SOP2 - semafory

Plan prezentacji

- Problem producent-konsument
- Problem czytelników i pisarzy
- Problem jedzących filozofów

Producent-konsument

var

```
bufor: array [0..n-1] of produkt;
pierwszy: 0..n-1;
licznik: 0..n;
```

Producent-konsument

PRODUCENT:

```
repeat
    wyprodukuj jednostkę produktu;
    P(puste);
    P(S);
    bufor [(pierwszy+licznik) mod n] :=
      wyprodukowana jednostka;
    licznik := licznik + 1;
    V(S);
    V(pełne)
until false
```

Producent-konsument

KONSUMENT:

```
repeat
    P(pełne);
    P(S);
    pobrana jednostka := bufor [pierwszy];
    pierwszy := (pierwszy + 1) mod n;
    licznik := licznik - 1;
    V(S);
    V(puste);
    skonsumuj pobraną jednostkę
until false
```

Problem czytelników i pisarzy [1]

- **Opis problemu – wersja podstawowa:**
  - Jest n czytelników, m pisarzy i jedna wspólna strona.
  - Każdy czytelnik czyta informacje ze strony, każdy pisarz może pisać na stronie;
  - Wielu czytelników może naraz czytać dane ze strony;
  - Pisarz zajmuje stronę na wyłączność (żaden inny proces – pisarz czy czytelnik nie może używać w tym czasie strony);
  - Nie ma ograniczeń czasowych na czytanie i pisanie, ale operacje te są skończone;

Problem czytelników i pisarzy [2]

- Rozwiązanie:

  Pisarz:

  Robi coś;

  Chce pisać;

  Pisze;

  Informuje, że skończył pisać;

  Czytelnik:

  Robi coś;

  Chce czytać;

  Czyta;

  Informuje, że skończył czytać;

Problem czytelników i pisarzy [3] - założenia

```
int no_of_r = 0;

/* liczba procesów - czytelników
aktualnie czytających*/

Boolean semaphore sp, w = 1, 1 ;

/* w służy do synchronizacji
dostępu do no_of_r,
sp służy do synchronizacji dostępu
do strony */
```

Problem czytelników i pisarzy [3] - czytelnik

```
process :: reader;
void main()
{
  while(1)
  {
    P(w);
    no_of_r ++;
    if (no_of_ r == 1)
      P(sp);
    /* pierwszy czytający proces opuszcza sp */
    V(w);
    /* czyta ze strony*/
    P(w);
    no_of_r --;
    if (no_of_r == 0)
      V(sp);
    /* ostatni czytający proces podnosi semafor sp */
    V(w);
  };
};
```

Problem czytelników i pisarzy [3] - pisarz

```
process :: writer;
void main()
{
  while(1)
  {
    P(sp);
    /* każdy piszący proces musi opuścić
semafor sp*/
    /* pisze na stronie */
    V(sp);
  };
};
```

Problem czytelników i pisarzy [4] – priorytet pisarzy

- **Problem czytelników i pisarzy z priorytetem dla pisarzy.**
  - Dodajemy dwa dodatkowe warunki do zadania:
  - Pisarze mają priorytet, czyli, jeśli jakikolwiek pisarz chce pisać, żaden czytelnik nie może zacząć czytać.
  - Nie ma priorytetu pomiędzy czytelnikami.

Problem czytelników i pisarzy [5] – założenia

```
int no_of_r,no_of_w = 0,0;
/*
  no_of_r - liczba aktualnie czytających procesów
  no_of_w - liczba procesów, które chcą pisać
*/
Boolean semaphore sp,sr = 1, 1;
Boolean semaphore w1, w2, w3= 1, 1, 1;
/*
  w1 - do dostępu do no_of_r
  w2 - do dostępu do no_of_w
  w3 - dodatkowe „drzwi” dla czytelników
  sp - dostęp do czytanej strony
  sr - dla priorytetu pisarzy
*/
```

Problem czytelników i pisarzy [6] – czytelnik

```
process:: reader;
void main()
{
  while(1)
  {
    P(w3);
    P(sr);
    P(w1);
    no_of_r++;
    if (no_of_r == 1)
      P(sp);
    V(w1);
    V(sr);
    V(w3);
    /* czyta */
    P(w1);
    no_of_r--;
    if ( no_of_r == 0)
      V(sp);
    V(w1);
  };
};
```

```
process :: writer;
void main();
{
  while(1)
  {
    P(w2);
    no_of_w++;
    if (no_of_w == 1 )
      P(sr);
    V(w2);
    P(sp);
    /* pisze */
    V(sp);
    P(w2);
    no_of_w--;
    if (no_of_w==0)
      V(sr);
    V(w2);
  };
};
```

Semafory [2]

- **Uogólnione operacje na semaforach**
  - Wartość semafora jest zmieniana o wartość całkowitą n.
  - P(s ,n);
    - Waiting until s >= n;
    - s=s-n;
  - V(s,n);
    - s = s+n;

Czytelnicy i pisarze [7]

- **Problem czytelników i pisarzy na semaforach uogólnionych**

```
semaphore w = M;
```

```
process :: reader
{
    while(1)
    {
        P(w,1);
        /*M może zwiększyć
        w o 1 */
        /* teraz czyta */
        V(w,1);
    };
};
```

```
process :: writer
{
    while(1)
    {
        P(w,M);
        /* tylko jeden proces może
         zmniejszyć w o M*/
        /* teraz pisze*/
        V(w,M);
    };
};
```

Czytelnicy i pisarze [8]

- **Problem czytelników i pisarzy na semaforach uogólnionych z priorytetem pisarzy**

```
semaphore sp, r = M,M ;
/*
M>= liczby możliwych czytelników
sp podobnie jak we wcześniejszym przykładzie
r - dla realizowania priorytetu pisarzy
*/
```

Czytelnicy i pisarze [9]

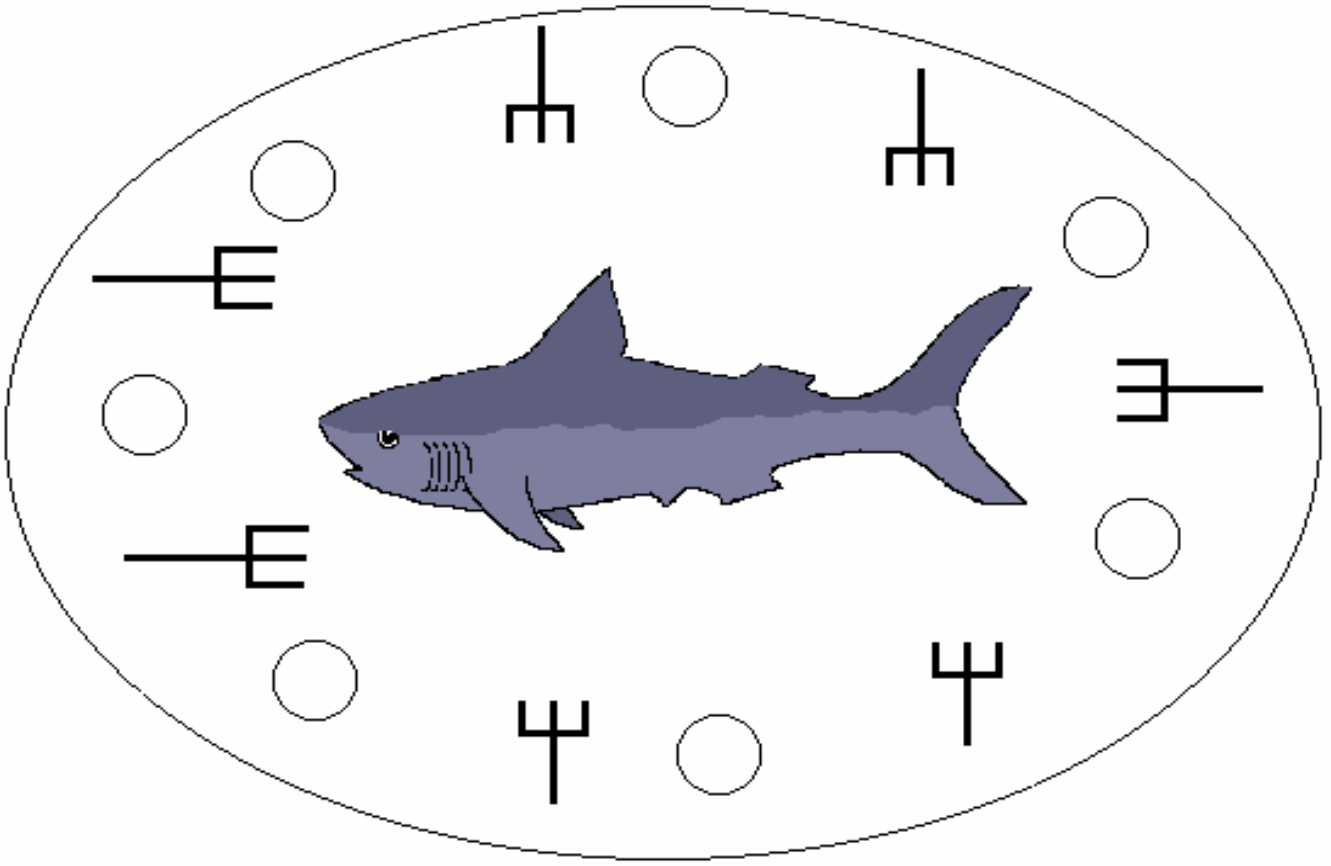
```
process :: reader;
void main()
{
    while(1)
    {
        P(r,M);
        P(sp,1);
        V(r,M-1);
        /*pisarze mogą
      zwiększyć semafor r o 1*/
        V(r,1);
        /*czytelnik czeka jeśli
         jakiś pisarz czeka*/
        /* teraz czyta */
        V(sp,1);
    };
};
```

```
process :: writer
void main()
{
    while(1)
    {
        P(r,1);
        P(sp,M);
        /* pisze */
        V(sp,M);
        V(r,1);
    };
};
```

Problem jedzących filozofów [1]

- **Problem jedzących filozofów – czasami nazywany problemem pięciu filozofów**
  - Grupa filozofów siedzi przy okrągłym stole
  - Każdy rozmyśla, aż zgłodnieje
  - Kiedy filozof jest głodny je
    - Bierze po jednym widelcu z każdej z jego stron
  - Jak skończy jeść kontynuuje rozmyślania
  - Na stole jest tylko 5 widelców, sytuację tą przedstawia rysunek (nast. Slajd)
  - Jak zsynchronizować działania filozofów by uniknąć zakleszczenie oraz zagłodzenia?

Problem jedzących filozofów [2]



Problem jedzących filozofów [3]

- **<u>Problem jedzących filozofów – czasami nazywany problemem pięciu filozofów</u>**
  - N filozofów siedzi dokoła okrągłego stołu, każdy na swoim miejscu. Jest n talerzy i n widelców na stole. Talerze znajdują się na wprost filozofów, widelce leżą pomiędzy talerzami.
  - Czynności filozofa:
    - Pętla:
      Myśli
      Chce podnieść swoje widelce
      Je posługując się widelcami
      Odkłada widelce

Problem jedzących filozofów [4]

- **Rozwiązanie 1.**

```
var widelec:array[0..4] of semaphore;
i:integer;

begin
  for i:=0 to 4 do widelec[i]:=1;
  parallel_begin
   filozof(0); filozof(1); filozof(2);
     filozof(3); filozof(4);
  parallel_end
end
```

Problem jedzących filozofów [5]

- **Rozwiązanie 1.**

```
procedure filozof(i:integer);
begin
  repeat
    myśl;
    P(widelec[i]);
    P(widelec[(i+1)mod 5]);
    jedz;
    V(widelec[i]);
    V(widelec[(i+1)mod 5]);
  forever
end;
```

Problem jedzących filozofów [6]

- **Rozwiązanie 1.**
  - Semafory zapewniają wzajemne wykluczanie przy dostępie do widelców
  - Zapewnione bezpieczeństwo (jedzenie po zakończeniu oczekiwania na 2 widelce-semafory)
  - Możliwość blokady jeśli filozofowie się zsynchronizują i równocześnie biorą lewe widelce

Problem jedzących filozofów [7]

- **Rozwiązanie 2. Ograniczenie liczby talerzy**
  - Ograniczenie talerzy do 4
  - Każdy widelec to semafor, początkowo równy 1
  - Wzięcie widelca = P
  - Odłożenie widelca = V
  - Talerze to semafor o początkowej wartości 4
  - Widelce ponumerowane od 0..4

Problem jedzących filozofów [8]

```
var
  paleczki: array [0..4] of semaphore;
  talerze: semaphore = 4;

  procedure filozof(i: 0..4);
  begin
    repeat
   myśl;
      P(talerz);
      P(paleczki[i]);
      P(paleczki[(i + 1) mod 5]);
      jedz;
      V(paleczki[i]);
      V(paleczki[(i + 1) mod 5]);
      V(talerz);
    until false
  end;
```

Problem jedzących filozofów [7]

- **Rozwiązanie 3. Hierarchia zasobów**
  - Częściowe uporządkowanie i hierarchia zasobów
  - Zasób = widelec
  - Widelce pobierane są w ustalonej kolejności – np. rosnącej
  - Widelce odkładane są w odwrotnej kolejności – malejącej

  - Brak zakleszczeń
  - Nie jest szczególnie wydajne

Problem jedzących filozofów [7]

- **Rozwiązanie 4. Ze stanami filozofów**

```
state[i] = 0 i-ty filozof myśli
state[i] = 1 i-ty filozof chce jeść
state[i] = 2 i-ty filozof je
resource fork [N];
Boolean semaphore filsem [N] = (N) 0;
/*każdy filozof ma swój własny semafor do czekania*/
int state [N] = (N) 0 ;
Boolean semaphore w = 1;
/* w do dostępu do tablicy stanów */

void test (int k);
{
  if ((state [(k-1)% N] <> 2 ) and ( state[k]== 1) and ( state[(k+1)%N] <>
     2 ))
  {
    state [k] = 2;
    V(filsem[k]);
  };
};
```

Problem jedzących filozofów [7]

```
process Philosopher (int name);
{
while(1)
{ /* myśli */
  /* chce jeść */
  P(w);
  state[name] = 1;
  test (name);
  V(w);
  P ( filsem[name] );
  /* filsem[name] moŜe zostać opuszczony, jeśli został podniesiony w procedurze test(name)
  */
  request (fork[name] , fork [(name+1) % N] ) ;
  /* je */
  release (fork[name] , fork [(name+1) % N]) ;
  P(w);
  state[name] = 0;
  test ((name-1) % N);
  test ((name+1) % N);
  /* sprawdzamy obydwu sąsiadów czy czekają */
  /* odpowiednie semafory są podnoszone */
  V(w);
  };
};
```