Programowanie współbieżne i rozproszone w języku Java

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Programowanie współbieżne i rozproszone w języku Java

Przemysław Stpiczyński
Marcin Brzuszek

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Przemysław Stpiczyński
(Instytut Matematyki UMCS, Instytut Informatyki Teoretycznej i Stosowanej PAN)
Marcin Brzuszek

PROGRAMOWANIE WSPÓŁBIEŻNE I ROZPROSZONE W JĘZYKU JAVA

Recenzent: Andrzej Bobyk

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Spis treści

Przedmowa **1**

1 Podstawowe pojęcia dotyczące współbieżności **3**
1.1. Programowanie współbieżne 4
1.2. Poprawność programów 9
1.3. Problem wzajemnego wykluczania 15
1.4. Zadania 22

2 Semafory **27**
2.1. Definicja i własności 28
2.2. Problem *producent* - *konsument* 31
2.3. Problem ucztujących filozofów 33
2.4. Zadania 38

3 Monitory **43**
3.1. Podstawowe pojęcia 44
3.2. Problem czytelników i pisarzy 47
3.3. Zadania 51

4 Wybrane techniki **53**
4.1. Blokady 54
4.2. Operacje RMW 57
4.3. Zadania 59

5 Programowanie rozproszone **61**
5.1. Remote Method Invocation 62
5.2. Standard CORBA 69
5.3. Aplikacja do prowadzenia rozmów 74
5.4. Zadania 80

6 Rozwiązania zadań **83**
6.1. Podstawowe pojęcia dotyczące współbieżności 85
6.2. Semafory 97

6.3. Monitory 115
6.4. Wybrane techniki 128
6.5. Programowanie rozproszone 138

BIBLIOGRAFIA **143**

Przedmowa

Konstrukcja poprawnych programów współbieżnych i rozproszonych stanowi jedno z najważniejszych wyzwań programistycznych. Na polskim rynku wydawniczym znajduje się szereg pozycji traktujących o programowaniu współbieżnym [1–3,7,8]. Na szczególną uwagę zasługują podstawowe pozycje autorstwa Ben-Ari'ego [1–3], w których wykorzystywany jest przestarzały już język Concurrent Pascal oraz mało popularny w Polsce język Ada. Programista zainteresowany programowaniem w języku Java ma do dyspozycji materiały dostarczane wraz ze środowiskiem programistycznym JDK [9–11], w których często trudno jest odnaleźć najważniejsze idee dotyczące rozwiązywania problemów związanych ze współbieżnością.

Niniejszy skrypt zawiera materiały wykorzystywane w trakcie wykładu i ćwiczeń z przedmiotu *Programowanie współbieżne i rozproszone*, który prowadzony jest dla kierunków *Informatyka* oraz *Matematyka* na Wydziale Matematyki, Fizyki i Informatyki Uniwersytetu Marii Curie-Skłodowskiej w Lublinie. Najważniejsze mechanizmy i podejścia do rozwiązywania problemów związanych ze współbieżnością i programowaniem rozproszonym zostały zaprezentowane na przykładzie języka Java.

Skrypt stanowi uzupełnienie najważniejszych pozycji książkowych poświęconych programowaniu współbieżnemu, głównie dzięki licznym zadaniom, które opatrzone zostały przykładowymi rozwiązaniami. Czytelników zainteresowanych poszerzeniem wiadomości zachęcamy do zapoznania się z książkami [6, 7].

Rozdział 1

Podstawowe pojęcia dotyczące współbieżności

1.1. Programowanie współbieżne **4**
1.1.1. Procesy i wątki w języku Java 4
1.1.2. Komunikacja poprzez przerwania 7
1.2. Poprawność programów **9**
1.2.1. Przeplot i atomowość instrukcji 9
1.2.2. Metody i instrukcje synchronizowane 13
1.2.3. Bezpieczeństwo i żywotność 14
1.3. Problem wzajemnego wykluczania **15**
1.4. Zadania **22**
1.4.1. Nazywanie wątków w przypadku dziedziczenia po klasie `Thread` 22
1.4.2. Nazywanie wątków w przypadku rozszerzania interfejsu `Runnable` 23
1.4.3. Algorytm Dekkera 23
1.4.4. Algorytm Dorana-Thomasa 24
1.4.5. Algorytm Petersona 24
1.4.6. Algorytm Petersona dla N wątków 25
1.4.7. Szeregowanie instrukcji 26
1.4.8. Tablica wątków 26

W niniejszym rozdziale przedstawimy podstawowe pojęcia związane z programowaniem współbieżnym oraz realizacją współbieżności w języku Java [6]. Więcej informacji na ten temat można znaleźć w książkach [1, 2], gdzie do prezentacji zagadnień został wykorzystany język Ada [12].

1.1. Programowanie współbieżne

Pod pojęciem *proces* będziemy rozumieli program wykonywany na komputerze pod kontrolą systemu operacyjnego. W chwili startu procesowi przydzielane są potrzebne zasoby (pamięć, czas procesora, dostęp do urządzeń wejścia-wyjścia). Współczesne systemy komputerowe wykonują jednocześnie (współbieżnie, równolegle) wiele różnych procesów, które współzawodniczą ze sobą w dostępie do zasobów kontrolowanych przez system operacyjny, co określane jest mianem *wielozadaniowości*.

W praktyce wiele ważnych problemów może być efektywnie rozwiązywanych w postaci *programów współbieżnych*, czyli programów napisanych w pewnym języku programowania w taki sposób, że ich wykonanie jest realizowane jako grupa powiązanych ze sobą procesów współbieżnych. Takie powiązanie będzie na ogół polegać na współzawodnictwie w dostępie do pewnego zasobu systemu komputerowego oraz komunikacji pomiędzy procesami. Problem organizacji dostępu do zasobów będzie wymagał synchronizacji działania procesów. Komunikacja między procesami będzie jedną z metod rozwiązywania problemów związanych z synchronizacją procesów.

Terminem *programowanie współbieżne* będziemy określać techniki i notacje programistyczne używane dla wyrażenia potencjalnej równoległości wykonania więcej niż jednego procesu oraz służące rozwiązywaniu zagadnień i problemów związanych z synchronizacją i komunikacją procesów wykonywanych współbieżnie [1].

1.1.1. Procesy i wątki w języku Java

W przypadku języka Java mamy do czynienia z dwoma programowymi typami jednostek kodu wyrażającymi współbieżność. Są to *procesy* oraz *wątki*. Komunikacja pomiędzy procesami może być realizowana przy pomocy wspieranych przez system operacyjny mechanizmów komunikacji. Listing 1.1 prezentuje możliwość tworzenia nowych procesów przy pomocy klas `Process` oraz `ProcessBuilder`.

Listing 1.1. Tworzenie procesów w języku Java

```
1 try{
2   Process p =
3     new ProcessBuilder("c:/apps/bin/putty.exe", "").start();
4 }catch(Exception e){
5   System.out.println("Błąd: "+e.getMessage());
6 }
```

Wątki (ang. *threads*), zwane czasami *lekkimi procesami* (ang. *lightweiht processes*) są znacznie wygodniejszym narzędziem do wyrażenia współbieżności w języku Java. Istnieją zawsze w ramach pewnego procesu (każdy proces ma przynajmniej jeden wątek), a zatem mają dostęp do pamięci procesu. Ich tworzenie wymaga mniejszej ilości zasobów. Język Java dostarcza wygodne mechanizmy do tworzenia oraz zarządzania wątkami (zostaną one przedstawione w dalszej części niniejszego opracowania). Rozważmy przedstawioną na listingu 1.2 definicję klasy `Watek`. Dziedziczy ona po klasie `Thread`. W konstruktorze możemy przekazać parametr typu `String`, który posłuży nam do własnego numerowania tworzonych wątków, czyli obiektów klasy `Watek`. Metoda `run()` zawiera instrukcje – kod wątku. Ich zadaniem jest wyświetlenie na ekranie komputera stosownego komunikatu wraz z numerem wątku. Wykorzystujemy tutaj klasę `ThreadID` (listing 1.3), w której zdefiniowaliśmy statyczną metodę `get()` zwracającą numer wątku ustalony przy tworzeniu obiektu.

Listing 1.2. Prosta klasa wątku

```
1 public class Watek extends Thread {
2
3   public Watek(String num){
4     super(num);
5   }
6   public void run(){ // instrukcje wykonywane
7                      // w ramach wątku
8     System.out.println("Pozdrowienia z wątku "
9                                   + ThreadID.get());
10  }
11 }
```

Listing 1.3. Klasa do identyfikacji wątków

```
1 public class ThreadID {
2   public static int get() {
3     return Integer.parseInt(
4         Thread.currentThread().getName());
5  }
6 }
```

Listing 1.4 zawiera przykładowy prosty program, którego zadaniem jest utworzenie dziesięciu obiektów klasy `Watek` (linie 7–9), gdzie każdy obiekt otrzymuje numer - kolejną liczbę całkowitą począwszy od zera) oraz uruchomienie utworzonych wątków (linie 11–13). Poszczególne wątki wykonywane są współbieżnie z wątkiem głównym, który definiuje kod metody `main()`.

Listing 1.4. Tworzenie i start wątków

```
1 public class Main {
2
3   public static void main(String[] args) {
4
5     Thread[] w = new Thread[10];
6
7     for(int i = 0; i < 10; i++){
8       w[i] = new Watek("" + i);
9     }
10
11    for(int i = 0; i < 10; i++) {
12      w[i].start();
13    }
14  }
15 }
```

Wykonanie programu spowoduje wyświetlenie komunikatów następującej postaci:

```
Pozdrowienia z wątku 7
Pozdrowienia z wątku 2
Pozdrowienia z wątku 5
Pozdrowienia z wątku 0
Pozdrowienia z wątku 6
Pozdrowienia z wątku 3
Pozdrowienia z wątku 8
Pozdrowienia z wątku 4
Pozdrowienia z wątku 9
Pozdrowienia z wątku 1
```

Oczywiście jest to tylko przykładowy wynik. Należy pamiętać, że wątki wykonują się współbieżnie, zatem kolejność wyświetlania przez nie komunikatów na ekran będzie przypadkowa.

W przypadku, gdy klasa typu „wątek” dziedziczy już po innej klasie nie ma możliwości zastosowania bezpośredniego dziedziczenia po klasie `Thread`. Można wówczas zdefiniować wątek jako klasę implementującą interfejs `Runnable` i posłużyć się nieco inną metodą tworzenia obiektów - wątków. Ilustrują to listingi 1.5 oraz 1.6.

Listing 1.5. Klasa implementująca interfejs Runnable

```
1 public class Watek implements Runnable {
2
3   public void run(){ // instrukcje wykonywane
4                      // w ramach wątku
5     System.out.println("Pozdrowienia z wątku "
6                                   + ThreadID.get());
7   }
8 }
```

Listing 1.6. Tworzenie i start wątków (Runnable)

```
1 public class Main {
2
3   public static void main(String[] args) {
4
5     Thread[] w = new Thread[10];
6
7     for(int i = 0; i < 10; i++){
8       w[i] = new Thread(new Watek(),"" + i);
9     }
10
11    for(int i = 0; i < 10; i++) {
12      w[i].start();
13    }
14  }
15 }
```

1.1.2. Komunikacja poprzez przerwania

W języku Java istnieje prosta możliwość komunikowania się wątków poprzez przerwania. Wątek na listingu 1.7 powtarza dziesięciokrotnie czekanie przez dwie sekundy (statyczna metoda `sleep()` klasy `Thread`). Jeśli w czasie „drzemki" otrzyma sygnał *interrupt*, wówczas wykonanie metody `sleep(2000)` kończy się wyrzuceniem wyjątku `InterruptedException`, którego obsługa polega tutaj na wyświetleniu stosownego komunikatu oraz zakończeniu wykonywania pętli (instrukcja `break`).

Listing 1.7. Obsługa przerwań

```
1 public class Watek extends Thread {
2
3   public void run(){
4
5     for(int i=0;i<10;i++){
6       try{
7         Thread.sleep(2000);
```

```
 8                System.out.println("Spałem 2 sekundy");
 9            }catch(InterruptedException e){
10                System.out.println("Dostałem sygnał interrupt");
11                break;
12            }
13        }
14    }
15 }
```

Listing 1.8 demonstruje możliwości wysyłania przerwań. W liniach 6–7 jest tworzony i uruchamiany wątek. Następnie po upływie pięciu sekund (linia 8) wątek metody `main()` wysyła sygnał *interrupt* (linia 9). Następnie czeka za zakończenie działania wątku reprezentowanego w zmiennej `w` (wywołanie metody `join()`) i wyświetla komunikat o zakończeniu pracy.

Listing 1.8. Obsługa przerwań

```
 1 public class Main {
 2
 3     public static void main(String[] args)
 4                                          throws Exception {
 5
 6         Thread w = new Watek();
 7         w.start();
 8         Thread.sleep(5000);
 9         w.interrupt();
10         w.join();
11         System.out.println("KONIEC");
12     }
13 }
```

Wykonanie programu spowoduje wyświetlenie następujących komunikatów:

```
spałem 2 sekundy
spałem 2 sekundy
Ktoś wysłał mi sygnał interrupt
KONIEC
```

Wątek dwukrotnie wykonał metodę `Thread.sleep(2000)`. W trakcie trzeciego wykonania otrzymał sygnał *interrupt*, obsłużył wyjątek przerywając wykonanie pętli.

Warto tutaj wspomnieć, że programista ma do dyspozycji również inne metody związane z obsługą przerwań.

```
1 public static boolean interrupted()
2 public boolean isInterrupted()
```

Pierwsza z nich służy do badania, czy bieżący wątek otrzymał sygnał *interrupt* i jeśli tak, wówczas kasuje ten sygnał, zaś druga bada stan przerwania innego wątku bez kasowania sygnału.

1.2. Poprawność programów

Wykonanie programu współbieżnego będzie polegać na przeplataniu ze sobą instrukcji działających współbieżnie procesów lub wątków. Będzie się to działo niezależnie od tego, czy mamy do czynienia z systemem wieloprocesorowym, czy też z klasycznym komputerem wyposażonym w jeden procesor.

1.2.1. Przeplot i atomowość instrukcji

Przeplotem będziemy nazywać model wykonania programów współbieżnych umożliwiający ich analizę. Przypuśćmy, że wykonanie programu współbieżnego P składa się z dwóch procesów (wątków) P_0 oraz P_1. Wówczas:
— konkretne wykonanie programu P jest jednym z wykonań jakie można otrzymać przeplatając ze sobą instrukcje procesów P_0 i P_1,
— zbiór wszystkich takich przeplotów wyczerpuje zbiór wszystkich możliwych zachowań programu P.

W konkretnym wykonaniu programu nie można przewidzieć jak będzie wyglądał ciąg przeplatanych ze sobą instrukcji. Należy przyjąć, że będzie on generowany w sposób niedeterministyczny. Oczywiście w przypadku nietrywialnego programu liczba możliwych przeplotów może być bardzo duża i nie ma możliwości przeanalizowania wszystkich zachowań programu. Zwykle wskazuje się przeplot, dla którego nie są spełnione wymagane własności programu.

Generalnie dostęp działających współbieżnie wątków do danych alokowanych we współdzielonej pamięci niesie ze sobą dwa rodzaje problemów:
— nakładanie się wątków wykonujących operacje na tych samych danych (ang. *thread interference*),
— błędy związane ze spójnością obrazu pamięci (ang. *memory consistency errors*).

Z pojęciem przeplotu jest ściśle związany problem ustalenia, które instrukcje mogą być ze sobą przeplatane, a które są *atomowe* i niepodzielne, czyli ich wykonanie nie może być „przeplecione” z innym wątkiem. Jako przykład rozważmy przedstawioną na listingu 1.9 klasę `Licznik`.

Listing 1.9. Klasa Licznik (sekwencyjny)

```
1 public class Licznik {
2
3     private long c=0;
4
5     public  void inc(){
6         c++;
7     }
8     public long get(){
9         return c;
10    }
11 }
```

Działanie obiektu tej klasy w środowisku współbieżnym nie będzie poprawne. Istotnie, wykonanie programu z listingu 1.10 nie spowoduje na ogół wyświetlenia wartości 1000000. Dzieje się tak, gdyż (między innymi) operacja zwiększenia wartości zmiennej oraz operacje odczytu i zapisu wartości typu `long` nie są atomowe.

Listing 1.10. program współbieżny korzystający z klasy Licznik

```
1 class Watek extends Thread {
2
3     private Licznik l;
4
5     public Watek(Licznik l){
6         this.l=l;
7     }
8     public void run(){
9         for(int i=0;i<500000;i++){
10            l.inc();
11        }
12    }
13 }
14
15 public class Main {
16
17     public static void main(String[] args) throws Exception{
18
19         Licznik l=new Licznik();
20
21         Thread w0=new Watek(l);
22         Thread w1=new Watek(l);
23
24         w0.start();
25         w1.start();
26
27         w0.join();
28         w1.join();
29
```

```
30            System.out.println("Licznik="+l.get());
31        }
32 }
```

Następujące odwołania do zmiennych są operacjami atomowymi:
— odczyt i zapis do zmiennych typów referencyjnych oraz typów prostych z wyjątkiem `long` oraz `double`,
— odczyt i zapis do zmiennych zadeklarowanych ze słowem kluczowym `volatile`.

Problemy związane ze spójnością obrazu pamięci występują wtedy, gdy dwa lub więcej wątków „widzi” inaczej pewne dane. Jeśli jeden wątek modyfikuje pole pewnego obiektu, a następnie inny wątek odczytuje wartość tego pola, to powinien on widzieć dokonaną modyfikację. Będzie to miało miejsce wtedy, gdy pomiędzy tymi dwoma zdarzeniami będzie zachodziła relacja *następstwa czasowego* (ang. *happens-before*). Wystąpi to w kilku przypadkach. Najważniejsze z nich są następujące [6]:
— gdy uruchamiany jest nowy wątek (wywołanie metody `start()`), wówczas wszystkie instrukcje, które są w relacji *happens-before* z instrukcją wywołującą metodę `start()` są w takiej samej relacji z instrukcjami w wątku rozpoczynającym działanie,
— gdy wątek kończy działanie i powoduje to zakończenie wykonywania metody `join()`, wówczas każda z instrukcji w wątku kończącym działanie jest w relacji następstwa czasowego z każdą instrukcją wątku po instrukcji wywołującej metodę `join()`,
— zapis do zmiennych zadeklarowanych ze słowem kluczowym `volatile` ustanawia relację następstwa czasowego z następującymi po tym operacjami odczytu wartości takiej zmiennej, czyli modyfikacje zmiennych `volatile` są widoczne dla wszystkich wątków.

W API języka Java dostępny jest pakiet `java.util.concurrent.atomic`, w którym dostępne są między innymi klasy `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicIntegerArray`, `AtomicLongArray`. Oferują one atomową realizację ważnych operacji na danych poszczególnych typów oraz atomowy dostęp do składowych tablic. Przykładowo listing 1.11 pokazuje najważniejsze operacje dla klasy `AtomicInteger`, zaś listing 1.12 zawiera najważniejsze operacje na tablicach o atomowych składowych typu całkowitego.

Listing 1.11. Klasa AtomicInteger

```
1 int       addAndGet(int delta)
2 // atomowo dodaje wartość do zmiennej i zwraca
3 // wartość po aktualizacji
4 boolean       compareAndSet(int expect, int update)
5 // atomowo ustawia wartość na update o ile bieżąca
6 // wartość jest równa expect
```

```
 7 int       decrementAndGet()
 8 // atomowo odejmuje  wartość 1 od zmiennej i zwraca
 9 // wartość po aktualizacji
10 int       get()
11 // zwraca bieżącą wartość
12 int       getAndAdd(int delta)
13 // atomowo zwraca bieżącą wartość i dodaje wartość delta
14 int       getAndDecrement()
15 // atomowo zwraca bieżącą wartość i zmniejsza o jeden
16 int       getAndIncrement()
17 // atomowo zwraca bieżącą wartość i zwiększa o jeden
18 int       getAndSet(int newValue)
19 // atomowo zwraca bieżącą wartość i ustawia nową
20 int       incrementAndGet()
21 // atomowo zwiększa o jeden i zwraca wartość
22 void      lazySet(int newValue)
23 // ustawia nową wartość (o ile jest różna od bieżącej)
24 void      set(int newValue)
25 // atomowo ustawia zadaną wartość
```

Listing 1.12. Klasa AtomicIntegerArray

```
 1 AtomicIntegerArray(int length)
 2 // tworzy tablicę o danej liczbie składowych
 3 // zainicjowanych wartościami 0
 4 AtomicIntegerArray(int[] array)
 5 // tworzy tablicę o danej liczbie składowych
 6 // zainicjowanych wartościami przekazanymi w tablicy
 7 int       addAndGet(int i, int delta)
 8 // operacja addAndGet dla składowej  o numerze i
 9 boolean       compareAndSet(int i, int expect, int update)
10 // operacja compareAndSet dla składowej  o numerze i
11 int       decrementAndGet(int i)
12 // operacja compareAndSet dla składowej  o numerze i
13 int       get(int i)
14 //  zwraca składowej  o numerze i
15 int       getAndAdd(int i, int delta)
16 // operacja getAndAdd składowej  o numerze i
17 int       getAndDecrement(int i)
18 // operacja getAndDecrement składowej  o numerze i
19 int       getAndIncrement(int i)
20 // operacja getAndIncrement dla składowej  o numerze i
21 int       getAndSet(int i, int newValue)
22 // operacja getAndSet składowej  o numerze i
23 int       incrementAndGet(int i)
24 // operacja incrementAndGet dla składowej  o numerze i
25 int       length()
26 // zwraca liczbę składowych
27 void      set(int i, int newValue)
28 // ustawia wartość składowej  o numerze i
```

1.2.2. Metody i instrukcje synchronizowane

W języku Java dostępne jest bardzo wygodne narzędzie do synchronizacji oraz zapewniania spójności we współbieżnym dostępie do danych. Stanowią je metody oraz instrukcje synchronizowane (ang. *synchronized methods*, *synchronized statements*). Rozważmy następującą (listing 1.13) definicję klasy `Licznik`, gdzie metody zostały zadeklarowane ze słowem kluczowym `synchronized`.

Listing 1.13. Klasa Licznik (współbieżny)

```
1 public class Licznik {
2
3     private long c=0;
4
5     public synchronized  void inc(){
6         c++;
7     }
8
9     public synchronized long get(){
10        return c;
11    }
12 }
```

Dzięki zadeklarowaniu metod jako synchronizowane, wykonanie metody będzie się odbywało jako instrukcja atomowa. Jeśli pewien wątek chce rozpocząć wykonywanie metody synchronizowanej, wówczas będzie oczekiwać na zakończenie wykonania metod synchronizowanych przez inny wątek wykonujący metodę synchronizowaną. Zakończenie wykonania metody synchronizowanej ustanawia relację następstwa czasowego pomiędzy instrukcją wywołującą daną metodę, a odwołaniami do obiektu, na rzecz którego wykonano metodę. Oznacza to, że modyfikacje dokonane w metodzie synchronizowanej są widoczne dla wszystkich wątków. Użycie instrukcji synchronizowanych daje nam jeszcze ciekawsze możliwości. Rozważmy definicję klasy `Licznik` przedstawioną na listingu 1.14 [6].

Listing 1.14. Klasa Licznik (współbieżny)

```
1 public class Licznik {
2
3     private long c1=0;
4     private long c2=0;
5
6     private Object lock1=new Object();
7     private Object lock2=new Object();
8
9     public  void inc1(){
10      synchronized(lock1){
```

```
11            c1++;
12         }
13      }
14
15      public  void inc2(){
16         synchronized(lock2){
17            c2++;
18         }
19      }
20
21   }
```

Metody **inc1()** oraz **inc2()** nie są synchronizowane, a zatem ich wykonanie będzie mogło być realizowane współbieżnie. Nie stanowi to problemu, gdyż każda z metod odwołuje się do innych pól. Jednoczesne wykonanie tej samej metody przez dwa różne wątki jest potencjalnie niebezpieczne (dostęp do tego samego pola), ale modyfikacja pola będzie się odbywała w trybie wzajemnego wykluczania, gdyż jest realizowane jako instrukcja synchronizowana i jej wykonanie wymaga wyłącznego dostępu wątku do obiektu. Jest to przykład użycia *monitorów*, które szerzej omówimy w rozdziale 3.

1.2.3. Bezpieczeństwo i żywotność

W przypadku programów sekwencyjnych mówimy o dwóch rodzajach poprawności programów [14]. Program nazywamy *lokalnie poprawnym*, gdy zgodność danych wejściowych ze specyfikacją oraz zakończenie obliczeń implikują zgodność ze specyfikacją wyników. W tym przypadku abstrahujemy od własności sprzętowo-programowych komputera i jego specyficznych właściwości (na przykład własności arytmetyki liczb zmiennopozycyjnych). Program nazywamy *globalnie poprawnym*, gdy zgodność danych wejściowych ze specyfikacją implikuje zakończenie obliczeń oraz to, że wyniki będą spełniały określone własności.

W przypadku programów współbieżnych mamy często do czynienia z programami działającymi w pętlach nieskończonych i w takim przypadku nie można mówić o zakończeniu obliczeń. Pojęciu poprawności lokalnej będzie odpowiadać pojęcie *bezpieczeństwa*, zaś poprawności globalnej pojęcie *żywotności* [1].

Definicja 1.1. Bezpieczeństwem *nazywamy własność programu, która jest zawsze spełniona.*

Definicja 1.2. Żywotnością *nazywamy własność programu, która w chwili obecnej jest spełniona lub będzie spełniona kiedyś w przyszłości.*

Konsekwencją braku żywotności może być *zakleszczenie* (ang. *deadlock*), gdy żaden wątek nie jest w stanie kontynuować swojego działania, bądź też *zagłodzenie* wątku, z którym mamy do czynienia, gdy wątek nie może kontynuować swojego działania. Przykłady ilustrujące brak żywotności podamy w następnym podrozdziale.

1.3. Problem wzajemnego wykluczania

Jako przykłady spełnienia własności bezpieczeństwa i żywotności oraz konsekwencje braku ich spełnienia rozważmy następujący problem wzajemnego wykluczania [1]:

1. Program składa się z $n > 1$ wątków działających współbieżnie, przy czym każdy wątek działa w pętli nieskończonej wykonując kolejno instrukcje *sekcji lokalnej* i *sekcji krytycznej.*
2. Wymaga się aby instrukcje sekcji krytycznych poszczególnych wątków nie przeplatały się ze sobą.
3. W celu zapewnienia realizacji wzajemnego wykluczania wykonań sekcji krytycznych, przed wejściem do sekcji krytycznej wykonywane są instrukcje zwane *protokołem wstępnym*, zaś po sekcji krytycznej instrukcje *protokołu końcowego.*
4. Wątek może sié zatrzymać wyłącznie w sekcji lokalnej i nie może to zakłócić działania pozostałych wątków.
5. Nie może wystąpić zakleszczenie - w przypadku współzawodnictwa przy wejściu do sekcji krytycznej, przynajmniej jeden wątek musi do niej wejść.
6. Żaden wątek nie może zostać zagłodzony - jeśli wątek chce wejść do sekcji krytycznej, kiedyś musi to nastąpić.
7. Przy braku współzawodnictwa przy wchodzeniu do sekcji krytycznej, wątek który che to uczynić powinien wejść jak najszybciej.

Własność bezpieczeństwa oznacza tutaj, że sekcje krytyczne będą wykonywane w trybie wzajemnego wykluczania, zaś żywotność to, że każdy wątek będzie mógł po raz kolejny wejść do sekcji krytycznej. Przejawem braku żywotności jest zagłodzenie wątku, który mimo chęci nie będzie mógł wejść do sekcji krytycznej oraz zakleszczenie, gdy żaden wątek nie będzie mógł tego uczynić.

Tak jak w książkach Ben-Ari'ego [1–3] rozważymy teraz cztery próby rozwiązania problemu wzajemnego wykluczania dla dwóch wątków, które będą ilustrować różne problemy dotyczące poprawności programów współbieżnych. Pierwsza próba (listing 1.15) zakłada, że zmienna `czyjaKolej` rozstrzyga o kolejności wchodzenia do sekcji krytycznych. Wątek „czeka

w pętli” w protokole wstępnym na swoją kolej. Po wyjściu z sekcji krytycznej przekazuje drugiemu wątkowi prawo do wejścia.

Listing 1.15. Problem wzajemnego wykluczania (pierwsza próba)

```
 1 public class Main implements Runnable {
 2
 3   volatile static int czyjaKolej = 0;
 4   private int mojNum;
 5
 6   public void run() {
 7
 8     mojNum = ThreadID.get();
 9
10     while (true) {
11
12       // sekcja lokalna
13       try {
14         Thread.sleep((long) (2500 * Math.random()));
15       } catch (InterruptedException e) {
16       }
17
18       // protokół wstępny
19       while (czyjaKolej != mojNum) {
20         Thread.yield();
21       }
22       // sekcja krytyczna
23       System.out.println("Wątek " + mojNum + " start SK");
24       try {
25         Thread.sleep((long) (2100 * Math.random()));
26       } catch (InterruptedException e) {
27       }
28       System.out.println("Wątek " + mojNum + " stop SK");
29       // protokół końcowy
30       czyjaKolej = 1 - mojNum;
31     }
32   }
33
34   public Main() {
35   }
36
37   public static void main(String[] args) {
38
39     new Thread(new Main(), "0").start();
40     new Thread(new Main(), "1").start();
41   }
42 }
```

Można łatwo wykazać następujące własności.

Własność 1.1. *Rozwiązanie „pierwsza próba" (listing 1.15) ma własność wzajemnego wykluczania.*

Dowód. Przypuśćmy, że rozwiązanie nie ma własności wzajemnego wykluczania i gdy wątek 1 wchodził do sekcji krytycznej w chwili t_1, wątek 0 już w niej był począwszy od chwili t_0, przy czym $t_0 < t_1$. Wątek 0 wchodząc do sekcji krytycznej musiał odczytać wartość zmiennej `czyjaKolej` równą 0. W odcinku czasu $[t_0, t_1]$ pozostawał w sekcji krytycznej, a zatem nie mógł zmienić wartości zmiennej `czyjaKolej` na 1. W chwili t_1, gdy wątek wchodził do sekcji krytycznej musiał odczytać `czyjaKolej==1`. Otrzymujemy zatem sprzeczność. ■

Własność 1.2. *W rozwiązaniu „pierwsza próba" (listing 1.15) nie wystąpi zakleszczenie.*

Dowód. Przypuśćmy, że w rozwiązaniu wystąpiło zakleszczenie. Oznacza to, że oba wątki wykonują w nieskończoność pętle w protokołach wstępnych. Zatem wątek 0 odczytuje w nieskończoność `czyjaKolej==1`, zaś wątek 1 odczytuje `czyjaKolej==0`, czyli otrzymujemy sprzeczność. ■

Własność 1.3. *W rozwiązaniu „pierwsza próba" (listing 1.15) nie wystąpi zagłodzenie.*

Dowód. Przypuśćmy, że w rozwiązaniu wystąpiło zagłodzenie wątku 0. Oznacza to, że wątek 0 wykonuje w nieskończoność pętlę w protokole wstępnym odczytując w nieskończoność `czyjaKolej==1`, zaś wątek 1 wchodzi cyklicznie do sekcji krytycznej. Otrzymujemy sprzeczność, gdyż w protokole końcowych wątek 1 ustawi wartość zmiennej `czyjaKolej` na 0. ■

Rozwiązanie ma jednak dwie istotne wady:

— wątki muszą wchodzić do sekcji krytycznych naprzemiennie, zatem ten, który chce wchodzić częściej i tak będzie musiał czekać na wolniejszego,
— jeśli jeden wątek zakończy działanie w sekcji lokalnej, drugi nie będzie mógł wchodzić do sekcji krytycznej.

Rozważmy zatem drugą próbę (listing 1.16). Mamy tutaj tablicę o dwóch składowych, które informują o tym, czy wątek jest w sekcji krytycznej. Zatem w protokole wstępnym wątek sprawdza, czy drugi jest poza sekcją krytyczną i jeśli tak jest, wchodzi do swojej sekcji krytycznej. W rozwiązaniu może jednak dojść do tego, że oba wątki wejdą do sekcji krytycznych. Istotnie, gdy wątek wyjdzie z pętli w protokole wstępnym, faktycznie już jest w sekcji krytycznej. Zatem oba wątki po odczytaniu wartości zmiennych równych 1, wchodzą do sekcji krytycznych.

Listing 1.16. Problem wzajemnego wykluczania (druga próba)

```
1  public class Main implements Runnable {
2
3    static AtomicIntegerArray k = new AtomicIntegerArray(2);
4
5    static {
6      k.set(0, 1);
7      k.set(1, 1);
8    }
9    private int mojNum;
10
11   public void run() {
12
13     mojNum = ThreadID.get();
14
15     while (true) {
16
17       // sekcja lokalna
18       try {
19         Thread.sleep((long) (2500 * Math.random()));
20       } catch (InterruptedException e) {
21       }
22
23       // protokół wstępny
24       while (k.get(1 - mojNum) == 0) {
25         Thread.yield();
26       }
27       k.set(mojNum, 0);
28       // sekcja krytyczna
29       System.out.println("Wątek " + mojNum + " start SK");
30       try {
31         Thread.sleep((long) (2100 * Math.random()));
32       } catch (InterruptedException e) {
33       }
34       System.out.println("Wątek " + mojNum + " stop SK");
35       // protokół końcowy
36       k.set(mojNum, 1);
37     }
38   }
39
40   public Main() {
41   }
42
43   public static void main(String[] args) {
44
45     new Thread(new Main(), "0").start();
46     new Thread(new Main(), "1").start();
47   }
48 }
```

Kolejna trzecia próba (listing 1.17) zawiera drobną poprawkę. Zanim wątek zacznie sprawdzać, czy drugi jest poza sekcją krytyczną, ustawia swoją zmienną na 0, rezerwując sobie tym samym prawo do wejścia do sekcji krytycznej. Można wskazać przeplot, w którym dojdzie do zakleszczenia. Stanie się tak, gdy każdy wątek ustawi wartość swojej zmiennej na 0 i będzie wykonywał w nieskończoność pętlę w protokole wstępnym. Można jednak wykazać następującą własność [1]:

Własność 1.4. *Rozwiązanie „trzecia próba” (listing 1.17) ma własność wzajemnego wykluczania.*

Listing 1.17. Problem wzajemnego wykluczania (trzecia próba)

```
1  public class Main implements Runnable {
2
3    static AtomicIntegerArray k = new AtomicIntegerArray(2);
4    private int mojNum;
5
6    static {
7      k.set(0, 1);
8      k.set(1, 1);
9    }
10
11   public void run() {
12
13     mojNum = ThreadID.get();
14
15     while (true) {
16
17       // sekcja lokalna
18       try {
19         Thread.sleep((long) (2500 * Math.random()));
20       } catch (InterruptedException e) {
21       }
22
23       // protokół wstępny
24       k.set(mojNum, 0);
25       while (k.get(1 - mojNum) == 0) {
26         Thread.yield();
27       }
28       // sekcja krytyczna
29       System.out.println("Wątek " + mojNum + " start SK");
30       try {
31         Thread.sleep((long) (2100 * Math.random()));
32       } catch (InterruptedException e) {
33       }
34       System.out.println("Wątek " + mojNum + " stop SK");
35       // protokół końcowy
36       k.set(mojNum, 1);
37     }
```

```
38    }
39
40    public Main() {
41    }
42
43    public static void main(String[] args) {
44
45      new Thread(new Main(), "0").start();
46      new Thread(new Main(), "1").start();
47    }
48  }
```

W czwartej próbie modyfikujemy rozwiązanie tak, by wątek, który odczytał, że drugi chce wejść lub już jest w sekcji krytycznej, na pewien czas wycofał się. Otrzymujemy w ten sposób listing 1.18.

Listing 1.18. Problem wzajemnego wykluczania (czwarta próba)

```
1  public class Main implements Runnable {
2
3    static AtomicIntegerArray k = new AtomicIntegerArray(2);
4    private int mojNum;
5
6    static {
7      k.set(0, 1);
8      k.set(1, 1);
9    }
10
11   public void run() {
12
13     mojNum = ThreadID.get();
14
15     while (true) {
16
17       // sekcja lokalna
18       try {
19         Thread.sleep((long) (2500 * Math.random()));
20       } catch (InterruptedException e) {
21       }
22
23       // protokół wstępny
24       k.set(mojNum, 0);
25       while (k.get(1 - mojNum) == 0) {
26         k.set(mojNum, 1);
27         Thread.yield();
28         k.set(mojNum, 0);
29       }
30       // sekcja krytyczna
31       System.out.println("Wątek " + mojNum + " start SK");
32       try {
```

```
33            Thread.sleep((long) (2100 * Math.random()));
34          } catch (InterruptedException e) {
35          }
36          System.out.println("Wątek " + mojNum + " stop SK");
37          // protokół końcowy
38          k.set(mojNum, 1);
39        }
40      }
41
42      public Main() {
43      }
44
45      public static void main(String[] args) {
46
47        new Thread(new Main(), "0").start();
48        new Thread(new Main(), "1").start();
49      }
50    }
```

Poprawnym rozwiązaniem problemu jest algorytm Dekkera, który można uznać za połączenie próby pierwszej i czwartej (listing 1.19).

Listing 1.19. Problem wzajemnego wykluczania – algorytm Dekkera

```
1
2
3  public class Dekker implements Runnable {
4
5    static AtomicIntegerArray k = new AtomicIntegerArray(2);
6    static volatile int czyjaKolej = 0;
7    int mojNum;
8
9    static {
10     k.set(0, 1);
11     k.set(1, 1);
12   }
13
14   public void run() {
15
16     mojNum = ThreadID.get();
17
18     while (true) {
19
20       // sekcja lokalna
21       try {
22         Thread.sleep((long) (2500 * Math.random()));
23       } catch (InterruptedException e) {
24       }
25
26       // protokół wstępny
27       k.set(mojNum, 1);
```

```
28         while (k.get(1 - mojNum) == 1) {
29
30           if (czyjaKolej != mojNum) {
31             k.set(mojNum, 0);
32             while (czyjaKolej != mojNum) {
33               Thread.yield();
34             }
35             k.set(mojNum, 1);
36           }
37         }
38
39         // sekcja krytyczna
40         System.out.println("Wątek " + mojNum + " start SK");
41         try {
42           Thread.sleep((long) (1000 * Math.random()));
43         } catch (InterruptedException e) {
44         }
45         System.out.println("Wątek " + mojNum + " stop SK");
46
47         // protokół końcowy
48         czyjaKolej = 1 - mojNum;
49         k.set(mojNum, 0);
50       }
51     }
52
53     public Dekker() {
54     }
55
56     public static void main(String[] args) {
57
58       new Thread(new Dekker(), "0").start();
59       new Thread(new Dekker(), "1").start();
60     }
61   }
```

1.4. Zadania

1.4.1. Nazywanie wątków w przypadku dziedziczenia po klasie `Thread`

Zdefiniuj klasę wątku jako rozszerzenie klasy `Thread`. Utwórz trzy wątki nadając im nazwy: *Watek1*, *Watek2* oraz *Watek3*. Wystartuj wszystkie trzy wątki. Niech każdy z nich wypisze swoją nazwę.

1.4.2. Nazywanie wątków w przypadku rozszerzania interfejsu `Runnable`

Zdefiniuj klasę wątku jako implementację interfejsu `Runnable`. Utwórz trzy wątki nadając im nazwy: *Zadanie1*, *Zadanie2* oraz *Zadanie3*. Uruchom tak utworzone wątki. Niech każdy wątek, łącznie z wątkiem głównym, wypisze swoją nazwę.

1.4.3. Algorytm Dekkera

Zaimplementuj algorytm Dekkera rozwiązujący problem wzajemnego wykluczania dla dwóch wątków, ale bez użycia klasy `AtomicIntegerArray`. Posłuż się poniższym pseudokodem [3, 5].

```
var
  chce1: boolean := false;
  chce2: boolean := false;
  kto_czeka: 1..2 := 1;
```

```
thread w1;
begin
  while true do
  begin

    sekcja lokalna;

    chce1 := true;
    while (chce2) do
    begin
      if (kto_czeka = 1)
      begin
        chce1 := false;
        while(kto_czeka = 1)
        do
        begin
          //nie rób nic
        end;
        chce1 := true;
      end;
    end;

    sekcja krytyczna;

    kto_czeka := 1;
    chce1 := false;

  end;
end;
```

```
thread w2;
begin
  while true do
  begin

    sekcja lokalna;

    chce2 := true;
    while (chce1) do
    begin
      if (kto_czeka = 2)
      begin
        chce2 := false;
        while(kto_czeka = 2)
        do
        begin
          //nie rób nic
        end;
        chce2 := true;
      end;
    end;

    sekcja krytyczna;

    kto_czeka := 2;
    chce2 := false;

  end;
end;
```

1.4.4. Algorytm Dorana-Thomasa

Zaimplementuj algorytm Dorana-Thomasa rozwiązujący problem wzajemnego wykluczania dla dwóch wątków. Posłuż się poniższym pseudokodem [3].

```
var
  chce1: boolean := false;
  chce2: boolean := false;
  kto_czeka: 1..2 := 1;
```

```
thread w1;
begin
  while true do
  begin

    sekcja lokalna;

    chce1 := true;

    if (chce2) do
    begin
      if (kto_czeka = 1)
      begin
        chce1 := false;
        while(kto_czeka = 1)
        do
        begin
          //nie rób nic
        end;
        chce1 := true;
      end;
      while(chce2) do
      begin
        //nie rób nic
      end;
    end;

    sekcja krytyczna;

    chce1 := false;
    kto_czeka := 1;

  end;
end;
```

```
thread w2;
begin;
  while true do
  begin

    sekcja lokalna;

    chce2 := true;

    if (chce1) do
    begin
      if (kto_czeka = 2)
      begin
        chce2 := false;
        while(kto_czeka = 2)
        do
        begin
          //nie rób nic
        end;
        chce2 := true;
      end;
      while(chce1) do
      begin
        //nie rób nic
      end;
    end;

    sekcja krytyczna;

    chce2 := false;
    kto_czeka := 2;

  end;
end;
```

1.4.5. Algorytm Petersona

Zaimplementuj algorytm Petersona rozwiązujący problem wzajemnego wykluczania dla dwóch wątków. Posłuż się poniższym pseudokodem [3, 5].

```
var
  chce1: boolean := false;
  chce2: boolean := false;
  kto_czeka: 1..2 := 1;
```

```
thread w1;
begin
  while true do
  begin

    sekcja lokalna;

    chce1 := true;
    kto_czeka := 1;
    while
      chce2 and (kto_czeka = 1)
    do
    begin
      // nic nie rób
    end;

    sekcja_krytyczna;

    chce1 := false;

  end
end;
```

```
thread w2;
begin
  while true do
  begin

    sekcja lokalna;

    chce2 := true;
    kto_czeka := 2;
    while
      chce1 and (kto_czeka = 2)
    do
    begin
      // nic nie rób
    end;

    sekcja_krytyczna;

    chce2 := false;

  end
end;
```

1.4.6. Algorytm Petersona dla N wątków

Zaimplementuj algorytm Petersona rozwiązujący problem wzajemnego wykluczania dla N wątków. Posłuż się poniższym pseudokodem [3, 5].

```
var
  chce: array[1..N] of integer;
  kto_czeka: array[1..N-1] of integer;

thread w (i: 1..N);
begin
  while true do
  begin

    sekcja lokalna;

    for bariera := 1 to N - 1 do
    begin
      chce[i] := bariera;
      kto_czeka[bariera] := i;
      while ∃_{j<>i} chce[j] >= bariera
                    and (kto_czeka[bariera] = i)
```

```
        begin
          // nie rób nic
        end;
      end

      sekcja krytyczna;

      chce[i] := 0;

    end
  end
```

1.4.7. Szeregowanie instrukcji

Program współbieżny składa się z trzech wątków A, B i C oraz wątku głównego. Wątki oprócz intrukcji niewymagających synchronizacji, wykonują również pewne instrukcje, które muszą wykonać się w odpowiedniej kolejności.

Instrukcja B z wątku B może się rozpocząć dopiero po zakończeniu wątku A, a *instrukcja C* z wątku C może się rozpocząć dopiero po zakończeniu wątku B. Wątek główny wykonuje „instrukcję końcową”, która może zostać wykonana nie wcześniej niż po zakończeniu wątku A, B i C. Napisz definicje wszystkich wątków, do synchronizacji wykorzystując jedynie funkcję **join()**.

1.4.8. Tablica wątków

Zdefiniuj klasę wątku jako rozszerzenie klasy `Thread`. Utwórz 30 wątków tego typu, zamieszczając referencje do nich w tablicy. Następnie uruchom wszystkie wątki. Niech wątek główny zaczeka na ich zakończenie, po czym wypisze na ekranie komunikat informujący o tym.

Rozdział 2

Semafory

2.1. Definicja i własności **28**
2.2. Problem *producent* - *konsument* **31**
2.3. Problem ucztujących filozofów **33**
2.4. Zadania **38**
2.4.1. Problem ucztujących filozofów z wykorzystaniem semafora *jadalnia* 38
2.4.2. Problem ucztujących filozofów z niesymetrycznym sięganiem po pałeczki 39
2.4.3. Rzut monety w rozwiązaniu problemu ucztujących filozofów 39
2.4.4. Producent-konsument z buforem jednoelementowym 39
2.4.5. Wielu producentów i konsumentów z buforem cyklicznym 40
2.4.6. Producent-konsument z dwuetapowym buforem 41
2.4.7. Problem czytelników i pisarzy z priorytetem dla czytelników 41
2.4.8. Problem czytelników i pisarzy z priorytetem dla pisarzy 41
2.4.9. Problem śpiącego golibrody 42
2.4.10. Szeregowanie instrukcji 42

Rozważania z poprzedniego rozdziału pokazują dobitnie, że rozwiązywanie problemów związanych z programowaniem współbieżnym wymaga zdefiniowania łatwych w użyciu mechanizmów. W tym rozdziale zajmiemy się semaforami.

2.1. Definicja i własności

Rozważmy następującą definicję semaforów, która została sformułowana przez Dijkstrę w odniesieniu do procesów współbieżnych.

Definicja 2.1. *Semaforem S nazywamy zmienną przyjmującą wartości całkowite nieujemne. Jedynymi dopuszczalnymi dla semaforów operacjami są:*
— ***S.init(n)****: dopuszczalne jedynie przed pierwszym użyciem, jednokrotne nadanie semaforowi wartości początkowej* ***n****,*
— ***S.wait()****: jeśli $S > 0$ wówczas $S := S - 1$, w przeciwnym razie wstrzymaj wykonanie procesu, który wywołał tę operację,*
— ***S.signal()****: w razie gdy są jakieś procesy wstrzymane w wyniku wykonania operacji* ***S.wait()*** *na tym semaforze, wówczas wznów wykonanie jednego z nich, w przeciwnym razie $S := S + 1$.*
Operacje ***wait()*** *i* ***signal()*** *są operacjami atomowymi, czyli ich wykonania na danym semaforze nie mogą być ze sobą przeplatane.*

Warto zaznaczyć, że operacja **signal()** nie precyzuje, który z wątków ma być wznowiony. Najczęściej procesy oczekujące na wznowienie są kolejkowane. Podkreślmy również, że poza operacjami **wait()** i **signal()** nie są dozwolone żadne inne operacje. W szczególności nie ma możliwości testowania wartości semafora. Odnotujmy również kolejną ważną własność semaforów [1, 2].

Własność 2.1. *Niech S_0 oznacza początkową wartość semafora S. Niech dalej $\#\text{signal}$ oraz $\#\text{wait}$ oznaczają odpowiednio liczbę zakończonych operacji* ***signal()*** *i* ***wait()*** *na semaforze S. Wówczas $S \geq 0$ oraz*

$$S = S_0 + \#\text{signal} - \#\text{wait}. \tag{2.1}$$

Dla synchronizacji współbieżnych wątków w języku Java dysponujemy klasą `Semaphore` dostępną w pakiecie `java.util.concurrent`. Jej funkcjonalności rozszerzają standardową definicję semafora o dodatkowe operacje. Na listingu 2.1 podajemy konstruktory i metody odpowiadające definicji 2.1.

Listing 2.1. Podstawowe funkcjonalności klasy Semaphore

```
1 Semaphore(int permits)
2 // tworzy obiekt "semafor" o zadanej wartości początkowej
3 Semaphore(int permits, boolean fair)
4 // tworzy obiekt "semafor" o zadanej wartości początkowej
5 // z gwarancją tego, żę wątki wstrzymywane są kolejkowane
6 //
7 void     acquire() throws InterruptedException
8 // operacja wait()
9 void     acquireUninterruptibly()
10 // operacja wait() bez wyrzucania wyjątku
11 void     release()
12 // operacja signal()
```

Jako przykład prostego zastosowania semaforów rozważmy rozwiązanie problemu wzajemnego wykluczania (listing 2.2). Protokoły *wstępny* i *końcowy* sprowadzają się do wykonania operacji **wait()** i **signal()** na semaforze zainicjowanym wartością 1.

Listing 2.2. Problem wzajemnego wykluczania (semafor)

```
1 public class Main implements Runnable {
2
3   static Semaphore s = new Semaphore(1,true);
4   private int mojNum;
5
6   public void run() {
7
8     while (true) {
9
10       // sekcja lokalna
11       try {
12         Thread.sleep((long) (2500 * Math.random()));
13       } catch (InterruptedException e) {
14       }
15
16      // protokół wstępny
17      s.acquireUninterruptibly();
18      // sekcja krytyczna
19      System.out.println("Wątek " + ThreadID.get()+ " w SK");
20      try {
21        Thread.sleep((long) (2100 * Math.random()));
22      } catch (InterruptedException e) {
23      }
24      System.out.println("Wątek " + ThreadID.get()+ " w SK");
25      // protokół końcowy
26      s.release();
27    }
28   }
29
```

```
30     public Main(String num) {
31         super(num);
32         mojNum=ThreadID.get();
33     }
34
35     public static void main(String[] args) {
36
37        new Thread(new Main("0")).start();
38        new Thread(new Main("1")).start();
39     }
40 }
```

Odnotujmy teraz za książkami [1, 2] najważniejsze własności podanego rozwiązania.

Twierdzenie 2.1. *Rozwiązanie posiada własność wzajemnego wykluczania.*

Dowód. Niech $\#SK$ oznacza liczbę wątków w sekcji krytycznej. Z analizy algorytmu wynika, że

$$\#SK = \#wait - \#signal.$$

Wykorzystując własność (2.1) otrzymujemy

$$S = 1 + \#signal - \#wait,$$

skąd $S = 1 - \#SK$. Zatem ostatecznie

$$\#SK + S = 1. \tag{2.2}$$

Z uwagi na to, że $S \geq 0$ musi zachodzić $\#SK \leq 1$, czyli ostatecznie tylko jeden wątek może znajdować się w sekcji krytycznej. ■

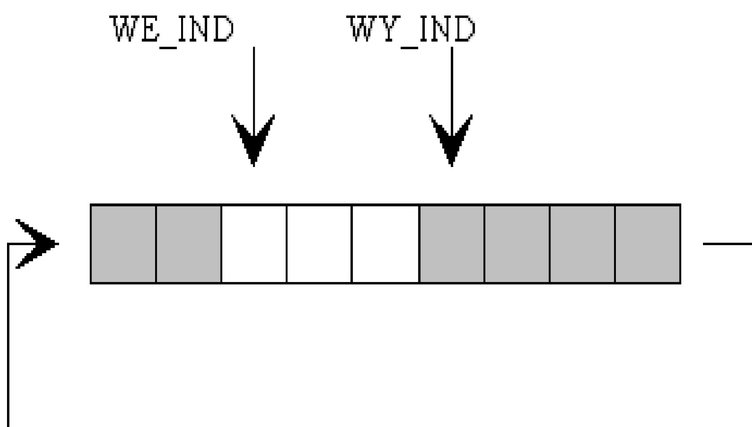
Twierdzenie 2.2. *W rozwiązaniu nie wystąpi zakleszczenie.*

Dowód. Aby wystąpiło zakleszczenie, oba wątki musiałyby zatrzymać się wykonując operację **wait()**. Zatem musiałoby być $\#SK = 0$ oraz $S = 0$, czyli na mocy (2.2). ■

Twierdzenie 2.3. *W rozwiązaniu nie wystąpi zagłodzenie.*

Dowód. W przypadku zagłodzenia wątek musiałby zatrzymać się wykonując operację **wait()**, zaś drugi wykonywałby w nieskończoność swój cykl. Zatem wykonanie operacji **signal()** wznowiłoby wstrzymywany wątek. ■

Twierdzenie 2.4. *Rozwiązanie nie zawodzi w przypadku braku współzawodnictwa.*



Rysunek 2.1. Bufor cykliczny

Dowód. Istotnie, w przypadku braku współzawodnictwa, na mocy własności (2.2) musi być $S = 1$, zatem wykonanie **signal()** nie wstrzymuje wątku, który chce wejść do sekcji krytycznej. ■

Na koniec podkreślmy, że przedstawione wyżej rozwiązanie jest poprawne również dla więcej niż dwóch wątków.

2.2. Problem *producent - konsument*

Rozważmy teraz następujący problem programowania współbieżnego zwany *problemem producenta-konsumenta*. Zakładamy tutaj, że program współbieżny składa się z dwóch wątków (procesów), gdzie oba wątki działają w pętlach nieskończonych, przy czym wątek *producent* ma za zadanie produkować dane, zaś *konsument* przetwarzać je w kolejności ich produkowania. W rozwiązaniu (listing 2.3) wykorzystamy *bufor cykliczny* (rysunek 2.1).

Listing 2.3. Problem producent-konsument

```
1 import java.util.concurrent.Semaphore;
2 import java.util.concurrent.atomic.AtomicIntegerArray;
3
4 public class Main {
5
6   static final int MAX=10;
7   static AtomicIntegerArray buf =
8                    new AtomicIntegerArray(MAX);
9   static Semaphore elementy = new Semaphore(0);
10  static Semaphore miejsca = new Semaphore(MAX);
11
```

```
12    static int weInd=0;
13    static int wyInd=0;
14
15    static Thread p = new Producent();
16    static Thread k = new Konsument();
17
18    static class Producent extends Thread {
19
20      public void run() {
21
22        while (true) {
23          try {
24            Thread.sleep((long) (2000 * Math.random()));
25          } catch (InterruptedException e) {
26          }
27
28          int x=(int)(100 * Math.random());
29          miejsca.acquireUninterruptibly();   // miej.wait();
30          buf.set(weInd,x);
31          weInd=(weInd+1)%MAX;
32          System.out.println("Wyprodukowałem␣"+x);
33          elementy.release();   // elem.signal()
34        }
35      }
36    }
37
38    static class Konsument extends Thread {
39
40      public void run() {
41
42        while (true) {
43          int x;
44          elementy.acquireUninterruptibly();
45          x=buf.get(wyInd);
46          wyInd=(wyInd+1)%MAX;
47          miejsca.release();
48          try {
49            Thread.sleep((long) (2000 * Math.random()));
50          } catch (InterruptedException e) {
51          }
52          System.out.println("Skonsumowałem␣"+x);
53        }
54      }
55    }
56
57    public static void main(String[] args) {
58        p.start();
59        k.start();
60    }
61  }
```

Poprawność rozwiązania problemu *producenta-konsumenta* oznacza, że
— w rozwiązaniu producent nie będzie wstawiał danych do pełnego bufora, zaś konsument pobierał z bufora pustego (bezpieczeństwo),
— w rozwiązaniu nie wystąpi zakleszczenie ani zagłodzenie (żywotność).

Istotnie, łatwo zauważyć, że przedstawione rozwiązanie spełnia te wymagania. Analiza kodu źródłowego pokazuje, że zachodzi następująca własność rozwiązania

$$elementy + miejsca = MAX. \tag{2.3}$$

Niech $\#E$ oznacza liczbę elementów w buforze. Wówczas zachodzą następujące własności [1]:

$$\#E = elementy \tag{2.4}$$

własności

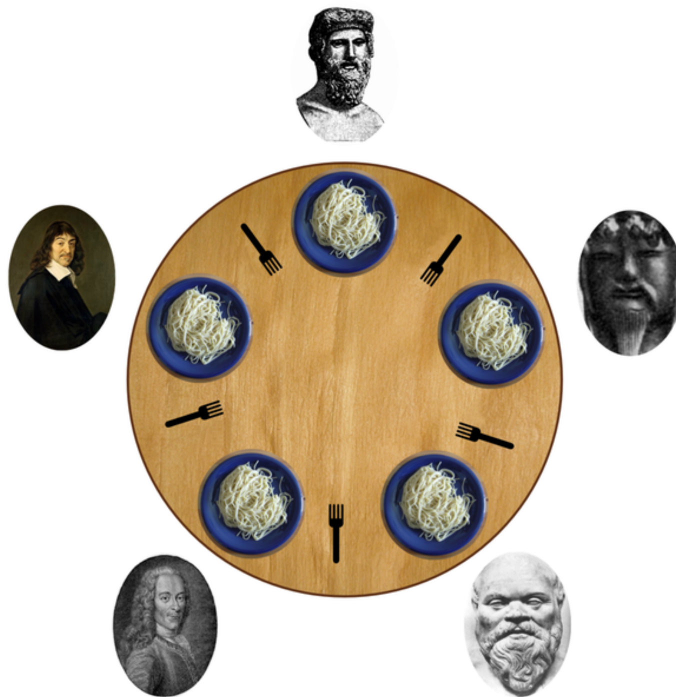
$$\#E = MAX - miejsca. \tag{2.5}$$

Własność (2.4) oznacza inaczej, że semafor *elementy* zlicza liczbę elementów w buforze (zwiększany o jeden przy wstawianiu, zmniejszany o jeden przy pobieraniu elementu). Własność (2.5) jest prostą konsekwencją (2.4) i (2.3). Producent nie będzie wstawiał do pełnego bufora, gdyż będzie wstrzymywany na semaforze *miejsca*, zaś konsument nie będzie pobierał z pustego bufora, gdyż będzie wstrzymywany na semaforze *elementy*. W rozwiązaniu nie wystąpi zakleszczenie, gdyż oba semafory musiałyby mieć jednocześnie wartość zero, co przeczy równości (2.3). Nie wystąpi również zagłodzenie, gdyż zarówno producent jak i konsument wykonują po wstawieniu oraz pobraniu elementu operację **signal()**.

2.3. Problem ucztujących filozofów

Przypuśćmy, że przy stole ucztuje pięciu filozofów $P_0, P_1, \ldots, P_4$ (rysunek 2.2), którzy działają w pętlach nieskończonych wykonując na przemian sekcję lokalną *myślenie* oraz sekcję krytyczną *jedzenie*, do czego potrzebne są dwa widelce. Na stole umieszczono pięć widelców $f_0, f_1, \ldots, f_4$, z których każdy leży po lewej stronie filozofa. Filozof w chwili gdy ma rozpocząć jedzenie podnosi najpierw jeden widelec (po swojej lewej albo prawej stronie), a następnie drugi widelec. Problem polega takim zaprojektowaniu korzystania przez filozofów z widelców, aby spełnione były następujące własności:

1. filozof je wtedy i tylko wtedy, gdy ma dwa widelce,
2. dwóch filozofów nie może w tym samym czasie korzystać z tego samego widelca,
3. nie wystąpi zakleszczenie,
4. żaden filozof nie będzie zagłodzony,



Rysunek 2.2. Problem ucztujących filozofów (źródło: Benjamin D. Esham / Wikimedia Commons)

5. rozwiązanie działa w przypadku braku współzawodnictwa.

Na listingu 2.4 przedstawiono próbę rozwiązania problemu. Każdemu filozofowi odpowiada jeden wątek, zaś rolę widelców pełnią semafory.

Własność 2.2. *Żaden widelec nie jest nigdy trzymany jednocześnie przez dwóch filozofów.*

Dowód. Jedzenie stanowi sekcję krytyczną wątku. Niech $\#P_i$ będzie liczbą filozofów trzymających widelec i, wówczas (patrz dowód dotyczący własności wzajemnego wykluczania przy pomocy semaforów) mamy

$$widelec_i + \#P_i = 1.$$

Oczywiście $widelec_i \geq 0$, zatem $\#P_i \leq 1$. ■

Listing 2.4. Problem ucztujących filozofów (zakleszczenie)

```
1 import java.util.concurrent.Semaphore;
2
3 public class Filozof extends Thread {
4
5   static final int MAX=5;
6   static Semaphore [] widelec = new Semaphore[MAX];
7
8   int mojNum;
9
10  public Filozof(int nr){
11    mojNum=nr;
12  }
13
14
15  public void run() {
16
17    while (true) {
18
19        try {
20          Thread.sleep((long) (7 * Math.random()));
21        } catch (InterruptedException e) {
22        }
23
24
25        widelec[mojNum].acquireUninterruptibly();
26        widelec[(mojNum+1)%MAX].acquireUninterruptibly();
27
28        System.out.println("Zaczyna␣jeść␣"+mojNum);
29        try {
30          Thread.sleep((long) (5 * Math.random()));
31        } catch (InterruptedException e) {
32        }
33        System.out.println("Kończy␣jeść␣"+mojNum);
```

```
34
35          widelec[mojNum].release();
36          widelec[(mojNum+1)%MAX].release();
37
38       }
39    }
40
41
42    public static void main(String[] args) {
43
44       for (int i=0;i<MAX;i++) {
45          widelec[i]=new Semaphore(1);
46       }
47
48       for (int i=0;i<MAX;i++) {
49          (new Main(i)).start();
50       }
51
52    }
53 }
```

W rozwiązaniu może jednak wystąpić zakleszczenie. Istotnie, w sytuacji, gdy każdy z filozofów chwyci jednocześnie swój widelec (po lewej stronie), żaden z filozofów nie będzie mógł rozpocząć jedzenia. Rozwiązaniem tego problemu będzie ograniczenie do czterech liczby filozofów trzymających jednocześnie widelce. Otrzymamy w ten sposób rozwiązanie przedstawione na listingu 2.5.

Listing 2.5. Problem ucztujących filozofów

```
1  import java.util.concurrent.Semaphore;
2
3  public class Filozof extends Thread {
4
5    static final int MAX=5;
6    static Semaphore [] widelec = new Semaphore[MAX];
7    static Semaphore jadalnia = new Semaphore(MAX-1);
8
9    int mojNum;
10
11   public Filozof(int nr){
12     mojNum=nr;
13   }
14
15
16   public void run() {
17
18     while (true) {
19
20       try {
```

```
21              Thread.sleep((long) (7 * Math.random()));
22          } catch (InterruptedException e) {
23          }
24
25          jadalnia.acquireUninterruptibly();
26          widelec[mojNum].acquireUninterruptibly();
27          widelec[(mojNum+1)%MAX].acquireUninterruptibly();
28
29          System.out.println("Zaczyna jeść "+mojNum);
30          try {
31            Thread.sleep((long) (5 * Math.random()));
32          } catch (InterruptedException e) {
33          }
34          System.out.println("Kończy jeść "+mojNum);
35
36          widelec[mojNum].release();
37          widelec[(mojNum+1)%MAX].release();
38
39          jadalnia.release();
40      }
41    }
42
43
44    public static void main(String[] args) {
45
46      for (int i=0;i<MAX;i++) {
47          widelec[i]=new Semaphore(1);
48      }
49
50      for (int i=0;i<MAX;i++) {
51          (new Main(i)).start();
52      }
53
54    }
55  }
```

Odnotujmy własności tego rozwiązania.

Własność 2.3. *W rozwiązaniu nie wystąpi zakleszczenie.*

Dowód. Oczywiście najwyżej cztery lewe widelce będą blokowane przez filozofów, zatem jeden z nich uzyska dostęp do dwóch widelców i będzie mógł rozpocząć jedzenie. ■

Własność 2.4. *W rozwiązaniu nie wystąpi zagłodzenie.*

Dowód. Przypuśćmy, że filozof i został zagłodzony. Możliwe są następujące trzy przypadki.

1. *Proces jest wstrzymywany na semaforze jadalnia*, który ma przez cały czas wartość 0. Może to nastąpić tylko w przypadku, gdy pozostali filozofowie czekają w nieskończoność na widelce, gdyż gdyby jeden z nich jadł, wówczas po zakończeniu wywołałby operację **signal()** na semaforze *jadalnia*, co spowodowałoby wznowienie procesu.
2. *Filozof jest zablokowany w oczekiwaniu na swój lewy widelec.* Oznacza to, że jego sąsiad z lewej strony wykonał wcześniej operację **signal()** na swoim prawym widelcu, co doprowadzi do wznowienia pracy filozofa.
3. *Filozof jest zablokowany w oczekiwaniu na prawy widelec.* To oznacza, że filozof z prawej strony wziął swój lewy widelec i może być blokowany w oczekiwaniu na swój prawy widelec. Skoro jednak jest blokowany przez swój prawy widelec, to kontynuując to rozumowanie dla następnych widelców, z własności semafora *jadalnia* wynika, że jeden z wątków musi być poza sekcją krytyczną i nie może być blokowany w oczekiwaniu na widelec. Zatem dojdzie do kolejnych wznowień wątków, co w końcu doprowadzi do wznowienia działania rozważanego filozofa.

2.4. Zadania

2.4.1. Problem ucztujących filozofów z wykorzystaniem semafora *jadalnia*

Rozwiąż za pomocą semaforów problem ucztujących filozofów „w wersji chińskiej".

Przy wspólnym stole zasiadło pięciu chińskich filozofów. Filozofowie z zamiłowaniem oddają się czynności myślenia, jednakże muszą się co jakiś czas posilać. Na stole, przy którym siedzą, znajduje się pięć talerzy (po jednym dla każdego), jeden półmisek z ryżem (wspólny dla wszystkich, do którego dostęp jest nieograniczony) oraz pięć pałeczek, niezbędnych do jedzenia ryżu, które zostały ułożone na przemian z talerzami, tak że każdy filozof może mieć dostęp do dwóch pałeczek, jednej po prawej oraz jednej po lewej stronie talerza, przy czym każdą pałeczkę współdzieli z sąsiadem. Zdefiniuj wątek Filozofa, zapewniając następujące warunki:

a) Każdy filozof może myśleć do woli, niezależnie od pozostałych.
b) Filozof może zacząć jeść tylko jeśli udało mu się podnieść dwie pałeczki.
c) Pałeczka nie może być podniesiona przez dwóch filozofów w tym samym czasie.
d) Pałeczki podnoszone są pojedynczo, najpierw pałeczka z lewej strony.
e) Program powinien działać zarówno w przypadku ostrej rywalizacji o jedzenie, jak i w przypadku braku rywalizacji.

f) Nie może dochodzić do zakleszczeń (zarówno w wersji deadlock jak i livelock) ani do zagłodzenia (w tym przypadku rozumianego dosłownie).

2.4.2. Problem ucztujących filozofów z niesymetrycznym sięganiem po pałeczki

Problem ucztujących filozofów można rozwiązać również zamieniając kolejność sięgania po pałeczki jednego z filozofów. Czterech spośród pięciu filozofów, najpierw sięga po pałeczkę z lewej strony, a potem tę z prawej, natomiast jeden filozof czynność tę wykonuje odwrotnie. Zaimplementuj takie rozwiązanie tego problemu.

2.4.3. Rzut monety w rozwiązaniu problemu ucztujących filozofów

Dowodzi się również, że problem ucztujących filozofów jest rozwiązany poprawnie, jeśli każdy filozof, o tym, którą pałeczkę podniesie jako pierwszą, zdecyduje rzutem monety, podniesie wylosowaną pałeczkę (jeśli nie jest wolna to zaczeka na nią), a następnie sprawdzi czy druga pałeczka jest wolna. Jeśli tak, to może jeść. Jeśli natomiast druga pałeczka nie jest wolna to odkłada pałeczkę, którą już trzyma, i podejmuje kolejną próbę jedzenia, ponownie rzucając monetą [3,13]. Zaimplementuj również takie rozwiązanie.

2.4.4. Producent-konsument z buforem jednoelementowym

Zaimplementuj rozwiązanie problemu producenta-konsumenta przy następujących założeniach:

a) Istnieje co najmniej jeden wątek producenta oraz co najmniej jeden wątek konsumenta.
b) Producent „produkuje” daną wstawiając ją do bufora.
c) Konsument „konsumuje” daną pobierając ją z bufora.
d) Bufor jest jednoelementowy.
e) Na początku bufor jest pusty.
f) Producent może wstawić wyprodukowany element do bufora tylko wtedy, gdy jest on pusty.
g) Konsument może skonsumować element z bufora tylko jeśli bufor nie jest pusty.
h) Jeśli jakiś producent zechce wyprodukować element i wstawić do bufora, to prędzej czy później dostanie taką możliwość.
i) Jeśli jakiś konsument zechce skonsumować element z bufora, to prędzej czy później dostanie taką możliwość.
j) Producenci i konsumenci wykonują się w pętli nieskończonej, a swoje działania podejmują w losowych odstępach czasu.

k) Zarówno producent jak i konsument, po wykonanej operacji wypisuje na ekranie komunikat, np. „Wątek 2 wyprodukował 101” lub „Wątek 5 skonsumował 101”.

2.4.5. Wielu producentów i konsumentów z buforem cyklicznym

Na stronie 31 znajduje się rozwiązanie problemu producenta-konsumenta przy następujących założeniach:

a) Istnieje jeden wątek producenta oraz jeden wątek konsumenta.
b) Istnieje współdzielony bufor wieloelementowy o określonym z góry rozmiarze.
c) Na początku bufor jest pusty.
d) Producent „produkuje” daną wstawiając ją do bufora. Produkowane dane mogą być na przykład losowe.
e) Konsument „konsumuje” daną pobierając ją z bufora.
f) Elementy są produkowane i wstawiane do bufora na kolejne wolne miejsca, od początku bufora.
g) Elementy konsumowane są z bufora kolejno zgodnie z zasadą: wcześniej wyprodukowany, wcześniej skonsumowany.
h) Bufor jest cykliczny, to znaczy, jeśli producent doszedł do końca bufora to próbuje wstawiać od początku, pod warunkiem, że są tam miejsca (zwolniły się, bo elementy zostały skonsumowane). Podobnie konsument, jeśli skonsumował element z ostatniego miejsca, wówczas próbuje konsumować od początku bufora, pod warunkiem, że są tam jakieś elementy wyprodukowane.
i) Do bufora nie może być wstawionych więcej elementów niż jest miejsc, zatem jeśli producent wyprodukował daną, a nie ma jej gdzie wstawić to czeka aż zwolni się miejsce, to znaczy, aż jakaś dana zostanie skonsumowana przez konsumenta.
j) Jeżeli konsument chce skonsumować daną, a bufor jest pusty, to musi zaczekać aż producent wstawi jakiś element do bufora.
k) Dopuszczone jest jednoczesne wstawianie danej do bufora oraz jej pobieranie.
l) Producent i konsument wykonują się w pętli nieskończonej, a swoje działania podejmują w losowych odstępach czasu.
m) Zarówno producent jak i konsument, po wykonanej operacji wypisuje na ekranie komunikat, np. „Wątek 2 wyprodukował 101” lub „Wątek 5 skonsumował 101”.

Zmodyfikuj to rozwiązanie dopuszczając współbieżne działanie wielu wątków producenta i wielu wątków konsumenta, ale tak, aby w tym samym

czasie do bufora mógło odwoływać się maksymalnie dwa wątki: jeden producenta i jeden konsumenta.

2.4.6. Producent-konsument z dwuetapowym buforem

Zaimplementuj w języku Java, i z użyciem mechanizmu semaforów, rozwiązanie problemu producenta-konsumenta wyspecyfikowanego w następujący sposób:

a) Istnieje n wątków P1, ..., Pn, jeden wątek S oraz jeden wątek K.
b) Istnieją dwa współdzielone bufory b1 i b2. Obydwa bufory są cykliczne, wieloelementowe, o określonym z góry rozmiarze.
c) Wątki P1, ..., Pn, niezależnie od siebie, cyklicznie, produkują dane, wstawiając je do bufora b1.
d) Proces S pobiera po dwie dane z bufora b1, przekształca ją na jedną daną i wstawia do bufora b2.
e) Dane z bufora b1 pobierane są zgodnie z zasadą: wcześniej wyprodukowane, wcześniej pobrane.
f) Proces K czeka na całkowite wypełnienie bufora b2, po czym konsumuje cały pełny bufor na raz.
g) Do żadnego bufora nie może być wstawionych więcej danych niż jest miejsc, zatem jeśli istnieje wątek, który chce wstawić daną do bufora, a nie ma miejsca, to czeka aż miejsce się zwolni.
h) Jeżeli wątek chce pobrać daną, a bufor jest pusty, to również czeka.
i) Na początku obydwa bufory są puste.
j) Wszystkie wątki działają w pętli nieskończonej (cyklicznie), a swoje działania podejmują w losowych odstępach czasu.
k) Wątki wypisują (lub w inny sposób przedstawiają) informacje o tym co wykonały.

2.4.7. Problem czytelników i pisarzy z priorytetem dla czytelników

Zaimplementuj problem czytelników i pisarzy z priorytetem dla czytelników (czytelnik może rozpocząć czytanie jeśli pisarz aktualnie nie pisze) z wykorzystaniem mechanizmu semaforów. Wykorzystaj klasę `Semaphore` z pakietu `java.util.concurrent`.

2.4.8. Problem czytelników i pisarzy z priorytetem dla pisarzy

Zaimplementuj problem czytelników i pisarzy z priorytetem dla pisarzy (czytelnik nie rozpoczyna czytania jeśli pisarz oczekuje na swoją kolej) z wykorzystaniem mechanizmu semaforów. Wykorzystaj klasę `Semaphore` z pakietu `java.util.concurrent`.

2.4.9. Problem śpiącego golibrody

Napisz program rozwiązujący problem śpiącego golibrody (ang. *Sleeping Barber Problem*).

W zakładzie fryzjerskim znajduje się tylko jeden fotel oraz poczekalnia z określoną liczbą miejsc. Pracuje w nim jeden fryzjer. Do zakładu przychodzą klienci, którzy chcą się ostrzyc. Zaprogramuj funkcjonowanie zakładu fryzjerskiego zachowując następujące własności:

a) Kiedy fryzjer kończy strzyc klienta, klient wychodzi a fryzjer zagląda do poczekalni czy są kolejni klienci. Jeśli jakiś klient czeka w poczekalni wówczas zaprasza go na fotel i zaczyna strzyżenie. Jeśli poczekalnia jest pusta, wówczas fryzjer ucina sobie drzemkę na fotelu.
b) Klient, który przychodzi do zakładu, sprawdza co robi fryzjer. Jeśli fryzjer strzyże, wówczas klient idzie do poczekalni, i jeśli jest wolne miejsce to je zajmuje i czeka. Jeśli nie ma wolnego miejsca, wówczas klient rezygnuje. Jeśli fryzjer śpi, to budzi go.

2.4.10. Szeregowanie instrukcji

Program współbieżny składa się z trzech wątków A, B i C wykonujących w pętlach nieskończonych odpowiednio instrukcje: *Instrukcja A*, *Instrukcja B* oraz *Instrukcja C*. Wykonanie instrukcji *Instrukcja C* przez wątek C ma być zawsze poprzedzone zakończeniem wykonywania *Instrukcji A* oraz *Instrukcji B* przez wątki A i B, zaś każde wykonanie *Instrukcji A* oraz *Instrukcji B* (z wyjątkiem pierwszych wykonań) ma być poprzedzone zakończeniem wykonywania *Instrukcji C* przez wątek C. Wykorzystując semafory podaj definicje klas odpowiadających wątkom A, B, C. Dodaj program, w którym uruchomisz te wątki.

Rozdział 3

Monitory

3.1. Podstawowe pojęcia **44**
3.2. Problem czytelników i pisarzy **47**
3.3. Zadania **51**
3.3.1. Mechanizm bariery za pomocą monitorów . . . 51
3.3.2. Semafor za pomocą monitorów 51
3.3.3. Semafor binarny za pomocą monitorów 51
3.3.4. Szeregowanie instrukcji z wykorzystaniem monitorów 51
3.3.5. Problem ucztujących filozofów z wykorzystaniem monitorów 51

Rozważymy teraz wykorzystanie w programach współbieżnych monitorów jako wygodnego narzędzia do efektywnej implementacji mechanizmów zapewniających synchronizację i komunikację między wątkami.

3.1. Podstawowe pojęcia

W rozdziale pierwszym omówiliśmy na przykładzie klasy `Licznik` (listing 1.13) wykorzystanie metod synchronizowanych dla realizacji synchronizacji wątków wymagających wyłącznego dostępu do obiektu. Obiekt wyposażony w takie metody będziemy nazywać *monitorem*. W języku Java każdy obiekt może funkcjonować jako monitor. W praktyce często wykonanie pewnych metod synchronizowanych na monitorze będzie możliwe pod warunkiem zaistnienia pewnej sytuacji. Na przykład w problemie producenta i konsumenta, producent będzie mógł wstawić do bufora pod warunkiem, że bufor nie jest pełny, a konsument będzie mógł pobrać element z bufora tylko wtedy, gdy nie jest pusty. Istnieje zatem konieczność wstrzymywania wykonania metod synchronizowanych, gdy nie są spełnione warunki umożliwiające kontynuację ich wykonania. Takiego mechanizmu dostarczają metody **wait()**, **notify()** i **notifyAll()** klasy `Object`, które mogą być wywoływane tylko z wnętrza metod lub instrukcji synchronizowanych, a zatem wątek, który je wywołuje musi przejąć na własność monitor.

Wykonanie metody **wait()** powoduje, że wątek zostaje wstrzymany i zwraca wyłączny dostęp do monitora, aż do chwili gdy:
— jakiś inny wątek wywoła metodę **notify()** i dany wątek zostanie wybrany jako wątek, który ma być wznowiony,
— jakiś inny wątek wywoła metodę **notifyAll()**,
— wątek otrzyma sygnał przerwania (zostanie na nim wywołana metoda **interrupt()**,
— upłynie czas określony w parametrze wywołania **wait()**, co oczywiście nie jest możliwe dla bezparametrowej metody (listing 3.1).

Zauważmy zatem, że wywołanie metod **notify()** i **notifyAll()** nie powoduje automatycznej kontynuacji wykonywania metody synchronizowanej, gdyż wątek, który wznowił działanie, będzie musiał współzawodniczyć z innymi wątkami o dostęp do monitora.

Listing 3.1. Podstawowe funkcjonalności dotyczące monitorów

```
1 // oczekiwanie na notify() lub notifyAll()
2 public final void wait()
3                    throws InterruptedException
4 // oczekiwanie na notify() lub notifyAll()
5 // w ciągu zadanego czasu
6 public final void wait(long timeout)
```

```
7                       throws InterruptedException
8 // wznowienie jednego wątku, który wywołał wait()
9 public final void notify()
10 // wznowienie wszyskich wątków, które wywołały wait()
11 public final void notifyAll()
```

Jako przykład zastosowania monitorów rozważmy prostą implementację klasy `Bufor` (listingi 3.2 i 3.3), która służy do rozwiązania problemu *producenta - konsumenta*. Wątki *Producent* i *Konsument* wykorzystują metody synchronizowane klasy `Bufor` do wstawiania i pobierania elementów. Zauważmy, że jednocześnie tylko jeden wątek może korzystać z bufora. W metodzie **wstaw()** wątek producenta oczekuje na niepełny bufor i jeśli w trakcie sprawdzania stwierdzi jego pełność – wstrzymuje swoje działanie. Podobnie wątek konsumenta oczekuje w pętli na elementy w buforze.

Listing 3.2. Klasa Bufor

```
1 public interface IBufor {
2   public void wstaw(int x);
3   public int pobierz();
4 }
5
6 public class Bufor implements IBufor {
7
8     private final int MAX;
9     private int [] buf;
10
11    private int weInd=0;
12    private int wyInd=0;
13    private int licz=0;
14
15    public Bufor(int m){
16      MAX=m;
17      buf=new int[MAX];
18    }
19
20    public synchronized void wstaw(int x) {
21
22        while(licz==MAX){
23          try {
24            wait();
25          }catch(InterruptedException e){
26          }
27
28        }
29        licz++;
30        buf[weInd]=x;
31        weInd=(weInd+1)%MAX;
32        notifyAll();
```

```
33
34      }
35
36      public synchronized int pobierz() {
37
38          while(licz==0){
39            try {
40              wait();
41            }catch(InterruptedException e){
42            }
43          }
44
45          licz--;
46          int x=buf[wyInd];
47          wyInd=(wyInd+1)%MAX;
48          notifyAll();
49          return x;
50      }
51  }
```

Listing 3.3. Klasy Producent i Konsument

```
1   public class Producent extends Thread {
2
3       private IBufor buf;
4
5       public Producent(IBufor buf){
6         this.buf=buf;
7       }
8
9       public void run() {
10          while (true) {
11              try {
12                  Thread.sleep(2000);
13              } catch (InterruptedException e) {
14              }
15              int x=(int)(100 * Math.random());
16              buf.wstaw(x);
17              System.out.println(x);
18          }
19      }
20
21  }
22
23  public class Konsument extends Thread {
24
25      private IBufor buf;
26
27      public Konsument(IBufor buf){
28        this.buf=buf;
```

```
29      }
30
31      public void run() {
32          while (true) {
33              int x=buf.pobierz();
34              try {
35                  Thread.sleep(2000);
36              } catch (InterruptedException e) {
37              }
38
39              System.out.println("     "+x);
40          }
41      }
42
43  }
```

3.2. Problem czytelników i pisarzy

Rozważmy teraz kolejny ważny problem programowania współbieżnego, a mianowicie problem *czytelników i pisarzy*. Działające wątki, które wymagają dostępu do pewnych wspólnych danych będą się dzielić na dwie grupy:
— *czytelników*, których zadaniem jest odczyt wspólnych danych,
— *pisarzy*, którzy modyfikują wspólne dane.
Dodatkowo będziemy zakładać, że jednocześnie wielu czytelników będzie mogło odczytywać dane pod warunkiem, że nie modyfikuje ich żaden pisarz. Modyfikacja danych przez pisarza wyklucza odczyt danych przez czytelników oraz ich modyfikację przez innego pisarza.

Listing 3.4 zawiera klasę `Arbiter` wyposażoną w metody **startCzytania()**, **stopCzytania()**, **startPisania()** oraz **stopPisania()**, przy pomocy których wątki czytelników i pisarzy zgłaszają arbitrowi chęć rozpoczęcia czytania lub pisania oraz zakończenie tych czynności. Czytelnicy mogą rozpocząć czytanie gdy nikt nie pisze, zaś pisarze rozpoczynają pisanie, gdy nikt nie czyta i nikt nie pisze.

Listing 3.4. Rozwiązanie problemu czytelników i pisarzy (mozliwość zagłodzenia pisarzy)

```
1  public class Arbiter {
2
3      private int liczCzyt=0;
4      private boolean ktosPisze=false;
5
6      public synchronized void startCzytania() {
7
8          while(!ktosPisze){
```

```
 9          try {
10             wait();
11          }catch(InterruptedException e){
12          }
13       }
14       liczCzyt++;
15    }
16
17
18    public synchronized void startPisania () {
19
20       while((liczCzyt!=0)||(ktosPisze)){
21          try {
22             wait();
23          }catch(InterruptedException e){
24          }
25       }
26       ktosPisze=true;
27    }
28
29    public synchronized void stopPisania() {
30       ktosPisze=false;
31       notifyAll();
32    }
33
34    public synchronized void stopCzytania() {
35       liczCzyt--;
36       notifyAll();
37    }
38
39 }
40
41 public class Czytelnik extends Thread {
42
43   private Arbiter a;
44   private int nr;
45
46   public Czytelnik(Arbiter a,int nr){
47      this.a=a;
48      this.nr=nr;
49   }
50
51   public void run() {
52
53      while (true) {
54         a.startCzytania();
55         System.out.println("Start CZYTANIA "+nr);
56         // czytanie ...
57         System.out.println("Stop CZYTANIA "+nr);
58         a.stopCzytania();
59      }
```

```
60    }
61
62 }
63
64 public class Pisarz extends Thread {
65
66    private Arbiter a;
67    private int nr;
68
69    public Pisarz(Arbiter a,int nr){
70          this.a=a;
71          this.nr=nr;
72    }
73
74    public void run() {
75
76       while (true) {
77          a.startPisania();
78          System.out.println("Start PISANIA "+nr);
79          // pisanie ...
80          System.out.println("Stop PISANIA "+nr);
81          a.stopPisania();
82       }
83    }
84
85 }
```

Łatwo zauważyć, że rozwiązanie może doprowadzić do zagłodzenia pisarzy, gdyż oczekujący pisarz może być wstrzymywany przez czytelników, którzy mogą rozpoczynać czytanie gdy nikt nie pisze. Na listingu 3.5 podajemy rozwiązanie, w którym pisarz zgłasza chęć rozpoczęcia pisania poprzez wywołanie metody **chcePisac()**. Czytelnicy nie mogą rozpocząć czytania, gdy są pisarze oczekujący na możliwość rozpoczęcia pisania. Oczywiście może to doprowadzić do zagłodzenia czytelników.

Listing 3.5. Rozwiązanie problemu czytelników i pisarzy (klasy Arbiter i Pisarz)

```
1 public class Arbiter  {
2
3    private int liczCzyt=0;
4    private int liczPisa=0;
5    private boolean ktosPisze=false;
6
7    public synchronized void startCzytania() {
8
9          while(liczPisa!=0){
10            try {
11               wait();
```

```
12             }catch(InterruptedException e){
13             }
14           }
15           liczCzyt++;
16     }
17
18      public synchronized void startPisania () {
19
20           while((liczCzyt!=0)||(ktosPisze)){
21             try {
22               wait();
23             }catch(InterruptedException e){
24             }
25           }
26           ktosPisze=true;
27      }
28
29      public synchronized void chcePisac() {
30          liczPisa++;
31      }
32
33
34      public synchronized void stopPisania() {
35          ktosPisze=false;
36          liczPisa--;
37          notifyAll();
38      }
39
40      public synchronized void stopCzytania() {
41          liczCzyt--;
42          notifyAll();
43      }
44
45 }
46
47 public class Pisarz extends Thread {
48
49   private Arbiter a;
50   private int nr;
51
52   public Pisarz(Arbiter a,int nr){
53        this.a=a;
54        this.nr=nr;
55   }
56
57   public void run() {
58
59     while (true) {
60          a.chcePisac();
61          a.startPisania();
62          System.out.println("Start PISANIA "+nr);
```

```
63          // pisanie ...
64          System.out.println("Stop PISANIA "+nr);
65          a.stopPisania();
66      }
67    }
68
69 }
```

3.3. Zadania

3.3.1. Mechanizm bariery za pomocą monitorów

Bariera jest mechanizmem, który powoduje wstrzymanie wątków z pewnej grupy w określonym punkcie synchronizacyjnym, do czasu, aż ostatni wątek z grupy osiągnie ten punkt. Zaimplementuj mechanizm bariery za pomocą monitorów. Dołącz również kod, w którym bariera zostanie użyta.

3.3.2. Semafor za pomocą monitorów

Zaimplementuj za pomocą monitorów semafor zliczający.

3.3.3. Semafor binarny za pomocą monitorów

Zaimplementuj za pomocą monitorów semafor binarny.

3.3.4. Szeregowanie instrukcji z wykorzystaniem monitorów

Program współbieżny składa się z trzech wątków A, B i C, wykonujących w pętlach nieskończonych odpowiednio instrukcje: *Instrukcja A*, *Instrukcja B* oraz *Instrukcja C*. *Instrukcja A* wykonywana przez wątek A ma być wykonana przed *instrukcją B* z wątku B. Natomiast *instrukcja B* powinna być zawsze wykonana przed *instrukcją C* z wątku C. Wykorzystując monitory podaj definicje klas odpowiadających wątkom A, B, C. Dodaj program, w którym uruchomisz te wątki.

3.3.5. Problem ucztujących filozofów z wykorzystaniem monitorów

Napisz poprawne rozwiązanie problemu ucztujących filozofów z wykorzystaniem monitorów. Na rozwiązanie ma składać się klasa *Filozof* oraz klasa monitora zajmująca się przydziałem widelców, w której mają być metody *podniesWidelce* i *polozWidelce* [4].

Rozdział 4

Wybrane techniki

4.1. Blokady **54**
4.2. Operacje RMW **57**
4.3. Zadania **59**
4.3.1. Algorytm piekarniany 59
4.3.2. Wzajemne wykluczanie z wykorzystaniem klasy `Reentrantlock` 59
4.3.3. Problem czytelników i pisarzy z wykorzystaniem `ReentrantReadWriteLock` 59
4.3.4. Problem ucztujących filozofów z wykorzystaniem interfejsu `Lock` 59
4.3.5. Problem producenta-konsumenta z wykorzystaniem interfejsu `BlockingQueue` . . . 60

Przedstawimy teraz wybrane techniki programowania współbieżnego w języku Java. Ich stosowanie znacznie ułatwia rozwiązywanie trudniejszych problemów.

4.1. Blokady

Jednym z ważniejszych mechanizmów programowania współbieżnego są blokady (ang. *locks*). W języku Java są to obiekty klas implementujących interfejs przedstawiony na listingu 4.1.

Listing 4.1. Interfejs Lock

```
1 public interface Lock{
2
3   void lock();
4
5   void lockInterruptibly() throws InterruptedException;
6
7   boolean tryLock();
8
9   boolean tryLock(long time, TimeUnit unit)
10                              throws InterruptedException;
11  void unlock();
12
13  Condition newCondition();
14 }
```

Najczęściej blokady są wykorzystywane do rozwiązania problemu wzajemnego wykluczania. Ogólny schemat ich wykorzystania został przedstawiony na listingu 4.2. Wątek, który chce wejść do sekcji krytycznej wywołuje metodę **lock()**, co wstrzymuje go do momentu założenia blokady (w danej chwili tylko jeden wątek może założyć blokadę). Następnie (po zakończeniu wykonywania metody **lock()**) wątek wykonuje instrukcje sekcji krytycznej z obsługą ewentualnych błędów. Na koniec wątek zwalnia blokadę wywołując metodę **unlock()**. W interfejsie `Lock` dostępne są metody warunkowego nałożenia blokady **tryLock()** oraz metody **lock()**, które oczekują na założenie blokady przez określony czas, bądź z możliwością zakończenia oczekiwania na skutek otrzymanego sygnału przerwania.

Listing 4.2. Schemat wykorzystania blokady

```
1 Lock blokada = ...;
2 blokada.lock();
3 try {
4   // instrukcje sekcji krytycznej
5   ...
```

```
6 }catch(Exception e){
7   // obsługa błędów z sekcji krytycznej
8   ...
9 }finally{
10    blokada.unlock(); // wykonywane zawsze !
11 }
```

Poprawna implementacja interfejsu `Lock` jest zagadnieniem trudnym. Na listingu 4.3 przedstawiamy algorytm Patersona implementacji blokady dla dwóch wątków. Można udowodnić, że algorytm spełnia własność wzajemnego wykluczania oraz nie doprowadza do zakleszczenia i zagłodzenia [7].

Programista ma do dyspozycji klasę `ReentrantLock`, która implementuje interfejs `Lock` dla dowolnej liczby wątków.

Listing 4.3. Algorytm Patersona implementacji metod lock() i unlock() z interfejsu Lock (dla dwóch wątków)

```
1 public class LockPeterson implements Lock {
2
3     private AtomicIntegerArray flag;
4     private volatile int turn;
5
6     public LockPeterson() {
7         flag = new AtomicIntegerArray(2);
8         flag.set(0, 0);
9         flag.set(1, 0);
10    }
11
12    public void lock() {
13        int i = ThreadID.get();
14        int j = 1 - i;
15        flag.set(i, 1);
16        turn=j;
17
18        while (flag.get(j) == 1 && turn==j) {
19            Thread.yield();
20        }
21    }
22
23    public void unLock() {
24        int i = ThreadID.get();
25        flag.set(i, 0);
26    }
27 }
```

Interfejs `Lock` oferuje również bardzo ciekawe możliwości związane z metodą **newCondition()**, która zwraca obiekt klasy implementującej interfejs `Condition` (listing 4.4). Dzięki temu wątek, który nałożył blokadę będzie

mógł oczekiwać do momentu zajścia określonego warunku (zdarzenia). Wywołanie metody **await()** na danym warunku (obiekcie `Condition`) powoduje atomowe zwrócenie blokady i wstrzymanie wątku. Jego „obudzenie" nastąpi, gdy zostanie wywołana metoda **signalAll()** na danym warunku albo metoda **signal()** i wątek zostanie wybrany do obudzenia, bądź też wątek otrzyma sygnał przerwania. W każdym przypadku obudzony wątek będzie musiał ponownie założyć blokadę.

Listing 4.4. Interfejs Condition

```
1  public interface Condition{
2    void await() throws InterruptedException;
3    void awaitUninterruptibly();
4    long awaitNanos(long nanosTimeout)
5                               throws InterruptedException;
6    boolean await(long time, TimeUnit unit)
7                               throws InterruptedException;
8    boolean awaitUntil(Date deadline)
9                               throws InterruptedException;
10   void signal();
11   void signalAll();
12 }
```

Na listingu 4.5 przedstawiono implementację bufora (zamieszczoną w dokumentacji JDK [1]) przy pomocy blokady (obiek klasy `ReentrantLock`) oraz dwóch warunków (obiektów `Condition`) oznaczających odpowiednie niepełność i niepustość bufora. Wątki producenta i konsumenta wstawiają oraz pobierają z bufora tylko, gdy spełnione są właściwe warunki. Jeśli tak nie jest, wówczas wątki oczekują na ich spełnienie.

Listing 4.5. Implementacja bufora (przedstawiona w dokumentacji JDK)

```
1   class BoundedBuffer {
2     final Lock lock = new ReentrantLock();
3     final Condition notFull  = lock.newCondition();
4     final Condition notEmpty = lock.newCondition();
5
6     final Object[] items = new Object[100];
7     int putptr, takeptr, count;
8
9     public void put(Object x) throws InterruptedException {
10      lock.lock();
11      try {
12        while (count == items.length)
13          notFull.await();
14        items[putptr] = x;
```

[1] Zobacz dokumentację dla interfejsu `Condition`, która jest dostępna na stronie `http://docs.oracle.com/javase/7/docs/api/`

```
15          if (++putptr == items.length) putptr = 0;
16          ++count;
17          notEmpty.signal();
18        } finally {
19          lock.unlock();
20        }
21      }
22
23      public Object take() throws InterruptedException {
24        lock.lock();
25        try {
26          while (count == 0)
27            notEmpty.await();
28          Object x = items[takeptr];
29          if (++takeptr == items.length) takeptr = 0;
30          --count;
31          notFull.signal();
32          return x;
33        } finally {
34          lock.unlock();
35        }
36      }
37    }
```

4.2. Operacje RMW

Rozważmy rejestr (obiekt klasy `AtomicInteger`) przechowujący wartości całkowite i wyposażony w operacje (metody) typu RMW (ang. *read- modify-write*):

— **getAndSet(v)**, która atomowo zastępuje poprzednią wartość rejestru wartością **v** i zwraca poprzednia wartość,

— **getAndIncrement()**, która atomowo dodaje 1 do poprzedniej wartości i zwraca poprzednią wartość rejestru,

— **getAndAdd(k)**, która działa jak **getAndIncrement()**, ale dodaje wartość **k**,

— **compareAndSet(e,u)**, która ma dwa argumenty: wartość spodziewaną **e** oraz nową wartość **u**; jeśli wartość rejestru równa się **e**, wówczas zostaje atomowo zmieniona na **u**, a w innym przypadku pozostaje bez zmian,

— **get()**, która zwraca wartość rejestru.

Tego typu rejestr może być wykorzystany do rozwiązywania wielu ważnych problemów programowania współbieżnego. Jako przykład rozważmy następujący *problem konsensusu* [7]. Obiekt konsensusu jest wyposażony w metodę **decide(v)**, którą każdy z n wątków wywołuje z dowolnymi wartościami **v**, przy czym:

— wszystkie wątki otrzymują jako wynik tę samą wartość,
— zwracana wartość jest jedną z wartości wejściowych przekazanych w metodzie **decide(v)**.

Formalnie zapiszmy to w postaci przedstawionego na listingu 4.6 interfejsu oraz klasy abstrakcyjnej [7].

Listing 4.6. Interfejs Consensus i klasa ConsensusProtocol

```
1 public interface Consensus<T>{
2   T decide(T v);
3 }
4
5 public abstract class ConsensusProtocol<T>
6                                 implements Consensus<T>{
7   protected T[] proposed;
8
9   public ConsensusProtocol(int n){
10    proposed = (T[]) new Object(n);
11  }
12
13  void propose(T v){
14    proposed[ThreadID.get()]=v;
15  }
16
17  abstract public T decide(T v);
18
19 }
```

Listing 4.7 prezentuje rozwiązanie problemu konsensusu przy pomocy operacji **compareAndSet(e,u)**. Każdy wątek wywołuje metodę **decide()** podając swoją wartość, która jest zapisywana w składowej tablicy `proposed[]` o takim numerze, jak numer wątku. Rejestr atomowy **r** ma domyślnie wartość −1. Pierwszy wątek, który wywoła metodę **compareAndSet()** ustawi wartość rejestru na swój numer. Pozostałe wątki będą otrzymywać na wyjściu wartość ustawioną przez ten wątek.

Listing 4.7. Rozwiązanie problemu konsensusu

```
1 class CASConsensus<T> extends ConsensusProtocol<T> {
2
3    private final int FIRST = -1;
4    private AtomicInteger r = new AtomicInteger(FIRST);
5
6    public CASConsensus(int n){
7      super(n);
8    }
9
10   public T decide(T v) {
11     propose(v);
```

```
12        int i = ThreadID.get();
13        if (r.compareAndSet(FIRST, i))
14           return proposed[i];
15        else
16           return proposed[r.get()];
17     }
18  }
```

4.3. Zadania

4.3.1. Algorytm piekarniany

Zaimplementuj algorytm piekarniany rozwiązujący problem wzajemnego wykluczania dla N wątków.

4.3.2. Wzajemne wykluczanie z wykorzystaniem klasy `Reentrantlock`

Napisz rozwiązanie problemu wzajemnego wykluczania z wykorzystaniem klasy `ReentrantLock` z pakietu *java.util.concurrent.locks*.

4.3.3. Problem czytelników i pisarzy z wykorzystaniem `ReentrantReadWriteLock`

Napisz rozwiązanie problemu czytelników i pisarzy z wykorzystaniem klasy `ReentrantReadWriteLock`.

4.3.4. Problem ucztujących filozofów z wykorzystaniem interfejsu `Lock`

Napisz rozwiązanie ucztujących filozofów z wykorzystaniem interfejsu `Lock`. Rozwiązanie powinno spełniać następujące założenia:

a) Każdy filozof w pętli wywołuje kolejno metody **mysl()** i **jedz()**.
b) Czas wykonywania metody **mysl()** jest dla każdego filozofa losowy i początkowo ustalany w konstruktorze.
c) Po wywołaniu metody **jedz()**, filozof podejmuje 5 prób sięgnięcia po pałeczki.
d) Z każdą pałeczką ma być skojarzony obiekt typu `Lock` (klasy `ReentrantLock`).
e) Sięganie po pałeczki (każdej z osobna) ma być realizowane za pomocą jednej z metod **tryLock** z interfejsu `Lock`.
f) Jeśli próba się nie powiedzie wówczas odpowiedni wątek wypisuje na ekranie napis, wówczas filozof podejmuje kolejną próbę.

g) Jeśli nie powiodło się 5 prób, wówczas filozof kończy metodę **jedz()** głodny i myśli dalej, ale o połowę krócej niż ostatnio.
h) Po trzech kolejnych nieudanych próbach wykonania **jedz()** filozof „umiera z głodu”.
i) Jeśli natomiast filozofowi uda się wykonać **jedz()**, wówczas przechodzi z powrotem do metody **mysl()**, wcześniej losowo wybierając czas wykonywania tej metody.

4.3.5. Problem producenta-konsumenta z wykorzystaniem interfejsu `BlockingQueue`

Napisz prostą ilustrację problemu producenta-konsumenta z wykorzystaniem interfejsu `BlockingQueue` z pakietu *java.util.concurrent*.

Rozdział 5

Programowanie rozproszone

5.1. Remote Method Invocation **62**
5.1.1. Prosta aplikacja RMI 62
5.1.2. Dynamiczne ładowanie klas 66
5.2. Standard CORBA **69**
5.3. Aplikacja do prowadzenia rozmów **74**
5.4. Zadania **80**
5.4.1. NWD dużych liczb 80
5.4.2. NWD jako zadanie dla interfejsu `Compute` . . . 80
5.4.3. Serwer „echo” 81

Zajmiemy się teraz metodami realizacji programowania rozproszonego w języku Java. Omówimy na przykładach dwa popularne interfejsy oparte na paradygmacie zdalnego wywoływania procedur (ang. *Remote Procedure Call*), a mianowicie RMI (ang. *Remote Method Invocation*) oraz CORBA (ang. *Common Object Request Broker Architecture*).

5.1. Remote Method Invocation

RMI (ang. *Remote Method Invocation*) jest stosunkowo prostym mechanizmem umożliwiającym zdalne wywoływanie metod na rzecz obiektów będących instancjami klas zdefiniowanych w języku Java [11]. Zakłada on, że aplikacja typu *serwer* udostępnia obiekt implementujący zdalny interfejs (ang. *Remote Interface*). Aplikacja typu *klient* uzyskuje referencję do takiego obiektu za pośrednictwem mechanizmu *RMI Registry* i wywołuje zdalnie na jego rzecz metody opisane w zdalnym interfejsie. Parametry aktualne wywołań są serializowane (klasy tych obiektów muszą implementować interfejs `Serializable`) i przekazywane do serwera wraz z żądaniem wywołania metody. Podobnie serializowane są obiekty – wyniki zwracane do miejsca wywołania.

5.1.1. Prosta aplikacja RMI

Jako przykład użycia RMI rozważmy aplikację typu *klient-serwer*. Klient wysyła do serwera łańcuch znaków, który jest przetwarzany przez serwer, a dokładnie doklejana jest do niego data i godzina. Następnie przetworzony łańcuch jest wysyłany do miejsca wywołania metody, czyli do klienta. Tworzenie aplikacji rozpoczynamy od zdefiniowania zdalnego interfejsu (listing 5.1).

Listing 5.1. Zdalny interfejs

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface Hello extends Remote {
5     String sayHello(String name) throws RemoteException;
6 }
```

Wymaga się aby zdalny interfejs spełniał następujące warunki:
— dziedziczył po interfejsie `java.rmi.Remote`,
— każda metoda, która ma być wywoływana zdalnie była zadeklarowana jako wyrzucająca wyjątek klasy `java.rmi.RemoteException`,

— klasy parametrów formalnych oraz typ wyniku implementowały interfejs `Serializable`.

Następnym krokiem jest zwykle implementacja zdalnego interfejsu (listing 5.2). Zauważmy, że klasa `HelloImpl` dziedziczy po klasie bibliotecznej `UnicastRemoteObject` z pakietu `java.rmi.server`. Istnieje też nieco inny wariant, który omówimy później.

Listing 5.2. Implementacja zdalnego interfejsu

```
1  import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3  import java.util.*;
4
5  public class HelloImpl extends UnicastRemoteObject
6      implements Hello {
7
8      public HelloImpl() throws RemoteException {
9          super();
10     }
11
12     public String sayHello(String name)
13                                throws RemoteException {
14         System.out.println("Wykonuję metodę sayHello!");
15         return  "Hello "+name+" "+(new Date().toString());
16     }
17
18 }
```

Następnie możemy napisać prostą aplikację serwera (listing 5.3). Najważniejszymi czynnościami, które trzeba w niej uwzględnić jest utworzenie obiektu `RMISecurityManager`, który będzie odpowiedzialny za bezpieczeństwo aplikacji. Następnie tworzony jest serwis nazw, w którym rejestrowany jest pod odpowiednią nazwą utworzony obiekt zdalny.

Listing 5.3. Aplikacja serwera

```
1  import java.rmi.Naming;
2  import java.rmi.RMISecurityManager;
3  import java.rmi.registry.*;
4
5  public class HelloServer {
6
7    public static void main(String args[]) {
8
9      // utwórz i zainstaluj security managera
10     // odpowiedzialnego za bezpieczeństwo aplikacji
11     if (System.getSecurityManager() == null) {
12       System.setSecurityManager
13                      (new RMISecurityManager());
```

```
14      }
15
16      try {
17        // utwórz serwis nazw obiektów RMI Registry
18        Registry reg=
19                LocateRegistry.createRegistry(6999);
20
21        // utwórz obiekt ...
22        HelloImpl obj = new HelloImpl();
23
24        // ... i zarejstruj go pod nazwą "HelloServer"
25        Naming.rebind("//localhost:6999/HelloServer", obj);
26
27        System.out.println("HelloServer ");
28        System.out.println("zapisany w rejestrze");
29      }catch (Exception e){
30        System.out.println("HelloImpl err: "
31                                      + e.getMessage());
32      }
33    }
34 }
```

Ostatnią czynnością jest stworzenie aplikacji klienckiej (listing 5.4), w której uzyskujemy referencję do zdalnego obiektu, wywołujemy zdalnie metodę `sayHello` i wyświetlamy otrzymany wynik.

Listing 5.4. Aplikacja klienta

```
1 import java.rmi.Naming;
2 import java.rmi.RemoteException;
3
4 public class HelloCli {
5
6   public static void main(String[] args) {
7
8         String message = "?";
9         Hello obj = null;
10
11      try {
12          obj = (Hello)Naming.lookup(
13                  "//matrix.umcs.lublin.pl:6999" +
14                          "/HelloServer");
15          message = obj.sayHello("Przemek");
16          System.out.println(message);
17
18      } catch (Exception e) {
19          System.out.println("Błąd: "+e.getMessage());
20      }
21    }
22 }
```

Aby uruchomić aplikację serwera należy wcześniej utworzyć plik zawierający politykę bezpieczeństwa dla aplikacji. Najłatwiej jest to uczynić przy pomocy programu `policytool`, który wchodzi w skład środowiska do tworzenia i uruchamiania aplikacji w języku Java. Na potrzeby uruchamiania aplikacji można się posłużyć plikiem postaci takiej, jak na listingu 5.5.

Listing 5.5. Testowy plik z polityką bezpieczeństwa

```
1 grant {
2     // zezwalamy na wszystko
3     permission java.security.AllPermission;
4 };
```

Kompilację i uruchomienie programu realizujemy następująco:
— dla serwera
```
server$ javac *.java
server$ java -Djava.security.policy=pol2   HelloServer
```
— dla klienta
```
klient$ javac *.java
klient$ java HelloCli
```
W przypadku, gdy klasa implementująca zdalny interfejs dziedziczy już po pewnej klasie, wówczas można ją zadeklarować jako klasę implementującą zdalny interfejs, zaś w programie serwera posłużyć się statyczną metodą `exportObject` klasy `UnicastRemoteObject` (listing 5.6).

Listing 5.6. Eksport zdalnego obiektu

```
1  public class HelloImpl implements Hello {
2
3      public HelloImpl() throws RemoteException {
4          super();
5      }
6
7      public String sayHello(String name)
8                              throws RemoteException {
9          return  name+" "+(new Date().toString());
10     }
11
12 }
13
14 public class HelloServer {
15
16   public static void main(String args[]) {
17
18     if (System.getSecurityManager() == null) {
19         System.setSecurityManager(new RMISecurityManager());
20     }
21
22     try {
```

```
23          Registry reg=LocateRegistry.createRegistry(6999);
24          HelloImpl obj = new HelloImpl();
25          Remote robj=
26                  UnicastRemoteObject.exportObject(obj,0);
27          Naming.rebind("//localhost:6999/HelloServer", robj);
28      }catch (Exception e){
29          System.out.println("Błąd: " + e.getMessage());
30      }
31    }
32 }
```

5.1.2. Dynamiczne ładowanie klas

Zajmiemy się teraz nieco bardziej skomplikowaną aplikacją [11], która wykorzystuje technikę dynamicznego ładowania klas w RMI . Będzie ona udostępniać funkcjonalność prostego serwera aplikacji, gdzie klient będzie wysyłać do serwera zadania obliczeniowe w postaci obiektów odpowiednio zdefiniowanych klas, które nie muszą być zdefiniowane w czasie tworzenia aplikacji serwera. Rozważymy w tym celu definicję interfejsów przedstawionych na listingach 5.7 oraz 5.8.

Listing 5.7. Interfejs Task - zadanie obliczeniowe

```
1 import java.io.Serializable;
2 public interface Task extends Serializable {
3   Object execute();
4 }
```

Listing 5.8. Interfejs zdalny Compute

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface Compute extends Remote {
5     Object executeTask(Task t) throws RemoteException;
6 }
```

Implementacją interfejsu `Compute` jest przedstawiona na listingu 5.9 klasa `ComputeEngine`. Zadaniem metody `executeTask` jest wykonanie otrzymanego w parametrze `task` zadania obliczeniowego, które musi implementować interfejs `Task`, a zatem posiadać metodę `execute()`. Listing 5.10 przedstawia prosty program serwera.

Listing 5.9. Implementacja interfejsu Compute

```
1  import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3
4  public class ComputeEngine
5        extends UnicastRemoteObject implements Compute {
6
7     public ComputeEngine() throws RemoteException {
8        super();
9     }
10
11    public Object executeTask(Task t)
12                            throws RemoteException {
13       Object o=t.execute();
14       System.out.println("koniec !");
15       return o;
16    }
17 }
```

Listing 5.10. Program serwera

```
1  import java.rmi.Naming;
2  import java.rmi.RemoteException;
3  import java.rmi.RMISecurityManager;
4  import java.rmi.server.UnicastRemoteObject;
5  import java.rmi.registry.*;
6  import java.util.*;
7
8  public class ComputeServer {
9
10   public static void main(String args[]) {
11
12     if (System.getSecurityManager() == null) {
13       System.setSecurityManager(new RMISecurityManager());
14     }
15     try {
16       Registry reg=LocateRegistry.createRegistry(6999);
17       ComputeEngine obj = new ComputeEngine();
18       Naming.rebind("//127.0.0.1:6999/ComputeEngine", obj);
19     } catch (Exception e) {
20        System.out.println("HelloImpl err: "
21                                          + e.getMessage());
22     }
23   }
24 }
```

Jako przykład rozważmy proste zadanie obliczeniowe, polegające na dodawaniu dużych liczb całkowitych. Na listingu 5.11 przedstawiono klasę implementującą interfejs `Task`. Listing 5.12 zawiera prosty program klienta wysyłającego do serwera zadanie obliczeniowe i wyświetlającego wynik obliczeń.

Listing 5.11. Zadanie obliczeniowe

```
1  import java.math.BigInteger;
2
3  public class BigSum implements Task {
4
5    private BigInteger d1;
6    private BigInteger d2;
7
8    public BigSum(BigInteger d1, BigInteger d2) {
9      this.d1=d1;
10     this.d2=d2;
11   }
12
13   public Object execute() {
14     return d1.add(d2);
15   }
16
17 }
```

Listing 5.12. Program klienta

```
1  import java.math.BigInteger;
2  import java.io.*;
3  import java.rmi.Naming;
4
5  public class ComputeCli {
6
7    public static void main(String[] args) {
8
9      try {
10       Compute obj = (Compute)Naming.lookup("//" +
11                 "127.0.0.1:6999" + "/ComputeEngine");
12       BufferedReader in=new BufferedReader(
13                          new InputStreamReader(System.in));
14       System.out.print("x1= ");
15       String z1=in.readLine();
16       System.out.print("x2= ");
17       String z2=in.readLine();
18
19       BigSum task=new BigSum(
20                   new BigInteger(z1),new BigInteger(z2));
21       BigInteger wynik = (BigInteger)obj.executeTask(task);
22       System.out.println(wynik);
```

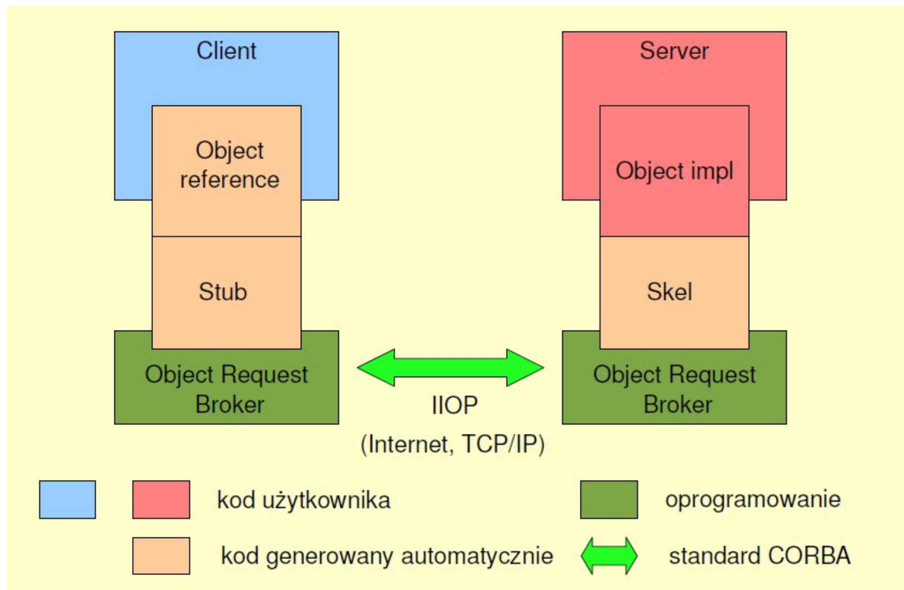
```
23
24     } catch (Exception e) {
25         System.out.println("Błąd: "+ e.getMessage());
26     }
27   }
28
29 }
```

Kompilację i uruchomienie programu realizujemy następująco:
— dla serwera
```
server$ javac *.java
server$ java -Djava.security.policy=policy \
          -Djava.rmi.server.codebase=file:///.  ComputeServer
```
— dla klienta
```
klient$ javac *.java
klient$ java ComputeCli
```
Pliki `*.class` z definicjami klas implementującymi interfejs `Task` należy umieszczać w katalogu bieżącym, w którym znajduje się program serwera. Możliwe jest również wykorzystanie protokołu `http` i serwera stron WWW jako mechanizmu transportującego pliki zawierające skompilowane definicje klas. Wtedy we własności `codebase` należy podać pełny URL dla plików `*.class`.

5.2. Standard CORBA

Standard CORBA (ang. *Common Object Request Broker Architecture*) opisuje architekturę i mechanizmy umożliwiające zdalne wywoływanie procedur (metod) w środowisku heterogenicznym, gdzie serwer i klient mogą być pisane w różnych językach programowania [9]. Podobnie jak w przypadku RMI, standard CORBA zakłada, że aplikacja typu *serwer* udostępnia obiekt implementujący interfejs zdalny napisany w języku IDL (ang. *Interface Description Language*).

Podstawowe elementy architektury CORBA zostały przedstawione na rysunku 5.1. Najważniejszymi jej elementami są:
— **IDL**: język definiowania interfejsów, na podstawie których automatycznie generowany jest kod źródłowy odpowiedzialny za komunikację ze zdalnymi obiektami,
— **ORB**: (ang. *Object Request Broker*), mechanizm komunikacji między aplikacjami wykorzystującymi rozproszone obiekty, którego ważnym elementem jest POA (ang. *Portable Object Adapter*),
— **IIOP**: (ang. Internet InterORB Protocol), czyli protokół komunikacji pomiędzy ORB-ami.



Rysunek 5.1. Zdalne wywoływanie metod w aplikacji CORBA

Rozważmy teraz prosty przykład aplikacji klient-serwer w architekturze CORBA. Listing 5.13 zawiera prosty interfejs, w którym zdefiniowano dwie metody:

echoString przetwarza i zwraca otrzymany w parametrze wejściowym (słowo kluczowe `in`) łańcuch znaków,

shutdown służy do zdalnego (z poziomu aplikacji klienta) zamknięcia aplikacji serwera.

Listing 5.13. Opis interfejsu w języku IDL

```
1 interface Echo {
2   string echoString (in string mesg);
3   oneway void shutdown();
4 };
```

Na listingu 5.14 przedstawiamy prostą implementację zdefiniowanego wyżej interfejsu. Klasa `EchoPOA` została wygenerowana przez generator interfejsów `idlj`, który należy uruchomić następującym poleceniem:

```
server$ idlj -fall echo.idl
server$
```

Listing 5.14. Implementacja interfejsu

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.*;

class EchoImpl extends EchoPOA {
    private ORB orb;

    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    public String echoString(String mesg) {
        String data=new Date().toString();
        System.out.println(data+": "+mesg);
        return mesg+" "+data;
    }

    public void shutdown() {
        orb.shutdown(false);
    }
}
```

Listing 5.15. Program serwer

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

public class EchoServer {

  public static void main(String args[]) {
    try{
      // tworzenie i inicjacja ORB-a
      ORB orb = ORB.init(args, null);

      // pobranie referencji do rootpoa
      POA rootpoa =
        POAHelper.narrow
       (orb.resolve_initial_references("RootPOA"));

      // aktywacja POAManager-a
      rootpoa.the_POAManager().activate();

      // utworzenia servant-a i rejestracja ORB-a
```

```
24          EchoImpl echoImpl = new EchoImpl();
25          echoImpl.setORB(orb);
26
27          // pobranie referencji do servanta
28          org.omg.CORBA.Object ref =
29               rootpoa.servant_to_reference(echoImpl);
30          Echo href = EchoHelper.narrow(ref);
31
32          System.out.println("EchoServer gotowy ...\n");
33          System.out.println(""+href);
34
35          // oczekiwanie na wywołania klientów
36          orb.run();
37        }catch (Exception e) {
38            System.err.println("BŁĄD: " + e);
39        }
40
41        System.out.println("EchooServer kończy pracę");
42
43      }
44 }
```

Listing 5.16. Program klient

```
 1 import org.omg.CosNaming.*;
 2 import org.omg.CosNaming.NamingContextPackage.*;
 3 import org.omg.CORBA.*;
 4
 5 public class EchoClient {
 6     static Echo echoImpl;
 7
 8     public static void main(String args[]){
 9      try{
10         // inicjacja ORB-a
11          ORB orb = ORB.init(args, null);
12
13         // wzięcie referencji do obiektu
14          echoImpl =
15            EchoHelper.narrow(orb.string_to_object(args[0]));
16
17         // wyświetlenie odpowiedzi serwera
18          System.out.println(echoImpl.echoString(args[1]));
19
20      } catch (Exception e) {
21          System.out.println("ERROR : " + e) ;
22          e.printStackTrace(System.out);
23      }
24    }
25
26 }
```

Listingi 5.15 i 5.16 zawierają odpowiednio kody źródłowe programów serwera i klienta. Uruchomienie programu serwera należy przeprowadzić przedstawionym poniżej poleceniem. W odpowiedzi otrzymamy komunikat o gotowości serwera oraz łańcuch znaków IOR (ang. *Interoperable Object Reference*) stanowiący znakową reprezentację referencji do obiektu.

```
server$ java EchoServer
EchoServer gotowy ...

IOR:000000000000000d49444c3a4563686f3a312e30000000000000000100
0000000000008a000102000000000f3231322e3138322e39332e3231340000
b210000000000031afabcb0000000020201796fb0000000100000000000000
0100000008526f6f74504f41000000000800000001000000001400000000000
000200000001000000200000000000010001000000020501000100010020000
01010900000001000101000000002600000000020002

server$
```

Uruchomienie klienta (polecenie `java EchoClient`) powinno zawierać dwa parametry. W pierwszym podajemy łańcuch znaków reprezentujący referencję do obiektu (czyli IOR) oraz łańcuch znaków, który ma być przesłany do serwera. Oczywiście uruchamianie programów z wykorzystaniem IOR-a może być uciążliwe i dlatego znacznie wygodniej jest posłużyć się serwisem **NameService** udostępnianym przez ORB-a. Wówczas program serwera po utworzeniu obiektu rejestruje referencję pod konkretną nazwą, zaś program serwera uzyskuje referencję za pośrednictwem tegoż serwisu. Listingi 5.17 i 5.18 prezentują modyfikacje programów serwera i klienta, które uwzględniają obsługę serwisu nazw. Uruchomienie odpowiednich programów po stronie serwera należy przeprowadzić przy pomocy następujących poleceń.

```
server$ orbd -ORBInitialPort 1050 -ORBInitialHost localhost &
server$ java EchoClient -ORBInitialPort 1050 \
                             -ORBInitialHost localhost
```

Program klienta uruchamiamy ze wskazaniem komputera i portu, na którym działa serwer nazw.

```
client$ java EchoServer -ORBInitialPort 1050 \
                        -ORBInitialHost server.umcs.pl
client$
```

Listing 5.17. Obsługa serwisu nazw w programie serwera

```
1 org.omg.CORBA.Object objRef =
2     orb.resolve_initial_references("NameService");
3 NamingContextExt ncRef =
```

```
4     NamingContextExtHelper.narrow(objRef);
5 String name = "EchoServer";
6 NameComponent path[] = ncRef.to_name( name );
7 ncRef.rebind(path, href);
```

Listing 5.18. Obsługa serwisu nazw w programie klienta

```
1 org.omg.CORBA.Object objRef =
2     orb.resolve_initial_references("NameService");
3 NamingContextExt ncRef =
4     NamingContextExtHelper.narrow(objRef);
5 String name = "EchoServer";
6 echoImpl = EchoHelper.narrow(ncRef.resolve_str(name));
```

5.3. Aplikacja do prowadzenia rozmów

Rozważmy teraz projekt bardziej złożonej aplikacji, który zrealizujemy wykorzystując mechanizmy architektury CORBA. Założenia przedstawiają się następująco:

— elementami składowymi są programy serwer oraz klient, który jest uruchamiany przez uczestników rozmowy,
— klient rejestruje się na serwerze i dostaje na czas sesji unikalny numer,
— każdy klient wysyła do serwera komunikaty, które są wyświetlane na ekranach wszystkich zarejestrowanych klientów,
— po wyrejestrowaniu, klient kończy udział w rozmowie.

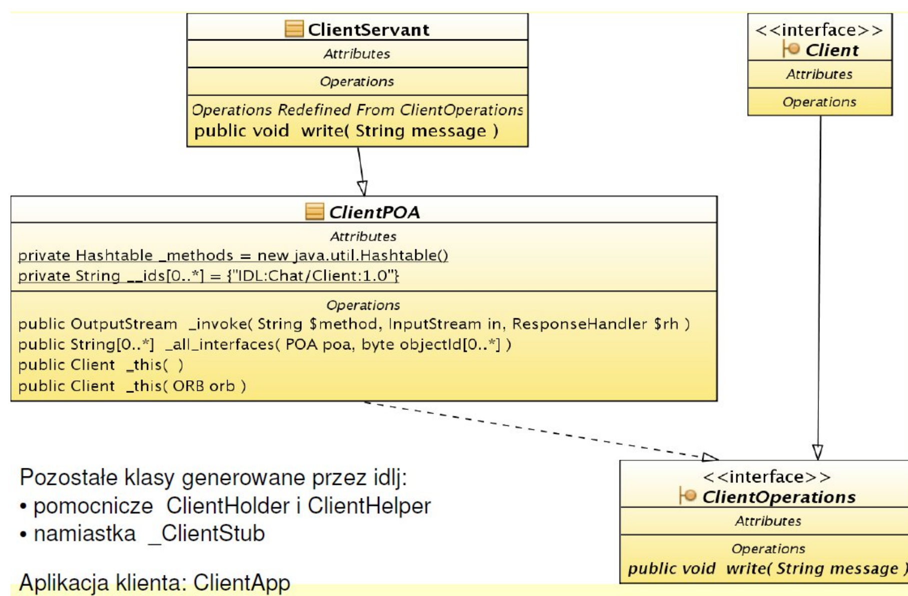
Na listingu 5.19 przedstawiono moduł języka IDL, który zawiera dwa interfejsy dla aplikacji klienta i serwera, gdyż obie aplikacje będą działać jako serwer i klient w architekturze CORBA. Metoda **write()** w interfejsie `Client` będzie służyć serwerowi do wysyłania komunikatów na ekran aplikacji klienta, który rejestrując się na serwerze przy pomocy metody **signIn()** będzie przekazywać referencję zdalną do swojego obiektu (serwanta implementującego interfejs `Client`). Wyrejestrowanie się będzie realizowane przy pomocy metody **signOut()**. Najważniejszą metodą interfejsu `Server` jest wywoływana asynchronicznie metoda **sendMessage()**, przy pomocy której klient przesyła do serwera komunikat rozsyłany przez serwer do wszystkich klientów.

Listing 5.19. Moduł IDL z interfejsami serwera i klienta

```
1 module Chat {
2   interface Client{
3     oneway void write(in string message);
4   };
```

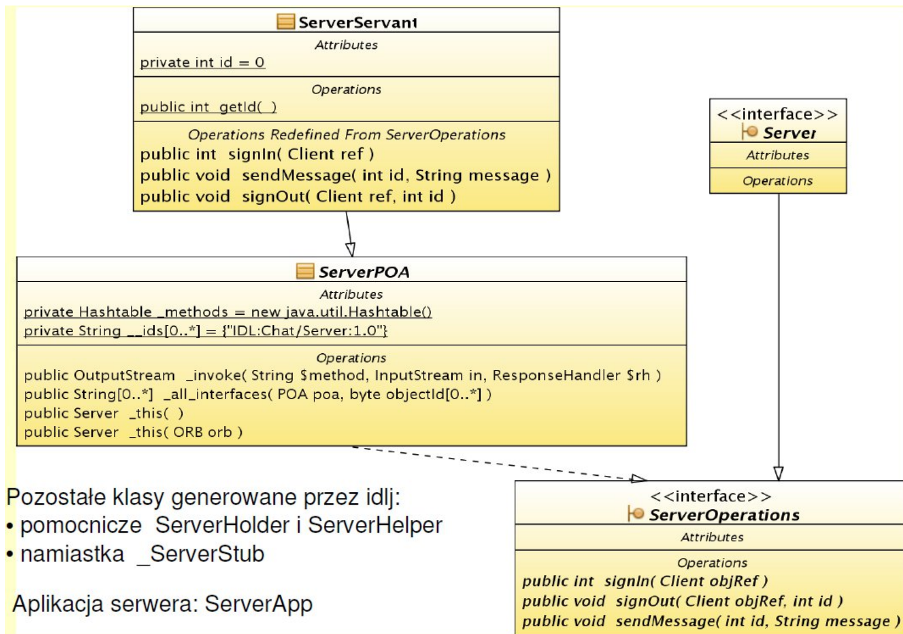
```
5   interface Server{
6     long signIn(in Client objRef);
7     oneway void signOut(in Client objRef, in long id);
8     oneway void sendMessage(in long id, in string message);
9   };
10 };
```

Na rysunkach 5.2 oraz 5.3 przedstawiono diagramy klas ilustrujące zależności pomiędzy najważniejszymi klasami po stronie klienta i serwera. Z wyjątkiem klas `ClientServant` oraz `ServerServant` zostały one wygenerowane automatycznie przez kompilator `idlj`. Wymienione klasy (dziedziczące odpowiednio po `ClientPOA` i `ServerPOA`) implementują zdalne obiekty (serwanty) po stronie klienta i serwera.



Rysunek 5.2. Podstawowe klasy klienta

Na rysunku 5.4 przedstawiono w postaci diagramu sekwencji ogólny schemat działania aplikacji. Program serwera (klasa `ServerApp`) rozpoczyna działanie od utworzenia obiektu serwanta, który rejestruje w serwisie nazw (usługa `NameService`) pod nazwą `ChatServer`. Aplikacja klienta (klasa `ClientApp`) pobiera referencję do obiektu `ChatServer` i rejestruje się w aplikacji serwera. Następnie pobiera od użytkownika kolejne komunikaty, które przesyła do serwera wywołując metodę **sendMessage()** na rzecz



Rysunek 5.3. Podstawowe klasy serwera

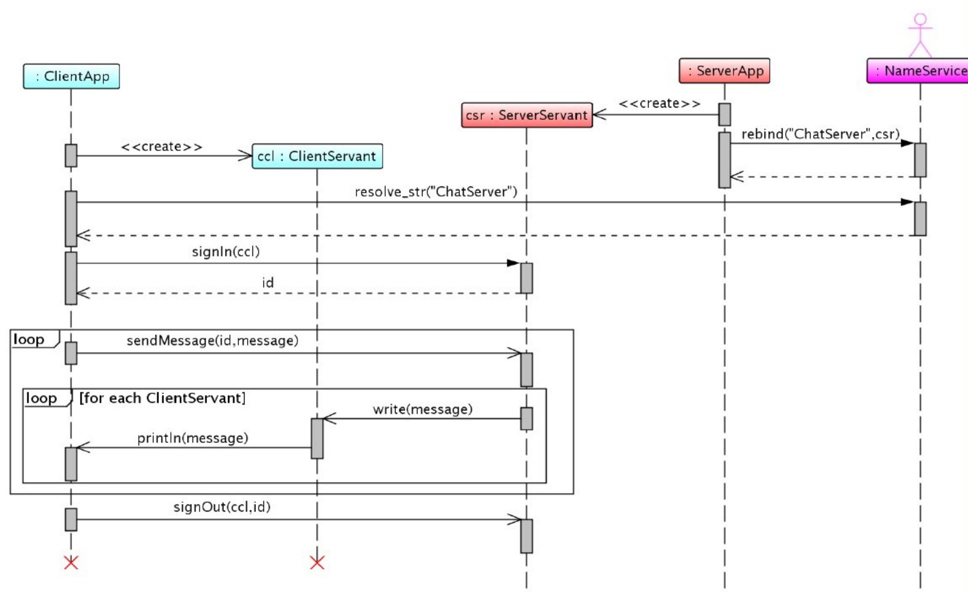
obiektu `ChatServer`, który rozsyła ją do wszystkich zarejestrowanych klientów wywołując metodę **write()** na rzecz ich serwantów. Aplikacja klienta wyrejestrowuje się z serwera za pomocą wywołania **signOut()**.

Listing 5.20 prezentuje prostą implementację serwanta po stronie klienta. Kod metody **write()** ogranicza się do wyświetlenia komunikatu na ekranie.

Listing 5.20. Implementacja serwanta po stronie klienta

```
1 import Chat.*;
2 import org.omg.CORBA.*;
3
4 public class ClientServant extends ClientPOA{
5
6   public void write(String message){
7     System.out.println(message);
8   }
9 }
```

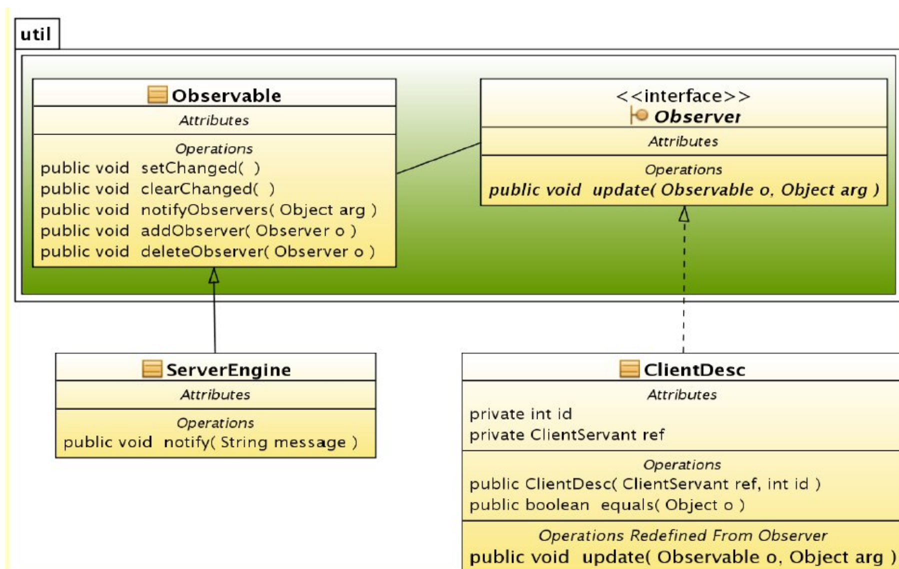
Na listingu 5.20 zaprezentowano implementację serwanta po stronie serwera. Wykorzystano *wzorzec projektowy obserwatora*, którego najważniejsze elementy ilustruje rysunek 5.5. Obiekt klasa `ServerEngine` jest obserwowany przez obiekty klasy `ClientDesc` implementującej interfejs `Observer`.



Rysunek 5.4. Schemat działania aplikacji

Listing 5.21. Implementacja serwanta po stronie serwera

```
1 class ServerEngine extends Observable {
2    public void notify(String m){
3      setChanged();
4      notifyObservers(m);
5      clearChanged();
6    }
7 }
8
9 class ClientDesc implements Observer{
10
11    Client ref;
12    int id;
13
14    ClientDesc(Client ref,int id) {
15       this.ref=ref;
16       this.id=id;
17    }
18
19    public void update(Observable o,java.lang.Object arg){
20      try {
21        this.ref.write((String)arg);
22      }catch(Exception e){
```



Rysunek 5.5. Wykorzystanie wzorca obserwatora

```
23          System.out.println("Błąd zapisu do "+ref);
24          o.deleteObserver(new ClientDesc(ref,0));
25          System.out.println("Klient został usunięty");
26        }
27      }
28
29      public boolean equals(java.lang.Object o){
30        return (this.id==((ClientDesc)o).id)||
31          (this.ref==((ClientDesc)o).ref);
32      }
33
34  }
35
36  public class ServerServant extends ServerPOA {
37
38    private static int id=0;
39    private ServerEngine lc=new ServerEngine();
40
41    public static synchronized int getId(){
42      id++;
43      return id;
44    }
45
46    public int signIn(Client ref){
47      int id=getId();
48      lc.addObserver(new ClientDesc(ref,id));
```

```
49      return id;
50    }
51
52    public void sendMessage(int id,String message){
53      String mess=id+": "+message;
54      System.out.println(mess);
55      lc.notify(mess);
56    }
57
58    public void signOut(Client ref,int id){
59      lc.deleteObserver(new ClientDesc(ref,id));
60    }
61
62 }
```

Listingi 5.22 i 5.23 zawierają najważniejsze elementy aplikacji serwera i klienta. Rola serwera ogranicza się do zainicjowania środowiska CORBA i rejestracji serwanta `tt ChatServer`. Aplikacja klienta, po wzięciu referencji do obiektu `tt ChatServer`, pobiera od użytkownika kolejne komunikaty i przesyła je do serwera.

Listing 5.22. Elementy kodu serwera

```
1 //inicjacja ORBa
2   ORB orb = ORB.init(args, null);
3 //wzięcie referencji do RootPOA i aktywacja POAManager-a
4   POA rootpoa =
5        POAHelper.narrow(
6        orb.resolve_initial_references("RootPOA"));
7    rootpoa.the_POAManager().activate();
8 //tworzenie servanta
9    ServerServant cs = new ServerServant();
10   org.omg.CORBA.Object ref =
11           rootpoa.servant_to_reference(cs);
12   Server csr = ServerHelper.narrow(ref);
13 //wzięcie referencji do serwisu NameService
14   org.omg.CORBA.Object objRef =
15   orb.resolve_initial_references("NameService");
16   NamingContextExt ncRef =
17      NamingContextExtHelper.narrow(objRef);
18 //rejestracja serwanta w NameService
19   NameComponent path[] = ncRef.to_name("ChatServer");
20   ncRef.rebind(path,csr);
21 //oczekiwanie na zdalne wywołania
22   orb.run();
```

Listing 5.23. Elementy kodu klienta

```
1 //zainicjowanie środowiska CORBA
2  ORB orb = ORB.init(args, null);
3  POA rootpoa =
4        POAHelper.narrow(
5        orb.resolve_initial_references("RootPOA"));
6   rootpoa.the_POAManager().activate();
7   org.omg.CORBA.Object objRef =
8    orb.resolve_initial_references("NameService");
9   NamingContextExt ncRef =
10   NamingContextExtHelper.narrow(objRef);
11 //tworzenie serwanta
12  ClientServant ccs = new ClientServant();
13  org.omg.CORBA.Object ref =
14    rootpoa.servant_to_reference(ccs);
15  Client href = ClientHelper.narrow(ref);
16 //pobranie referencji do zdalnego obiektu
17  Server cser =
18    ServerHelper.narrow(ncRef.resolve_str("ChatServer"));
19 //rejestracja na serwerze
20  int myId=cser.signIn(href);
21  System.out.println("Twój numer : " + myId);
22 //pętla główna
23  BufferedReader in =
24   new BufferedReader(new InputStreamReader(System.in));
25   String s;
26   while (!(s=in.readLine()).equals("/#")) {
27     cser.sendMessage(myId,s);
28   }
29   cser.signOut(href,myId);
```

5.4. Zadania

5.4.1. NWD dużych liczb

Podaj definicję interfejsu zdalnego (Java RMI) dla serwisu wyznaczania NWD (największego wspólnego dzielnika) dwóch dużych liczb całkowitych dodatnich, które przekazywane są jako łańcuchy znaków zawierających cyfry (użyj klasy `BigInteger`). Podaj definicję klasy implementującej ten interfejs oraz definicje zdalnego serwera oraz klienta.

5.4.2. NWD jako zadanie dla interfejsu `Compute`

Zdefiniuj liczenie NWD dwóch dużych liczb dodatnich (jak w poprzednim zdaniu), ale tym razem jako zadanie obliczeniowe dla interfejsu `Compute` z listingu 5.8.

5.4.3. Serwer „echo”

Wykorzystując Java RMI, napisz prosty serwer „echo”. Funkcja echo serwera powinna zwracać parametr, który przyjmuje. Rozwiązanie powinno zawierać deklarację interfejsu, definicję klasy, która ten interfejs implementuje oraz klasę serwera. Nie musisz implementować klasy klienta.

Rozdział 6

Rozwiązania zadań

6.1. Podstawowe pojęcia dotyczące współbieżności **85**
6.1.1. Rozwiązanie zadania 1.4.1 85
6.1.2. Rozwiązanie zadania 1.4.2 85
6.1.3. Rozwiązanie zadania 1.4.3 86
6.1.4. Rozwiązanie zadania 1.4.4 88
6.1.5. Rozwiązanie zadania 1.4.5 90
6.1.6. Rozwiązanie zadania 1.4.6 92
6.1.7. Rozwiązanie zadania 1.4.7 94
6.1.8. Rozwiązanie zadania 1.4.8 95
6.2. Semafory **97**
6.2.1. Rozwiązanie zadania 2.4.1 97
6.2.2. Rozwiązanie zadania 2.4.2 98
6.2.3. Rozwiązanie zadania 2.4.3 99
6.2.4. Rozwiązanie zadania 2.4.4 101
6.2.5. Rozwiązanie zadania 2.4.5 103
6.2.6. Rozwiązanie zadania 2.4.6 105
6.2.7. Rozwiązanie zadania 2.4.7 107
6.2.8. Rozwiązanie zadania 2.4.8 110
6.2.9. Rozwiązanie zadania 2.4.9 112
6.3. Monitory **115**
6.3.1. Rozwiązanie zadania 3.3.1 115
6.3.2. Rozwiązanie zadania 3.3.2 118
6.3.3. Rozwiązanie zadania 3.3.3 120
6.3.4. Rozwiązanie zadania 3.3.4 122
6.3.5. Rozwiązanie zadania 3.3.5 125
6.4. Wybrane techniki **128**
6.4.1. Rozwiązanie zadania 4.3.1 128
6.4.2. Rozwiązanie zadania 4.3.2 131
6.4.3. Rozwiązanie zadania 4.3.3 132
6.4.4. Rozwiązanie zadania 4.3.4 134
6.4.5. Rozwiązanie zadania 4.3.5 136
6.5. Programowanie rozproszone **138**
6.5.1. Rozwiązanie zadania 5.4.1 138

6.5.2. Rozwiązanie zadania 5.4.2 140
6.5.3. Rozwiązanie zadania 5.4.3 140

6.1. Podstawowe pojęcia dotyczące współbieżności

6.1.1. Rozwiązanie zadania 1.4.1

Jeśli klasa wątku dziedziczy po klasie `Thread`, wówczas nazwę wątku uzyskamy wywołując bezpośrednio funkcję **getName()** z tej klasy. Do nadania nazwy wykorzystujemy jednoargumnetowy konstruktor klasy `Thread`.

Listing 6.1. Plik NazwanyWatek.java

```
1  public class NazwanyWatek extends Thread {
2
3    public NazwanyWatek(String nazwa) {
4      super(nazwa);
5    }
6
7    @Override
8    public void run() {
9      System.out.println("Wątek: " + getName());
10   }
11 }
```

Listing 6.2. Plik Main.java

```
1  class Main {
2
3    public static void main(String[] args) {
4      NazwanyWatek w1 =
5                new NazwanyWatek("Watek1");
6      w1.start();
7      NazwanyWatek w2 =
8                new NazwanyWatek("Watek1");
9      w2.start();
10     NazwanyWatek w3 =
11               new NazwanyWatek("Watek1");
12     w3.start();
13   }
14 }
```

6.1.2. Rozwiązanie zadania 1.4.2

Jeśli definiujemy klasę wątku jako rozszerzenie interfejsu `Runnable`, wówczas najpierw musimy uzyskać referencję do bieżącego wątku, a potem wywołać metodę **getName()** z klasy `Thread`. W analogiczny sposób usykamy nazwę wątku głównego.

Listing 6.3. Plik Zadanie.java

```
1 class Zadanie implements Runnable {
2
3   public void run() {
4     Thread currentThread = Thread.currentThread();
5     System.out.println("Wątek: "
6                         + currentThread.getName());
7   }
8 }
```

Listing 6.4. Plik Main.java

```
1 public class Main {
2
3   public static void main(String[] args) {
4
5     (new Thread(new Zadanie(), "Zadanie1")).start();
6     (new Thread(new Zadanie(), "Zadanie2")).start();
7     (new Thread(new Zadanie(), "Zadanie3")).start();
8
9     Thread currentThread = Thread.currentThread();
10    System.out.println("Wątek główny: "
11                        + currentThread.getName());
12  }
13 }
```

6.1.3. Rozwiązanie zadania 1.4.3

Listing 6.5. Plik Dekker.java

```
1 class Watek extends Thread {
2
3   private int numer;
4   private static volatile boolean chce1 = false;
5   private static volatile boolean chce2 = false;
6   private static volatile int kto_czeka = 1;
7
8   Watek(String nazwa) {
9     super(nazwa);
10  }
11
12  public int getNumer() {
13    return numer;
14  }
15
16  @Override
17  public void run() {
```

```
18
19      numer = ThreadID.get();
20
21      if (numer == 1) {
22        while (true) {
23          // sekcja lokalna
24          try {
25            Thread.sleep((long) (2500 * Math.random()));
26          } catch (InterruptedException e) {
27          }
28          // protokół wstępny
29          chce1 = true;
30          while (chce2) {
31            if (kto_czeka == 1) {
32              chce1 = false;
33              while (kto_czeka == 1) {
34                //nic nie rób
35              }
36              chce1 = true;
37            }
38          }
39          //sekcja krytyczna
40          System.out.println("Wątek " + numer + " start SK");
41          try {
42            Thread.sleep((long) (1000 * Math.random()));
43          } catch (InterruptedException e) {
44          }
45          System.out.println("Wątek " + numer + " stop SK");
46          // protokół końcowy
47          kto_czeka = 1;
48          chce1 = false;
49        }
50      }
51      if (numer == 2) {
52        while (true) {
53          // sekcja lokalna
54          try {
55            Thread.sleep((long) (2500 * Math.random()));
56          } catch (InterruptedException e) {
57          }
58          // protokół wstępny
59          chce2 = true;
60          while (chce1) {
61            if (kto_czeka == 2) {
62              chce2 = false;
63              while (kto_czeka == 2) {
64                //nic nie rób
65              }
66              chce2 = true;
67            }
68          }
```

```
69            //sekcja krytyczna
70            System.out.println("Wątek " + numer + " start SK");
71            try {
72              Thread.sleep((long) (1000 * Math.random()));
73            } catch (InterruptedException e) {
74            }
75            System.out.println("Wątek " + numer + " stop SK");
76            // protokół końcowy
77            kto_czeka = 2;
78            chce2 = false;
79          }
80        }
81      }
82  }
83
84  public class Dekker {
85    public static void main(String[] args)
86                                  throws InterruptedException {
87      System.out.println("Algorytm Dekkera - 2 wątki");
88      Watek t1 = new Watek("1");
89      Watek t2 = new Watek("2");
90      t1.start();
91      t2.start();
92    }
93  }
```

6.1.4. Rozwiązanie zadania 1.4.4

Listing 6.6. Plik DoranThomas.java

```
1  class Watek extends Thread {
2
3    private int numer;
4    private static volatile boolean chce1 = false;
5    private static volatile boolean chce2 = false;
6    private static volatile int kto_czeka = 1;
7
8    Watek(String nazwa) {
9      super(nazwa);
10   }
11
12   public int getNumer() {
13     return numer;
14   }
15
16   @Override
17   public void run() {
18
19     numer = ThreadID.get();
```

```
20
21      if (numer == 1) {
22        while (true) {
23          // sekcja lokalna
24          try {
25            Thread.sleep((long) (2500 * Math.random()));
26          } catch (InterruptedException e) {
27          }
28          // protokół wstępny
29          chce1 = true;
30
31          if (chce2) {
32            if (kto_czeka == 1) {
33              chce1 = false;
34              while (kto_czeka == 1) { //nic nie rób
35              }
36              chce1 = true;
37            }
38            while (chce2) { // nic nie rób
39            }
40          }
41          //sekcja krytyczna
42          System.out.println("Wątek " + numer + " start SK");
43          try {
44            Thread.sleep((long) (1000 * Math.random()));
45          } catch (InterruptedException e) {
46          }
47          System.out.println("Wątek " + numer + " stop SK");
48          // protokół końcowy
49          chce1 = false;
50          kto_czeka = 1;
51        }
52      }
53      if (numer == 2) {
54        while (true) {
55          // sekcja lokalna
56          try {
57            Thread.sleep((long) (2500 * Math.random()));
58          } catch (InterruptedException e) {
59          }
60          // protokół wstępny
61          chce2 = true;
62
63          if (chce1) {
64            if (kto_czeka == 2) {
65              chce2 = false;
66              while (kto_czeka == 2) { //nic nie rób
67              }
68              chce2 = true;
69            }
70            while (chce1) { // nic nie rób
```

```
71              }
72            }
73            //sekcja krytyczna
74            System.out.println("Wątek " + numer + " start SK");
75            try {
76              Thread.sleep((long) (1000 * Math.random()));
77            } catch (InterruptedException e) {
78            }
79            System.out.println("Wątek " + numer + " stop SK");
80            // protokół końcowy
81            chce2 = false;
82            kto_czeka = 2;
83          }
84        }
85      }
86  }
87
88  public class DoranThomas {
89    public static void main(String[] args)
90            throws InterruptedException {
91      System.out.println("Algorytm Dorana-Thomasa - 2 wątki");
92      Watek t1 = new Watek("1");
93      Watek t2 = new Watek("2");
94      t1.start();
95      t2.start();
96    }
97  }
```

6.1.5. Rozwiązanie zadania 1.4.5

Listing 6.7. Plik Peterson.java

```
1   class Watek extends Thread {
2
3     private int numer;
4     private static volatile boolean chce1 = false;
5     private static volatile boolean chce2 = false;
6     private static volatile int kto_czeka = 1;
7
8     Watek(String nazwa) {
9       super(nazwa);
10    }
11
12    @Override
13    public void run() {
14
15      numer = ThreadID.get();
16
17      if (numer == 1) {
```

```
18
19         while (true) {
20           // sekcja lokalna
21           try {
22             Thread.sleep((long) (2500 * Math.random()));
23           } catch (InterruptedException e) {
24           }
25           // protokół wstępny
26           chce1 = true;
27           kto_czeka = 1;
28
29           while (chce2 && (kto_czeka == 1)) {
30             //nic nie rób
31           }
32           //sekcja krytyczna
33           System.out.println("Wątek " + numer + " start SK");
34           try {
35             Thread.sleep((long) (1000 * Math.random()));
36           } catch (InterruptedException e) {
37           }
38           System.out.println("Wątek " + numer + " stop SK");
39           // protokół końcowy
40           chce1 = false;
41         }
42       }
43       if (numer == 2) {
44
45         while (true) {
46           // sekcja lokalna
47           try {
48             Thread.sleep((long) (2500 * Math.random()));
49           } catch (InterruptedException e) {
50           }
51           // protokół wstępny
52           chce2 = true;
53           kto_czeka = 2;
54
55           while (chce1 && (kto_czeka == 2)) {
56             //nic nie rób
57           }
58           //sekcja krytyczna
59           System.out.println("Wątek " + numer + " start SK");
60           try {
61             Thread.sleep((long) (1000 * Math.random()));
62           } catch (InterruptedException e) {
63           }
64           System.out.println("Wątek " + numer + " stop SK");
65           // protokół końcowy
66           chce2 = false;
67         }
68       }
```

```
69    }
70  }
71
72  public class Peterson {
73    public static void main(String[] args)
74            throws InterruptedException {
75      System.out.println("Algorytm␣Petersona␣−␣2␣wątki");
76      Watek t1 = new Watek("1");
77      Watek t2 = new Watek("2");
78      t1.start();
79      t2.start();
80    }
81  }
```

6.1.6. Rozwiązanie zadania 1.4.6

Listing 6.8. Plik PetersonNWatkow.java

```
1   class Watek extends Thread {
2
3     public static final int N = 5;
4     private int numer;
5     static AtomicIntegerArray chce
6                               = new AtomicIntegerArray(N + 1);
7     static AtomicIntegerArray kto_czeka
8                               = new AtomicIntegerArray(N);
9
10    static {
11      for (int i = 0; i <= N; i++) {
12        chce.set(i, 0);
13      }
14      for (int i = 0; i <= N − 1; i++) {
15        kto_czeka.set(i, 0);
16      }
17    }
18
19    Watek(String nazwa) {
20      super(nazwa);
21    }
22
23    @Override
24    public void run() {
25
26      numer = ThreadID.get();
27
28      while (true) {
29
30        // sekcja lokalna
31        try {
```

```
32            Thread.sleep((long) (2500 * Math.random()));
33          } catch (InterruptedException e) {
34          }
35
36          // protoków wstępny
37          for (int bariera = 1; bariera <= N - 1; bariera++) {
38            chce.set(numer, bariera);
39            kto_czeka.set(bariera, numer);
40
41            for (int j = 1; j <= N; j++) {
42              if (j != numer) {
43                while ((chce.get(j) >= bariera)
44                        && (kto_czeka.get(bariera) == numer)) {
45                }
46              }
47            }
48          }
49
50          //sekcja krytyczna
51          System.out.println("Wątek " + numer + " start SK");
52          try {
53            Thread.sleep((long) (1000 * Math.random()));
54          } catch (InterruptedException e) {
55          }
56          System.out.println("Wątek " + numer + " stop SK");
57
58          // protokół końcowy
59          chce.set(numer, 0);
60        }
61      }
62  }
63
64  public class PetersonNWatkow {
65
66    public static void main(String[] args)
67            throws InterruptedException {
68
69      System.out.println("Algorytm Petersona - "
70                            + Watek.N + " wątków");
71
72      Watek watki[] = new Watek[Watek.N];
73
74      for (int i = 0; i < Watek.N; i++) {
75        watki[i] = new Watek(""+ (i + 1));
76        watki[i].start();
77      }
78    }
79  }
```

6.1.7. Rozwiązanie zadania 1.4.7

Listing 6.9. Plik Main.java

```
class WatekA extends Thread {

  @Override
  public void run() {

    System.out.println("Instrukcje wątku A");

  }
}

class WatekB extends Thread {

  Thread poprzednik;

  public WatekB(Thread poprzednik) {
    this.poprzednik = poprzednik;
  }

  @Override
  public void run() {

    //sekcja niesynchronizowana
    try {
      Thread.sleep((long) (2500 * Math.random()));
    } catch (InterruptedException ex) {
    }

    try {
      poprzednik.join();
      System.out.println("Insrukcja B");
    } catch (InterruptedException ex) {
    }
  }
}

class WatekC extends Thread {

  Thread poprzednik;

  public WatekC(Thread poprzednik) {
    this.poprzednik = poprzednik;
  }

  @Override
  public void run() {

    //sekcja niesynchronizowana
```

```
48       try {
49         Thread.sleep((long) (5000 * Math.random()));
50       } catch (InterruptedException ex) {
51       }
52
53       try {
54         poprzednik.join();
55         System.out.println("Insrukcja C");
56       } catch (InterruptedException ex) {
57       }
58     }
59 }
60
61 public class Main {
62
63   public static void main(String[] args) {
64
65     WatekA a = new WatekA();
66     WatekB b = new WatekB(a);
67     WatekC c = new WatekC(b);
68
69     a.start();
70     b.start();
71     c.start();
72
73     try {
74       a.join();
75       b.join();
76       c.join();
77     } catch (InterruptedException ex) {
78     }
79       System.out.println("Instrukcja końcowa");
80   }
81 }
```

6.1.8. Rozwiązanie zadania 1.4.8

Listing 6.10. Plik Watek.java

```
1 class Watek extends Thread {
2
3   public Watek(String nazwa) {
4     super(nazwa);
5   }
6
7   @Override
8   public void run() {
9     try {
10      Thread.sleep((long) (10000 * Math.random()));
```

```
11      } catch (InterruptedException e) {
12      }
13      System.out.println("Komunikat z wątku "
14                                           + getName());
15    }
16
17    public static void main(String[] args)
18                      throws InterruptedException {
19
20      Watek[] tab = new Watek[10];
21
22      for (int i = 0; i < 10; ++i) {
23        tab[i] = new Watek(""+i);
24      }
25
26      for (int i = 0; i < 10; ++i) {
27        tab[i].start();
28      }
29
30      for (int i = 0; i < 10; ++i) {
31        System.out.println("Czekam na wątek "
32                                        + tab[i].getName());
33        tab[i].join();
34      }
35
36      System.out.println("Wszystkie wątki zakończyły się");
37    }
38  }
```

6.2. Semafory

6.2.1. Rozwiązanie zadania 2.4.1

Listing 6.11. Plik Filozof.java

```
1 import java.util.concurrent.Semaphore;
2
3 public class Filozof extends Thread {
4
5   static final int MAX = 5;
6   static Semaphore[] paleczka = new Semaphore[MAX];
7   static Semaphore jadalnia = new Semaphore(MAX - 1);
8   int mojNum;
9
10  public Filozof(int nr) {
11    mojNum = nr;
12  }
13
14  @Override
15  public void run() {
16
17    while (true) {
18
19      // myślenie
20      System.out.println("Myślę " + mojNum);
21      try {
22        Thread.sleep((long) (5000 * Math.random()));
23      } catch (InterruptedException e) {
24      }
25
26      jadalnia.acquireUninterruptibly();
27      paleczka[mojNum].acquireUninterruptibly();
28      paleczka[(mojNum + 1) % MAX].acquireUninterruptibly();
29
30      System.out.println("Zaczynam jeść " + mojNum);
31      try {
32        Thread.sleep((long) (3000 * Math.random()));
33      } catch (InterruptedException e) {
34      }
35      System.out.println("Kończę jeść " + mojNum);
36
37      paleczka[mojNum].release();
38      paleczka[(mojNum + 1) % MAX].release();
39
40      jadalnia.release();
41    }
42  }
43
44  public static void main(String[] args) {
```

```
45
46      for (int i = 0; i < MAX; i++) {
47        paleczka[i] = new Semaphore(1);
48      }
49
50      for (int i = 0; i < MAX; i++) {
51        (new Filozof(i)).start();
52      }
53
54    }
55  }
```

6.2.2. Rozwiązanie zadania 2.4.2

Listing 6.12. Plik Filozof.java

```
1  import java.util.concurrent.Semaphore;
2
3  public class Filozof extends Thread {
4
5    static final int MAX = 5;
6    static Semaphore[] paleczka = new Semaphore[MAX];
7    int mojNum;
8
9    public Filozof(int nr) {
10     mojNum = nr;
11   }
12
13   @Override
14   public void run() {
15
16     while (true) {
17
18       // myślenie
19       System.out.println("Myślę " + mojNum);
20       try {
21         Thread.sleep((long) (5000 * Math.random()));
22       } catch (InterruptedException e) {
23       }
24
25       if (mojNum == 0) {
26         paleczka[(mojNum+1)%MAX].acquireUninterruptibly();
27         paleczka[mojNum].acquireUninterruptibly();
28       } else {
29         paleczka[mojNum].acquireUninterruptibly();
30         paleczka[(mojNum+1)%MAX].acquireUninterruptibly();
31       }
32
33       System.out.println("Zaczynam jeść " + mojNum);
```

```
34        try {
35          Thread.sleep((long) (3000 * Math.random()));
36        } catch (InterruptedException e) {
37        }
38        System.out.println("Kończę jeść " + mojNum);
39
40        paleczka[mojNum].release();
41        paleczka[(mojNum + 1) % MAX].release();
42
43      }
44    }
45
46    public static void main(String[] args) {
47
48      for (int i = 0; i < MAX; i++) {
49        paleczka[i] = new Semaphore(1);
50      }
51
52      for (int i = 0; i < MAX; i++) {
53        (new Filozof(i)).start();
54      }
55
56    }
57  }
```

6.2.3. Rozwiązanie zadania 2.4.3

Listing 6.13. Plik Filozof.java

```
1  import java.util.Random;
2  import java.util.concurrent.Semaphore;
3
4  public class Filozof extends Thread {
5
6    static final int MAX = 5;
7    static Semaphore[] paleczka = new Semaphore[MAX];
8    int mojNum;
9    Random losuj;
10
11   public Filozof(int nr) {
12     mojNum = nr;
13     losuj = new Random(mojNum);
14   }
15
16   @Override
17   public void run() {
18
19     while (true) {
20
```

```
21        // myślenie
22        System.out.println("Myślę " + mojNum);
23        try {
24          Thread.sleep((long) (5000 * Math.random()));
25        } catch (InterruptedException e) {
26        }
27
28        int strona = losuj.nextInt(2);
29
30        boolean podnioslDwiePaleczki = false;
31
32        do {
33          if (strona == 0) {
34            paleczka[mojNum].acquireUninterruptibly();
35
36            if (!(paleczka[(mojNum+1)%MAX].tryAcquire())) {
37              paleczka[mojNum].release();
38            } else {
39              podnioslDwiePaleczki = true;
40            }
41
42          } else {
43            paleczka[(mojNum+1)%MAX].acquireUninterruptibly();
44
45            if (!(paleczka[mojNum].tryAcquire())) {
46              paleczka[(mojNum+1)%MAX].release();
47            } else {
48              podnioslDwiePaleczki = true;
49            }
50          }
51        } while (podnioslDwiePaleczki == false);
52
53        System.out.println("Zaczynam jeść " + mojNum);
54        try {
55          Thread.sleep((long) (3000 * Math.random()));
56        } catch (InterruptedException e) {
57        }
58        System.out.println("Kończę jeść " + mojNum);
59
60        paleczka[mojNum].release();
61        paleczka[(mojNum+1)%MAX].release();
62
63      }
64    }
65
66    public static void main(String[] args) {
67
68      for (int i = 0; i < MAX; i++) {
69        paleczka[i] = new Semaphore(1);
70      }
71
```

```
72        for (int i = 0; i < MAX; i++) {
73          (new Filozof(i)).start();
74        }
75
76      }
77  }
```

6.2.4. Rozwiązanie zadania 2.4.4

Listing 6.14. Plik PKbuf1el.java

```
 1 import java.util.concurrent.Semaphore;
 2
 3 public class PKbuf1el {
 4
 5   static volatile int buf;
 6   static Semaphore elem = new Semaphore(0);
 7   static Semaphore miej = new Semaphore(1);
 8
 9   static class Producent extends Thread {
10
11     private int mojNum;
12
13     public Producent(int nr) {
14       this.mojNum = nr;
15     }
16
17     @Override
18     public void run() {
19
20       while (true) {
21
22         try {
23           Thread.sleep((long) (2000 * Math.random()));
24         } catch (InterruptedException e) {
25         }
26
27         int x = (int) (100 * Math.random());
28
29         miej.acquireUninterruptibly();
30
31         buf = x;
32         System.out.println("Wątek " + mojNum
33                            + " wyprodukował " + x);
34
35         elem.release();
36       }
37     }
38   }
```

```
39
40     static class Konsument extends Thread {
41
42       private int mojNum;
43
44       public Konsument(int nr) {
45         this.mojNum = nr;
46       }
47
48       @Override
49       public void run() {
50
51         while (true) {
52
53           try {
54             Thread.sleep((long) (2000 * Math.random()));
55           } catch (InterruptedException e) {
56           }
57           int x;
58           elem.acquireUninterruptibly();
59
60           x = buf;
61           System.out.println("Wątek " + mojNum
62                              + " skonsumował " + x);
63
64           miej.release();
65
66         }
67       }
68     }
69
70     public static void main(String[] args) {
71
72       Thread p1 = new Producent(1);
73       Thread p2 = new Producent(2);
74       Thread p3 = new Producent(3);
75       Thread k1 = new Konsument(4);
76       Thread k2 = new Konsument(5);
77
78       p1.start();
79       p2.start();
80       p3.start();
81       k1.start();
82       k2.start();
83     }
84   }
```

6.2.5. Rozwiązanie zadania 2.4.5

Listing 6.15. Plik PKbufCykl.java

```
1
2 import java.util.concurrent.Semaphore;
3
4 public class PKbufCykl {
5
6   static final int MAX = 10;
7   static int[] buf = new int[MAX];
8   static Semaphore elem = new Semaphore(0);
9   static Semaphore miej = new Semaphore(MAX);
10  static Semaphore produkowanie = new Semaphore(1);
11  static Semaphore konsumowanie = new Semaphore(1);
12  static int weInd = 0;
13  static int wyInd = 0;
14
15  static class Producent extends Thread {
16
17    private int mojNum;
18
19    public Producent(int nr) {
20      this.mojNum = nr;
21    }
22
23    @Override
24    public void run() {
25
26      while (true) {
27
28        try {
29          Thread.sleep((long) (1000 * Math.random()));
30        } catch (InterruptedException e) {
31        }
32
33        produkowanie.acquireUninterruptibly();
34        int x = (int) (100 * Math.random());
35        miej.acquireUninterruptibly(); // miej.wait();
36        buf[weInd] = x;
37        weInd = (weInd + 1) % MAX;
38        System.out.println("Wątek␣" + mojNum
39                           + "␣wyprodukował␣" + x);
40        elem.release(); // elem.signal()
41        produkowanie.release();
42      }
43    }
44  }
45
46  static class Konsument extends Thread {
47
```

```
48      private int mojNum;
49
50      public Konsument(int nr) {
51        this.mojNum = nr;
52      }
53
54      @Override
55      public void run() {
56
57        while (true) {
58
59          try {
60            Thread.sleep((long) (2000 * Math.random()));
61          } catch (InterruptedException e) {
62          }
63
64          konsumowanie.acquireUninterruptibly();
65          int x;
66          elem.acquireUninterruptibly();
67          x = buf[wyInd];
68          wyInd = (wyInd + 1) % MAX;
69          System.out.println("Wątek " + mojNum
70                             + " skonsumował " + x);
71          miej.release();
72          konsumowanie.release();
73        }
74      }
75    }
76
77    public static void main(String[] args) {
78
79      Thread p1 = new Producent(1);
80      Thread p2 = new Producent(2);
81      Thread p3 = new Producent(3);
82      Thread k1 = new Konsument(4);
83      Thread k2 = new Konsument(5);
84
85      p1.start();
86      p2.start();
87      p3.start();
88      k1.start();
89      k2.start();
90
91    }
92  }
```

6.2.6. Rozwiązanie zadania 2.4.6

Listing 6.16. Plik Main.java

```
1
2 import java.util.concurrent.Semaphore;
3 import java.util.concurrent.atomic.AtomicIntegerArray;
4
5 public class Main extends Thread {
6
7   static final int rozmiar_b1 = 15;
8   static final int rozmiar_b2 = 5;
9   static AtomicIntegerArray b1
10                          = new AtomicIntegerArray(rozmiar_b1);
11  static AtomicIntegerArray b2
12                          = new AtomicIntegerArray(rozmiar_b2);
13  static Semaphore b1_jestMiejsce = new Semaphore(rozmiar_b1);
14  static Semaphore b1_wstawianie = new Semaphore(1);
15  static Semaphore b1_saElementy = new Semaphore(0);
16  static Semaphore b2_jestMiejsce = new Semaphore(rozmiar_b2);
17  static Semaphore b2_saElementy = new Semaphore(0);
18  static volatile int b1_weInd = 0;
19  static volatile int b1_wyInd = 0;
20  static volatile int b2_weInd = 0;
21
22  static class P extends Thread {
23
24    private int mojNum;
25
26    public P(int nr) {
27      this.mojNum = nr;
28    }
29
30    @Override
31    public void run() {
32      while (true) {
33
34        try {
35          Thread.sleep((long) (10000 * Math.random()));
36        } catch (InterruptedException e) {
37        }
38
39        b1_jestMiejsce.acquireUninterruptibly();
40        b1_wstawianie.acquireUninterruptibly();
41
42        int element = (int) (100 * Math.random());
43
44        b1.set(b1_weInd, element);
45
46        System.out.println("Wątek P" + mojNum + ": do b1["
47                + b1_weInd + "] wstawiam " + element);
```

```
48
49             b1_weInd = (b1_weInd + 1) % rozmiar_b1;
50
51             b1_wstawianie.release();
52             b1_saElementy.release();
53           }
54         }
55       }
56
57       static class S extends Thread {
58
59         private int przeksztalc(int element1, int element2) {
60           return element1 + element2;
61         }
62
63         @Override
64         public void run() {
65           while (true) {
66
67             try {
68               Thread.sleep((long) (10000 * Math.random()));
69             } catch (InterruptedException e) {
70             }
71
72             b1_saElementy.acquireUninterruptibly(2);
73
74             int e1 = b1.get(b1_wyInd);
75
76             System.out.println("Wątek S: z b1[" + b1_wyInd
77                     + "] pobieram " + e1);
78             b1_wyInd = (b1_wyInd + 1) % rozmiar_b1;
79
80             int e2 = b1.get(b1_wyInd);
81
82             System.out.println("Wątek S: z b1[" + b1_wyInd
83                     + "] pobieram " + e2);
84             b1_wyInd = (b1_wyInd + 1) % rozmiar_b1;
85
86             b1_jestMiejsce.release(2);
87
88             int wynik = przeksztalc(e1, e2);
89
90             b2_jestMiejsce.acquireUninterruptibly();
91
92             b2.set(b2_weInd, wynik);
93             System.out.println("Wątek S: do b2[" + b2_weInd
94                     + "] wstawiam " + wynik);
95             b2_weInd = (b2_weInd + 1) % rozmiar_b2;
96
97             b2_saElementy.release();
98
```

```
 99          }
100        }
101      }
102
103      static class K extends Thread {
104
105        @Override
106        public void run() {
107          while (true) {
108
109            try {
110              Thread.sleep((long) (10000 * Math.random()));
111            } catch (InterruptedException e) {
112            }
113
114            b2_saElementy.acquireUninterruptibly(rozmiar_b2);
115
116            for (int i = 0; i < rozmiar_b2; ++i) {
117              System.out.println("Wątek K: z b2[" + i
118                      + "] pobieram " + b2.get(i));
119            }
120            b2_jestMiejsce.release(rozmiar_b2);
121          }
122        }
123      }
124
125      public static void main(String[] args) {
126
127        for (int i = 0; i < 10; i++) {
128          (new P(i)).start();
129        }
130        (new S()).start();
131        (new K()).start();
132      }
133    }
```

6.2.7. Rozwiązanie zadania 2.4.7

Listing 6.17. Plik Main.java

```
1 import java.util.concurrent.Semaphore;
2
3 public class Main {
4
5   static final int LICZ_PISA = 2;
6   static final int LICZ_CZYT = 9;
7   static Semaphore sprawdzam = new Semaphore(1);
8   static Semaphore pisanie = new Semaphore(1);
9   static volatile int liczbaCzytelnikow = 0;
```

```
10
11    static class Czytelnik extends Thread {
12
13      private int mojNum;
14
15      public Czytelnik(int nr) {
16        this.mojNum = nr;
17      }
18
19      @Override
20      public void run() {
21
22        while (true) {
23
24          try {
25            Thread.sleep((long) (5000 * Math.random()));
26          } catch (InterruptedException e) {
27          }
28
29
30          sprawdzam.acquireUninterruptibly();
31          liczbaCzytelnikow = liczbaCzytelnikow + 1;
32          if (liczbaCzytelnikow == 1) {
33            pisanie.acquireUninterruptibly();
34          }
35          sprawdzam.release();
36
37          System.out.println("Start CZYTANIA " + mojNum);
38
39          try {
40            Thread.sleep((long) (7000 * Math.random()));
41          } catch (InterruptedException e) {
42          }
43
44          System.out.println("Stop CZYTANIA " + mojNum);
45
46          sprawdzam.acquireUninterruptibly();
47          liczbaCzytelnikow = liczbaCzytelnikow - 1;
48          if (liczbaCzytelnikow == 0) {
49            pisanie.release();
50          }
51          sprawdzam.release();
52
53        }
54      }
55    }
56
57    static class Pisarz extends Thread {
58
59      private int mojNum;
60
```

```
61      public Pisarz(int nr) {
62        this.mojNum = nr;
63      }
64
65      @Override
66      public void run() {
67
68        while (true) {
69
70          try {
71            Thread.sleep((long) (5000 * Math.random()));
72          } catch (InterruptedException e) {
73          }
74
75          pisanie.acquireUninterruptibly();
76
77          System.out.println("Start PISANIA " + mojNum);
78
79          try {
80            Thread.sleep((long) (2000 * Math.random()));
81          } catch (InterruptedException e) {
82          }
83
84          System.out.println("Stop PISANIA " + mojNum);
85
86          pisanie.release();
87
88        }
89      }
90    }
91
92    public static void main(String[] args) {
93
94      for (int i = 0; i < LICZ_CZYT; i++) {
95        (new Czytelnik(i)).start();
96      }
97
98      for (int i = 0; i < LICZ_PISA; i++) {
99        (new Pisarz(i)).start();
100     }
101   }
102 }
```

6.2.8. Rozwiązanie zadania 2.4.8

Listing 6.18. Plik Main.java

```java
import java.util.concurrent.Semaphore;

public class Main {

  static final int LICZ_PISA = 2;
  static final int LICZ_CZYT = 9;
  static Semaphore czytelnik_sprawdzam = new Semaphore(1);
  static Semaphore pisarz_sprawdzam = new Semaphore(1);
  static Semaphore chcePisac = new Semaphore(1);
  static Semaphore pisanie = new Semaphore(1);
  static volatile int liczbaCzytelnikow = 0;
  static volatile int liczbaPisarzy = 0;

  static class Czytelnik extends Thread {

    private int mojNum;

    public Czytelnik(int nr) {
      this.mojNum = nr;
    }

    @Override
    public void run() {

      while (true) {

        try {
          Thread.sleep((long) (5000 * Math.random()));
        } catch (InterruptedException e) {
        }

        chcePisac.acquireUninterruptibly();
        czytelnik_sprawdzam.acquireUninterruptibly();
        liczbaCzytelnikow = liczbaCzytelnikow + 1;
        if (liczbaCzytelnikow == 1) {
          pisanie.acquireUninterruptibly();
        }
        czytelnik_sprawdzam.release();
        chcePisac.release();

        System.out.println("Start CZYTANIA " + mojNum);

        try {
          Thread.sleep((long) (7000 * Math.random()));
        } catch (InterruptedException e) {
        }

```

```
48          System.out.println("Stop CZYTANIA " + mojNum);
49
50          czytelnik_sprawdzam.acquireUninterruptibly();
51          liczbaCzytelnikow = liczbaCzytelnikow - 1;
52          if (liczbaCzytelnikow == 0) {
53            pisanie.release();
54          }
55          czytelnik_sprawdzam.release();
56
57        }
58      }
59    }
60
61    static class Pisarz extends Thread {
62
63      private int mojNum;
64
65      public Pisarz(int nr) {
66        this.mojNum = nr;
67      }
68
69      @Override
70      public void run() {
71
72        while (true) {
73
74          try {
75            Thread.sleep((long) (5000 * Math.random()));
76          } catch (InterruptedException e) {
77          }
78
79          pisarz_sprawdzam.acquireUninterruptibly();
80          liczbaPisarzy = liczbaPisarzy + 1;
81          if (liczbaPisarzy == 1) {
82            chcePisac.acquireUninterruptibly();
83          }
84          pisarz_sprawdzam.release();
85          pisanie.acquireUninterruptibly();
86
87          System.out.println("Start PISANIA " + mojNum);
88
89          try {
90            Thread.sleep((long) (2000 * Math.random()));
91          } catch (InterruptedException e) {
92          }
93
94          System.out.println("Stop PISANIA " + mojNum);
95
96          pisanie.release();
97          pisarz_sprawdzam.acquireUninterruptibly();
98          liczbaPisarzy = liczbaPisarzy - 1;
```

```
99          if (liczbaPisarzy == 0) {
100           chcePisac.release();
101         }
102         pisarz_sprawdzam.release();
103
104       }
105     }
106   }
107
108   public static void main(String[] args) {
109
110     for (int i = 0; i < LICZ_CZYT; i++) {
111       (new Czytelnik(i)).start();
112     }
113
114     for (int i = 0; i < LICZ_PISA; i++) {
115       (new Pisarz(i)).start();
116     }
117   }
118 }
```

6.2.9. Rozwiązanie zadania 2.4.9

Listing 6.19. Plik SpiacyGolibroda.java

```
1 import java.util.concurrent.Semaphore;
2
3 public class SpiacyGolibroda {
4
5   static final int LICZ_KLIENTOW = 100;
6   static Semaphore klientJestGotowy = new Semaphore(0);
7   static Semaphore golibrodaJestGotowy = new Semaphore(0);
8   static Semaphore sprawdzamPoczekalnie = new Semaphore(1);
9   static volatile int liczbaWolnychMiejscWPoczekalni = 3;
10
11  static class Golibroda extends Thread {
12
13    @Override
14    public void run() {
15
16      while (true) {
17
18        klientJestGotowy.acquireUninterruptibly();
19
20        sprawdzamPoczekalnie.acquireUninterruptibly();
21        liczbaWolnychMiejscWPoczekalni += 1;
22        golibrodaJestGotowy.release();
23        sprawdzamPoczekalnie.release();
24
```

```
25          System.out.println("Golibroda strzyże.");
26          try {
27            Thread.sleep((long) (500 * Math.random()));
28          } catch (InterruptedException e) {
29          }
30        }
31      }
32    }
33
34    static class Klient extends Thread {
35
36      private int mojNum;
37
38      public Klient(int nr) {
39        this.mojNum = nr;
40      }
41
42      @Override
43      public void run() {
44
45        while (true) {
46
47          try {
48            Thread.sleep((long) (50000 * Math.random()));
49          } catch (InterruptedException e) {
50          }
51
52          sprawdzamPoczekalnie.acquireUninterruptibly();
53          System.out.println("Liczba wolnych miejsc w poczekalni: "
54                  + liczbaWolnychMiejscWPoczekalni);
55
56          if (liczbaWolnychMiejscWPoczekalni > 0) {
57
58            liczbaWolnychMiejscWPoczekalni -= 1;
59            klientJestGotowy.release();
60            sprawdzamPoczekalnie.release();
61            golibrodaJestGotowy.acquireUninterruptibly();
62
63            System.out.println("Klient "+mojNum+" jest strzyżony");
64
65          } else {
66            sprawdzamPoczekalnie.release();
67            System.out.println("Klient "+mojNum+" rezygnuje");
68          }
69        }
70      }
71    }
72
73    public static void main(String[] args) {
74
75      (new Golibroda()).start();
```

```
76
77        for (int i = 0; i < LICZ_KLIENTOW; i++) {
78            (new Klient(i)).start();
79        }
80    }
81 }
```

6.3. Monitory

6.3.1. Rozwiązanie zadania 3.3.1

Listing 6.20. Plik Bariera.java

```
1 package monitorbariera;
2
3 public class Bariera {
4
5     private int N;
6     private volatile int licznik;
7
8     public Bariera(int N) {
9         this.N = N;
10        this.licznik = 0;
11    }
12
13    public synchronized void bariera() {
14        licznik++;
15        if (licznik < N) {
16            try {
17                wait();
18            } catch (InterruptedException ex) {
19            }
20        } else {
21            licznik = 0;
22            notifyAll();
23        }
24    }
25 }
```

Listing 6.21. Plik Watek.java

```
1 package monitorbariera;
2
3 import java.util.Random;
4
5 public class Watek extends Thread {
6
7     private int id;
8     private Bariera2 bariera;
9     private static Random r = new Random();
10
11    public Watek(int id, Bariera2 bariera) {
12        this.id = id;
13        this.bariera = bariera;
14    }
15
```

```
16      @Override
17      public void run() {
18          System.out.println("Wątek " + id
19                                          + " przed barierą.");
20          try {
21              sleep(r.nextInt(3000));
22          } catch (InterruptedException ex) {
23          }
24
25          bariera.bariera();
26
27          System.out.println("Wątek " + id
28                                          + " po barierze.");
29      }
30  }
```

Listing 6.22. Plik Main.java

```
1  package monitorbariera;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          int N = 4;
8
9          Watek watki[] = new Watek[N];
10         Bariera2 b = new Bariera2(N);
11
12         for(int i=0; i<N; i++){
13             watki[i] = new Watek(i, b);
14             watki[i].start();
15         }
16     }
17 }
```

Klasa Bariera może również wyglądać tak.

Listing 6.23. Plik Bariera.java

```
1  package monitorbariera;
2
3  public class Bariera{
4      private volatile int N;
5
6      public Bariera(int N){
7          this.N = N;
8      }
9
10     public synchronized void bariera(){
```

```
11          N--;
12          if(N>0){
13              try {
14                  wait();
15              } catch (InterruptedException ex) {
16              }
17          }
18          else{
19              notifyAll();
20          }
21          N++;
22      }
23  }
```

Jeśli chcielibyśmy móc wywoływać metodę bariera() bez tworzenia obiektu typu Bariera, wówczas metoda bariera() może zostać zaimplementowana na przykład jako statyczna metoda klasy wątku.

Listing 6.24. Plik Watek.java

```
1  package monitorbarierastatyczna;
2
3  import java.util.Random;
4
5  public class Watek extends Thread {
6
7      private int id;
8      private static volatile int N = 0;
9
10     private static Random r = new Random();
11
12     public Watek() {
13         this.id = N;
14         this.N++;
15     }
16
17     public static synchronized void bariera() {
18         N--;
19         if(N>0){
20             try {
21                 Watek.class.wait();
22             } catch (InterruptedException ex) {
23             }
24         }
25         else{
26             Watek.class.notifyAll();
27         }
28         N++;
29     }
30
```

```
31    @Override
32    public void run() {
33        System.out.println("Wątek " + id + " przed barierą.");
34        try {
35            sleep(r.nextInt(3000));
36        } catch (InterruptedException ex) {
37        }
38
39        bariera();
40
41        System.out.println("Wątek " + id + " po barierze.");
42    }
43 }
```

Listing 6.25. Plik Main.java

```
1  package monitorbarierastatyczna;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          int N = 4;
8
9          Watek watki[] = new Watek[N];
10
11         for(int i=0; i<N; i++){
12             watki[i] = new Watek();
13             watki[i].start();
14         }
15     }
16 }
```

6.3.2. Rozwiązanie zadania 3.3.2

Listing 6.26. Plik Semafor.java

```
1  public class Semafor {
2      private int wartosc;
3
4      public Semafor(int wartosc){
5          this.wartosc = wartosc;
6      }
7
8      public synchronized void P(){
9          if(wartosc==0){
10             try {
11                 wait();
```

```java
            } catch (InterruptedException ex) {
            }
        }
        wartosc--;
    }

    public synchronized void V(){
        wartosc++;
        notify();
    }
}
```

Listing 6.27. Plik Watek.java

```java
import java.util.Random;

public class Watek extends Thread {

    private int id;
    private static Semafor semafor = new Semafor(2);
    private Random r = new Random();

    public Watek(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        try {
            sleep(r.nextInt(5000));
        } catch (InterruptedException ex) {
        }

        semafor.P();
        System.out.println("Wątek " + id + " wchodzi.");
        try {
            sleep(r.nextInt(7000));
        } catch (InterruptedException ex) {
        }
        System.out.println("Wątek " + id + " wychodzi.");
        semafor.V();
    }
}
```

Listing 6.28. Plik Main.java

```java
public class Main {

    public static void main(String[] args) {
```

```
4
5           int N = 5;
6
7           Watek watki[] = new Watek[5];
8
9           for (int i = 0; i < N; i++) {
10              watki[i] = new Watek(i);
11              watki[i].start();
12          }
13      }
14  }
```

6.3.3. Rozwiązanie zadania 3.3.3

Listing 6.29. Plik SemaforBinarny.java

```
1  public class SemaforBinarny {
2
3      private boolean czyZajety;
4
5      public SemaforBinarny(){
6          this.czyZajety = true;
7      }
8
9      public synchronized void P(){
10         if(czyZajety==false){
11             try {
12                 wait();
13             } catch (InterruptedException ex) {
14             }
15         }
16         czyZajety = false;
17     }
18
19     public synchronized void V(){
20         czyZajety = true;
21         notify();
22     }
23 }
```

Listing 6.30. Plik Watek.java

```
1  import java.util.Random;
2
3  public class Watek extends Thread {
4
5      private int id;
6      private static SemaforBinarny semafor
```

```
7                                       = new SemaforBinarny();
8       private Random r = new Random();
9
10      public Watek(int id) {
11          this.id = id;
12      }
13
14      @Override
15      public void run() {
16          try {
17              sleep(r.nextInt(5000));
18          } catch (InterruptedException ex) {
19          }
20
21          semafor.P();
22          System.out.println("Wątek␣" + id + "␣wchodzi.");
23          try {
24              sleep(r.nextInt(7000));
25          } catch (InterruptedException ex) {
26          }
27          System.out.println("Wątek␣" + id + "␣wychodzi.");
28          semafor.V();
29      }
30  }
```

Listing 6.31. Plik Main.java

```
1   public class Main {
2
3       public static void main(String[] args) {
4
5           int N = 5;
6
7           Watek watki[] = new Watek[5];
8
9           for (int i = 0; i < N; i++) {
10              watki[i] = new Watek(i);
11              watki[i].start();
12          }
13      }
14  }
```

6.3.4. Rozwiązanie zadania 3.3.4

Listing 6.32. Plik Main.java

```
1
2  class Synchronizator {
3
4    private volatile int licznik;
5
6    public Synchronizator(int licznik) {
7      this.licznik = licznik;
8    }
9
10   public synchronized void synchAwe()
11           throws InterruptedException {
12     while (licznik == 0 || licznik == 1) {
13       wait();
14     }
15   }
16
17   public synchronized void synchAwy()
18           throws InterruptedException {
19     if (licznik == 2) {
20       licznik = 0;
21     } else {
22       licznik = 1;
23     }
24     notifyAll();
25   }
26
27   public synchronized void synchBwe()
28           throws InterruptedException {
29     while (licznik == 0 || licznik == 2) {
30       wait();
31     }
32   }
33
34   public synchronized void synchBwy()
35           throws InterruptedException {
36     if (licznik == 1) {
37       licznik = 0;
38     } else {
39       licznik = 2;
40     }
41     notifyAll();
42   }
43
44   public synchronized void synchCwe()
45           throws InterruptedException {
46     while (licznik != 0) {
47       wait();
```

```
48      }
49    }
50
51    public synchronized void synchCwy()
52            throws InterruptedException {
53      licznik = 3;
54      notifyAll();
55    }
56  }
57
58  class A extends Thread {
59
60    Synchronizator s;
61
62    public A(Synchronizator s) {
63      this.s = s;
64    }
65
66    @Override
67    public void run() {
68      while(true){
69        try {
70          s.synchAwe();
71        } catch (InterruptedException ex) {
72        }
73        System.out.println("Instrukcja A");
74        try {
75          s.synchAwy();
76        } catch (InterruptedException ex) {
77        }
78      }
79    }
80  }
81
82  class B extends Thread {
83
84    Synchronizator s;
85
86    public B(Synchronizator s) {
87      this.s = s;
88    }
89
90    @Override
91    public void run() {
92      while(true){
93        try {
94          s.synchBwe();
95        } catch (InterruptedException ex) {
96        }
97        System.out.println("Instrukcja B");
98        try {
```

```
99          s.synchBwy();
100       } catch (InterruptedException ex) {
101       }
102     }
103   }
104 }
105
106 class C extends Thread {
107
108   Synchronizator s;
109
110   public C(Synchronizator s) {
111     this.s = s;
112   }
113
114   @Override
115   public void run() {
116     while(true){
117       try {
118         s.synchCwe();
119       } catch (InterruptedException ex) {
120       }
121       System.out.println("Instrukcja C");
122       try {
123         s.synchCwy();
124       } catch (InterruptedException ex) {
125       }
126     }
127   }
128 }
129
130 public class Main {
131
132   public static void main(String[] args) {
133
134     Synchronizator s = new Synchronizator(3);
135
136     Thread a = new A(s);
137     Thread b = new B(s);
138     Thread c = new C(s);
139
140     a.start();
141     b.start();
142     c.start();
143   }
144 }
```

6.3.5. Rozwiązanie zadania 3.3.5

Podobnie jak w rozwiązaniu problemu czytelników i pisarzy (patrz strona 47) do rozwiązania została wykorzystana dodatkowa klasa – klasa monitora *PrzydzialWidelcow* [4], pełniąca rolę arbitra w dostępie do widelców.

Listing 6.33. Plik PrzydzialWidelcow.java

```
import java.util.concurrent.atomic.AtomicIntegerArray;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PrzydzialWidelcow {

  private final Lock lock = new ReentrantLock(true);
  private final Condition moznaJesc[] = new Condition[5];
  private final AtomicIntegerArray widelec
                                  = new AtomicIntegerArray(5);

  public PrzydzialWidelcow() {
    for (int i = 0; i < 5; i++) {
      moznaJesc[i] = lock.newCondition();
    }
    for (int i = 0; i < 5; i++) {
      widelec.set(i, 2);
    }
  }

  public void podniesWidelce(int i)
                                throws InterruptedException {
    lock.lock();
    try {
      if (widelec.get(i) < 2) {
        moznaJesc[i].await();
      }
      widelec.getAndDecrement((i + 1) % 5);
      widelec.getAndDecrement((i + 4) % 5);
    } finally {
      lock.unlock();
    }
  }

  public void polozWidelce(int i) {
    lock.lock();
    try {
      widelec.getAndIncrement((i + 1) % 5);
      widelec.getAndIncrement((i + 4) % 5);
      if (widelec.get((i + 1) % 5) == 2) {
        moznaJesc[(i + 1) % 5].signal();
      }
```

```
44         if (widelec.get((i + 4) % 5) == 2) {
45           moznaJesc[(i + 4) % 5].signal();
46         }
47       } finally {
48         lock.unlock();
49       }
50     }
51 }
```

Listing 6.34. Plik Filozof.java

```
1  public class Filozof extends Thread {
2
3    private int mojNum;
4    private PrzydzialWidelcow przyWidelcow;
5
6    public Filozof(int nr, PrzydzialWidelcow przyWidelcow) {
7      mojNum = nr;
8      this.przyWidelcow = przyWidelcow;
9    }
10
11   @Override
12   public void run() {
13
14     while (true) {
15       try {
16
17         System.out.println("Myślę " + mojNum);
18         try {
19           Thread.sleep((long) (5000 * Math.random()));
20         } catch (InterruptedException e) {
21         }
22
23         przyWidelcow.podniesWidelce(mojNum);
24
25         System.out.println("Zaczynam jeść " + mojNum);
26         try {
27           Thread.sleep((long) (3000 * Math.random()));
28         } catch (InterruptedException e) {
29         }
30         System.out.println("Kończę jeść " + mojNum);
31
32         przyWidelcow.polozWidelce(mojNum);
33
34       } catch (InterruptedException ex) {
35       }
36     }
37   }
38 }
```

Listing 6.35. Plik Main.java

```
1 public class Main {
2
3   public static void main(String[] args) {
4
5     PrzydzialWidelcow przyWidelcow = new PrzydzialWidelcow();
6
7     for (int i = 0; i < 5; i++) {
8       (new Filozof(i, przyWidelcow)).start();
9     }
10  }
11 }
```

6.4. Wybrane techniki

6.4.1. Rozwiązanie zadania 4.3.1

Listing 6.36. Plik Watek.java

```
public class Watek extends Thread {

  private Lock lock;

  public Watek(String num, Lock lock) {
    super(num);
    this.lock = lock;
  }

  @Override
  public void run() {

    while (true) {

      // sekcja lokalna
      try {
        Thread.sleep((long) (2500 * Math.random()));
      } catch (InterruptedException e) {
      }

      // protokół wstępny
      lock.lock();

      //sekcja krytyczna
      System.out.println("Wątek "
                         + ThreadID.get() + " start SK");
      try {
        Thread.sleep((long) (1000 * Math.random()));
      } catch (InterruptedException e) {
      }
      System.out.println("Wątek "
                         + ThreadID.get() + " stop SK");

      // protokół końcowy
      lock.unLock();
    }
  }
}
```

Listing 6.37. Plik Lock.java

```
1 public interface Lock {
2
3     public void lock();
4
5     public void unLock();
6 }
```

Listing 6.38. Plik Piekarnia.java

```
1 import java.util.concurrent.atomic.AtomicIntegerArray;
2
3 class Piekarnia implements Lock {
4
5   private static AtomicIntegerArray flaga;
6   private static AtomicIntegerArray etykieta;
7
8   public Piekarnia(int N) {
9     etykieta = new AtomicIntegerArray(N);
10    for (int i = 0; i < N; i++) {
11      etykieta.set(i, 0);
12    }
13    flaga = new AtomicIntegerArray(N);
14    for (int i = 0; i < N; i++) {
15      flaga.set(i, 0);
16    }
17  }
18
19  public void lock() {
20    int mojNum = ThreadID.get();
21    flaga.set(mojNum, 1);
22    int maks = Etykieta.maks(etykieta);
23    etykieta.set(mojNum, maks + 1);
24    while (nieMojaKolej(mojNum)) {
25    }
26  }
27
28  public void unLock() {
29    flaga.set(ThreadID.get(), 0);
30  }
31
32  private boolean nieMojaKolej(int mojNum) {
33    for (int k = 0; k < etykieta.length(); k++) {
34      if (k != mojNum && (flaga.get(k) == 1)
35              && Etykieta.jestPrzed(k, mojNum) == 1) {
36        return true;
37      }
38    }
39    return false;
40  }
```

```
41
42   static class Etykieta {
43
44     static int maks(AtomicIntegerArray label) {
45       int c = 0;
46       for (int i = 0; i < label.length(); i++) {
47         c = Math.max(c, label.get(i));
48       }
49       return c;
50     }
51
52     static int jestPrzed(int t1, int t2) {
53       if (etykieta.get(t1) < etykieta.get(t2)
54          || (etykieta.get(t1) == etykieta.get(t2)
55                        && t1 < t2))
56       {
57         return 1;
58       } else {
59         return 0;
60       }
61     }
62   }
63 }
```

Listing 6.39. Plik AlgorytmPiekarniany.java

```
1 public class AlgorytmPiekarniany {
2
3     public static void main(String[] args)
4                             throws InterruptedException{
5
6        final int N = 5;
7
8         Lock blokada = new Piekarnia(N);
9
10        Thread watki[] = new Watek[N];
11
12        for (int i = 0; i < N; i++) {
13            watki[i] = new Watek(Integer.toString(i),
14                                                blokada);
15            watki[i].start();
16        }
17    }
18 }
```

6.4.2. Rozwiązanie zadania 4.3.2

Listing 6.40. Plik Watek.java

```
import java.util.concurrent.locks.Lock;

public class Watek extends Thread {

  private Lock lock;

  public Watek(String num, Lock lock) {
    super(num);
    this.lock = lock;
  }

  @Override
  public void run() {
    System.out.println("Sekcja lokalna " + ThreadID.get());

    while (true) {
      lock.lock();
      try {
        System.out.println("Początek sekcji krytycznej "
                + ThreadID.get());
        Thread.sleep((int) (3000 * Math.random()));
      } catch (InterruptedException e) {
      } finally {
        System.out.println("Koniec sekcji krytycznej "
                + ThreadID.get());
        lock.unlock();
      }
    }
  }
}
```

Listing 6.41. Plik Main.java

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Main {

  public static void main(String[] args)
          throws InterruptedException {

    final int N = 5;

    Lock lock = new ReentrantLock();

    Thread watki[] = new Watek[N];
```

```
14
15     for (int i = 0; i < N; i++) {
16       watki[i] = new Watek(Integer.toString(i + 1), lock);
17       watki[i].start();
18     }
19   }
20 }
```

6.4.3. Rozwiązanie zadania 4.3.3

Listing 6.42. Plik Main.java

```
1 import java.util.concurrent.locks.ReadWriteLock;
2 import java.util.concurrent.locks.ReentrantReadWriteLock;
3
4 public class Main {
5
6   static class Czytelnik extends Thread {
7
8     private ReadWriteLock b;
9     private int nr;
10
11     public Czytelnik(ReadWriteLock b, int nr) {
12       this.b = b;
13       this.nr = nr;
14     }
15
16     @Override
17     public void run() {
18
19       while (true) {
20
21         try {
22           Thread.sleep((long) (5000 * Math.random()));
23         } catch (InterruptedException e) {
24         }
25
26         b.readLock().lock();
27         System.out.println("Czytelnik " + nr + " czyta");
28
29         try {
30           Thread.sleep((long) (2000 * Math.random()));
31         } catch (InterruptedException e) {
32         }
33
34         System.out.println("Czytelnik " + nr + " przeczytał");
35         b.readLock().unlock();
36       }
37     }
```

```
38    }
39
40    static class Pisarz extends Thread {
41
42      private ReadWriteLock b;
43      private int nr;
44
45      public Pisarz(ReadWriteLock b, int nr) {
46        this.b = b;
47        this.nr = nr;
48      }
49
50      @Override
51      public void run() {
52
53        while (true) {
54
55          try {
56            Thread.sleep((long) (5000 * Math.random()));
57          } catch (InterruptedException e) {
58          }
59
60          b.writeLock().lock();
61          System.out.println("Pisarz " + nr + " pisze");
62
63          try {
64            Thread.sleep((long) (2000 * Math.random()));
65          } catch (InterruptedException e) {
66          }
67
68          System.out.println("Pisarz " + nr + " napisał");
69          b.writeLock().unlock();
70        }
71      }
72    }
73
74    public static void main(String[] args) {
75
76      ReadWriteLock blokada = new ReentrantReadWriteLock();
77
78      for (int i = 0; i < 5; i++) {
79        (new Czytelnik(blokada, i)).start();
80      }
81
82      for (int i = 0; i < 2; i++) {
83        (new Pisarz(blokada, i)).start();
84      }
85    }
86  }
```

6.4.4. Rozwiązanie zadania 4.3.4

Listing 6.43. Plik Filozof.java

```
1 import java.util.concurrent.TimeUnit;
2 import java.util.concurrent.locks.Lock;
3 import java.util.concurrent.locks.ReentrantLock;
4
5 public class Filozof extends Thread {
6
7   static final int MAX = 5;
8   static Lock[] paleczka = new ReentrantLock[MAX];
9   int liczbaPozostalychProbJedz = 3;
10  int mojNum;
11  int czasMyslenia;
12
13  public Filozof(int nr) {
14    mojNum = nr;
15    this.czasMyslenia = (int) (5000 * Math.random());
16  }
17
18  private void mysl() {
19
20    System.out.println("Myślę " + mojNum);
21    try {
22      Thread.sleep(czasMyslenia);
23    } catch (InterruptedException e) {
24    }
25  }
26
27  private void jedz() throws InterruptedException {
28
29    boolean probuje = true;
30    int liczbaProbWezPaleczki = 0;
31
32    while (probuje) {
33
34      if (paleczka[mojNum].tryLock(100,
35                                   TimeUnit.MILLISECONDS)){
36        if (paleczka[(mojNum + 1) % MAX].tryLock(50,
37                                                  TimeUnit.MILLISECONDS)){
38          System.out.println("Zaczynam jeść " + mojNum);
39          try {
40            Thread.sleep((long) (3000 * Math.random()));
41          } catch (InterruptedException e) {
42          }
43          System.out.println("Kończę jeść " + mojNum);
44
45          paleczka[mojNum].unlock();
46          paleczka[(mojNum + 1) % MAX].unlock();
47
```

```
48             probuje = false;
49             liczbaPozostalychProbJedz = 3;
50             czasMyslenia = (int) (5000 * Math.random());
51           } else {
52             paleczka[mojNum].unlock();
53             liczbaProbWezPaleczki++;
54           }
55         } else {
56           liczbaProbWezPaleczki++;
57         }
58 
59         if (liczbaProbWezPaleczki == 5) {
60           probuje = false;
61           liczbaPozostalychProbJedz--;
62           czasMyslenia = czasMyslenia / 2;
63         }
64       }
65     }
66 
67     @Override
68     public void run() {
69 
70       try{
71       while (liczbaPozostalychProbJedz > 0) {
72         mysl();
73         jedz();
74       }
75       System.out.println("Umieram z głodu " + mojNum);
76 
77       }catch(InterruptedException e) {
78         return;
79       }
80     }
81 
82     public static void main(String[] args) {
83 
84       for (int i = 0; i < MAX; i++) {
85         paleczka[i] = new ReentrantLock();
86       }
87 
88       for (int i = 0; i < MAX; i++) {
89         (new Filozof(i)).start();
90       }
91     }
92 }
```

6.4.5. Rozwiązanie zadania 4.3.5

Listing 6.44. Plik Main.java

```
1 import java.util.concurrent.ArrayBlockingQueue;
2 import java.util.concurrent.BlockingQueue;
3
4 public class Main {
5
6   static class Producent extends Thread {
7
8     private BlockingQueue bufor;
9     private int mojNum;
10
11    public Producent(int nr, BlockingQueue bufor) {
12      this.mojNum = nr;
13      this.bufor = bufor;
14    }
15
16    @Override
17    public void run() {
18      while (true) {
19        try {
20          Thread.sleep((long) (5000 * Math.random()));
21        } catch (InterruptedException e) {
22        }
23
24        Integer el = (int) (100 * Math.random());
25
26        try {
27          bufor.put(el);
28        } catch (InterruptedException ex) {
29        }
30
31        System.out.println("Producent " + mojNum
32                           + " wstawił element " + el);
33      }
34    }
35  }
36
37  static class Konsument extends Thread {
38
39    private BlockingQueue bufor;
40    private int mojNum;
41
42    public Konsument(int nr, BlockingQueue bufor) {
43      this.mojNum = nr;
44      this.bufor = bufor;
45    }
46
47    @Override
```

```
48      public void run() {
49        while (true) {
50          try {
51            Thread.sleep((long) (5000 * Math.random()));
52          } catch (InterruptedException e) {
53          }
54
55          Integer el = null;
56          try {
57            el = (Integer) bufor.take();
58          } catch (InterruptedException ex) {
59          }
60
61          System.out.println("Konsument " + mojNum
62                             + " pobrał element " + el);
63        }
64      }
65    }
66
67    public static void main(String[] args) {
68      BlockingQueue b;
69      b = new ArrayBlockingQueue(10);
70
71      for (int i = 0; i < 4; i++) {
72        (new Producent(i, b)).start();
73      }
74      for (int i = 0; i < 2; i++) {
75        (new Konsument(i, b)).start();
76      }
77    }
78  }
```

6.5. Programowanie rozproszone

6.5.1. Rozwiązanie zadania 5.4.1

Listing 6.45. Plik LiczacyNWD.java

```java
import java.math.BigInteger;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface LiczacyNWD extends Remote {

  public BigInteger nwd(BigInteger liczba1,
                                   BigInteger liczba2)
            throws RemoteException;
}
```

Listing 6.46. Plik Mat.java

```java
import java.math.BigInteger;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Mat extends UnicastRemoteObject
        implements LiczacyNWD {

  public Mat() throws RemoteException {
    super();
  }

  public BigInteger nwd(BigInteger liczba1,
                                   BigInteger liczba2)
            throws RemoteException {
    return liczba1.gcd(liczba2);
  }
}
```

Listing 6.47. Plik MatSerwer.java

```java
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.registry.*;

public class MatSerwer {

  public static void main(String args[]) {

    if (System.getSecurityManager() == null) {
```

```
10          System.setSecurityManager(new RMISecurityManager());
11        }
12
13        try {
14          Registry reg =
15                LocateRegistry.createRegistry(6999);
16
17          Mat obj = new Mat();
18
19          Naming.rebind("//localhost:6999/MatServer", obj);
20
21        } catch (Exception e) {
22          System.out.println("MatSerwer, błąd: "
23                                  + e.getMessage());
24        }
25    }
26 }
```

Listing 6.48. Plik NWDKlient.java

```
1 import java.math.BigInteger;
2 import java.rmi.Naming;
3
4 public class NWDKlient {
5
6   public static void main(String[] args) {
7
8     try {
9       LiczacyNWD obj = (LiczacyNWD) Naming.lookup(
10            "//serwer.umcs.lublin.pl:6999"
11            + "/MatSerwer");
12
13       BigInteger nwd;
14       nwd = obj.nwd(new BigInteger("12345"),
15                         new BigInteger("15"));
16       System.out.println(nwd);
17
18     } catch (Exception e) {
19       System.out.println("NWDKlient, błąd: "
20                              + e.getMessage());
21     }
22   }
23 }
```

6.5.2. Rozwiązanie zadania 5.4.2

Listing 6.49. Plik NWD.java

```
1  import java.math.BigInteger;
2
3  public class NWD implements Task {
4
5    private BigInteger liczba1;
6    private BigInteger liczba2;
7
8    public NWD(BigInteger liczba1, BigInteger liczba2) {
9      this.liczba1=liczba1;
10     this.liczba2=liczba2;
11   }
12
13   public Object execute() {
14     return liczba1.add(liczba2);
15   }
16
17 }
```

6.5.3. Rozwiązanie zadania 5.4.3

Listing 6.50. Plik Echo.java

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3
4  public interface Echo extends Remote {
5
6    public String echo(String s) throws RemoteException;
7
8  }
```

Listing 6.51. Plik KlasaEcho.java

```
1  import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3
4  public class KlasaEcho extends UnicastRemoteObject
5          implements Echo {
6
7    public KlasaEcho() throws RemoteException {
8      super();
9    }
10
```

```
11   public String echo(String s) throws RemoteException {
12     return s;
13   }
14 }
```

Listing 6.52. Plik SerwerEcho.java

```
1
2 import java.rmi.Naming;
3 import java.rmi.RMISecurityManager;
4 import java.rmi.registry.*;
5
6 public class SerwerEcho {
7
8   public static void main(String args[]) {
9
10     if (System.getSecurityManager() == null) {
11       System.setSecurityManager(new RMISecurityManager());
12     }
13
14     try {
15       Registry reg =
16               LocateRegistry.createRegistry(6999);
17
18       Echo obj = new KlasaEcho();
19
20       Naming.rebind("//localhost:6999/SerwerEcho", obj);
21
22     } catch (Exception e) {
23       System.out.println("SerwerEcho, błąd: "
24                                              + e.getMessage());
25     }
26   }
27 }
```

Bibliografia

[1] M. Ben-Ari. *Podstawy programowania współbieżnego.* Wydawnictwa Naukowo-Techniczne, 1989.

[2] M. Ben-Ari. *Podstawy programowania współbieżnego i rozproszonego.* Wydawnictwa Naukowo-Techniczne, 1996.

[3] M. Ben-Ari. *Podstawy programowania współbieżnego i rozproszonego.* Wydawnictwa Naukowo-Techniczne, 2009.

[4] Z. Czech. *Wprowadzenie do obliczeń równoległych.* Wydawnictwo Naukowe PWN, 2010.

[5] M. Engel. Programowanie współbieżne i rozproszone. Materiały do wykładu, dostępne online, 1999.

[6] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[7] M. Herlihy, N. Shavit. *Sztuka programowania wieloprocesorowego.* Wydawnictwo naukowe PWN, 2010.

[8] W. Iszkowski, M. Maniecki. *Programowanie współbieżne.* Wydawnictwa Naukowo-Techniczne, 1982.

[9] Oracle Technology Network. Getting Started with Java IDL. http://docs.oracle.com/javase/1.4.2/docs/guide/idl/GShome.html.

[10] Oracle Technology Network. The Java Tutorials. Lesson: Concurrency. http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html.

[11] Oracle Technology Network. The Java Tutorials. Trail: RMI. http://docs.oracle.com/javase/tutorial/rmi/index.html.

[12] I. Pyle. *Java Concurrency in Practice.* PWN, 1986.

[13] R. Segala. The essence of coin lemmas. *Electr. Notes Theor. Comput. Sci.*, 22:188–207, 1999.

[14] S. Ząbek. *Strukturalne techniki programowania.* Wydawnictwo UMCS, 2006.