

Programowanie współbieżne

Laboratorium nr 11

„Wykonywane w danym momencie programy nazwano procesami [1] - izolowanymi, niezależnie wykonywanymi programami, dla których system operacyjny przydzielał pamięć, uchwytty plików i sprawdzał reguły bezpieczeństwa. ”

„Wątki [1] wprowadzają wiele strumieni wykonywania programu istniejących w tym samym czasie w jednym procesie. Współdziela zasoby procesu takie jak pamięć i uchwytty plików, ale każdy wątek ma własny wskaźnik instrukcji, stos i zmienne lokalne. ”

Klasa `public abstract class Process:`

- `public abstract InputStream getInputStream()` - uzyskanie dostępu do strumienia standardowego wejścia procesu,
- `public abstract OutputStream getOutputStream()` - uzyskanie dostępu do strumienia standardowego wyjścia procesu,
- `public abstract InputStream getErrorStream()` - uzyskanie dostępu do strumienia standardowego wyjścia błędów procesu,
- `public abstract int waitFor()` - oczekiwanie na zakończenie procesu,
- `public abstract int exitValue()` - sprawdzenie stanu końcowego procesu,
- `public abstract void destroy()` - zniszczenie procesu.

```
public static void testRuntime() throws IOException, InterruptedException{
    Runtime runtime = Runtime.getRuntime();

    Process process = runtime.exec("ipconfig /all");
    BufferedReader input = new BufferedReader(
        new InputStreamReader(process.getInputStream()));
    String s = null;
    while ((s = input.readLine()) != null)
        if (s.length() > 0)
            System.out.println(s);
    input.close();
    process.waitFor();

    Process process2 = runtime.exec("notepad");
    process2.waitFor();
}
```

Od wersji Javy 1.5 metoda `ProcessBuilder.start()` jest zalecaną metodą tworzenia procesów.

Klasa `public final class ProcessBuilder:`

- `ProcessBuilder(List<String> command)` lub `ProcessBuilder(String... command)` - tworzy obiekt klasy z podanym programem systemu operacyjnego i argumentami,
- `public List<String> command()` - zwraca programem systemu operacyjnego oraz argumenty,
- `public ProcessBuilder command(List<String> command)` lub `public ProcessBuilder command(String... command)` - ustawi program systemu operacyjnego i argumenty,
- `public File directory()` - zwraca katalog roboczy,
- `public ProcessBuilder directory(File directory)` - ustawi katalog roboczy,
- `public Map<String, String> environment()` - zwraca środowisko,
- `public boolean redirectErrorStream()` - zwraca informację o tym, czy standardowe wyjście i standardowe wyjście błędów są połączone,

- `public ProcessBuilder redirectErrorStream(boolean redirectErrorStream)` - ustawia właściwość `redirectErrorStream`,
- `public Process start()` - uruchamia nowy proces.

```
public static void testProcessBuilder()
    throws IOException, InterruptedException {
    ProcessBuilder processBuilder =
        new ProcessBuilder("C:\\WINDOWS\\system32\\ipconfig", "/all");
    processBuilder.directory(new File("C:\\"));
    Process process = processBuilder.start();
    process.waitFor();
}
```

Klasa `public class Thread` (wybrane metody):

```
public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() - zwraca procedurę
obsługi wywoływanej gdy wątek zakończy się w wyniku wystąpienia niewyłapanego wyjątku (istnieje
także getDefaultUncaughtExceptionHandler()),
public long getId() - zwraca identyfikator wątku,
public final String getName() - zwraca nazwę wątku,
public Thread.State getState() - zwraca stan wątku (NEW, RUNNABLE, BLOCKED, WAITING,
TIMED_WAITING, TERMINATED),
public final boolean isDaemon() - sprawdza czy wątek jest tzw. wątkiem demonem, czyli wątkiem
działającym w tle,
public void join() - oczekuje na zakończenie się wątku, posiada także warianty przeciążone,
public void run() - metoda, która ma się wykonywać współbieżnie,
public final void setDaemon(boolean on) - służy do oznaczania wątku jako tzw. wątku demona lub
wątku użytkownika,
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh) - ustawia
procedurę obsługi wywoływanej gdy wątek zakończy się w wyniku wystąpienia niewyłapanego wyjątku
(istnieje także setDefaultUncaughtExceptionHandler()),
public final void setName(String name) - ustawia nazwę wątku,
public static void sleep(long millis) - usypia wątek na określony czas (chwilowo przerywa jego
wykonanie), posiada także wariant przeciążony,
public void start() - metoda służy do rozpoczęcia wykonywania się wątku,
public static void yield() - powoduje, że aktualnie wykonujący się wątek chwilowo jest
wstrzymywany i pozwala na wykonanie się innych wątków.
```

Interfejs `Runnable`:

```
public interface Runnable{
    void run();
}
```

Uruchomienie zadania z wykorzystaniem interfejsu `Runnable`:

```
class MyRunnable implements Runnable {
    public void run() {
        ...
    }
}
public class MyThreadTest {

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

Uruchomienie zadania z wykorzystaniem klasy Thread:

```
class MyThread extends Thread {
    @Override
    public void run() {
        ...
    }
}

public class MyThreadTest {

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

„Podstawową abstrakcją wykonania zadania [1] w bibliotekach Javy nie jest Thread, ale Executor (...).”
Interfejs Executor:

```
interface Executor {

    void execute(Runnable command);
}
```

„Obiekt klasy Executor [8] stanowi pośrednik pomiędzy klientem a wykonywanym zadaniem; klient zamiast uruchamiać zadanie wprost, zdaje się na obiekt wykonawcy.”

Bezpośrednie uruchomienie zadania [8]:

```
new Thread(new MojeZadanie()).start();
```

może zostać zastąpione poprzez [8]:

```
Executor executor = ...;
executor.execute(new MojeZadanie());
```

Przykład pochodzi z „Java™ Platform Standard Ed. 6”, java.util.concurrent.Executor

```
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    }
}
```

Przykład pochodzi z „Java™ Platform Standard Ed. 6”, java.util.concurrent.Executor

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Pule wątków [1]:

Executors.newFixedThreadPool() - „pula wątków o stałym rozmiarze tworzy wątki przy nadchodzeniu nowych zadań aż do osiągnięcia maksymalnego rozmiaru. Następnie stara się utrzymać stały rozmiar puli (dodaje nowy wątek, gdy poprzedni wyłączy się lub zgłosi wyjątek).”

Executors.newCachedThreadPool() - „(...) Dodaje nowe wątki, gdy wzrasta zapotrzebowanie na moc, ale nie ogranicza rozmiaru puli.”

`Executors.newSingleThreadExecutor()` - „jednowątkowy system wykonawczy tworzy jeden wątek zadaniowy przetwarzający zlecenia.”

`Executors.newScheduledThreadPool()` - „pula wątków o stałym rozmiarze obsługująca zadania opóźnione i okresowe.”

Interfejs `ExecutorService`:

```
public interface ExecutorService:
```

- `boolean awaitTermination(long timeout, TimeUnit unit)` - blokuje się dopóki wszystkie zadania zakończą się po żądaniu wyłączenia, lub nastąpi limit czasu lub bieżący wątek zostanie przerwany,

- przeciążone wersje metod `invokeAll`, `submit` oraz `invokeAny` służą do wykonywania zadań,

- `boolean isShutdown()` - zwraca prawdę jeżeli wykonawca został wyłączony,

- `boolean isTerminated()` - zwraca prawdę jeżeli wszystkie zadania zakończyły wyłączenie,

- `void shutdown()` - inicjuje zamknięcie, przyjęte wcześniej zadania są wykonywane, ale nowe zadania nie są przyjmowane,

- `List<Runnable> shutdownNow()` - próbuje zatrzymać wszystkie aktywnie wykonywane zadania; zatrzymuje przetwarzanie oczekujących zadań; zwraca listę zadań, które oczekiwały na wykonanie.

Przykład implementacji interfejsu `Executor`:

```
enum PriorityLevel {
    NO_PRIORITY(0), LOW_PRIORITY(1), MEDIUM_PRIORITY(2), HIGH_PRIORITY(3);

    private int value;

    PriorityLevel(int value) {
        this.value = value;
    }

    int getValue() {
        return value;
    }
}

class PriorityTask implements Runnable, Comparable<PriorityTask> {
    private PriorityLevel priorityLevel;
    private int taskNumber;
    private Runnable runnable;

    PriorityTask(int taskNumber, PriorityLevel priority, Runnable runnable) {
        this.taskNumber = taskNumber;
        this.priorityLevel = priority;
        this.runnable = runnable;
    }

    @Override
    public void run() {
        System.out.printf("%-30s %d\n", "START " + this,
            System.currentTimeMillis());
        runnable.run();
    }
}
```

```

@Override
public int compareTo(PriorityTask t) {
    return t.priorityLevel.getValue() - this.priorityLevel.getValue();
}

@Override
public String toString() {
    return taskNumber + ": " + priorityLevel;
}
}

class PriorityExecutor implements Executor {

    PriorityExecutor(final ExecutorService executor, final boolean sleep) {
        new Thread() {
            @Override
            public void run() {
                try {
                    if(sleep)
                        TimeUnit.SECONDS.sleep(1);
                    while (true)
                        executor.execute(tasks.take());
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                } finally {
                    executor.shutdown();
                }
            }
        }.start();

        final PriorityBlockingQueue<Runnable> tasks =
            new PriorityBlockingQueue<Runnable>();

        @Override
        public void execute(Runnable r) {
            tasks.add(r);
        }
    }

    public class TestPriorityExecutor {

        public static void main(String args[]) {
            Runnable r = new Runnable() {
                @Override
                public void run() {
                    try {
                        TimeUnit.SECONDS.sleep(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            };

            PriorityExecutor executor =
                new PriorityExecutor(Executors.newSingleThreadExecutor(), false);
            executor.execute(new PriorityTask(1, PriorityLevel.NO_PRIORITY, r));
            executor.execute(new PriorityTask(2, PriorityLevel.LOW_PRIORITY, r));
            executor.execute(new PriorityTask(3, PriorityLevel.MEDIUM_PRIORITY, r));
            executor.execute(new PriorityTask(4, PriorityLevel.HIGH_PRIORITY, r));
            executor.shutdown();
        }
    }
}

```

Interfejs Callable:

```
public interface Callable<V>{
    V call();
}
```

Uruchomienie zadania z wykorzystaniem interfejsu Callable:

```
private static <T extends Callable<U>, U> void test(List<T> tasks) {
    List<Future<U>> results = new ArrayList<Future<U>>();
    ExecutorService executor = Executors.newCachedThreadPool();
    for (T task : tasks)
        results.add(executor.submit(task));
    for (Future<U> result : results)
        try {
            System.out.println(result.get());
        } catch (InterruptedException e) {
            e.getCause().printStackTrace();
        } catch (ExecutionException e) {
            e.getCause().printStackTrace();
        }
    executor.shutdown();
}

class MyCallableTaskDouble implements Callable<Double> {
    private Double a, b;

    MyCallableTaskDouble(Double a, Double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public Double call() throws Exception {
        return a / b;
    }
}

private static void testDouble() {
    ArrayList<MyCallableTaskDouble> tasksDouble =
        new ArrayList<MyCallableTaskDouble>();
    tasksDouble.add(new MyCallableTaskDouble(1d, 0d));
    tasksDouble.add(new MyCallableTaskDouble(-1d, 0d));
    tasksDouble.add(new MyCallableTaskDouble(0d, 0d));
    test(tasksDouble);
}
```

Przykładowa treść laboratorium:

1. Przy pomocy ProcessBuilder lub Runtime wyświetlić listę uruchomionych usług (net start) lub informacje o sterownikach (driverquery).

2. Uruchomić podane zadania wykorzystując:

- Runnable,
- Thread,
- Callable,
- Executors.newFixedThreadPool,
- Executors.newCachedThreadPool(),
- Executors.newSingleThreadExecutor().

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 <http://java.sun.com>
- [9] Dokumentacja JavaDoc 1.7 <http://java.sun.com>
- [10] The Java Language Specification, Third Edition,
<http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>