

# Synchronizacja procesów i wątków

---

Sieciowe Systemy Operacyjne

# Zakres tematyczny

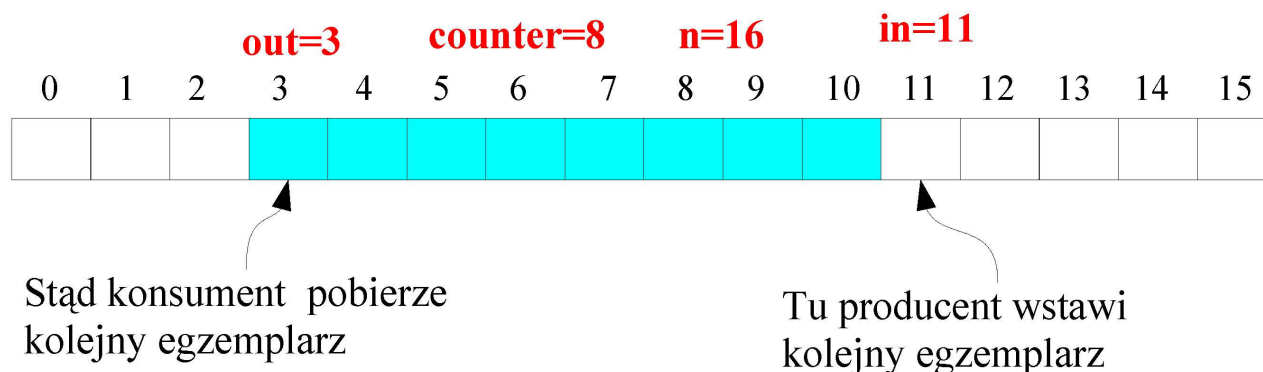
- Potrzeba synchronizacji
- Problem producent-konsument
- Producent-konsument z buforem cyklicznym
- Problem sekcji krytycznej
- Semafor zliczający
- Semafor i sekcja krytyczna
- Semafor binarny
- Zakleszczanie procesów
- Znane problemy algorytmiczne
- Monitory

# Potrzeba synchronizacji

- Procesy wykonują się współbieżnie.
- Jeżeli w 100% są izolowane od siebie, nie ma problemu.
- Problem, jeżeli procesy komunikują się lub korzystają ze wspólnych zasobów.
  - Przykład: Proces A przygotowuje wyniki, a proces B drukuje je na drukarce: Jak zapewnić, aby B nie zaczął drukować przed zakończeniem przygotowania wyników przez A.
- Potrzeba utrzymywania wspólnych zasobów w spójnym stanie.
  - Np. proces A dodaje element do listy (z dowiązaniami), a jednocześnie B przegląda listę, która w momencie trwania operacji dodania ma stan niespójny
- Potrzeba synchronizacji dotyczy także współbieżnych wątków.
  - W kolejnych slajdach będę używał – zgodnie z literaturą pojęcia “proces”, jednakże wszystkie przykłady oparte są na założeniu, że procesy wykonują się współbieżnie. Zatem bardziej odpowiednie byłoby użycie terminu wątki.
  - Np. dwa wątki wywołują funkcję malloc, która przydziela pamięć.

## Problem producenta-konsumenta (z ograniczonym buforem – ang. bounded buffer)

- Jeden proces (producent) generuje (produkuje) dane a drugi (konsument) je pobiera (konsumuje). Wiele zastosowań w praktyce np. drukowanie.
- Jedno z rozwiązań opiera się na wykorzystaniu tablicy działającej jak bufor cykliczny, co pozwala na zamortyzowanie chwilowych różnic w szybkości producenta i konsumenta. Tę wersję problemu nazywa się problemem z *ograniczonym buforem*.
- Problem: jak zsynchronizować pracę producenta i konsumenta – np. producent zapełnia bufor, konsument usiłuje pobrać element z pustego bufora.
- Dwie wersje: dla jednego producenta i konsumenta i wielu producentów i konsumentów.





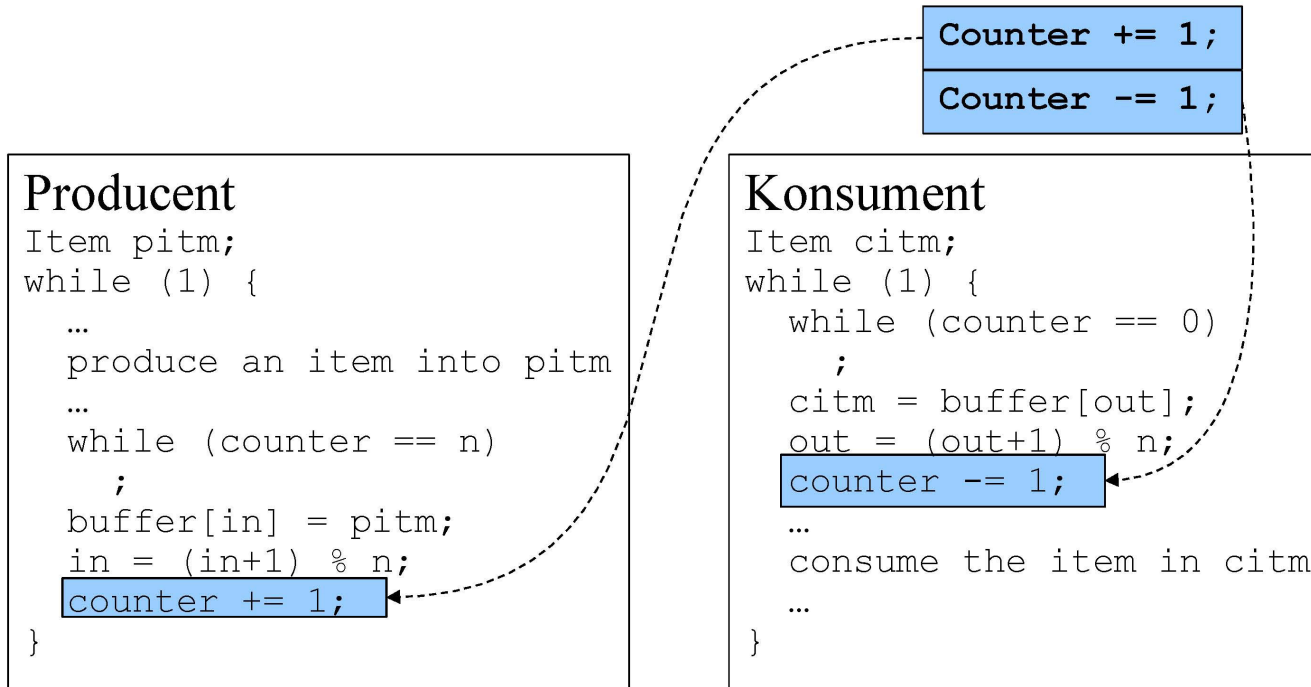
# Producent konsument z buforem cyklicznym

## Zmienne wspólne

```
const int n;                // rozmiar bufora
typedef ... Item;
Item buffer[n];             // bufor
int out=0;                  // indeks konsumenta
int in = 0;                 // indeks producenta
counter = 0;                // liczba elementów w buforze
```

- Producent umieszcza element w buforze na pozycji *in*
  - Czekaj, jeżeli  $counter == n$ , tzn. bufor pełny
- Konsument pobiera element z bufora z pozycji *out*
  - Czekaj, jeżeli  $counter == 0$  tzn. bufor pusty.
- Zmienne *in* oraz *out* zmieniane są zgodnie z regułą
  - $i = (i+1) \% n$
  - Wartości kolejnych indeksów do tablicy buffer
    - Jeżeli  $i == n-1$  to  $nowei = 0$

# Rozwiązanie (prymitywne) dla **jednego** producenta i konsumenta z wykorzystaniem aktywnego czekania

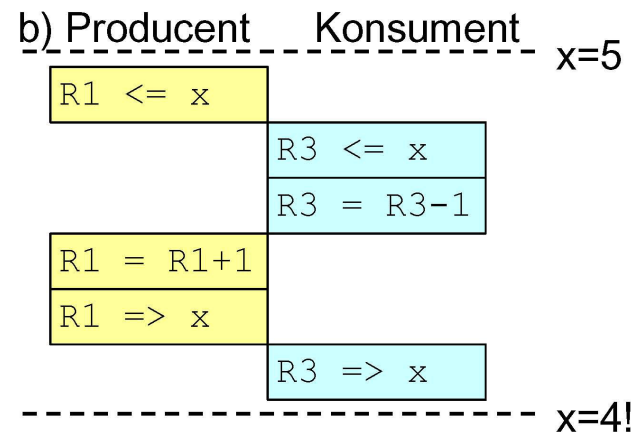
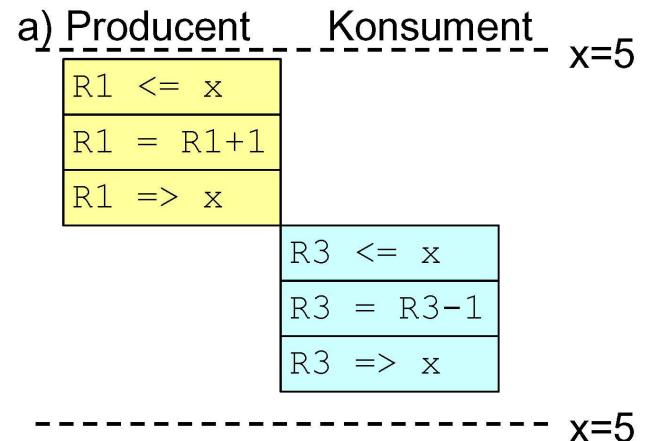


- Dygresja: dlaczego rozwiązanie oczywiście niepoprawne dla więcej niż jednego konsumenta albo producenta ?
- Counter jest zmienną współdzieloną przez obydwa procesy.
- Co się może stać gdy jednocześnie obydwa procesy spróbują ją zmienić ?

# Gdzie tkwi problem?

- Architektura RISC: ładuj do rejestru, zwiększ wartość, zapisz wynik.
- Niech  $x$  oznacza jest modyfikowaną zmienną *counter*.
  - Przyjmijmy, że  $x=5$
- Rozważmy dwie możliwe kolejności wykonywania instrukcji poszczególnych procesów.
  - a) Poprawna wartość 5.
  - b) Niepoprawna wartość 4.
- Wybór jednej z tych wielkości niedeterministyczny.

***Sytuacja wyścigu  
(ang. race condition)***

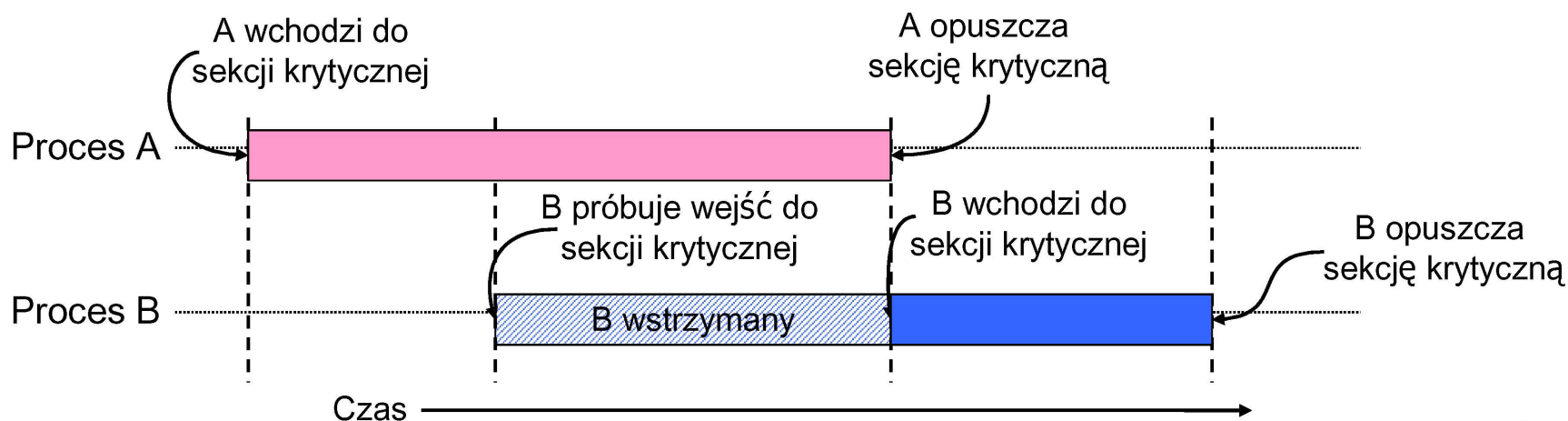


## Wyścigi - races

- O warunku wyścigu mówimy gdy wynik zależy od kolejności wykonywania instrukcji procesów. W poprawnym programie nie powinno być wyścigów. Innymi słowy
- Uwaga: Ze względu na niedeterminizm (nigdy nie wiemy w jakiej kolejności wykonają się procesy) do błędu może (ale nie musi dojść). W związku z tym przydatność testowania do badanie poprawności programów współbieżnych jest mocno ograniczona. **Nic tu nie zastąpi analizy kodu** (a często mniej lub bardziej formalnych dowodów poprawności).
- Typowe sytuacje: błąd objawia się przeciętnie raz na trzy miesiące nieprzerwanej pracy programu.
- 
- W naszym przykładzie musimy zapewnić, aby jeden proces w danej chwili mógł odwoływać się do zmiennej Counter. Innymi słowy dostęp do tej zmiennej powinien znajdować się wewnątrz **sekcji krytycznej**.

# Problem sekcji krytycznej

- Chcemy, aby w jednej chwili w sekcji krytycznej mógł przebywać tylko jeden proces
- Założenia
  - Proces na przemian przebywa w sekcji krytycznej albo wykonuje inne czynności
  - Proces przebywa w sekcji krytycznej przez skończony czas.
- Rozwiązanie
  - Czynności wykonywane przy wejściu do sekcji - *protokół wejścia*
  - Czynności wykonywane przy wyjściu z sekcji - *protokół wyjścia*



# Warunki dla rozwiązania sekcji krytycznej

- Wzajemne wykluczanie.
  - W danej chwili tylko jeden proces może być w sekcji krytycznej.
- Postęp
  - Proces który nie wykonuje sekcji krytycznej nie może blokować procesów chcących wejść do sekcji.
- Ograniczone czekanie
  - Proces nie może czekać na wejście do sekcji krytycznej w nieskończoność

# Rozwiązania problemu sekcji krytycznej

- Wyłącz przerwania
  - Nie działa w systemach wieloprocessorowych
  - Problematiczne w systemach z ochroną
  - Wykorzystywane do synchronizacji w obrębie jądra (zakładając jeden procesor)
- Rozwiązania z czekaniem aktywnym
  - Algorytm Petersona dla dwóch procesów – wymaga trzech zmiennych !!!
  - Algorytm piekarni dla wielu procesów.
  - Rozwiązania wykorzystujące specjalne instrukcje maszynowe np. rozkaz zamiany:
    - XCHG rejestr, pamięć
    - Architektura systemu musi zapewniać atomowe wykonanie instrukcji.
    - W systemach wieloprocessorowych nie jest to trywialne

## Przykład realizacji z wykorzystaniem instrukcji XCHG

- Niech instrukcja XCHG (zmienna, wartość) nadaje zmiennej nową wartość i jednocześnie zwraca starą. Zakładamy, że jest to instrukcja atomowa – nie może być przerwana.
  - Na ogół wartość przechowywana jest w rejestrze. Implementacja tej instrukcji nie jest trywialna – wymaga dwóch cykli dostępu do pamięci. Na szczęście to problem projektantów sprzętu.
- Możemy podać stosunkowo proste rozwiązanie problemu sekcji krytycznej.

```
int lock=0; // zmienna wykorzystana do synchronizacji
           // lock==1 oznacza, że jakiś proces jest w sekcji krytycznej

void Process() {

    while (1) {

        // Wynik xchg równy jeden oznacza, że poprzednia wartość była równa 1
        // zatem ktoś inny był w sekcji krytycznej
        while(xchg(lock,1)==1); // Protokół wejścia
        // Proces wykonuje swoją sekcję krytyczną, wiemy że lock==1

        lock=0; // Protokół wyjścia
        // Proces wykonuje pozostałe czynności
    }
}
```



## Dlaczego potrzebujemy XCHG ?

- Ktoś mógłby zaproponować “ulepszenie” nie wymagające tej instrukcji.

```
while (lock==1); // czekamy, aż inni opuszczą sekcję krytyczną.  
lock=1; // wchodzimy do sekcji krytycznej i zabraniamy tego innym.
```

- Niestety to “ulepszenie” jest *niepoprawne* - prowadzi do wyścigu. Dlaczego ?  
Wskazówka: wiele się może zmienić pomiędzy wyjściem z pętli while a przypisaniem zmiennej lock.

# Czekanie aktywne

- Marnowany jest czas procesora
  - Zmarnowany czas można by przeznaczyć na wykonanie innego procesu.
- Uzasadnione gdy:
  - Czas oczekiwania stosunkowo krótki (najlepiej krótszy od czasu przełączenia kontekstu)
  - Liczba procesów  $\cong$  Liczba procesorów
- Przykład zastosowania jądro Linux-a w wersji SMP
  - Funkcje typu *spin\_lock*
- Alternatywą do czekania aktywnego jest przejście procesu w stan zablokowany
  - Semaforey
  - Monitory

# Semafor zliczający

- Zmienna całkowita  $S$  i trzy operacje: nadanie wartości początkowej, oraz Wait i Signal.
- Definicja klasyczna (E. Dijkstra):
  - Wait (czekaj): while ( $S \leq 0$ ) ;  $S--$
  - Signal(sygnalizuj):  $S++$
  - Operacje Wait i Signal są operacjami atomowymi
- Początkowa wartość  $S$  – liczba wywołań operacji Wait bez wstrzymywania.
- Definicja klasyczna oparta jest na aktywnym czekaniu. W praktyce używa się innej definicji opartej na usypianiu procesów (M. Ben-Ari) :
  - Wait: Jeżeli  $S > 0$ , to  $S = S - 1$ , w przeciwnym wypadku wstrzymaj (przełącz w stan oczekujący) wykonywanie procesu – proces ten nazywany **wstrzymanym przez semafor**.
  - Signal: Jeżeli są procesy wstrzymane przez semafor, to obudź jeden z nich, w przeciwnym wypadku  $S = S + 1$ .
- Implementacja według powyższej definicji m.in. w standardzie POSIX threads.
  - funkcje `sem_init`, `sem_wait` oraz `sem_post` (odpowiednik signal)
  - funkcja `sem_trywait` – nie wstrzymuje procesu, ale zwracająca kod błędu jeżeli proces byłby wstrzymany.

# Implementacja semafora

- Semafor to: (a) Bieżąca wartość + (b) Lista (np. FIFO) procesów oczekujących
- Nieco zmodyfikowana (ale równoważna z definicją Ben Ariego) implementacja – zakładamy że wartość zmiennej może być ujemna – wtedy przechowuje ona liczbę wstrzymanych procesów.
- Zakładamy dostępność dwóch funkcji na poziomie jądra systemu:
  - Sleep: realizuje przejście procesu Aktywny=>Oczekujący
  - Wakeup: Oczekujący=>Gotowy
- Oczywiście Wait i Signal muszą być operacjami atomowymi – ich wykonanie nie może być przerwane przełączeniem kontekstu do innego procesu.

```
class Semaphore {
    int value;
    ProcessList pl;
public:
    Semaphore(int a) {value=a;}
    void Wait ();
    void Signal ();
};

Semaphore::Wait() ()
{
    value -= 1;
    if (value < 0) {
        Add(this_process,pl)
        Sleep (this_process);
    }
}

Semaphore::Signal () {
    value += 1;
    if (value <= 0) {
        Process P=Remove (P)
        Wakeup (P);
    }
}
```

# Rozwiązanie sekcji problemu sekcji krytycznej przy pomocy semaforów

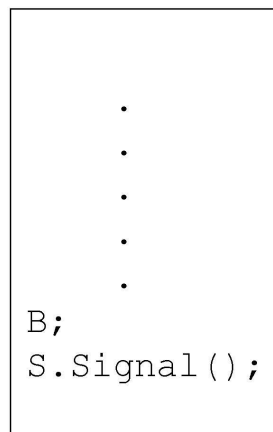
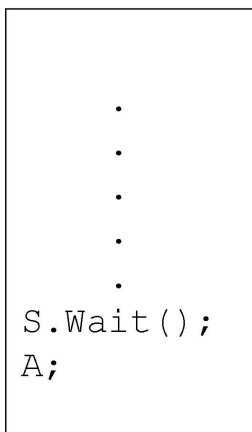
```
Semaphore Sem(1);

void Process() {
    while (1) {
        Sem.Wait();
        // Proces wykonuje swoją sekcję krytyczną
        Sem.Signal();
        // Proces wykonuje pozostałe czynności
    }
}
```

- Protokół wejścia i wyjścia są trywialne, ponieważ semaforey zaprojektowano jako narzędzie do rozwiązania problemu sekcji krytycznej
- Zmodyfikujemy warunki zadania, tak że w sekcji krytycznej może przebywać jednocześnie co najwyżej  $K$  procesów.
- **Pytanie.** Co należy zmienić w programie ?

# Zastosowanie semafora to zapewnienia określonej kolejności wykonywania instrukcji procesów

- Chcemy aby instrukcja A jednego procesu wykonała się po instrukcji B drugiego. Używamy semafora S zainicjalizowanego na zero.



# Problem producent-konsument z wykorzystaniem semaforów

```
const int n;  
Semaphore empty(n), full(0), mutex(1);  
Item buffer[n];
```

## Producent

```
int in = 0;  
Item pitem;  
while (1) {  
    // produce an item into pitem  
    empty.Wait();  
    mutex.Wait();  
    buffer[in] = pitem;  
    in = (in+1) % n;  
    mutex.Signal();  
    full.Signal();  
}
```

## Konsument

```
int out = 0;  
Item citem;  
while (1) {  
    full.Wait();  
    mutex.Wait();  
    citem = buffer[out];  
    out = (out+1) % n;  
    mutex.Signal();  
    empty.Signal();  
    // consume item from citem  
}
```

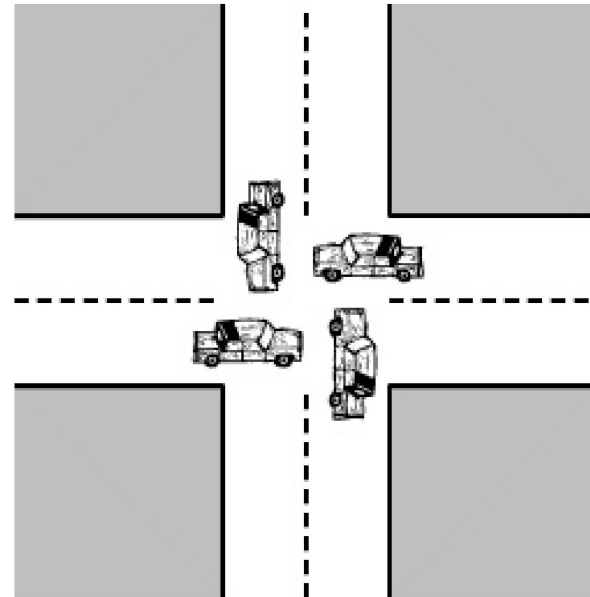
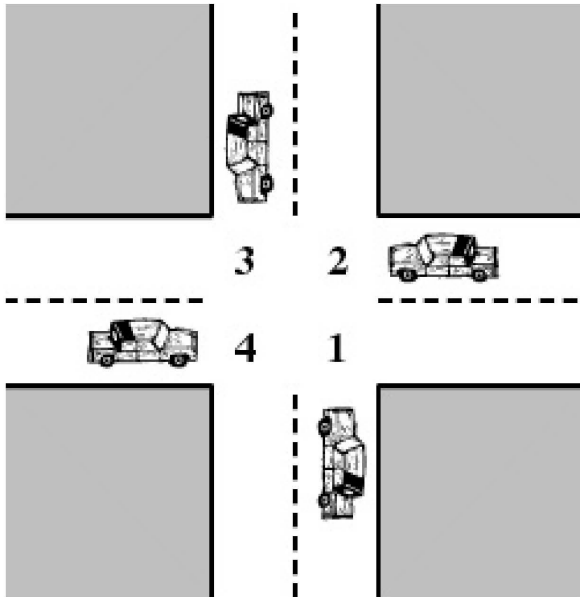
- Semafor *mutex* zapewnia wzajemne wykluczanie przy dostępie do zmiennych współdzielonych.
- Semafor *full* zlicza liczbę elementów w buforze (pełnych miejsc w tablicy). Wstrzymuje konsumenta gdy w buforze nie ma żadnego elementu.
- Semafor *empty* zlicza liczby pustych miejsc w tablicy. Wstrzymuje producenta gdy w tablicy nie ma wolnego miejsca..

# Semaforey binarne

- Zmienna może przyjmować tylko wartość zero lub jeden
  - Operacje mają symbole WaitB, SingalB
  - Wartość jeden oznacza, że można wejść do semafora (wykonać WaitB)
  - Wartość zero oznacza że operacja WaitB wstrzyma proces.
- Mogą być prostsze w implementacji od semaforów zliczających.
- Implementacje
  - Mutexy w POSIX threads. (pthread\_mutex\_create, pthread\_mutex\_lock, pthread\_mutex\_unlock).
  - W win32 mutexy noszą nazwę sekcji krytycznych
  - W Javie mutex jest związany z każdym obiektem
    - Słowo kluczowe *synchronized*.
    - Więcej o Javie przy omawianiu monitorów



## Blokada (Zakleszczenie, ang. deadlock)



- Zbiór procesów jest w stanie blokady, kiedy każdy z nich czeka na zdarzenie, które może zostać spowodowane wyłącznie przez jakiś inny proces z tego zbioru.
- Samochody nie mają wstecznego biegu = Brak wywłaszczeń zasobów

## Przykład blokady

- Sekwencja instrukcji prowadząca do blokady.
  - $P_0$  wykonał operacje `A.Wait()`
  - $P_1$  wykonał operacje `B.Wait()`
  - $P_0$  usiłuje wykonać `B.Wait()`
  - $P_1$  usiłuje wykonać `A.Wait()`
  - $P_0$  czeka na zwolnienie B przez  $P_1$
  - $P_1$  czeka na zwolnienie B przez  $P_0$
  - ***Będą czekały w nieskończoność !!!***
- Do blokady ***może*** (ale nie musi) dojść.
- ***Pytanie:*** Jak w tej sytuacji zagwarantować brak blokady ?

```
Semaphore A(1), B(1);
```

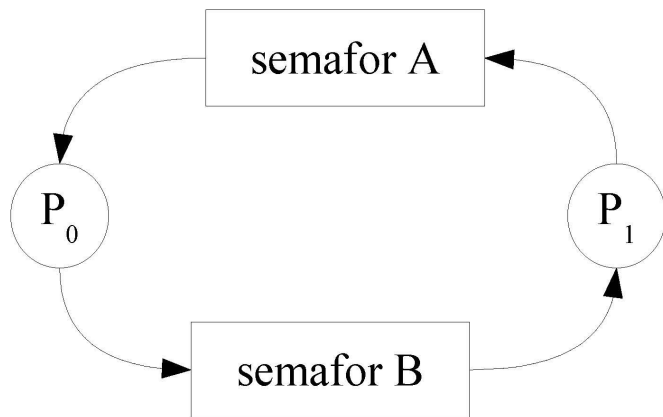
Proces  $P_0$

```
A.Wait();  
B.Wait();  
.  
.  
.  
B.Signal();  
A.Signal();
```

Proces  $P_1$

```
B.Wait();  
A.Wait();  
.  
.  
.  
A.Signal();  
B.Signal();
```

## Opis formalny: graf alokacji zasobów



Okrąg oznacza proces, a prostokąt zasób.

- Strzałka od procesu do zasobu  $\Rightarrow$  proces czeka na zwolnienie zasobu
- Strzałka od zasobu do procesu  $\Rightarrow$  proces wszedł w posiadanie zasobu.

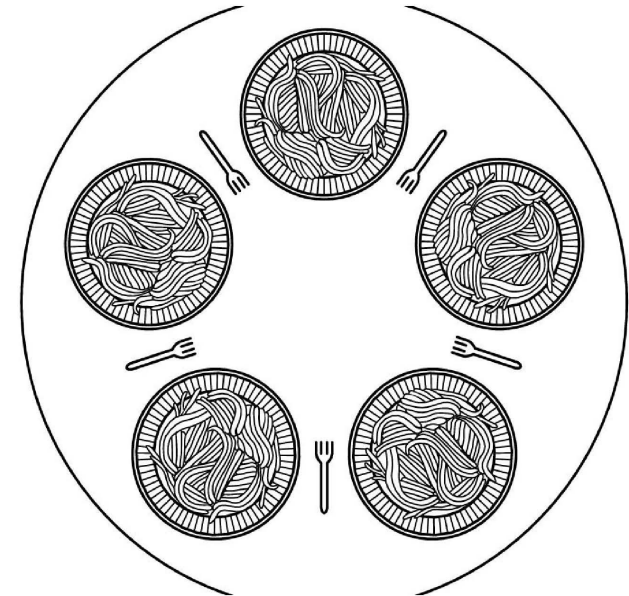
- Stan blokady ma miejsce, wtedy i tylko wtedy gdy w grafie alokacji zasobów występuje *cykl*.
- Jedną z metod uniknięcia blokady  $\Rightarrow$  nie dopuszczaj do powstania cyklu. Np. każdy proces wchodzi w posiadanie zasobów w określonym porządku (identycznym dla wszystkich procesów).
- W literaturze (Silberschatz i wsp.) opisano wersję z więcej niż jednym egzemplarzem zasobu (np. drukarki)

## Zagłodzenie (ang. starvation)

- Proces czeka w nieskończoność, pomimo że zdarzenie na które czeka występuje. (Na zdarzenie reagują inne procesy)
- Przykład: Jednokierunkowe przejście dla pieszych, przez które w danej chwili może przechodzić co najwyżej jedna osoba.
  - Osoby czekające na przejściu tworzą kolejkę.
  - Z kolejki wybierana jest zawsze najwyższa osoba
  - Bardzo niska osoba może czekać w nieskończoność.
- Zamiast kolejki priorytetowej należy użyć kolejki FIFO (wybieramy tę osobę, która zgłosiła się najwcześniej).
- Inny przykład: z grupy procesów gotowych planista krótkoterminowy przydziela zawsze procesor najpierw procesom profesorów a w dalszej kolejności procesom studentów.
  - Jeżeli w systemie jest wiele procesów profesorów, to w kolejce procesów gotowych znajdzie się zawsze co najmniej jeden i proces studenta będzie czekał w nieskończoność na przydział procesora.

# Problem pięciu filozofów

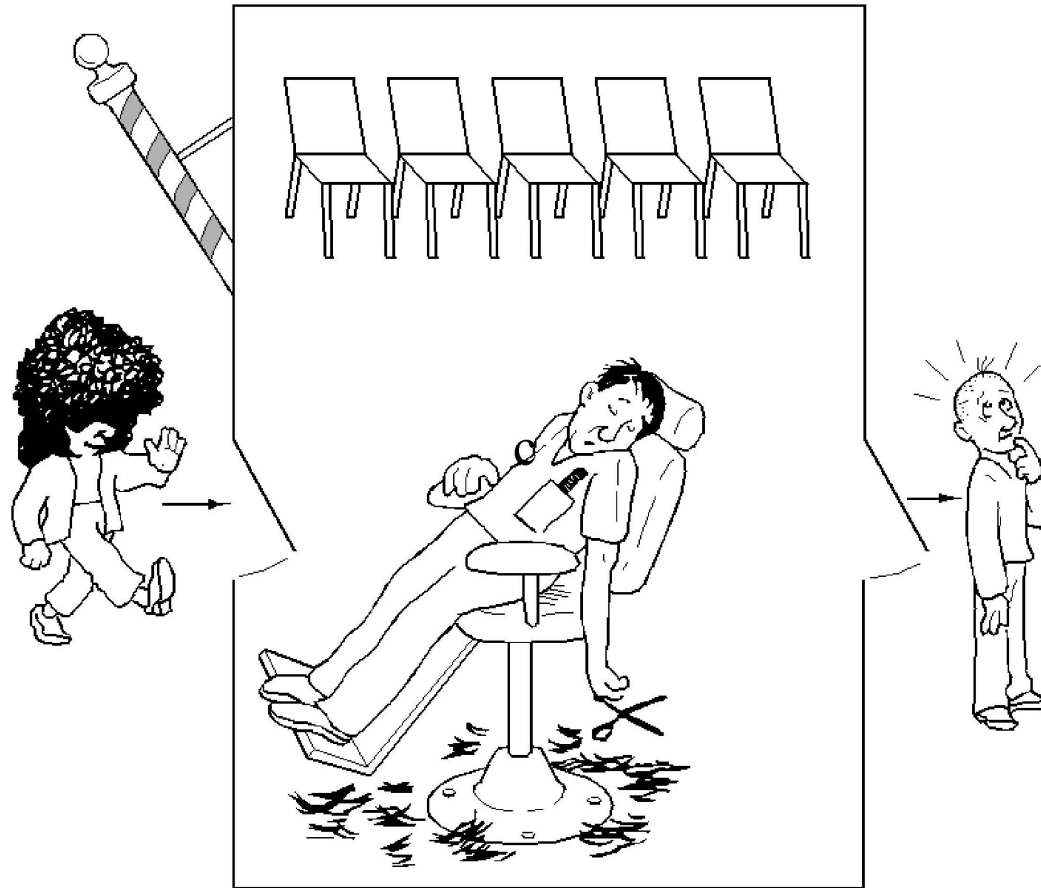
- Każdy filozof siedzi przed jednym talerzem
- Każdy filozof na przemian myśli i je
- Do jedzenia potrzebuje dwóch widelców
  - Widelec po lewej stronie talerza.
  - Widelec po prawej stronie talerza.
- W danej chwili widelec może być posiadany tylko przez jednego filozofa.
- Zadanie: Podaj kod dla procesu *i-tego* filozofa koordynujący korzystanie z widelców.



## Problem czytelników i pisarzy

- Modyfikacja problemu sekcji krytycznej.
- Wprowadzamy dwie klasy procesów: *czytelników* i *pisarzy*.
- Współdzielony obiekt nazywany jest *czytelnią*.
- W danej chwili w czytelni może przebywać
  - Jeden proces pisarza i żaden czytelnik.
  - Dowolna liczba czytelników i żaden pisarz.
- Rozwiązanie *prymitywne*: Potraktować czytelnię jak obiekt wymagający wzajemnego wykluczania wszystkich typów procesów.
  - Prymitywne, ponieważ ma bardzo słabą wydajność. Jeżeli na wejście do czytelni czeka wielu czytelników i żaden pisarz to możemy wpuścić od razu wszystkich czytelników
- W literaturze opisano rozwiązania:
  - Z możliwością zagłódenia pisarzy
  - Z możliwością zagłódenia czytelników
  - Poprawne

## Problem śpiącego fryzjera



- Jeden proces *fryzjera* i wiele procesów *klientów*.
- Współdzielone zasoby:  $n$  krzeseł w poczekalni i jedno krzesło fryzjera
- Napisz program koordynujący pracę fryzjera i klientów

## Wady semaforów

- Jeden z pierwszych mechanizmów synchronizacji
- Generalnie jest to mechanizm bardzo niskiego poziomu - trochę odpowiadający programowaniu w assemblerze.
- Duża podatność na błędy, trudno wykazać poprawność programu
- Przykład: Jeżeli zapomnimy o operacji V, nastąpi blokada
- Bardziej strukturalne mechanizmy synchronizacji
  - Regiony krytyczne
  - Monitory



## Regiony krytyczne

- Współdzielona zmienna  $v$  typu  $T$  jest deklarowana jako:

**var  $v$ : shared  $T$**

- Dostęp do zmiennej  $v$  wykonywany przy pomocy operacji

**region  $v$  when  $B$  do  $S$**

- $B$  jest wyrażeniem logicznym
- Tak długo, jak instrukcja  $S$  się wykonuje, żaden inny proces nie może się odwołać do zmiennej  $v$ .
- Jeżeli wyrażenie  $B$  nie jest spełnione, to proces jest wstrzymywany do momentu jego spełnienia.

# Przykład: producent-konsument z ograniczonym buforem

```
var buffer: shared record  
    pool: array [0..n-1] of item;  
        count,in,out: integer  
end;
```

```
region buffer when count < n  
    do begin  
        pool[in] := nextp;  
        in := in + 1 mod n;  
        count := count + 1;  
    end;
```

```
region buffer when count > 0  
    do begin  
        nextc := pool[out];  
        out := out + 1 mod n;  
        count := count - 1;  
    end;
```

- Deklaracja zmiennej współdzielonej.
- Wstawienie elementu *nextp* do bufora. (Producent).
- Usunięcie elementu *nextc* z bufora (Konsument).

## Idea monitora (a właściwie zmiennej warunkowej)

- Udostępnienie procesom operacji pozwalającej procesowi wejść w stan uśpienia (zablokowania) – *wait* oraz operacji *signal* pozwalającej na uśpienie obudzonego procesu.
- Ale tu natrafiamy na (stary) problem wyścigów, który ilustruje poniższy przykład:

```
if (Jeszcze_nie_bylo_zdarzenia_muszę_wykonać_wait)
    wait() // to zaczekam
```

- Co się stanie, jeżeli zdarzenie na które czeka proces zajdzie po instrukcji if, ale przed uśpieniem procesu ? Proces zgubi zdarzenie (i być może nigdy się nie obudzi)
- W takim razie wykonajmy cały ten kod wewnątrz sekcji krytycznej ?
  - Ale gdy proces wykona wait() - to przejdzie w stan uśpienia nie zwalniając sekcji krytycznej. Przy próbie wejścia do sekcji przez inny proces na pewno dojdzie do blokady.
- Rozwiązanie: Atomowa operacja wait powodująca jednoczesne uśpienie procesu i wyjście z sekcji krytycznej

# Monitory

```
monitor mon {  
    int foo;  
    int bar;  
    public void proc1(...) {  
    }  
    public void proc2(...) {  
    }  
};
```

- Pseudokod przypominający definicję klasy C++.
- Współdzielone zmienne *foo* oraz *bar* są dostępne wyłącznie z procedur monitora.
- Procesy synchronizują się wywołując procedury monitora (np. *proc1* i *proc2*)
  - Tylko jeden proces (wątek) może w danej chwili przebywać w procedurze monitora. Gwarantuje to automatycznie wzajemne wykluczanie.
- Mówimy że proces “*przebywa wewnątrz monitora*”.

## Zmienne warunkowe (ang. condition)

- Problem: proces postanawia zaczekać wewnątrz monitora aż zajdzie zdarzenie sygnalizowane przez inny proces.
  - Jeżeli proces po prostu zacznie czekać, nastąpi blokada, bo żaden inny proces nie będzie mógł wejść do monitora i zasygnalizować zdarzenia.
- Zmienne warunkowa (typu condition). Proces, **będący wewnątrz monitora**, może wykonać na niej dwie operacje.
  - Niech deklaracja ma postać: *Condition C*;
  - *C.wait()* Zawiesza wykonanie procesu, i jednocześnie zwalnia monitor pozwalając innym procesom wejść do monitora.
  - *C.signal()* Jeżeli nie ma procesów zawieszonych przez operację *wait* nic się nie dzieje. W przeciwnym wypadku dokładnie jeden proces zawieszony przez operację *wait* zostanie wznowiony. (od następnej instrukcji po *wait*).
  - Możliwa jest trzecia operacja *C.signallAll()* wznowiająca wszystkie zawieszone procesy.

# Przykład 1: Implementacja semafora zliczającego przy pomocy monitora

```
monitor Semafor {
    int Licznik;
    Condition NieZero;

    // coś na kształt konstruktora
    Semafor(int i) {
        Licznik=i;
    }

    void P() {
        if (Licznik==0)
            NieZero.wait();
        Licznik=Licznik-1;
    }

    void V() {
        Licznik=Licznik+1;
        NieZero.signal();
    }
};
```

- Deklaracja:
  - Semafor S(1);
- Proces **potrzebujący** wzajemnego wykluczania.

S.P()

// sekcja krytyczna

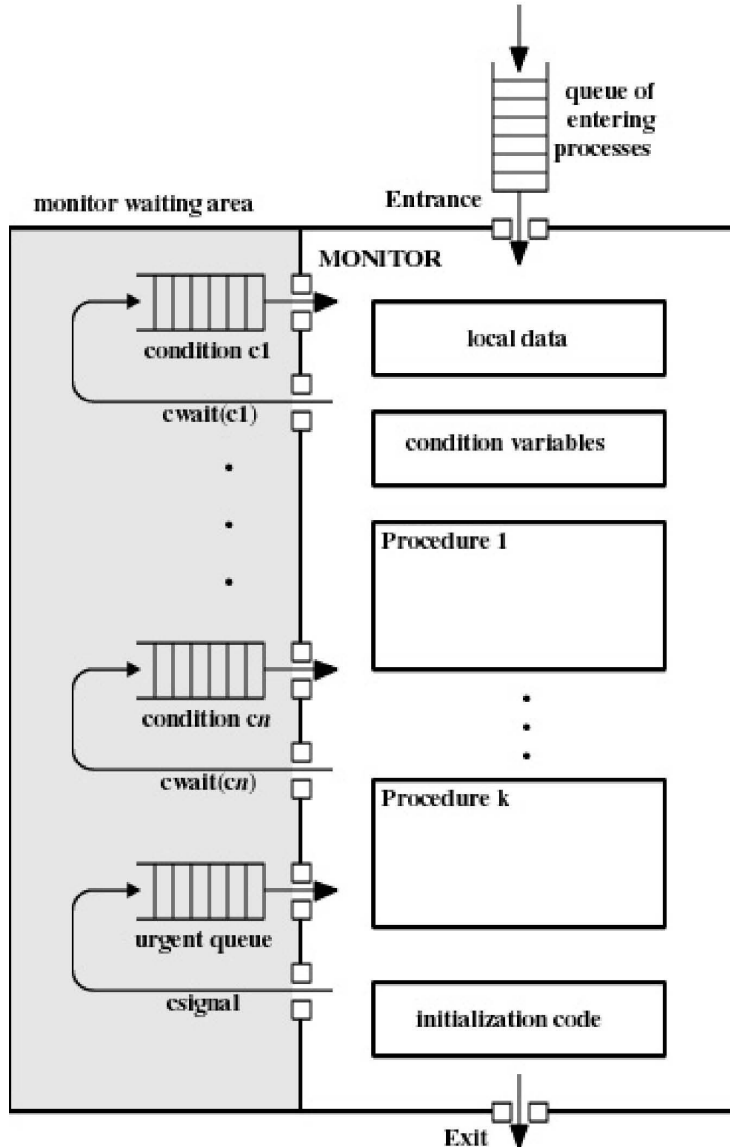
S.V()

// pozostałe czynności

# Semantyka Hoare'a i semantyka Mesa

- Przypuśćmy że proces P wykonał operację *wait* i został zawieszony. Po jakimś czasie proces Q wykonuje operację *signal* odblokowując P.
- Problem: który proces dalej kontynuuje pracę: P czy Q. Zgodnie z zasadą działania monitora tylko jeden proces może kontynuować pracę.
- Semantyka Mesa
  - Proces który wywołał operację *signal* (Q) kontynuuje pierwszy.
  - P może wznowić działanie, gdy Q opuści monitor.
  - Wydaje się być zgodna z logiką, po co wstrzymywać proces który zgłosił zdarzenie.
- Semantyka Hoare'a
  - Proces odblokowany (P) kontynuuje jako pierwszy.
  - Może ułatwiać pisanie poprawnych programów. W przypadku semantyki Mesa nie mamy gwarancji, że warunek, na jaki czekał P jest nadal spełniony (P powinien raz jeszcze sprawdzić warunek).
- Aby uniknąć problemów z semantyką najlepiej przyjąć że operacja *signal* jest zawsze ostatnią operacją procedury monitora.

# Struktura monitora



- Zasada działania: W danej chwili w procedurze monitora może przebywać jeden proces.
- Z każdą zmienną warunkową związana jest kolejka procesów, które wywołały *wait* i oczekują na zasygnalizowanie operacji.
- Po zasygnalizowaniu warunku proces (który wykonał operację *signal*) przechodzi do kolejki *urgent queue*.
  - Zatem implementacja realizuje semantykę Hoare'a



## Przykład 2: Problem producent-konsument z buforem cyklicznym

```
monitor ProducentKonsument {
    int Licznik=0,in=0,out=0;
    Condition Pełny;
    Condition Pusty;
    int Bufor[N];

    void Wstaw(int x) {
        if (Licznik==n)
            Pełny.wait();
        Bufor[in]=x;
        in=(in+1)%N;
        Licznik++;
        Pusty.signal();
    }

    int Pobierz() {
        if (Licznik==0)
            Pusty.wait();
        int x=Bufor[out];
        out=(out+1)%N;
        Licznik=Licznik-1;
        Pełny.signal();
        return x;
    }
};
```

- Rozwiązanie dla wielu konsumentów i wielu producentów.
- Przyjmujemy, że w buforze są przechowywane liczby całkowite (int).
- Producent chcąc wstawić element do bufora wywołuje procedurę monitora *Wstaw*.
- Konsument chcąc pobrać element z bufora wywołuje *Pobierz*.
- Gdy bufor jest pusty, to konsumenci są wstrzymywani na zmiennej warunkowej *Pusty*.
- Gdy bufor jest pełny to producenci są wstrzymywani na zmiennej warunkowej *Pełny*.

# Tworzenie wątku w Javie

```
class Worker extends Thread
{
    public void run() {
        System.out.println("Wątek roboczy");
    }
}

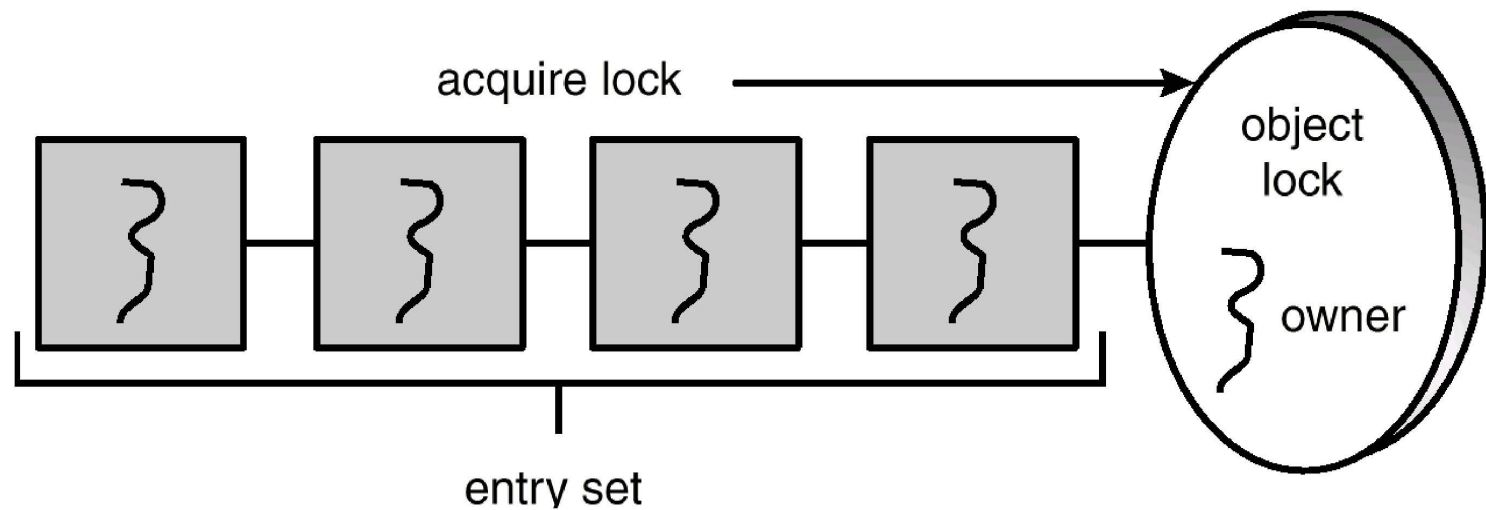
public class First
{
    public static void main(String args[]) {
        Worker runner = new Worker();
        runner.start();
        System.out.println("Wątek główny");
        runner.join();
    }
}
```

- Dwie metody:
  - Rozszerzenie klasy Thread
  - Implementacja interfejsu Runnable
- Rozszerzenie klasy Thread
  - Metoda run jest wykonywana w odrębnym wątku
  - Deklarujemy obiekt klasy
  - Metoda start() uruchamia wątek.
    - Reprezentowany przez obiekt klasy Worker.
  - Metoda join() zawiesza aktualny wątek do momentu zakończenia wątku reprezentowanego przez obiekt klasy Thread.

# Metody synchronizowane w Javie

- Z *każdym* obiektem w Javie związany jest *zamek* (ang. lock).
- Metoda jest zsynchronizowana, jeżeli przed jej deklaracją stoi słowo kluczowe *synchronized*.
- Zamek gwarantuje wzajemne wykluczanie metod synchronizowanych obiektu
  - Aby wykonać metodę synchronizowaną wątek musi wejść w posiadanie zamka.
  - Wątek kończąc metodę synchronizowaną zwalnia zamek
  - Jeżeli wątek próbuje wywołać metodę synchronizowaną, a zamek jest już posiadany przez inny wątek (wykonujący właśnie metodę synchronizowaną), to jest zostaje on zablokowany i dodany kolejki wątków oczekujących na zwolnienie zamka.

## Blokada (ang. lock) obiektu w Javie



# Producent-konsument w Javie z semi-aktywnym oczekiwaniem

```
class ProducentKonsument {
    int Licznik=0,in=0,out=0;
    static final int N=100;
    int Bufor[N];

    public void Wstaw(int x) {
        while (Licznik==N)
            Thread.yield();
        synchronized(this) {
            Bufor[in]=x;
            in=(in+1)%N;
            Licznik++;
        }
    }

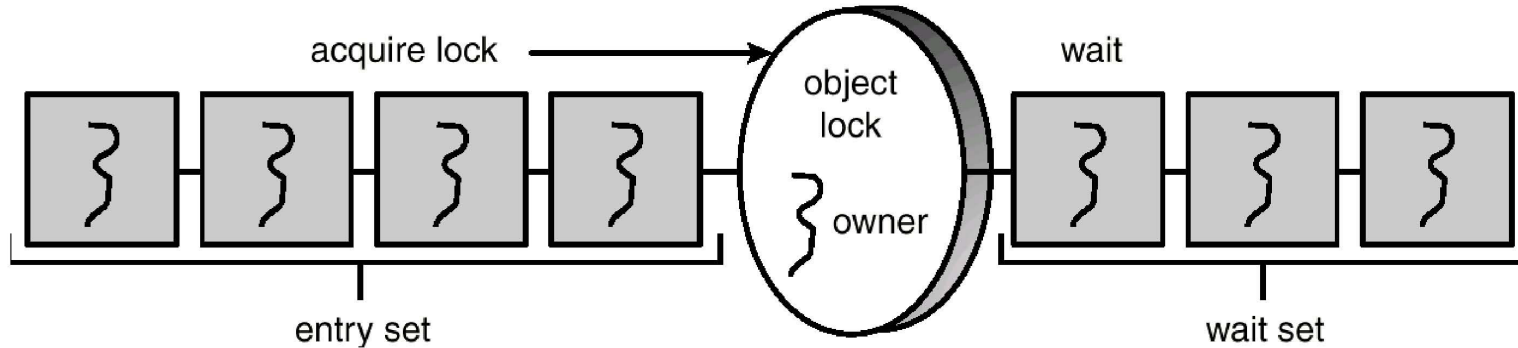
    public int Pobierz() {
        while(Licznik==N)
            Thread.yield();
        synchronized(this) {
            int x=Bufor[out];
            out=(out+1)%N;
            Licznik=Licznik-1;
        }
        return x;
    }
}
```

- Thread.yield pozwala na przekazanie sterowania innemu wątkowi lub procesowi.
  - *Ciągle jest to aktywne czekanie - nie zalecane.*
- Synchronizowany blok kodu
  - Często nie ma konieczności synchronizowania całej metody
  - W danej jeden wątek może wykonywać synchronizowany blok kodu jednego obiektu.
  - Wątek ten posiada blokadę obiektu
- W książce “Applied Operating Systems Concepts” użyto synchronizowanych metod. Czy jest to poprawne ?
- Powyższe rozwiązanie jest poprawne wyłącznie dla jednego procesu konsumenta i jednego procesu producenta. (*Dlaczego ?*).

## Metody wait oraz notify

- Wątek posiadający blokadę obiektu może wykonać metodę *wait*. (tego obiektu)
  - Wątek natychmiast zwalnia blokadę (inne wątki mogą wejść w posiadanie blokady)
  - Zostaje uśpiony..
  - Umieszczany jest w kolejce wątków zawieszonych (ang. wait set) obiektu.
  - Należy obsłużyć wyjątek InterruptedException.
- Wątek posiadający blokadę obiektu może wykonać metodę *notify*.
  - Metoda ta sprawia, że jeden wątek z kolejki wątków zawieszonych zostanie przesunięty do kolejki wątków oczekujących na zwolnienie blokady.
- Metoda *notifyAll* powoduje przeniesienie wszystkich wątków zawieszonych.
- W Javie istnieją również metody *suspend* i *resume*.
  - **Są niebezpieczne i nie należy ich stosować !!!**

## Blokada obiektu w Javie (wersja ostateczna)



- Jeden wątek jest właścicielem – wykonuje kod synchronizowany
- Entry Set - wątki oczekujące na wejście w posiadanie zamka.
- Wait Set – wątki zawieszone poprzez metodę *wait*
- Wywołanie metody *notify* przenosi wątek z wait set do entry set
- Obiekt w Javie odpowiada monitorowi z **maksymalnie jedną zmienną** warunku. To rozwiązanie obniża wydajność synchronizacji - każdy wątek po obudzeniu, musi sprawdzić czy obudziło go zdarzenie na które czekał.
  - W przypadku producenta - konsumenta z ograniczonym buforem mamy dwa typy zdarzeń (bufor niepełny oraz bufor pełny)



# Producent konsument z wykorzystaniem metod wait/notify

```
class ProducentKonsument {
    int Licznik=0,in=0,out=0;
    int Bufor[N];

    public synchronized void Wstaw(int x) {
        while (Licznik==N)
            try { wait(); }
            catch (InterruptedException e) {;}
        Bufor[in]=x;
        in=(in+1)%N;
        Licznik++;
        notifyAll();
    }

    public synchronized int Pobierz() {
        while (Licznik==0)
            try { wait(); }
            catch (InterruptedException e) {;}
        int x=Bufor[out];
        out=(out+1)%N;
        Licznik=Licznik-1;
        notifyAll();
        return x;
    }
};
```

- Słabość metod wait/notify - trzeba „do skutku” sprawdzać warunek.



## Synchronizacja w Javie, a monitory.

- W monitorze możemy zadeklarować wiele zmiennych warunkowych.
- Klasa w Javie w przybliżeniu odpowiada monitorowi z jedną zmienną warunkową.
- Różnice są widoczne w przypadku rozwiązania problemu producent-konsument.
  - Wersja z monitorami wykorzystuje dwie zmienne warunkowe
  - W wersji w Javie konsument oczekujący na pojawienie się elementu w buforze może zostać powiadomiony przez innego konsumenta.
  - Z tego powodu po obudzeniu należy raz jeszcze sprawdzić warunek (pętla while).
- Brak zmiennych warunkowych prowadzi do niskiej wydajności: np. budzeni są wszyscy czekający konsumenci ale tylko jeden z nich może kontynuować.
- Specyfikacja Javy mówi, że wątek wywołujący metodę notify kontynuuje pierwszy.
  - Odpowiada to semantyce Mesa.

# Biblioteka POSIX Threads - implementacja monitora

- Dostarcza typy i operacje dla semaforów (`sem_t`) oraz mutexów (`pthread_mutex_t`) realizujących wzajemne wykluczanie.
- Dostarcza typ (`pthread_cond_t`) dla zmiennych warunkowych i niepodzielną operację `pthread_cond_wait(condition,mutex)` usypiającą wątek na zmiennej warunkowej i jednocześnie zwalnającą blokadę mutex. Po obudzeniu wątku oczekującego na zmiennej warunku (przez `pthread_cond_signal` albo `pthread_cond_broadcast`) nastąpi ***ponowna automatyczna re-akwizycja mutexu, przed powrotem z funkcji pthread\_cond\_wait.***
- Ponadto mamy operację na zmiennych warunkowych `pthread_cond_signal` (obudza jeden zawieszony wątek) i `pthread_cond_broadcast` (obudza wszystkie zawieszone wątki).

```
pthread_mutex_t mutex; // Realizuje wzajemne wykluczanie wew. monitora
```

```
void Funkcja_Monitora() {  
    pthread_mutex_lock(&mutex); // Wejście do monitora  
    .....  
    if (          ) {  
        pthread_mutex_unlock(&mutex);  
        return; // Teraz też opuszczamy monitor - pamiętać o return !!!  
    }  
    .....  
    pthread_mutex_unlock(&mutex); // Opuszczenie monitora  
}
```

# POSIX Threads - zmienne warunkowe monitora

- Każda zmienna warunkowa deklarowana jest jako zmienna typu `pthread_condition_t` i inicjowana przy pomocy `pthread_cond_init`:

```
pthread_cond_t condition;  
pthread_cond_init(&condition, NULL);
```

- Jeżeli wątek przebywający wewnątrz funkcji monitora (a zatem posiadający mutex), zechce wykonać operację wait na zmiennej warunku, może wykonać następujący kod:

```
// Zwolnienie muteksa i oczekiwanie na zmiennej warunku.  
pthread_cond_wait(&condition, &mutex);  
// Obudzenie nastąpi po wykonaniu operacji signal lub broadcast  
// i re-akwizycja muteksa.
```

- Wątek chcący wykonać operację signal monitora (i przebywający wewnątrz monitora tzn. posiadający muteks) wykonuje następujący kod:

```
// Czy to jest semantyka Hoare'a czy też Mesa ?  
pthread_cond_signal(&condition);
```

## Przekazywanie komunikatów (ang. message passing)

- Dostarcza dwie operacje.
  - *send*(odbiorca,dane)
  - *receive*(nadawca,dane)
- Send i receive mogą wymagać podania kanału.
- Idealne dla problemu producent konsument.
- Na ogół wymaga kopiowania danych => możliwy spadek wydajności.
- Dobra metoda synchronizacji dla systemów rozproszonych.
  - Wydajna realizacja pamięci współdzielonej w systemie rozproszonym jest bardzo trudna.

# Typy operacji send oraz receive

- Blokujące send i blokujące receive.
  - Obydwa procesy są zablokowane do momentu przekazania komunikatu.
  - Nazywane *spotkaniem* (Ada, CSP, Occam, Parallel C)
- Nieblokujące send i blokujące receive.
  - Proces wywołujący send nie musi czekać na przekazanie komunikatu.
  - Komunikat umieszczany jest w buforze
- Nieblokujące send i nieblokujące receive.
  - Żaden z pary procesów nie musi czekać na przekazanie komunikatu.
  - Operacja receive sygnalizuje brak komunikatu.
  - Dodatkowe operacje test\_completion i wait\_completion.

# Implementacje przekazywania komunikatów

- Spotkania w Adzie
- Gniazda
  - Wykorzystujące protokół TCP/IP
  - Gniazda domeny Uniksa.
- Biblioteki PVM oraz MPI.
  - Zaprojektowane z myślą o obliczeniach równoległych.
- Zdalne wywołanie procedury (ang. remote procedure call, RPC)
- Zdalne wywołanie metody (ang. remote method invocation, RMI)
- Kolejki komunikatów w Uniksie
- Nazwane (i nienazwane) potoki w Uniksie

# Dziękuję za uwagę



dr inż. Jacek Czerniak  
*[jczerniak@ukw.edu.pl](mailto:jczerniak@ukw.edu.pl)*