Programowanie współbieżne
Laboratorium nr 15

1 Anulowanie zadań

"Nie istnieje bezpieczny sposób zatrzymania wątku Javy z wykorzystaniem wywłaszczenia [1], więc nie istnieje również bezpieczny sposób zatrzymania w ten sposób zadania. Pozostaje użycie mechanizmów współpracy, w których to zadanie i kod żądający anulowania współpracują ze sobą, używając uzgodnionego protokołu."

Metoda stop [2] "Zamyka ona wszystkie oczekujące metody, włącznie z metodą run. Jeśli zostanie zastosowana na rzecz wątku, natychmiast zdejmuje on wszystkie blokady, które założył. To może prowadzić do uszkodzenia obiektów."

1.1 Anulowanie zadania przy pomocy atrybutu

```
import java.util.concurrent.TimeUnit;

class CancellableTask extends Thread {
    private volatile boolean canceled;

    public void cancel() {
        System.out.println("CANCEL " + System.nanoTime());
        canceled = true;
    }

    @Override
    public void run() {
        System.out.println("BEGIN " + System.nanoTime());
        try {
            while (!canceled) {
                TimeUnit.SECONDS.sleep(1);
                System.out.println("BETWEEN " + System.nanoTime());
                TimeUnit.SECONDS.sleep(1);
            }
        } catch (InterruptedException e) {
            System.out.println("INTERRUPT " + System.nanoTime() + " " + isInterrupted());
        }
        System.out.println("END " + System.nanoTime());
    }
}

public class TestCancellableTask {
    public static void main(String[] args) throws InterruptedException {
        CancellableTask thread = new CancellableTask();
        thread.start();
        TimeUnit.MILLISECONDS.sleep(500);
        thread.cancel();
    }
}
```

Przykład 1: Przykład anulowania zadania przy pomocy atrybutu {src/TestCancellableTask.java}

1.2 Anulowanie zadania przy pomocy przerwania

"Każdy wątek zawiera w sobie status przerwania [1]; wywołanie metody przerwania dla wątku ustawia zmienną statusową na wartość true. Klasa Thread zwiera metody przerywania wątku i odpytywania o jego stan, (...). Metoda interrupt() przerywa docelowy wątek natomiast metoda isInterrupted() zwraca status

aktualnego wątku. Wyjątkowo niefortunnie nazwa metoda statyczna interrupted() czyści status przerwania i zwraca jego wcześniejszy stan. To jedyny sposób wyczyszczenia stanu wątku."

```
import java.util.concurrent.TimeUnit;

class InterruptableTask extends Thread {
    public void cancel() {
        System.out.println("CANCEL " + System.nanoTime());
        interrupt();
    }

    @Override
    public void run() {
        System.out.println("BEGIN " + System.nanoTime());
        try {
            while (!isInterrupted()) {
                TimeUnit.SECONDS.sleep(1);
                System.out.println("BETWEEN " + System.nanoTime());
                TimeUnit.SECONDS.sleep(1);
            }
        } catch (InterruptedException e) {
            System.out.println("INTERRUPT " + System.nanoTime() + " " + isInterrupted());
        }
        System.out.println("END " + System.nanoTime());
    }
}

public class TestInterruptableTask {
    public static void main(String[] args) throws InterruptedException {
        InterruptableTask thread = new InterruptableTask();
        thread.start();
        TimeUnit.MILLISECONDS.sleep(500);
        thread.cancel();
    }
}
```

Przykład 2: Przykład anulowania zadania przy pomocy przerwania {src/TestInterruptableTask.java}

1.3 Anulowanie zadania przy pomocy Future

`Interface java.util.concurrent.Future<V>:`

- `cancel(boolean mayInterruptIfRunning)`

  Jeżeli zadanie nie zaczęło się wykonywać nigdy nie zostanie uruchomione, jeżeli natomiast wykonanie zadania już się rozpoczęło, to tylko w przypadku gdy jako wartość parametru przekazano true, wątek wykonujący zadanie powinien być przerwany; metoda zwraca informację o powodzeniu anulowania.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

class FutureCancellableTask extends Thread {
    public void run() {
        System.out.println("BEGIN " + System.nanoTime());
        try {
            TimeUnit.SECONDS.sleep(1);
            System.out.println("BETWEEN " + System.nanoTime());
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
```

```
            System.out.println("INTERRUPT " + System.nanoTime());
        }
        System.out.println("END " + System.nanoTime());
    }
}

public class TestFuture {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<?> future = executor.submit(new FutureCancellableTask());
        TimeUnit.MILLISECONDS.sleep(500);
        future.cancel(true);
        executor.shutdown();
    }
}
```

Przykład 3: Przykład anulowania zadania przy pomocy Future {src/TestFuture.java}

1.4 Anulowanie zadania przy pomocy tzw. 'pigułki z trucizną'

"Kolejnym sposobem przekonania usługi producent - konsument [1] do zatrzymania się jest pigułka z trucizną - specjalny obiekt umieszczony w kolejce, którego obecność oznacza: 'gdy tu dotrzesz, zatrzymaj się'".

2 Radzenie sobie z blokowaniem niedającym szans na przerwanie

2.1 Synchroniczna operacja we-wy dla gniazda z java.io.

"Synchroniczna operacja we-wy dla gniazda z java.io[1]. Częstą postacią blokującego we-wy w aplikacjach serwerowych jest odczyt lub zapis danych z gniazda. Niestety metody read() i write() z InputStream i OutputStream nie reagują na przerwanie, ale zamknięcie gniazda spowoduje zgłoszenie przez te blokujące metody wyjątku SocketException."

```
import java.io.IOException;
import java.net.ServerSocket;
import java.util.concurrent.TimeUnit;

public class TestSocket {
    public static void main(String[] args) throws Exception {
        final ServerSocket socket = new ServerSocket(12345);
        Thread socketThread = new Thread() {
            @Override
            public void run() {
                try {
                    System.err.println("BEFORE " + System.nanoTime());
                    socket.accept();
                    System.err.println("AFTER " + System.nanoTime());
                    // ...
                } catch (IOException e) {
                    System.err.println("EXCEPTION " + System.nanoTime());
                    e.printStackTrace();
                }
            }
        };
        socketThread.start();
        TimeUnit.SECONDS.sleep(5);
        System.err.println("INTERRUPT " + System.nanoTime());
        socketThread.interrupt();
        TimeUnit.SECONDS.sleep(5);
        System.err.println("CLOSE " + System.nanoTime());
        socket.close();
```

```
    }
}
```

Przykład 4: Radzenie sobie z blokowaniem niedającym szans na przerwanie. Synchroniczna operacja we-wy dla gniazda z java.io. {src/TestSocket.java}

2.2 Synchroniczna operacja we-wy dla gniazda z java.nio.

"Synchroniczna operacja we-wy dla gniazda z java.nio[1]. Przerwanie wątku oczekującego na InterruptibleChannel powoduje zgłoszenie wyjątku ClosedByInterruptException i zamknięcie kanału (co w konsekwencji oznacza zgłoszenie wyjątku ClosedByInterruptException przez pozostałe wątki zablokowane na kanale). Zamknięcie kanału powoduje zgłoszenie przez zablokowane na nim wątki wyjątku AsynchronousCloseException. Większość standardowych kanałów implementuje wersję InterruptableChannel."

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.ServerSocketChannel;
import java.util.concurrent.TimeUnit;

public class TestNioSynchronous {
    public static void main(String[] args) throws Exception {
        final boolean interrupt = true;
        final ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.socket().bind(new InetSocketAddress(54321));
        Thread socketThread = new Thread() {
            @Override
            public void run() {
                try {
                    System.err.println("BEFORE " + System.nanoTime());
                    serverChannel.accept();
                    System.err.println("AFTER " + System.nanoTime());
                    // ...
                } catch (IOException e) {
                    System.err.println("EXCEPTION " + System.nanoTime());
                    e.printStackTrace();
                }
            }
        };
        socketThread.start();
        if (interrupt) {
            TimeUnit.SECONDS.sleep(5);
            System.err.println("INTERRUPT " + System.nanoTime());
            socketThread.interrupt();
        } else {
            TimeUnit.SECONDS.sleep(5);
            System.err.println("CLOSE " + System.nanoTime());
            serverChannel.close();
        }
    }
}
```

Przykład 5: Radzenie sobie z blokowaniem niedającym szans na przerwanie. Synchroniczna operacja we-wy dla gniazda z java.nio. {src/TestNioSynchronous.java}

2.3 Asynchroniczne we-wy z użyciem Selector.

"Asynchroniczne we-wy z użyciem Selector[1]. Jeżeli wątek jest zablokowany na operacji Selector.select() (z pakietu java.nio.channels), metoda close() powoduje jego natychmiastowy powrót przez zgłoszenie wyjątku ClosedSelectorException."

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.util.concurrent.TimeUnit;

public class TestNioAsynchronous {
    public static void main(String[] args) throws Exception {
        final boolean interrupt = true;
        final Selector socketSelector = Selector.open();
        final ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.socket().bind(new InetSocketAddress(54321));
        serverChannel.configureBlocking(false);
        serverChannel.register(socketSelector, SelectionKey.OP_ACCEPT);
        final ServerSocketChannel serverChannel2 = ServerSocketChannel.open();
        serverChannel2.socket().bind(new InetSocketAddress(54322));
        serverChannel2.configureBlocking(false);
        serverChannel2.register(socketSelector, SelectionKey.OP_ACCEPT);

        Thread socketThread = new Thread() {
            @Override
            public void run() {
                try {
                    System.err.println("BEFORE " + System.nanoTime());
                    int s = -1;
                    if ((s = socketSelector.select()) > 0) {
                        System.err.println("AFTER " + System.nanoTime());
                        // ...
                    }
                    System.err.println("END " + s + System.nanoTime());
                } catch (Exception e) {
                    System.err.println("EXCEPTION " + System.nanoTime());
                    e.printStackTrace();
                }
            }
        };
        socketThread.start();
        if (interrupt) {
            TimeUnit.SECONDS.sleep(5);
            System.err.println("INTERRUPT " + System.nanoTime());
            socketThread.interrupt();
        } else {
            TimeUnit.SECONDS.sleep(5);
            System.err.println("CLOSE " + System.nanoTime());
            socketSelector.close();
        }
    }
}
```

Przykład 6: Radzenie sobie z blokowaniem niedającym szans na przerwanie. Asynchroniczne we-wy z użyciem Selector. {src/TestNioAsynchronous.java}

2.4 Przejęcie blokady.

"Przejęcie blokady[1]. Jeżeli wątek został zablokowany w oczekiwaniu na blokadę wewnętrzną, nie można nic zrobić, by odblokować w prosty sposób. Pozostaje się postarać, by szybko uzyskał blokadę i wykonał wystarczająco dużo poleceń, by zauważył przerwanie. Klasy jawnych blokad (Lock) oferują metodę lockInterruptibly(), która dopuszcza oczekiwanie na blokadę i przyjmowanie przerwań.(...)"

3 Klasa Lock

"Przed Javą 5.0 [1] jedynym mechanizmem koordynującym dostęp do współdzielonych danych były słowa kluczowe synchronized i volatile. W Javie 5.0 znalazła się dodatkowa możliwość - klasa ReentrantLock. W przeciwieństwie do sugestii, które pojawiły się w innych publikacja, klasa ta nie ma na celu zastąpienie blokowania wewnętrznego ale raczej ma stanowić jego uzupełnienie w sytuacjach, gdy blokady wewnętrzne okazują się mało elastyczne."

Interfejs java.util.concurrent.locks.Lock:

- void lock() - próba założenie blokady, jeżeli blokada nie jest aktualnie dostępna wątek zostaje zablokowany,
- void lockInterruptibly() - próba założenia blokady, jeżeli blokada nie jest aktualnie dostępna wątek zostaje zablokowany, ale z możliwością otrzymywania przerwań,
- boolean tryLock() - próba założenia blokady, pod warunkiem, że blokada jest aktualnie dostępna, istnieje także przeciążona wersja z ograniczeniem czasowym,
- boolean void unlock() - zwalnia blokadę,
- Condition newCondition() - zwraca obiekt warunku (klasa Condition), powiązany z tym obiektem klasy Lock.

"Czasowe i odpytywane tryby [1] zajmowania blokady uzyskiwane dzięki tryLock() dopuszczają stosowanie bardziej wyrafinowanych sposobów ratowania się w sytuacji blokady niż akwizycja bezwarunkowa."

"Uzyskiwane blokad za pomocą tryLock() [1] umożliwia odzyskanie sterowania, jeśli nie uda się zająć wszystkich wymaganych blokad. W tym momencie zwalniane są już zajęte blokady i następuje ponowna próba blokowania (lub przynajmniej poinformowanie o nieudanym zajęciu blokady)"

Ochrona obiektu za pomocą Lock [8]:

```
Lock l = ...;
l.lock();
try {
    ...
} finally {
    l.unlock();
}
```

Interfejs java.util.concurrent.locks.Condition:

- void await() - aktualny wątek oczekuje, aż zostanie wywołana metoda signal(), signalAll(), lub otrzyma przerwanie, istnieją także metody oczekujące z dodatkowym ograniczeniem czasowym (boolean await(long time, TimeUnit unit), long awaitNanos(long nanosTimeout), boolean awaitUntil(Date deadline)),
- void awaitUninterruptibly() - aktualny wątek oczekuje, aż zostanie wywołana metoda signal(), signalAll(),
- void signal() - wybudzenie jednego wątku,
- void signalAll() - wybudzenie wszystkich wątków.

Użycie obiektu warunku Condition [8]:

```
final Lock lock = new ReentrantLock();
final Condition writeCondition  = lock.newCondition();
final Condition readCondition  = lock.newCondition();

void write() throws InterruptedException{
     lock.lock();
     try {
          while (...)
               writeCondition.await();
```

```
            -- write
            readCondition.signal();
        } finally {
            lock.unlock();
        }
}
void read() throws InterruptedException{
        lock.lock();
        try {
            while (...)
                readCondition.await();
            --read
            writeCondition.signal();
        } finally {
            lock.unlock();
        }
}
```

Interfejs java.util.concurrent.locks.ReadWriteLock:

- Lock readLock() - zwraca obiekt Lock używany podczas odczytu,
- Lock writeLock() - zwraca obiekt Lock używany podczas zapisu.

"Blokada ReadWriteLock [4] optymalizuje sytuacje, w których struktura danych jest zapisywana stosunkowo rzadko, za to jest często odczytywana. Klasa ReadWriteLock pozwala na równoczesne zakładanie blokady odczytu przez wiele zadań odczytujących dopóty, dopóki nie pojawi się zadanie zapisujące. Po założeniu blokady zapisu żadne z zadań odczytujących nie uzyska dostępu aż do zwolnienia blokady zapisu."

Klasy, które implementują Lock:

- ReentrantLock,
- ReentrantReadWriteLock.ReadLock,
- ReentrantReadWriteLock.WriteLock.

Klasa, która implementuje ReadWriteLock:

- ReentrantReadWriteLock.

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class Buffer {
    private int i = 0;
    final private ReentrantReadWriteLock.ReadLock readLock;
    final private ReentrantReadWriteLock.WriteLock writeLock;

    Buffer() {
        ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
        readLock = lock.readLock();
        writeLock = lock.writeLock();
    }

    int get(long tid) throws InterruptedException {
        System.out.println("BEFORE GET " + tid + " " + System.nanoTime());
        readLock.lock();
        System.out.println("AFTER GET " + tid + " " + System.nanoTime());

        try {
            TimeUnit.SECONDS.sleep(1);
            return i;
```

```
        } finally {
            readLock.unlock();
            System.out.println("END GET " + tid + " " + System.nanoTime());
        }
    }

    void set(long tid, int i) throws InterruptedException {
        System.out.println("BEFORE SET " + tid + " " + System.nanoTime());
        writeLock.lock();
        System.out.println("AFTER SET " + tid + " " + System.nanoTime());
        try {
            TimeUnit.SECONDS.sleep(5);
            this.i = i;
        } finally {
            writeLock.unlock();
            System.out.println("END SET " + tid + " " + System.nanoTime());
        }
    }
}

class Reader extends Thread {
    private Buffer b;

    Reader(Buffer b) {
        this.b = b;
    }

    public void run() {
        int i = 0;
        long tid = getId();
        try {
            while (i < 10)
                b.get(tid);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Writer extends Thread {
    private Buffer b;

    Writer(Buffer b) {
        this.b = b;
    }

    public void run() {
        int i = 0;
        long tid = getId();
        try {
            while (i < 5)
                b.set(tid, i);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class TestLock {
    public static void main(String[] args) throws InterruptedException {
        final Buffer b = new Buffer();
        for (int i = 0; i < 2; i++) {
```

```
            TimeUnit.MILLISECONDS.sleep(500);
            new Reader(b).start();
        }
        TimeUnit.MILLISECONDS.sleep(500);
        new Writer(b).start();
    }
}
```

Przykład 7: Przykład użycia klasy Lock. {src/TestLock.java}

4 Zmienne niepodzielne

"Klasy zmiennych niepodzielnych [1] są niejako uogólnieniem zmiennych ulotnych o obsługę niepodzielnej operacji warunkowej odczyt, modyfikacja, zapis."

Pakiet java.util.concurrent.atomic: AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicIntegerFieldUpdater, AtomicLong, AtomicLongArray, AtomicLongFieldUpdater, AtomicMarkableReference, AtomicReference, AtomicReferenceArray, AtomicReferenceFieldUpdater, AtomicStampedReference.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class TestAtomic {
    final static int SIZE = 2;
    final static int LOOPS = 1000;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService e = Executors.newCachedThreadPool();
        for (int i = 0; i < SIZE; i++)
            e.execute(new Counter());
        e.shutdown();
        e.awaitTermination(1, TimeUnit.DAYS);
        System.out.println("Counter is : " + Counter.counter);
        System.out.println("Difference : " + (double) (SIZE * LOOPS - Counter.counter.get()) / (↩
            SIZE * LOOPS));
    }
}

class Counter implements Runnable {
    static AtomicInteger counter = new AtomicInteger(0);

    @Override
    public void run() {
        int i = 0;
        while (i < TestAtomic.LOOPS) {
            ++i;
            counter.incrementAndGet();
        }
        System.out.println(Thread.currentThread() + " : " + i);
    }
}

/* lub */
/*
class Counter implements Runnable {
    volatile static int counter;

    @Override
    public void run() {
```

```
        int i = 0;
        while (i < TestAtomic.LOOPS) {
            ++i;
            ++counter;
        }
        System.out.println(Thread.currentThread() + " : " + i);
    }
}
*/
```

Przykład 8: Przykład użycia zmiennych niepodzielnych. {src/TestAtomic.java}

5 Współbieżność i Swing

Zasady, których należy przestrzegać łącząc wątki oraz Swing [2]: "Jeżeli jakieś zadanie jest czasochłonne, należy je wykonać w osobnym wątku roboczym nigdy w wątku dystrybucji zdarzeń."

Zasada jednego wątku: "Nie należy operować na komponentach Swing w żadnym innym wątku niż wątek dystrybucji zdarzeń."

"Od zasady jednego wątku jest kilka wyjątków:

- Słuchaczy zdarzeń można bezpiecznie dodawać i usuwać w każdym wątku. Oczywiście metody słuchaczy są wywoływane w wątku dystrybucji zdarzeń.
- Niektóre metody Swing są bezpieczne wątkowo. W dokumentacji API są one oznaczone specjalnym hasłem 'This method is thread safe, although most Swing methods are not.' (Metoda ta jest bezpieczna wątkowo, mimo iż większość metod biblioteki Swing nie jest). (...)"

Metody dodawania zadań do wątku dystrybucji zdarzeń:

- SwingUtilities.invokeLater(Runnable runnable);
- EventQueue.invokeLater(Runnable runnable);
- SwingUtilities.invokeAndWait(Runnable runnable);
- EventQueue.invokeAndWait(Runnable runnable);

6 Przykładowa treść laboratorium

1. Stworzyć aplikację demonstrującą anulowanie zadania wykonywanego przez wątek przy pomocy:
   - współdzielonej zmiennej,
   - przerwania,
   - interfejsu Future,
   - tzw. 'pigułki z trucizną'.

   2. Stworzyć aplikację składającą się z co najmniej dwóch wątków czytających oraz jednego modyfikującego wspólne dane np. wybrany atrybut w klasie. Współbieżny dostęp do danych powinien zostać zapewniony przy pomocy:
      - zmiennej niepodzielnej,
      - obiektu klasy Lock,
      - obiektu klasy ReadWriteLock.

Literatura

[1] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D., Java Współbieżność dla praktyków, Helion 2007

[2] Horstmann C.S., Cornell G., Java Podstawy, Helion, Wyd. VIII, 2009

[3] Horstmann C.S., Cornell G., Java Techniki zaawansowane, Helion, Wyd. VIII, 2009

[4] Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006.

[5] Bloch J.: Java Efektywne programowanie, Wyd. II, Helion, 2009.

[6] Brackeen D., B. Barker, L. Vanhelsuwe: Java Tworzenie gier, Helion, 2004.

[7] Silberschatz A., Galvin P. B., Gagne G.: Podstawy systemów operacyjnych, WNT, 2005

[8] Dokumentacja JavaDoc 1.6 htp://java.oracle.com

[9] Dokumentacja JavaDoc 1.7 htp://java.oracle.com

[10] The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf