Programowanie współbieżne i rozproszone

WYKŁAD 3

Krzysztof Pancerz

Wzajemne wykluczanie

- **Wzajemne wykluczanie** (ang. *mutual exclusion*) - sytuacja, w której co najwyżej jeden proces może realizować określone działanie.
- Wzajemne wykluczanie jest zapewniane przez zastosowanie semaforów lub monitorów.

Wzajemne wykluczanie

- Każdy z *N* procesów wykonuje nieskończoną pętlę złożoną z ciągu instrukcji, który można podzielić na dwa podciągi:
  - sekcję krytyczną,
  - sekcję lokalną (niekrytyczną).

Wzajemne wykluczanie

- **Wzajemne wykluczanie** → Instrukcje z sekcji krytycznych procesów nie mogą się przeplatać.
- **Brak zakleszczenia** → Jeśli pewne procesy próbują wejść do swoich sekcji krytycznych, to jednemu z nich musi się to w końcu udać.
- **Brak zagłodzenia** → Każdy proces, który chce wejść do sekcji krytycznej, w końcu do niej wejdzie.

Wzajemne wykluczanie

- Wykonywanie sekcji krytycznej musi cechować postęp, tzn. gdy dowolny proces rozpocznie wykonywanie instrukcji znajdujących się w niej, to w skończonym czasie zakończy ich wykonywanie.
- Wykonywanie sekcji lokalnych nie musi odbywać się z postępem (proces wykonujący sekcję lokalną może się zakończyć lub wejść do pętli nieskończonej i nigdy nie zakończyć jej wykonywania).

Semafory

- **Semafor** - struktura danych o dwóch polach:
  - *w* – pole przyjmujące nieujemne wartości całkowite,
  - *q* – kolejka procesów.
- Semafor pozwala rozwiązywać problemy synchronizacji procesów współbieżnych.

Semafory

- Na semaforze można wykonywać tylko dwie operacje:
  - *wait* – oczekiwanie,
  - *signal* – sygnalizowanie.
- Operacje *wait* oraz *signal* są operacjami atomowymi (tj. nie mogą być wykonywane w przeplocie z żadnymi innymi instrukcjami).

Semafory

- Operacja oczekiwania wstrzymuje wykonywanie procesu oraz:
  - umieszcza go w kolejce *q* oczekujących jeśli wartość *w* jest niedodatnia,
  - zmniejsza wartość *w* o 1 w przeciwnym razie.
- Operacja sygnalizowania:
  - wznawia wykonywanie procesu z kolejki *q* jeśli jest ona niepusta,
  - zwiększa wartość *w* o 1 w przeciwnym razie.

Semafory

- Przed korzystaniem z semafora:
  - pole *w* jest inicjowane dowolną nieujemną wartością całkowitą.
  - kolejka *q* jest pusta.

- Wykorzystanie w semaforze kolejki FIFO zapobiega zagłodzeniu procesów.

Semafory

- Semafor ogólny – semafor, w którym pole *w* może przyjmować dowolną nieujemną wartość całkowitą.
- Semafor binarny (muteks) – semafor, w którym pole *w* może przyjmować jedynie wartość 0 lub 1.

Semafory

- Semafor silny – jeśli operacja sygnalizowania *signal* wykonywana jest na semaforze nieskończoną liczbę razy, to w końcu każdy czekający proces zakończy wykonywanie operacji *wait*.
- Semafor słaby – jeśli wartość zmiennej związanej z semaforem jest stale dodatnia, to w końcu każdy czekający proces zakończy wykonywanie operacji *wait*.
- Semafor z kolejką FIFO jest semaforem silnym.

Semafory

- Niewłaściwa kolejność wykonywania operacji *wait* i *signal*, bądź pominięcie którejś z nich może spowodować blokadę lub niepoprawne działanie programu współbieżnego.
- Lokalizacja tego typu błędów, szczególnie w dużych programach współbieżnych, jest bardzo trudna.

Semafor w problemie wzajemnego wykluczania

Proces *A*	Proces *B*

pętla
{
  sekcja lokalna
  semafor.wait()
  sekcja krytyczna
  semafor.signal()
}
	pętla
	{
	  sekcja lokalna
	  semafor.wait()
	  sekcja krytyczna
	  semafor.signal()
	}

Analogicznie wygląda sytuacja dla *N* procesów.
Spełnione zostają warunki:
- wzajemnego wykluczania,
- braku zakleszczenia.

Jednak w tym przypadku może dojść do zagłodzenia.

Monitory

- **Monitor** – specjalny moduł lub pakiet zawierający struktury danych oraz procedury operujące na tych strukturach danych.
- Monitor pozwala rozwiązywać problemy synchronizacji procesów współbieżnych.
- Monitor zapewnia większe bezpieczeństwo niż semafor.
- Procesy współbieżne mogą modyfikować struktury danych monitora tylko przez wywołania procedur, nie mogą tego robić bezpośrednio.

Monitory

- Dostęp do struktur danych monitora odbywa się z wzajemnym wykluczaniem procesów. W danej chwili tylko jeden proces może wykonywać procedurę monitora.
- Proces wykonujący procedurę monitora nazywany jest procesem aktywnym wewnątrz monitora.

Monitory

- Jeśli w procesie *A* wywołana zostaje procedura monitora, a wewnątrz monitora aktywny jest proces *B*, to wykonywanie procesu zostaje wstrzymane do momentu aż proces *B* opuści monitor.
- Wykonywanie procedur monitora zachodzi ze wzajemnym wykluczaniem procesów.

Monitory

- Do synchronizacji w monitorze wykorzystywane są tzw. zmienne warunkowe (ang. *conditional variables*) oraz operacje oczekiwania *wait* i sygnalizowania *signal* działające na tych zmiennych.
- Operacje *wait* i *signal* działają inaczej niż operacje o tych samych nazwach w semaforach.

Monitory

- Jeśli w procedurze monitora nie można kontynuować działania wskutek niespełnienia jakiegoś warunku, to wykonuje się operację *wait* na zmiennej *a* odpowiadającej temu warunkowi.
- Wykonanie tej operacji powoduje wstrzymanie procesu i skierowanie go do kolejki FIFO związanej ze zmienną *a*.

Monitory

- Wstrzymany proces przestaje być aktywny i nie blokuje dostępu do monitora.
- Pozwala to innemu procesowi wejść do monitora i wykonać działania, w wyniku których warunek związany ze zmienną *a* zostanie spełniony.
- Proces ten może wówczas zasygnalizować spełnienie warunku przez wykonanie operacji *signal* na zmiennej *a*.

Monitory

- Wznowienie procesu z kolejki związanej ze zmienną *a* może nastąpić tylko wtedy gdy proces sygnalizujący przestanie być aktywny (tj. opuści monitor).
- Trzeba mieć pewność, że w momencie opuszczania monitora sygnalizowany wcześniej warunek jest nadal spełniony. Zaleca się więc aby operacja sygnalizowania była ostatnią operacją wykonywaną w procedurze monitora.

Monitory

- Zmienne warunków mają inny charakter niż semafory.
- Zmiennym warunków nie przypisuje się wartości.
- Wykonanie przez proces operacji *wait* na zmiennej warunku powoduje skierowanie go do kolejki.
- Wykonanie przez proces operacji *signal* na zmiennej warunku powoduje wznowienie procesu z kolejki jeśli nie jest ona pusta.

Monitory a semafory

- Semafory są konstrukcjami niskiego poziomu.
- Monitory są konstrukcjami wyższego poziomu o większym stopniu strukturalizacji.

Semafor w pakiecie java.util.concurrent

- Semafor reprezentowany jest przez klasę **Semaphore**.

Operacja	Metoda klasy Semaphore
wait	**acquire**
signal	**release**

Semafor w pakiecie java.util.concurrent

- Konstruktory klasy **Semaphore**:
  - `Semaphore(int w)`
  - `Semaphore(int w, boolean f)`

  - *w* - początkowa wartość licznika określającego liczbę wątków, które mogą jednocześnie korzystać z sekcji krytycznej.
  - *f* - zmienna określająca sposób przydzielania dostępu do sekcji krytycznej:
    - *true* - przyznawanie dostępu w kolejności w jakiej pojawiły się żądania dostępu,
    - *false* - kolejność dostępu nieokreślona.

Semafor w pakiecie java.util.concurrent

- Wybrane metody klasy **Semaphore**:
  - `acquire() throws InterruptedException`
  - `release()`

Monitory w języku Java

- Język Java nie posiada specjalnej konstrukcji monitorowej.
- Każdy obiekt posiada klucz, który może zostać wykorzystany do synchronizacji dostępu do pól tego obiektu.
- Zdefiniowanie metody synchronizowanej (**synchronized**) powoduje, że wątek musi zdobyć klucz do obiektu przed jej wykonaniem.

Monitory w języku Java

- Z metody synchronizowanej można wywołać inną metodę synchronizowaną danego obiektu i wówczas wątek zachowuje wówczas klucz do tego obiektu.
- Po powrocie z ostatniej metody synchronizowanej wywołanej na danym obiekcie wątek oddaje klucz do tego obiektu.
- Brak jest zasady uczciwości, tj. klucz zdobędzie dowolny z wątków rywalizujących o wywołanie metody synchronizowanej.

Monitory w języku Java

- Wstrzymanie wykonywania wątku w celu oczekiwania na określony stan danego obiektu odbywa się przez wywołanie metody **wait**.
- Metody **notify** oraz **notifyAll** przypominają operacje *signal* monitora.

Monitory w języku Java

- Metoda **notify** zwalnia jeden (dowolny) wątek ze zbioru wątków wstrzymanych.
- Metoda **notifyAll** zwalnia wszystkie wątki ze zbioru wątków wstrzymanych.
- Zanim wątki wznowione będą mogły się ponownie wykonywać muszą ponownie zdobyć klucz (oczywiście pojedynczo).
- Wątki wznowione nie mają pierwszeństwa przed innymi wątkami próbującymi zdobyć klucz.

Monitory w języku Java

- Wątek wznowiony nie może oczekiwać, że warunek, na który oczekiwał, jest dalej prawdziwy, musi więc go ponownie sprawdzić.

```
synchronized metoda()
{
   while (!warunek)
   { obiekt.wait(); }
   // ...
}
```

Blokady w pakiecie java.util.concurrent.locks

- Pakiet **java.util.concurrent.locks** udostępnia blokady - obiekty stanowiące alternatywę dla metod synchronizowanych.
- Ogólne działanie blokady:
  - Zanim możliwy będzie dostęp do współdzielonego zasobu, włączana jest blokada, która go chroni.
  - Gdy zakończą się operacje na tym zasobie, blokada zostaje zwolniona.
  - Jeśli w międzyczasie inny wątek próbuje uzyskać dostęp do zasobu, zostaje on zawieszony do momentu zwolnienia blokady.

Blokady w pakiecie java.util.concurrent.locks

- Interfejs **Lock** służy do implementacji blokad.
- Wybrane metody interfejsu **Lock**:
  - **`void lock()`** - zakłada blokadę.
  - **`boolean tryLock()`** - próbuje założyć blokadę bez czekania aż to będzie możliwe, zwraca *true* jeśli blokada została założona albo *false* w przeciwnym razie.
  - **`Condition newCondition()`** - zwraca obiekt typu **Condition** powiązany z blokadą.
  - **`void unlock()`** - zwalnia blokadę.
- Klasa **ReentrantLock** implementuje interfejs **Lock**.

Blokady w pakiecie java.util.concurrent.locks

- Interfejs **ReadWriteLock** służy do implementacji blokad zawierających osobne mechanizmy blokowania dostępu do zapisu i odczytu.
- W tym przypadku możliwe jest udostępnianie zasobu do odczytu dla kilku wątków, jeśli zasób nie jest w danym momencie zapisywany.
- Klasa **ReentrantReadWriteLock** implementuje interfejs **ReadWriteLock**.