6. SYNCHRONIZACJA PROCESÓW

Procesy mogą: -bezpośrednio dzielić logiczną przestrzeń adresową (kody funkcji, dane),
-dzielić dane tylko za pośrednictwem plików.

➔ **Współbieżny dostęp** do **danych dzielonych może powodować** ich **niespójność.**

Szkodliwa rywalizacja *(race condition):* kilka procesów **współbieżnie** wykonuje działania na **tych samych** danych

☛ Wynik tych działań zależy **od porządku**, w jakim procesy miały do nich dostęp.

Wspólne zmienne: liczba całkowita **i** , tablica **V**.

Zmienna **a** przyjmie **różne** wartości, zależnie od tego czy procesy działały oddzielnie, czy w warunkach rywalizacji.

```
P1: .....
  i = i + 1;
  a = V[i];
```

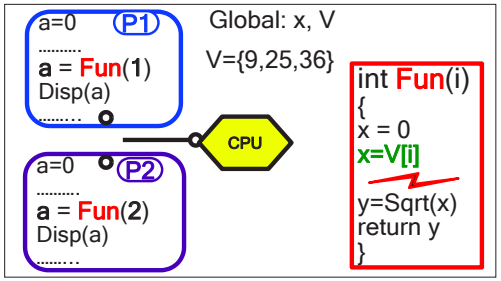
```
P2: .....
  i = i + 1;
  a = V[i];
```

```
i = i+1;
i = i+1;
```

Problem zniknie jeżeli dostęp do pewnego **ciągu instrukcji** -w **każdym procesie** - będzie odpowiednio zsynchronizowany.

➔ **Współbieżny dostęp** do **kodu funkcji może powodować niespójność wyników.**

Funkcja **Fun()** jest funkcją współdzieloną, umieszczona we wspólnym dla wszystkich procesów fragmencie pamięci.



Proces **P1** wywołuje funkcję **Fun()** i zostaje przerwany po wykonaniu instrukcji **x = V[i]**.
Zmienna **x** przyjmuje wartość **x = 9**.

Proces **P2** staje się aktywny i wywołuje funkcję **Fun()**, którą kończy bez przeszkód.
Zmienna **x** przyjmie wartość **x = 25**.
Proces **P2** wyświetli wartość **a = 5**.

Proces **P1** wznawia działanie.
Zmienna **x** ma wartość x=25 gdyż została **nadpisana** przez proces **P2**.
Proces **P1** wyświetli wartość **a = 5**; oczekiwano **a = 3**

Jeśli procesy zawierają fragmenty kodów, odwołujące się do wspólnych zmiennych, to dostęp do tych kodów należy koordynować.
Koordynację procesów należy wymuszać.

Potrzebne są **mechanizmy** gwarantujące spójności danych, poprzez sterowanie wykonywaniem procesów, użytkujących wspólną logiczną przestrzeń adresową.

6.1. Sekcja krytyczna

Sekcja krytyczna *(critical section)*: system złożony z **n** procesów {P1, P2, P3,...,Pn}, i każdy ma **segment kodu**, w którym Procesy mogą zmieniać **wspólne zmienne**.

Jeżeli jeden Proces wykonuje **swoją sekcję krytyczną**, wówczas inne Procesy **nie** są dopuszczone do wykonywania **swoich** sekcji krytycznych.

+Wykonanie **sekcji krytycznej** podlega **wzajemnemu wykluczaniu** *(mutual exclusion)* w czasie.

```
repeat                          // proces P_i
  sekcja WE(zasob)
  kod Sekcji Krytycznej Procesu_i
  sekcja WY(zasob)
  Reszta Kodu
until false
```

➔ Proces musi **prosić** o pozwolenie wejścia do **swojej** sekcji krytycznej.
Kod realizujący taką prośbę to **sekcja WEjściowa** *(entry section).*
Po sekcji krytycznej występuje **sekcja WYjściowa** *(exit section).*
☛ Argumentami obu funkcji jest **nazwa zasobu** podlegającego rywalizacji.
Kod nie angażujący zasobu wspólnego, nazywa się **Resztą** *(remainder section).*

Rozwiązanie problemu **sekcji krytycznej musi** spełniać **trzy** warunki:

1. **wzajemne wykluczanie**: jeśli Proces **P** działa w swojej sekcji krytycznej, to inne procesy nie działają w swoich sekcjach krytycznych.
2. **postęp:** tylko Procesy **nie wykonujące swoich Reszt** mogą starać się wejść do swoich sekcji krytycznych (jeśli żaden proces nie działa w sekcji krytycznej a istnieją procesy, które chcą wejść do sekcji krytycznych), i wyboru tego **nie** można **odwlekać** bez końca.
3. **ograniczone czekanie:** istnieje graniczna wartość ***liczby wejść*** innych Procesów do **ich** sekcji krytycznych **po tym**, jak dany Proces zgłosił chęć wejścia do swojej sekcji krytycznej i zanim uzyskał pozwolenie.

Zgłasza → **C z e k a** (pewien czas) → **Wchodzi**

Podstawowe rozkazy maszynowe (***Pobierz, Zapisz, Sprawdź***) wykonywane są niepodzielnie.
Jeżeli dwa takie rozkazy wykonywane są współbieżnie, to wynik jest równoważny wykonaniu sekwencyjnemu w nieznanym porządku.

Jeśli rozkazy ***Pobierz*** i ***Zapisz*** będą wykonane współbieżnie, to rozkaz ***Pobierz*** natrafi na **starą albo nową** wartość w pamięci, ale **nie na jakąś kombinację** ich obu.

Realizacja wzajemnego wykluczenia:

❶ **programowa**: **P**roces przejmuje całą odpowiedzialność.

Procesy mogą być elementami programów systemowych lub aplikacji, ale muszą współpracować ze sobą, celem wymuszenia wzajemnych wykluczeń.

Nie otrzymując wsparcia ani od *SO*, ani od języków programowania.

☛Programowe implementacje wzajemnych wykluczeń znacznie obciążają CPU i wykazują dużą podatnością na błędy implementacyjne.

❷ **specjalne rozkazy maszynowe**: mniej obciążają procesor, ale nie zawsze są rozwiązaniem optymalnym.

❸ **mechanizmy wspomagające**: ze strony systemu operacyjnego lub języka programowania.

6.2. Programowa realizacja synchronizacji

Programową implementować wzajemnego wykluczenia można realizować na maszynach:
-jednoprocesorowych,
-wieloprocesorowych ze **współdzieloną** pamięcią główną.

Zazwyczaj stosuje się wykluczenia wzajemne na poziomie dostępu do pamięci.

Nie zakłada się wsparcia ze strony sprzętu, systemu operacyjnego lub języka programowania.

Próby równoczesnego dostępu (odczyt i zapis) do tej samej lokalizacji pamięci kontroluje rodzaj programu arbitrażowego, lecz **kolejność** przyznawania dostępu **nie jest** wcześniej określana.

Założenia do **algorytmów** (**1** ÷ **3**): dwa procesy **P0** i **P1** wykonują się w tym ***samym czasie.***

Dostęp do **wspólnych** zmiennych może mieć w tym samym czasie jeden proces.

6.2.1. Algorytm 1 (wspólny znacznik)

Wprowadza się zmienną całkowitą ***numer*** , nadając jej statut zmiennej **GLOBALNEJ**.

Proces **P0** lub **P1** chcąc wykonać swoją **sekcję krytyczną** musi najpierw sprawdzić zawartość zmiennej ***numer***.

Jeśli jest ona równa **numerowi** Procesu, to może on rozpoczynać przetwarzanie, w przeciwnym razie musi czekać.

```
                numer = 0
repeat          // proces P0
 while numer ≠ 0 do NIC
Sekcja Krytyczna (200ms)
 numer ← 1;
 Reszta
until false
```

numer=0

co 20 ms

```
                numer = 1
repeat          // proces P1
 while numer ≠ 1 do NIC
Sekcja Krytyczna
 numer ← 0;
 Reszta
until false
```

Procesy **P0** i **P1** mają wspólny znacznik ***numer*** z wartościami początkowymi 0 lub 1.

Jeśli ***numer*** = **0**, to w **Sekcji Krytycznej** może pracować tylko proces **P0.**

Jeśli ***numer*** = **1**, to w **Sekcji Krytycznej** może pracować tylko proces **P1.**

Oczekujący proces ***systematycznie odczytuje*** wartość zmiennej ***numer*** do czasu, aż uzyska zgodę (zmiana wartości *numer*) na przetwarzanie swojej sekcji krytycznej.

Jest to procedura **aktywnego oczekiwania**, gdyż wstrzymany proces nie robi nic produktywnego (ciągle sprawdza wartości zmiennej sterującej jego stanem).

Gdy proces skończy przetwarzanie swojej sekcji krytycznej, **musi** nadać zmiennej ***numer*** wartość równą numerowi **konkurującego** z nim Procesu.

Wzajemne wykluczania: rozwiązanie gwarantuje spełnienie**.**

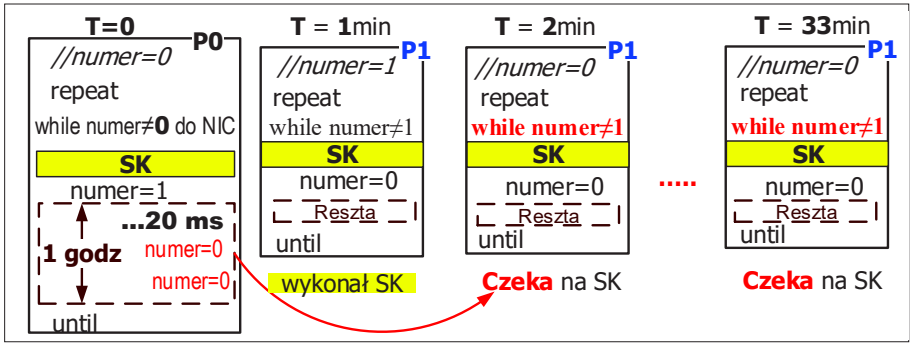
Warunek postępu: **nie** jest spełniony, gdyż wykonywanie przez Procesy sekcji krytycznych jest ***ściśle naprzemienne***.

● Tempo przetwarzania zależy od wolniejszego procesu.

P0 wykonuje swoją sekcję krytyczną **1-raz** godzinę (długa reszta).

P1 60 razy na godzinę (krótka reszta).

P1 musi dostosować dostęp do swojej sekcji krytycznej, do tempa działanie procesu **P0**.



Proces **P0** jest już poza swoją **SK** i wykonuje resztę kodu.

P1 wszedł **raz** do swojej **SK** i nie może wejść ponownie, aż **P0** znowu nie wejdzie do swojej SK i ustawi zmienną ***numer*** na **1**.

Wady Algorytmu 1:

☛ Jeżeli proces **P0** jest poza sekcją krytyczną i wykonuje **Resztę**, zaś ***numer == 0*** (przez pomyłkę) to proces **P1** **nie** może wejść do **swojej sekcji krytycznej.**

☛ Gdy jeden z procesów zawiesi się (w **sekcji krytycznej**, czy poza nią), drugi zostanie zablokowany.

Algorytm 1 nie zachowuje wystarczającej informacji o **stanie każdego** procesu, pamięta tylko, **któremu** procesowi wolno wejść do jego sekcji krytycznej.

6.2.2. Algorytm 2 (indywidualne znaczniki)

Potrzebne są informacje o stanie obu procesów.

Oba procesy powinny mieć własne klucze do uruchomienia sekcji krytycznych.

Zmienną ***numer*** zastępuje tablica ***Znacznik[0..1]*** z wartościami początkowymi ***FALSE***.

- **Każdy proces** ustawia **swój** znacznik. Może sprawdzić wartość znacznika 2-go Procesu, ale nie może go zmienić.
- **Przed uruchomieniem** swojej sekcji krytycznej procesPierwszy sprawdza znacznik procesuDrugiego, tak długo aż przyjmie on wartość ***false***, co oznacza, że **drugi** proces ***nie przetwarza*** swojej sekcji krytycznej.
- procesPierwszy przystępuje do przetwarzania swojej sekcji krytycznej, a po jej zakończeniu przywraca znacznikowi wartość ***false***.

```
Znacznik[0] = Znacznik[1] = false      // T0

repeat                                 // proces P0
❶ Znacznik[0] ← true
❷ while Znacznik[1] do NIC
    Sekcja Krytyczna
    Znacznik[0] ← false
    Reszta
until false
```

```
Znacznik[0] = Znacznik[1] = false      // T0
Znacznik[0] = true; Znacznik[1] = false // T1
repeat                                 // proces P1
❶ Znacznik[1] ← true
❷ while Znacznik[0] do NIC
    Sekcja Krytyczna
    Znacznik[1] ← false
    Reszta
until false
```

❶ **P0** ustawia **Znacznik[0] = true**, co sygnalizacje gotowości wejścia procesu do sekcji krytycznej.

❷ Następnie **P0** sprawdza, czy **P1** nie jest **jeszcze** w trakcie realizacji swojej sekcji krytycznej.

❸ Jeśli **P1** *przetwarza jeszcze*, to **P0** zaczeka, dopóki proces **P1** nie zasygnalizuje, że opuścił swoją sekcję krytyczną (ustawiając **Znacznik[1]** = ***false***).
Wówczas **P0** będzie mógł wejść do swojej sekcji krytycznej.
Opuszczając sekcję krytyczną **P0** nada ***swojemu*** znacznikowi wartość ***false***, pozwalając wejść drugiemu procesowi.

- **Rozwiązanie spełnia warunek wzajemnego wykluczania.**
  Gdy **P0** ustawi **Znacznik[0] = true**, **P1** nie może przejść do przetwarzania swojej sekcji krytycznej.
  Trwa to dopóki **P0** nie skończy swojej sekcji krytycznej i nie przywróci **Znacznik[0]** wartości **false**.
  Niech w trakcie przestawiania **Znacznik[0]** proces **P1** przetwarza już swoją sekcję krytyczną, wówczas **P0** zostanie zablokowany przez rozkaz **while** (Znacznik[**1**]). // Znacznik[**1**]==**1**
- **Nie jest spełniony warunek postępu.**
  Jeżeli **oba** Procesy nadadzą swoim znacznikom wartość ***true***, zanim którykolwiek rozpocznie przetwarzanie swojej sekcji krytycznej, tzn.: **T0**: **P0** ustala **Znacznik[0]** ← true
  T0+: **P1** ustala **Znacznik[1]** ← true
  to procesy **P0** i **P1 zapętlą się** w swoich instrukcjach **while.**
  Zapętlenie może się zdarzyć: -w środowisku wieloprocesorowym działających współbieżnie,
  -jeśli przerwanie (np. od czasomierza) wystąpi bezpośrednio po wykonaniu kroku **T0** i przełączy CPU z jednego procesu na drugi.

Algorytm 2 jest uzależniony od dokładnych następstw czasowych obu procesów.

❒ Problem podatności na błędy

Zamiana lokalizacji instrukcji umożliwi przebywanie obu procesów w sekcjach krytycznej w tym samym czasie.

```
Znacznik[0] = Znacznik[1] = false
repeat                                 // proces P0
  while Znacznik[1] do NIC
  Znacznik[0] ← true
  Sekcja Krytyczna
  Znacznik[0] ← false
  Reszta
until false
```

```
Znacznik[0] = Znacznik[1] = false
repeat                                 // proces P1
  while Znacznik[0] do NIC
  Znacznik[1] ← true
  Sekcja Krytyczna
  Znacznik[1] ← false
  Reszta
until false
```

1. **P0** wykonuje **while** i stwierdza, że *Znacznik[1] == false* → *idzie dalej*
2. **P1** wykonuje **while** i stwierdza, że *Znacznik[1] == false* → *idzie dalej*
3. **P0** ustawia *Znacznik[0]←true* oraz
   P1 ustawia *Znacznik[1]←true*
   i oba procesy jednocześnie przystępują do przetwarzania **sekcji krytycznej**.

❸ Jeśli jeden z procesów zawiesi się **poza** sekcją krytyczną (włącznie z fragmentami kodu nadającymi wartość znacznikom ∂),
drugi Proces **nie** zostanie zablokowany, gdyż może przetwarzać swoją sekcję krytyczną, ponieważ znacznik pierwszego ma wartość ***false***.

- Jeżeli proces: -zawiesi się **wewnątrz** sekcji krytycznej lub
  -po nadaniu znacznikowi wartości ***true***,
  wówczas **drugi** proces ***zostanie zablokowany***.

```
Znacznik[0] = Znacznik[1] = false

repeat                                 // proces P0
❶ Znacznik[0] ← true
❷ while Znacznik[1] do NIC
    Sekcja Krytyczna
    Znacznik[0] ← false
    Reszta
until false
```

```
Znacznik[0] = Znacznik[1] = false

Repeat                                 // proces P1
❶ Znacznik[1] ← true
❷ while Znacznik[0] do NIC
    Sekcja Krytyczna
    Znacznik[1] ← false
    Reszta
until false
```

6.2.3. Algorytm 3

Należy obserwować oba procesy i narzucić kolejność działania obu.

Procesy mają dwie wspólne zmienne: tablicę **Znacznik[0..1]**
zmienną **numer** = (**0**, **1**).

Na początku: **Znacznik**[**0..1**] = **false**;
wartość zmiennej ***numer*** jest nieistotna(0 lub 1).

Aby wejść do *swojej* sekcji krytycznej **P0** ustawia: **Znacznik[0] ← true** oraz
numer = **1**,
zakładając, że 2-gi Proces też chce wejść do sekcji krytycznej.

```
                                    // proces P0
repeat
  Znacznik[0] ← true
  numer ← 1
 while (Znacznik[1] and numer =1) do NIC
  Sekcja Krytyczna
  Znacznik[0] ← false
  Reszta
until false
```

```
                                    // proces P1
repeat
  Znacznik[1] ← true
  numer ← 0
 while (Znacznik[0] and numer =0) do NIC
  Sekcja Krytyczna
  Znacznik[1] ← false
  Reszta
until false
```

P0 chce rozpocząć przetwarzanie swojej sekcji krytycznej i ustawia **Znacznik[0]← true**.

Jeżeli **Znacznik[1]** procesu **P1** ma wartość ***false*** to **P0** może wejść do sekcji krytycznej.

W przeciwnym razie sprawdza zmienną ***numer***; wartość **0** oznacza, że **P0 ma prawo** rozpocząć wykonywanie sekcji krytycznej.

☛ Po zakończeniu swojej sekcji krytycznej **P0** ustawia **Znacznik[0]←false**; przekazując prawo wykonywania sekcji krytycznej procesowi **P1**.

Gdy **dwa** procesy chcą wejść w tym samym czasie, to zmienna ***numer*** otrzyma wartość **0** oraz **1** w tym samym czasie.

Tylko jedna wartość zostanie utrwalona i decyduje ona, który z dwu procesów może wejść do ***swojej*** sekcji krytycznej w **pierwszej** kolejności.

Algorytm 3 spełnia warunki:
wzajemnego wykluczania, postępu i ***ograniczonego czekania***.

Dowód poprawności zawarty jest w pozycji [1].

6.3. Sprzętowe wspomaganie synchronizacji

Niepodzielne rozkazy sprzętowe:

-**Sprawdzające** i **Zmieniające** zawartość słowa albo

-**Zamieniające** (Swap) wartości dwu słów,
można użyć do rozwiązania problemu sekcji krytycznej.

Definiujemy rozkazy: **TestujUstal** oraz **Swap**, których nie można przerwać.

Jeśli dwa takie rozkazy wykonywane są jednocześnie (każdy w innym CPU), to będą wykonane sekwencyjnie w dowolnym porządku.

➔ Wprowadzamy zmienną boolowską z wartością początkową ***key*** **=** ***false***.

❒ **Wzajemne wykluczanie** w maszynie zawierającej rozkaz **TestujUstal**

```
// definicja funkcji atomowej
boolean TestujUstal(boolean  arg)
  TestujUstal ← arg
  arg ← true
```

Jeżeli argument funkcji **TestujUstal** ma wartość ***false*** to zostaje zastąpiony wartością ***true***.

```
// T0     key= false        proces Pi
repeat
  while TestujUstal(key) do NIC
      Sekcja Krytyczna
  key ← false
      Reszta
until false
```

```
// T1     key= true         proces Pk
repeat
  while TestujUstal(key) do NIC
      Sekcja Krytyczna
  key ← false
      Reszta
until false
```

☛ Sekcję krytyczną uruchomi Proces, który napotka wartość ***key***=***false***.

Wykonanie funkcji **TestujUstal** zmienia wartość ***key*** na ***true***, co gwarantuje wykluczenie pozostałych procesów z dostępu do sekcji krytycznej.

☛ Proces, który **pierwszy** wykona funkcję **TestujUstal** może wykonać swoją Sekcję Krytyczną.

Inne procesy starające się realizować swoje sekcje krytyczne wchodzą w ***blokadę wirującą*** lub przechodzą w stan **Czekania**.

Gdy proces skończy przetwarzanie swojej sekcji krytycznej ustala wartość ***key*** na ***false***, dając tym samym szansę innym Procesom.

❒ **Wzajemne wykluczanie** w maszynie zawierającej rozkaz **Swap**

```
// definicja funkcji atomowej
Swap(boolean a, b)
boolean temp
  temp ← a
  a ← b
  b ← temp
```

Każdy Proces ma dodatkową **lokalną** zmienną z wartością początkową **mem** = ***true***.

```
// key = false                    proces Pi
repeat
 mem ← true
      repeat
          Swap(key, mem)
      until (mem = false)
 Sekcja Krytyczna                // key = true
   key ← false
       Reszta
until false
```

```
// key = true                     proces Pk
repeat
 mem ← true
      repeat
          Swap(key, mem)
      until (mem = false)
 Sekcja Krytyczna
   key ← false
       Reszta
until false
```

☛ Tylko proces, który **pierwszy** stwierdzi, że zmienna ***mem*** ma wartość ***true*** może rozpocząć przetwarzanie *swojej* sekcji krytycznej.

Wykluczenie pozostałych procesów gwarantuje nadanie zmiennej ***key*** wartości ***true***, która z powrotem przyjmuje wartość ***false*** po opuszczeniu przez proces sekcji krytycznej.

Oba algorytmy synchronizacji nie spełniają warunku **ograniczonego czekania**.

Zalety: dowolna liczba Procesów i CPU; możliwość kontroli wielu sekcji krytycznych.

Wady: aktywne czekanie; wybór kolejnego procesu jest arbitralny, co może grozić ***zagłodzeniem***.

6.4. Semafory

Holenderki matematyk Dijkstra w rozprawie z 1965 roku omawia konstruowanie SO w postaci zbioru współpracujących ze sobą sekwencyjnych procesów oraz realizacje wspomagających mechanizmów.

Zasada: Kilka procesów może ze sobą współpracować poprzez wymianę **prostych sygnałów**.

☛ Można wymusić zatrzymanie się procesu do czasu otrzymania odpowiedniego sygnału.

☛ Do sygnalizacji wykorzystuje się specjalne zmienne nazywane **semaforami**.

Proces transmituje sygnały poprzez **semafory**, wykonując funkcję ***Signal(S)***.

Do odbierania sygnałów przekazywanych semaforami służy funkcja ***Wait(S).***

Proces pozostaje w zawieszeniu, gdy oczekuje na możliwość przetransmitowania odpowiedniego sygnału.

Semafor S *(semaphore)* to **zmienna całkowita**, dostępna tylko za pomocą dwu operacji ***niepodzielnych***: **Czekaj** ora z **Sygnalizuj**.

Czekaj(S): { **while S ≤ 0 do** NIC; **S ← S - 1**;} // opuszczenie semafora

Sygnalizuj(S): **S ← S + 1**; // podniesienie semafora

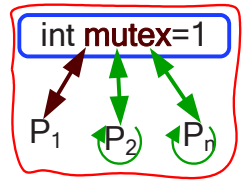
➔ Zmiany wartości semafora realizują **tylko** funkcje **Czekaj** i **Sygnalizuj**.

1. Semafory mogą być inicjowane z wartościami dodatnimi (standardowa wartość **1**).
2. Operacja **Czekaj(S)** lub **P(S)** zmniejsza wartość semafora.
   Dla ujemnej wartość semafora, proces wykonujący procedurę **Czekaj** jest zablokowany.
3. Operacja **Sygnalizuj(S)** lub **V(S)** zwiększa wartość semafora.
   Wznawia jeden ze wstrzymanych przez **Czekaj** procesów.
   Definicja nie określa, który proces zostanie odblokowany.

Jeżeli jeden proces modyfikuje wartość semafora to inny proces nie może tej wartości zmieniać.

Wykonie *Czekaj(S)* nie może być przerwane podczas sprawdzania wartości S i modyfikacji.

Semafory rozwiązują problem sekcji krytycznej z udziałem **n** procesów, (problem wzajemnych wykluczeni).



Wspólny semafor ***mutex*** dzielony jest przez **n** procesów.

Każdy z procesów **Pi** zorganizowany jest według schematu obok, który umożliwia wzajemne wykluczanie.

```
mutex ← 1                          // struktura Pi
repeat
  Czekaj(mutex)                    // mutex=0
  Sekcja krytyczna
  Sygnalizuj(mutex)                // mutex=1
      Reszta
until  false
```

Czekaj(S): { **while S ≤ 0 do** NIC; **S ← S - 1**;}

Na początku ***mutex*** ma wartość **1**.

W każdym Procesie operacja **Czekaj(*mutex*)** wykonywana jest bezpośrednio przed wejściem do sekcji krytycznej.

Jeśli wartość ***mutex*** jest **nie**dodatnia, proces zostaje zawieszony.

Kiedy wartość **mutex'a** wynosi **1** - *opuszcza pętlę i zostaje zmniejszona jego wartości do* ***0*** - zaś Proces przystępuje do przetwarzania swojej sekcji krytycznej.

Mutex przestaje być dodatni, żaden inny Proces nie może wejść do sekcji krytycznej.

Inicjacja nadaje **semaforowi** wartość **1**.

Pierwszy Proces, który wykona operację **Czekaj** natychmiast przetwarza ***swoją*** sekcję krytyczną.

Wyzerowanie semafora powoduje, że kolejne procesy usiłujące przetwarzać sekcję krytyczną -podejmując operację **Czekaj-** zostaną zablokowane.

Kiedy pierwszy Proces skończy sekcję krytyczną -wykona operację **Sygnalizuj**(***mutex***)- wartość semafora zostanie zwiększona i ***kolejny*** Proces rozpoczyna swoją sekcję krytyczną.

Zwykły semafor **nie** likwiduje problemu „**zagłodzenia**".

Algorytmy wzajemnych wykluczeń pozbawione „**głodzenia**" wymagają ***silnych semaforów***.

● **Semafory wspomagają rozwiązywanie problemów synchronizacji**

Rozważmy dwa współbieżnie wykonywane procesy: **P1** z instrukcją **Ins1**

P2 z instrukcją **Ins2**.

Instrukcja **Ins2** powinna zostać wykonana po **zakończeniu** wykonania instrukcji **Ins1**.

Wprowadzamy wspólną dla procesów **P1** i **P2** zmienną ***synch*** z początkową wartością **0**.

Następnie dołączamy do :

procesu **P1** instrukcje: **Ins1**;
Sygnalizuj(***synch***); // ustawia *synch* = **1**

procesu **P2** instrukcje: **Czekaj**(***synch***); // dla t_0 *synch* =0
Ins2;

Wartość początkowa ***synch*** = **0**,

proces **P2** wykona instrukcję **Ins2** dopiero po wywołaniu przez proces **P1** instrukcji **Sygnalizuj**(***synch***).

6.4.1. Silny semafor

Przedstawione wcześniej rozwiązania problemu wzajemnego wykluczania wymagają **aktywnego czekania**.

Gdy jeden Proces jest w swojej **sekcji krytycznej**, pozostałe Procesy usiłujące wejść do sekcji krytycznych muszą **nieustannie wykonywać instrukcję pętli** w sekcji wejściowej.

Aktywne czekanie marnuje cykle CPU.

Semafor tego rodzaju nazywany jest **wirującą blokadą** (*spinlock),* ponieważ oczekujący z powodu zamkniętego semafora Proces „wiruje" w miejscu.

Wirujące blokady przydatne są w systemach wieloprocesorowych.

Zaleta wirującej blokady:

Proces czekający pod SEMAFOREM nie wymaga przełączania kontekstu, przełączanie kontekstu jest kosztowne.

Modyfikując operacje **Czekaj** i **Sygnalizuj** można ominąć aktywne czekanie.

➡ Proces, który wykonuje operację **Czekaj** i zastaje niedodatnią wartość semafora, musi **CZEKAĆ**.

● **Zamiast czekać aktywnie, Proces można ZABLOKOWAĆ.** ●

Operacja blokowania umieszcza proces w kolejce związanej z **danym semaforem** i powoduje przełączenie stanu procesu na **CZEKANIE**.

Następnie sterowanie zostaje przekazane do planisty krótkoterminowego, który wybiera do wykonania inny Proces.

☛ Działanie Procesu **zablokowanego** pod semaforem **S** powinno zostać wznowione wskutek wykonania operacji **Sygnalizuj** przez inny Proces.

☛ Wznowienie Procesu realizuje operacja **BUDZENIE**, zmieniająca stan Procesu z **CZEKANIA** na **GOTOWOŚĆ**.

☛ Proces przechodzi do **kolejki** procesów **Gotowych** do wykonania.

Nowy semafor zdefiniowany jest jako struktura.

Ma pole z wartością całkowitą i listę procesów.

```
struct semaphore {
        int  wartosc;
        list of Proces  L;
        };
```

Kiedy proces wykonuje **Czekaj**(**S**) a wartość semafora jest **niedodatnia** musi **czekać** - dołącza się go do **Listy** procesów.

Czekaj(S)
S.wartosc ← **S.**wartosc **-** 1
if S.wartosc **<** 0 **then**
Dołącz dany proces do **S.L**
BLOKUJ ten proces

Sygnalizuj(S)
S.wartosc ← **S.**wartosc + 1
if S.wartosc **<** 0 **then**
Usuń proces **P** z **S.L**
OBUDZ(P)

Operacja **Sygnalizuj** usuwa jeden proces z **L**isty czekających procesów i **BUDZI** go.
Operacja **BLOKUJ** wstrzymuje proces, który ją wywołuje.
Operacja **OBUDZ(P)** wznawia zablokowany proces **P**.

```
repeat
  Czekaj(S)
      SekcjaKrytyczna
  Sygnalizuj(S)
      Reszta
until  false
```

Przedstawione Operacje dostarczane są przez **SO** jako funkcje systemowe.

W klasycznej definicji semafora z aktywnym czekaniem wartość semafora **nigdy nie jest** ujemna.

W powyższej implementacji semafory mogą przyjmować **wartości ujemne**.

☛ **Moduł** ujemnej wartości semafora określa **liczbę** procesów czekających na ten semafor.
Wynika to ze **zmiany porządku** instrukcji zmniejszania i sprawdzania wartości zmiennej w implementacji operacji **Czekaj**.

➔ Procesy zablokowane najdłużej opuszczają kolejkę pierwsze.

Semafory implementujące tę zasadę nazywa się **silnymi semaforami**.

Semafor zliczający *(counting semaphore)* - przebiera dowolne wartości całkowite.

Blokowanie nieskończone *(indefinite blocking)* zwane **głodzeniem** *(starvation)* - procesy czekają w nieskończoność pod semaforem.
Może powstać, jeśli przy dodawaniu i usuwaniu procesów z **L**isty związanej z semaforem użyje się porządku **LIFO** (ostatni na wejściu -pierwszy na wyjściu).

Listę czekających procesów można zbudować, dodając pole dowiązań do bloku kontrolnego każdego procesu.
Każdy semafor zawiera wartość całkowitą i wskaźnik do listy bloków kontrolnych procesów.
Kolejka →**FIFO**→ jest sposobem dodawania i usuwania procesów z **L**isty, gwarantującym **ograniczone czekanie**;
Semafor zawiera wskaźniki do początku i końca tej kolejki.

Problem zakleszczenia

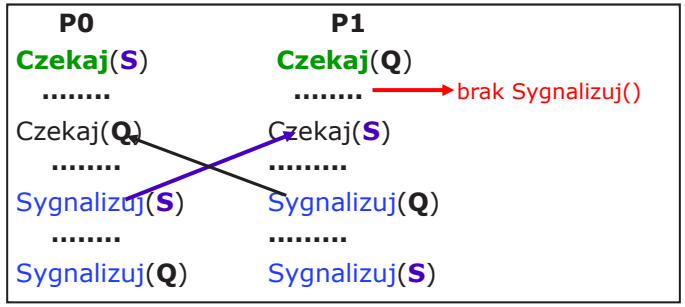
Semafor z kolejką oczekujących Procesów może spowodować, że kilka Procesów czeka w nieskończoność na **zdarzenie**, które może spowodować tylko ***jeden z czekających*** Procesów.

☛ Zdarzeniem tym jest wykonanie operacji **Sygnalizuj**.

Procesy, które znajdą się w takim stanie ulegają **zakleszczeniu** *(deadlock)*.

Dwa procesy **P0** i **P1** mają dostęp do **dwu** semaforów **S** i **Q**, ustawionych na **1**.

Niech proces **P0** wykona operację **Czekaj**(**S**),
a potem proces **P1** wykona operację **Czekaj**(**Q**).



Kiedy proces **P0** przejdzie do operacji **Czekaj**(**Q**),
będzie musiał czekać, aż proces **P1** wykona operację **Sygnalizuj**(**Q**).
Kiedy proces **P1** przejdzie do operacji **Czekaj**(**S**),
rozpocznie czekanie aby proces **P0** wykonał operację **Sygnalizuj**(**S**).

Operacje **Sygnalizuj** nie mogą już być wykonane, więc procesy **P0** i **P1** są **zakleszczone**.

Zbiór Procesów jest w stanie zakleszczenia, gdy każdy Proces w tym zbiorze oczekuje na zdarzenie, które może być spowodowane tylko przez inny Proces z tego ZBIORU.

Semafor binarny

Semafor binarny *(binary semaphore)* – jest zmienną logiczną i przybiera wartości **0** lub **1**.

Podnoszenie semafora

Sygnalizuj(BS): **BS** ← 1 // inne oznaczenie **VB**

Opuszczanie semafora

Czekaj(BS): **while BS** = 0 **do** NIC; **BS ← 0** // inne oznaczenie **PB**

Semafor binarny nie pamięta liczby wykonanych na nim operacji podnoszenia.

Semafor binarny rozwiązuje problem wzajemnego wykluczania.

Kolejne wykonania operacji **VB** na semaforze binarnym daje ten sam efekt co pojedyncze wykonanie.

W jakiej kolejności procesy wykonają operacje **PB** i **VB** zależy od prędkości CPU lub sposobu szeregowania procesów.

```
BS ← 1
repeat
  Czekaj(BS)            // SB=0
  Sekcja krytyczna
  Sygnalizuj(BS)        // SB=1
    Reszta
until false
```

Ponieważ semafor binarny nie pamięta liczby wykonanych operacji **VB**, więc sekwencja:

PB(BS), VB(BS), VB(BS), PB(BS)

może dać inny wynik niż sekwencja:

PB(BS), VB(BS), PB(BS), VB(BS).

Uniknięcie niejednoznaczności wymaga założenia, że **podnoszenie podniesionego** semafora binarnego jest ***błędem*** wykonania.

Semafor zliczający S

Semafor zliczający **S** można utworzyć przy użyciu semaforów binarnych.

```
semafor_binarny S1, S2
int C
```

Początkowo **S1 = 1**, **S2 = 0**.

Wartość zmiennej **C** określana jest według początkowej wartości semafora zliczającego **S**.

```
// operacja Sygnalizuj semafora S
Czekaj(S1);
C ← C + 1
if C ≤ 0 then Sygnalizuj(S2) else Czekaj(S1)
```

```
// operacja Czekaj semafora S
Czekaj(S1);   C ← C - 1
if C < 0 then
        Sygnalizuj(S1)
        Czekaj(S2)
Sygnalizuj(S1)
```


6.4.2. Podsumowanie

Błąd programisty nie zachowujący sekwencji: Czekaj(*mutex*)

```
...
Sygnalizuj(mutex)      // S = S + 1
```

Wszystkie procesy dzielą zmienną semaforową ***mutex***, początkowo równą **1**.

-Przed WEjściem do sekcji krytycznej każdy proces musi wykonać operację ***Czekaj(mutex)***.

-Przy WYjściu z sekcji krytycznej musi wykonaćoperację ***Sygnalizuj(mutex)***.

Niech nastąpi zamiana kolejność wykonania operacji ***Czekaj*** i ***Sygnalizuj*** na semaforze ***mutex***.

W tej sytuacji sekcje krytyczne może wykonywać jednocześnie wiele procesów, naruszając zasadę wzajemnego wykluczania.

```
Sygnalizuj(mutex)
Sekcja Krytyczna
Czekaj(mutex)
```

Wykrycie błędu może nastąpić tylko wtedy, gdy kilka procesów ***rzeczywiście*** będzie pracowało jednocześnie w sekcjach krytycznych.

➔ Sytuacja taka nie musi być odtwarzalna.

----------Wzajemne wykluczania dla **dwu procesów----------**

```
S = 1                  // proces P1
repeat
  Czekaj(S)            // S=0
  Sekcja krytyczna_P1
  Sygnalizuj(S)        // S=1
    Reszta P1
until false
```

```
S = 1                  // proces P2
repeat
  Czekaj(S)
  Sekcja krytyczna_P2
  Sygnalizuj(S)
    Reszta P2
until false
```

Proces **P1 chce** wejść do sekcji krytycznej, wykonuje instrukcję **Czekaj(S)**.

Jeśli **S** = 1, to **P1** wchodzi do swojej sekcji krytycznej i wartość **S** będzie zmniejszona.

Gdy **P1** wyjdzie z sekcji krytycznej, wykona instrukcję **Sygnalizuj(S)**, wartością **S** będzie znowu **1**.

Jeśli **P2** spróbuje wejść do swojej sekcji krytycznej, zanim **P1** wyjdzie, to **S** = 0 i **P2** zostanie wstrzymany przez **S**.

Gdy **P1** wyjdzie ze swojej sekcji, jego **Sygnalizuj(S)** wznowi **P2**.

Przedstawione rozwiązanie: -ma własności wzajemnego wykluczania,
-nie występuje blokada,
-nie dojdzie do zagłodzenia.

Jeśli **P1** jest wstrzymany, wartością semafora musi być **0**.
P2 jest w sekcji krytycznej.

Gdy **P2** wyjdzie z sekcji krytycznej, wykona **Sygnalizuj(S)**, co wznowi proces wstrzymany przez ***S***.

Wstrzymanym przez ***S*** jest **P1**, zostanie on wznowiony i wejdzie do swojej sekcji krytycznej.

❒ Dwie definicje semafora

● Definicja semafora Dijkstry

Opuszczenie: **Czekaj,** aż **S > 0**; **S ← S – 1** //while S ≤ 0 do NIC

Podniesienie: S ← S + 1

```
S = 0                                   // P1
repeat
  { while S ≤ 0 do NIC; S ← S – 1 }
  Sekcja krytyczna
  S ← S + 1
     Reszta
until  false
```

⇦20ms⇨

```
S = 0                                   // P5
repeat
  { while S ≤ 0 do NIC; S ← S – 1 }
  Sekcja krytyczna
  S ← S + 1
     Reszta
until  false
```

....

Jeśli proces ***rozpoczynający*** **Opuszczanie** semafora /**Czekaj**(S)/ stwierdzi, że jego wartość **nie jest dodatnia**, musi **czekać**;

zakończy **Opuszczanie** semafora, gdy INNY proces wykona operację **Podniesienia**.

W takim przypadku **Opuszczanie** musi być **przerwane** w trakcie sprawdzania warunku **S > 0**.

Podczas tej przerwy jakiś inny proces może ***rozpocząć*** **Opuszczanie** semafora i ta operacja także musi być **przerwana**.

➔ Może wystąpić wiele **rozpoczętych** a nie ***zakończonych*** operacji **Opuszczania**.

W chwili, gdy jakiś proces **Podniesie** semafor (zwiększy **S**), tylko jedna z operacji **Opuszczenia** wykona się do końca, ponieważ zbadanie warunku **S > 0** i wykonanie **S ← S - 1**, to jedna niepodzielna operacja.

☛ W definicji klasycznej przyjmuje się, że operacja **Opuszczenia** semafora jest **niepodzielna tylko wtedy**, gdy sprawdzany w niej **warunek** jest **spełniony**.

● Definicja semafora M. Ben-Ariego

Opuszczenie **Czekaj(S):** jeśli **S > 0**, to **S ← S - 1**,

w przeciwnym razie ***Wstrzymaj*** działanie Procesu wykonującego tę operację.

Proces został wstrzymany przez semafor **S**.

Podniesienie **Sygnalizuj(S):** jeśli są procesy ***Wstrzymane*** przez semafor **S**, to **WZNÓW** jeden z nich, w **przeciwnym** razie **S ← S + 1**.

Zgodnie w/wym definicją podniesienie semafora w chwili, gdy czekają na to inne procesy, powoduje, że któryś z nich **na pewno** będzie wznowiony.

Definicja klasyczna tego nie zapewnia.

Proces, który **Podniesie** opuszczony semafor, może zaraz potem **sam** wykonać operację **Opuszczania** i stwierdzić przed innymi procesami, że wartość semafora jest dodatnia, po czym zmniejszyć ją o jeden.

```
Repeat                  // P
  Czekaj(S)
    SekcjaKrytyczna
  Sygnalizuj(S) // S=S+1
     Reszta
until  false
```


Semafor w systemie wieloprocesorowym

Warunkiem poprawnego działania semaforów jest ich niepodzielne wykonywanie.

Należy zagwarantować, że żadne **dwa** Procesy nie wykonają operacji **Czekaj** ani **Sygnalizuj** na tym samym Semaforze w tym samym czasie.

Jest to problem sekcji krytycznej.

> W środowisku jednoprocesorowym można zabronić wykrywania przerwań podczas wykonywania operacji **Czekaj** i **Sygnalizuj**.
>
> Do chwili przywrócenia przerwań, kiedy to planista może znów przejąć nadzór nad CPU, działa tylko proces bieżący.

W środowisku wieloprocesorowym zakaz wykrywania przerwań nie skutkuje.

Rozkazy z różnych procesów (wykonywane w różnych CPU) mogą ulegać dowolnym przeplotom.

Jeśli nie ma specjalnych rozkazów sprzętowych, to można zastosować dowolne z poprawnych rozwiązań programowych problemu sekcji krytycznej.

➔W tym przypadku sekcje krytyczne będą się składały z funkcji **Czekaj** i **Sygnalizuj**➔

Definicje operacji **Czekaj** i **Sygnalizuj** nie wyeliminowały całkowicie aktywnego czekania.

> Nastąpiło **usunięcie *aktywnego czekania*** z Wejść do sekcji krytycznych w programach użytkowych.
>
> Ograniczono ***aktywne czekanie*** wyłącznie do sekcji krytycznych operacji **Czekaj** i **Sygnalizuj**, które są krótkie.

☛ ***Nowa*** Sekcja Krytyczna jest rzadko kiedy zajmowana i aktywne czekanie zdarza się rzadko.

☛ W programach **użytkowych** sekcje krytyczne mogą być długie lub prawie zawsze zajęte.

❒ Drobna uwaga

W środowisku **jedno**procesorowym problem sekcji krytycznej można rozwiązać:

wstrzymując obsługę **przerwań** administracyjnych

w trakcie realizacji ***kodu modyfikującego*** zmienne dzielone.

Daje to pewność, że ciąg rozkazów zostanie wykonany sekwencyjnie, bez wywłaszczenia.

Wykonanie innego rozkazu nie jest możliwe, więc nie nastąpi nieoczekiwana zmiana wspólnej zmiennej.

> Wyłączanie przerwań w środowisku **wieloprocesorowym** jest ***czasochłonne***, gdyż wymaga przekazywania komunikatów do wszystkich procesorów.

Opóźnia to wejście do każdej sekcji krytycznej, co powoduje spadek wydajności systemu.

6.4.3. Problem czytelników i pisarzy

Obiekt danych (np. plik) **podlega dzieleniu między kilka procesów współbieżnych.**

Niektóre z procesów będą tylko czytać, natomiast inne mogą go **też** uaktualniać (czytać i pisać).

Procesy realizujące czytanie nazywane są **Czytelnikami** – nie modyfikują zasobów.

Procesy modyfikujące zasoby nazywane są **Pisarzami**.

Jednoczesny dostęp wielu **Czytelników** do dzielonego obiektu nie jest groźny.

Gdyby **Pisarz** i inny proces (obojętne który) miały jednoczesny dostęp do dzielonego obiektu, mogłoby to spowodować **chaos**.

Należy zagwarantować wyłączność dostępu Pisarzy do obiektu dzielonego.

Problem synchronizacyjny nosi nazwę problemu **Czytelników i Pisarzy** (*Readers-Writers problem).*

Problem Czytelników i Pisarzy ma dwie główne wersje.
(wynikają one z zastosowanych priorytetów)

1-sa: żaden **Czytelnik** nie powinien czekać na dostęp do zasobu, chyba że ***właśnie*** **Pisarz otrzymał** ten zasób.

Przybywający Czytelnicy nie powinni czekać na zakończenie pracy ***innych*** Czytelników tylko dlatego, że **czeka Pisarz**.

(**Pisarz** wejdzie, gdy nie będzie już Czytelników - może otrzymać tylko dostęp wyłączny)

2-ga: gdy **Pisarz** jest gotowy (Czeka), to rozpoczyna pisanie tak szybko, jak to jest możliwe, tzn. zaraz po opuszczeniu zasobu przez ostatni Proces, który przybył **przed** nim.

Jeśli **Pisarz** czeka na dostęp do obiektu, to ***nowy*** **Czytelnik** nie rozpocznie czytania.

Każda z dwu wersji może powodować głodzenie.

W **1-szym** przypadku głodzenie grozi **Pisarzom**,

w **2-gim** **Czytelnikom**.

❒ **Schemat dla 1-szej** wersji (ryzyko głodzenia Pisarzy)

```
semaphore  mutex, pis
integer    liczba_Czyt
```

Procesy **Czytelników** dzielą zmienne:

mutex i **pis** -wartości początkowe **1**.

liczba_Czyt -liczba procesów czytających obiekt (wartość początkowa **0**)

pis -steruje wzajemnym wykluczaniem pracy **Pisarzy**,
(wspólny dla procesów Czytelników i Pisarzy)

jest używany przez: ***1-go wchodzącego*** lub
ostatniego opuszczającego sekcję krytyczną **Czytelnika**.

Nie używają go Czytelnicy wchodzący/wychodzący, *gdy inni Czytelnicy są w sekcjach krytycznych.*

mutex -gwarantuje wzajemne wykluczanie przy aktualizacji zmiennej **liczba_Czyt**.

Na **Pisarza** przebywającego w sekcji krytycznej oczekuje **n** Czytelników, to

- jeden Czytelnik stoi w kolejce do semafora ***pis,***
- **n-1** Czytelników ustawia się w kolejce do semafora ***mutex.***

Jeżeli **Pisarz** wykona ***Sygnalizuj(pis),*** to można:

-uaktywnić czekających Czytelników, lub

-uaktywnić pojedynczego **Pisarza**.

➔ Wybór należy do planisty.

```
// struktura procesu  Pisarza
Czekaj(pis)
  ........
  PROCES PISANIA
  .......
Sygnalizuj(pis)
```

```
                    // struktura procesu  Czytelnika
Czekaj(mutex)                          // mutex=0
  liczba_Czyt ← liczba_Czyt + 1
  if liczba_Czyt = 1 then Czekaj(pis)
Sygnalizuj(mutex)
    .........
      proces czytania                  // sekcja krytyczna
  Czekaj(mutex);
    ........
  liczba_Czyt ← liczba_Czyt - 1
  if liczba_Czyt = 0 then Sygnalizuj(pis)
  Sygnalizuj(mutex)
```

6.5. Monitory

W systemie wykorzystującym **wyłącznie semafory**, odpowiedzialność za poprawne ich użycie rozproszone jest między osoby implementujące system.

Brak w kodzie instrukcji **Sygnalizuj**(S) po sekcji krytycznej sprawi, że w systemie pojawią się, trudne w lokalizacji błędy.

Monitory dostarczają mechanizmu dla programowania współbieżnego, skupiając **odpowiedzialność** za poprawność w kilku modułach.

Sekcje krytyczne, takie jak przydzielanie urządzeń We/Wy, pamięci, itd., skupione są w uprzywilejowanym programie.

Można definiować oddzielne monitory dla każdego obiektu lub grupy obiektów.

Procesy żądają usług od różnych monitorów.

Jeśli **ten sam monitor** wywoływany jest przez 2 procesy, to **implementacja gwarantuje**, że te **dwa** procesy będą obsłużone z gwarancją wzajemnego wykluczania.

Składnia monitora hermetyzuje dane i działające na nich funkcje w pojedyncze moduły.

Interfejs monitora składa się ze zbioru funkcji, które operują na danych ukrytych w module.

Monitor zapewnia wzajemne wykluczanie wykonań funkcji występujących w jego interfejsie.

Konstrukcja stosowana w językach wysokiego poziomu do synchronizacji.

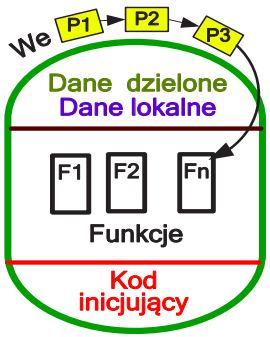
Monitory można implementować jako biblioteki programowe, co pozwala umieszczać monitorowe blokady w dowolnych obiektach.

Rozwiązanie zaproponował po raz pierwszy **C. Hoare**.

Implementację struktury monitorów zawiera wiele języków programowania (Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3 i Java).

Monitor to **moduł programowy**, składający się:

- z jednej lub kilku funkcji,
- ciągu inicjującego;
- danych lokalnych.



Rys. 6.1. Monitor

1. Dane lokalne dostępne są wyłącznie za pośrednictwem funkcji wewnątrz monitora (żadna funkcja zewnętrzna nie ma dostępu do zmiennych lokalnych).
2. **Proces** uruchamia monitor, wywołując jedną z jego funkcji.
3. W danej chwili pod kontrolą monitora może działać tylko **jeden** proces;

Inny proces, wywołujący w tym samym czasie monitor, zostanie ZAWIESZONY do czasu zwolnienia monitora.

Zmienne monitora dostępne są w tym samym czasie tylko **jednemu procesowi**, stąd **współdzielone struktury danych** można zabezpieczać umieszczając w **obrębie** monitora.

Monitor pełni rolę **mechanizmu wzajemnych wykluczeń**, jeśli w monitorze zawarte są zasoby, które powinny być chronione przed innymi procesami.

Monitor zawiera zbiór operacji zdefiniowanych przez programistę:

- deklaracje zmiennych, określających stan monitora,
- treści funkcji realizujących działania na nim.

☛ **Typu** monitor nie mogą używać bezpośrednio dowolne procesy.

Funkcje zdefiniowane wewnątrz **MONITORA** korzystają tylko ze zmiennych zadeklarowanych w nim lokalnie i ze swoich parametrów formalnych.

Zmienne lokalne w monitorze dostępne są tylko za pośrednictwem lokalnych funkcji.

Monitor gwarantuje, że w jego wnętrzu w danym czasie może być **aktywny** tylko **jeden proces**, więc programista **nie musi** budować jawnych barier synchronizacyjnych.

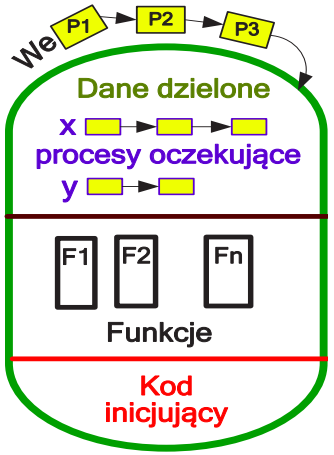
❁ Przetwarzanie współbieżne wymaga stosowych narzędzi synchronizacyjnych

Proces wywołuje monitor i pracując pod jego kontrolą może zostać w pewnym momencie **zawieszony** w oczekiwaniu na spełnienie określonego warunku.

Potrzebny jest mechanizm, który: -umożliwi zawieszenie procesu (będącego wewnątrz monitora),

-zwolni monitor, aby inny proces mógł go wykorzystać.

Gdy **oczekiwany warunek wystąpi**, a monitor znowu będzie dostępny, proces **musi** zostać wznowiony i uruchomić **MONITOR** od miejsca ostatniego zawieszenia.



Rys 6.2. Monitor ze zmiennymi warunkowymi

Zmienne warunkowe, dostępne tylko w obrębie MONITORA to dodatkowe narzędzia synchronizacyjne.

Można zdefiniować wiele zmiennych typu: ***condition*** **x, y,**

Zmienne typu ***condition*** dotyczą operacji:

Czekaj i **Sygnalizuj**.

Czekaj(***con***) -zawiesza wykonanie wywołującego tę operacje Proces i ustawia go na koniec kolejki związanej ze zmienną ***con***, z monitora może teraz korzystać inny Proces;

Sygnalizuj(***con***) -wznawia wykonywanie Procesu zawieszonego przez funkcję **Czekaj**(***con***) do czasu spełnienia określonego warunku.

Jeśli w stanie zawieszenia pozostaje kilka Procesów, funkcja wybiera **1-szy** z kolejki związanej ze zmienna ***con***;

Jeśli brak jest takich procesów, funkcja nic nie robi.

☛Proces wywołujący operację **Czekaj**(***con***) zostaje zawieszony do czasu, aż inny proces wywoła operację **Sygnalizuj**(***con***).

☛Operacja **Sygnalizuj**(***con***) wznawia jeden z zawieszonych procesów.

Jeśli żaden proces nie jest zawieszony, to stan zmiennej ***con*** jest taki, jak gdyby operacji **Sygnalizuj**(***con***) nie wykonano.

Proces uruchamia MONITOR przez wywołanie dowolnej, należącej do MONITORA funkcji.

Inne procesy, żądające dostępu do monitora, umieszczane są w kolejce.

➧ Proces działający pod kontrolą monitora może spowodować **swoje zawieszenie**, czekając na warunek ***x*** za pomocą funkcji **Czekaj**(***x***).

Tak zawieszony proces umieszczany jest w kolejce i może ponownie wywołać monitor, **gdy wystąpi** odpowiedni warunek.

Kiedy proces wykryje zmianę wartości zmiennej warunkowej ***x***, wykonuje operację **Sygnalizuj**(***x***), która sygnalizuje kolejce procesów oczekujących, że określony warunek został spełniony.

❒ **Wznawianie procesów** w **obrębie monitora**

Założenie: warunek **x** wstrzymuje kilka Procesów;

jeżeli jakiś proces wykona operację **Sygnalizuj**(**x**), to który ze wstrzymanych Procesów należy wznowić jako pierwszy?

Schemat najprostszy: proces oczekujący **najdłużej** zostanie wznowiony jako pierwszy.

Można zastosować konstrukcję warunkową:

Czekaj(*x*, prio)

gdzie **prio** -**numer priorytetu**, przechowywany wraz z nazwą zawieszanego Procesu.

Wykonanie **Sygnalizuj**(***x***), wznawia jako pierwszy, proces z najmniejszym numerem priorytetu.

```
// Monitor_przydziału-zasobu
boolean zajęty
condition x
Przydziel(int prio)
  if zajety then Czekaj(x, prio)
  zajety ← true
Zwolnij()
    zajety ← false
    Sygnalizuj(x)
zajety ← false
```

➔ Rozważymy monitor nadzorujący przydział pojedynczego zasobu rywalizującym procesom.

Każdy proces, ubiegając się o przydział zasobu, podaje **maksymalny czas** używania zasobu.

Monitor przydziela zasób temu spośród procesów, który zamawia go na **najkrótszy** czas.

Proces ubiegający się o dostęp do zasobu, musi działać w porządku podanym niżej, gdzie **OBM** jest obiektem typu ***Monitor_przydziału***-zasobu.

```
OBM.Przydziel(p)
dostęp do zasobu
OBM.Zwolnij
```

❒ Potencjalne błędy nie wykrywane przez Monitor

Błędy w programowaniu dostępu:

dostęp do zasobu **OBM**.Przydziel(p) **OBM**.Zwolnij	**OBM**.Zwolnij **dostęp do zasobu** **OBM**.Przydziel(p)

W wyniku **błędnego kodu** proces może:

- uzyskać dostęp do zasobu bez otrzymania pozwolenia na jego używanie,
- nigdy nie zwolnić zasobu, po uzyskaniu do niego dostępu,
- usiłować zwolnić zasób, którego nigdy nie zamawiał,
- zamówić ten sam zasób dwukrotnie (nie zwalniając go uprzednio).

Jednym z możliwych rozwiązań powyższego problemu jest umieszczenie operacji dostępu do zasobu **wewnątrz** danego **monitora**.

Wtedy planowanie dostępu odbywałoby się według algorytmu wbudowanego w monitor, **a nie przez algorytm użytkownika**.

➔ Zapewnienie poprawności systemu wymaga sprawdzenie dwóch warunków:

1. w procesach użytkownika musi być zachowana poprawna **kolejność** wywołań MONITORA.
2. żaden niezależny proces nie pominie możliwości wzajemnego wykluczania organizowanego przez MONITOR i **nie spróbuje** uzyskać **bezpośredniego** dostępu do zasobu dzielonego.

Tylko spełnienie obu warunków może gwarantować, że nie wystąpią błędy synchronizacji i algorytm planowania nie ulegnie załamaniu.

Sprawdzenie obu warunków możliwe jest tylko w małym, statycznym systemie.

🕮 Przeanalizować poniższe problemy 🕮

🕮 Problem 1

- ☛ Gdy proces **P** wywołuje operację **Sygnalizuj(x)**, istnieje ***zawieszony*** proces **Q**, związany z warunkiem **x**.
- ☛ **Jeśli** proces **Q** *chce* wznowić działanie, to sygnalizujący to proces **P** musi poczekać;
  w przeciwnym razie **P** i **Q** stałyby się jednocześnie aktywne wewnątrz monitora.

Istnieją dwie możliwości:

1. **P** zaczeka, aż **Q** opuści monitor, albo zaczeka na inny warunek (schemat Hoare).
2. **Q** zaczeka, aż **P** opuści monitor, albo zaczeka na inny warunek

Proces **P** działa już w obrębie monitora, co sugeruje wybór wersji **2**.

Zezwalając procesowi **P** kontynuować pracę, to warunek, w oczekiwaniu na który **Q** pozostawał wstrzymany, może nie być już spełniony, gdy **Q** zostanie wznowiony.

Rozwiązanie pośrednie: Gdy proces **P** wykonuje operację **Sygnalizuj(x)**, wówczas natychmiast opuszcza monitor, i proces **Q** jest niezwłocznie wznawiany.

🕮 Problem 2

Problem obsługi istniejących kolejek na zewnątrz i wewnątrz **Monitora**.

- ☛ Proces **P wykonuje** funkcję **Monitora**, inne Procesy, które w tym samym czasie też chcą wejść do Monitora (rozpocząć wykonanie jego funkcji), czekają w **kolejce Wejściowej** (zewnętrznej)
- ☛ Procesy wstrzymane w wyniku wykonania operacji ***Czekaj(con),*** czekają (wewnątrz Monitora) w **kolejkach związanych ze zmiennymi *condition***.
- ☛ Na wejście do **Monitora** mogą też czekać procesy, które wykonały operację ***Sygnalizuj(con)***, ale same zostały **Wstrzymane**;
  są one **odłożone na stos**, wracają do Monitora w kolejności **odwrotnej do tej**, w jakiej go opuszczały.

Proces **P** będący wewnątrz **Monitora** wykonał operację ***Sygnalizuj(con)***, i został odłożony na stos procesów, a do **Monitora** wejdzie pierwszy czekający w kolejce związanej ze zmienną **con**.

Jeśli proces **P** opuści **Monitor** w jednym z dwu przypadków:
-wykonując operację ***Czekaj(con),*** (będzie wstrzymany w kolejce związanej z ***con***),
-kończąc wykonywanie procedury **Monitora**,
to do Monitora wejdzie proces odłożony, który poprzednio wznowił proces **P**.

Jeśli takiego procesu nie ma, to stos procesów odłożonych jest **pusty** i do **Monitora** może wejść proces czekający w kolejce **Wejściowej** (zewnętrznej)

6.6. Transakcje niepodzielne

Jeśli **dwie sekcje krytyczne** wykonywane są współbieżnie, to wynik jest równoważny wykonaniu ich **po kolei,** ale w **nieznanym** porządku.

```
S = 1                    // P1
.....
Czekaj(S)                // S=0
   Sekcja
 Krytyczna P1
Sygnalizuj(S)            // S=1
.....
```

```
S = 1                    // P2
.....
Czekaj(S)                // S=0
   Sekcja
 Krytyczna P2
Sygnalizuj(S)            // S=1
.....
```

Istnieją sytuacje, w których musimy **mieć pewność**, że sekcja krytyczna zostanie wykonana w oczekiwanej KOLEJNOŚCI albo **nie wykonuje się jej**.

W przelewach pieniężnych istotna jest kolejność WPŁAT i POBRAŃ.

System operacyjny można rozpatrywać jako manipulatory danych, i wykorzystywać w nim techniki oraz **modele baz danych.**

Techniki zarządzania plikami można uogólnić, uwzględniając metody baz danych.

Transakcja to zbiór instrukcji (operacji), które wykonują logicznie **spójną** funkcję.

☛ Przetwarzanie transakcji **musi** być **niepodzielne** pomimo **awarii** systemu.

Transakcja to fragment programu, który ma dostęp do obiektów danych przechowywanych w różnych plikach na **dysku**.

Transakcja to ciąg operacji **Czytaj** i **Pisz**, zakończonych operacją **Zatwierdź** lub **Zaniechaj**.

Zatwierdź -transakcja zakończyła się pomyślnie.

Zaniechaj -wykonanie transakcji nie dobiegło do końca z powodu błędu logicznego.

☛ Skutków transakcji **zatwierdzonej** nie można cofnąć przez **Zaniechanie** transakcji.

● **Transakcja może nie zakończyć się z powodu awarii systemu.**

Stan danych dostępnych w transakcji może być wówczas inny niż po niepodzielnym jej wykonaniu, ponieważ zaniechana transakcja mogła już zdążyć pozmieniać niektóre dane.

Niepodzielność transakcji wymaga, aby transakcja **Zaniechana** nie pozostawiła śladów w danych, które zdążyła już zmienić.

Dane ***zmienione*** przez **zaniechaną** transakcję należy **odtworzyć** do stanu, jaki był przed rozpoczęciem wykonywania transakcji.

Mówimy, że transakcja została **wycofana** *(rolled back).*

6.6.1. Współbieżne transakcje niepodzielne

Współbieżne wykonanie transakcji realizowane jest kolejno w pewnym dowolnym porządku.

Tę cechę, zwaną **szeregowalnością** *(serializability)*, można realizować wykonując każdą transakcję w **`sekcji krytycznej`**, co zapewnia niepodzielność współbieżnie wykonywanych transakcji.

```
S = 1
.....
Czekaj(S)                //S=0
  Transakcja T1
Sygnalizuj(S)            // S=1
.....
```

```
S = 1
.....
Czekaj(S)                //S=0
  Transakcja T2
Sygnalizuj(S)            // S=1
.....
```

-Transakcje korzystają ze wspólnego **S**emafora ***S*** (na początku równego **1**).

-Transakcja rozpoczyna działanie od wykonania operacji **Czekaj(*S*).**

-Po zatwierdzeniu transakcji lub jej zaniechaniu, wykonuje się operację **Sygnalizuj(*S*)**.

Istnieją sytuacje, w których ***można dopuścić*** do zachodzenia w tym samym czasie działań wykonywanych w TRANSAKCJACH przy zachowaniu szeregowalności.

❒ **Plan szeregowy**

Transakcje **T0** i **T1** **Czytają** i **Zapisują** dwa zbiory danych **A** i **B**.

Transakcje wykonywane są niepodzielnie i transakcja **T0** poprzedza transakcję **T1**.

Plan szeregowy *(serial schedule)*: każda transakcja wykonywana jest niepodzielnie.

Jest poprawny, gdyż jest równoważny niepodzielnemu wykonaniu poszczególnych składowych transakcji w pewnym dowolnym porządku.

Plan nieszeregowy: jeżeli instrukcje wchodzące w skład dwu transakcji będą się przeplatać.

Plan nieszeregowy może czasami dawać wyniki równoważne szeregowemu.

Plan 1 -szeregowy

	T0	T1
1	Czytaj(A)	↓
2	**Pisz(A)**	
3	**Czytaj(B)**	
4	Pisz(B)	
5		**Czytaj(A)**
6		**Pisz (A)**
7		Czytaj(B)
8		Pisz(B)

Operacja konfliktowa *(conflicting operations).*

W planie **P** występują **po sobie** dwie operacje O_i i O_k należące odpowiednio do transakcji T_i i T_k

Operacje O_i i O_j pozostają w **konflikcie**, jeżeli działają na tych samych danych i co najmniej jedna z nich jest operacją **Pisania**.

W **Planie 1** operacja **Pisz(A)** należąca do transakcji **T0** jest w konflikcie z operacją **Czytaj(A)** należącą do transakcji **T1.**

Operacja **Pisz(A)** z transakcji **T1** **nie** pozostaje w konflikcie z operacją **Czytaj(B)** z transakcji T**0** gdyż dotyczą innych danych.

Niech O_i i O_k będą sąsiednimi operacjami w planie **P**.

Jeśli O_i i O_k są operacjami w **różnych** transakcjach oraz O_i i O_k nie pozostają w konflikcie, to wolno zamienić ***porządek wykonania*** operacji O_i i O_k, co daje nowy plan **P***.

Oczekuje się, że plan **P***, będzie równoważny planowi **P** gdyż wszystkie operacje występują w obu planach w tym samym porządku z wyjątkiem O_i i O_k, których kolejność wykonania jest bez znaczenia.

Operacja **Pisz(A)** z transakcji **T1** nie pozostaje w konflikcie z operacją **Czytaj(B)** z transakcji **T0**, więc można zamienić te operacje w celu utworzenia równoważnego planu.

Plan 2 -nieszeregowy

	T0	T1
1	Czytaj(A)	
2	**Pisz(A)**	
3		**Czytaj(A)**
4		**Pisz(A)**
5	**Czytaj(B)**	
6	Pisz(B)	
7		Czytaj(B)
8		Pisz(B)

Plan 1 - szeregowy

	T0	T1
1	Czytaj(A)	
2	**Pisz(A)**	
3	**Czytaj(B)**	
4	Pisz(B)	
5		**Czytaj(A)**
6		**Pisz (A)**
7		Czytaj(B)
8		Pisz(B)

Dalszy ciąg bezkonfliktowych zamian kolejności wykonania operacji:

- **Czytaj**(**B**) z transakcji **T0** ↔ **Czytaj**(**A**) z transakcji **T1**
- **Pisz**(**B**) z transakcji **T0** ↔ **Pisz**(**A**) z transakcji **T1**
- **Pisz**(**B**) z transakcji **T0** ↔ **Czytaj**(**A**) z transakcji **T1**

W efekcie powstaje *nieszeregowy* **Plan 2,** który jest równoważny planowi szeregowemu **Plan 1**.

Niezależnie od początkowego stanu systemu, końcowy wynik działania **Planu 2** będzie taki sam jak plan szeregowego **Plan 1**.

Plan szeregowalny z **uwzględnieniem konfliktów** *(conflict serializable):*

jeżeli dany plan można przekształcić w plan szeregowy za pomocą ciągu zamian bezkonfliktowych operacji.

Plan 2 jest szeregowalny z uwzględnieniem konfliktów.

ANEX 6.1 Algorytm piekarni *(bakery algorithm)*

Jest to ***programowy*** **algorytm wieloprocesowy** (dla **n** procesów)

Przy wejściu do piekarni każdy klient dostaje numerek.
Obsługę rozpoczyna się od klienta z najmniejszym numerem.

Algorytm piekarni nie gwarantuje, że dwa procesy (klienci) nie dostaną tego samego numeru.

➔ Jeśli P_i i P_k otrzymają ten sam numer oraz **i < k**, to P_i będzie obsłużony jako pierwszy.

Nazwy procesów są jednoznaczne uporządkowane, zatem algorytm jest deterministyczny.

Algorytm wykorzystuje dwie wspólne Tablice o wartościach początkowych **false** i **0**:

-tablicę logiczną **Znacznik[0..n - 1]**,

-numeryczną **numer[0..n - 1]**.

Definicja pomocnicza: (**a**, b) < (**c**, d) ≡ jeśli **a** < **c** lub jeśli a = c i b < d.

```
                                                    // proces Pi
repeat
  Znacznik[i] ← true
  numer[i] ← max(numer[0], ..., numer[n-1]) + 1
  for k ← 0 to n - 1 do
      while Znacznik[k] do NIC
      while numer[k] ≠ 0 and (numer[k], k) < (numer[i], i) do NIC
  Sekcja Krytyczna
  numer[i] ← 0;
  Znacznik [i] ← false
  Reszta
until false
```

Można wykazać że:

jeżeli proces P_i znajduje się w sekcji krytycznej, a proces P_z ma wybrany swój **numer[z] ≠ 0** to

(numer[i], i) < (numer[z], z)

Dla wykazania wzajemnego wykluczania zakłada się, że proces P_i, jest w sekcji krytycznej, a P_z próbuje wejść do sekcji krytycznej.

Gdy proces P_z przejdzie do wykonania drugiej instrukcji ***while*** przy **k = i**, wówczas sprawdzi, że

- numer[i] ≠ 0
- (numer[i], i) < (numer[z], z)

i będzie wykonywać pętlę ***while*** dopóki proces P_i, nie wyjdzie ze swojej sekcji krytycznej.

Warunki ***postępu*** i ***ograniczonego czekania*** są spełnione.

Wynika z faktu że procesy wchodzą do sekcji krytycznych na zasadzie „pierwszy zgłoszony - pierwszy obsłużony".

ANEX 6.2. Problem obiadujących filozofów

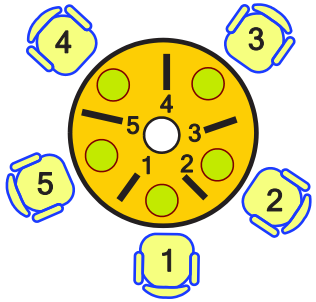
Pięciu filozofów siedzi przy okrągłym stole i zajmują się tylko myśleniem i jedzeniem.

Na środku stołu stoi miska ryżu, a naokoło leży **pięć** pałeczek.

Kiedy filozof myśli, to tylko myśli.

Czasami filozof jest głodny.

Wówczas próbuje ująć dwie pałeczki leżące najbliżej jego miejsca.



☛Za każdym razem filozof może podnieść tylko **jedną** pałeczkę.

Kiedy filozof **zdobędzie obie** pałeczki, rozpoczyna jedzenie, nie rozstając się z pałeczkami ani na chwilę.

Po spożyciu posiłku filozof odkłada obie pałeczki na stół.

Problem filozofów to klasyczne zagadnienie synchronizacji.

Jak przydzielić zasoby do wielu procesów aby uniknąć zakleszczenia i głodzenia.

❒ **Proste rozwiązanie:** każda **pałeczka** jest **semaforem.**

semaphore Paleczka[5] = {1, 1, 1, 1, 1} // numeracja wektora od **1**

```
                                  // struktura i-tego filozofa
repeat
 Czekaj(Paleczka[i])
 Czekaj(Paleczka[(i + 1) mod 5])
   ..........
   PROCES JEDZENIA        // sekcja krytyczna
   ..........
 Sygnalizuj(Paleczka[i])
 Sygnalizuj(Paleczka[(i+1) mod 5])
   ..........
   Proces myślenia
   .........
until false
```

1 mod 5 = 1
2 mod 5 = 2
3 mod 5 = 3
4 mod 5 = 4
5 mod 5 = 0

Biorąc pałeczkę filozof wykonuje operację **Czekaj** odnoszącą się do danego semafora.

Odkładając pałeczkę wykonując operację **Sygnalizuj** dotyczącą odpowiedniego semafora.

Proste rozwiązanie **zakłada**, że żadni **dwaj sąsiedzi** nie będą jedli jednocześnie.

Nie wyklucza powstania zakleszczenia.

5-ciu filozofów poczuło głód jednocześnie i każdy podniósł jedną pałeczkę leżącą po jego ***lewej*** stronie.

Elementy tablicy **Paleczka** przyjmują wartość **0**.

Próby któregokolwiek z filozofów podniesienia **prawej** pałeczki zakończą się niepowodzeniem.

Sposoby rozwiązywania problemu zakleszczenia:

- co najwyżej **czterech** filozofów może zasiadać naraz,
- filozof może podnieść pałeczki tylko wtedy, gdy **obie** są dostępne (czynność realizowana w sekcji krytycznej)
- zastosować rozwiązanie asymetryczne:
  -filozof o numerze nieparzystym podnosi najpierw pałeczkę z lewej strony, a potem z prawej,
  -filozof *parzysty* rozpoczyna od pałeczki z prawej strony, a potem sięga do lewej.

Rozwiązania problemu filozofów muszą gwarantować, iż żaden z nich nie zostanie zagłodzony.

Rozwiązanie wolne od zakleszczeń nie eliminuje automatycznie możliwości blokowania nieskończonego.

❁ **Jeden z filozofów staje się Kierownikiem** ❁

Metodyka ogranicza liczbę przebywających filozofów w jadalni do 4-ch.

Wprowadza się dodatkowy semafor ***kierownik*** o wartości początkowej **4**.

Zatem dopuszcza do rywalizacji o pałeczki co najwyżej **czterech** filozofów.

semaphore *kierownik* = **4**

Semafor ***kierownik*** jest semaforem z kolejką oczekujących.

```
                                  // struktura i-tego filozofa
repeat
 Czekaj(kierownik)
   Czekaj(paleczka[i])
     Czekaj(paleczka[(i + 1) mod 5])
   ..........   PROCES JEDZENIA
     Sygnalizuj(paleczka[i])
   Sygnalizuj(paleczka[(i+1) mod 5])
 Sygnalizuj(kierownik)
   Proces myślenia
   .........
until false
```

Rozwiązanie gwarantuje uniknięcie zakleszczenia.

Dowód poprawności działania zawarty jest w poz. 3 literatury uzupełniającej.

ANEX 6.3. Niepodzielność transakcji, gdy awarie powodują utratę informacji w PAO

❒ Odtwarzanie za pomocą rejestru

Niepodzielność uzyskuje się, zapisując w pamięci **TRWAŁEJ** informacje określające zmiany wykonywane przez transakcję w danych, do których ma ona dostęp.

Metoda rejestrowania z **wyprzedzeniem** *(write-ahead logging).*

System utrzymuje w pamięci **trwałej** strukturę danych nazywaną **rejestrem** *(log).*

Każdy rekord **rejestru** opisuje jedną operację **Pisania** w transakcji.

Rekord **rejestru** zawiera następujące pola:

- **nazwa transakcji** -nazwa transakcji wykonującej operację pisania,
- **nazwa jednostki danych** -nazwa zapisywanej jednostki danych,
- **stara** wartość -wartość jednostki danych przed zapisem,
- **nowa** wartość -wartość jednostki danych po zapisie.

Rejestr zawiera też rekordy przechowujące istotne zdarzenia występujące podczas przetwarzania transakcji, takie jak:

-początek transakcji,

-jej zatwierdzenie lub zaniechanie.

Przed wykonaniem transakcji T_i, w **rejestrze** zapisuje się rekord: <T_i **, rozpoczęcie**>.

➔ **Każda** należąca do wykonywanej transakcji operacja **Pisz** jest **poprzedzana** wpisem do rekordu w **rejestrze**.

Zatwierdzenie transakcji tworzy **zapis** w **rejestrze** jako rekord: <T_i **, zatwierdzenie**>.

Zawartość **rejestru** używa się do rekonstrukcji danych przetwarzanych przez różne transakcje.

Nie zezwala się na faktyczne uaktualnienie jednostki danych, ***zanim nie zostanie zapisany*** odpowiedni rekord w rejestrze przechowywanym w pamięci trwałej.

Przed wykonaniem operacji **Pisz(x),** należy w trwałym rejestrze zapisać rekord dotyczący **x**.

Algorytm rekonstrukcji korzysta z dwu funkcji:

Wycofaj: odtwarza wszystkie dane uaktualnione przez transakcję T_i nadając im stare wartości;

Przywróć: nadaje nowe wartości wszystkim danym uaktualnionym przez transakcję T_i.

Rejestr zawiera zbiór danych uaktualnionych przez transakcję T_i oraz odpowiadających im starych i nowych wartości.

Operacje **idempotentne:** wielokrotne ich wykonywanie daje ten sam efekt co wykonanie jednorazowe.

Gwarancją poprawności działania w przypadku awarii występującej podczas odtwarzania jest **idempotentność** operacji:

Wycofaj i **Przywróć**.

W przypadku **Zaniechania** transakcji T_i odtworzenie stanu zmienionych przez nią danych realizuje operacja **Wycofaj**(T_i).

Po awarii systemu, na podstawie analizy **rejestru** ustala się, które transakcje należy **przywrócić**, a które **wycofać**.

-Transakcja T_i musi być **wycofana**, jeżeli w rejestrze znajduje się rekord <T_i **, rozpoczęcie>** lecz **nie ma** w nim rekordu <T_i **, zatwierdzenie>**.

-Transakcja T_i, musi być **przywrócona**, jeżeli rejestr zawiera dwa rekordy:

<T_i **, rozpoczęcie>** i <T_i **, zatwierdzenie>**.

❒ Punkty kontrolne

Rozstrzygnięcie, które transakcje muszą być przywrócone, a które wycofane często wymaga przeglądania całego rejestru.

- Proces przeglądania jest czasochłonny.
- Część transakcji, których skutki powinny być przywrócone, dokonało już aktualizacji danych, o których na podstawie rejestru można by wnosić, że wymagają modyfikacji.

  Powtórne wykonanie tych zmian nie spowoduje szkody (idempotentność działań), jednak wydłuży czas rekonstrukcji.

Punkty kontrolne *(checkpoints)* pozwalają zmniejszyć koszta przywracania.

System **rejestruje z wyprzedzeniem** operacje **Pisania** i **co pewien** czas tworzy **punkty kontrolne**, wykonując następujące czynności:

1. wszystkie rekordy aktualnie będące w **PAO** są **zapisywane** w pamięci **TRWAŁEJ**.
2. wszystkie zmienione dane w pamięci **ulotnej** są umieszczane w pamięci **TRWAŁEJ**.
3. w **rejestrze** transakcji (w pamięci trwałej) zapisuje się rekord **<punkt kontrolny>**.

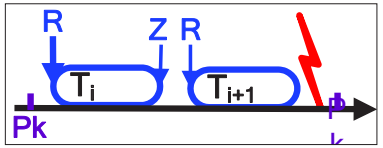
● Zatwierdzenie transakcji T_i nastąpiło przed punktem kontrolnym.

Rekord <T_i **, zatwierdzenie>** pojawia się w rejestrze **przed** rekordem **<punkt kontrolny>**.

Zmiany dokonane przez transakcję T_i, musiały być zapisane w pamięci trwałej albo przed punktem kontrolnym, albo w trakcie wykonywania operacji transakcji.

W czasie rekonstrukcji operacja **Przywróć** w odniesieniu do transakcji T_i, jest zbyteczna.

Po awarii procedura rekonstrukcji przegląda **rejestr** aby odnaleźć ostatnią transakcję **T,** której wykonywanie rozpoczęto **przed** wstawieniem ostatniego punktu kontrolnego.



Wsteczny przegląd rejestru odnajduje:

- pierwszy rekord **<punkt kontrolny>,**
- następnie 1-szy rekord < T_i **rozpoczęcie>** występujący po nim.

Po odnalezieniu transakcji T_i, operacje **Przywróć** i **Wycofaj** wykonuje się tylko w odniesieniu do samej transakcji T_i i **tych**, których wykonywanie **rozpoczęło się po niej**.

Oznaczmy zbiór tych transakcji jako **T** *(pomijając resztę rejestru).*

Dla wszystkich transakcji T_k ze zbioru **T** dla których w rejestrze:

-występuje rekord < T_k, **zatwierdzenie>** należy:

wykonać operację **Przywróć**(T_k).

-**nie** występuje rekord < T_k, **zatwierdzenie>** należy:

wykonać operację **Wycofaj**(T_k).

ANEX 6.4. Sekcja krytyczna w systemie Windows

Sekcja krytyczna to **fragment kodu**, który **musi uzyskać** wyłączny dostęp do dzielonego zasobu, **zanim** rozpocznie się wykonać.

Utworzyć zmienną strukturalną typu:

```
CRITICAL_SECTION csZasób;
```

Zainicjalizować składowe zmiennej strukturalnej:

```
InitializeCriticalSection(&csZasób);
```

Umieścić między wywołaniami poniższych funkcji odwołania do dzielonych zasobów

```
EnterCriticalSection(&csZasób);
fragment kodu działający na zmiennych dzielonych
LeaveCriticalSection(&csZasób);
```

Wątki korzystające z zasobu muszą znać adres chroniącej go zmiennej strukturalnej.

Funkcja **EnterCriticalSection** sprawdza czy jakiś wątek **już** korzysta z podanej sekcji krytycznej.

Jeżeli inny wątek niż wołający **ma** już **dostęp** do sekcji krytycznej, ***EnterCriticalSection*** wprowadza **wołający** wątek w stan **Oczekiwania**.

System pamięta o czekającym wątku i jak tylko aktualny „użytkownik" sekcji krytycznej wywoła funkcję ***LeaveCriticalSection*** wznowi działanie zawieszonego wątku.

Sekcje krytyczne nie nadają się do synchronizowania wątków w różnych Procesach

---

Struktura CRITICAL_SECTION zdefiniowana jest w pliku WinNT.h
jako RTL_CRITICAL_SECTION,
struktura RTL_CRITICAL_SECTION jest typem zdefiniowanym w WinBase.h.

W WINDOWS wątki czekające na sekcję krytyczną nigdy nie zostają zablokowane.

Wywołanie ***EnterCriticalSection*** ulega przeterminowaniu, co powoduje zgłoszenie wyjątku.

Czas [s], który musi upłynąć określa wartość *CritcalSectionTimeout* w podkluczu rejestru:
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager
Standardowo wynosi 2 592 000, czyli około 30 dni.
Nie należy ustawiać poniżej 3 sekund.

Jeśli aplikacja zawiera kilka niepowiązanych ze sobą dzielonych zasobów, efektywniej jest utworzyć dla każdego z nich **ODDZIELNĄ** zmienną typu CRITICAL_SECTION.

```
double tabA[MAX], tabC[MAX];          // zmienne globalne

CRITICAL_SECTION  cs_A,  cs_C;

UINT WINAPI FuncThreadInit(LPVOID par)
{
int i;
        ...KOD FUNKCJI;
EnterCriticalSection(&cs_A);
   for (i = 0; i < MAX; i++ )  tabA[i] = Generuj(a, b);
LeaveCriticalSection(&cs_A);
        ...CIĄG DALSZY KODU.....;
...     CIĄG DALSZY KODU.....;
   EnterCriticalSection(&cs_C);
     for (i = 0; i < MAX; i++ )  tabC[i] = pow(i, 2.2);
   LeaveCriticalSection(&cs_C);
        ...KOD FUNKCJI;
return(0);
}
```

Zamiast **EnterCriticalSection** można użyć funkcji:

BOOL **TryEnterCriticalSection**(*PCRITICAL_SECTION* ***pcs***);

Funkcja **nie** przenosi wywołującego *swoją sekcję krytyczną* wątku w stan **Oczekiwania**.

Jeśli sekcja jest akurat zajęta przez inny wątek, funkcja zwraca FALSE.

W każdym innym przypadku zwraca wartość TRUE.

Wątek może szybko **sprawdzić**, swoje **szanse** na dostęp do zasobu dzielonego, zamiast przechodzić w stan **O**czekiwania.

❒ **Blokada wirowa** aktywna tylko dla maszyn wieloprocesorowych

Standardowo Wątek próbujący wejść do sekcji krytycznej zajętej przez inny Wątek, przechodzi natychmiast w stan **Oczekiwania**; zmienia swój tryb wykonania z użytkownika na tryb jądra (zajmuje około 1000 cykli CPU).

☛ Windows umożliwia włączenie **blokady wirowej** do obsługi sekcji krytycznych.

BOOL **Initialize**CriticalSectionAndSpinCount(
PCRITICAL_SECTION *pcs*,
DWORD ***dwSpinCount***
);

Z chwilą wywołania *EnterCriticalSection* funkcja uruchamia **pętlę**, w której próbuje określoną liczbę razy (***dwSpinCount***) uzyskać dostęp do zasobu.

Gdy próby zakończą się niepowodzeniem, następuje przejście na tryb jądra i wejście w stan Oczekiwania.