# Zarządzanie procesami; **problemy współbieżnych procesów**

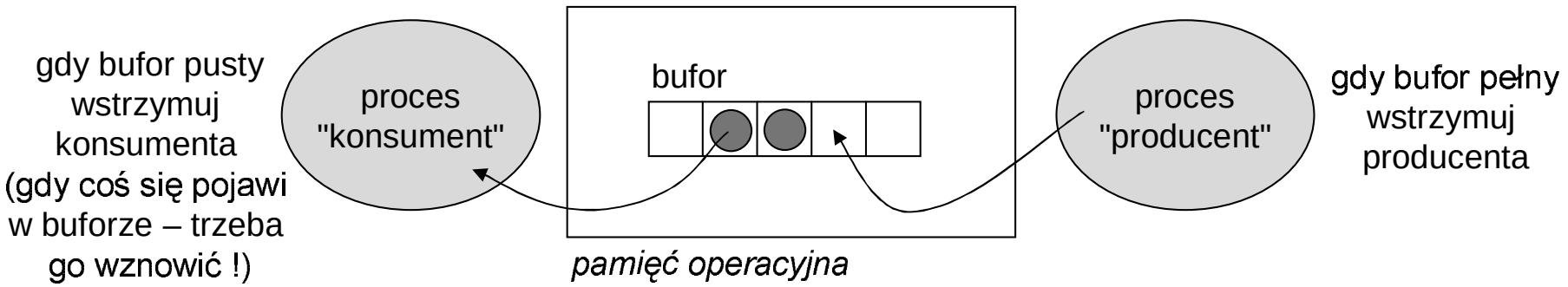
- współbieżne procesy = procesy które działają równocześnie
- istnieją następujące rodzaje interakcji między procesami:
  - procesy nieświadome swojego istnienia

    pojawiają się problemy przy dostępie do *zasobów krytycznych,* czyli takich do których dostęp w danej chwili może mieć co najwyżej jeden proces (przykład: drukarka, ale bez spool-ingu)
  - procesy [lub wątki] współpracujące:
    - **przez (współ)dzielenie**

      dzielenie pamięci operacyjnej lub pliku
    - **przez komunikację**

      jawne wysyłanie komunikatów między sobą

Przykład współpracy procesów "przez dzielenie" (*problem producenta & konsumenta*) :



# Zarządzanie procesami; **problemy współbieżnych procesów**

- ... zajmiemy się teraz procesami/ wątkami "współpracującymi przez dzielenie" lub "nieświadomymi swego istnienia"
- takie procesy często mają dostęp do wspólnych danych w pamięci operacyjnej
- współbieżny dostęp [modyfikacje] do wspólnych danych może powodować ich niespójność (np. współbieżne operacje na liście elementów znajdującej się w pamięci operacyjnej ... taka lista jest *zasobem krytycznym*)
- (?) jak się unika powyższego problemu ?
  Odp: wstrzymuje się procesy gdy to jest potrzebne
- ***synchronizacja procesów*** to właśnie wstrzymywanie działania procesów w odpowiednich momentach (i wznawianie)

Przykład: problem ze współbieżnym dostępem do zmiennej "licznik";
zmienna licznik jest dostępna dla procesów P0 i P1
na początku licznik:=0

**Proces P0:**

```
for i:=1 to 10 do
begin
   licznik:=licznik+1   { l1: ax:= licznik
                          l2: ax:= ax + 1
                          l3: licznik:= ax
end
```

**Proces P1:**

```
for i:=1 to 10 do
begin
   licznik:=licznik+1   { l1: ax:= licznik
                          l2: ax:= ax + 1
                          l3: licznik:= ax
end
```

*... jeśli wszystko jest ok. to po zakończeniu P0 i P1 licznik=20*

# Zarządzanie procesami; **problemy współbieżnych procesów**

- problem dostępu do *zasobu krytycznego* = problem sekcji krytycznej
  - **zasób krytyczny** to taki który w danej chwili może być używany (posiadany) przez co najwyżej jeden proces (np.: bufor w prod&kons, zmienna "licznik", lista elementów)
- **def** problemu sekcji krytycznej
  - mamy n procesów: P0, P1, ... , P{n-1}
  - procesy te mają fragmenty kodu w których używają *zasobu krytycznego*; fragmenty te nazywamy **sekcjami krytycznymi** (np. w problemie prod&kons sekcją krytyczna jest wykonywanie operacji na buforze)
  - rozwiązanie problemu sekcji krytycznej to mechanizm który zapewnia:
    - **wzajemne wykluczanie procesów**
      w danej chwili co najwyżej 1 proces przebywa w sekcji krytycznej
    - **postęp**
      jeśli w danej chwili żaden proces nie przebywa w sek. kryt.
      oraz są procesy próbujące wejść do sek. kryt.,
      to jeden z nich zostanie wpuszczony
    - **ograniczone czekanie**
      każdemu procesowi uda się wejść do sek. kryt.
      po ograniczonej liczbie prób wejścia

# Zarządzanie procesami; **problemy współbieżnych procesów**

- problem sek. kryt. rozwiązujemy dodając odpowiednie "wstawki" w kodzie programu (*sekcja wejściowa* i *wyjściowa*)

**Proces P0:**
repeat
  pozostały kod
  sekcja wejściowa
  sekcja krytyczna
  sekcja wyjściowa
  pozostały kod
until false

jej zadaniem jest wstrzymywanie procesu (gdy nie może wejść do sek. kryt.)

- konkretne rozwiązanie problem sek. kryt.: **algorytm Petersena**
  - algorytm ten rozwiązuje problem sek. kryt. tylko dla 2 procesów P0 i P1 !!!
  - ***założenie***: w danej chwili każda komórka pamięci operacyjnej ma jedną wartość (co ma miejsce np. gdy mamy 1 procesor !)
  - ***zmienne pomocnicze* (globalne/ dzielone)** : [zainicjowane zerami !]
    flag: array[0..1] of integer,
    turn: integer;

# Zarządzanie procesami;

## algorytm Petersena = rozwiązanie prob. sek. kryt.

**Proces P0:**

```
repeat
   pozostały kod
   flag[0]:=1;
   turn:=0;
   while flag[1]=1 and turn=0 do nic;
   sekcja krytyczna
   flag[0]:=0;
   pozostały kod
until false
```

**Proces P1:**

```
repeat
   pozostały kod
   flag[1]:=1;
   turn:=1;
   while flag[0]=1 and turn=1 do nic;
   sekcja krytyczna
   flag[1]:=0;
   pozostały kod
until false
```

aktywne czekanie *(to wada !)*

**Jak to działa ?**

- *wzajemne wykluczanie* -rozważyć dwa przypadki:
  - gdy P0 przebywa w sekcji krytycznej a P1 próbuje wejść
  - gdy P0 i P1 równocześnie próbują wejść do sekcji krytycznej
- *postęp* – niemożliwe żeby oba procesy były równocześnie wstrzymywane gdyż warunki (flag[1]=1 and turn=0) i (flag[0]=1 and turn=1) nie mogą być równocześnie spełnione
- *ograniczone czekanie* – pokazać że P0 i P1 wchodzą do sek. kryt. na przemian (niezależnie od tego jak procesor się przełącza między P0 i P1 !)

# Zarządzanie procesami;
## algorytm Petersena c.d.

- **drobna zmiana w algorytmie Petersena powoduje błąd !!! (zamieniłem miejscami 2 pierwsze linijki sek. wejściowej)**

**Proces P0:**

```
repeat
  pozostały kod
  turn:=0;
  flag[0]:=1;
  while flag[1]=1 and turn=0 do nic;
  sekcja krytyczna
  flag[0]:=0;
  pozostały kod
until false
```

**Proces P1:**

```
repeat
  pozostały kod
  turn:=1;
  flag[1]:=1;
  while flag[0]=1 and turn=1 do nic;
  sekcja krytyczna
  flag[1]:=0;
  pozostały kod
until false
```

**Jak dochodzi do błędu (przebywania P0 i P1 równocześnie w sek. kryt.) ?**

1. procesor wykonuje P1 i przełącza się w oznaczonym miejscu na P0
2. P0 wchodzi do sekcji krytycznej (turn=0)
3. procesor wraca do P1 i też wchodzi do sekcji krytycznej !!!

# Zarządzanie procesami; **problemy współbieżnych procesów**

- są inne sposoby rozwiązywania(?) problemu sekcji krytycznej ...
- **wyłączanie przerwań** w obszarze sek. kryt
  - problemy pojawiają się gdy procesor przełącza się w środku sekcji krytycznej
  - możliwe tylko gdy sek. kryt. jest bardzo krótka ... i nie wymaga działającej obsługi przerwań
  - intel 8086: *cli* -przerwania zakazane, *sti* –przerwania dozwolone
  - wada: nie można zróżnicować sek. kryt. zasobu A i sek.kryt. zasobu B
- **specjalne rozkazy procesora**, np. TestAndSet():

```
def rozkazu TestAndSet:
function TestAndSet(var cel:boolean)
   :boolean;
begin
   TestAndSet:=cel;
   cel:=true;
end
```

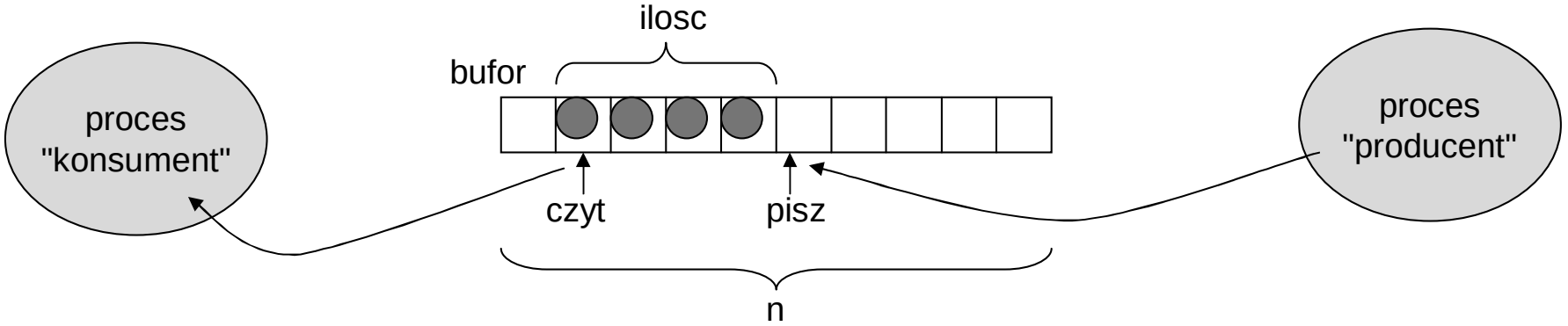
*na początku*:
zamek:=false;

```
Proces Pi:
repeat
   pozostały kod
   while TestAndSet(zamek) do nic;
   sekcja krytyczna
   zamek:=false;
   pozostały kod
until false
```

*Uwaga 1*: to działa dla dowolnej liczby procesów !
*Uwaga 2*: nie ma gwarancji *ograniczonego czekania* !

# Zarządzanie procesami;
## problem producenta/ konsumenta

- **rozwiązanie z *aktywnym czekaniem***




**Proces Konsument:**

```
repeat
  while ilosc=0 do nic;
  elem:=bufor[czyt];
  czyt:=(czyt+1) mod n;
  ilosc:=ilosc-1;
  konsumuje element "elem"
until false
```

**Proces Producent:**

```
repeat
  produkuje element "elem"
  while ilosc=n do nic;
  bufor[pisz]:=elem;
  pisz:=(pisz+1) mod n;
  ilosc:=ilosc+1;
until false
```

tu trzeba zapewnić
wzajemne wykluczanie !!!
(to jest sekcja krytyczna)

# Zarządzanie procesami;
## algorytm piekarni

- **rozwiązanie problemu sekcji krytycznej dla wielu procesów (przy pomocy zmiennych globalnych w pamięci operacyjnej)**
- **mamy n- procesów** P0, P1, ..., P{n-1} próbujących wchodzić do sekcji krytycznej ...
- **zmienne globalne:**
  wybrane: array[0..n-1] of boolean; {wartość początkowa: fałsz}
  numer: array[0..n-1] of integer; {wartość początkowa: 0}
- **specjalna relacja "<":** (a,b) < (c,d) ⇔a<c lub (a=c i b<d)

**sekcja wejściowa procesu Pi:**

```
wybrane[i]:=prawda;
numer[i]:=max(numer[0], numer[1], ..., numer[n-1])+1;
wybrane[i]:=fałsz;
for k:=0 to n-1 do
begin
   while wybrane[k] do nic;
   while numer[k]<>0 and (numer[k],k)<(numer[i],i) do nic;
end;
```

**sekcja wyjściowa procesu Pi:**

```
numer[i]:=0;
```

# Zarządzanie procesami;
## algorytm piekarni c.d.

**sekcja wejściowa procesu Pi:**

```
wybrane[i]:=prawda;
numer[i]:=max(numer[0], numer[1], ..., numer[n-1])+1;
wybrane[i]:=fałsz;
for k:=0 to n-1 do
begin
   while wybrane[k] do nic;
   while numer[k]<>0 and (numer[k],k)<(numer[i],i) do nic;
end;
```

**sekcja wyjściowa procesu Pi:**

```
numer[i]:=0;
```

pętla oczekująca na wyjście z sekcji krytycznej procesów "mniejszych" od Pi

- zasada działania algorytmu piekarza:
  - wpuszcza się do sekcji krytycznej "najmniejszy" proces (najmniejszy w sensie specjalnej relacji "<")
  - kolejne procesy próbujące wejść do sekcji krytycznej otrzymują kolejne numerki
  - jeśli kilka procesów równocześnie próbuje wejść do sekcji krytycznej to mogą otrzymać *ten sam numerek !*; wtedy w relacji "<" bierze się pod uwagę identyfikatory "i" procesów
  - dlaczego nie można opierać się wyłącznie na identyfikatorach "i" procesów ? (odp: ... "zagłodzenie" procesów nieminimalnych)

# Zarządzanie procesami; **semafor i monitor**

- semafory i monitory to narzędzia "wyższego poziomu" służące do:
  - rozwiązywania problemu sek. kryt. ... i innych występujących we współbieżnie działających procesach
  - do synchronizowania procesów
    (patrz: *wstrzymywanie* w problemie prod/kons lub w problemie sek. kryt.)
  - nie ma w nich *aktywnego czekania* !!!
  - są dostępne przez odpowiednie fun.sys. lub przez fun.sys. + konstrukcje języka programowania (klasy C++)

# Zarządzanie procesami; **semafor**

- **def** semafora ogólnego
  - semafor ogólny to zmienna całkowita "s" na której można wykonywać następujące operacje:
    - **wait(s)** ⇔
      jeśli s>0 to s:=s-1;
      jeśli s=0 to wstrzymaj działanie procesu *[wywołującego wait(s)]*
    - **signal(s)** ⇔
      jeśli są procesy wstrzymane podczas wait(s) to wznów jeden z nich;
      jeśli nie ma to s:=s+1
    - **s:=i** ⇔inicjowanie semafora wartością "i"
- semafor FIFO/ semafor losowy:
  - wstrzymane procesy są umieszczane w "kolejce procesów wstrzymanych na sem."
  - FIFO: wznawiamy pierwszy w kolejce proces, a wstrzymywane umieszczamy na jej końcu (na zasadzie FIFO)
  - losowy: wznawiamy proces losowo wybrany z kolejki

# Zarządzanie procesami; **semafor**

- rozwiązanie problemu sekcji krytycznej przy pomocy semafora ogólnego
  - nie ma aktywnego czekania !
  - jeśli to jest semafor FIFO to oprócz wzaj.wykl. i postępu mamy(?) gwarancje ograniczonego czekania !
- (patrz: następny slajd ...)

*na początku*: s:=1;

**Proces P0:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

**Proces P1:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

**Proces P2:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

# Zarządzanie procesami; **semafor**

**wait(s)** ⇔jeśli s>0 to s:=s-1;
jeśli s=0 to wstrzymaj działanie procesu *[wywołującego wait(s)]*
**signal(s)** ⇔jeśli są procesy wstrzymane podczas wait(s) to wznów jeden z nich;
jeśli nie ma to s:=s+1
**s:=i** ⇔inicjowanie semafora wartością "i"

*na początku*: s:=1;

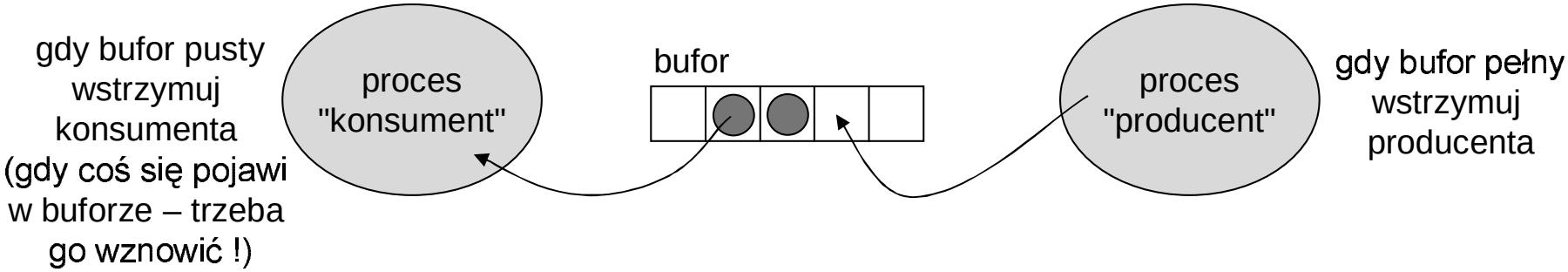
**Proces P0:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

**Proces P1:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

**Proces P2:**
repeat
pozostały kod
wait(s);
sekcja krytyczna
signal(s);
pozostały kod
until false

# Zarządzanie procesami; **semafor**

- rozwiązanie problemu producenta/ konsumenta przy pomocy semafora ogólnego ...




*mamy 2 semafory:* **w** i **p** ("w" jak wolne elementy, "p" jak pełne)
*na początku*: w:=rozmiar_bufora; p:=0;

**Proces Konsument:**

```
repeat
  pozostały kod
  wait(p);
  odczyt elementu z bufora
  signal(w);
  pozostały kod
until false
```

**Proces Producent:**

```
repeat
  pozostały kod
  wait(w);
  zapis elementu do bufora
  signal(p);
  pozostały kod
until false
```

# Zarządzanie procesami; **inne problemy współbieżności**

- **problem czytelników i pisarzy**
  - ... istnieje czytelnia oraz dwie kategorie osób: czytelnicy i pisarze; w danej chwili w czytelni może przebywać dowolna liczba czytelników lub dokładnie jeden pisarz
  - (jest to abstrakcja problemu dostępu do bazy danych)
  - jak to rozwiązać przy pomocy semaforów ogólnych ?

mamy dwa semafory ogólne **R** i **W**;
zainicjowane następująco: **R:="rozmiar czytelni"**, **W:=1**
przed i za sekcją kodu w której dana "osoba" przebywa w czytelni
dodajemy sekcje wejściową i wyjściową (*wchodzi, *wychodzi)

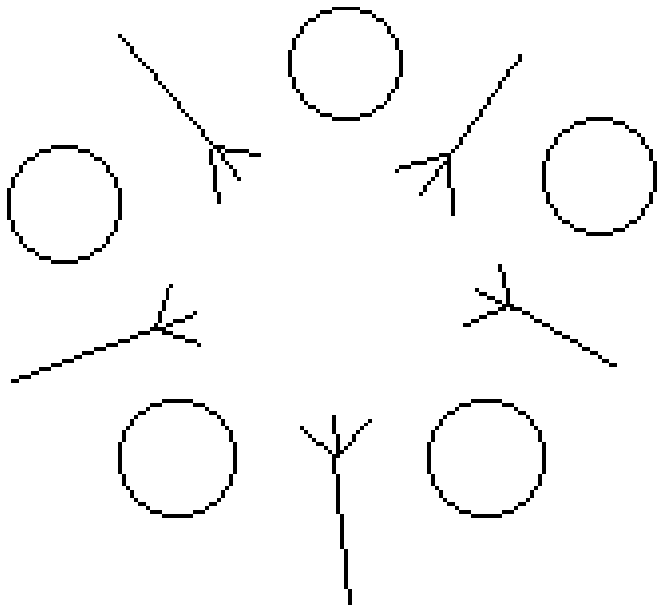
operacja zajmowania miejsc w czytelni

| **czytelnik wchodzi** | **pisarz wchodzi** |
|---|---|
| wait(R); | wait(W); |
| | for i:=1 to "rozmiar czytelni" do wait(R); |
| **czytelnik wychodzi** | **pisarz wychodzi** |
| signal(R); | for i:=1 to "rozmiar czytelni" do signal(R); |
| | signal(W); |

... co by się stało gdyby nie było wait(W)/signal(W) ? [**zakleszczenie**]

# Zarządzanie procesami; **inne problemy współbieżności**

- **problem 5 filozofów**
  - Pięciu głodnych filozofów siedzi przy okrągłym stole, przy talerzach z rybą. Aby jeść rybę potrzebne są 2 widelce, jednak między dwoma sąsiednimi filozofami leży tylko 1 widelec.
  - Wszyscy filozofowie na przemian myślą i jedzą (w nieskończonej pętli).
  - Zauważmy że wprowadzenie wzajemnego wykluczania dla każdego widelca nie rozwiązuje problemu: może się zdarzyć że wszyscy filozofowie zaczną podnosić widelce począwszy od lewego - wtedy nie bedą mogli podnieść prawego widelca - wystąpi tzw zakleszczenie ...
  - *jak to rozwiązać ?*
    Odp: ograniczyć liczbę filozofów równocześnie próbujących jeść (przy pomocy semafora ogólnego)



# Zarządzanie procesami; **semafor binarny**

- **def** semafora binarnego
  - semafor binarny to zmienna "s" przyjmująca wartości 0 i 1, na której można wykonywać następujące operacje:
    - **wait(s)** ⇔
      jeśli s=1 to s:=0;
      jeśli s=0 to wstrzymaj działanie procesu *[wywołującego wait(s)]*
    - **signal(s)** ⇔
      jeśli są procesy wstrzymane podczas wait(s) to wznów jeden z nich;
      jeśli nie ma to s:=1
    - **s:=i** ⇔inicjowanie semafora wartością 0 lub 1
- czym semafor binarny różni się od ogólnego?
  - sem. binarny "nie pamięta" liczby wykonanych operacji signal()
  - sem. binarny często wymaga używania zmiennych pomocniczych
- *problemy z semaforami*:
  - rozwiązanie problemu producenta/ konsumenta przy pomocy semafora binarnego (wersja problemu z "nieskończonym" buforem)
  - pokażemy że programowanie z użyciem semaforów jest trudne ...
  - (patrz: następny slajd ...)

# Zarządzanie procesami; **semafor binarny c.d.**

- rozwiązanie problemu producenta/ konsumenta przy pomocy semafora binarnego (wersja problemu z "nieskończonym" buforem – tylko konsument musi czasami czekać)

*mamy semafory binarne:*

**Zwłoka** -zainicjowany 0; służy do wstrzymywania konsumenta gdy bufor pusty

***S*** –zainicjowany 1; ochrona sekcji krytycznej przy dostępie do bufora

*zmienne pomocnicze:*

**N** –zainicjowana 0; jest to liczba nieskonsumowanych elementów w buforze

**Proces Producent:**

```
repeat
  wait(S);
  { produkuje element
  i zapisuje do bufora }
  N:=N+1;
  if N=1 then signal(Zwłoka);
  signal(S);
  { pozostały kod }
until false
```

**Proces Konsument:**

```
wait(Zwłoka);
repeat
  wait(S);
  { odczyt elementu z bufora
  i jego konsumpcja }
  N:=N-1;
  signal(S);
  if N=0 then wait(Zwłoka);
  { pozostały kod }
until false
```

# Zarządzanie procesami;

## semafor binarny c.d.

- scenariusz prowadzący do błędu w rozwiązaniu problemu producenta i konsumenta przy pomocy sem. bin.
  1. konsument właśnie skonsumował ostatni element bufora, N=0; procesor przełącza się na producenta w zaznaczonym punkcie ...
  2. producent produkuje nowy element przy czym wykonuje signal(Zwłoka), N=1, Zwłoka=1; procesor przełącza się na konsumenta ...
  3. konsument NIE wykona wait(Zwłoka) bo N<>0
  4. konsument konsumuje (ostatni) element z bufora, wykonuje wait(Zwłoka) – ale NIE zostanie wstrzymany bo Zwłoka=1 !!!
  5. konsument konsumuje nieistniejący element z bufora ...

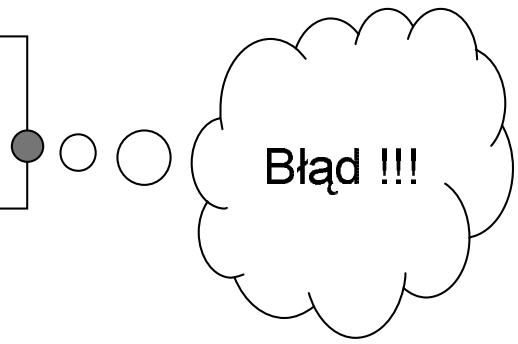
**Proces Producent:**

```
repeat
  wait(S);
   { produkuje element
   i zapisuje do bufora }
  N:=N+1;
  if N=1 then signal(Zwłoka);
  signal(S);
   { pozostały kod }
until false
```

**Proces Konsument:**

```
wait(Zwłoka);
repeat
  wait(S);
   { odczyt elementu z bufora
   i jego konsumpcja }
  N:=N-1;
  signal(S);
  if N=0 then wait(Zwłoka);
   { pozostały kod }
until false
```

*programowanie z użyciem semaforów jest trudne ...*



# Zarządzanie procesami; **monitory**

- jest to "nowocześniejsze" narzędzie do rozwiązywania problemów współbieżności, pozbawione niektórych wad semaforów ...
- **def** monitora:
  - Monitor jest zbiorem procedur i zmiennych, spełniającym poniższe warunki:

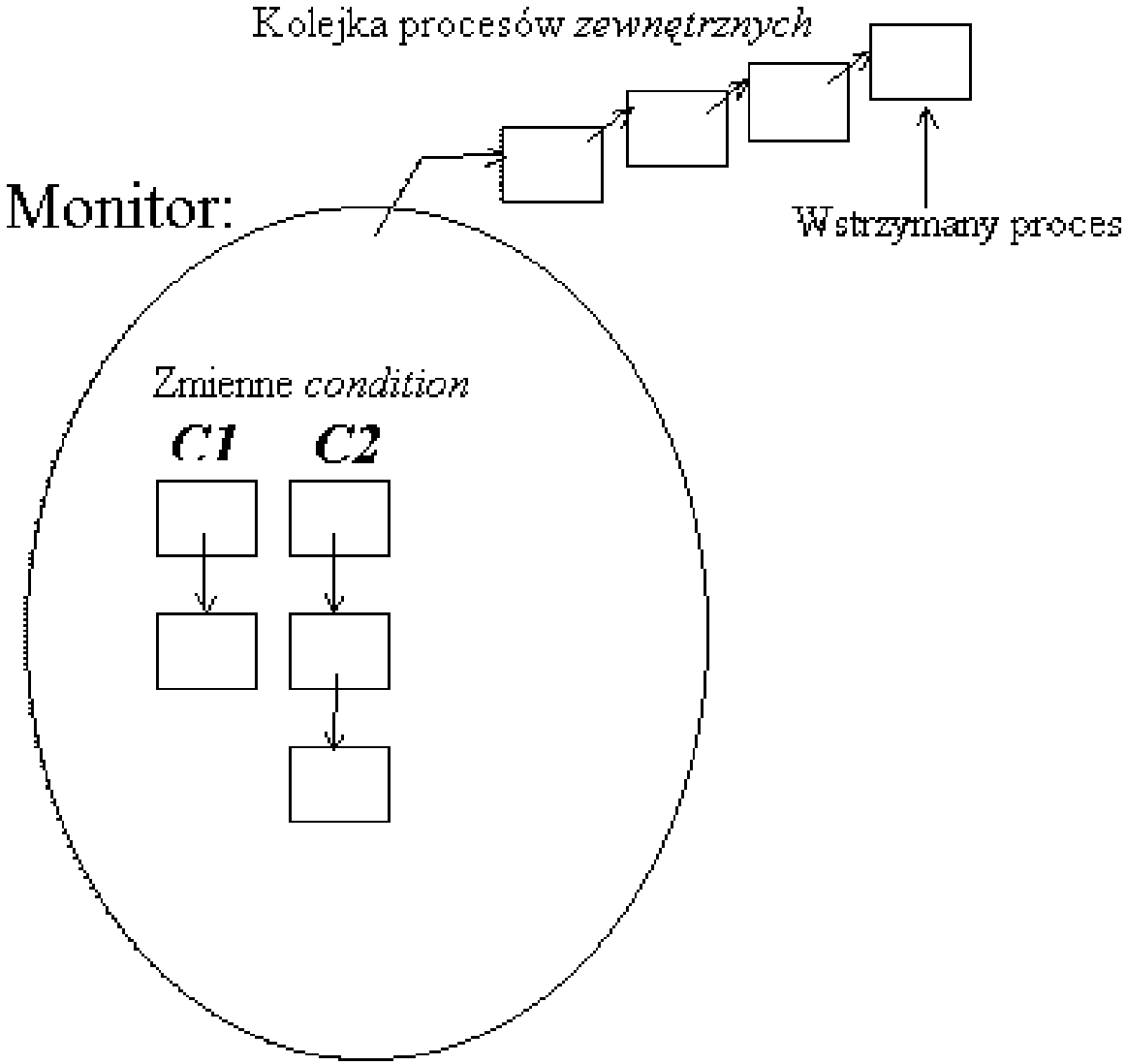
```
monitor semaforX; { przykład monitora j. BACI }
    var x: integer; c:condition;
    procedure waitX();
    begin
       if x=0 then waitc(c) else x:=x-1;
    end;
    procedure signalX();
    begin
       if empty(c) then x:=x+1 else signalc(c);
    end;
end;
```

  - Zmienne są widoczne tylko wewnątrz monitora (wewnątrz bloku monitora), natomiast procedury są widoczne na zewnątrz. Tak więc dostęp do zmiennych monitora jest możliwy tylko poprzez procedury monitora.
  - W danej chwili tylko jeden proces może *przebywać* w monitorze (tj wykonywać jego procedurę) - *sekcje krytyczne* trzeba umieszczać w procedurach monitora

# Zarządzanie procesami; **monitory c.d.**

- **def** monitora c.d:
  - Istnieją dwa rodzaje kolejek zawierających wstrzymane procesy:
    - *kolejka procesów zewnętrznych* (próbujących uruchomić procedurę monitora)
    - kolejki związane ze zmiennymi typu *conditon*.
  - Gdy monitor jest *zajęty* (pewien proces wykonuje jedną z jego procedur) i nastąpi próba wywołania procedury monitora przez inny proces, to ten inny proces zostanie wstrzymany i umieszczony na końcu kolejki procesów zewnętrznych. Gdy monitor zostanie zwolniony, to wtedy wznawiany jest pierwszy proces z kolejki procesów zewnętrznych.
  - Istnieje możliwość wstrzymywania i wznawiania procesów wewnątrz monitora przy pomocy zmiennych typu *condition*, na których można wykonywać operacje "waitc()" oraz "signalc()".
  - Po wykonaniu **waitc(c)** gdzie "c" jest zmienna typu *conditon* proces jest usypiany i umieszczany w kolejce związanej ze zmienną "c". Monitor jest *zwalniany*, może go zająć inny proces.
  - Po wykonaniu **signalc(c)** budzony jest jeden z procesów wybrany losowo z kolejki zmiennej "c", a proces który wykonał "signalc()" jest usypiany i umieszczany na początku kolejki procesów zewnętrznych.
  - Istnieje możliwość sprawdzenia, czy kolejka zmiennej "c" jest pusta przy pomocy "empty(c)".

# Zarządzanie procesami;
## **monitory c.d.**



# Zarządzanie procesami; **monitory c.d.**

- rozwiązanie problemu producenta i konsumenta przy pomocy monitorów:

```
monitor ProdKons;
var {deklaracje bufora i jego indeksów ...}
  P,W:integer;
    {zmienne liczące ile jest pełnych/wolnych elementów w buforze}
  Kons,Prod:condition;
    {kolejki wstrzymanych konsumentów i producentów}
procedure WstawElement(el:integer); {PRODUCENT}
begin
  if W=0 then waitc(Prod);
  {wstaw "el" do bufora} W:=W-1; P:=P+1;
  signalc(Kons);
end;
procedure OdczytajElement(var el:integer); {KONSUMENT}
begin
  if P=0 then waitc(Kons);
  {odczytaj element "el" z bufora} W:=W+1; P:=P-1;
  signalc(Prod);
end; end;
```

# Zarządzanie procesami; **monitory c.d.**

- przykłady "prawdziwych" monitorów (problem prod & kons):
  - Linux, język C++, biblioteka wątkowa "pthread", *monitor01.cc*
  - język Java, *monitor01.java*