - P – pochodzi od holenderskiego *probiren*(**testuj**) (WAIT)
   - V – pochodzi od *verhogen*(**inkrementuj**)(SIGNAL)

   Operacja P na semaforze S działa w sposób następujący:
   if S > 0
   then S := S – 1
   else (wait on S)

   Stosowane w trial section; procces, który wywołał tą instrukcję jest zawieszany (włączony do zbioru zadań skojarzonych z tym semaforem)

   Operacaja V na semaforze S działa w poniższy sposób:
   if (one or more processes are waiting on S)
   then (let one of these processes proceed)
   else S := S + 1 // jeżeli zbiór procesów dla semafora S jest pusty następuje zwiększenie S

   //Nie ma kolejności wykonywanie procesów istnieje iniebezpieczeństwo zbyt długiego oczekiwania przez proces na procesor.

   Z semaforem skojarzony jest zbiór procesów, które mogą byćc do tego zbioru dołączane/odłączane.
   Rozwiązanie to jest w pełni skalowalne dla n procesów. Operacje P i V są atomowe.

```
1.  program SEMAPHOREEXAMPLE;
2.  var active: SEMAPHORE;
3.  procedure PROCESSONE;
4.  begin
5.    while True do
6.      begin
7.        P(active);
8.        criticalSectionOne;
9.        V(active);
10.       otherStuffOne;
11.     end
12. end;
13. begin
14.     semaphore_initialize(active,1);
15.     parbegin
16.         PROCESSONE;
17.         ...
18.         PROCESSNTH;
19.     parend
20. end.
```

Opis działania programu:
Na początku ustawiane jest active = 1. Pierwszy proces przed wejściem do sekcji krytyczznej wykonuje operację P(active), co powoduje zmniejszenie active do 0 – tak więc następne procesy będą czekały. Po wyjściu z sekcji krytycznej proces pierwszy wykonuje instrukcję V(active) – jeśli jakieś procesy czekają, to activve dalej będzie miało wartość 0, ale kolejny proces uzyska dostęp do sekcji krytycznej, a zarazem pozostałe procesy nie będą miały dostępu do sekcji krytycznej. W przypadku braku oczekujących procesów active jest zwiększane o 1 (= 1) i jeżeli jakiś proces będzize chciał wejść do sekcji krytycznej to wykona P i wyzeruje active. Itd.

4. Problem producenta – konsumenta: **Przy założeniu że zapis i odczyt z bufora nie mogą być jednoczesne – konieczna jest sekcja krytyczna.**

```
1.  program PRODUCERCONSUMER;
2.  var emptyBuffers, fullBuffers, active: SEMAPHORE;
3.  procedure PRODUCER;
4.  begin
5.    while True do
6.     begin
7.       produceNextRecord;
8.       P(emptyBuffers);
9.       P(active);
10.      addToBuffer;
11.      V(active);
12.      V(fullBuffers);
13.    end
14. end;
```

```
15. procedure CONSUMER;
16. begin
17.   while True do
18.    begin
19.      P(fullBuffers);
20.      P(active);
21.      takeFromBuffer;
22.      V(active);
23.      V(emptyBuffers);
24.      processNextRecord;
25.    end
26. end;
27. begin
28.   semaphore_initialize(active, 1);
29.   semaphore_initialize(emptyBuffers, N);
30.   semaphore_initialize(fullBuffers, 0);
31.   parbegin
32.       PRODUCER;
33.       CONSUMER;
34.   parend
35. end.
```

2. określenie liczby:
pustych buforów, pełnychbuforów, aktywnegoSemafora(zabezpiecza, aby dostęp do dodania lub wzięcia bufora miał tylko producent lub konsument, aby dostęp do bufora nie był jednoczesny.

8. Jeśli nie ma pustych buforów to CZEKA (WAIT) ; jeśli są- zmniejszenie ich liczby o 1.

9.zabezpieczenie żeby w tym samym czasie było tylko dodawanie lub pobieranie z bufora, a nie oba na raz.

12. Jeśli konsument czeka na zapełnienie – może już pobrać i działać dalej/ w innym przypadku zwiększenie liczby zapełnionych.

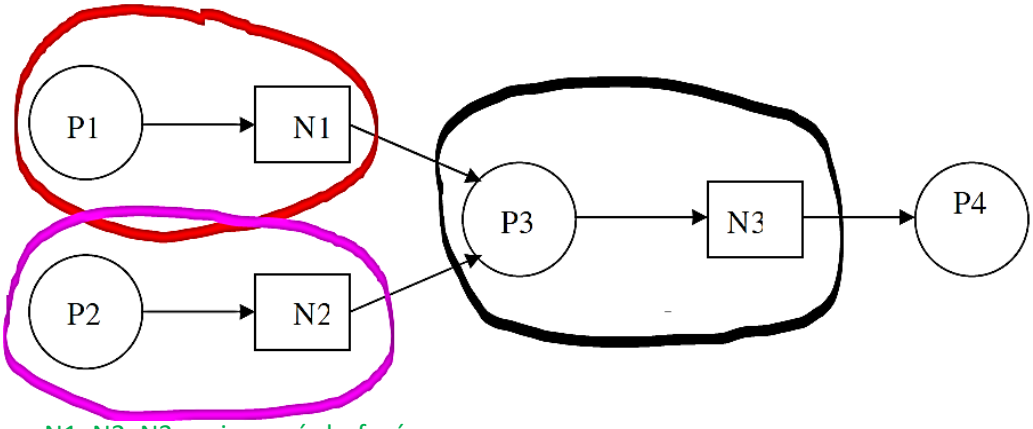
19. Jeśli nie ma pełnych- niech czekają; jeśli są pełne – zmniejszenie ich liczby o 1.

23. Jeśli producent czeka na opóźnienie- może już dodać i działać dalej/ w innym wypadku zwiększenie liczby pustych buforów o 1.

29. Początkowo N pustych buforów.

30. Początkowo 0 pełnych buforów.

5. Inny przykład problemu producent-konsument (dla dwóch producentów i jednego konsumenta):



N1, N2, N3- pojemnośc buforów.
P1,P2 – zwykli producenci. P4 – zwykły konsument.
P3 – zarówno konsument jak i producent, aby produkować coś dla P4 musi otrzymać coś zarówno od P1 jak i P2. Implementacja P3:
eb – emptybuffer; fb-fullbuffer

```
procedure P3:
  begin
    while True do
      begin
        parbegin //równoległe pobieranie buforów 1 i 2- jeśli jeden jest pusty, to nie blokuje
//pobierania drugiego
                begin
                  P(fb1);
                  P(active1); //osobna sekcja krytyczna dla każdego bufora
                  pobierz_z_buf1; //operacje na jednym nie blokują operacji na innym buforze
                  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 wysłać element do
//bufora3
      P(active3);
      Wyślij_do_buf3;
      V(active3);
      V(fb3);
      end;
  end;
```

Na początku programu ustawiamy wartości: active1,2,3 = 1; fb1,2,3 = 0; eb1,2,3 = N1,N2,N3;
Następnie równolegle uruchamiane są procesy P1,2,3,4.

6. Semafory binarne:
   Semafory binarne Sb mogą przyjmowaćc tylko dwie wartości: 0 i 1.
   Przez Pb i Vb oznaczone są operacje na semaforach binarnych odpowiadające operacjom P i V.
   Pb- działa analogicznie.
   Vb – działa jak V z wyjątkiem sytuacji gdy mamy wartość semafora = 1, wtedy jej nie zwiększa.

   Implementacja binarnych operacji semaforowych z aktywnym czekaniem:

   - Pb(Sb):

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

     Czyli działa tak długo jak pActive = False; Jedynym powodem wyjścia z pętli może być zmiana Sb poprzez instukcję Vb(Sb) w innym przypadku do Sb przypisywane będzie cały czas False(0);

   - Vb(Sb):

     ```
     Sb := true;
     ```

7. Implementacja ogólnych operacji semaforowych:

   ```
   procedure P_S;
   begin
     while S <= 0 do;
     S := S – 1
   end;
   procedure V_S;
   begin
    S := S + 1
   end;
   ```

   Powyższe rozwiązanie ma 3 wady:
   - Jest nieatomowe
   - Wiele procesów może modyfikować S jednocześnie, nie ma zabezpieczenia przed przyjęciem ujemnej wartości prze zmienną semaforową- np. gdyby dwa procesy jeednocześnie 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)
   - W rozwiązaniu tym zastosowanu aktywne czekanie – proces czekający, mimo iż właściwie nic nie może zrobić, korzysta z czasu procesora, aż do przełączenia kontekstu.

Atomowość rozwiązania moglibysmy zapewnić umieszczając jakąś część kodu jako sekcję krytyczną z ograniczonym dostępem.

Implementacja operacji semaforowych z aktywnym czekaniem:

```
1.  program PV_IMPLEMENTATION;
2.  var active, delay: BOOLEAN;
3.  var NS: INTEGER;
4.  procedure PIMPLEMENTATION;
5.  var pActive, pDelay: BOOLEAN;
6.  begin
7.    pActive:=True;
8.    while pActive do
9.       testandset(pActive,active);
10.   NS:=NS-1;
11.   if NS ≥ 0 then
12.    begin
13.      S:=S-1;
14.      active:=False;
15.    end
16.   else
17.    begin
18.      active:=False;
19.      pDelay:=True;
20.      while pDelay do
21.         testandset(pDelay,delay)
22.    end
23. end;
```

```
24. procedure VIMPLEMENTATION;
25. var vActive: BOOLEAN;
26. begin
27.   vActive:=True;
28.   while vActive do
29.      testandset(vActive,active);
30.   NS:=NS+1;
31.   if NS > 0 then
32.      S:=S+1
33.   else
34.      delay:=False;
35.   active:=False
36. end;

37. begin
38.   active:=False;
39.   delay:=True;
40. end.
```

8. konieczne jest wzajemne wykluczanie – tylko 1 proces może wykonywać operacje na semaforze. Zapewnia je powyższa pętla – proces czeka aż active przyjmie wartośc false. Instrukcja testandset zapewnia atomowość operacji.

10-13. Sekcja krytyczna.

11. Równoważne if S > 0 (bo teraz NS := S – 1) – zmienna semaforowa nieujemna(zabezpieczenie nieujemności)

13. Ponieważ tylko 1 proces jest w sekcji krytycznej – można bezpiecznie zmniejszyć wartość S.

14. active dostaje wartość false- proces wychodzi z sekcji krytycznej, więc inny proces może wejść do sekcji.

18. Wyjście z sekcji krytycznej, teraz inny proces może do niej wejść.

19. Proces ma czekać.

20. Czeka tak długo aż delay nie dostanie wartości False(aż V nie ustawi delay na false).

31. Jeśli nic nie czeka (NS > 0 czyli S > 0) następuje zwiększenie S.

33. Jeśli jednak czekają jakieś procesy na dostęp do „sekcji krytycznej” to delay := False, i jeden z procesów przestaje czekać.

38. Active := false oznacza ze jeden z procesów może wejśc do sekcji krytycznej.

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

Poprawione rozwiązanie : (?)

W instrukcji P:

usunąć linię 14, a w lini 23: active := False;

W instrukcji V:

usunąć linie 35, a w lini 33: active := False

Wykład V: Synchronizacja cz. III:

Implementacja operacji P i V bez aktywnego

```
1.  program PV_IMPLEMENTATION;
2.    var active, delay: BOOLEAN;
3.    var NS: INTEGER;

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

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

czekania:

7.wyłączenie przerwań, żeby szybciej działały operacje.

20. Blokowanie procesu, który wywołał operację.

21. Usunięcie powyższego procesu z listy procesów gotowych do działania.

22. zwolnienie sekcji krytycznej (16V.)

23. włączenie przerwań i uruchomieenie innego procesu

13V. usunięcie procesu czekającego z listy proccesów czekających na S.

14V. Dodanie tego procesu do listy gotowych

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

1. Inna implementacja operacji semaforowych:
   a. Operacja wait (P(S)) i signal (V(S)):

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

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

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

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

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

Powyższe rozwiązanie nie spełnia zasady atomowości – nie ma wzajemnego wykluczania i może być jednocześnie wykonywanych kilka operacji na tym samym semaforze.

S.L -> Lista procesów dla danego semafora S
11. Zawieszenie procesu.
19. Pobranie procesu P z listy S.L
20. Odwieszenie procesu P.

Poniższe operacje nie są implementacjami, tylko opisami znaczenia tych operacji semaforowych, co w praktyce oznacza, że nie mają zapewnionej atomowości operacji.

   b. Operacja lock w i unlock w :

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

```
4.  unlock w:
5.     w : = 0;
```

   c. Operacja ENQ(r) :

```
6.  ENQ(r):
7.    if inuse[r] then     // resource r is used
8.      begin
9.        Insert p on r-queue;
10.       Block p
11.     end                // queue associated with r
12.   else
13.     inuse[r]:= True ;
```

Opis działania:
Jeśli zasób r jest zajęty – następuje dołączenie procesu p do kolejki z nim skojarzonej, jeśli zasób jest wolny- proces p może na nim operować, ale inne procesy NIE. (na zasobie r)

   d. Operacja DEQ(r) :

```
14. DEQ(r):
15.   p:=Remove from r-queue
16.   if p ≠ Ω
17.     then Activate p   // p = Ω means that queue was empty
18.   else inuse[r] : = False ;
```

Opis działania:
Zdejmuje process p z kolejki i jeśli kolejka nie jest pusta uaktywnia go, jeśli jest pusta (w przeciwnym wypadku do pełnej kolejki) możemy operować na zasobie r.

e. Operacja WAIT(e) :

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

Opis działania:
Tylko jeden proces może czekać na nadejście jakiegoś zdarzenia e.
WAIT(e) – jeśli nikt nie czeka na e, to procesem czekającym staje się proces p, który zostaje zablokowany.

f. Operacja POST(e):

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

Operacja Post(e) – działa podobnie jak operacja WAIT, jeżeli żaden proces nie jest ustawiany to zdarzenie e dostaje flagę True, a następnie sprawdzmy czy zdarzenie e czeka, jeśli tak to następuje aktywacja procesu skojarzonego ze zdarzeniem e.

g. Operacja Block(i):

```
1.  Block(i):
2.    if ¬ wws[i]    // wait for Wakeup flag associated with process i
3.      then Block process i
4.      else wws[i]:=False;
```

Opis działania:
Block(i) – jeśli i jest gotowy – zostaje zablokowany , w innym wypadku flaga wws dla i zostaje ustawiona na False; (wws False jeśli następuje blokowanie zablokowanego procesu i)

h. Operacja Wakeup(i):

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

Opis działania:
Wakeup(i) – jeśli i jest gotowy – flaga wws jest ustawiana na True, jeśli i nie jest gotowy następuje uaktywnienie procesu i .

wws – true jeśli następuje pobudka obudzonego procesu

2. Event Counters:
   Three operations are defined on a event counter E:
   - read(E) - return the current value of E.
   - advance(E) – automatically increment E by 1.
   - await(E, v) – wait until E has a value of v or more.

   Zakładamy atomowość powyższych operacji.
   Poniżej przedstawiamy rozwiązanie problemu producenta konsumenta używając licznika zdarzeń (Event Counter).

```
1.  #include "prototypes.h"
2.  #define N 100                       // number of slots in the buffer
3.  typedef int EVENT_COUNTER;          // event counters are a special kind of int
4.  EVENT_COUNTER in=0;                 // counts items inserted into buffer
5.  EVENT_COUNTER out=0;                // counts items removed from buffer
6.  void PRODUCER(void) {
7.   int item, sequence =0;
8.   while(True)                        // infinite loop
9.    produce_item(&item);              // generate something to put in buffer
10.   sequence=sequence+1;              // count items produced so far
11.   await(out, sequence-N);           // wait until there is room in buffer
12.   enter_item(item);                 // put item in slot (sequence -1) % N
13.   advance(&in);                     // let consumer know about another item
14.  }
15. }
16. void CONSUMER(void) {
17. INT item, sequence=0;
18.  while(True) {                      // infinite loop
19.   sequence=sequence+1;              // number of item to remove from buffer
20.   await(in, sequence);              // wait until required item is present
21.   remove_item(&item);               // take item from slot (sequence-1)%N
22.   advance(&out);                    // let producer know that item is gone
23.   consume_item(item);               // do something with the item
24.  }
25. }
```

   11. czeka aż out >= sequence – N
   13. in++;
   20. Czeka aż in >= sequence
   22. out++;

Opis działania programu:
Producent zmienia licznik-in, konsument licznik-out.
Producent czeka aż out >= sequence – N, czyli liczbą wyjętych z bufora jest >= niż (liczba wyprowadzonych – minus pojemność bufora), czyli
(liczba wyjętych + poj. Bufora) >= liczba wyprodukowanych –(oznacza to że można dalej produkować)
Konsument czeka, aż in >= sequence, czyli liczba wyprodukowanych >= liczba skonsumowanych.

Programowe mechanizmy synchronizacji:

3. Regiony krytyczne:
   Niech następująca deklaracja zmiennej v typu T określa zmienna dzieloną przez wiele procesów.
   var v: **shared** T;
   Zmienna v będzie dostępna tylko w obrębie instrukcji region o następującej postaci:
   region v **do** S;
   Dla każdej deklaracji:
   var v: **shared** T;
   Kompilator generuje semafor v-mutex z wartością początkowa 1.
   Dla każdej instrukcji:
   region v **do** S;
   Kompilator generuje następujący kod:

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

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

5. Rozwiązanie problemu producent – konsument przy użyciu warunkowego regionu krytycznego:

```
1.  var buffer: shared record
2.                  pool: array [0..n - 1] of ITEM;
3.                  count, in, out: INTEGER;
4.                end;
5.  region buffer when count < n
6.    do begin
7.       pool[in]:=nextp;
8.       in:=(in + 1) mod n;
9.       count:=count + 1;
10.   end;
11.  region buffer when count > 0
12.   do begin
13.      nextk:=pool[out];
14.      out:=(out + 1) mod n;
15.      count:=count - 1;
16.   end;
```

Proces produkujący umieszcza nową jednostkę nextp w buforze dzielonym wykonując powyższą instrukcję.

Proces konsumujący usuwa jednostkę w bufora dzielonego i zapamiętuje ją w nextk za pomocą powyższej instrukcji

2. Bufor właściwy
3. Liczniki
5. Jeśli bufor nie jest pełny wykonuje poniższe operacje
7. Dodanie do bufora.
8. Zapamiętanie miejsca w buforze dla następnego elementu
9. Zwiększenie licznika elementów w buforze.

Implementacja warunkowych regionów krytycznych:

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

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

2. początkowo ustawione były na:
xMutex := 1; xDelay := 0 (semafory binarne)
xDelay do odczekania na B = True;
xMutex do czekania na wejście do S
xCount = xTemp := 0;

4. Do wzajemnego wykluczania.
7. Zwiększenie liczby czekających na spełnienie B(True).
8. Jakiś inny może wejść do sprawdzenia B i jeśli dla niego B = True do S (zwolnienie sekcji krytycznej)
9. Czekanie na xDelay aż jakis wychodzący pozwoli testować B.
10. Jeśli teraz jakiś proces wychodząc z S wykona signal(xDelay) – czekający przejdzie dalej, w pętli while wykonywane jest testowanie B dla kolejnych procesów.
12. Testowanie jednego więcej
13. Jeśli wciąż mniej testowanych niż czekających przepuszczenie kolejnego czekającego do na spełnienie B do testowania
15. xTemp = xCount oznacza, że to już ostatni proces testujący – pozwala on więc jakiemuś procesowi czekającemu na wejście do regionu krytycznego wejść.
17. Czekanie na przepuszczenie przez wychodzącego z regionu – i jeśli wtedy B jest spełnione, proces przejdzie do S, w przeciwnym wypadku cała pętla od nowa.
19. Jeśli B = true dla jakiegoś procesu – już nie czeka i zmniejsza liczbę czekajacych na dostęp do S.
22. Jeśli jakieś czekają
23. Żeby testowanie B przez czekające na B mogło przebiegać poprawnie ustawiamy x-temp = 0;
24. Przepuszczenie pierwszego procesu czekającego na B do testowania B proces ten przepuści następne dod testowania B
27. Nikt nie czeka na spełnienie B – ktoś może zacząć wykonywać S.

Opis działania programu:

Za każdym razem, kiedy jakiś proces opuści region krytyczny, następuje dla wszystkich procesów czekających na spełnienie warunku B sprawdzenie 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 czekajacych, przepuszcza kolejnego (ostatni – jakiegoś czekajacego na zewnątrz warunkowego regionu krytycznego), po czym może wejść do pętli lub czeka na xDelay. Jeśli jakiś proces wyjdzie z petli w trakcie testowania, to nie podniesie on wartości xDelay w tej pętli i zrobi to dopiero jak wykona S.

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

```
region v do
  begin
    S1;
    await(B);
    S2;
  end;
```

- Sprawdzenie wyrażenia B (czekanie aż B będzie True)

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

   Dwie opcje rozwiązania powyższego problemu:
   - Priorytet czytelnika – żaden czytelnik nie powinien czekać, chyba że pisarz pisze czyli nie powinien czekać na zakończenie pracy innych czytelników tylko dlatego, że czeka na to też jakiś pisarz. Czytelnicy nie mają obowiązku czekać na dostęp do zasobu. Pisarz musi czekać na opuszczenie zasobu przez wszystkie inne procesy.
   - Priorytet pisarza – jeśli pisarz jest gotowy, to rozpocznie wykonywanie swojej pracy tak wcześnie jak to tylko możliwe – jeśli jakiś pisarz czeka, to żaden nowy czytelnik nie rozpocznie czytania. W tym wariancie może dojść do zagłodzenia oczekujących czytelników.

8. Rozwiązanie problemu pisarzy i czytelników z użyciem semaforów: z priorytetem czytelnika:

```
1.  shared var
2.     nReaders : INTEGER;
3.     mutex, wmutex, srmutex : SEMAPHORE;
4.  procedure READER;
5.  begin
6.    P(mutex);
7.    if nReaders=0 then
8.       begin
9.          nReaders:=nReaders + 1;
10.         P(wmutex);
11.      end
12.   else
13.      nReaders:=nReaders + 1;
14.   V(mutex);
15.   read(f);
16.   P(mutex);
17.   nReaders:=nReaders - 1;
18.   if nReaders = 0 then
19.      V(wmutex);
20.   V(mutex);
21. end ;
22. procedure WRITER(d: data);
23. begin
24.    P(srmutex);
25.    P(wmutex);
26.    write(f, d);
27.    V(wmutex);
28.    V(srmutex);
29. end;

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

2. Liczba czytelników czytających w danej chwili.
6. Zabezpieczenie żeby tylko jeden czytelnik mógł na raz zmienić liczbę czytelników.
7. Pierwszy czytelnik dopuszczony do czytania, zwiększa liczbę czytelników i opuszcza semafor dla pisarzy(lub czeka jeśli ktoś pisze).
12. Kolejni czytelnicy tylko zwiększają liczbę czytelników, bo semafor już jest opuszczony dla pisarzy.
14. Żeby wielu mogło równolegle czytać jeśli zakończą pierwszą sekcję krytyczna (7-13).
16. Wejście do drugiej sekcji krytycznej.
19. Podniesienie semafora pisarzom jeśli nikt nie czyta.
20. Wyjście z drugiej sekcji krytycznej i podniesienie semafora jeśli nikt nie czyta.
25. Zabezpieczenie żeby na raz mógł korzystać tylko jeden pisarz.
27. Teraz inny pisarz może pisać.
31. Wszystkie semafory binarne.

Opis działania programu:
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(pisarze). Jeśli jednak pisarz skończy, to podniesie wmutex – i wtedy mogą już wejść czytelnicy, bo czekają tylko na wmutex, a pisarze muszą jeszcze najpierw mieć podniesiony srmutex co nastąpi później.

9. Rozwiązanie problemu pisarzy i czytelników z użyciem regionów krytycznych: z priorytetem pisarzy:

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

Proces czytelnika

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

Proces pisarza

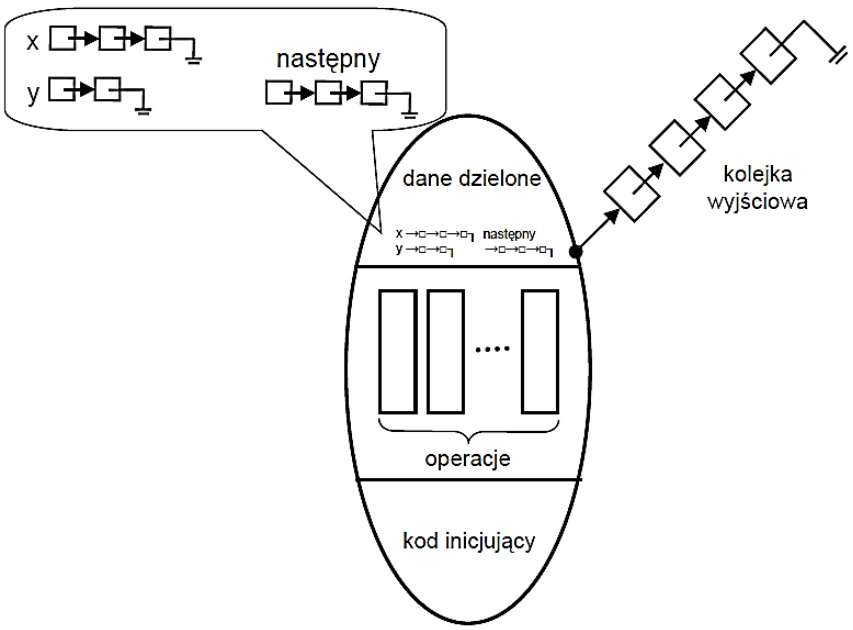
```
17. region v do
18.   begin
19.     nWriters:=nWriters + 1;
20.     await((not busy)and(nReaders=0);
21.     busy:=True;
22.   end;
23. ...
24. write file
25. ...
26. region v do
27.   begin
28.     nWriters:=nWriters- 1;
29.     busy:=False;
30.   end;
```

3. Czy jakiś pisarz pisze.
7. Czekanie aż żaden pisarz nie czeka – a więc priorytet dla pisarzy.
8. Jeśli żaden pisarz nie czeka zwiększeni liczby czytających.
20. Czekanie aż nikt nie pisze i nikt nie czyta

Algorytm sam w sobie ma proste działanie, a priorytet jest zauważalny w pierwszej chwili, gdy widzimy, że proces czytelnika oczekuje na zwolnienie zasobów przez wszystkich oczekujących pisarzy.

10. *Monitory:*

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 the bodies of producers or functions that implement operations on the type. The syntax of 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 ;
```

Programista, który chce napisać przykrojony na miarę własnych potrzeb schemat synchronizacji, może zdefiniować jedną lub kilka zmiennych typu warunek:
var x, y : CONDITION;
Jedynymi operacjami, które mogą dotyczyć warunku są:

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

Jeśli proces odwoła się do zajętego Monitora – proces odwołujący 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 powstałby deadlock(zakleszczenie).

11. *Problem producenta-konsumenta rozwiązany z użyciem Monitorów*:

```
1.  type PRODUCER_CONSUMER = monitor
2.  var full, empty : CONDITION;
3.  count : INTEGER;

4.  procedure entry ENTER;
5.   begin
6.     if count = N then full.wait;
7.     enter_item;
8.     count:=count + 1;
9.     if count = 1 then empty.signal;
10.  end;

11. procedure entry REMOVE;
12.  begin
13.    if count = 0 then empty.wait;
14.    remove_item;
15.    count:=count - 1;
16.    if count=N - 1 then full.signal;
17.  end;

18. begin
19.    count:=0;
20. end monitor;
```

```
21. procedure PRODUCER;
22.  begin
23.    while True do
24.     begin
25.       produce_item;
26.       PRODUCER_CONSUMER.enter;
27.     end
28.   end;

29. procedure CONSUMER;
30.  begin
31. while True do
32.  begin
33.    PRODUCER_CONSUMER.REMOVE;
34.    consume_item
35.  end;
36.  end.
```

12. Alokacja zasobów z wykorzystaniem monitora:

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

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

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

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

13. Rozwiązanie problemu czytelników i pisarzy z wykorzystaniem monitorów: z priorytetem czytelnika:

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

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

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

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

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

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

3. Ocenia czy ktoś pisze.
7. Jeśli ktoś pisze to czekanie aż zmieni OKtoRead
10. Tylko jeden czytelnik może zacząć czytanie w danej jednostce czasu.
16. Priorytet czytelnika – przepuszczenie pisarza dopiero wtedy gdy nikt nie czyta.
27. Priorytet czytelnika – przepuszczanie czytelników, a pisarzy tylko wtedy gdy nie czeka żaden inny czytelnik.

14. *Implementacja Monitora*:

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

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

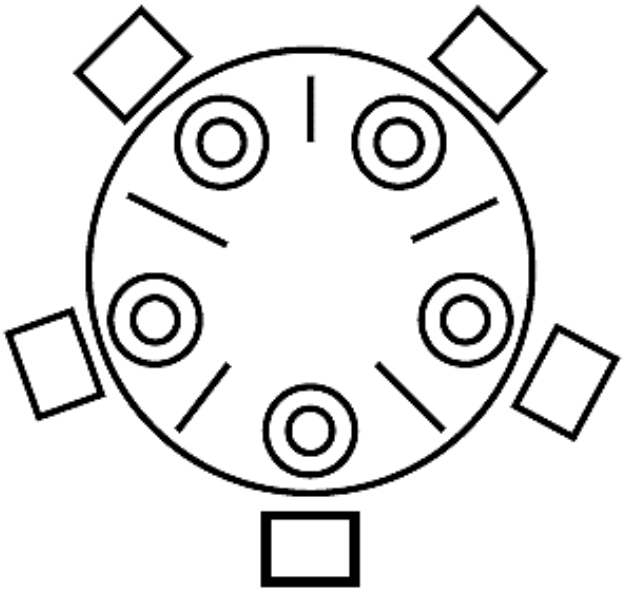
```
x.signal:
14. if xCount > 0
15.   then
16.     begin
17.       nextCount:=nextCount+1;
18.       signal(xSem);
19.       wait(next);
20.       nextCount:=nextCount-1;
21.     end.
```

- mutex- semafor gwarantujący wzajemne wykluczanie
- nextCount – ilość procesów, które powołują się na x.signal i są uśpione
- next – semafor służący do zawieszenia procesu używającego x.signal
- xCount – ilość procesów czekających na x.signal
- xSem – semafor służący do zawieszenia procesu przy użyciu x.wait

1. Czekamy sami
5. Po wyjściu można przepuścić
6. Czekającego na next
7. A jeśli nikt nie czekał, to mutex może wejdzie
9. Teraz czekamy i jakiś inny może wejść do monitora – jeśli czeka na next,
10. Lub jeśli nie czeka na next to może wejść mutex.
13. Wykonanie po odczekaniu x.signal
14. Jeśli jakieś wykonały x.wait i czekają na xSignal trzeba im ustąpić.
17. Teraz więcej czeka na next
18. Przepuszczamy jakiegoś czekającego na x.signal
19. I sami czekamy na signal(next)
20. Po odczekaniu czeka o jeden mnie w next.

Opis działania programu:
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 działałby naraz w monitorze – wprowadza się dodatkowy semafor next z wartością początkową 0, za którego pomocą procesy sygnalizujące mogą same wstrzymać swoje wykonanie.

15. Problem jedzących filozofów:
    The dining philosophers problem is a classic problem that has formed the basis for a large class of synchronization problems. In one version of this problem five philosophers are sittingin 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 ofeach 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.



16. Rozwiązanie problemu jedzących filozofów z wykorzystaniem monitorów:

```
1.  type DINNING_PHILOSOPHERS = monitor
2.    var state : array [0..4] of  (Thinking, Hungry, Eating);
3.    var self : array [0..4] of CONDITION;
4.  procedure entry PICKUP(i: 0..4);
5.  begin
6.    state[i]:=Hungry;
7.    test (i);
8.    if state[i] ≠ eating
9.      then self[i].wait;
10. end ;

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

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

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

3. Za pomocą tablicy self – głodni będą czekać na widelce
7. Sprawdzenie czy można jeść
8. Jeśli nie można jeść niech filozof i czeka.
14.15. Sprawdzenie czy sąsiedzi mogą jeść.
23. Obaj sąsiedzi nie mogą jeść(19 i 21 linia) więc k-ty filozof (i) musi być głodny sam
24. Wtedy k-ty filozof może rozpocząć jedzenie
25. K-ty filozof może przestać czekać.

Rozwiązanie to zapewnia, że dwaj sąsiedzi nigdy nie będą jedli równocześnie; nie dojdzie też do deadlocka – ale istnieje w tym rozwiązaniu możliwość zagłodzenia filozofa. W przypadku gdy jeden z filozofów cały czas będzie miał jedzących sąsiadów.

Wykład VI: Synchronizacja cz IV:

1. Operacje wymiany komunikatów:
   Wymiana komunikatów realizowana jest z użyciem dwóch podstawowych operacji komunikacyjnych:
   - send (P, m)
   - receive(Q, m)

   Gdzie:
   m – przesyłany komunikat (wiadomośc eng. Message)
   P – jest odbiorcą komunikatu
   Q – jest nadawcą komunikatu

2. Łącza:
   Łącze komunikacyjne jest elementem umożliwiającym transmisję informacji między interfejsami odległych węzłów. Wyróżnia się łącza jedno i dwukierunkowe. Wyposażone są one w bufory o określonej pojemności

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

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

   Łącza mogą gwarantować również, w sposób niewidoczny dla użytkownika, że żadna wiadomość nie jest tracona, duplikowana lub zmieniana -są to tzw. **Łącza niezawodne**.

   Czas transmisji w łączu niezawodnym może być ograniczony lub jedynie określony jako skończony lecz nieprzewidywalny. W pierwszym przypadku mówimy o **transmisji synchronicznej** lub **z czasem deterministycznie nieograniczonym** (w szczególności równym zero), a w drugim – o **transmisji asynchronicznej** lub z **czasem niedeterministycznym**.

   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/ deterministycznie nieograniczonym) lub asynchroniczne(o nieokreślonym czasie transmisji/ czasem niedeterministycznym);
   - Niezawodne(gwarantujące, że żadna wiadomość nie jest tracona, duplikowana lub zmieniana) lub zawodne (wiadomość może być utracona, zduplikowana lub zmieniona).

3. Określanie nadawców i odbiorców:
   Procesy mogą komunikować się pośrednio i bezpośrednio. 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 m 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 znajomości nadawcy. W tym wypadku operacje send i reveive są zdefiniowane następująco:
   - send(P, m) -> nadaj komunikat m do procesu P
   - receive(id, m) -> odbierz komunikat od dowolnego procesu; pod id ostanie podstawiona nazwa procesu, od którego nadszedł komunikat

   W komunikacji pośredniej komunikaty są nadawane i odbierane poprzez skrzynki pocztowe (nazywane tez portami).
   Abstrakcyjna skrzynka pocztowa jest obiektem, w którym procesy mogą umieszczać komunikaty, i z którego komunikaty mogą być pobrane. Każda skrzynka pocztowa ma jednoznaczną identyfikację. Proces może komunikować się z innymi procesami za pomocą różnych skrzynek pocztowych. W tym wypadku operacja send i receive są zdefiniowane następująco:
   - send(A, m) -> nadaj komunikat do skrzynki A;
   - receive(A, m) -> odbierz komunikat ze skrzynki A;

   Łącza komunikacyjne mają tutaj 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
4. Skrzynka pocztowa :
   Skrzynka może być własnością procesu lub systemu. Jeżeli skrzynka należy do procesu (tzn. jest przypisana lub zdefiniowana jakaś część procesu), to rozróżnia się jej **właściciela** (który za jej pośrednictwem może tylko odbierać komunikaty) i **użytkownika** (który może tylko nadawać komunikaty do danej skrzynki).

   W wielu przypadkach, proces ma możliwość zadeklarowania **zmiennej typu skrzynka_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 odbieranie komunikatów może jednak zostać przekazany innym procesom za pomocą odpowiednich instrukcji systemowych.

5. Operacje synchroniczne i asynchroniczne:
   Kanały o niezerowej pojemności umożliwiają realizację następujących operacji komunikacji:
   - **Nieblokowanych (asynchronicznych):**
     Proces nadający przekazuje komunikat do kanału (bufora) i natychmiast kontynuuje swoje działanie kolei, a proces odbierający odczytuje stan kanału wejściowego, lecz nawet gdy kanał jest pusty, proces kontynuuje swoje działanie
   - **Blokowanych (synchronicznych):**
     Nadawca jest wstrzymywany do momentu, gdy wiadomość zostanie odebrana przez adresata, natomiast odbiorca – do momentu, gdy oczekiwana wiadomość pojawi się w jego buforze wejściowym.

   W komunikacji **synchronicznej**, nadawca i odbiorca są blokowani, aż odpowiedni odbiorca odczyta przesłaną do niego wiadomość.
   W przypadku komunikacji **asynchronicznej**, nadawca lub odbiorca komunikuje się w sposób nieblokowany.
6. *Rozwiązanie problemu producenta-konsumenta przy użyciu Łączy-komunikatów*:

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

17.odbiór blokowany
19. Wysyłanie asynchroniczne

Opis działania programu:
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 sygnał. (Wynika to z faktu że operacje wykonywane są asynchroniczne).

7. Problem Producenta-Konsumenta za pomocą wymiany wiadomości przy założeniu, że jedynym mechanizmem komunikacji i synchronizacji jest wymiana wiadomości -bez operacji add i take- rekordy przesyłane są 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
        reveive(consumer, record); // czeka na rekord do odebrania ze skrzynki
        send(producer, empty); // pobrał – producent może dalej działać (jeśli czekał na empty)
        consume;
  end;
end;

begin
  I := N; // przypisanie do I wielkości bufora
  while I > 0 do
  begin
     send(producer, empty);
     I := I – 1;
  end;
  parbegin
     Producer;
     Consumer;
  parend
end;
```

Powyższy algorytm jest rozwiązaniem zadania przedstawionego w treści. I synchronizuje producenta i konsumenta korzystając jedynie z operacji send i receive.

Wykład VII: Zakleszczenie (deadlock):

1. *Zakleszczenie*:

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

   W każdej chwili zadanie Pj jest schrakteryzowane przez:

   - **Wektor maksymalnych żądań:**
     oznaczający maksymalne żadanie zasobowe zadania Pj w dowolnej chwili czasu
     $C(P_j) = [C_1(Pj), C_2(Pj),\ldots., C_s(Pj)]^T$

   - **Wektor aktualnego przydziału**
     $A(P_j) = [A_1(Pj), A_2(Pj),\ldots., A_s(Pj)]^T$

   - **Wektor rang** zdefiniowany jako różnica między wektorami C i A.
     $H(P_j)=C(P_j) - A(P_j)$ //ile zadanie może jeszcze maxymalnie zażądać

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

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

   Gdzie

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

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

   - **Żądanie przydziału dodatkowych zasobów:**
     $$\boldsymbol{\rho^a(P_j) = [\rho_1^a(P_j), \rho_2^a(P_j), \ldots, \rho_s^a(P_j)]^T}$$
     Gdzie $\boldsymbol{\rho_k^a(P_j)}$ jest liczbą jednostek zasobu R_k żadanych dodatkowo przez Pj

   - **Żądanie zwolnienia zasobu:**
     $$\boldsymbol{\rho^r(P_j) = [\rho_1^r(P_j), \rho_2^r(P_j), \ldots, \rho_s^r(P_j)]^T}$$
     Gdzie $\boldsymbol{\rho_1^r(P_j)}$ jest liczbą jednostek zasobu R_k zwalnianych przez Pj.

   Łatwo wykazać:

   $\forall_k \forall_j \; \boldsymbol{\rho_k^a(P_j) \leq H_k(P_j)}$ nie można zażądać więcej niż wynosi wartość wektora rang

   $\forall_k \forall_j \; \boldsymbol{\rho_k^r(P_j) \leq A_k(P_j)}$ nie można zwolnić więcej niż się ma przydzielone

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

$$\forall_{k} \rho_k^a(P_j) \leq f_k \qquad j = 1, 2, \dots, s$$ 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:

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

Stan systemu nazywany **stanem bezpiecznym** (safe) ze względu na zakleczenie, jeżeli istnieje sekwencja wykonywania zadań przebywających w systemie oznacza $\{P^1, P^2, \dots, P^n\}$ i nazywana **sekwejnsją bezpieczną**, spełniającą następującą zależność:

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

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

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 wynikającego z alokacji zasobów jest **bezpieczna**, jeżeli stan końcowy jest stanem bezpiecznym.

2. Zakleszczenie:

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

$$\underset{\Omega \neq \Phi}{\exists} \quad \underset{j \in \Omega}{\forall} \quad \underset{k}{\exists} \rho_k^a > f_k + \sum_{i \neq \Omega} A_k(P_i)$$

   Gdzie Ω jest zbiorem indeksów (lub zbiorem zadań)

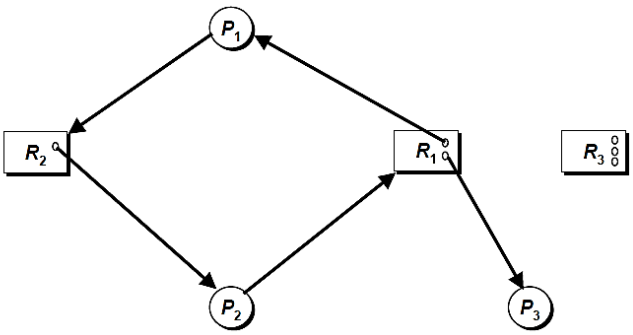
   Istnieje niepusty zbiór taki, że dla każdego procesu z tego zbioru istnieje jakieś żądanie przez ten proces dodatkowych zasobów / zasobu (nieprzywłaszczalny), które przekracza (liczbę wolnych + liczbę zasobów w dyspozycji procesów niezakleszczonych)

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

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

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

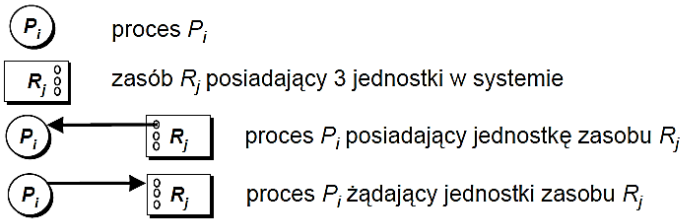
3. Graf alokacji zasobów:



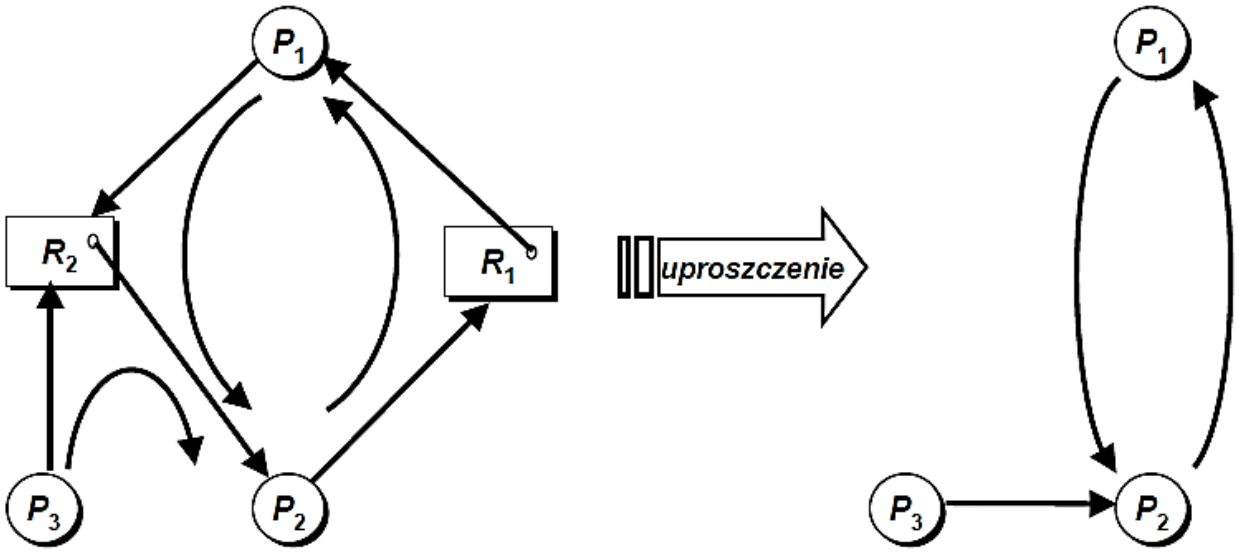
- P1 żąda jedynej jednostki R2
- P2 posiada jedyną jednostkę R2
- P2 żąda jednostki R1
- P1 posiada jedną z dwóch jednostek R1
- P3 posiada drugą z dwóch jednostek R1
- R3 ma 3 jednostki nieużywane i nikt go nie chce

Tutaj nie ma deadlocka (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.

Legenda:



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



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…

5. Warunki konieczne wystąpienia zakleszczenia:
   Warunkami koniecznymi wystąpienia zakleszczenia są:
   a. **Wzajemne wykluczanie:**
      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)
   b. **Zachowywanie zasobu:**
      Proces oczekujący na przydzielenie dodatkowych zasobów nie zwalnia zasobów będących aktualnie w jego dyspozycji.
   c. **Nieprzywłaszczalność:**
      Zasoby są nieprzywłaszczalne tzn. ich zwolnienie może być zainicjowane jedynie przez proces dysponujący w danej chwili zasobem.
      Nie ma np. timeouta przerywającego zawłaszczenie
   d. **Istnienie cyklu oczekiwań:**
      Występuje pewien cykl procesów, z których każdy ubiega się o przydział dodatkowych zasobów będących w dyspozycji kolejnego procesu cyklu.
      (Cykl w WFG)
6. Przeciwdziałanie zakleszczeniu:
   a. Konstrukcje systemów immanentnie wolnych od zakleszczenia:
   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 wykonywanych równocześnie -> jeśli mamy np. 5 zasobów to maksymalnie 4 procesy – zawsze rezerwa zasobu, który może być przydzielony w razie zażądania.
   b. Detekcja zakleszczenia i odtwarzanie stanu wolnego od zakleszczenia:
   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 od zakleszczenia.
   Pełna swoboda w przydziale zasobów. Problemy – jak wykryć i odtworzyć w przypadku zakleszczenia. W razie deadlocka usuwanie procesów (trzeba minimalizować koszty usunięcia)
   c. Unikanie zakleszczenia:
   W podejściu tym zakłada się 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.
   Sprawdzanie każdego zadania, czy jego realizacja nie doprowadzi do stanu niebezpiecznego i jeśli tak to zawieszenie procesu. Każde zwolnienie zasobu – analiza procesów zawieszonych.
   d. Zapobieganie zakleszczeniu:
   W ogólności podejście to polega na wyeliminowaniu możliwości zajścia jednego z warunków koniecznych zakleszczenia.

7. *Algorytm Habermana*:
    a. Zainicjuj D := {1,2, … ,n} i f;
       D – zbiór procesów (na początku wszystkie mogą być potencjalnie zakleszczone)
    b. Szukaj zadania o indeksie j∈**D** takiego, że:
       $\rho^a(P_j) \leq f$ czyli takiego, którego żądanie może być spełnione
    c. Jeżeli zadanie takie nie istnieje, to zbiór zadań odpowiadający zbiorowi D jest zbiorem zadań zakleszczonych. **Zakończ wykonywanie algorytmu**.
    d. W przeciwnym razie, podstaw:
       D := D – {j}; f := f + A(P_j); założenie, że zadanie się skończy i zwolni zasoby
    e. Jeżeli D = Ø, 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))

8. Odtwarzanie stanu:
   Spośród zadań zakleszczonych wybierz zadanie (zadania), którego usunięcie spowoduje osiągniecie stanu wolnego od zakleszczenia najmniejszym kosztem.

Algorytm Holt'a

```
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 ≤ f_k ∧ I_k ≤ 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  c_0 > 0 then go to LS;
20.     if Y = true then answer "no"
21.       else answer "yes";
22. end.
```

Wady podejścia detekcji do odtwarzania stanu:

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

Zalety podejścia detekcji do odtwarzania stanu:

- Brak ograniczeń na współbieżność wykonywania zadań.
- Wysoki stopień wykorzystania zasobów
  Wszystkie zasoby wolne mogą być przydzielone jeśli procesy tego zażądają.
- Podejście unikania.

Idea algorytmu :
E – macierz trójwymiarowa zawierająca uporządkowaną tablicę żądań przydziału dodatkowych zasobów(H)
ρ – składa się z dwóch 2-wymiarowych tablic s*n „jeden za drugą":

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

Przeglądanie tablicy i sprawdzenie – 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 algorytmie 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 przeciwnym razie O(nlogn) – ponieważ trzeba ją najpierw posortować.

Przykład rozwiązania zadania algorytmem Holt'a:

Wynikiem algorytmu jest sekwencja procesów.

Kolumny w macierzach odpowiadają procesom, a wiersze zasobom (w liczbie jednostek)

Dany jest stan początkowy systemu:

$$A = \begin{bmatrix} 0 & 0 & 2 & 1 & 1 \\ 2 & 0 & 4 & 0 & 1 \\ 1 & 1 & 1 & 2 & 1 \end{bmatrix} \quad m = \begin{bmatrix} 5 \\ 9 \\ 9 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 5 & 3 & 5 & 3 \\ 2 & 1 & 5 & 9 & 6 \\ 2 & 2 & 4 & 2 & 6 \end{bmatrix}$$

a) Stosująca algorytm Holt'a sprawdź czy stan jest bezpieczny:

C – wektor maksymalnych żądań – czyli maksymalna ilość jakimi może dysponować

A – wektor aktualnego przydziału – czyli ilość zasobów jaką dysponuje j-ty proces

H – wektor rang czyli maksymalna ilość zasobów jaką może zażądać j-ty proces tworzona przez odjęcie C- A

m - wektor zasobów w systemie (wolne + przydzielone)

f – wektor zasobów wolnych (powstaje poprzez zsumowanie wartości każdego z wierszy macierzy A i odjęcie tej wartości od wartości z tego samego wiersza w macierzy m)

E - duże liczby oznaczają żądania zasobowe i odpowiadają odwołaniom E(0,x,y), natomiast małe liczby informują, do którego procesu należą dane żądania zasobowe E(1,x,y). Duże liczby to posortowana macierz H, małe liczby to numer procesu który posiadał tą określoną ilość zasobów.

c – wektor, którego c[0] to liczba procesów w systemie(w naszym przypadku 5), c[1…n] (c[1…5]) inicjalizujemy liczbą zasobów w systemie (liczbą wierszy w naszym przypadku 3).

I – wektor o rozmiarze równym ilości zasobów (3) i (w inicjalizacji) wartościach 1, wektor ten będzie nas informował które żądanie zasobowe z macierzy E będziemy starali się spełnić (na początek pierwsze)

$$I = [1,1,1]^T$$

$$H = C - A = \begin{bmatrix} 3 & 5 & 1 & 4 & 2 \\ 0 & 1 & 1 & 9 & 5 \\ 1 & 1 & 3 & 0 & 5 \end{bmatrix} \quad E = \begin{bmatrix} 1_3 & 2_5 & 3_1 & 4_4 & 5_2 \\ 0_1 & 1_2 & 1_3 & 5_5 & 9_4 \\ 0_4 & 1_1 & 1_2 & 3_3 & 5_5 \end{bmatrix} \quad f = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$c = \begin{matrix} [5 & 3 & 3 & 3 & 3 & 3] \\ & 3 & 3 & 2 & 3 & 3 \\ & 2 & 2 & 1 & 3 & 3 \\ & 1 & 1 & 0 & 2 & 3 \\ & & & P3 & & \end{matrix} \quad I1 = [2,4,5] \; f = \begin{bmatrix} 3 \\ 6 \\ 4 \end{bmatrix} itd.$$

Kolejne kroki wykonywania algorytmu: kolejne wiersze pod wektorem c powstają przez:

1. Pobieramy wartość z pierwszego wiersz wektora f (wektor zasobów wolnych) i sprawdzamy w tabeli E dla których procesów wystarczy zasobów, z tabeli E widzimy że jest jeden taki proces o numerze 3(mała cyfra) przy zasobie o wartości 1 (duża cyfra) od procesów odpowiadających kolejnym pozycjom wektor c odejmujemy 1.
2. Pobieramy wartość z drugiego wiersza f i sprawdzamy w drugim wierszu macierzy E, teraz widzimy że są 3 takie procesy 0, o numerze 1 2 3 itd.
3. Pobieramy wartość z trzeciego wiersza f i sprawdzamy w trzecim wierszu macierzy E, teraz widzimy że są 4 procesy dla których taka liczba zasobów wystarczy i są to procesy 4, 1, 2, 3 i od tych procesów odejmujemy 1. Jeżeli pojawi się 0 to oznacza że nie dochodzi do zakleszczenia a proces, który ma 0 (P3) to proces którego żądania jesteśmy w stanie zaspokoić.

Aktualizacja wektora In następuje poprzez dodanie liczby procesów, dla których zasobów wystarczy dlatego w pierwszym wierszu dodajemy 1, w drugim 3 a w trzecim 4.

Wektor f to poprzednia wartość wektor f + liczba zasobów zwolnionych przez proces P3 (dany proces zakończony zerem) dlatego mamy odpowiednio 1+ 2 = 3; 2+4 = 6; 3 + 1 = 4;

Algorytm ten wykonujemy tak długo aż nie dojdzie do zakleszczenia czyli brak 0 w ostatnim wierszu c, a to oznacza za mało zasobów, lub c[0] będzie 0, wtedy nie dojdzie do zakleszczenia na pewno.

b) Zakładając że konsekwentnie stosowane jest pojęcie unikania, podaj sekwencję stanów systemu (zaznaczając procesy zawieszone) odpowiadającą następującemu ciągowi zasobów:

- w chwili t_1: $\rho^a(P_2)=\begin{bmatrix}0\\1\\0\end{bmatrix}$
- w chwili $t_2 > t_1$: $\rho^a(P_3)=\begin{bmatrix}0\\1\\1\end{bmatrix}$
- w chwili $t_3 > t_2$: $\rho^r(P_4)=\begin{bmatrix}1\\0\\1\end{bmatrix}$

Jeżeli przy p(ro) jest a to chodzi o przydzielenie dodatkowych zasobów
Jeżeli przy p(ro) jest r to chodzi o zwolnienie zasobów przez proces
Zakładając że mamy przydzielany zasób dla P2 mamy:

$$A=\begin{bmatrix}0&0&2&1&1\\2&1&4&0&1\\1&1&1&2&1\end{bmatrix}\quad H=C-A=\begin{bmatrix}3&5&1&4&2\\0&1&1&9&5\\1&1&3&0&5\end{bmatrix}$$

Przydzielony zasób dla procesu 2
Macierz A przedstawia stan systemu po przydzieleniu procesowi P2 jednej jednostki zasobu II. Łącznie przydzielone zostało:

0+0+2+1+1 = 4 jednostki dla zasobu I
2+1+4+0+1 = 8 jednostki dla zasobu II
1 + 1+ 1 +2 +1 =6 jednostek dla zasobu III

Pozostało więc wolne:
To obliczenie generujemy na podstawie obliczonego w podpunkcie a) f ostatecznego, czyli ilości wolnych zasobów po zakończeniu procesów! Wartość f wynosiła wtedy:

$$f=\begin{bmatrix}5\\9\\9\end{bmatrix}, a\ nam\ po\ odjęciu\ wykorzystywanych\ zasobów\ zostało\ f1=\begin{bmatrix}5-4=1\\9-8=1\\9-6=3\end{bmatrix},$$

Czyli 5 - 4 = 1 jednostka zasobu I (pierwszego) 9 – 8 = 1 jednostka zasobu II, 9 – 6 = 3 jednostki zasobu III. Jest to wystarczające do zakończenia procesu P1. Dokonując analizy analogicznej do powyższej dochodzi się do wniosku, że wystarczy to do zakończenia procesu P3 i P5.
Z użyciem tak otrzymanego f1 dokonujemy analizy takiej jak w przypadku algorytmu Holta w podpunkcie a) i na tej podstawie zauważamy że wystarczy do zakończenia procesów P1,P2,P3 i P5 (kolejne wartości wolnych zasobów): $f2=\begin{bmatrix}3\\5\\4\end{bmatrix}, f3=\begin{bmatrix}3\\7\\5\end{bmatrix}, f4=\begin{bmatrix}4\\8\\6\end{bmatrix}$
Dla procesu P4 nie wystarcza zasobu II, jeżeli zażąda on maksymalną deklarowaną liczbę jednostek tego zasobu. Gdyby zatem proces P4 zażądał maksymalną liczbę jednostek zasobu B, a inne procesy nie zwolnią przydzielonych jednostek tego zasobu, wówczas nastąpi zakleszczenie. Jest to więc stan zagrożenia, co oznacza, że zakleszczenie jest potencjalnie możliwe. Stosując strategię unikania zakleszczenia, nie można dopuścić do takiego stanu i tym samym **nie można przydzielić dodatkowo jednostki zasobu II procesowi P2.**

Wykonując podpunkt 2 i 3 analitycznie do powyższych obliczeń. Po sprawdzeniu czy w tych przypadkach nie dochodzi do zakleszczeń naszym zadaniem jest sprawdzić czy przy połączeniu powyższych podpunktów nie powstanie sekwencja bez zakleszczenia, do tego wykorzystamy niespełniony wcześniej podpunkt 1 i spełniany podpunkt 3.
Po wykonaniu obliczeń analitycznych do algorytmu Hebermana dotaniemy 5 różnych wartości f i uzyskamy w rezultacie bezpieczną sekwencję P3,P1,P5,P2,P4. Żądanie procesu P2 może być spełnione!

Powyższe zadanie w pełni rozwiązane pojawi się niedługo w sekcji zadań.

9. *Algorytm podejścia unikania*:
    a. 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 ρ = H (najgorszy możliwy przypadek). Jeśli nie żaden przydział nie doprowadzi do deadlocku.

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

Zalety podejścia unikania:
- 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:
    c. 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:
    a. Uporządkuj jednoznacznie zbiór zasobów.
    b. Narzuć zadaniom ograniczenie na żądania przydziału zasobów, polegające na możliwości żądania zasobów tylko zgodnie z uporządkowaniem zasobów.

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

10. *Algorytm Wait-Die i Wound-Wait*:

Algorytm Wait-Die:

Rozwiązanie negujące zachowywanie zasobów:

a. Uporządkuj jednoznacznie zbiór zadań według etykiet czasowych.
b. Jeżeli zadanie P_1, będące w konflikcie z zadaniem P_2, jest starsze (ma mniejszą etykietę czasową), to zadanie P_2 jest **odrzucane (abort)** i zwalnia wszystkie posiadane zasoby. W przeciwnym razie P_1 **czeka(wait)** na zwolnienie zasobu przez P_2.

Algorytm Wound-Wait:

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

a. Uporządkuj jednoznacznie zbiór zadań według etykiet czasowych.
b. Jeżeli zadanie P_1, będące w konflikcie z zadaniem P_2, jest starsze (ma mniejszą etykietę czasową), to zadanie P_2 jest odrzucane(abort) i zwalnia wszystkie posiadane zasoby. W przeciwnym razie P_1 czeka(wait) na zwolnienie zasobu przez P_2.

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

Wady podejścia zapobiegania:

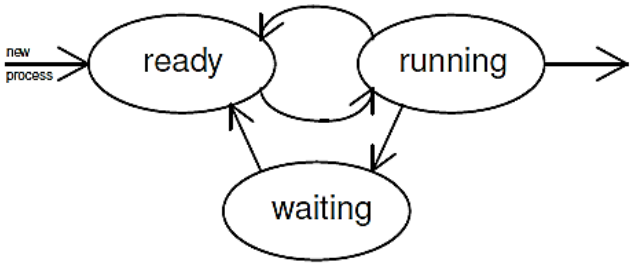
- Ograniczony stopień wykorzystania zasobów.

Zalety podejścia zapobiegania:

- Prostota i mały narzut czasowy.

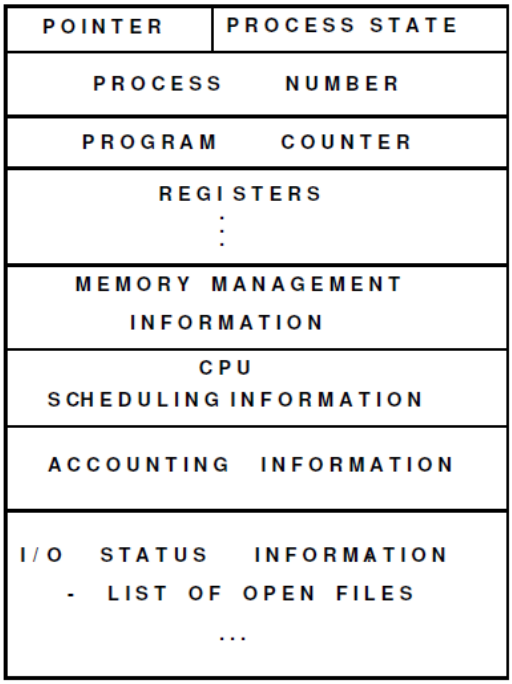
Zarządzanie procesami w systemie operacyjnym:

- Proces sekwencyjny jest działalnością wynikajacą z wykonywania programu wraz z jego danymi przez procesor sekwencyjnie. ~Alen Shaw.
- Proces jest jednostką pracy systemu. Proces jest czymś więcej niż kodem programu z określoną bieżącą czynnością. W ogólności proces obejmuje również stos zawierający dane tmczasowe (takie jak parametry procedur, adresy powrotów, zmienne tymczasowe) sekcje danych zawierające zmienne globalne oraz zestaw informacji pomocniczych. ~Abraham Silberschatz
- Proces jest wykonywanym programem wraz z bieżącymi wartościami licznika rozkazów, rejestrów i zmiennych.~Andrew Tanenbaum
- **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



Accounting information – informacja o kosztach przetrwania – czasami chcemy preferować procesy o mniejszym zapotrzebowaniu na zasoby systemu i wtedy konieczna jest informacja 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	

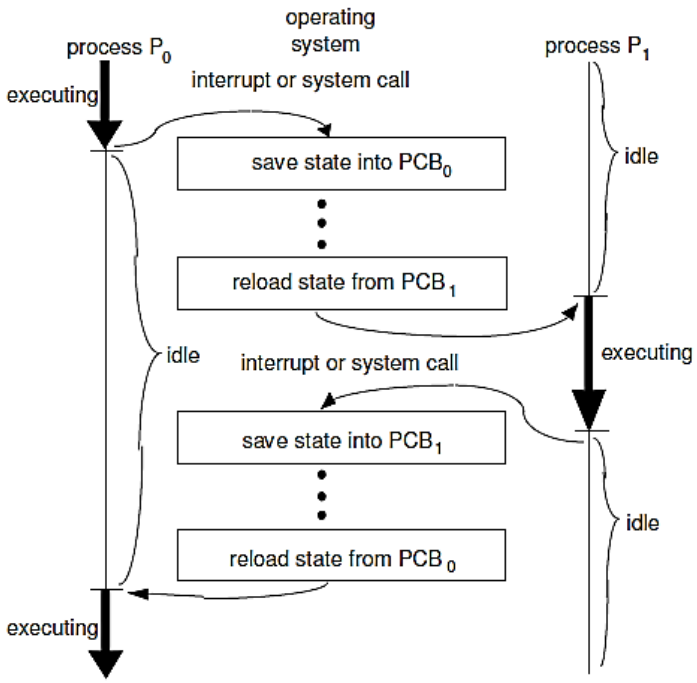
Process state – opisuje stan wykonywania procesu;
Process number – jest unikalnym identyfikatorem procesu;
Program counter i register – opisują stan procesora;
Memory managment information - to informacja opisująca obszar przydzielonej procesorowi pamięci operacyjnej;
Accounting information – to informacja opisująca rozliczenia;
CPU scheduling information – to informacja określająca priorytet procesu;
I/O status information – to informacjadotycząca oczekiwanych zdarzeń, otwartych plików itp.

Przełączenie kontekstu

- **Przełączenie kontekstu** polega na zmianie przydziału procesora. Realizowane jest ono z wykorzystaniem systemu przerwań. Załóżmy, że wykonywany jest pewien proces użytkowwnika i pojawiło się przerwanie „dyskowe”. Wówczas:
    a. Licznik rozkazów, słowo stanu i podstawowe rejestry są skłądowane na stosie systemowym przez sprzęt obsługujący przerwanie.
    b. Wykonywany jest skok do adresu wskazywanego rpzez wektor przerwań.

  Kolejne kroki są wykonywane programowo – przez odpowiednie moduły systemu operacyjnego.:

    c. Procedura obsługi przerwań rozpoczzyna się od zapamiętania wszystkich pozostałych rejestrów tablicy procesów.
    d. Numer identyfikacyjny (id) bieżącego procesu i wskaźnik do tablicy procesów są zapamiętywane pod odpowiednimi zmiennymi systemowymi.
    e. Identyfikowany jest proces, który zainicjował wykonywanie operacji dyskowej spośród procesów znajdujących się w stanie zawieszenia.
    f. Zidentyfikowany proces przechodzi w stan gotowości.
    g. Wywołany zostaje moduł szeregujący (proces scheduler) w celu zdecydowania, któremu z procesów znajdujących się w stanie gotowości ma być przydzielony procesor.



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 proces. Proces tworzący nowe procesy za pomocą funkcji systemowej utwórz(create) proces. Proces tworzący nowe procesy nazywa się **procesem macierzystym**, utworzone zaś przez niego procesy **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
- 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 wspólne ze swoimi potomkami
- Proces macierzysty oczekuje (funkcja systemowa **wait**), dopóki wszystkie jego procesy potomne nie zakończą pracy.

Do wypełnienia swoich 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:

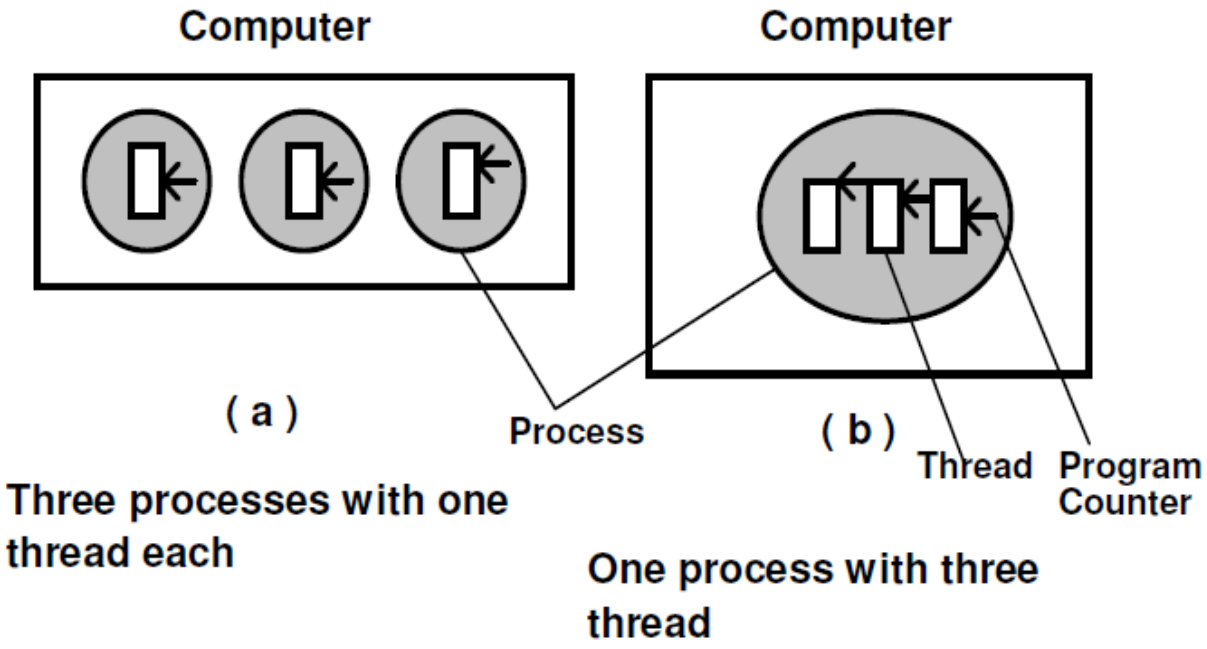
- Na jego stan nie wpływa żaden inny proces;
- Jego działanie jest deterministycznie tzn. wynik jego pracy jest zależny wyłącznie od jego stanu wejściowego (chyba, że zawiera instrukcje wyboru niedeterministycznego)
- Jego działanie jest powtarzalne, tzn. wynik pracy procesu niezależnego jest zawsze taki sam przy takich samych danych (chyba, że zawiera instrukcje wyboru niedeterministycznego)
- Jego wykonywanie może być wstrzymywane i wznawianie bez żadnych szkodliwych skutków

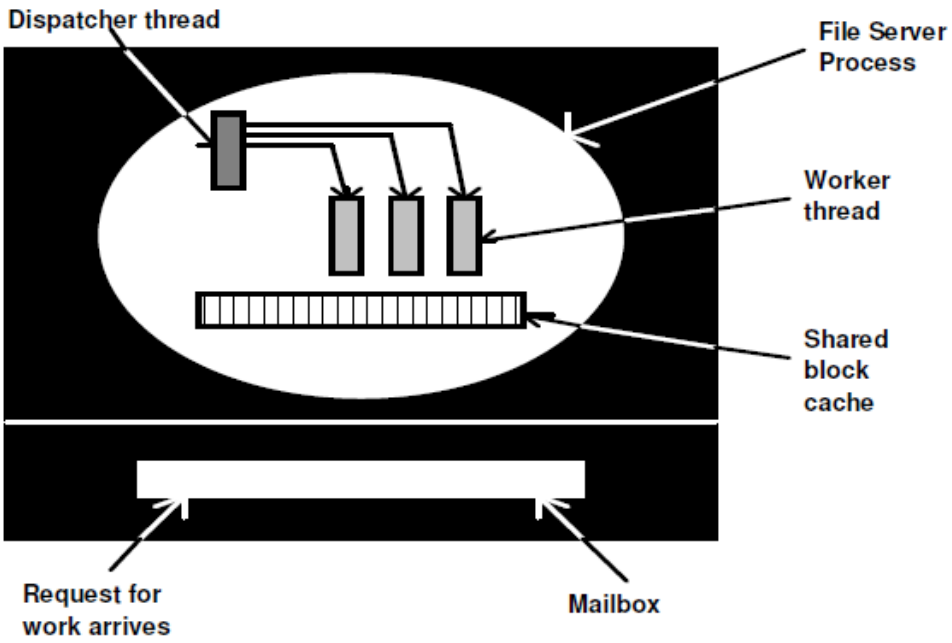
Proces współpracujący ma następujace własnosci:

- 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 wykonywania innych procesów
- Wynik działania procesu może zależeć on od względnej kolejności wykonywania wszystkich współpracujacych 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.

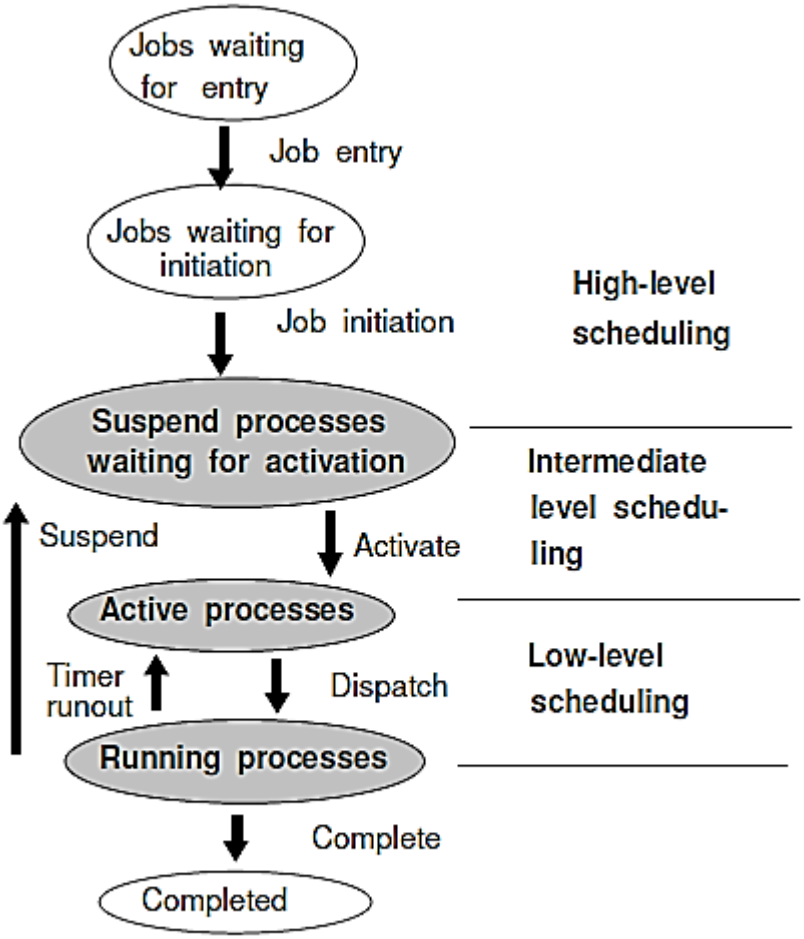


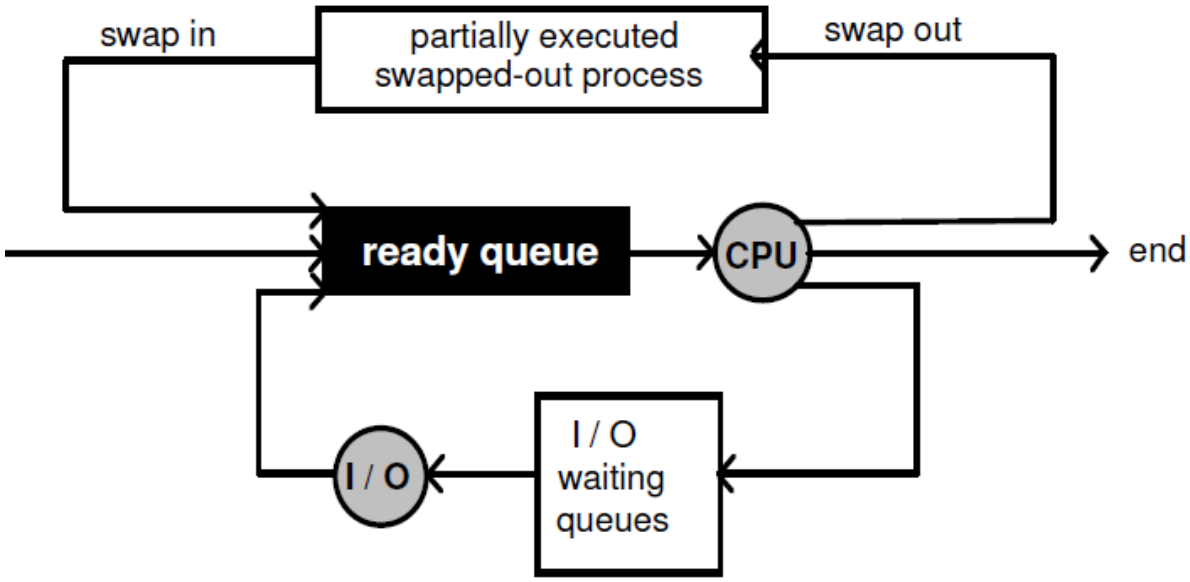


Team model- równożędne wątki

Model hierarchiczny: Dispatcher – zarządca Work – uaktywniany na żądanie zarządcy

11. Algorytm szeregowania:





12. Kryteria oceny algorytmów szeregowania:
    - Długość uszeregowania C_{max}- czas wykonania zbioru procesów
    - Przepustowość – liczba procesów wykonana w ciągu jednostki czasu
    - Śreni 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 dwiee podstawowe grupy:

- **Z wywłaszczaniem/podzielne** – procesor może być odebrany procesowi, a zawieszony proces może być kontynuowany na innym procesorze
- **Bez wywłaczania/niepodzielne** – proces utrzymuje procesor aż do zakończenia pracy

Algorytm FCTS (First Come First Served) – szereguje on zadania zgodnie z porządkiem przybywania:

- 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 (Short Job First) – szereguje zadania zgodnie z porządkiem określonym przez czasy ich wykonywania – najpierw wykonywane jest zadanie najkrótsze:

- Algorytm bez wywłaszczania
- Procesor jest przydzielany procesowi, który ma najkrótszy przewidywany czas wykonania, a więc algorytm ten faworyzuje zadnia krótki
- Udowodniono, że jest to algorytm optymalny ze względu na średni czas przebywania procesów w systemie

Problemem powyższego są:

- Brak informacji o czasie wykonania danego procesu, można sprawdzić ile czasu za ostantim razem wykonywał się ten proces, ale jest to zbędnie czasochłonne
- Może dojść do zagłodzenia procesów o bardzo długich zadań

Algorytmy priorytetowe – 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 ogół według algorytmu FCFS
- Priorytety mogą być definiowane w sposób statyczny lub dynamiczny
- Priorytety mogą być przydzielane dynamicznie po to, aby osiagnąć określone cele systemowe, np. jeśli specjalny proces zażąda przydziału procesora, powinien go otrzymać natychmiast
- Planowanie priorytetowe może być wywłączające lub niewywłaszczające
- Podstawowym problemem w planowaniu priorytetowym jest stałe blokowanie-liveloock

Algorytm rotacyjny (Round Robin) – w algorytmie rotacyjnym procesor jest przydzielany zadaniom kolejno na określony odcinek czasu:

- Kwant czasu przydziału procesora jest najczęściej rzędu 10 do 100msek
- 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 trafi na koniec kolejki

Podstawowym problemem przy konstrukcji RR jest określenie długości kwantu czasu.

Jeśli kwant czasu jest bardzo długi to upodabnia się do FIFO (dla kwantu czasu nieskończonego każde zadanie się w nim zakończy czyli będzie zwykła kolejka).
Jeżeli kwant czasu będzie bardzo krótki – upodabnia się do SJF (im krótsze zadanie tym większe prawdopodobieństwo że wyliczy się ono od razu w 1 kwancie czasu, a długie będą ciągle przeywane i odsylane na koniec).

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

Algorytmy wielopoziomowe – zbiór procesów gotowych jest rozdzielony na wiele kolejek, a szeregowane w każdej kolejce realizowane jest według określonego algorytmu:

- Procesy mogą lecz nie muszą być na stałe przypisywane do określonej kolejki
- 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 zwrotny**m możliwe jest przemieszczanie procesów między kolejkami.

Szeregowanie procesów w systemie UNIX:

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 przelicza na nowo wartości priorytetów, według następującej reguły:
nowy_priorytet = wartość_bazowa + wykorzystanie_procesora

W ramach określonej kolejki jest wykorzystywany algorytm RR. Współpracuje on z systemową procedurą timeout.

ROZWIĄZYWANIE ZADAŃ EGZAMINACYJNYCH: