Systemy operacyjne

# Systemowe mechanizmy synchronizacji procesów

**Wykład prowadzą:**
**Jerzy Brzeziński**
**Dariusz Wawrzyniak**


Systemy operacyjne



**Plan wykładu**

- Definicja semafora
- Klasyfikacja semaforów
- Implementacja semaforów
- Zamki
- Zmienne warunkowe
- Klasyczne problemy synchronizacji



Systemy operacyjne



**Semafory**

- Semafor jest zmienną całkowitą nieujemną lub — w przypadku semaforów binarnych — zmienną typu logicznego.
- Na semaforze można wykonywać dwa rodzaje operacji:
  - P — opuszczanie semafora (hol. proberen)
  - V — podnoszenie semafora (hol. verhogen)
- Synchronizacja polega na blokowaniu procesu w operacji opuszczania semafora, jeśli semafor jest już opuszczony.

**Rodzaje semaforów (1)**

- Semafor binarny — zmienna semaforowa przyjmuje tylko wartości true (stan podniesienia, otwarcia) lub false (stan opuszczenia, zamknięcia).
- Semafor ogólny (zliczający) — zmienna semaforowa przyjmuje wartości całkowite nieujemne, a jej bieżąca wartość jest zmniejszana lub zwiększana o 1 w wyniku wykonania odpowiednio operacji opuszczenia lub podniesienia semafora.





**Rodzaje semaforów (2)**

- Semafor uogólniony — semafor zliczający, w przypadku którego zmienną semaforową można zwiększać lub zmniejszać o dowolną wartość, podaną jako argument operacji.
- Semafor dwustronnie ograniczony — semafor ogólny, w przypadku którego zmienna semaforowa, oprócz dolnego ograniczenia wartością 0, ma górne ograniczenie, podane przy definicji semafora.





**Implementacja semafora ogólnego na poziomie maszynowym**

- Opuszczanie semafora

```
procedure P(var s: Semaphore)
   begin
      while s = 0 do nic;   } instrukcje muszą być
      s := s - 1;           } wykonane atomowo
   end;
```

- Podnoszenie semafora

```
procedure V(var s: Semaphore)
   begin
      s := s + 1;           } instrukcja musi być
   end;                     } wykonana atomowo
```

Systemy operacyjne

**Implementacja semafora binarnego na poziomie maszynowym**

- Opuszczanie semafora

```
procedure P(var s: Binary_Semaphore)
   begin
      while not s do nic;   } instrukcje muszą być
      s := false;           } wykonane atomowo
   end;
```

- Podnoszenie semafora

```
procedure V(var s: Binary_Semaphore)
   begin
     s := true;             } instrukcja musi być
   end;                     } wykonana atomowo
```

Systemowe mechanizmy synchronizacji procesów (7)

Systemy operacyjne

**Implementacja semafora ogólnego na poziomie systemu operacyjnego (1)**

- Struktury danych

```
type Semaphore = record
   wartość: Integer;
   L: list of Proces;
end;
```

- Opuszczanie semafora

```
procedure P(var s: Semaphore) begin
   s.wartość := s.wartość - 1;
   if s.wartość < 0 then begin
      dołącz dany proces do s.L
      zmień stan danego procesu na „oczekujący”
   end;
end;
```

Systemowe mechanizmy synchronizacji procesów (8)

Systemy operacyjne

**Implementacja semafora ogólnego na poziomie systemu operacyjnego (2)**

- Podnoszenie semafora

```
procedure V(var s: Semaphore)
  begin
     s.wartość := s.wartość + 1;
     if s.wartość ≤ 0 then begin
        wybierz i usuń jakiś/kolejny proces z kolejki s.L
        zmień stan wybranego procesu na „gotowy”
     end;
 end;
```

Systemowe mechanizmy synchronizacji procesów (9)

**Wzajemne wykluczanie z użyciem semaforów**

```
shared mutex: Semaphore := 1;
P(mutex);        ← sekcja wejściowa
sekcja krytyczna;
V(mutex);        ← sekcja wyjściowa
reszta;
```





**Mechanizmy synchronizacji POSIX — zmienne synchronizujące**

- Rodzaje zmiennych synchronizujących:
  - zamek — umożliwiająca implementację wzajemnego wykluczania,
  - zmienna warunkowa — umożliwia usypianie i budzenie wątków.
- Zmienne synchronizujące muszą być współdzielone przez synchronizowane wątki.
- Zanim zmienna zostanie wykorzystana do synchronizacji musi zostać zainicjalizowana.





**Operacje na zmiennych synchronizujących**

- Zamek — umożliwia implementację wzajemnego wykluczania. Operacje:
  - lock — zajęcie (zaryglowanie) zamka
  - unlock — zwolnienie (odryglowanie) zamka
  - trylock — nieblokująca próba zajęcia zamka
- Zmienna warunkowa — umożliwia usypianie i budzenie wątków. Operacje:
  - wait — uśpienie wątku,
  - signal — obudzenie jednego z uśpionych wątków
  - broadcast — obudzenie wszystkich uśpionych wątków

**Zamek — interfejs**

- Typ: **`pthread_mutex_t`**
- Operacje:
  - **`pthread_mutex_lock(pthread_mutex_t *m)`** — zajęcie zamka
  - **`pthread_mutex_unlock(pthread_mutex_t *m)`** — zwolnienie zamka
  - **`pthread_mutex_trylock(pthread_mutex_t *m)`** — próba zajęcia zamka w sposób nie blokujący wątku w przypadku niepowodzenia







**Zamek — implementacja**

- **`pthread_mutex_lock`**
  - zajęcie zamka, jeśli jest wolny
  - ustawienie stanu wątku na oczekujący i umieszczenie w kolejce, jeśli zamek jest zajęty
- **`pthread_mutex_unlock`**
  - ustawienie zamka na wolny, jeśli kolejka oczekujących wątków jest pusta
  - wybranie wątku z niepustej kolejki wątków oczekujących i ustawienie jego stanu na gotowy.
- **`pthread_mutex_trylock`**
  - zajęcie zamka lub kontynuacja przetwarzania





**Zmienna warunkowa — interfejs**

- Typ: **`pthread_cond_t`**
- Operacje:
  - **`pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`** — uśpienie wątku na zmiennej warunkowej,
  - **`pthread_cond_signal(pthread_cond_t *c)`** — obudzenie jednego z wątków uśpionych na zmiennej warunkowej,
  - **`pthread_cond_broadcast(pthread_cond_t *c)`** — obudzenie wszystkich wątków uśpionych na zmiennej warunkowej.

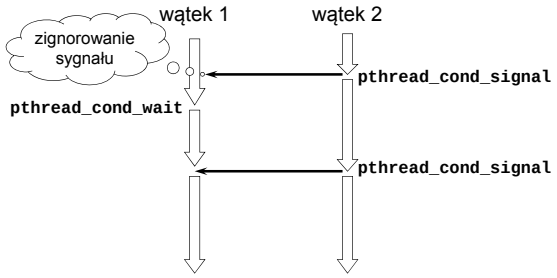
**Zmienna warunkowa — implementacja**

- **`pthread_cond_wait`**
  - ustawienie stanu wątku na oczekujący i umieszczenie go w kolejce
- **`pthread_cond_signal`**
  - wybranie jednego wątku z kolejki i postępowanie takie, jak przy zajęciu zamka
  - zignorowanie sygnału, jeśli kolejka jest pusta
- **`pthread_cond_broadcast`**
  - ustawienie wszystkich wątków oczekujących na zmiennej warunkowej w kolejce do zajęcia zamka, a jeśli zamek jest wolny zmiana stanu jednego z nich na gotowy.





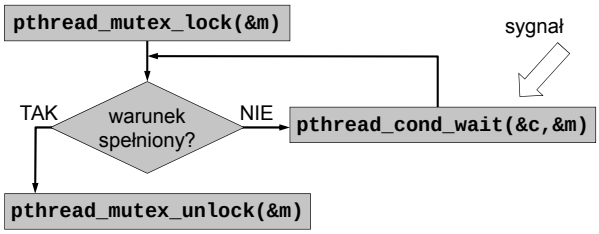
**Zasada funkcjonowania zmiennej warunkowej**



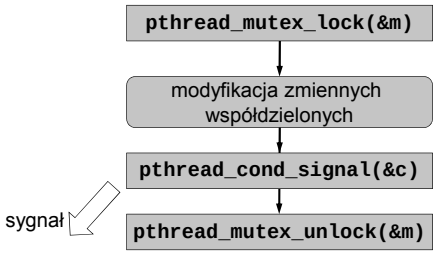





**Użycie zmiennych warunkowych (schemat 1) — wątek oczekujący**



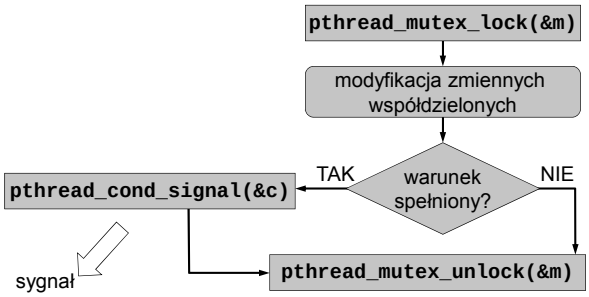
**Użycie zmiennych warunkowych (schemat 1) — wątek sygnalizujący**







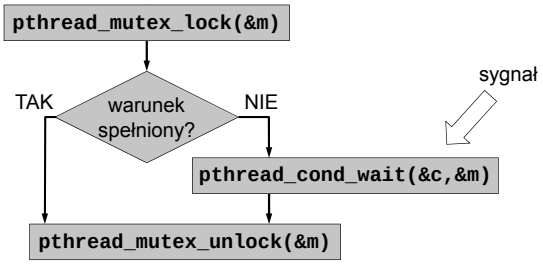

**Użycie zmiennych warunkowych (schemat 2) — wątek sygnalizujący**








**Użycie zmiennych warunkowych (schemat 2) — wątek oczekujący**



**Klasyczne problemy synchronizacji**

- Problem producenta i konsumenta — problem ograniczonego buforowania w komunikacji międzyprocesowej
- Problem czytelników i pisarzy — problem synchronizacji dostępu do zasobu w trybie współdzielonym i wyłącznym
- Problem pięciu filozofów — problem jednoczesnego dostępu do dwóch zasobów (ryzyko głodzenia i zakleszczenia)
- Problem śpiących fryzjerów — problem synchronizacji w interakcji klientserwer przy ograniczonym kolejkowaniu





**Problem producenta i konsumenta**

- Producent produkuje jednostki określonego produktu i umieszcza je w buforze o ograniczonym rozmiarze.
- Konsument pobiera jednostki produktu z bufora i konsumuje je.
- Z punktu widzenia producenta problem synchronizacji polega na tym, że nie może on umieścić kolejnej jednostki, jeśli bufor jest pełny.
- Z punktu widzenia konsumenta problem synchronizacji polega na tym, że nie powinien on mięć dostępu do bufora, jeśli nie ma tam żadnego elementu do pobrania.





**Synchronizacja producenta i konsumenta za pomocą semaforów ogólnych** (1)

- Dane współdzielone

```
const n: Integer := rozmiar bufora;
shared buf: array [0..n-1] of ElemT;
shared wolne:  Semaphore := n;
shared zajęte: Semaphore := 0;
```

**Synchronizacja producenta i konsumenta za pomocą semaforów ogólnych (2)**

- Producent

```
local i: Integer := 0;
local elem: ElemT;
while ... do begin
   produkuj(elem);
   P(wolne);
   buf[i] := elem;
   i := (i+1) mod n;
   V(zajęte);
end;
```





**Synchronizacja producenta i konsumenta za pomocą semaforów ogólnych (3)**

- Konsument

```
local i: Integer := 0;
local elem: ElemT;
while ... do begin
   P(zajęte);
   elem := buf[i];
   i := (i+1) mod n;
   V(wolne);
   konsumuj(elem);
end;
```





**Problem czytelników i pisarzy**

- Dwa rodzaje użytkowników — czytelnicy i pisarze — korzystają ze wspólnego zasobu — czytelni.
- Czytelnicy korzystają z czytelni w trybie współdzielonym, tzn. w czytelni może przebywać w tym samym czasie wielu czytelników.
- Pisarze korzystają z czytelni w trybie wyłącznym, tzn. w czasie, gdy w czytelni przebywa pisarz, nie może z niej korzystać inny użytkownik (ani czytelnik, ani inny pisarz).
- Synchronizacja polega na blokowaniu użytkowników przy wejściu do czytelni, gdy wymaga tego tryb dostępu.

**Synchronizacja czytelników i pisarzy za pomocą semaforów binarnych (1)**

- Dane współdzielone

```
shared l_czyt: Integer := 0;
shared mutex_r: Binary_Semaphore := true;
shared mutex_w: Binary_Semaphore := true;
```





**Synchronizacja czytelników i pisarzy za pomocą semaforów binarnych (2)**

- Czytelnik

```
while ... do begin
   P(mutex_r);
   l_czyt := l_czyt + 1;
   if l_czyt = 1 then P(mutex_w);
   V(mutex_r);
   czytanie;
   P(mutex_r);
   l_czyt := l_czyt - 1;
   if l_czyt = 0 then V(mutex_w);
   V(mutex_r);
end;
```





**Synchronizacja czytelników i pisarzy za pomocą semaforów binarnych (3)**

- Pisarz

```
while ... do
   P(mutex_w);
   pisanie;
   V(mutex_w);
end;
```

**Problem pięciu filozofów**

- Przy okrągłym stole siedzi pięciu filozofów, którzy na przemian myślą (filozofują) i jedzą makaron ze wspólnej miski.
- Żeby coś zjeść, filozof musi zdobyć dwa widelce, z których każdy współdzieli ze swoim sąsiadem.
- Widelec dostępny jest w trybie wyłącznym — może być używany w danej chwili przez jednego filozofa.
- Należy zsynchronizować filozofów tak, aby każdy mógł się w końcu najeść przy zachowaniu reguł dostępu do widelców oraz przy możliwie dużej przepustowości w spożywaniu posiłków.





**Synchronizacja 5 filozofów za pomocą semaforów binarnych (1)**

```
shared dopuść: Semaphore := 4;
shared widelec: array[0..4] of
                Binary_Semaphore := true;
```





**Synchronizacja 5 filozofów za pomocą semaforów binarnych (2)**

- Filozof nr *i* (*i* = 0 ... 4)

```
while ... do begin
   myślenie;
   P(dopuść);
   P(widelec[i]);
   P(widelec[(i+1) mod 5]);
   jedzenie;
   V(widelec[i]);
   V(widelec[(i+1) mod 5]);
   V(dopuść);
end;
```

**Problem śpiących fryzjerów**

- W salonie fryzjerskim jest poczekalnia z *p* miejscami oraz *n* foteli, obsługiwanych przez fryzjerów.
- Do salonu przychodzi klient, budzi fryzjera, po czym fryzjer znajduje wolny fotel i obsługuje klienta.
- Jeśli nie ma wolnego fotela, klient zajmuje jedno z wolnych miejsc w poczekalni.
- Jeśli nie ma miejsca w poczekalni, klient odchodzi.
- Problem polega na zsynchronizowaniu fryzjerów oraz klientów w taki sposób, aby jeden fryzjer w danej chwili obsługiwał jednego klienta i w tym samym czasie klient był obsługiwany przez jednego fryzjera.





**Synchronizacja śpiących fryzjerów za pomocą semaforów (1)**

- Dane współdzielone

```
const  p: Integer := pojemność poczekalni;
const  n: Integer := liczba foteli;
shared l_czek:  Integer := 0;
shared mutex:   Binary_Semaphore := true;
shared klient:  Semaphore := 0;
shared fryzjer: Semaphore := 0;
shared fotel:   Semaphore := n;
```





**Synchronizacja śpiących fryzjerów za pomocą semaforów (2)**

- Klient

```
while ... do begin
   P(mutex);
   if l_czek < p then begin
      l_czek := l_czek + 1;
      V(klient);
      V(mutex);
      P(fryzjer);
      strzyżenie;
   end
   else V(mutex);
end;
```

**Synchronizacja śpiących fryzjerów za pomocą semaforów (3)**

- Fryzjer

```
while ... do begin
   P(klient);
   P(fotel);
   P(mutex);
   l_czek := l_czek - 1;
   V(fryzjer);
   V(mutex);
   strzyżenie;
   V(fotel);
end;
```





**Monitory**

- Monitory jest strukturalnym mechanizmem synchronizacji, którego definicja monitora obejmuje:
  - hermetycznie zamkniętą definicję zamiennych (pól),
  - definicję wejść, czyli procedur umożliwiających wykonywanie operacji na polach monitora.
- Wewnątrz monitora może być aktywny co najwyżej jeden proces.
- Proces może zostać uśpiony wewnątrz monitora na zmiennej warunkowej.
- Uśpiony proces może zostać obudzony przez wysłanie sygnału związanego z daną zmienną warunkową.





**Ogólny schemat definicji monitora**

```
type nazwa_monitora = monitor
  deklaracje zmiennych
  procedure entry proc_1(...);
     begin ... end;
     ⋮
  procedure entry proc_n(...);
     begin ... end;
  begin
     kod inicjalizujący
  end.
```

Systemy operacyjne

**Ograniczony bufor cykliczny — definicja oparta na monitorze (1)**

```
type Buffer  =  monitor
   pula : array [0..n-1] of ElemT;
   wej, wyj, licz : Integer;
   pusty, pełny : Condition;
```



Systemy operacyjne

**Ograniczony bufor cykliczny — definicja oparta na monitorze (2)**

```
procedure entry wstaw(elem: ElemT);
   begin
      if licz = n then pełny.wait;
      pula[wej] := elem;
      wej := (wej + 1) mod n;
      licz := licz + 1;
      pusty.signal;
   end;
```



Systemy operacyjne

**Ograniczony bufor cykliczny — definicja oparta na monitorze (3)**

```
procedure entry pobierz(var elem: ElemT);
   begin
      if licz = 0 then pusty.wait;
      elem := pula[wyj];
      wyj := (wyj + 1) mod n;
      licz := licz - 1;
      pełny.signal;
   end;
```

**Ograniczony bufor cykliczny — definicja oparta na monitorze (4)**

```
begin
    wej := 0;
    wyj := 0;
    licz := 0;
end.
```





**Synchronizacja producenta i konsumenta za pomocą monitora**

- Dane współdzielone

```
shared buf: Buffer;
```

- Producent

```
local
   elem: ElemType;
while ... do begin
   produkuj(elem);
   buf.wstaw(elem);
end;
```

- Konsument

```
local
   elem: ElemType;
while ... do begin
   buf.pobierz(elem);
   konsumuj(elem);
end;
```





**Synchronizacja czytelników i pisarzy za pomocą monitora**

- Dane współdzielone

```
shared czytelnia: Monitor;
```

- Czytelnik

```
czytelnia.wejście_czytelnika;
czytanie;
czytelnia.wyjście_czytelnika;
```

- Pisarz

```
czytelnia.wejście_pisarza;
pisanie;
czytelnia.wyjście_pisarza;
```

**Regiony krytyczne**

- Region krytyczny jest fragmentem programu — oznaczonym jako *S*, wykonywanym przy wyłącznym dostępie do pewnej zmiennej współdzielonej, wskazanej w jego definicji — oznaczonej jako *v*.
- Wykonanie regionu krytycznego uzależnione jest od wyrażenia logicznego — *B*, a przetwarzanie blokowane jest do momentu, aż wyrażenie będzie prawdziwe.

```
shared v: T;
region v when B do S;
```





**Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (1)**

- Dane współdzielone

```
shared buf: record
     pula : array [0..n-1] of ElemT;
     wej, wyj, licz : Integer;
end;
```





**Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (2)**

- Producent

```
local elem: ElemT;
while ... do
   begin
      produkuj(elem);
      region buf when buf.licz < n do
         begin
            buf.pula[wej]:= elem;
            buf.wej := (buf.wej+1) mod n;
            buf.licz := buf.licz + 1;
         end;
   end;
```

**Synchronizacja producenta i konsumenta za pomocą regionu krytycznego (3)**

- Konsument

```
local elem: ElemT;
while ... do
   begin
      region buf when buf.licz > 0 do
         begin
            elem := buf.pula[wyj];
            buf.wyj := (buf.wyj+1) mod n;
            buf.licz := buf.licz - 1;
         end;
      konsumuj(elem);
   end;
```