Java – wątki

Podstawowe pojęcia

Proces to egzemplarz wykonywanego programu wraz z dynamicznie przydzielonymi mu przez system zasobami (np. pamięcią operacyjną, zasobami plikowymi, dostępem do urządzeń wejścia / wyjścia). Każdy proces ma swoją własną przestrzeń adresową.

Systemy wielozadaniowe pozwalają na równoległe wykonywanie wielu procesów, z których każdy ma swój kontekst i swoje zasoby.

Wątek to sekwencja działań, która wykonuje się w kontekście danego procesu (programu). Każdy proces ma co najmniej jeden wykonujący się wątek. W systemach wielowątkowych proces może wykonywać równolegle wiele wątków, które wykonują się w jednej przestrzeni adresowej procesu.

Podstawowe pojęcia

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na kwant czasu, po czym „oddaje procesor” innemu wątkowi.

Zmiana wątku :

- **współpraca** (*cooperative multitasking*) – wątek decyduje, kiedy oddać czas procesora innym wątkom,
- **wywłaszczanie** (*pre-emptive multitasking*) – systemowy zarządca wątków przydziela wątkowi kwant czasu procesora, po upływie którego odsuwa wątek od procesora i przydziela kolejny kwant czasu innemu wątkowi.

Tworzenie wątków

Uruchamianie i zarządzanie wątkami – klasa ***Thread.***

Uruchomienie wątku:

- Tworzymy obiekt klasy ***Thread***
- Wywołujemy na rzecz tego obiektu metodę ***start()***

Kod wykonujący się jako wątek określany jest przez obiekt implementujący interfejs ***Runnable***, który zawiera deklarację metody ***run()***.

Metoda ***run()*** określa, co ma robić wątek.

Klasa, której obiekty mają działać jako wątki musi zawierać metodę: ***public void run()*.**

Klasa *Thread* – metody składowe

Uruchamianie i zatrzymywanie:

start() – uruchomienie wątku,

stop() – zakończenie wątku (niezalecane),

run() – kod wykonywany w ramach wątku.

Identyfikacja:

currentThread() – zwraca identyfikator bieżącego wątku,

setName() – ustawienie nazwy wątku,

getName() – odczytanie nazwy wątku,

isAlive() – sprawdzenie, czy wątek działa,

toString() – uzyskanie atrybutów wątku.

Priorytety i szeregowanie:

getPriority() – odczytanie priorytetu wątku,

setPriority() – ustawienie priorytetu wątku,

yield() – wywołanie szeregowania.

Klasa Thread – metody składowe

Synchronizacja:

sleep() – zawieszenie wykonania wątku na dany czas,

join() – oczekiwanie na zakończenie innego wątku,

wait() – oczekiwanie w monitorze,

notify() – odblokowanie wątku zablokowanego na monitorze,

notifyAll() – odblokowanie wszystkich wątków zablokowanych na monitorze,

interrupt() – odblokowanie zawieszonego wątku,

suspend() – zablokowanie wątku,

resume() – odblokowanie wątku zawieszonego przez ***suspend***,

setDaemon() – ustanowienie wątku demonem,

isDaemon() – testowanie czy wątek jest demonem

Tworzenie wątków

Dwa sposoby tworzenia nowego wątku:

- poprzez dziedziczenie klasy ***Thread***
- poprzez implementację interfejsu ***Runnable***.

Drugi sposób – gdy klasa reprezentująca wątek dziedziczy po klasie innej niż ***Thread.***

Tworzenie wątku – sposób I

1. Tworzymy nową klasę dziedziczącą z klasy ***Thread***
2. Nadpisujemy metodę ***run()*** – kod do wykonania w ramach tworzonego wątku.

   ```
   class MyThread extends Thread {
   ....
   void run() {
   // kod wątku
   }
   ....
   }
   ```

3. Tworzymy obiekt nowej klasy ***MyThread*** (na przykład ***thr***):

   ```
   MyThread thr = new MyThread(...);
   ```

4. Wykonujemy metodę ***start()*** klasy ***MyThread*** odziedziczoną z klasy ***Thread***:

   ```
   thr.start();
   ```

Tworzenie wątku – sposób II

1. Tworzymy nową klasę implementującą interfejs ***Runnable***.
   W ramach tej klasy tworzymy metodę ***run()*** – czynności do wykonania w ramach wątku.

   ```
   class MyRunnable implements Runnable {
   public void run() {
   // kod wątku
   } }
   ```

2. Tworzymy obiekt nowej klasy

   ```
   MyRunnable r = new MyRunnable();
   ```

3. Tworzymy obiekt klasy ***Thread*** przekazując obiekt ***r*** jako parametr konstruktora klasy ***Thread***.

   ```
   Thread thr = new Thread(r);
   ```

4. Uruchamiamy wątek metodą ***start()*** klasy ***Thread***.

   ```
   thr.start();
   ```

Zakończenie pracy wątku

Wątek kończy pracę gdy zakończy się jego metoda ***run()***.

Istnieje **niezalecana** metoda ***stop()*** zatrzymująca działanie wątku. Wywołując tę metodę nie wiemy w jakim stanie jest zatrzymywany wątek – może pozostawić obiekt w nieprawidłowym stanie, jeśli przerwaliśmy wykonywanie jakiejś krytycznej operacji.

Jeśli chcemy programowo zakończyć pracę wątku, powinniśmy zapewnić w metodzie ***run()*** sprawdzenie warunku zakończenia i jeśli jest on spełniony, spowodować wyjście z metody ***run()***.

Stany wątków

- **utworzony** (*new thread*) – obiekt wątku został utworzony ale nie wykonano jeszcze metody ***start()*** (wątek nie jest jeszcze szeregowany)
- **wykonywalny** (*runnable*) – wątek posiada wszystkie zasoby, może być wykonywany, będzie wykonywany gdy tylko procedura szeregująca przydzieli mu czas procesora
- **zablokowany** (*blocked*) – wątek nie może być wykonywany ponieważ brakuje mu pewnych zasobów. Dotyczy to w szczególności operacji synchronizacyjnych (wątek zablokowany na wejściu do monitora, operacje ***wait***, ***sleep***, ***join***) i operacji wejścia-wyjścia
- ***zakończony*** (*dead*) – stan po wykonaniu metody ***stop()***. Zalecanym sposobem kończenia wątku jest zakończenie metody ***run()***.

Stany wątków

Przejście **wykonywalny -> zablokowany**:

- wątek chce wejść do zablokowanego monitora
- wykonana została metoda ***wait()***, ***join()***, ***suspend()***
- wywołano metodę ***sleep(...)***
- wątek wykonał operacje wejścia / wyjścia.

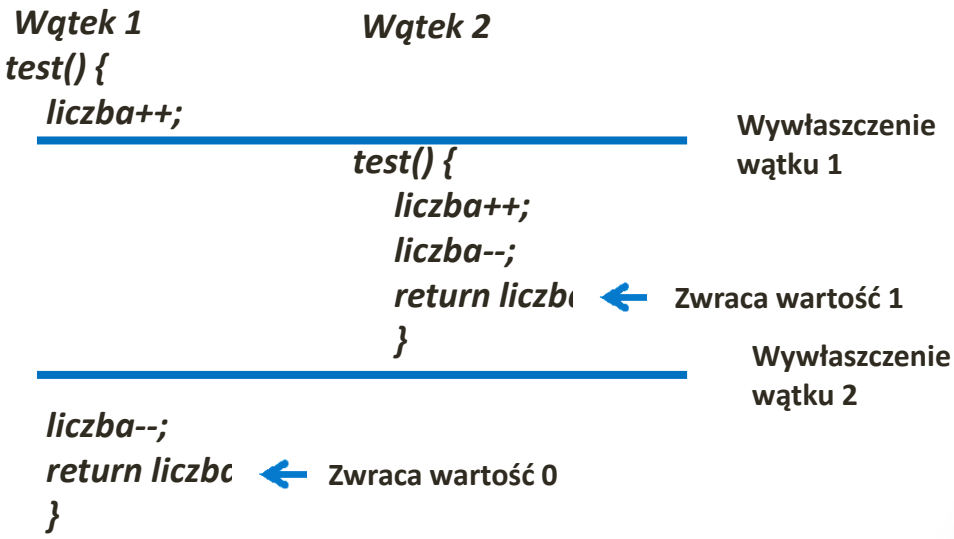
Przejście **zablokowany** -> **wykonywalny**:

- monitor został odblokowany
- inny wątek odblokował zablokowy wątek – wywołał metodę ***notify()***, ***notifyAll()***, ***resume()***, ***interrupt()***
- wątek zakończył wykonywanie metody ***sleep()*** – upłynął zadany czas
- wątek czekał na zakończenie operacji wejścia / wyjścia i operacja ta zakończyła się.

Synchronizacja wątków

Wątki korzystające ze współdzielonych zmiennych mogą być wywłaszczone w trakcie operacji (nawet pojedynczej) – stan współdzielonej zmiennej może okazać się niespójny.

```
class Liczba{
  private int liczba= 0;
  public int test() {
    liczba++;
    liczba--;
    return liczba;
    }
}
```



Synchronizacja wątków

Komunikacja między wątkami opiera się na wspólnej pamięci. W takim przypadku występuje zjawisko wyścigów (*race conditions*) – wynik działania procedur wykonywanych przez wątki zależy od kolejności ich wykonania.

Gdy kilka wątków ma dostęp do wspólnych danych i przynajmniej jeden je modyfikuje występuje konieczność synchronizowania dostępu do wspólnych danych.

Synchronizacja – mechanizm, który zapewnia, że kilka wątków wykonujących się w tym samym czasie:

- nie będzie równocześnie działać na tym samym obiekcie,
- nie będzie równocześnie wykonywać tego samego kodu.

Kod, który może być wykonywany w danym momencie tylko przez jeden wątek, nazywa się **sekcją krytyczną**. W Javie sekcje krytyczne wprowadza się jako bloki lub metody synchronizowane.

Synchronizacja wątków

Każdy egzemplarz klasy ***Object*** i jej podklas posiada **monitor** (ang. *lock*), który ogranicza dostęp do obiektu. Blokowanie obiektów jest sterowane słowem kluczowym ***synchronized***.

Synchronizacja w Javie może być wykonana na poziomie:

- metod – słowo kluczowe ***synchronized*** przy definiowaniu metody:

```
public synchronized int test()
{...}
```

- instrukcji – słowo kluczowe ***synchronized*** przy definiowaniu bloku instrukcji:

```
synchronized( liczba)
{ liczba++;
  liczba--;
}
```

Synchronizacja wątków

Kiedy wątek wywołuje na rzecz jakiegoś obiektu metodę synchronizowaną, automatycznie zamykany jest monitor (obiekt jest zajmowany przez wątek). Inne wątki usiłujące wywołać na rzecz tego obiektu metodę synchronizowaną (niekoniecznie tę samą) lub usiłujące wykonać instrukcje ***synchronized*** z podaną referencją do zajętego obiektu są blokowane i czekają na zakończenie wykonywania metody lub instrukcji ***synchronized*** przez wątek, który zajął obiekt (zamknął monitor).

Dowolne zakończenie wykonywania metody synchronizowanej lub instrukcji ***synchronized*** zwalnia monitor, dając czekającym wątkom możliwość dostępu do obiektu.

Koordynacja wątków

Koordynacja wątków polega na zapewnieniu właściwej kolejności działań wykonywanych przez różne wątki na wspólnym zasobie. Do koordynacji wątków stosuje się metody: ***wait()***, ***notify()***, ***notifyAll().***

Metoda *wait()*

public final void wait();

public final void wait(long timeout);

public final void wait(long timeout,int nanos)

throws InterruptedException

Wykonanie metody powoduje zawieszenie bieżącego wątku do czasu, gdy inny wątek nie wykona metody ***notify()*** lub ***notifyAll()*** odnoszącej się do wątku, który wykonał ***wait()***. Wątek wykonujący ***wait(...)*** musi być w posiadaniu monitora dotyczącego synchronizowanego obiektu. Wykonanie ***wait(...)*** powoduje zwolnienie monitora.

Metoda *notify()*

public final void notify();

Metoda powoduje odblokowanie jednego z wątków zablokowanych na monitorze pewnego obiektu poprzez ***wait()***. Definicja metody nie określa, który z czekających wątków zostanie odblokowany.

Odblokowany wątek nie będzie natychmiast wykonywany – musi on jeszcze zaczekać aż bieżący wątek zwolni blokadę monitora. Odblokowany wątek będzie konkurował z innymi o nabycie blokady monitora. Metoda może być wykonana tylko przez wątek, który jest właścicielem – zajmuje monitor obiektu.

Metoda *notifyAll()*

public final void notifyAll()

Metoda powoduje odblokowanie wszystkich wątków zablokowanych na monitorze pewnego obiektu poprzez uprzednie wykonanie ***wait()***.

Wątki będą jednak czekały aż wątek bieżący nie zwolni blokady monitora. Odblokowane wątki będą konkurowały o nabycie blokady monitora.

Synchronizacja wątków

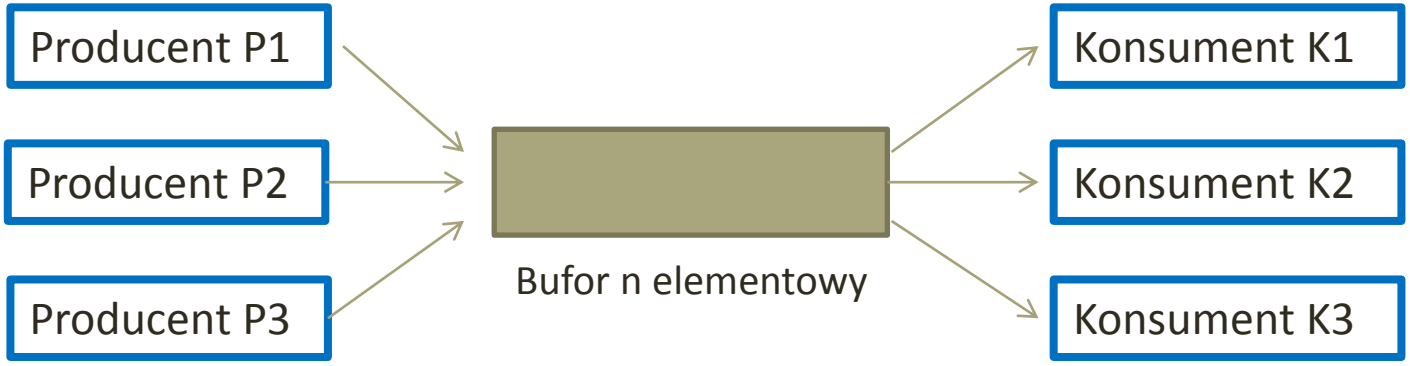
Synchronizacja bloku jest tylko częściową blokadą obiektu, na którym jest założona.

Inne metody mogą korzystać z synchronizowanego obiektu o ile nie usiłują synchronizować się na tym obiekcie.

Nie zawsze synchronizacja jest najlepszym rozwiązaniem i często warto pokusić się o rozwiązania alternatywne:

- używanie zmiennych lokalnych (każdy wątek działa na własnym zestawie zmiennych lokalnych),
- używanie metod działających na argumentach typów prostych (przekazywanych przez wartość) lub bezargumentowych,
- tworzenie klas niezmiennych (deklaracja wszystkich pól jako prywatnych, bez pisania metod modyfikujących pola klasy).

Problem producent - konsument



Kilku producentów i konsumentów korzysta ze wspólnego bufora.
Każdy producent co pewien czas generuje dane i umieszcza je w buforze.
Każdy konsument co pewien czas pobiera dane z bufora.

Priorytety

Każdy wątek ma priorytet – w Javie jest to liczba całkowita z zakresu od 1 do 10.

Wątek o priorytecie 10 będzie wykonywany w pierwszej kolejności, wątek o priorytecie 1 – w ostatniej.

Gdy gotowych do wykonania jest wiele wątków, maszyna wirtualna Javy uruchamia tylko ten o najwyższym priorytecie.

Domyślnym priorytetem dla tworzonych wątków jest 5.

Priorytety wątków są przydatne, gdy chcemy niektórym z nich przydzielić więcej czasu procesora (np. wątki do komunikacji z użytkownikami – krótki czas odpowiedzi), a niektórym mniej (np. wątki dokonujące obliczeń, wątki długotrwałe – brak wymagania co do prędkości realizacji).

Pola i metody klasy **Thread** związane z priorytetami:

public static final int MIN_PRIORITY = 1

public static final int NORM_PRIORITY = 5

public static final int MAX_PRIORITY = 10

public final void setPriority(int newPriority)

public final int getPriority()