PROCESY

- proces to ciąg czynności wykonywanych za pośrednictwem ciągu rozkazów, których wynikiem jest wykonanie pewnych zadań
- proces - program (pasywny), który się aktualnie wykonuje (aktywny)
- zasoby procesu: pamięć, procesor, urządzenia zewnętrzne
- proces jest dynamiczny
    - sekcja tekstu, licznik rozkazów, stos procesu, sekcja danych
- procesy:
    - współdziałają, aby uzyskać zamierzony cel,
    - współzawodniczą o korzystanie z ograniczonych zasobów, takich jak procesory, pamięć operacyjna lub pliki, są potencjalnie współbieżne
    - mechanizmy komunikowania się między procesami (kolejki, semafory, kolejki komunikatów, pamięć dzielona)

Stany procesu

- nowy
- aktywny
- oczekujący
- gotowy
- zakończony

REPREZENTACJA PROCESÓW W JĄDRZE SYSTEMU

- tablica procesów
- każdy proces reprezentowany w SO przez deskryptor procesu, nazywany też blokiem kontrolnym (PCB- ***process control block***) lub wektorem stanu

–identyfikator procesu

–stan procesu (nowy, gotowy, aktywny, oczekiwanie...)

–rejestry procesora: licznik rozkazów, rejestry ogólnego przeznaczenia i rejestry indeksowe, wskaźnik stosu

–inf. o planowaniu przydziału procesora: priorytet...

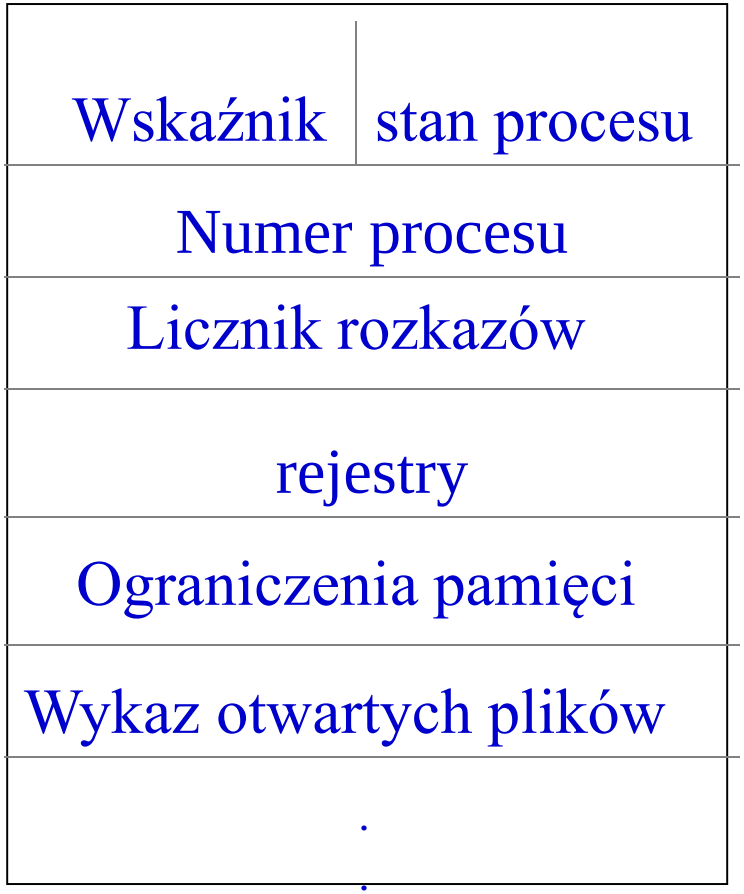
–inf. o zarządzaniu pamięcią (tablice stron, rejestry graniczne..)

–inf. do rozliczeń (czas procesora..)

–inf. o we/wy

środowisko ulotne procesu - kontekst procesu

Blok kontrolny procesu (PCB)



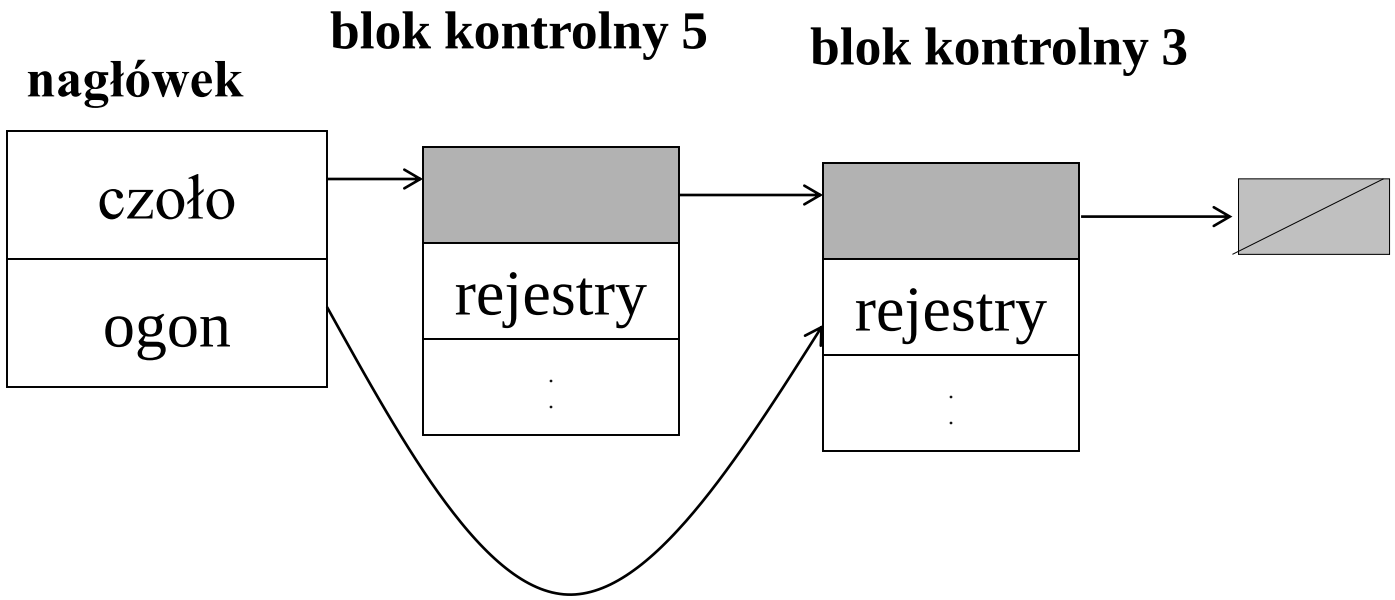
Planowanie procesów

- Kolejki zadań (*job queue*)

–kolejka procesów gotowych *(ready queue)* (lista powiązana bloków kontrolnych)

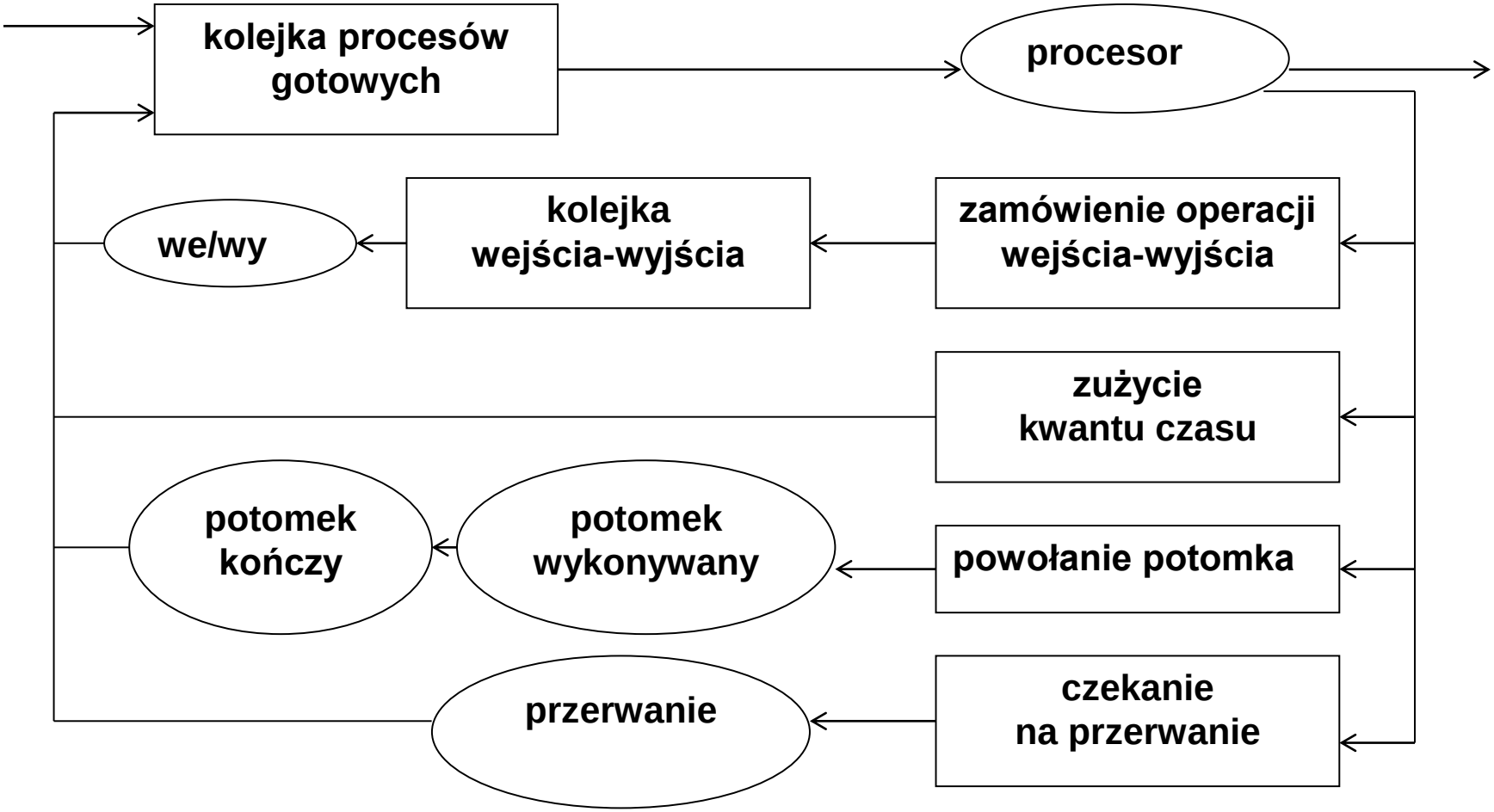
–kolejki do urządzeń *(device queue)*

(dyski, taśmy)



Kolejka procesów

Diagram kolejek w planowaniu procesów



- **planista długoterminowy** *(long term scheduler)*- planista zadań - ładuje procesy do pamięci

  (system wsadowy - wiele procesów gotowych do wykonania - na dyskach; system z podziałem czasu często nie ma planisty długoterminowego)

  nadzoruje stopień wieloprogramowości (wywoływany tylko gdy proces opuszcza system)

- **planista krótkoterminowy** *(short term scheduler)*- planista przydziału procesora *(CPU scheduler)*- wybiera jeden proces i przydziela mu procesor (min. co 100ms)

- **planista średnioterminowy** (*medium term scheduler)* umożliwia usunięcie procesu z pamięci

  zmniejszenie stopnia wieloprogramowości *(swapping)*

•Wykonanie procesu - naprzemiennie występujące cykle działań procesora i oczekiwań na op. we/wy

•częstość występowania fazy=f(czas trwania fazy)

- krzywa wykładnicza (wiele krótkich faz, mało długich)

procesy ograniczone przez we/wy - wiele krótkich faz

procesy ograniczone przez procesor - długie fazy procesora

•planowanie niewywłaszczeniowe

•planowanie wywłaszczeniowe

– **FCFS**
– **SJF**
– Planowanie priorytetowe
– Planowanie rotacyjne (**round-robin**)
– Wielopoziomowe planowanie kolejek

Kryteria planowania

- **wykorzystanie procesora** (40%-90%)
- **przepustowość** liczba procesów kończonych w jednostce czasu (10 proc/s - 1proc/1h)
- **czas cyklu przetwarzania** nadejście procesu - zakończenie procesu
- **czas oczekiwania** suma okresów czekania w kolejce proc. gotowych do wykonania
- **czas odpowiedzi** wysłanie żądania - początek pierwszej odpowiedzi

Planowanie metodą FCFS (*first come first served*)

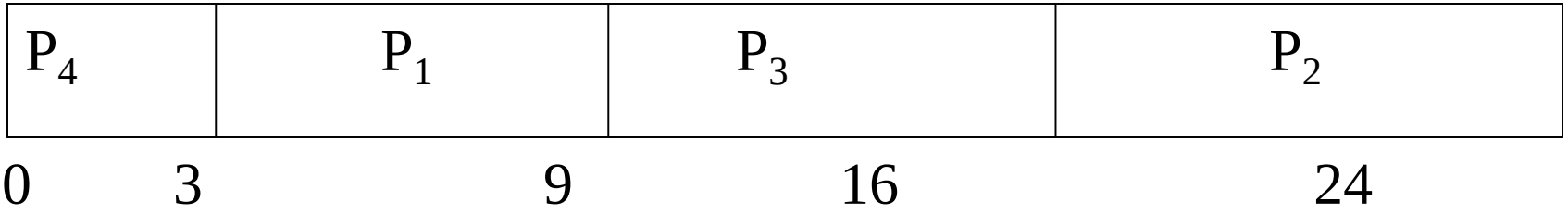
•proces, który pierwszy zamówił procesor pierwszy go otrzyma.
•Implementacja - za pomocą kolejki FIFO
•Blok kontrolny procesu wchodzącego do kolejki procesów gotowych jest dołączany do końca.
•Wolny procesor przydziela się procesowi z czoła kolejki procesów gotowych
•Średni czas oczekiwania bywa bardzo długi.

Najpierw najkrótsze zadanie (SJF *shortest job first*)

•długość najbliższej z przyszłych faz procesora (w przypadku równych faz - FCFS)
•minimalny średni czas oczekiwania
•planowanie długoterminowe

Niewywłaszczające; SJF też wywłaszczający

proces	czas trwania fazy
P_1	6
P_2	8
P_3	7
P_4	3



średni czas oczekiwania dla **SJF** = (0+3+9+16)/4=**7 ms**
dla algorytmu **FCFS** (0+6+14+21)/4=**10,25 ms**

SJF

wywłaszczający
niewywłaszczający

Czas startu	Proces	Długość fazy
0	P1	6
1	P2	4
2	P3	3
4	P4	1

Planowanie priorytetowe

- SFJ (PRI=1/dł. Fazy)
- priorytet definiowany wewnętrznie

  (limity czasu, wielkość pamięci, liczba otwartych plików)
- priorytet definiowany zewnętrznie

  (ważność procesu, opłaty, polityka)
- wywłaszczające lub nie wywłaszczające
- problem - nieskończone blokowanie

=głodzenie niskopriorytetowych procesów-

- postarzanie (*aging*) np. co 15 min PRI+=1

Planowanie rotacyjne – (round-robin)

- Systemy z podziałem czasu
- Kolejka procesów gotowych do wykonania - kolejka cykliczna
- Planista przydziału procesora przegląda tę kolejkę i każdemu procesowi przydziela kwant czasu
- Jeśli faza procesora w danym procesie przekracza 1 kwant czasu, to proces będzie wywłaszczony i wycofany do kolejki procesów gotowych
- Implementacja - kolejka FIFO
- Długi średni czas oczekiwania
- Wywłaszczający
- Duży kwant czasu - FCFS; mały - dzielenie procesora

Wielopoziomowe planowanie kolejek

- różne rodzaje procesów - różne wymagania na czasy odpowiedzi
- rozdzielenie kolejki procesów gotowych na osobne kolejki
- każda kolejka - własny algorytm planujący
- planowanie między kolejkami

Przykład: cztery kolejki

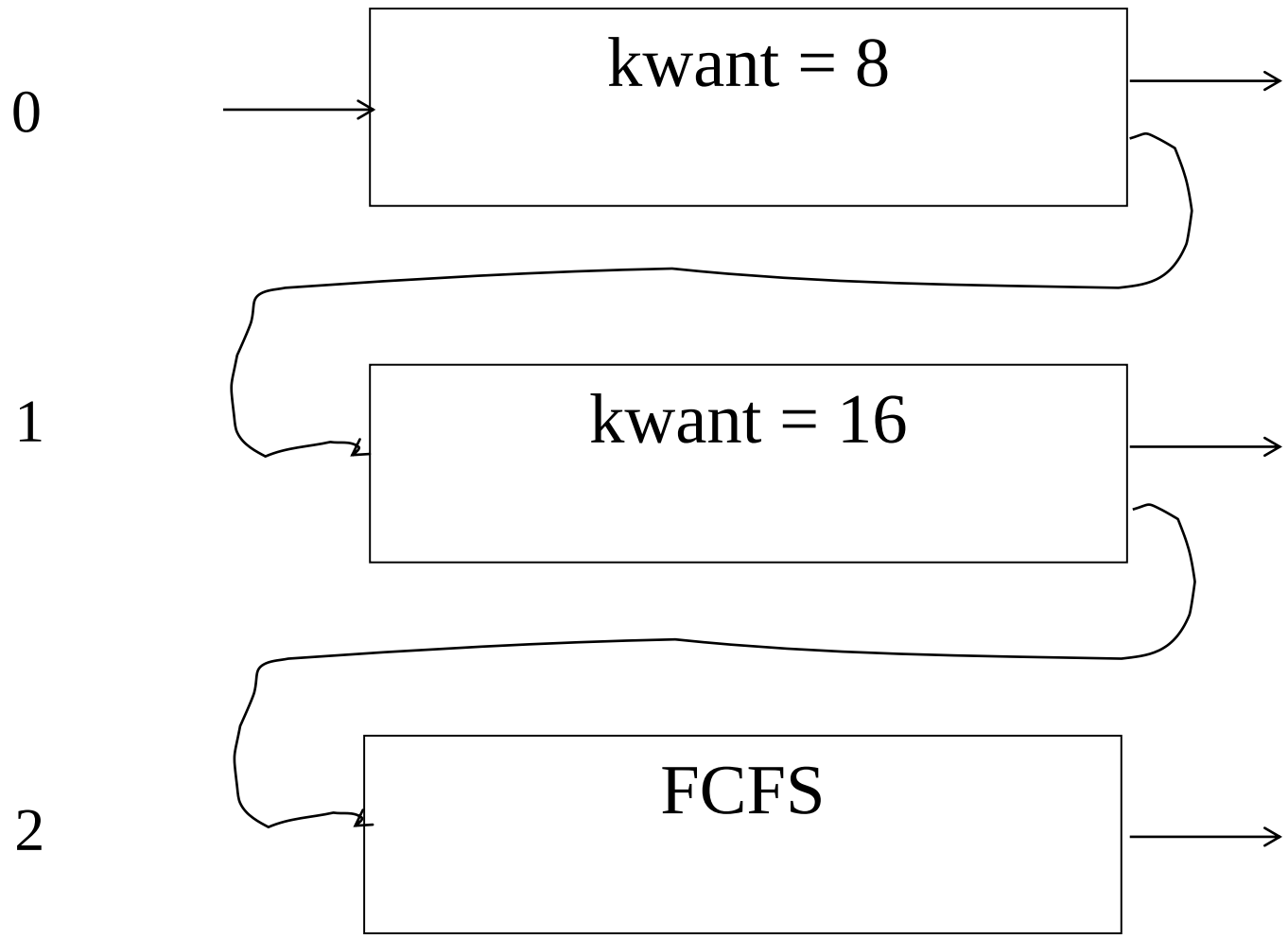
- procesy systemowe (max. priorytet)
- procesy interakcyjne
- procesy wsadowe
- procesy studenckie (min. priorytet)

- każda kolejka ma bezwzględne pierwszeństwo nad kolejkami o niższych priorytetach (żaden proces z kolejki procesów wsadowych nie może być wybrany dopóki kolejki procesów systemowych i interakcyjnych nie są puste)
- procesy nie mogą przemieszczać się między kolejkami
- niski koszt planowania, brak elastyczności

Inna możliwość - operowanie przedziałami czasu między kolejkami:
proc. pierwszoplanowe - 80% - metoda rotacyjna
proc. drugoplanowe - 20% - FCFS

Planowanie wielopoziomowe ze sprzężeniem zwrotnym

- umożliwia przemieszczanie procesów miedzy kolejkami
- rozdzielenie procesów o różnych rodzajach faz procesora
- proces zużywający za dużo czasu procesora przeniesiony do kolejki o niższym priorytecie
- pozostawienie procesów ograniczonych przez we/wy i procesów interakcyjnych w kolejkach o wyższych priorytetach
- proces oczekujący zbyt długo w niskopriorytetowej kolejce może zostać przeniesiony do kolejki o wyższym priorytecie (zapobiega głodzeniu)



Kolejki wielopoziomowe ze sprzężeniem zwrotnym

Planista wielopoziomowych kolejek ze sprzężeniem zwrotnym jest określony za pomocą następujących parametrów:

- liczba kolejek
- algorytm planowania dla każdej kolejki
- metoda wykorzystana do decydowania o awansowaniu procesu do kolejki o wyższym priorytecie
- metoda wykorzystana do decydowania o dymisjonowaniu procesu do kolejki o niższym priorytecie
- metoda określająca kolejkę, do której trafia proces potrzebujący obsługi

Planowanie wieloprocesorowe

- problem planowania pracy wielu procesorów się komplikuje
- ważnym czynnikiem - rodzaj zastosowanych procesorów
- procesory jednakowe (system **homogeniczny**);
- różne (system **heterogeniczny**)
- **różne procesory** - możliwości wyboru są ograniczone - każdy procesor ma własną kolejkę i własny algorytm planowania, procesy muszą być wykonywane przez konkretne procesory, procesy są samoistnie poklasyfikowane, a każdy procesor zajmuje się własnym planowaniem
- **jednakowe procesory** - metoda dzielenia obciążeń *(load sharing)* wspólna kolejka procesów gotowych do wykonania, przydziela się im dowolny z dostępnych procesorów

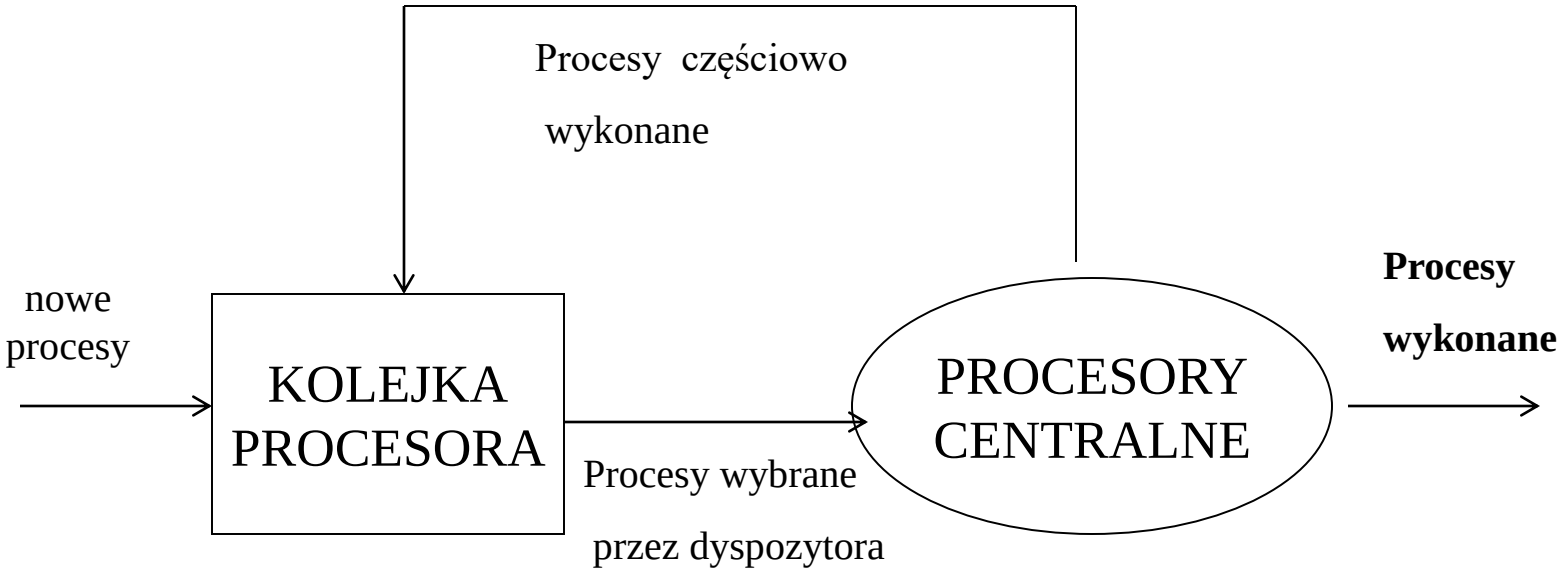
W schemacie tym można zastosować jedną z dwu metod planowania

- **każdy procesor sam planuje swoje działanie**
  - każdy procesor przegląda kolejkę procesów gotowych, z której wybiera proces do wykonania
  - należy zadbać o to, aby dwa procesory nie wybrały tego samego procesu, a także by nie ginęły procesy i kolejki.
- **wybór jednego procesora do spełniania roli planisty**

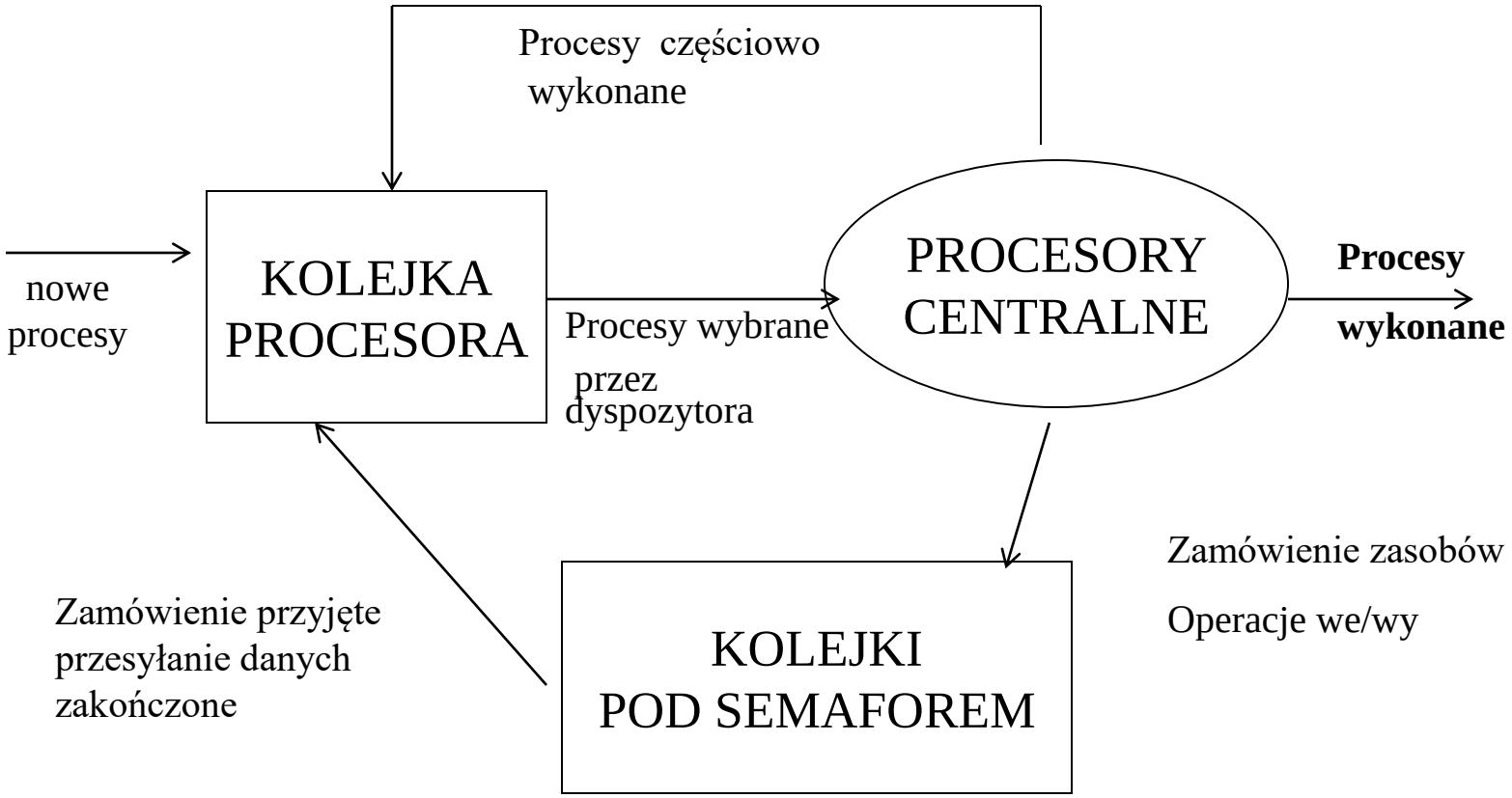
pozostałych procesorów - struktura zależności między procesami (master-slave)

wieloprzetwarzanie asymetryczne

Model planowania dla systemu ograniczonego przez procesor



Model SO, w którym procesory centralne nie są jedynymi ważnymi zasobami



Przemieszczanie procesów

- komputer wieloprogramowy - równoczesne wykonywanie wielu programów
- cała dostępna pamięć operacyjna - rozdzielona między pewną liczbę procesów
- żaden programista nie może wiedzieć z góry, jakie inne programy będą się znajdować w pamięci podczas wykonywania jego programu
- obszar pamięci przydzielony procesowi może ulegać zmianom podczas przebiegu procesu
- z chwilą zakończenia procesu zajmowaną przez niego pamięć można przeznaczać dla innych procesów, a wtedy pewne procesy trzeba przemieszczać w pamięci w celu lepszego jej wykorzystania.

Ochrona zawartości pamięci

zapewnienie nienaruszalności procesów – zasada:

żaden proces nie może zmienić zawartości komórek pamięci, które są bieżąco przydzielone innemu procesowi

Dostęp do obszarów dzielonych

- istnieją sytuacje, w których kilka procesów powinno mieć możliwość dostępu do tych samych obszarów pamięci operacyjnej

– jeżeli pewna liczba procesów wykonuje ten sam program - wszystkie procesy powinny posługiwać się tą samą kopią danego programu,

–kilka procesów dzieli jedną strukturę danych, a więc muszą one uzyskać wspólny dostęp do tego obszaru w pamięci, w którym znajduje się wspólna struktura danych

RODZAJE KOMUNIKACJI MIĘDZY PROCESAMI

- Wzajemne wyłączanie
- Synchronizacja
- Niedopuszczanie do blokad

RODZAJE KOMUNIKACJI MIĘDZY PROCESAMI - wzajemne wyłączanie

- Zasoby systemowe:

–podzielne (jednostki centralne, pliki przeznaczone do czytania, obszary pamięci chronione przed wprowadzeniem do nich zmian)

- działanie procesu może być przerwane, a zasób przekazany innemu procesowi, a potem ponownie kontynuowane

–niepodzielne (większość urządzeń zewnętrznych, pliki zapisywalne, obszary danych, które ulegają zmianom)

- Wzajemne wyłączanie - zapewnienie takich warunków działania systemu, by tylko jeden proces na raz mógł korzystać z zasobów niepodzielnych

Problem sekcji krytycznej

PROCESY:
P1: .... x:=x+1;....
P2:.....x:=x+1;..... **x - wspólna zmienna**

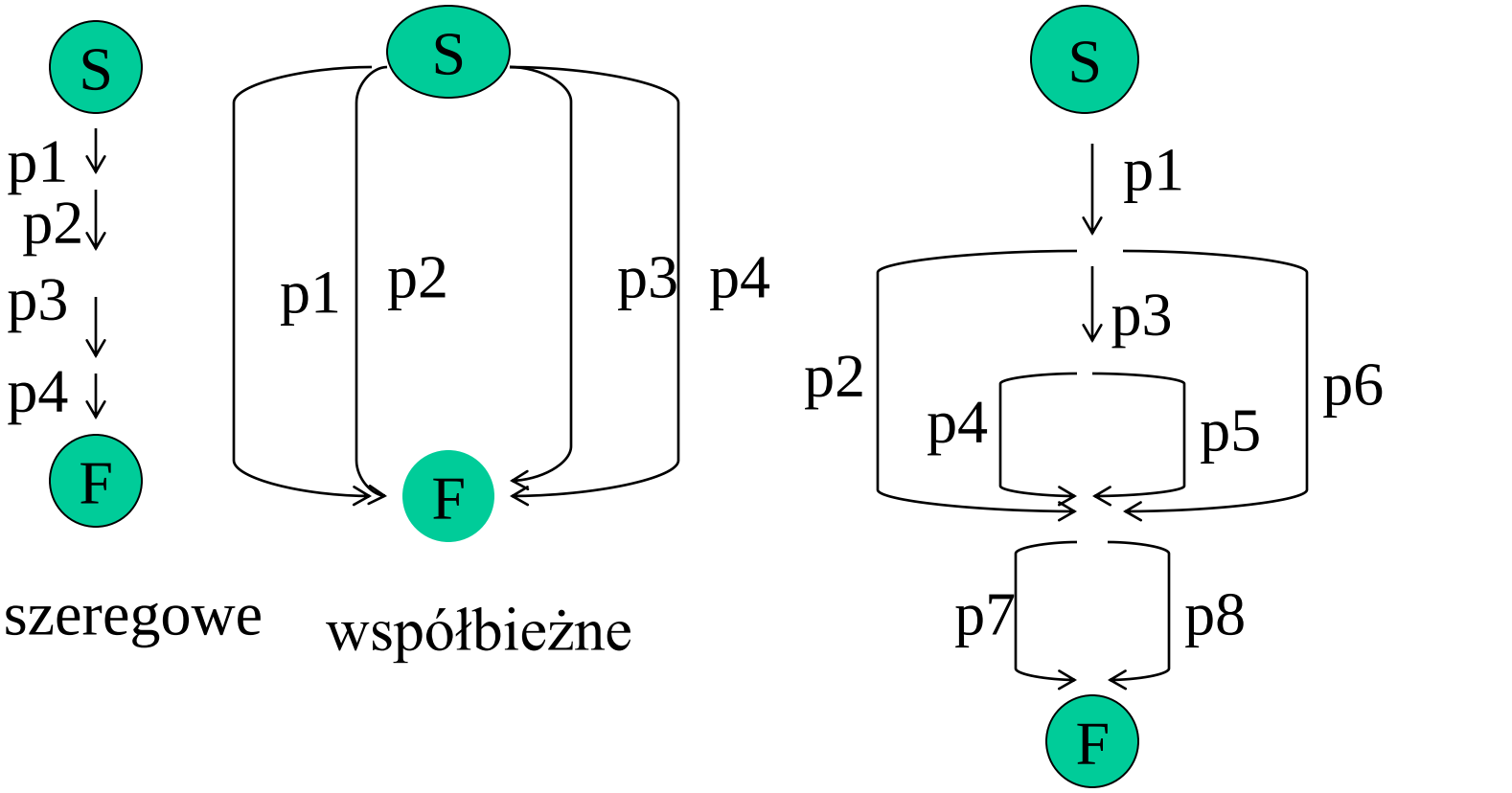
Sekwencja1:
P1:R1:=x ; R1:=R1+1 ; x:=R1;.....
P2:.....R2:=x;.....R2:=R2+1; x:=R2....

Sekwencja2:
P1:R1:=x ; R1:=R1+1 ; x:=R1;.....
P2:.....R2:=x; R2:=R2+1; x:=R2

RODZAJE KOMUNIKACJI MIĘDZY PROCESAMI - synchronizacja

- Nie można przewidzieć jaka będzie szybkość jednego procesu w stosunku do drugiego procesu,
- procesy przebiegają względem siebie asynchronicznie.
- Wyznacza się pewne punkty, w których procesy muszą synchronizować swoje działanie.
- Zadaniem systemu operacyjnego jest stworzenie mechanizmów umożliwiających synchronizację procesów.

Relacje pierwszeństwa między procesami



RODZAJE KOMUNIKACJI MIĘDZY PROCESAMI - blokada *(deadlock)*

- Blokada
  - –kilka procesów współzawodniczy o zasoby
  - –każdy proces aby móc nadal działać musi skorzystać z tych zasobów, których używa inny proces
  - –żaden proces nie może kontynuować swej pracy

- zadanie systemu operacyjnego - niedopuszczanie do powstania blokady lub ograniczanie jej skutków

Zakleszczenie (deadlock)

Proces A	Proces B
request(x)	request(y)
.	.
.	.
.	.
request(y)	request(x)
.	.
.	.
.	.
release(y)	release(x)
release(x)	release(y)

Warunki jakie musi spełniać wzajemne wykluczanie

- Symetria i równouprawnienie procesów - procesy są traktowane jako równoważne, nie mogą mieć przypisanych statycznych priorytetów, wszystkie decyzje dotyczące wyboru procesów muszą być względem nich sprawiedliwe
- Niezależność szybkości procesów - nie wolno czynić żadnych założeń dotyczących względnej szybkości.
- Skończony czas podejmowania decyzji - jeżeli więcej niż jeden proces chce wejść do sekcji krytycznej, to decyzja o tym, który proces zostanie wybrany musi być podjęta w skończonym czasie.
- Niezależność zaproponowanego mechanizmu od działania procesu poza sekcją krytyczną.

Synchronizacja 1
użycie zmiennej licznikowej

```
var x:integer:=0;
procedure P;
begin
    cycle
    instrukcje-A;
l1: x:=x+1;
    if x<>1 then x:=x-1; goto l1;  /we
    sekcja–krytyczna;
    x:=x-1;                                  /wy
    instrukcje-B
    end
end
```

WKW wejścia do s-k:

x 0->1

Problem: 2 równoległe procesy => x=2; żaden z nich nie wchodzi do s-k

Synchronizacja 2
użycie zmiennych lokalnych *l*

```
var x:integer:=0;
procedure P;
var l:integer:=1;
begin
    cycle
    instrukcje-A;
l1: x:=:l;                    /niepodzielna wymiana ;   we
    if l<>0 then  goto l1;    sekcja–krytyczna;
    x:=:l;                          /wy
    instrukcje-B
    end
end
```

WKW wejścia do s-k:

x 0->1

Problem: aktywne oczekiwanie

Synchronizacja 3

rozwiązanie Dijkstry – naprzemienne wykonywanie procesów

```
var x:integer:=2;
procedure P1;
begin
    cycle
    l1:  if x=2 then  goto l1
    sekcja–krytyczna;
    x:=2;
    instrukcje-A1;
    end
end
```

```
procedure P2;
begin
    cycle
    l2:  if x=1 then  goto l2
    sekcja–krytyczna;
    x:=1;
    instrukcje-A2;
    end
end
```

Problem: możliwe wstrzymywanie procesów poza s-k

Synchronizacja 4 - rozwiązanie Dekkera

```
var x:integer:=1;
    boolean c1,c2;
    c1:=c2:=true
procedure P1;
begin
    cycle
    a1:   c1:=false;
    l1:   if not c2 then
          begin if x=1 then goto l1;
             c1:=true;
    b1:     if x=2 then goto b1;
             goto a1
          end
    sekcja–krytyczna;
    x:=2; c1:=true;
    instrukcje-A1;
    end
end
```

```
procedure P2;
begin
    cycle
    a2:   c2:=false;
    l2:   if not c1 then
          begin if x=2 then goto l2;
             c2:=true;
    b2:     if x=1 then goto b2;
             goto a2
          end
    sekcja–krytyczna;
    x:=1; c2:=true;
    instrukcje-A2;
    end
end
```

Algorytm piekarniowy

- Synchronizuje 2 lub więcej procesów
- Proces, który chce wejść do SK bierze bilet (numerowany wg kolejności)
- Proces czeka, aż jego bilet będzie miał najniższą wartość

OPERACJE SEMAFOROWE

•Rok 1965, - Dijkstra'e wprowadza pojęcia semaforów oraz wykonywanych na nich operacji czekaj (wait) i sygnalizuj (signal).

•Semafor jest to nieujemna liczba całkowita, na której z wyjątkiem nadawania wartości początkowych mogą działać jedynie operacje ***czekaj*** i ***sygnalizuj***

Operacja sygnalizuj - V(s) - zwiększenie wartości semafora s o 1

• niepodzielna operacja

• zmienna semaforowa niedostępna dla innych procesów

Operacja czekaj P(s) - zmniejszenie wartości semafora s o 1, o ile to możliwe

• operacja niepodzielna

• może spowodować wstrzymanie jakiegoś procesu

- Jednoczesne wywołanie operacji P lub V na tym samym semaforze – operacje wykonane sekwencyjnie w dowolnym porządku
- Wybór procesu czekającego na dokończenie operacji P – arbitralny

OPERACJE SEMAFOROWE – sekcja krytyczna

czekaj (nazwa semafora)

sekcja krytyczna

sygnalizuj (nazwa semafora)

- Jeżeli sekcja jest wolna - proces wchodzi do niej bezpośrednio
- Jeżeli sekcja jest zajęta - proces jest czasowo zawieszany
- Po zwolnieniu sekcji krytycznej przez proces, sprawdza się, czy nie ma zawieszonych procesów oczekujących na wejście do tej sekcji.

Implementacja operacji semaforowych

P(s):

if s=0 then

umieszczenie procesu w kolejce semafora

end

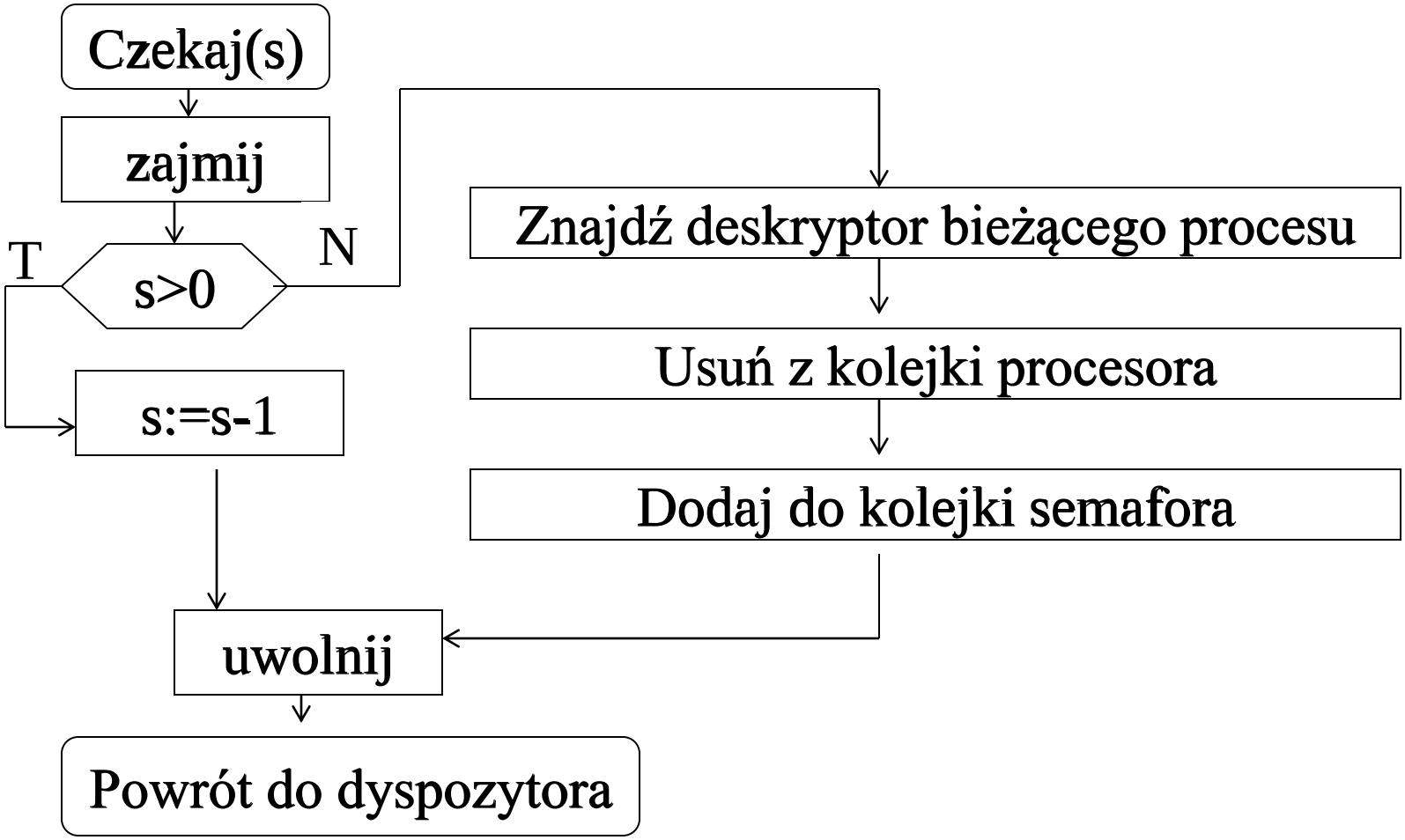
s:=s-1

V(s):

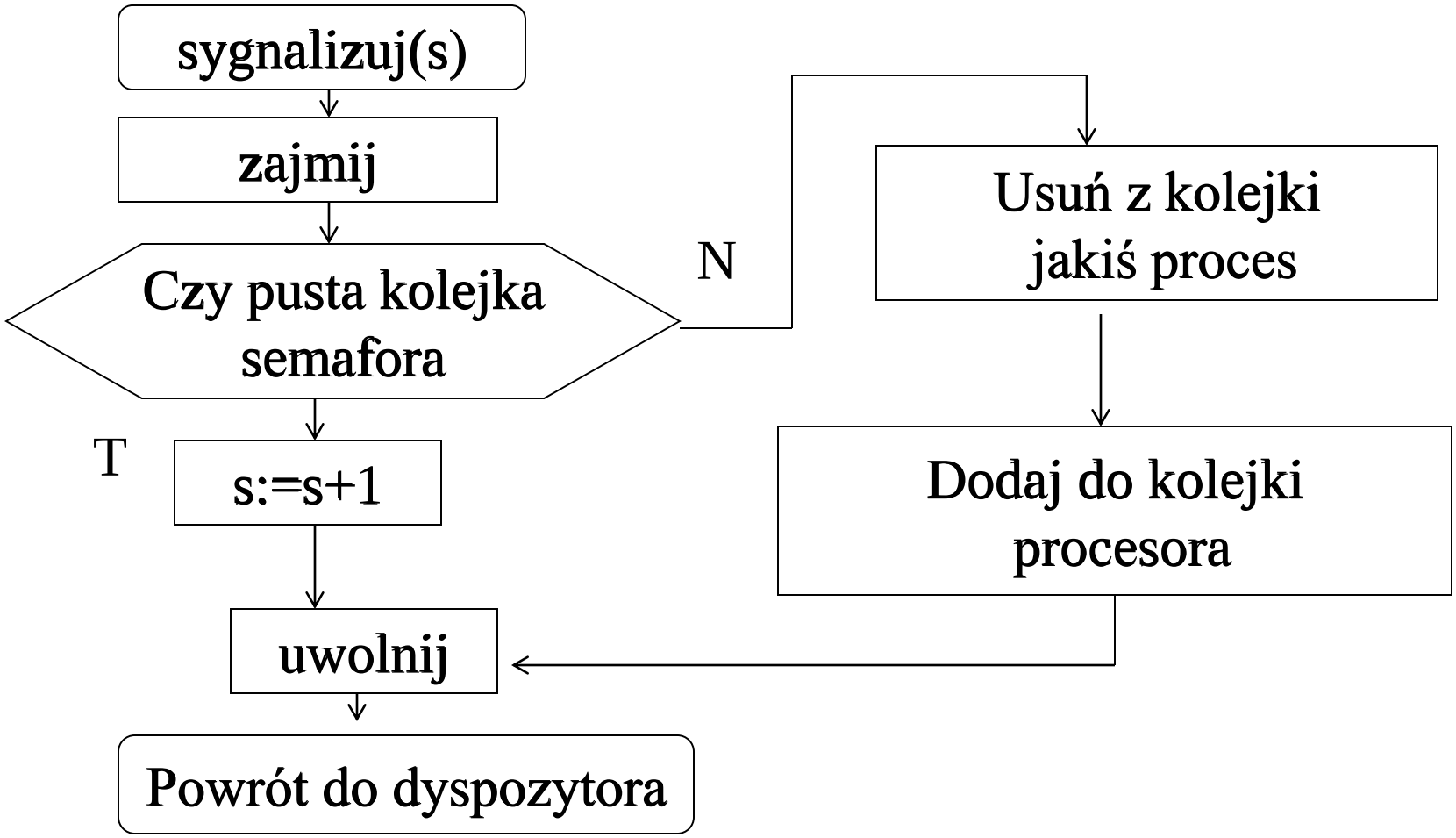
s:=s+1

if kolejka semafora nie jest pusta then reaktywuj proces

Implementacja operacji semaforowej czekaj - P



Implementacja operacji semaforowej sygnalizuj V



SEMAFORY BINARNE

–mogą przyjmować dwie wartości: 0, 1

–rozwiązanie problemów wzajemnego wykluczania

ZASTOSOWANIE SEMAFORÓW

- problemu wykluczenia wzajemnego
- problemy synchronizacji
- problemy komunikacji

Rejony krytyczne

•Zmienna dzielona v - obiekt typu *T*, na którym są wykonywane operacje wewnątrz sekcji krytycznej

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

•rejon krytyczny (sekcja krytyczna dla instrukcji I_1,..I_N)

region v **do** I_1;...;I_N **end**;

•wewnątrz rejonów krytycznych związanych z tą samą zmienną dzieloną może pracować tylko 1 proces

•wewnątrz rejonu krytycznego proces przebywa w skończonym czasie

•we. do rejonu krytycznego musi być możliwe dla dowolnego procesu w skończonym czasie

Warunkowe rejony krytyczne

var R: **shared** *T*;

region R **do**

$I_1;...;I_N$; await W_1;

.....

$I_i;...;I_{i+1}$; await W_j;

.....

end;

•uproszczone postacie:

with R **when** W **do** I; **region when** W **do** I; spr. war. przy we.

region R **do** I **await** W; spr. war. po wy.

monitory

•skoncentrowanie w 1 m-cu deklaracji wspólnych zmiennych dzielonych i operacji na nich działających

•dostęp do nich - poprzez wywołanie procedury monitora

•wzajemne wykluczanie kolejnych odwołań do procedur monitora – w jego wnętrzu aktywny tylko jeden proces

var id-monitora: **monitor**

deklaracje zmiennych;

procedury;

lista-udostepnionych nazw procedur;

end.

MECHANIZMY SYNCHRONIZACJI PROCESÓW

- **Synchronizacja programowa**

–Niepodzielne operacje (zamiana, testuj i ustal)
–Zakaz przerwań (środ. 1-procesorowe, wieloprocesorowe)
–Aktywne oczekiwanie (busy waiting)
– wirujaca blokada (spinlock)

- **Semafory**

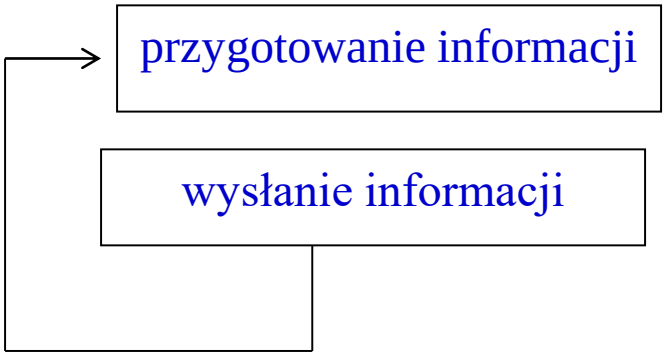
–Mechanizm pierwotny
–Nie wymaga aktywnego oczekiwania
–Nie jest strukturalny
–Możliwość błędu w systemie (blokada)

- **Strukturalne mechanizmy synchronizacji**

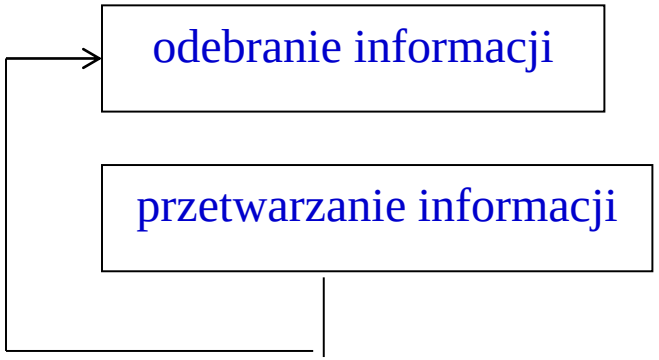
Producent - konsument

W systemie pracuje P (P>=1) procesów producenta i K (K>=1) procesów konsumenta. Każdy proces producenta przygotowuje porcję informacji, a następnie przekazuje ją procesowi konsumenta.

procedura producent:



procedura konsument



PRODUCENT - KONSUMENT

- Zakładamy, że operujemy na puli n buforów, z których każdy mieści jedną jednostkę.
- Semafor s_1 umożliwia wzajemne wyłączanie dostępu do puli buforów i ma początkową wartość 1 ($s_1=1$).
- Semafory pusty i pełny zawierają odpowiednio liczbę pustych i pełnych buforów.
- Semafor pusty ma wartość początkową n (pusty=n).
- Semafor pełny ma wartość początkową 0 (pełny=0).

PRODUCENT - KONSUMENT

producent

begin

repeat

produkowanie jednostki

wait (pusty)

wait (s_1)

dodanie jednostki do bufora

signal (s_1)

signal (pełny)

end

konsument

begin

repeat

wait (pełny)

wait (s_1)

pobranie jednostki z bufora

signal (s_1)

signal (pusty)

end

Zrealizuj zadanie synchronizacji

-procesy A, B, C wykonują następujące zadania (taski) w każdym przebiegu otwartej pętli. Proces A składa się z zadań t_{11} i t_{12}, proces B zawiera zadanie t_{21}, proces C natomiast zadania t_{31} oraz t_{32}.

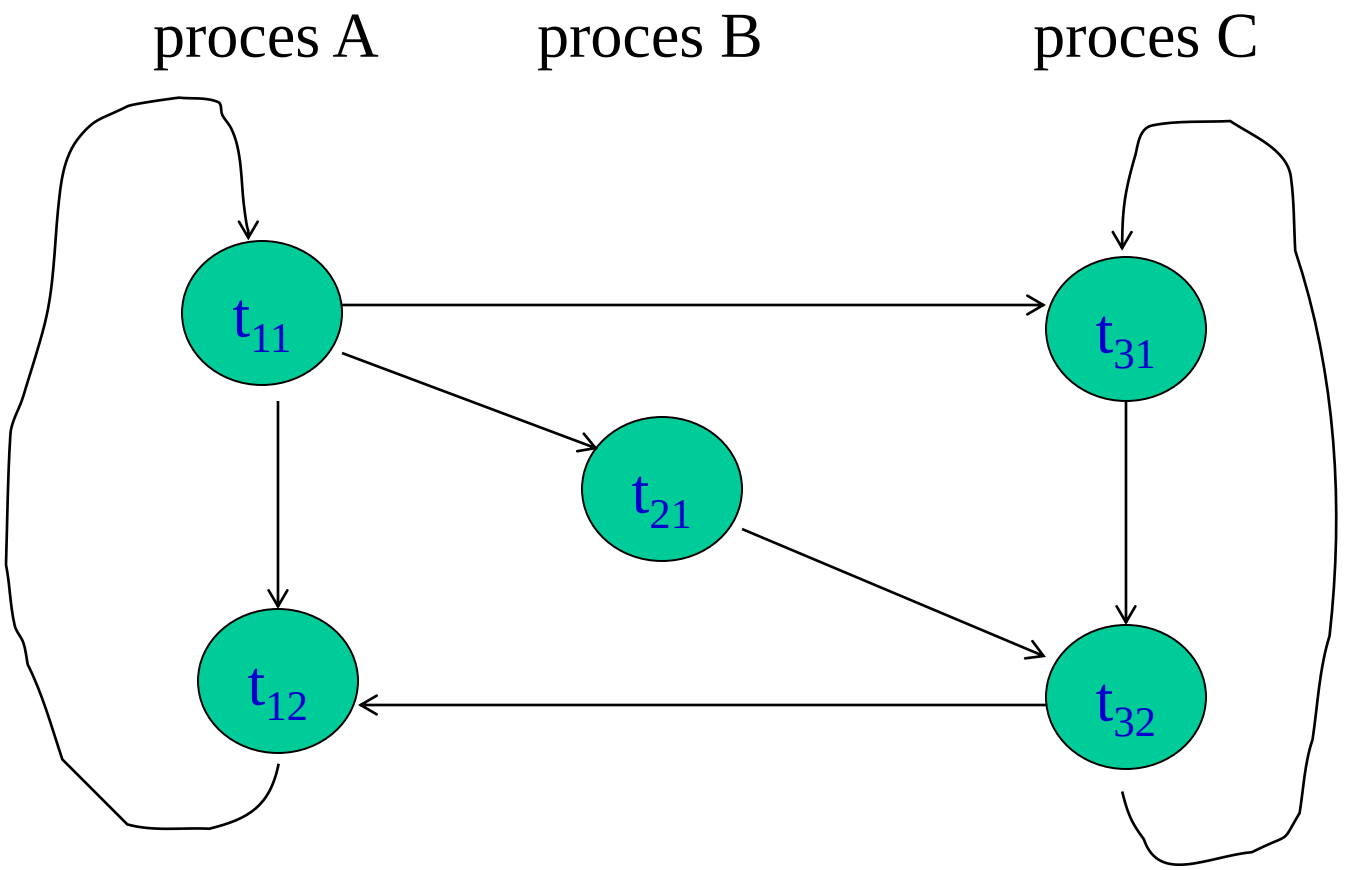
-Zbiór tasków $T=\{t_{11},t_{12},t_{21},t_{31},t_{32}\}$ jest częściowo uporządkowany relacją R określoną na zbiorze G

-$G = \{(t_{11}, t_{21}), (t_{11}, t_{31}), (t_{21}, t_{32}), (t_{32}, t_{12})\}$.

-Przy użyciu semaforów zsynchronizuj procesy A, B, C tak, aby w każdym przebiegu pętli wykonanie tasku $a \in T$ poprzedzało wykonanie $b \in T$, gdy $(a, b) \in G$

-Rozwiązanie przedstaw w postaci pseudokodów dla procesów A, B, C.

$$G = \{(t_{11}, t_{21}), (t_{11}, t_{31}), (t_{21}, t_{32}), (t_{32}, t_{12})\}$$



$s_1=0$; $s_2=0$; $s_3=0$ - wartości początkowe semaforów

Proces A

begin

repeat

do t_{11}

signal (s_1)

signal (s_2)

wait (s_4)

do t_{12}

end

Proces B

begin

repeat

wait (s_1)

do t_{21}

signal (s_3)

end

Proces C

begin

repeat

wait (s_2)

do t_{31}

wait (s_3)

do t_{32}

signal (s_4)

end

Problem czytelników i pisarzy

Dwie grupy procesów silnie konkurujących o zasoby:

•piszący

•czytający

piszący - muszą mieć zapewnione wykluczenie wzajemne względem siebie oraz względem procesów czytających przy korzystaniu z zasobu.

czytający - wiele procesów może jednocześnie być w posiadaniu zasobu, przy czym nie może z niego wtedy korzystać żaden z procesów piszących

W rozwiązaniu zastosowano dwa semafory binarne:
sp - dla zapewnienia wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów
w - dla zapewnienia wykluczenia wzajemnego procesowi czytającemu w chwilach rozpoczynania i kończenia korzystania z zasobu

Faworyzowane są procesy czytające - uzyskują one bezzwłoczny dostęp do zasobu z wyjątkiem chwil, w których korzysta z niego proces piszący.
sp=w=1

Czytanie

```
begin
repeat
  wait (w)
  lc=lc+1
  if lc=1 then wait (sp) end
  signal (w)
  czytanie
  wait (w)
  lc=lc-1
  if lc=0 then signal (sp) end
  signal (w)
end
```

Pisanie

```
begin
repeat
  wait (sp)
   pisanie
  signal (sp)
end
```

modyfikacja

- priorytet dla procesów piszących
- semafory:

w1 - wykluczenie wzajemne proc. czytających w chwili rozpoczynania i kończenia korzystania z zasobu,

sp - wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów

sc - ochrona we. do sekcji kryt. procesu czytającego

w2 - wykluczenie wzajemne proc. piszących w chwili rozpoczynania i kończenia korzystania z zasobu

w3 - zapewnienie priorytetu pisania nad czytaniem

w1=w2=w3=sc=sp=1

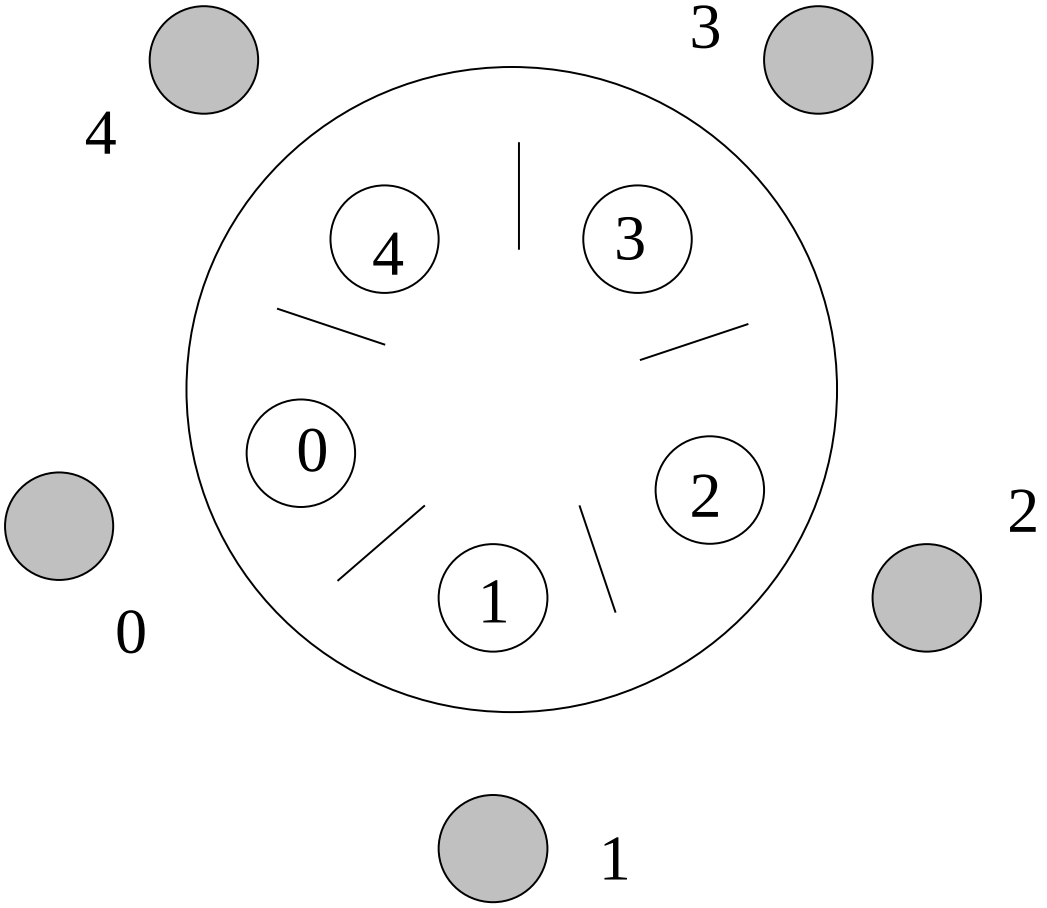
Czytanie

```
begin
repeat
  wait (w3)
     wait (sc)
        wait(w1)
           lc=lc+1
           if lc=1 then wait (sp) end
        signal (w1)
     signal(sc)
 signal(w3)
czytanie
wait (w1)
  lc=lc-1
  if lc=0 then signal (sp) end
signal (w1)
end
```

Pisanie

```
begin
repeat
 wait(w2)
    lp=lp+1
    if lp=1 then
      wait(sc) end
 signal (w2)
 wait(sp)
pisanie
signal(sp)
 wait(w2)
lp=lp-1;
if lp=0 then
            signal(sc) end;
signal(w2)
end
```

Pięciu filozofów



Filozof

begin

repeat

myślenie;

request (widelec[name], widelec[(name+1)mod5]);

jedzenie;

release(widelec[name], widelec[(name+1)mod5]);

end

Pięciu filozofów (1)

```
var widelec: array[0..4] of recource;
        sem:array[0..4] of Boolean semaphore:=true;
procedure filozof;
begin
repeat
      myślenie;
      wait(sem[name]);
      wait(sem[(name+1)mod 5]);
            request (widelec[name],widelec[(name+1)mod5]);
            jedzenie;
            release(widelec[name], widelec[(name+1)mod5]);
      signal(sem[name]);
      signal(sem[(name+1)mod 5]);
end
end
```

Pięciu filozofów (2)

stany

stan[i]=0 – myślenie
stan[i]=1 – chęć jedzenia
stan[i]=2 – jedzenie

```
var widelec: array[0..4] of recource;
          sem: array[0..4] of Boolean semaphore:=false;
          stan: array[0..4] of integer:=0;          /myślenie
          w:    Boolean semaphore:=true;


procedure test(k:integer);
begin
        if stan[(k-1) mod 5]<>2 and stan[k]=1 and
                        stan[(k+1) mod 5]<>2
        then
                stan[k]:=2;
        signal(sem[k]);
   end
end
```

Pięciu filozofów (2)

```
procedure filozof;
begin
repeat
        myślenie;
        wait(w);
                stan[name]:=1;
                test(name);
        signal(w);
        wait(sem[name]);
                request (widelec[name],widelec[(name+1)mod5]);
                jedzenie;
                release(widelec[name],widelec[(name+1)mod5]);
        wait(w);
                stan[name]:=0;
                test((name+1)mod 5);
                test((name-1)mod 5);
        signal(w)
end
end;
```

Pięciu filozofów (3) jadalnia

```
var widelec: array[0..4] of recource;
        sem:array[0..4] of Boolean semaphore:=true;
        jadalnia: semaphore:= 4;
procedure filozof;
begin
repeat
      myślenie;
      wait(jadalnia);
      wait(sem[name]);
      wait(sem[(name+1)mod 5]);
            request (widelec[name], widelec[(name+1)mod5]);
            jedzenie;
            release(widelec[name], widelec[(name+1)mod5]);
      signal(sem[name]);
      signal(sem[(name+1)mod 5]);
      signal(jadalnia);
end
end
```

Pięciu filozofów (4)

asymetryczne

```
var widelec: array[0..4] of recource;
        sem:array[0..4] of Boolean semaphore:=true;
procedure filozof4;
begin
repeat
      myślenie;
      wait(sem[0]);
      wait(sem[4]);
            request (widelec[0], widelec[4]);
            jedzenie;
            release(widelec[0], widelec[4]);
      signal(sem[4]);
      signal(sem[0]);
end
end
```

Pozostali filozofowie wykonują kod (1)

Pięciu filozofów

(1)– prawidłowa synchronizacja;
możliwość zakleszczenia
(2) - prawidłowa synchronizacja;
bez zakleszczenia;
możliwość zagłodzenia
(3) - prawidłowa synchronizacja;
bez zakleszczenia;
bez zagłodzenia
(4) - prawidłowa synchronizacja;
bez zakleszczenia;
bez zagłodzenia

Inne rozwiązania
- rozszerzone operacje semaforowe,
- monitory
- sieci Petriego

Warunki implementacji operacji semaforowych

- Zapewnienie nieprzerywalności operacji semaforowych
- Istnienie operacji zawieszenia i reaktywowania procesów poprzez bezpośredni dostęp do deskryptorów
- Istnienie operacji kolejkowych

Realizacja:

- programowa (so)
- sprzętowa (mikroprogramowe układy sterowania)

Modyfikacje operacji semaforowych

- Rozszerzone operacje semaforowe
  - Jednoczesne
  - Uogólnione
  - Jednoczesne uogólnione
  - Agerwali
  - inne

Jednoczesne operacje semaforowe

PD(s1,s2,…si,…,sn)

zawieszenie procesu do czasu,

gdy dla wszystkich *si (i=1,…,n): si >0*

for *i:=1* **to** *n* **do** *si:=si-1* **end;**

VD(s1,s2,…si,…,sm)

for *j:=1* **to** *m* **do** *sj:=sj+1* **end;**

Pięciu filozofów (5)

```
var widelec: array[0..4] of recource;
        sem:array[0..4] of Boolean semaphore:=true;
procedure filozof;
begin
repeat
      myślenie;
      PD(sem[name],sem[(name+1)mod 5]);
            request (widelec[name],widelec[(name+1)mod5]);
            jedzenie;
            release(widelec[name], widelec[(name+1)mod5]);
       VD(sem[name],sem[(name+1)mod 5]);
end
end
```

Uogólnione operacje semaforowe

n – nieujemne całkowite wyrażenie

PN(s, n)

zawieszenie procesu do czasu, gdy *s>=n*

s:=s-n;

VN(s)

s:=s+n;

Czytający - piszący

```
var w:semaphore:=M;

procedure czytanie;
begin
repeat
        PN(w,1);
           czytanie;
        VN(w,1)
end
end
```

```
procedure pisanie
begin
repeat
   PN(w,M);
      pisanie
   VN(w,M)
end
end
```

pisanie – możliwe, gdy w=M (nie ma odczytu ani zapisu)

czytanie – możliwe, gdy w>=0 (nie ma zapisu)

Czytający – piszący (2)

```
var w,r:semaphore:=M;

procedure czytanie;
begin
repeat
        PN(r,M);
        PN(w,1);
        VN(r,M-1);
        VN(r,1);
         czytanie;
        VN(w,1)
end
end
```

```
procedure pisanie
begin
repeat
   PN(r,1);
   PN(w,M);
    pisanie;
   VN(w,M);
    VN(r,1);
end
end
```

pisanie – wyższy priorytet

Jednoczesne uogólnione operacje semaforowe Cerf, Presser

PA(s1,a1,s2,a2,…si,ai,sn,an)

zawieszenie procesu do czasu,

gdy dla wszystkich *si (i=1,…,n): si > ai*

for *i:=1* **to** *n* **do** *si:=si-ai* **end;**

VA(s1,a1,s2,a2,…si,ai,sm,am)

for *j:=1* **to** *m* **do** *sj:=sj+aj* **end;**

operacje semaforowe Agerwali

PE(s1,s2,…si,…sn;~s_{n+1},…,~s_j,…,~s_{n+m})

zawieszenie procesu do czasu, gdy

dla wszystkich *si (i=1,…,n): si > 0*

i dla wszystkich s_j *(j=n+1,…,n+m):* s_j *= 0*

for *i:=1* **to** *n* **do** *si:=si-1* **end;**

VE(s1,s2,…sk,…,sl)

for *k=1* **to** *l* **do** *sk:=sk+1* **end;**

Czytający – piszący (3)

```
var A,R,M:semaphore:=0,0,1;

procedure czytanie;
begin
repeat
       PE(M;A);
       VE(M,R);
        czytanie;
       PE(R);


end
end
```

```
procedure pisanie
begin
repeat
 VE(A);
 PE(M;R);
 pisanie;
 VE(M);
 PE(A);
end
end
```

A – liczba procesów piszących, które chcą pisać
R - liczba procesów czytających
M – zapewnia wykluczanie

procesy

Odrębne:

- Unikatowy PID (2-32000)
- Zmienne
- Zbiory deskryptorów plików
- Przestrzeń stosu (lokalne zmienne, wywołania funkcji)
- Środowisko
- Licznik rozkazów

Dzielone:

- Kod programu – brak możliwości zapisu
- Biblioteki systemowe

Uruchamianie procesów

#include <stdlib.h>

system(const char *string)

zwraca: 127 – nie można uruchomić powłoki

-1 – inny błąd

kod wy polecenia

równoważne - **sh** –c string

oczekuje na zakończenie procesu chyba że:

system(„polecenie &”);

Zastepowanie procesu
(rodzina funkcji exec)

```
#include <unistd.h>

char **environ;

int execl(const char *path, const char *arg0,  (char *) 0);
int execlp(const char *path, const char *arg0,  (char *) 0);
int execle(const char *path, const char *arg0,  (char *) 0,const char *envp[]);


int execv(const char *path, const char *argv[]);
int execvp(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[], const char *envp[]);
```

```
.
.
#include….
int main()
{
printf(„start\n”);
execlp(„ps”, „ps”,”aux”,0);
printf(„koniec\n”);
exit(0);
}
```

- PID ps – a PID proc. mac.
- brak powrotu do pr. mac.
- proces potomny dziedziczy deskryptory plików

Duplikowanie procesu

#include <sys/types.h>

#include <unistd.h>

pid_t **fork**(void);

Zwraca:

- pid procesu potomnego - w procesie macierzystym
- 0 - w procesie potomnym
- -1 - błąd

Proces potomny dziedziczy: zmienne, deskryptory plików

```
#include <sys/types.h>
#include <unistd.h>
#include<stdio.h>
…..
switch (fork())
{          case –1;
   printf(„fork error”);
   exit(1);
case 0;
    /* akcja dla procesu potomnego */
          break;
default;
   /* akcja dla procesu macierzystego */
          break;
}
```

```
#include<sys/types.h>
#include<unistd.h>
….
switch (fork())
{          case –1;
    printf(„fork error”);
    exit(1);
case 0; /* proces potomny */
     execl(„./program”,„program”,NULL);
     exit(2);
default;  /* proces macierzysty */
}
```

getpid()

zwraca pid procesu wywołującego

zombie

- proces potomny po zakończeniu – pozostaje w systemie (aż pr. mac. wykona wait lub się zakończy)

sierota

- proces mac. zakończył pracę nieprawidłowo (PPID=1)

oczekiwanie na zakończenie potomka

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat);
pid_t waitpid(pid_t pid, int *stat, int opt);
```

wait - zwraca pid procesu, który się zakończył
pod adresem wskazywanym przez status umieszczany jest status zakończenia

waitpid – opt =np. WNOHANG – zapobiega wstrzymywaniu procesu wywołującego (zwraca 0 jeśli procesy potomne działają) - do testowania, czy proces potomny się zakończył

Zarządzanie procesami:

```
#include<stdio.h>
int main()
{int i,id,status;
fprintf(stdout,”Mój PID %d\n”,getpid());
fprintf(stdout,”Teraz tworze potomka\n” );
id=fork();
if (id==0)
fprintf(stdout,”id=%d  to pisze proces potomny PID= %d\n”,id,getpid());
else
{
wait(&status);
fprintf(stdout,”id =%d to pisze proces macierzysty id= %d\n”,id,getpid());
};
fprintf(stdout,”A to pisza obydwa procesy PID=%d\n”,getpid());}
```

Komunikacja między procesami:

- sygnały
- pliki
- łącza komunikacyjne
- łącza nazwane (kolejki FIFO)
- semafory
- komunikaty
- pamięć dzielona

Sygnały - jądro systemu oraz procesy mogą wysyłać sygnały do dowolnego procesu. Zestaw wszystkich sygnałów daje polecenie

kill -l

Sygnały są ponumerowane od 1 do 30

kill -9 PID - **SIGKILL**

SIKILL, SIGSTOP - nie może być przechwycony przez proces i potraktowany inaczej

SIGHUP - po zamknięciu sesji wszystkie procesy potomne procesu login dostają sygnał o numerze **1**, powodujący ich przerwanie pod warunkiem, że nie przejmują tego sygnału i nie podejmują innego działania

Można uruchomić program odporny na ten sygnał np. przez nohup dowolny program

Sygnały

fcja **signal** – manipulowanie sygnałami

```
#include<signal.h>

SIG_IGN – ignorowanie sygnału
SIG_DFL – przywraca domyślne działanie
(również po przechwyceniu danego sygnału)


….
void au(int sig)
{printf(„przechwytany sygnal %d\n”,sig);}
int main()
{……
(void) signal(SIGINT,au);
……..
}
```

bywa różnie w różnych wersjach systemu!!

Po przechwyceniu sygnału przywraca domyślną akcję lub nie

```
#include<signal.h>
int raise(int sig);
- do siebie

#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid, int sig);
- do procesu o identycznym uid

#include<unistd.h>
unsigned int alarm(unsigned int sec);
int pause(void)
alarm:
•wysyła sygnał SIGALARM za sec sekund
•każdy proces może mieć max. 1 zaplanowany alarm
pause – wstrzymuje działanie programu do otrzymania
   sygnału
```

sigaction – interfejs obsługi sygnałów – specyfikacja X/Open

```
#include<signal.h>
int sigaction(int sig, const struct sigaction *act, const struct sigaction
   *oact);
struct sigaction {
               void     (*sa_handler)(int);
               void     (*sa_sigaction)(int, siginfo_t *, void *);
               sigset_t   sa_mask;
               int        sa_flags;};
-----------------------------------------------------
struct sigaction sa;
sa.sa_sigaction = sighandler;
//sa.sa_flags = SA_SIGINFO;  -- sigaction zamiast handler -- niekoniecznie
sigemptyset(&sa.sa_mask);    //pusta maska - bez blokowania sygnalow
sigaction(SIGINT, &sa, NULL);
…..
void sighandler(int sig)
{printf(„przechwytany sygnal %d\n”,sig);}
}
-------------------------------------------------------------------
oact – tu ustawiana poprzednia reakcja na sygnał
sigemptyset, sigaddset,…+ sigprocmask -- modyfikacje zbioru sygnałów
```

Pliki – najczęstsza metoda komunikowania się procesów

(jeden proces tworzy plik za pomocą dowolnego edytora, drugi przetwarza ten tekst – porządkuje alfabetycznie)

problem: proces czytający może wyprzedzić proces piszący i uznać, że komunikacja została zakończona

-> łącza komunikacyjne

Łącza komunikacyjne

- nie są plikami
- chociaż mają swój i-węzeł
- nie ma dowiązania w systemie plików

•jeśli proces czytający zbyt wyprzedzi proces piszący -> oczekuje na dalsze dane;

•jeśli proces piszący zbyt wyprzedzi proces czytający -> zostaje uśpiony

•łącza komunikacyjne wykorzystywane z poziomu powłoki - potoki

•dotyczą procesów pokrewnych

•powolne

Kolejki FIFO

- łącza nazwane - kolejki FIFO (first-in-first-out)
- plik specjalny (typ pliku - p)
- może być otwarty przez każdy proces
- umożliwia współpracę wielu procesów piszących i czytających

  (gwarantują niepodzielność)
- powolne

Semafory

- uniemożliwiają dostępu do zasobów dwóm lub większej liczbie procesów
- flaga możliwa do ustawiana i opuszczania przez różne procesy

Komunikaty

- Procesy mogą przesłać do kolejki komunikatów niewielką ilość danych
- Procesy, które mają uprawnienia mogą pobierać z niej kolejki komunikaty

Pamięć dzielona

- najszybszy sposób komunikacji między procesami
- ten sam obszar pamięci jest przydzielany kilku procesom
- dane wygenerowane przez jeden proces są natychmiast dostępne dla innych procesów
- dostęp do pamięci dzielonej wymaga synchronizacji – semafory