Wykład 4

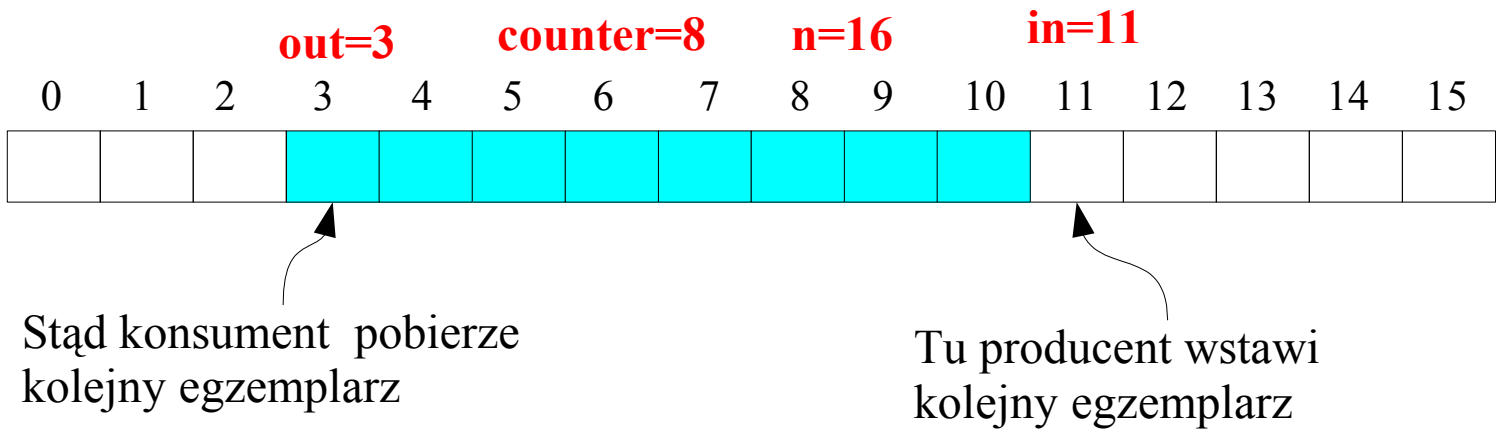
Synchronizacja procesów (i wątków) cześć I

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.



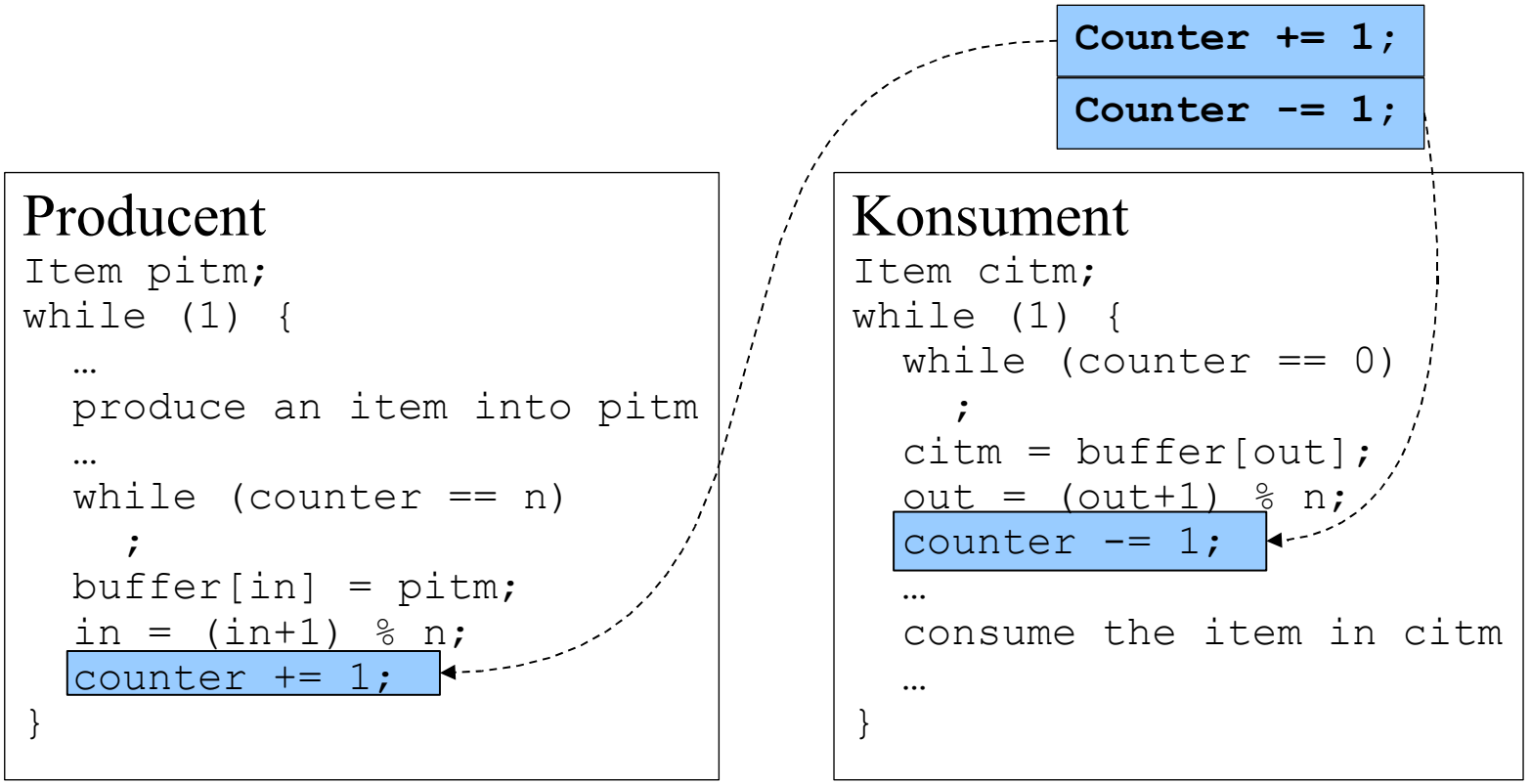
Producent konsument z buforem cyklicznym

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*
  - Czeka, jeżeli counter==n, tzn. bufor pełny
- Konsument pobiera element z bufora z pozycji *out*
  - Czeka, 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

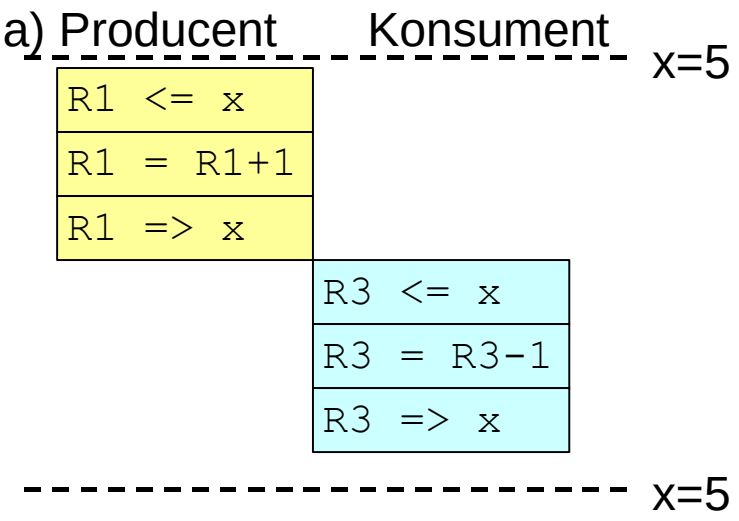
Rozwiązanie (prymitywne) dla **jednego** producenta i konsumenta z wykorzystaniem aktywnego czekania

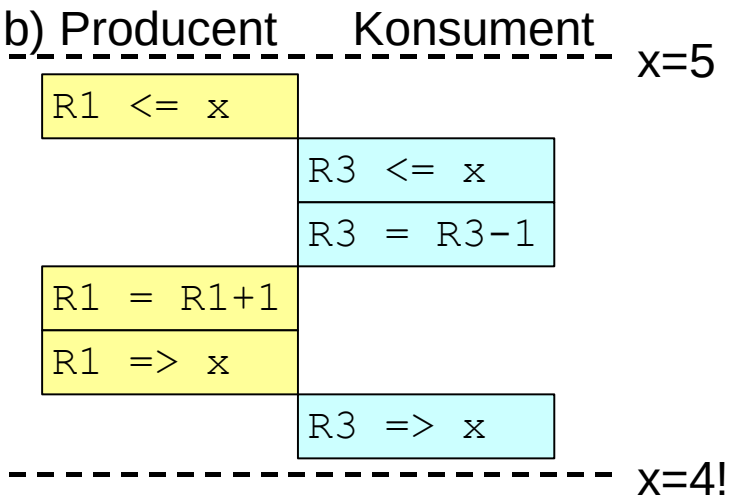


- Dygresja: dlaczego rozwiązanie oczywiście niepoprawne dla więcej niż jednego konsumenta albo producenta ?
- Counter jest zmienną współdzieloną przez obydwa procesy.
- Co się może stać gdy jednocześnie obydwa procesy spróbują ją zmienić ?

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)***



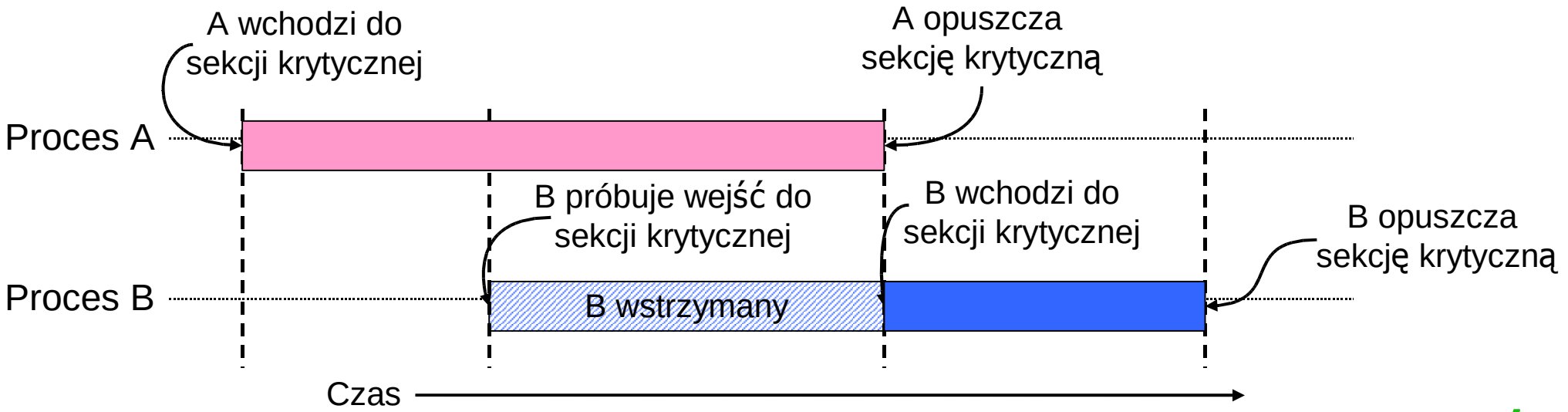


Wyścigi - races

- 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 wieloprocesorowych
  - Problematyczne 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 wieloprocesorowych 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 krytyczne
   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 sekcje 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 ≅ Liczba procesorów
- Przykład zastosowania jądro Linux-a w wersji SMP
  - Funkcje typu *spin_lock*
- Alternatywą do czekania aktywnego jest przejście procesu w stan zablokowany
  - Semafory
  - Monitory

Semafor zliczający

- Zmienna całkowita S i trzy operacje: nadanie wartości początkowej, oraz Wait i Signal.
- Definicja klasyczna (E. Dijkstra):
  - Wait (czekaj): while (S<=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 semafory zaprojektowano jako narzędzie do rozwiązania problemu sekcji krytycznej

- Zmodyfikujmy 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.

```
      .
      .
      .
      .
      .
S.Wait();
A;
```

```
      .
      .
      .
      .
      .
B;
S.Signal();
```

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

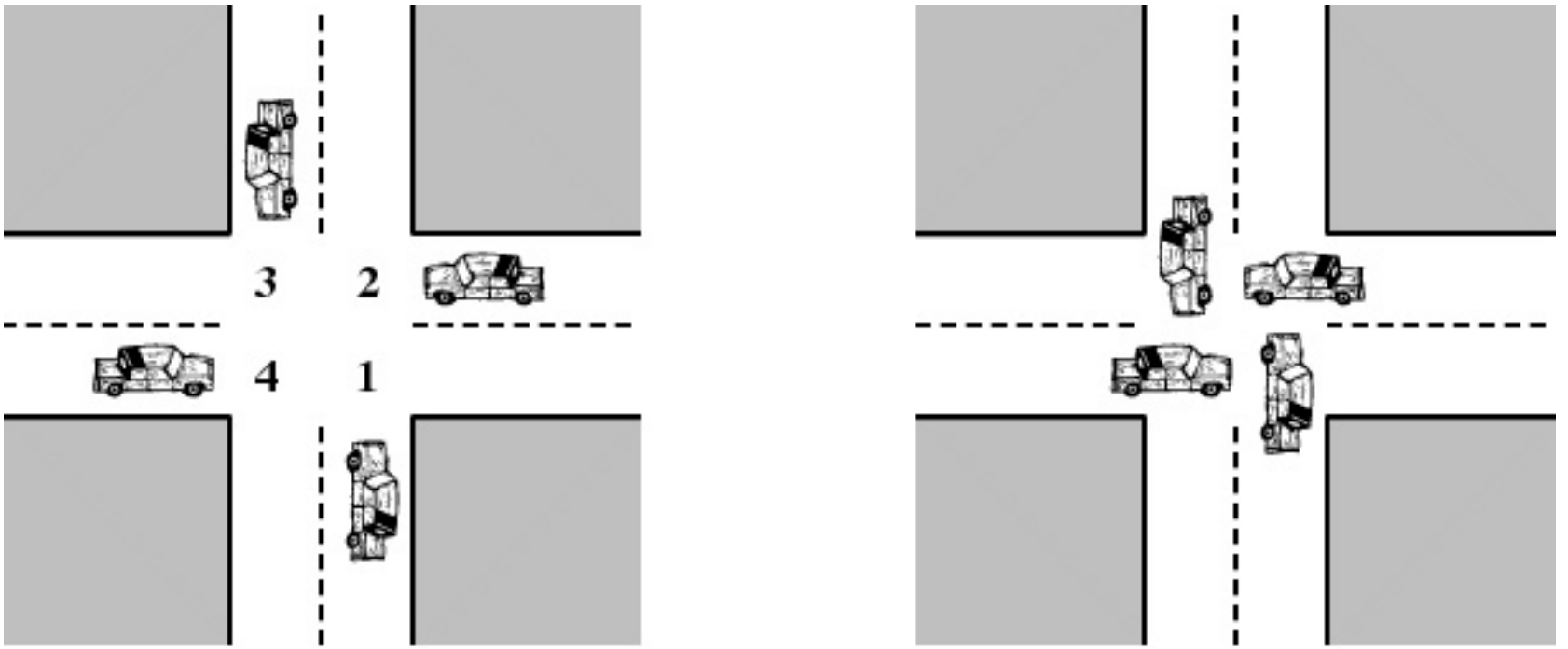
```
int out = 0;
Item citem;
while (1) {
  full.Wait();
  mutex.Wait();
  citem = buffer[out];
  out = (out+1) % n;
  mutex.Signal();
  empty.Signal();
  // consume item from citem
}
```

- 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..

Semafory 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, phread_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);
```

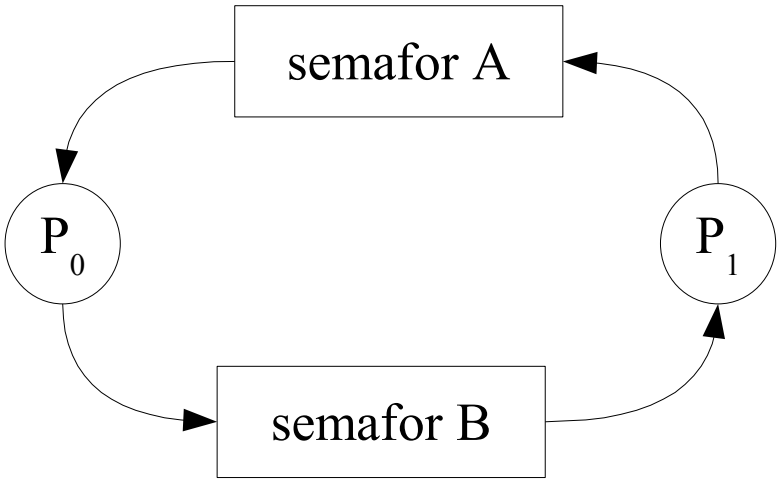
Proces P_0

```
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.

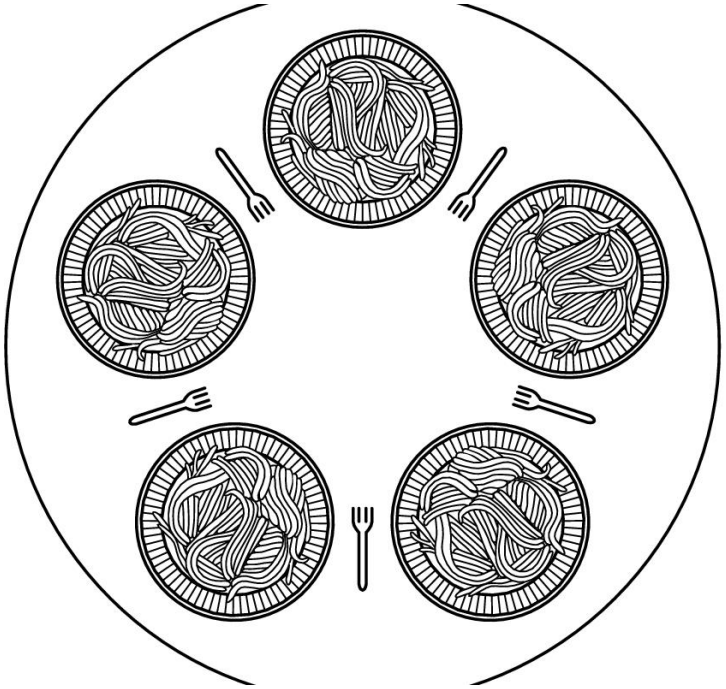
- Strzałka od procesu do zasobu => proces czeka na zwolnienie zasobu
- Strzałka od zasobu do procesu => proces wszedł w posiadanie zasobu.

- Stan blokady ma miejsce, wtedy i tylko wtedy gdy w grafie alokacji zasobów występuje ***cykl***.
- Jedna z metod uniknięcia blokady => 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 zagłodzenia pisarzy
  - Z możliwością zagłodzenia 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