Wykład 3.
Zarządzanie procesami.
Synchronizacja.

Jarosław Koźlak

Przetwarzanie równoległe a przetwarzanie sekwencyjne 1

- **Dane współdzielone**
- [J. Martyna, Wstęp do projektowania systemów operacyjnych]
- Przykład: obliczanie wartości wyrażenia:
- x:= (a+b)(a-b)/(c-d)(a/b-b/a)

- **Rozwiązanie sekwencyjne:**

1. a+b
2. a-b
3. (a+b)(a-b)
4. c-d
5. a/b
6. b/a
7. (a/b-b/a)
8. (c-d)(a/b-b/a)
9. (a+b)(a-b)/(c-d)(a/b-b/a)

Przetwarzanie równoległe a przetwarzanie sekwencyjne 2

1. **parbegin**

    wynik1:= a+b;

    wynik2:= a-b;

    wynik3:=c-d;

    wynik4:=a/b;

    wynik5:=b/a;

parend

2 wynik6:=wynik1*wynik2;

3 wynik7:=wynik4*wynik5;

4 wynik8:=wynik3*wynik7;

5 wynik9=wynik6/wynik8;

KOORDYNOWANIE PROCESÓW.

W wielozadaniowych systemach operacyjnych :
wiele procesów
mechanizmy synchronizacji i komunikacji między procesami
Przykład : dzielenie zmiennej.
dwa procesy P1 (rejestr R1) i P2 (rejestr R2)
a =5
P1: a=a+1
P2: a=a-1
Operacje P1.
R1 :=a;
R1= R1+1;
a= R1;
Operacje P2
R2 :=a;
R2= R2 -1;
a= R2;

t=0 P1: R1 :=a; { R1 = 5 }
t=1 P1: R1= R1+1; { R1 = 6 }
t=2 P2: R2 :=a; {R2 = 5 }
t=3 P2: R2= R2 -1; { R2 = 4}
t=4 P1: a= R1; {R1 = 6}
t=5 P2: a= R2; {R2 = 4}

Sekcja krytyczna

- **Sekcja krytyczna**
- segment kodu w którym można aktualizować wspólne dane.
- tylko jeden proces jednocześnie może przebywać w swojej sekcji krytycznej
- n procesów w systemie { P0 P1 ...Pn-1 }
- każdy proces ma segment kodu zwany sekcją krytyczną (ang. *critical section*}
- jednocześnie tylko jeden proces może wykonywać swoją sekcję krytyczną -> wykonanie sekcji krytycznej podlega wzajemnego wykluczaniu (ang. mutual exclusion) w czasie
- konieczność zdefiniowania protokołu określającego współpracę między procesami

Rodzaje kodu

Wyróżnia się następujące fragmenty kodu:

sekcję krytyczną (ang. critical section)
sekcję wejściową (ang. entry section)
sekcję wyjściową (ang. exit section)
resztę (ang. exit section)

```
repeat
    sekcja wejściowa
    sekcja krytyczna
    sekcja wyjściowa
    reszta
until false
```

Problem sekcji krytycznej

Warunki, jakie musi spełniać rozwiązanie problemu sekcji krytycznej:

1. Wzajemne wykluczanie:

Jeżeli proces Pi działa w swej sekcji krytycznej, to żaden inny proces nie działa w sekcji krytycznej.

2. Postęp:

Jeżeli żaden proces nie działa w sekcji krytycznej i istnieją procesy, które chcą wejść do sekcji krytycznych, to tylko procesy nie wykonujące swoich reszt mogą kandydować jako następne do wejścia do sekcji krytycznych i wybór ten nie może być odwlekany w nieskończoność.

3. Ograniczone czekanie:

Musi istnieć wartość graniczna liczby wejść innych procesów do ich sekcji krytycznych po tym, gdy dany proces zgłosił chęć wejścia do swojej sekcji krytycznej i zanim uzyskał na to pozwolenie.

Błędne rozwiązania problemu sekcji krytycznej: Algorytm 1.

- **dzielona zmienna całkowita *numer***

```
Proces Pi

repeat
    while numer <> i do nic;
    sekcja krytyczna;
    numer := j;
    reszta;
until false;
```

Rozwiązanie spełnia warunek wyłączania.

Rozwiązanie nie spełnia warunku postępu,

Narzuca naprzemienne wywoływanie sekcji krytycznych.

Błędne rozwiązania problemu sekcji krytycznej: Algorytm 2.

- **tablica pamiętająca stany**

```
var flaga:  array[0..1] of boolean  {  ustawione
pocz.  na false }
{ flaga[i] =  true oznacza, że proces Pi  jest
  gotowy  do wejścia do sekcji krytycznej }
repeat
      flaga[i] := true;
      while flaga[j] do  nic;
            sekcja krytyczna;
      flaga[i] :=  false;
      reszta;
until false
```

Rozwiązanie spełnia warunek wyłączania.
Rozwiązanie nie spełnia warunku Postępu,

```
t0:  P0   flaga[0] = true;
t1:  P1   flaga[1] = true;
```

Algorytm rozwiązujący problem sekcji krytycznej (2 procesy)

```
Procesy: P0  i P1
j=1-i

Proces i:
var znacznik: array[0..1] of boolean; {określą, że dany
       proces jest gotowy  do wejścia sekcji krytycznej}
numer: 0..1; {określa, który proces ma prawo do  wejścia
       do sekcji krytycznej}
znacznik[0]:= false;
znacznik[1]:= false;
numer:= dowolna;
repeat
           znacznik[i]:=true;
           numer:=j;
           while(znacznik[j] and numer=j) do nic;
           sekcja krytyczna
           znacznik[i]:=false;
        reszta
until false;
```

Algorytm rozwiązujący problem sekcji krytycznej (2 procesy) - Dowód poprawności:

1. Proces Pi wchodzi do sekcji krytycznej, gdy :

znacznik[j]=false lub numer=i

Proces Pj wchodzi do sekcji krytycznej, gdy :

znacznik[i]=false lub numer=j

Aby oba procesy mogły działać jednocześnie w sekcjach krytycznych, konieczne jest, by :

znacznik[i]=znacznik[j]=true i (numer=i oraz numer=j)

a *numer* może przyjąć jedną z wartości **i** lub **j => sprzeczność**

procesy Pi i Pj nie mogą jednocześnie wykonywać pętli **while**

2 i 3. Proces Pi można powstrzymać przed wejściem do sekcji krytycznej tylko wtedy, gdy utknie w pętli while z warunkiem: *znacznik[j]=true* i *numer =j*

- Jeżeli Pj nie jest gotowy do wejścia do sekcji krytycznej, to znacznik[j]=false i do swojej sekcji może wejść proces Pi
- Jeżeli Pj jest gotowy do wejścia (tzn. znacznik[j]=true) i zaczyna wykonywać pętlę **while** to mogą być dwie sytuacje:
  - numer=i (wtedy do sekcji krytycznej wejdzie proces Pi)
  - lub numer=j (wtedy do sekcji krytycznej wejdzie proces Pj)

Proces wychodząc z sekcji krytycznej umożliwia drugiemu procesowi wejście do sekcji krytycznej

Proces oczekujący na **wejście wejdzie do sekcji krytycznej** (2-postęp) po **najwyżej jednym przejściu drugiego procesu** (3-warunek ograniczonego czekania).

Algorytm rozwiązujący problem sekcji krytycznej: rozwiązanie wieloprocesowe (tzw. algorytm piekarni, ang. bakery algorithm)

Zasady:

- przy wejściu do sklepu klient dostaje numerek
- obsługę rozpoczyna się od klienta z najmniejszym numerem
- algorytm nie gwarantuje, że dwa procesy (klienci) nie dostaną tego samego numeru
- w przypadku kolizji pierwszy zostanie obsłużony proces o wcześniejszej nazwie

Notacja:

- $(a,b) < (c,d)$, jeśli $a<c$ lub jeśli $a==c$ i $b<d$;
- $\max(a_0...,a_{n-1})$ jest taką liczbą k, że $k \geq a_i$ dla $i=0,...,n-1$

Algorytm piekarni - realizacja

```
{wspólne struktury danych}
var wybrane: array[0..n-1] of boolean;
        numer: array[0..n-1] of integer;
        for i:=0 to n-1 do wybrane[i]:= false;
        for i:=0 to n-1 do numer[i]=0;
repeat
        wybrane[i]:= true;
        numer[i]:=max(numer[0],numer[1],...,numer[n-1])+1;
        wybrane[i]:=false;
        for j:=0 to n-1
                do begin
                while wybrane[j] do nic;
                while numer[j] ≠ 0
                        and (numer[j],j) < (numer[i],i) do nic
                end
        sekcja krytyczna
        numer[i]:=0
        reszta
until false
```

Narzędzia do synchronizacji: Semafory.

- semafor (ang. semaphore) – narzędzie synchronizacji
- typ semafora: semafor zliczający (ang. counting semaphore)
- semafor – zmienna całkowita, na której mogą być wykonywane 3 operacje:
  - inicjalizacja (nadanie jakiejś wielkości całkowitej)
  - **P – czekaj** (ang. *wait,* z holend. *proberen*)
  - **V – sygnalizuj** (ang. *signal*, z holend. *verhogen*)

czekaj(S):

```
while S ≤ 0 do nic;
S:= S-1;
```

sygnalizuj(S):

```
S:=S+1;
```

- **Założenie realizacyjne:**
  - **operacje *czekaj* i *sygnalizuj* są niepodzielne !**

Przykłady użycia semaforów (1)

Zastosowanie semaforów do rozwiązania problemu sekcji krytycznej.

- wspólny semafor *mutex* (ang. mutual exclusion - wzajemne wykluczenie) jest dzielony przez n procesów
- wartość początkowa: *mutex*=1

- Proces:

```
repeat
        czekaj(mutex)
        sekcja krytyczna
        sygnalizuj(mutex)
until false
```

Przykłady użycia semaforów (2)

Dwa współbieżnie wykonywane procesy:

- **P1** z instrukcją **S1** i **P2** z instrukcją **S2**
- **S2** ma być wykonana **po** instrukcji **S1**

```
semafor synch,
inicjowanie: synch=0;

P1
  S1;
  sygnalizuj(synch);
P2
  czekaj(synch);
  S2;
```

Semafory: Implementacja

- wada przedstawionej definicji semafora (i przedstawionego rozwiązania problemu wzajemnego wykluczania) : aktywne czekanie (ang. *busy waiting*) - procesy usiłujące wejść do sekcji krytycznej muszą wykonywać instrukcje pętli marnując cykle procesora.

- inna nazwa takiego semafora: wirująca blokada (ang. *spinlock*) - oczekujący z powodu zamkniętego semafora proces *„wiruje”* w miejscu

- takie rozwiązanie może być użyteczne w systemach wieloprocesorowych, zaletą jest brak konieczności przełączania kontekstu

Semafory: Rozwiązanie bez stosowania aktywnego czekania

```
type semaphore = record
   wartość:integer;
   L: list of proces; {gdy proces musi czekać pod
                       semaforem, to jest dołączany do listy}
end

czekaj(S): S.wartość:=S.wartość-1;
         if S.wartość<0
         then begin
              dołącz dany proces do S.L;
              blokuj;
         end

sygnalizuj(S): S.wartość:=S.wartość+1;
               if S.wartość ≤ 0
                      then begin
                        usuń jakiś proces P z S.L;
                        obudź(P);
                      end
```

Semafory: Rozwiązanie bez stosowania aktywnego czekania 2

- **Uwaga:** w tej realizacji wartość semafora może być ujemna, jej wartość bezwzględna oznacza wtedy liczbę procesów czekających na semafor

- **Realizacja listy czekających procesów:**
  - np. przez dodanie pola dowiązania do bloku kontrolnego każdego procesu

Semafory: Semafor binarny

- Semafor binarny: może przyjmować dwie wartości: 0 lub 1

- **Implementacja semafora zliczającego S przy użyciusemaforów binarnych**

```
var   S1: semafor-binarny;
   S2: semafor-binarny;
   C:integer;
   Inicjacja: S1=1; S2=0;
          C= wartość początkowa semafora zliczającego S;
```

czekaj(S):

```
czekaj(S1);
C:=C-1;
if C<0  then begin
        sygnalizuj(S1);
              czekaj(S2);
        end
sygnalizuj(S1);
```

sygnalizuj(S):

```
czekaj(S1);
C:=C+1;
if C≤0
        then sygnalizuj(S2)
sygnalizuj(S1);
```

Synchronizacja – Monitory

- monitor
  - składnik programu złożony z deklaracji zmiennych wspólnych, dzielonych przez kooperujące procesy, oraz zbioru wszystkich procedur działających na zmiennych
  - monitor – kolekcja wszystkich regionów krytycznych związanych z tą samą zmienną dzieloną, zebranych w jednym module
- może istnieć możliwość definiowania typów monitorowych – schematów działania monitora i potem utworzenia wielu monitorów przez powołanie wielu zmiennych typu monitorowego

```
var identyfikator-monitora: monitor
  deklaracje-zmiennych;
  procedury;
  lista-udostępnionych-na-zewnątrz-nazw-
  procedur;
end.
```

Synchronizacja – Monitory (2)

- na zmiennych monitora mogą być wykonywane operacje wyłącznie w procedurach monitora
- procedury monitora mogą być wywoływane z zewnątrz (innych procesów, innych monitorów)
- monitor zapewnia wykluczające się wykonywanie swoich procedur – w tym samym czasie może może być wykonywana przez dowolny proces co najwyżej jedna procedura monitora -> uzyskujemy wzajemne wykluczenie na zmiennych monitora
- Wywołanie procedury monitora:

```
identyfikator-monitora.nazwa-procedury(parametry)
```

Monitory – warunkowe opuszczania monitorów i komunikacja między procesami

Zastosowanie operacji kolejkowych monitora:

var kol: queue (* czasem – condition zamiast queue*)

- **delay(kol) (czasem - wait** zamiast **delay) –** zawieszenie wywołującego procedurę procesu w kolejce *kol* z jednoczesnym opuszczeniem monitora i umożliwieniem wejścia do niego innemu procesowi
- **continue (kol) (czasem – signal** zamiast **continue) –** jeżeli w kolejce *kol* istnieje co najmniej jeden proces zawieszony w wyniku operacji *delay*, to dokładnie jeden z nich zostaje wyjęty i przeniesiony do kolejki wejściowej z atrybutem pierwszeństwa. Przydzielenie temu procesowi monitora powoduje jego uaktywnienie od instrukcji następnej po instrukcji wywołania procedury *delay*, która spowodowała jego zawieszenie
- **clear(kol) –** procedura przywracająca pierwotny stan kolejki – pusty (przy powstawaniu monitora kolejki są zawsze puste)
- **empty (kol) –** funkcja sprawdzająca, czy kolejka jest pusta (wartość true) czy nie (wartość false)

Rodzaje monitorów

- Monitory z sygnalizacją (C. Hoar)
  - Funkcja *signal/continue* wywołana przez proces w monitorze wznawia wykonanie procesu zawieszonego funkcją delay/wait, proces wywołujący sygnał opuszcza monitor (kończąc wykonanie funkcji monitora) lub zostaje zablokowany
  - Wady:
    - gdy proces wykonujący *signal* nie kończy swego działania z monitorem, wówczas potrzebne są dwa dodatkowe przełączenia stanu procesu – jego zawieszanie i wznowienie
    - Szeregowanie procesów związanych z sygnałem musi być niezawodne
- Monitory z powiadamianiem i rozgłaszaniem
  - powiadamianie: procedura *signal* zastąpiona przez *notify,* informuje odpowiednią kolejkę o zajściu warunku, proces z kolejki zostaje wznowiony nie natychmiast, ale wtedy, gdy będzie taka możliwość (wolny monitor)
  - rozgłaszanie (*broadcast*): umożliwia uruchomienie wszystkich procesów oczekujących w kolejce związanej z danym warunkiem, gdy będą ku temu możliwości

Inne mechanizmy synchronizacyjne

Metoda synchronizacyjna	Działanie
Muteks (Mutex)	Podobny do semafora binarnego. Podstawowa różnica: proces, który ustawił muteks (ustawił wartość na 0) musi być także tym, który go zdejmie (ustawi wartość na 1)
Zmienna warunkowa (Condition Variable)	Używany do blokowania procesu lub wątku, do czasu spełnienia danego warunku.
Flagi zdarzen (Event Flags)	Zdanie w pamięci używane jako mechanizm synchronizacyjny. Kod aplikacji może skojarzyć różne zdarzenia z kolejnymi bitami flagi. Wątek może czekać na jedno zdarzenie lub na kombinację zdarzeń sprawdzając jedne lub wiele bitów odpowiednich flag. Wątek jest zablokowany do czasu aż wszystkie wskazane bity są ustawione (AND) lub przynajmniej jeden z bitów jest ustawiony (OR).
Skrzynka pocztowa /Komunikat (Mailboxes/Messages)	Środek do wymiany informacji między dwoma procesami, które mogą być użyte do synchronizacji.

Problemy synchronizacji: Problem producentów i konsumentów

Dwa rodzaje procesów.

- **Producenci:** wykonują wewnętrzną procedurę Produkuj – tworzą elementy danych, które potem muszą być wysłane do konsumentów
- **Konsumenci:** po otrzymaniu elementu danych wykonują pewne obliczenia w wewnętrznej procedurze Konsumuj

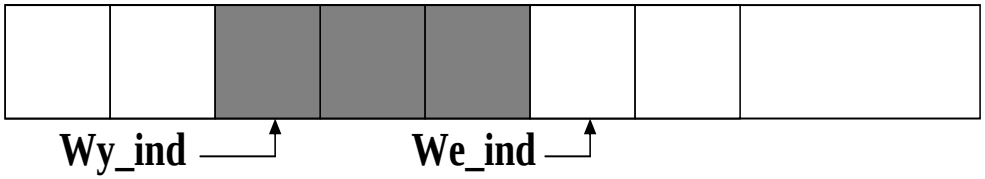
Przykłady:

- proces produkujący zestawienia, które konsumuje proces obsługi drukarki
- proces obsługi klawiatury, który produkuje wiersz tekstu wykorzystywany przez proces użytkownika

- rozpatrywany będzie 1 proces producenta i 1 konsumenta
- dwa rodzaje komunikacji:
  - synchroniczna
  - i asynchroniczna (za pośrednictwem bufora)

Producent- konsument – algorytm z nieskończonym buforem

- rozwiązanie wykorzystujące semafor reprezentujący liczbę elementów w buforze
- B: array(0..nieskończoność) of Integer;
- We_ind, Wy_ind: Integer:=0;
- Elementy: Semefor:=0;




```
task body Producent is
        I: Integer;
begin
        loop
            Produkuj(I);
            B(We_ind):=I;
            We_ind:=We_ind+1;
           Sygnalizuj(Elementy)
        end loop
end Producent
```

```
task body Konsument is
        I: Integer;
begin
      loop
           Czekaj(Elementy);
            I:=B(Wy_ind);
            Wy_ind:=Wy_ind+1;
            Konsumuj(I);
       end loop
end Konsument
```

Producent- konsument – algorytm ze skończonym buforem

```
B: array(0..N-1) of Integer;
We_ind, Wy_ind: Integer:=0;
Elementy: Semafor:=0;
Miejsca: Semafor:=N;
```

```
task body Producent is
        I: Integer;
begin
   loop
        Produkuj(I);
        Czekaj(Miejsca);
        B(We_ind):=I;
        We_ind:=(We_ind+1)mod N;
        Sygnalizuj(Elementy);
  end loop;
end Producent;
```

```
task body Konsument is
        I: Integer;
begin
  loop
        Czekaj(Elementy);
        I:=B(Wy_ind);
        Wy_ind:=(Wy_ind+1)mod N;
        Sygnalizuj(Miejsca);
        Konsumuj(I);
  end loop;
end Konsument
```

Problemy synchronizacji: Problem czytelników i pisarzy

- **Czytelnicy -** procesy nie wykluczają się nawzajem
- **Pisarze -** procesy wykluczają każdy inny proces (zarówno czytelnika jak i pisarza)

```
task body Czytelnik is
begin
   loop
        Zacznij_czytanie;
        Czytaj_dane;
        Zakończ_czytanie;
   end loop;
end Czytelnik

task body Pisarz is
begin
   loop
        Zacznij_pisanie;
        Zapisz_dane;
        Zakończ_pisanie;
   end loop;
end Pisarz
```

Problemy synchronizacji: Problem czytelników i pisarzy 2

Warunki synchronizacji:

- jeśli są oczekujący pisarze, to nowy czytelnik musi zaczekać (co najmniej) do zakończenia pisania przez pierwszego pisarza,
- jeżeli są oczekujący czytelnicy, to (wszyscy) będą wznawiani przed następnym pisaniem,

Problemy synchronizacji: Problem czytelników i pisarzy 3

```
monitor Monitor_czytelnicy_pisarze is
  Czytelnicy: Integer:=0;{Licznik czytelników, którzy czytają}
  Pisanie: Boolean:=False; {Czy jakiś proces pisze}
  Można_czytać, można_pisać: Warunek; {wstrzym. czyt. i pisarzy}

procedure Zacznij_czytanie is
begin
   if Pisanie or Niepusta(Można_pisać) then
        Czekaj(Można_czytać);
   end if
   Czytelnicy:=Czytelnicy+1;
   Sygnalizuj(Można_czytać);{Kaskada wznowień}
end Zacznij_czytanie

procedure Zakończ_czytanie is
begin
   Czytelnicy:=Czytelnicy-1;
   if Czytelnicy=0 then Sygnalizuj(Można_pisać);end if;
end Zakończ_czytanie
...
```

Problemy synchronizacji: Problem czytelników i pisarzy 4

```
…
procedure Zacznij_pisanie is
begin
   if Czytelnicy != 0 or Pisanie then
              Czekaj(Można_pisać);
       end_if
    Pisanie:=true;
end Zacznij_pisanie

procedure Koniec_pisania is
begin
   Pisanie:=False;
   if Niepusta(Można_czytać) then
        Sygnalizuj(Można_czytać);
   else
        Sygnalizuj(Można_pisać)
   end if;
end Koniec_pisania;

end Monitor_czytelnicy_pisarze
```

Rozwiązanie problemu czytelników i pisarzy używające semaforów. Czytelnicy mają wyższy priorytet (1)

```
int readcount;
semaphore x=1 /*for updating readcount*/, wsem =1 /*for mutual exclusion of access*/;
void reader()
{
	while(true){
		semWait(x);
		readcount++;
		if(readcount == 1)
			semWait(wsem);
		semSignal(x);
		READUNIT();
		semWait(x);
		readcount--;
		if(readcount==0)
			semSignal(wsem);
		semSignal(x);
	}
}
```

Rozwiązanie problemu czytelników i pisarzy używające semaforów. Czytelnicy mają wyższy priorytet (2)

```
void writer()
{
      while(true) {
                              semWait(wsem);
                              WRITEUNIT();
                              semSignal(wsem);
                }
}

void main()
{
      readcount =0;
      parbegin(reader,writer,…);
}
```

Rozwiązanie problemu czytelników i pisarzy używające semaforów. Pisarze mają wyższy priorytet (1)

- Nowi czytelnicy nie są dopuszczani, gdy przynajmniej jeden pisarz zadeklarował chęć zapisu
- Potrzebny dodatkowy semafore dla czytelników: tylko jeden czytelnik może czekać na semaforze rsem, pozostali czytelnicy czekają na semaforze z, przed czekaniem na rsem

Tylko czytelnicy w systemie	ustawiony *wsem* brak kolejek czekających
Tylko pisarze w systemie	ustawione *wsem* i *rsem* pisarze czekają na *wsem*
Zarówno czytelnicy jak i pisarze, czytanie najpierw	*wsem* ustawiony przez czytelnika *rsem* ustawiony przez pisarza wszyscy pisarze czekają na *wsem* jeden czytelnik czeka na *rsem* pozostali czytelnicy czekają na *z*
Zarówno czytelnicy jak i pisarze, pisanie najpierw	wsem ustawiony prez pisarza rsem ustawiony przez pisarza pisarze czekają na wsem jeden czytelnik czeka na rsem pozostali czytelnicy czekają na *z*

Rozwiązanie problemu czytelników i pisarzy używające semaforów. Pisarze mają wyższy priorytet (2)

```
int readercount, writercount;
semaphore x = 1, y = 1, z = 1, wsem  = 1, rsem = 1;

void reader()
{
	while(true){
		semWait(z);
		semWait(rsem)
		semWait(x)
		readcount++;
		if(readcount == 1)
			semWait(wsem);
		semSignal(x);
		semSignal(rsem);
		semSignal(z);
		READUNIT();
		semWait(x);
		readcount(x);
		if(readcount == 0) semSignal(wsem);
		semSignal(x);
	}
}
```

Rozwiązanie problemu czytelników i pisarzy używające semaforów. Pisarze mają wyższy priorytet (3)

```
void writer()
    {
        while(true){
                semWait(y);
                writecount++;
                if(writecount == 1)
                        semWait(rsem);
                semSignal(y);
                semWait(wsem);
                WRITEUNIT();
                semSignal(wsem);
                semWait(y);
                writecount--;
                if(writecount == 0) semSignal(rsem);
                semSignal(y);
        }
    }
```

Problemy synchronizacji: Problem 5 filozofów

- 5 filozofów przy stole, filozof myśli lub je,
- 5 widelców między filozofami,
- aby jeść filozof potrzebuje 2 widelców,

```
task body Filozof is
   begin
     loop
           Myśl;
           Protokół_wstępny;
           Jedz;
           Protokół_końcowy;
     end loop
   end Filozof;
```

Własności poprawnego rozwiązania:

- Filozof je tylko wtedy, gdy ma 2 widelce
- Dwu filozofów nie może jednocześnie trzymać tego samego widelca
- Nie występuje blokada
- Nikt nie może być zagłodzony
- Efektywne zachowanie przy braku współzawodnictwa

Problem 5 filozofów: Przykład rozwiązania

Przykład rozwiązania: Rozwiązanie z ograniczeniem liczby filozofów przebywających w jadalni.

```
Jadalnia: Semafor:=4;
Widelec: array(0..4) of Semafor;
task body Filozof is
begin
   loop
       Myśl;
       Czekaj(Jadalnia);
       Czekaj(Widelec(I));
       Czekaj(Widelec((I+1) mod 5));
       Jedz;
       Sygnalizuj(Widelec(I));
       Sygnalizuj(Widelec((I+1) mod 5));
       Sygnalizuj(Jadalnia);
     end loop;
end Filozof;
```

Problem 5 filozofów: Przykład rozwiązania (2) 1/2

```
#define N  5              /* ilość filozofów  */
#define LEFT  (i+N-1)%N   /* numer lewego sąsiada filozofa i */
#define RIGHT  (i+1)%N    /* numer prawego sąsiada filozofa i*/
#define THINKING 0        /* filozof myśli */
#define HUNGRY 1          /* filozof jest głodny */
#define EATING 2          /* filozof je*/
typedef int semaphore;    /* semafor to specjalny rodzaj liczby całkowitej*/
int state[N];             /* tablica z informacjami o stanach filozofów */
semaphore mutex=1;        /* wzajemne wykluczenie dla regionów krytycznych*/
semaphore s[N];           /* po jednym semaforze dla każdego filozofa*/


void philosopher(int i) {  /* i: numer filozofa, i=1…N-1*/
   while(TRUE)           /*powtarzaj w nieskończoność*/
   {
        think();         /* filozof myśli */
        take_forks(i); /* filozof pozyskuje dwa widelce lub się blokuje*/
        eat();           /* filozof je*/
        put_forks(i);  /* filozof odkłada dwa widelce*/
   }
}
```

Problem 5 filozofów: Przykład rozwiązania (2) 2/2

```
void take_forks(int i){/* numer filozofa, i=1…N-1 */
   down(&mutex);        /* wejście do sekcji krytycznej */
   state[i]=HUNGRY;     /* zaznaczenie, że filozof i jest głodny*/
   test[i];             /* próba pozyskania dwóch widelców*/
   up(&mutex);          /* wyjście z sekcji krytycznej*/
   down(&s[i]);       /* zablokowanie, jeśli widelce nie zostały pozyskane*/
}

void put_forks(int i) { /* numer filozofa, i=1…N-1 */
   down(&mutex);         /* wejście do sekcji krytycznej */
   state[i] = THINKING; /* filozof skończył jeść*/
   test(LEFT);           /* sprawdzenie, czy lewy sąsiad może teraz jeść */
   test(RIGHT);          /* sprawdzenie, czy prawy sąsiad może teraz jeść*/
   up(&mutex);           /* wyjście z sekcji krytycznej*/
}

void  test(i) {
   if(state[i] == HUNGRY && state[LEFT]!=EATING && state[RIGHT]!=EATING)
   {
        state[i] = EATING;
        up(&s[i]);
   }
}
```