Programowanie komputerów

Wykład 7: „Programowanie wielowątkowe w Javie”

dr inż. Walery Susłow

Współbieżność

- Programy współbieżne (concurrent software) – aplikacje potrafiące wykonywać kilka operacji w tym samym czasie, z uwzględnieniem zdarzeń priorytetowych. Przykład: reakcja na zdarzenie myszki/klawiatury edytora tekstu, który automatycznie formatuje tekst.
- Java została zaprojektowana do tworzenia programów współbieżnych dostarczając odpowiedni zbiór klas.
- Java dostarcza także wysokopoziomowe API do obsługi współbieżności

Problemy związane z programowaniem współbieżnym

- Safety: bezpieczeństwo programu.
- Livness: możliwość wystąpienia zakleszczenia.
- Niedeterministyczne zachowanie.
- Starty mocy obliczeniowej na przełączania pomiędzy wątkami na procesorze i synchronizację wątków.

Wątki i procesy

- Wątki i procesy są podstawowymi pojęciami i jednostkami wykonawczymi.
- W Javie jednostką wykonawczą są wątki (threads).
- System operacyjny posiada wiele wątków i procesów, które dzielą czas procesora.

Procesy

- Proces uzyskuje własne środowisko wykonawcze (execution environment), czyli zbiór zasobów przyporządkowanych (np. pamięć) na czas wykonania.
- Proces może być postrzegany jako pojedynczy program lub aplikacja.
- Procesy mogą się komunikować (Inter Process Communication, IPC).
- IPC służy również do komunikacji pomiędzy procesami umieszczonymi na różnych maszynach.
- JVM uruchamiana jest przeważnie jako pojedynczy proces, ale aplikacje mogą tworzyć nowe procesy.

Wątki (1)

- Wątki nazywane są czasami „lekkimi procesami”.
- Wątek, podobnie jak proces, ma do wyłączonej dyspozycji zasoby, natomiast zasoby nie są przyznawane tak suwerennie jak w przypadku procesu, np. używają wspólnej sekcji danych, zamiast własnej przestrzeni adresowej.
- Wątki działają w ramach procesu, współdzielą pliki i pamięć.

Wątki (2)

- Każda z aplikacji Java posiada co najmniej jeden wątek główny oraz kilka wątków pomocniczych (zarządzanie pamięcią, obsługa zdarzeń).
- Główny wątek aplikacji ma zdolność do tworzenia nowych wątków.
- Równoległe wykonywanie wątków jest możliwe jedynie na maszynach wieloprocesorowych, na jednoprocesorowych wątki wykonywane są współbieżnie.

Obiekt Thread (1)

- Każdy wątek jest skojarzony z instancją klasy Thread.
- Dwa podejścia do tworzenia współbieżnych aplikacji:
  - Bezpośrednie tworzenie wątku i zarządzanie nim poprzez utworzenie instancji klasy Thread.
  - Oddzielenie zarządzania wątkami od aplikacji poprzez umieszczenie ich w osobnym obiekcie (Executor).

Obiekt Thread (2)

- Klasa Thread udostępnia szereg metod m.in. statyczne, które pozwalają na uzyskanie informacji o wątku.
- http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html

Obiekt Thread (3)

Pola obiektu Thread:

static int MAX_PRIORITY

static int MIN_PRIORITY

static int NORM_PRIORITY

Konstruktory obiektu Thread:

Thread()

Thread(Runnable target)

Thread(Runnable target, String name)

Thread(String name)

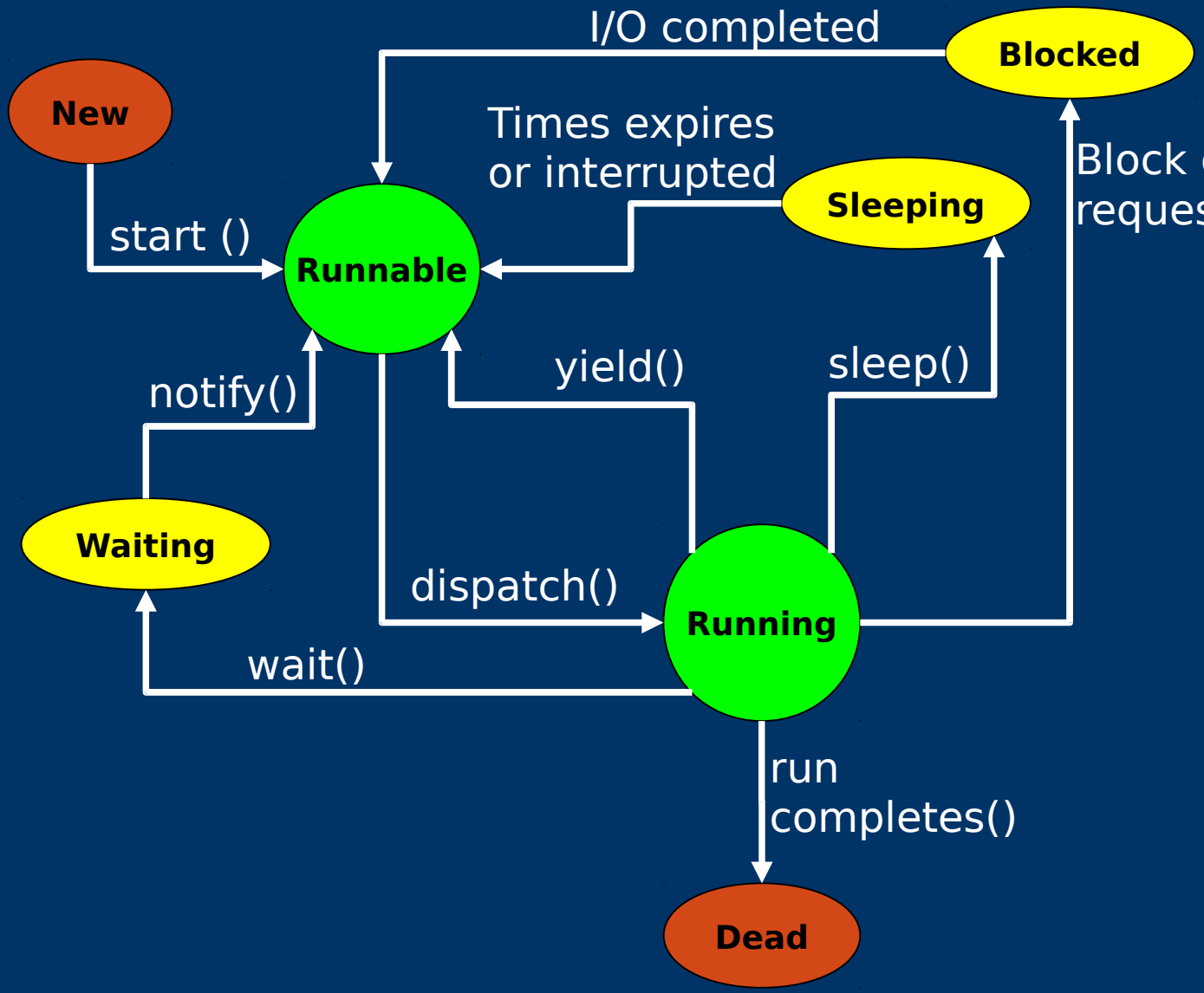
Thread(ThreadGroup group, Runnable target)

Thread(ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Thread(ThreadGroup group, String name)

Cykl życia wątku



New – nowy wątek, bezpośrednio, po wywołaniu konstruktora Thread.

Runnable – zachodzi po wykonaniu metody start().
Running – wykonywany jest kod wątku z metody run().

Not runnable – wątek zwalania zasoby oraz:
Blocked - oczekuje na zakończenie operacji wejścia/wyjścia;
Sleeping - oczekuje na upłynięcie określonego czasu;
Waiting - oczekuje na wywołania przez inny wątek.

Przykład 1. Dwa podejścia tworzenia wątków

Implementacja interfejsu Runnable

```
public class HelloRunnable
   implements Runnable {

   public void run()   {
System.out.println(„Pierwszy
  wđtek");
   }

   public static void
main(String args[]) {
   (new Thread(new
HelloRunnable())).start();
   }
}
```

Dziedziczenie z klasy Thread

```
public class HelloThread
   extends Thread {

    public void run() {
 System.out.println("Drugi
wątek");
    }

    public static void
main(String args[]) {
        (new
HelloThread()).start();
    }
}
```

Obydwa podejścia wymagają wystartowania wątku: Thread.start

Zatrzymywanie wątków

- Zatrzymywanie wątków realizowane jest poprzez Thread.sleep() na określony czas podawany w mili- lub nanosekundach.
- Brak 100% pewności, że wątek zostanie zatrzymany na precyzyjnie określony czas.
- Wydajny sposób na oddanie czasu procesora innym wątkom bądź procesom.
- Może zostać wykorzystane do oczekiwania na pojawienie się odpowiedniego stanu innego wątku.

Metody wait() i notify()

Wywołanie metod wait() i notify() może odbywać się wyłączenie w ramach bloku, który jest zsynchronizowany.

wait():

- zatrzymanie aktualnego wątku,
- interpreter umieszcza wątek w kolejce związanej z obiektem,
- blokada synchronizacyjna jest zwalniana dla tego obiektu.

notify():

- blokada synchronizacyjna zostaje zwolniona dla obiektu.
- Jeśli istnieje wątek związany z obiektem, żądający wykonania bloku synchronizowanego, uzyskuje on monit i jest usuwany z kolejki.
- Wątek związany jest wznawiany za wywołaniem wait(), które spowodowało zatrzymanie wątku pierwszego.

Przerwania (Interrupts)

- Przerwanie – zatrzymuje natychmiast wykonanie zadania przez wątek i zleca mu nowe zadanie.
- Zachowanie się wątku po wykonaniu przerwania definiuje programista, natomiast standardowo przyjęło się, że następuje „zabicie” wątku.
- Metoda sleep może rzucić wyjątkiem *InterruptedException*. Metody, które posługują się takim wyjątkiem przeważnie zaprojektowane zostały w ten sposób, aby kończyć działanie wątku.
- Mechanizm przerwań bazuje na fladze *interrupt status*, która jest ustawiana w momencie wywołania metody *Thread.interrupt()*.
- Sprawdzenie statusu flagi realizowane za pomocą metody *Thread.isInterrupted()*.

Metoda join()

- Metoda ta pozwala oczekiwać jednemu wątkowi na ukończenie wykonywania zadania przez drugi.
- Wywołanie metody T.join() w ciele danego wątku, gdzie t jest innym wątkiem powoduje zatrzymanie wykonywania danego wątku do momentu zakończenia wykonywania wątku T.
- Przeładowanie join() pozwala określić np. czas oczekiwania na wątek.
- join() odpowiada na wyjątek InterruptException.

Priorytety

- Wątki są ustawiane do uruchomienia w kolejce i każdemu z nich przypisywany jest priorytet od 1 do 10.
- Thread.NORM_PRIORITY - standardowy priorytet, który posiada wartość 5.
- Metoda *getPriority()* pozwala na pobranie priorytetu.

Synchronizacja

- Wątki komunikują się poprzez udostępnianie sobie pól obiektów. Jest to bardzo wydajna forma komunikacji natomiast może powodować błędy ze spójnością pamięci.
- Narzędziem do ochrony przed problemami związanymi ze współdzieleniem zasobów przez wątki jest synchronizacja.

Interferencja pomiędzy wątkami

- Stan: dwie metody wykonywane są w dwóch różnych wątkach na tym samym zbiorze danych.
- Problem: przeplatanie wątków, nawet pojedyncze wyrażenia mogą być wykonane etapowo przez JVM.

```
class Counter {
   private int c = 0;
   public void increment() {
      c++;
   }
   public void decrement() {
      c--;
   }
   public int value() {
      return c;
   }
}
```

Niespójność pamięci

- Niespójność pamięci pojawia się w momencie, gdy jeden z wątków odwołuje się do danych zmienionych bez jego wiedzy i traktuje te dane jako poprawne.
- Relacja *happen-before* pozwala na ustalenie dostępu do zasobu.
- Relacja *happen-before* tworzona jest w przypadku synchronizacji.

Metody synchronizacji

Synchronizacja może być na dwóch poziomach:

- Synchronizacja metody.
- Synchronizacja wyrażenia.

Konstruktor nie może być synchronizowany ponieważ w trakcie tworzenia obiektu, tylko jeden wątek tym się zajmuje.

Możliwe jest wykluczanie części kodu z pozostałymi metodami klasy po przez *synchronized(this){ … }*

```
public class Licznik2 {
   private int c = 0;
   public synchronized
              void increment() {
      c++;
   }
   public synchronized
              void decrement() {
      c--;
   }
   public synchronized
              int value() {
      return c;
   }
}
```

Żywotność

- Zakleszczenie (*deadlock*) – dwa lub więcej wątków zostają zablokowane – każdy czeka na każdy.
- Zagłodzenie (*starvation*) – sytuacja, w której dany wątek nie może otrzymać dostępu do zasobów i wykonać swojego zadania.
- *Livelock* – bardzo podobne do deadlock. Wątki nie zostają zatrzymane, natomiast nie wykonują postępów.

Wątek demon

- Wątek demon – nazywany wątkiem usługowym, działa na niskim priorytecie.
- Wątek demon pełni funkcje usługową względem innych wątków.
- Przykładem wątku demona jest *garbage collector*, który działa w tle non-stop.
- Wątek może ustawić flagę demona. W przypadku, gdy flaga ustawiona jest na false – wątek jest wątkiem użytkownika. Ustawienie flagi wątku musi być realizowane przed wystartowaniem wątku.

Zaawansowane sposoby synchronizacji

- Synchronizacja realizowana na monitach (blokowanie) – posiada szereg ograniczeń.
- Bardziej zaawansowane sposoby blokowania realizowane są przez pakiet: *java.util.concurrency.locks*.
- Przykładowy obiekt: Lock.
  - Lock zachowuje się podobnie jak blokada realizowana przez *synchronized()*.
  - Tylko jeden obiekt może trzymać obiekt Lock.
  - Lock wspiera mechanizmy notify()/wait().
  - Przewagą Lock jest możliwość cofnięcia żądania przejęcia zasobów, w przypadku, gdy są one niedostępne.