Programowanie współbieżne
Laboratorium nr 12

Uwaga !!!
Wyniki uruchomionych programów mogą zależeć od sprzętu (ilość procesorów, rdzeni itp.), systemu operacyjnego, obciążenia systemu operacyjnego, ilości wątków w programie (zmienna SIZE), wersji maszyny wirtualnej Javy, itp.

Atomowość
„(...) operacja atomowa (inaczej: ”niepodzielna”) [4] to czynność, której mechanizm zarządzania wątkami nie jest w stanie przerwać - jeśli operacja się rozpocznie, będzie wykonywana aż do zakończenia, a dopiero potem pojawi się możliwość zmiany kontekstu.”

Widoczność
„W systemach wieloprocesorowych [4] (...) do aspektów atomowości dochodzi kwestia widoczności, znacznie bardziej tu eksponowana niż w systemach jednoprocesorowych. Otóż zmiany wprowadzane przez jedno zadanie, nawet jeśli są atomowe w sensie niepodzielności, mogą być niewidoczne dla innych zadań (zmiany mogą się na przykład odbywać w lokalnej pamięci podręcznej procesora), przez co różne zadania będą różnie postrzegać stan aplikacji.”

Synchronizacja
„(...) mechanizm synchronizacji wymusza [4], aby zmiany wprowadzone przez zadanie w systemie wieloprocesorowy były widoczne dla wszystkich współbieżnych zadań aplikacji. Bez synchronizacji nie sposób określić momentu uwidocznienia zmiany.”

Pola volatile
„Pole z modyfikatorem volatile [1] wskazuje kompilatorowi i systemowi wykonawczemu, że zmienna jest współdzielona i wykonywanych na niej operacji w pamięci nie należy układać w innej kolejności niż wskazana w kodzie źródłowym. Zmienne ulotne nie są przechowywane ani w buforach, gdy są ukryte przed innymi procesorami.”

Czy usunięcie modyfikatorów volatile z definicji zmiennych i oraz j zmienia coś w działaniu programu?

```
public class TestVolatile {
	public static void main(String[] args) {
		new Thread() {
			public void run() {
				while (true)
					Test.one();
			}
		}.start();
		new Thread() {
			public void run() {
				while (true)
					Test.two();
			}
		}.start();
	}

	private static class Test {
		private volatile static int i = 0;
		private volatile static int j = 0;

		private static void one() {
			i++;
			j++;
		}
```

```
		private static void two() {
			int tmpI = i;
			int tmpJ = j;
			System.out.println("" + tmpI + ";" + tmpJ);
			if (tmpI > tmpJ + 1)
				System.exit(1);
		}
	}
}
```

„Gdy wątek odczytuje zmienną bez synchronizacji [1], może widzieć nieświeżą wartość, ale przynajmniej jest to wartość umieszczona tam przez inny watek, a nie jakaś losowa wartość. Mówi się w takiej sytuacji o bezpieczeństwie poprawności.
To bezpieczeństwo dotyczy wszystkich zmiennych poza jednym wyjątkiem - 64-bitowych zmiennych liczbowych (double i long) niezadeklarowanych jako volatile (...). Model pamięci Javy wymaga, by operacje pobierania i zapamiętywania były niepodzielne, ale dla nieulotnych zmiennych long i double maszyna wirtualna może potraktować odczyt i zapis 64-bitowy jako dwie operacje 32-bitowe. Jeżeli zapis i odczyt takiej nieulotnej zmiennej odbywa się w dwóch różnych wątkach, wątek odczytujący może przeczytać niższe 32 bity nowej wartości i wyższe 32 bity starej.”

Czy usunięcie modyfikatorów volatile z definicji zmiennej i zmienia coś w działaniu programu?

```
public class TestLongVolatile {
	public static void main(String[] args) {
		new Thread() {
			public void run() {
				while (true)
					TestLong.write();
			}
		}.start();
		new Thread() {
			public void run() {
				while (true)
					TestLong.read();
			}
		}.start();
	}

	private static class TestLong {

		static volatile long i = 0;

		public static void write() {
			long tmp = i & 0xFFFFFFFFL;
			tmp++;
			i = tmp | (tmp << 32);
		}

		public static void read() {
			long tmp = i;
			long l = tmp & 0xFFFFFFFFL;
			long h = tmp >>> 32;
			if (l != h) {
				System.err.println("!" + Long.toHexString(tmp) + " : " + h
						+ " != " + l);
				System.exit(0);
			}
		}
	}
}
```

Synchronizacja

„Java [1] ma wbudowany mechanizm blokad zapewniający niepodzielność: blok synchronized (...). Blok synchronized składa się z dwóch części: referencji do obiektu, który służy jako blokada, oraz blok kodu chroniony przez blokadę. Metoda synchronizowana to skrótowa wersja bloku synchronized obejmująca całe ciało metody. Właściwą blokadą jest wtedy obiekt, na rzecz, którego została wywołana metoda. Statyczne metody sychronizowane używają obiektu Class do zapewnienia blokady."

W których miejscach należy usunąć komentarz (lub komentarze), żeby program działał poprawnie?

```
public class Test extends Thread {
	final private static int SIZE = 4;

	static MyObject value = new MyObject();

	public void run() {
		int i;
		while (true) {
			i = value.method();
			switch (i) {
			case 0:
			case 1:
				break;
			default:
				System.err.println("ERROR " + i);
				System.exit(1);
			}
		}
	}

	public static void main(String[] args) {

		Test[] t = new Test[SIZE];
		for (int i = 0; i < SIZE; i++) {
			t[i] = new Test();
			t[i].start();
		}
	}
}

class MyObject {

	private int i = 0;

	/* synchronized */public int method() {
		/* synchronized (this) */{
			if (i == 0)
			/* synchronized (this) */{
				return ++i;
			}
		}
		/* synchronized (this) */{
			if (i == 1)
			/* synchronized (this) */{
				return --i;
			}
			return i;
		}
	}
}
```

Pola finalne
„Pól finalnych (ostatecznych) nie można modyfikować (choć obiekty, do których zawierają referencje, mogą się zmieniać). Co więcej, mają one specjalne znaczenie w modelu pamięci Javy. To użycie pól finalnych gwarantuje bezpieczeństwo inicjalizacyjne (...), które umożliwia swobodny dostęp i współdzielenie obiektów niezmiennych."

Obiekty niezmienne
„Obiekty niezmienne [5] są ze swojej natury bezpieczne dla wątków - nie wymagają synchronizacji. Nie mogą zostać uszkodzone przez wiele wątków, odwołujących się do nich równolegle."

„Obiekt jest niezmienny, jeżeli [1]:
- jego stanu nie można zmienić po zakończeniu działania konstruktora,
- wszystkie jego pola są typu final,
- jest poprawnie utworzony (referencja this nie ucieknie w trakcie konstrukcji)."

„Klasa niezmienna [5] to klasa, której obiekty nie mogą być modyfikowane. Wszystkie dane zawarte w każdym obiekcie tej klasy są podawane w czasie tworzenia obiektu i pozostają niezmienne przez cały czas istnienia tego obiektu. (...) Aby utworzyć klasę niezmienną należy pamiętać o pięciu regułach [5]:
1. Nie twórz żadnych metod modyfikujących obiekt.
2. Upewnij się, że żadna metoda nie może zostać przesłonięta.
3. Zadeklaruj wszystkie pola jako final.
4. Zadeklaruj wszystkie pola jako private.
5. Zapewnij wyłączny dostęp do dowolnych modyfikowanych komponentów."

```
public class Vector {
	final private int x, y, z;
	final public static Vector UNIT_VECTOR_X = new Vector(1, 0, 0);
	final public static Vector UNIT_VECTOR_Y = new Vector(0, 1, 0);
	final public static Vector UNIT_VECTOR_Z = new Vector(0, 0, 1);

	private Vector(int x, int y, int z) {
		this.x = x;
		this.y = y;
		this.z = z;
	}

	public Vector newInstance(int x, int y, int z) {
		if (x == 1 && y == 0 && z == 0)
			return UNIT_VECTOR_X;
		if (x == 0 && y == 1 && z == 0)
			return UNIT_VECTOR_Y;
		if (x == 0 && y == 0 && z == 1)
			return UNIT_VECTOR_Z;
		return new Vector(x, y, z);
	}

	public int getX() {
		return x;
	}

	public int getY() {
		return y;
	}

	public int getZ() {
		return z;
	}
```

```
	public Vector add(Vector v) {
		return new Vector(this.x + v.x, this.y + v.y, this.z + v.z);
	}
}
```

W rzeczywistej implementacji najlepiej:
- dodać metody implementujące inne operacje na wektorach np. minus(Vector v),
- przesłonić metody toString(), hashCode(), equals(Object o),
- zaimplementować interfejs Comparable<Vector>.

Publikacja obiektu
„Aby bezpiecznie opublikować obiekt [1], zarówno referencje do obiektu, jak i jego stan muszą być widoczne dla innych wątków dokładnie w tym samym momencie. Poprawnie skonstruowany obiekt bezpiecznie publikuj przez:
- inicjalizacja referencji do obiektu z poziomu elementu static,
- przechowywanie referencji w polu volatile lub obiekcie AtomicRefence,
- przechowywanie referencji w polu final poprawnie utworzonego obiektu,
- przechowywanie referencji w polu poprawnie chronionym blokadą.”

Czy poniższy przykład zapewnia bezpieczną publikację [1]?

```
public Holder holder;

	...

public void initialze(){
	holder = new Holder(42);
}
```

„Jeżeli obiekt może zmieniać się po utworzeniu [1], bezpieczna publikacja zapewnia jedynie jego widoczność w stanie z momentu publikacji. Synchronizację trzeba wtedy stosować nie tylko w momencie publikacji obiektu, ale również przy każdym dostępie do niego, by w ten sposób zapewnić widoczność kolejnych zmian.”

Klasa Object
– `public final void wait()` – wątek zostaje zatrzymany, aż do wywołania metody `notify()` lub `notifyAll()`, istnieją także przesłonięte wersje tej metody ograniczające maksymalny czas czekania (`public final void wait(long timeout)`) oraz `public final void wait(long timeout, int nanos)`),
– `public final void notify()` oraz public `public final void notifyAll()`, budzi wątek (lub wątki) czekające na monitorze obiektu.
Metody `wait()` oraz `nofify()` mogą być wywoływane tylko wewnątrz bloku lub metody synchronizowanej.

Czy umieszczenie pętli wewnątrz bloku synchronizowanego w zmienia coś w działaniu programu?

```
class Buffer {
	int value;
}

class ReaderThread extends Thread {
	final private Buffer b;

	ReaderThread(Buffer b) {
		this.b = b;
	}
```

```
	public void run() {
		try {
			while (true) {
				synchronized (b) {
					System.err.println(">");
					b.wait();
					System.err.println(">> ReaderThread " + b.value);
					b.notify();
					System.err.println(">>>");
				}
			}
		} catch (InterruptedException e) {
			e.printStackTrace();
			System.exit(1);
		}
	}
}

class WriterThread extends Thread {

	final private Buffer b;
	private int i = 0;

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

	public void run() {
		try {
			while (true) {
				synchronized (b) {
					int t = ++i;
					b.value = t;
					Thread.sleep(t);
					System.err.println("< WriterThread " + b.value);
					b.notify();
					System.err.println("<<");
					b.wait();
					System.err.println("<<<");
				}
			}
		} catch (InterruptedException e) {
			e.printStackTrace();
			System.exit(1);
		}

	}
}

public class Test extends Thread {

	public static void main(String[] args) {
		Buffer buffer = new Buffer();
		WriterThread writer = new WriterThread(buffer);
		ReaderThread reader = new ReaderThread(buffer);
		reader.start();
		writer.start();
	}
}
```

Przykładowa treść laboratorium:

1. Stworzyć niezmienną klasę opisującą figurę geometryczną (np. prostokąt) zawierająca informacje o położeniu figury i jej wymiarach. Klasa powinna umożliwiać wykonywanie następujących operacji: obracania (o wielokrotność 90^o), rozciągania oraz przesuwania figury. Opracować program testujący użycie stworzonej klasy.
2. Stworzyć program zawierający minimalnie dwa wątki, które będą odczytywać i zapisywać współdzieloną zmienną typu long. Dostęp do zmiennej powinien być zrealizowany wykorzystując:
- modyfikator volatile,
- blok lub bloki synchronizowane,
- metodę lub metody synchronizowane,
- metody wait() oraz notify() klasy Object.

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