Programowanie obiektowe

Wykład 13: Wątki

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.3
4 marca 2021

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska
Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca" współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych", realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca", współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

Wielowątkowość

- Aplikacja, szczególnie wyposażona w interfejs użytkownika, musi wykonywać kilka zadań równocześnie.
  - Pobieranie grafiki do wyświetlenia na ekranie.
- W przypadku maszyny wielordzeniowej można realizować równocześnie różne zadania.
- W przypadku maszyny jednoprocesowej jednoczesne wykonywanie różnych zadań jest symulowane przez sekwencyjne wykonywanie fragmentów zadań.
- W obydwu wypadkach podział wykonywania aplikacji na zadania nie odbywa się automatycznie i musi zostać świadomie zainicjowany.
- Java wykorzystuje do tego mechanizm wątków.

Proces a wątek

- W momencie uruchamiania aplikacji na maszynie wirtualnej Javy uruchamiamy proces.
- Każda aplikacja tworzy nowy proces, który ma własne, wydzielone zasoby.
- Wątki są uruchamiane w ramach procesu i współdzielą jego zasoby.
- Uruchamianie wątków jest znacznie szybsze niż rozpoczynanie nowego procesu i wątki bywają nazywane *lekkimi procesami*.

Tworzenie wątku

- Wątek można utworzyć na dwa sposoby.

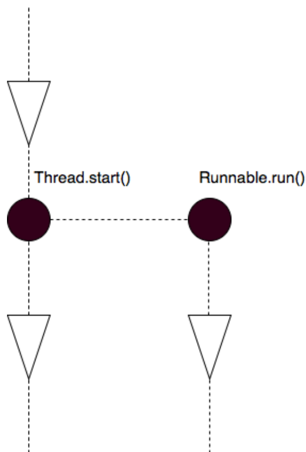
Nadpisanie klasy Thread

```
1 public class MyThread extends
      Thread {
2     @Override
3     public void run() {
4         //Zrób coś
5     }
6 }
7
8 MyThread thread = new MyThread();
9 thread.start();
```

Implementacja interfejsu Runnable

```
1 public class MyRunnableClass
      implements Runnable {
2     @Override
3     public void run() {
4         //Zrób coś
5     }
6 }
7
8 Thread thread = new Thread(new
      MyRunnableClass());
9 thread.start();
```

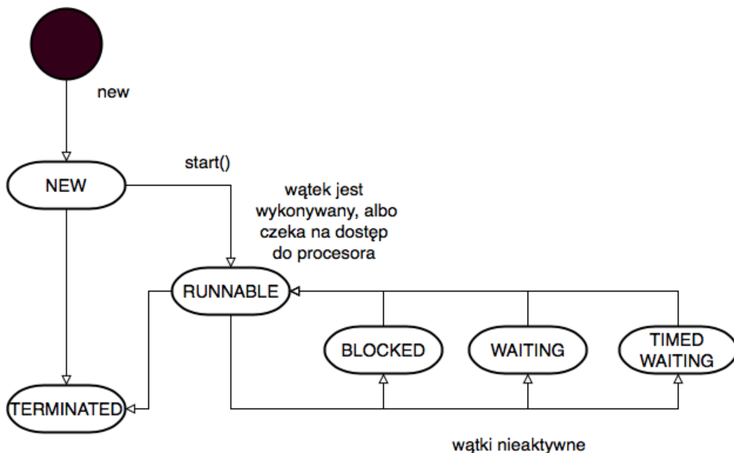
Uruchamianie wątku



Rysunek 1: Uruchamianie wątku

- Chociaż naszym celem jest uruchomienie kodu metody `run` to nie robimy tego bezpośrednio.
- Wywołanie metody `run` zakończyłoby się jej uruchomieniem w wątku głównym.
- Dopiero wywołanie metody `start` powoduje utworzenie nowego wątku i uruchomienie w nim kodu metody `run`.

Stany wątku



Rysunek 2: Możliwe stany wątku

Zatrzymywanie wątku

- Właściwym zachowaniem jest pozwolenie wątkowi na samodzielne zakończenie jego działania.
- Istniejące metody `stop` i `suspend` pozwalają na zatrzymanie pracy wątku, ale nie powinny być używane.
- Można wywołać metodę `interrupt` aby zgłosić chęć zatrzymania wątku, ale wątek musi nasłuchiwać czy nie została ona wywołana.

Nasłuchiwanie wystąpienia żądania przerwania wątku

```
1  while (!Thread.currentThread().isInterrupted()) {
2    //krok programu
3  }
```

- Nie można przerwać w ten sposób działania wątków nieaktywnych

Typy wątków

- W programie wielowątkowym wyróżniamy:
  - Wątek główny, który powstaje w momencie uruchomienia aplikacji, a kończy się po wykonaniu kodu metody `main`.
  - Wątki pomocnicze, tworzone podczas wykonywania programu i kończące się po wykonaniu kodu metody `run`.
  - Demony, czyli wątki pomocnicze oznaczone poprzez wywołanie metody `setDaemon(true)`, które służą do wykonywania asynchronicznych działań pomocniczych
- Aplikacja kończy działanie, gdy jedynymi niezakończonymi wątkami są demony lub gdy nie ma działających wątków.

Priorytety wątku

- Wątek w stanie `RUNNABLE` dostaje dostęp do procesora tylko na pewien czas.
- Przydział czasu procesora zależy od implementacji JVM i jest nieprzewidywalny.
- Długość przyznanego czasu zależy od priorytetu wątku.
- Wątki z większym priorytetem blokują wątki z mniejszym priorytetem.
- Priorytet można ustalić funkcją `setPriority` w zakresie
  MIN_PRIORITY wartość 1
  NORM_PRIORITY wartość 5 (domyślna)
  MAX_PRIORITY wartość 10
- Rzeczywista liczb poziomów priorytetów zależy od systemu operacyjnego

Problemy związane z wątkami

- Praca z wątkami jest niezmiernie trudna, bo przebieg wykonywania programu przestaje być liniowy.
- Co więcej z wątkami powiązane są specyficzne problemy nie występujące zazwyczaj w programowaniu jednowątkowym.
  - Głodzenie.
  - Brudne pisanie i czytanie.
  - Zakleszczenie.

Głodzenie

- Załóżmy, że nasz obiekt ma do realizacji, w osobnych wątkach, następujące zadania.
  - jedzenie,
  - picie,
  - spanie.
  - nauka.
- Zakładamy, że pierwsze trzy zadania są najważniejsze i przyznajemy im maksymalny priorytet. Ostatnie zadanie dostaje znacznie niższy priorytet. Jaki będzie skutek?
- System będzie preferował pierwsze trzy zadania i ograniczał realizację czwartego zadania, co może spowodować, że nie zostanie ono zrealizowane.
- Proces ten nazywamy *głodzeniem* wątku. Można mu zapobiegać zwiększając z czasem priorytety głodzonych wątków.

Wypłata z bankomatu

- Rozważmy następującą procedurę wypłaty pieniędzy z bankomatu.

Algorytm wypłaty z bankomatu

1. autoryzacja,
2. odczyt salda,
3. wypłata,
4. aktualizacja salda.

- Załóżmy, że mamy na koncie 100 złotych. Przeprowadźmy wypłatę 100 złotych w tym samym czasie na dwóch bankomatach.
- Kroki algorytmu wykonywane na obydwu bankomatach będą realizowane w niedeterministycznie naprzemiennych blokach.

Równoległa wypłata z bankomatu

- Przed operacją - konto: 100; na ręce: 0.

Równoległa wypłata z dwóch bankomatów

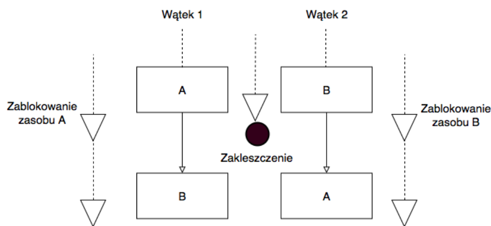
1. ATM1: autoryzacja,
2. ATM1: odczyt salda (100),
3. ATM2: autoryzacja,
4. ATM2: odczyt salda (100),
5. ATM1: wypłata (100),
6. ATM1: aktualizacja salda (0),
7. ATM2: wypłata (100),
8. ATM2: aktualizacja salda (0),

- Po operacji - konto: 0; na ręce: 200.

Brudne czytanie i pisanie

- Przedstawiony mechanizm to brudne czytanie i pisanie.
- Jeden z wątków odczytuje lub nadpisuje stan obiektu nie wiedząc, czy drugi nie wykonuje podobnych czynności w tym samym czasie.
- Mechanizmowi temu możemy zapobiegać blokując dostęp do aktualnie modyfikowanych elementów.
- Blokowanie może jednak prowadzić do kolejnego problemu *zakleszczenia*.

Zakleszczenie



Rysunek 3: Zakleszczenie dwóch wątków

- Zakleszczenie (*deadlock*) następuje gdy dwa lub więcej wątków blokują nawzajem potrzebne sobie zasoby.
- Wątek 1 zablokował zasób A i czeka na zasób B.
- Wątek 2 zablokował zasób B i czeka na zasób A.
- Żaden z wątków nie będzie mógł kontynuować pracy.
- W tym wypadku można uniknąć zakleszczenia, jeśli blokowanie zasobów będzie przebiegało kolejno.

Zabronione metody kontroli wątku

- Metoda `stop` niszczy wszystkie blokady założone przez wątek i zatrzymuje wszystkie wykonywane w nim metody.
    - Prowadzi do brudnego pisania i czytania, a nawet do utraty obiektów.
- Metoda `suspend` nie niszczy obiektów, ale powoduje, że nie da się usunąć blokad założonych przez zawieszony wątek.
    - Jeżeli metoda zawieszająca wątek chce skorzystać z zasobu, którego ten używał to nastąpi zakleszczenia.

Dozwolone metody kontroli wątku

sleep Zatrzymuje wykonywanie aktywnego wątku na pewien czas. Jeżeli inny wątek chce go zbudzić wywołuje wyjątek `InterruptedException`.

yield Aktualnie wykonywany wątek odstępuje dostęp do procesora. Zastąpi go wątek o nieniższym priorytecie.

join Przerywa aktualny wątek na czas wykonania wątku, którego metoda `join` została wywołana.

Przykład użycia metody sleep

Klasa MyTask

```
1   public class MyTask implements Runnable {
2
3       private String code;
4       private int sleepTime;
5
6       public MyTask(String code, int sleepTime) {
7           this.code = code;
8           this.sleepTime = sleepTime;
9       }
10
11      @Override
12      public void run() {
13          for(int i=0;i<10; i++){
14              System.out.println(code);
15              try {
16                  Thread.sleep(sleepTime);
17              } catch (InterruptedException e) {
18                  e.printStackTrace();
19              }
20          }
21      }
22  }
```

Przykład użycia metody join

Klasa MyJoinTask

```
1   public class MyJoinTask implements Runnable {
2
3       private Thread toBeJoined;
4
5       public MyJoinTask(Thread toBeJoined) {
6           this.toBeJoined = toBeJoined;
7       }
8
9       @Override
10      public void run() {
11          System.out.println("Join task is starting.");
12          try {
13              toBeJoined.join();
14          } catch (InterruptedException e) {
15              e.printStackTrace();
16          }
17          System.out.println("Join task is resuming.");
18      }
19  }
```

Zależności między wątkami

Klasa MyTasksExample

```
1   public class MyTasksExample {
2     public static void main(String[] args) {
3       System.out.println("Main thread is starting.");
4       Thread threadA = new Thread(new
            MyTask("A",1000));
5       Thread threadB = new Thread(new
            MyTask("B",500));
6
7       threadA.start();
8       threadB.start();
9
10      try {
11        Thread.sleep(3000);
12      } catch (InterruptedException e) {
13        e.printStackTrace();
14      }
15
16      Thread joinedThread = new Thread(new
            MyJoinTask(threadA));
17      joinedThread.start();
18      System.out.println("Main thread is ending.");
19    }
20  }
```

Wynik

```
Main thread is starting.
B0
A0
B1
A1
B2
B3
A2
B4
B5
Main thread is ending.
Join task is starting.
A3
B6
B7
A4
B8
B9
A5
A6
A7
A8
A9
Join task is resuming.
```

Mechanizmy blokowania w Javie

- Java oferuje szereg mechanizmów zapobiegających brudnemu czytaniu i pisaniu.
- Największą kontrolę oferują programiście interfejsy `Lock` i `Condition`, które blokują wejście do danego fragmentu kodu, jeżeli jest on wykonywany przez inny wątek [Horstmann, 2016].
- Istnieją też struktury z wbudowanymi mechanizmami blokowania, które nie wymagają dodatkowego oprogramowania np. `Vector`.
- Rozwiązaniem pośrednim, jeżeli chodzi o konieczność dodatkowego oprogramowania jest mechanizm synchronizacji.

Synchronizacja

- Synchronizacja powoduje zapewnienie ekskluzywności wykonywania bloków kodu powiązanych z danym obiektem.
- Jeżeli synchronizowany kod jest wykonywany to inne wątki chcące go wykonać są blokowane.
- Blokowane wątki otrzymują status `BLOCKED`.
- Gdy wykonywanie kodu jest zakończone, blokowane wątki są zwalniane i jeden z nich może zacząć wykonywać kod blokując pozostałe.
- Synchronizacja jest powiązana z danym obiektem, wątki nie będą blokowane wykonując ten sam kod na innym obiekcie.
- Jeżeli mamy kilka synchronizowanych bloków powiązanych z danym obiektem, to mają one wspólną blokadę.
- Synchronizację oznacza się słowem kluczowym `synchronized`

Synchronizacja obiektu

- Synchronizując dostęp do obiektu `account` możemy zabezpieczyć wypłaty z konta.

Zabezpieczenie mechanizmu wypłaty

```
1   void doWithdraw(String authorizationCode, int value){
2     Account account = bank.getAccountFor(authorizationCode);
3     synchronized(account) {
4     if(account.getBalance()>value)
5       account.setDebit(value);
6     }
7   }
```

Synchronizacja metod

- Słowo kluczowe `synchronized` może być też stosowane do metod.
- W takim wypadku synchronizacja obejmuje kod metody i obiekt z którego jest ona wywoływana.

Synchronizacja bloku

```
1   void doWithdraw(int value){
2     synchronized (this) {
3       if (getBalance() > value)
4         setDebit(value);
5     }
6   }
```

Synchronizacja metody

```
1   synchronized void doWithdraw(int
        value){
2     if(getBalance()>value)
3       setDebit(value);
4   }
```

- Oba zapisy są równoważne

Ograniczenia synchronizacji

- Metoda synchronizacji ma pewne ograniczenia.
    - Nie można przerwać wątku, który czeka na blokadzie.
    - Nie można określić maksymalnego czasu oczekiwania na odblokowanie dostępu.
    - Warunek odblokowania dostępu jest powiązany tylko z jednym obiektem, może się zdarzyć, że jest to niewystarczające.

Synchronizacja zmiennych

- Zamiast tworzyć synchronizowane bloki możemy określić, że dana zmienna jest współużytkowana przez kilka wątków deklarując ją jako ulotną `volatile`.
- Bezpieczne jest także współużytkowanie zmiennych finalnych `final`, które nie mogą być modyfikowane po inicjacji.
- W Javie istnieje też mechanizm zmiennych atomowych, `java.util.concurrent.atomic` który wprowadza operacje niepodzielne, nie mogące prowadzić do brudnego pisania i czytania [Horstmann, 2016].

Komunikacja między synchronizowanymi wątkami

- Wewnątrz synchronizowanego kodu możemy sterować dostępem do blokowanego obiektu.
  - **wait** przedstawia wątek w stan oczekiwania, aż nadejdzie powiadomienie czy można kontynuować pracę.
  - **notify** odblokowuje jeden z oczekujących wątków,
  - **notifyAll** odblokowuje wszystkie czekające wątki
- Odblokowanie wątku odbywa się poprzez wygenerowanie wyjątku `IllegalMonitorStateException`.

Równoległy odczyt i zapis

- Przygotujmy wątki równolegle zapisujące i odczytujące dane.

Odczyt

```
1  public class MyReader implements
       Runnable{
2    char buffer[];
3
4    public MyReader(char[] buffer){
5      this.buffer = buffer;
6    }
7
8    @Override
9    public void run() {
10     while(true){
11       System.out.println();
12       for(char c: buffer){
13         System.out.print(c);
14       }
15     }
16   }
17 }
```

Zapis

```
1  public class MyWriter implements
       Runnable{
2
3    char buffer[];
4    char x;
5    public MyWriter(char[] buffer,
         char x) {
6      this.buffer = buffer;
7      this.x = x;
8    }
9    @Override
10   public void run() {
11     while(true) {
12       for (int i = 0; i
             <buffer.length; i++){
13         buffer[i] = x;
14         try {
15           Thread.sleep(10);
16         } catch
               (InterruptedException
               e) {
17       e.printStackTrace();
18       }
19     }
20   }
21 }
```

Równoległy odczyt i zapis - testowanie

- Przetestujmy stworzone metody zapisując trzy rodzaje znaków.

Testowanie

```
1  char buffer[] = new char[10];
2  Thread reader = new Thread(new MyReader(buffer));
3  Thread writerA = new Thread(new MyWriter(buffer,'A'));
4  Thread writerB = new Thread(new MyWriter(buffer,'B'));
5  Thread writerC = new Thread(new MyWriter(buffer,'C'));
6
7  reader.start();
8  writerA.start();
9  writerB.start();
10 writerC.start();
```

Wynik

```
BBCABBBBBB
BBCABBBBBB
BBBCBBBBBB
BBBCBBBBBB
BBBCBBBBBB
BBBCABBBBB
BBBBCBBBBB
BBBBCBBBBB
BBBBCBBBBB
BBBBCBBBBB
BBBBCBBBBB
BBBBCBBBBB
```

- Mamy do czynienia z brudnym czytaniem i pisaniem.
- Pozbędziemy się jego korzystając z synchronizacji.

Synchronizowany odczyt

- Przygotujmy wątki równolegle zapisujące i odczytujące dane.

Synchronizowany odczyt

```
1   public class MySynchronizedReader implements Runnable{
2     char buffer[];
3
4     public MySynchronizedReader(char[] buffer) {
5       this.buffer = buffer;
6     }
7       @Override
8     public void run() {
9       synchronized(buffer) {
10        while (true) {
11          System.out.println();
12          for (char c : buffer) {
13            System.out.print(c);
14          }
15          buffer.notifyAll();
16          try {
17            buffer.wait();
18          } catch (InterruptedException e) {
19            e.printStackTrace();
20          }
21        }
22      }
23    }
24  }
```

Synchronizowany zapis

Synchronizowany zapis

```
1   public class MySynchronizedWriter implements Runnable{
2       char buffer[];
3       char x;
4       public MySynchronizedWriter(char[] buffer, char x) {
5           this.buffer = buffer;
6           this.x = x;
7       }
8       @Override
9       public void run() {
10          synchronized (buffer) {
11              while (true) {
12                  for (int i = 0; i < buffer.length; i++) {
13                      buffer[i] = x;
14                     try {
15                          Thread.sleep(10);
16                     } catch (InterruptedException e) {
17                          e.printStackTrace();
18                     }
19                  }
20                  buffer.notifyAll();
21                  try {
22                      buffer.wait();
23                  } catch (InterruptedException e) {
24                      e.printStackTrace();
25                  }
26              }
27          }
28      }
29  }
```

Wyniki testu i komentarze

Wynik

```
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
```

- Wprowadzenie synchronizacji wyeliminowało brudne pisanie i czytanie.
- Jednakże zastosowana metoda rodzi pewne problemy.
  - Możliwe jest głodzenie procesów i niektóre procesy pisania nie będą realizowane.
  - Możliwy jest brak odczytu informacji zapisanych do bufora, jeżeli zostaną od razu nadpisane przez następnego pisarza.
- Możemy skorzystać z istniejących mechanizmów Javy, aby rozwiązać te problemy.

Kolejka blokująca

- Kolejka blokująca jest kolejką o określonej długości, która przechowuje obiekty.
- Jeżeli kolejka jest pełna to wątek chcący dodać element do kolejki jest blokowany.
- Jeżeli kolejka jest pusta to wątki chcące pobrać element z kolejki są blokowane.
- Kolejka jest zdefiniowana przez interfejs BlockingQueue, który jest implementowany przez klasę ArrayBlockingQueue.

Czytanie i pisanie przy użyciu kolejki

- Przygotujmy wątki zapisujące i odczytujące z kolejki.

Odczyt

```
1   public class MyQueuedReader
        implements Runnable {
2     private BlockingQueue queue;
3
4     public
          MyQueuedReader(BlockingQueue
          queue) {
5       this.queue = queue;
6     }
7
8     public void run() {
9       try {
10        while (true) { System.out
11        .println(queue.take()); }
12      } catch (InterruptedException
            e) {
13        e.printStackTrace();
14      }
15    }
16  }
```

Zapis

```
1   public class MyQueuedWriter
        implements Runnable {
2     BlockingQueue queue;
3     String x;
4
5     public
          MyQueuedWriter(BlockingQueue
          queue, String x) {
6       this.queue = queue;
7       this.x = x;
8     }
9       @Override
10    public void run() {
11      while (true) {
12        try {
13          queue.put(x);
14        } catch
              (InterruptedException
              e) {
15          e.printStackTrace();
16        }
17      }
18    }
19  }
```

Test działania kolejki

Synchronizowany zapis

```
 1  BlockingQueue<String> queue = new
        ArrayBlockingQueue<>(10);
 2  Thread reader = new Thread(new MyQueuedReader(queue));
 3  Thread writerA = new Thread(new
        MyQueuedWriter(queue,"AAAAAAAAAA"));
 4  Thread writerB = new Thread(new
        MyQueuedWriter(queue,"BBBBBBBBBB"));
 5  Thread writerC = new Thread(new
        MyQueuedWriter(queue,"CCCCCCCCCC"));
 6
 7  reader.start();
 8  writerA.start();
 9  writerB.start();
10  writerC.start();
```

Wynik

```
BBBBBBBBBB
AAAAAAAAAA
CCCCCCCCCC
BBBBBBBBBB
AAAAAAAAAA
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
CCCCCCCCCC
BBBBBBBBBB
BBBBBBBBBB
BBBBBBBBBB
```

Podsumowanie

- Wątki są niezbędne w prawidłowo działającym programie.
  - Poprawnie działający interfejs graficzny.
  - Kosztowne prace realizowane w tle.
- Jednakże korzystanie z wątków wprowadza szereg zagadnień komplikujących pracę programisty.
- Współcześnie Java oferuje narzędzia pozwalające zminimalizować koszt oprogramowania wątków i niebezpieczeństwa związane z ich używaniem.

Bibliografia

[Horstmann, 2016] Horstmann, C. S. (2016).
Java. Podstawy.
Helion.