Programowanie współbieżne
Wykład 2

Rafał Skinderowicz

Program współbieżny a sekwencyjny

Program – zapis algorytmu, ciąg instrukcji do wykonania przez procesor:

- *statyczny*
- sekwencja instrukcji przekształcających dane wejściowych w wyjściowe

Program sekwencyjny – program przeznaczony do wykonania przez jeden procesor (rdzeń) – pojedynczy przepływ sterowania

Program współbieżny (ang. concurrent program) – składa się ze zbioru programów sekwencyjnych lub zadań obliczeniowych (ang. task), które mogą być wykonywane równolegle.

Proces

Proces – program wykonywany na komputerze pod kontrolą *systemu operacyjnego*.

- ma charakter dynamiczny
- ma przydzielone zasoby:
  - pamięć – osobna przestrzeń adresowa, zwykle podzielona na:
    - segment kodu – instrukcje wykonywanego programu
    - segment danych – zmienne globalne programu
    - stos - dla zmiennych automatycznych, wywoływania funkcji, przekazywania parametrów i zwracania wyników
    - sterta dla zmiennych dynamicznych
  - licznik programu
  - urządzenia we/wy
- ma co najmniej jeden aktywny wątek

Proces a system operacyjny

W systemie jednoprocesorowym nie ma prawdziwej równoległości – procesy wykonywane są naprzemiennie metodą przeplotu. Podobnie w ramach jednego procesu może działać wiele wątków.

Każdy proces może być w jednym ze stanów:

- nowy
- aktywny – wykonywany przez procesor
- czekający – np. na zajście zdarzenia
- gotowy – czkający na przydzielenie procesora
- zakończony

Wątek

Wątek

Wątek to część programu wykonywana jednocześnie w ramach jednego procesu.

- Wątki w ramach procesu wykonywane są współbieżnie i korzystają z tej samej przestrzeni adresowej – efektywna komunikacja
- Tworzenie i zarządzanie wątkami jest mniej kosztowne od tworzenia i zarządzania procesami
- Wątki nazywane są często *lekkimi* procesami
- Konieczność synchronizacji dostępu do współdzielonych zasobów

Równoległość a współbieżność

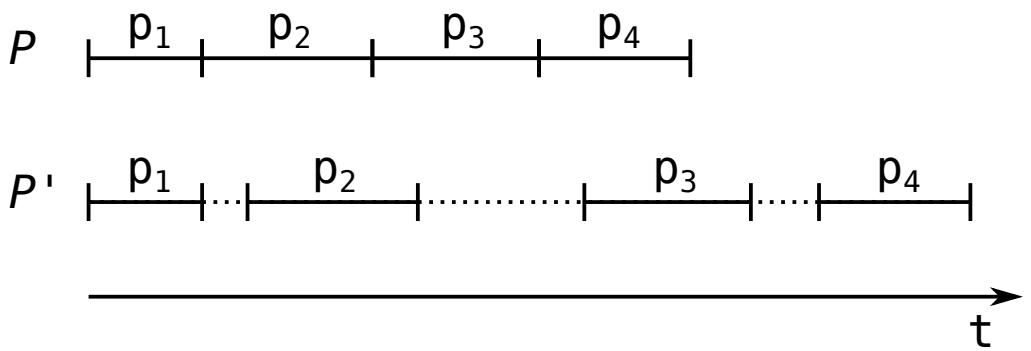
- Równoległość – wykonanie programów przez komputer nakłada się w czasie
- Współbieżność – wykonanie programów może, ale nie musi nakładać się w czasie – z punktu widzenia obserwatora zewnętrznego programy wykonują się jednocześnie, nawet w przypadku systemu jednoprocesorowego

Sposoby wykonania programów

- *sekwencyjne* – każda kolejna operacja wykonywana jest po zakończeniu poprzedniej
- *równoległe* – więcej niż jedna operacja wykonywana jest w tym samym czasie – wymaga więcej, niż jednego procesora
- *przeplatane* – wykonanie programu odbywa się fragmentami na przemian z fragmentami innych uruchomionych programów
- *współbieżne* – uogólnione pojęcie obejmujące zarówno wykonanie równoległe, jak i przeplatane

Proces sekwencyjny

Proces sekwencyjny składa się ze skończonego ciągu instrukcji. Indeks kolejnej instrukcji do wykonania wskazywany jest przez *wskaźnik instrukcji*.



Rysunek 1: Dwa równoważne wykonania procesu sekwencyjnego

Wykonanie współbieżne

- Program współbieżny składa się ze skończonego zbioru procesów sekwencyjnych, które zapisane są za pomocą skończonego zbioru instrukcji atomowych
- Wykonanie programu współbieżnego polega na wykonaniu ciągu instrukcji atomowych uzyskanego przez dowolne przeplecenie ze sobą instrukcji poszczególnych procesów.
- Każdy z możliwych przeplotów tworzy *scenariusz*.

Przeplot

- Przykład – załóżmy, że dane są procesy (lub wątki):
  - P złożony z instukcji (atomowych) p_1 i p_2
  - Q złożony z instrukcji q_1 i q_2
- Możliwe przeploty wykonania to:

$p_1 \longrightarrow q_1 \longrightarrow p_2 \longrightarrow q_2$

$p_1 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow p_2$

$p_1 \longrightarrow p_2 \longrightarrow q_1 \longrightarrow q_2$

$q_1 \longrightarrow p_1 \longrightarrow q_2 \longrightarrow p_2$

$q_1 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow q_2$

$q_1 \longrightarrow q_2 \longrightarrow p_1 \longrightarrow p_2$

Przeplot

- Przykład – załóżmy, że dane są procesy (lub wątki):
  - P złożony z instukcji (atomowych) p_1 i p_2
  - Q złożony z instrukcji q_1 i q_2
- Możliwe przeploty wykonania to:

  $p_1 \longrightarrow q_1 \longrightarrow p_2 \longrightarrow q_2$

  $p_1 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow p_2$

  $p_1 \longrightarrow p_2 \longrightarrow q_1 \longrightarrow q_2$

  $q_1 \longrightarrow p_1 \longrightarrow q_2 \longrightarrow p_2$

  $q_1 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow q_2$

  $q_1 \longrightarrow q_2 \longrightarrow p_1 \longrightarrow p_2$
- **Teoretycznie** nie wszystkie scenariusze są możliwe, np.

  $q_2 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow q_1$

Przeplot

- Przykład – załóżmy, że dane są procesy (lub wątki):
  - P złożony z instukcji (atomowych) p_1 i p_2
  - Q złożony z instrukcji q_1 i q_2
- Możliwe przeploty wykonania to:

  $p_1 \longrightarrow q_1 \longrightarrow p_2 \longrightarrow q_2$

  $p_1 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow p_2$

  $p_1 \longrightarrow p_2 \longrightarrow q_1 \longrightarrow q_2$

  $q_1 \longrightarrow p_1 \longrightarrow q_2 \longrightarrow p_2$

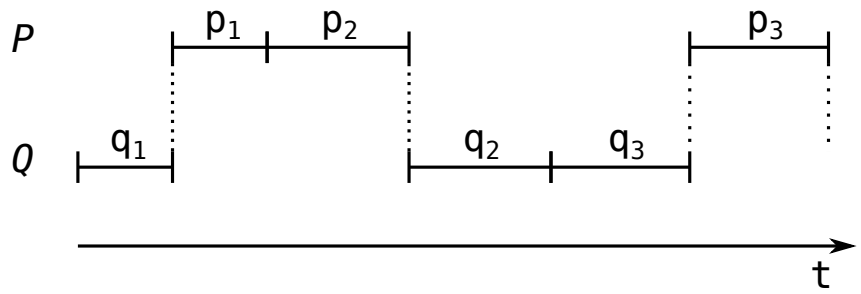
  $q_1 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow q_2$

  $q_1 \longrightarrow q_2 \longrightarrow p_1 \longrightarrow p_2$
- **Teoretycznie** nie wszystkie scenariusze są możliwe, np.

  $q_2 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow q_1$
- **W praktyce**, niestety, taki scenariusz też może być realny, w niektórych przypadkach, ale o tym później

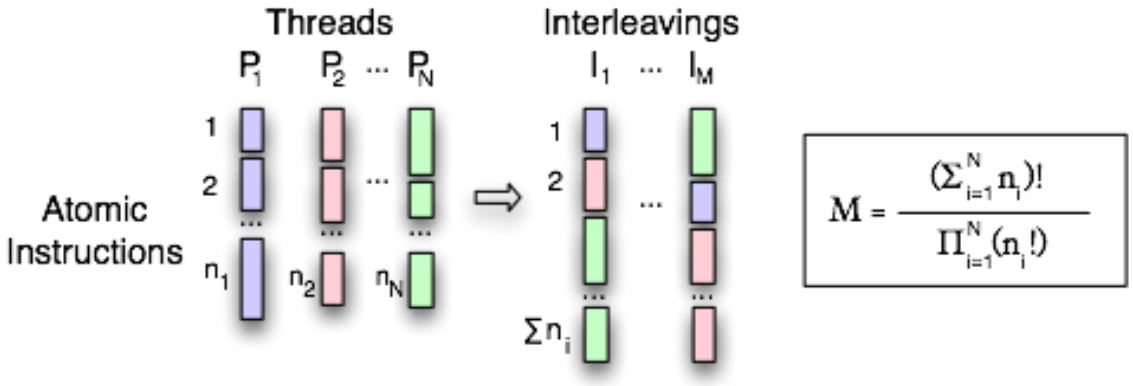
Wykonanie przeplatane

- Nic nie wiemy o względnej *kolejności*, ani *czasie* wykonania kolejnych instrukcji procesów współbieżnych.



Rysunek 2: Jeden z możliwych sposobów wykonania procesów P i Q metodą przeplotu.

Ile jest możliwych przeplotów?



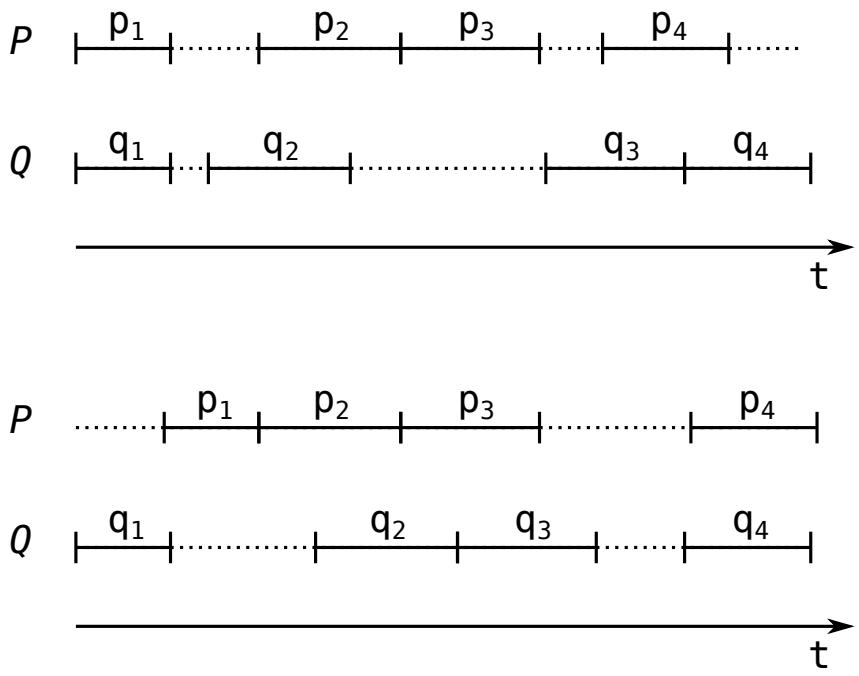
- Dla 2 wątków złożonych z 2 atomowych sekcji mamy 6 możliwych przeplotów
- Przy 8 sekcjach liczba ta wynosi 12870
- Przy 16 już **601080390** !

Wykonanie równoległe

- Wykonanie równoległe (jednoczesne) nie jest „prostsze”

Wykonanie równoległe

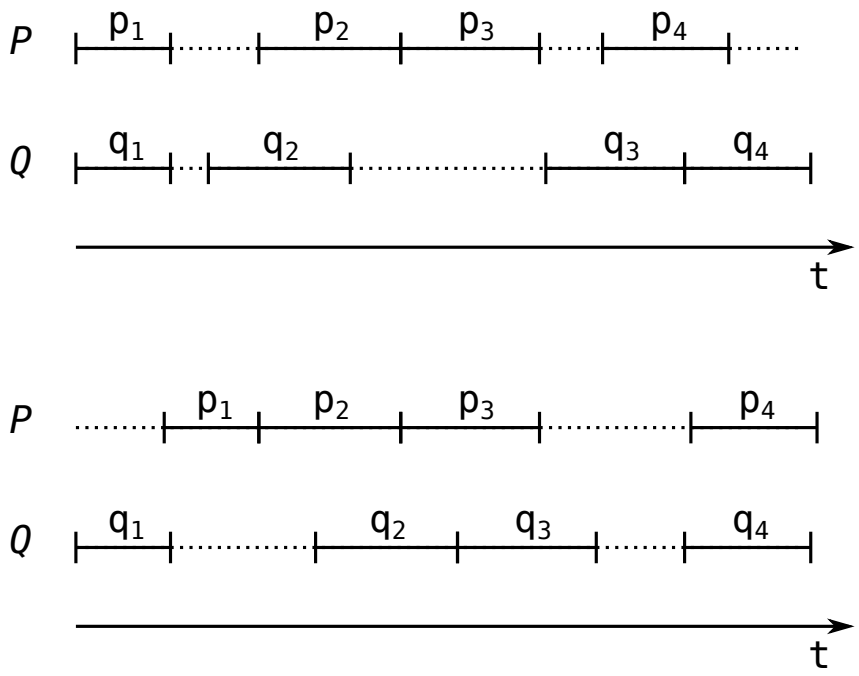
- Wykonanie równoległe (jednoczesne) nie jest „prostsze”



Rysunek 3: Przykład dwóch jednoczesnych wykonań procesów P i Q

Wykonanie równoległe

- Wykonanie równoległe (jednoczesne) nie jest „prostsze”



Rysunek 3: Przykład dwóch jednoczesnych wykonań procesów P i Q

- Nic nie wiemy o względnej *szybkości* wykonania procesów współbieżnych wykonywanych równolegle

Poprawność

- Program współbieżny powinien być poprawny dla każdego możliwego przeplotu

Poprawność

- Program współbieżny powinien być poprawny dla każdego możliwego przeplotu
- Możemy założyć, że z punktu widzenia pojedynczego wątku w programie współbieżnym, jego własne instrukcje wykonywane są jak w programie sekwencyjnym – zgodnie z porządkiem programu

Przykład 1.

Dany jest program z procesami P i Q:

int n := 0

P	Q
p_1 : n := n + 1	q_1 : n := n + 1

Przykład 1. c.d.

Jeżeli instrukcja: `n := n + 1` **jest atomowa**, to możliwe są następujące scenariusze wykonania:

Proces P	Proces Q	n
p_1 : **n := n + 1**	q_1 : n := n + 1	0
p_1 : (koniec)	q_1 : **n := n + 1**	1
p_1 : (koniec)	q_1 : (koniec)	2

Proces P	Proces Q	n
p_1 : n := n + 1	q_1 : **n := n + 1**	0
p_1 : **n := n + 1**	q_1 : (koniec)	1
p_1 : (koniec)	q_1 : (koniec)	2

Instrukcje atomowe

Instrukcje atomowe

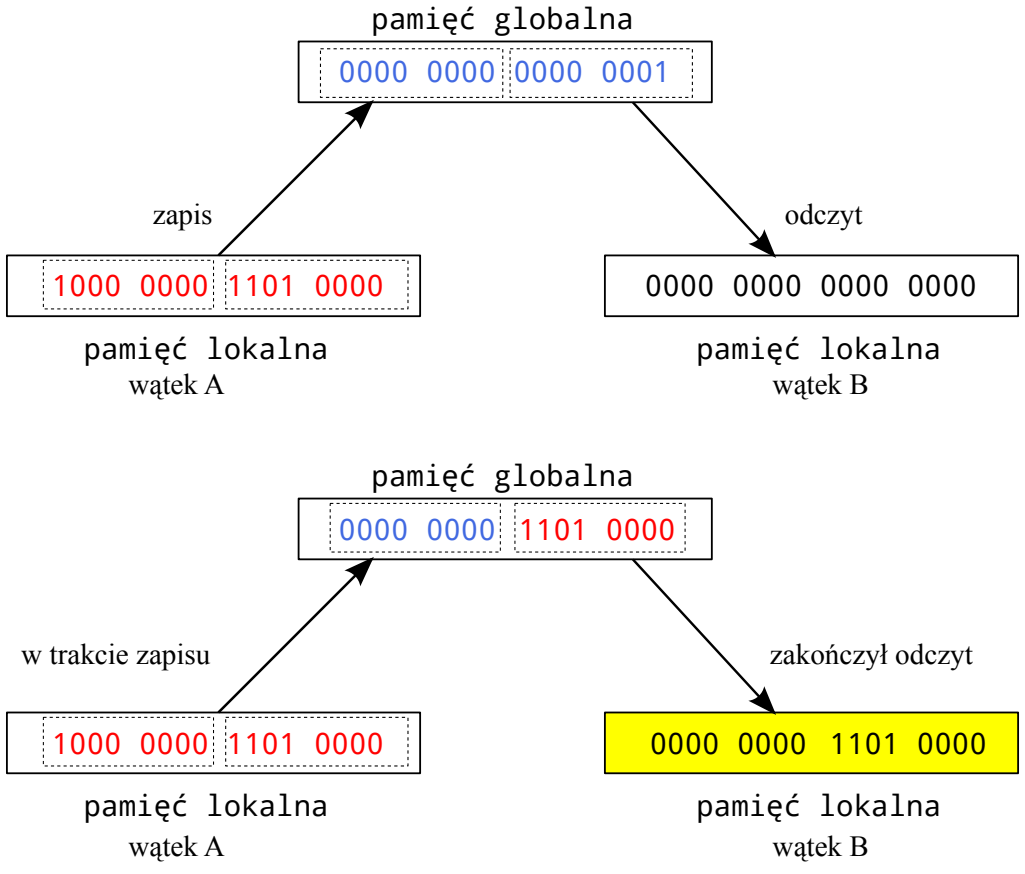
Instrukcje atomowe to instrukcje, które wykonywane są w całości bez możliwości przerwania przez instrukcje innego procesu

- Jeżeli dwie instrukcje atomowe wykonują się jednocześnie, to efekt jest taki jak gdyby wykonały się jedna po drugiej
- Lista instrukcji atomowych zależy od języka programowania
  - w językach asemblerowych możemy z łatwością korzystać z pełnej listy rozkazów danego procesora
  - w językach wysokiego poziomu dostępny jest jedynie pewien podzbiór instrukcji wspólnych dla różnych architektur

Instrukcje atomowe c.d.

- Najprostsze instrukcje atomowe to:
  - zapis do pamięci pojedynczego *słowa*
  - odczyt z pamięci pojedynczego *słowa*
  - uwaga – nie wszystkie zmienne mieszczą się w 1 słowie pamięci
- Bardziej złożone to operacje typu *odczyt-modyfikacja-zapis*:
  - np. get-and-set, get-and-increment, get-and-add, compare-and-set

Zmienne atomowe



Rysunek 4: Jednoczesna modyfikacja i odczyt zmiennej nieatomowej

Jeżeli zmienna jest **atomowa** to operacje zapisu i odczytu z niej są zawsze dokonywane w *całości*.

Przykład 1. c.d.

Załóżmy, że instrukcja: `n := n + 1` nie jest atomowa, lecz równoważna:

```
1 int tmp // zmienna lokalna
2 tmp := n
3 n := tmp + 1
```

Przykład 1. – scenariusz 1

Proces P	Proces Q	n	P.tmp	Q.tmp
p_1 : **tmp := n**	q_1 : tmp := n	0	?	?
p_2 : **n := tmp + 1**	q_1 : tmp := n	0	0	?
(koniec)	q_1 : **tmp := n**	1	-	?
(koniec)	q_2 : **n := tmp + 1**	1	-	1
(koniec)	(koniec)	2	-	-

Jak widać wynik jest poprawny (n = 2)

Przykład 1. – scenariusz 2

Proces P	Proces Q	n	P.tmp	Q.tmp
p_1 : **tmp := n**	q_1 : tmp := n	0	?	?
p_2 : n := tmp + 1	q_1 : **tmp := n**	0	0	?
p_2 : **n := tmp + 1**	q_2 : n := tmp + 1	0	0	0
(koniec)	q_2 : **n := tmp + 1**	1	-	0
(koniec)	(koniec)	1	-	-

Na skutek przeplotu instrukcji procesów P i Q otrzymaliśmy nieprawidłowy wynik **(n = 1)**

Instrukcje atomowe – c.d.

- Nawet tak prosta operacja jak inkrementacja nie musi być atomowa
- Poniższej funkcji w Javie

```
1 public void inc() {
2     ++counter;
3 }
```

- Odpowiada kod:

```
1 public void inc();
2 Code:
3    0: aload_0                    // Read obj. reference onto the stack
4    1: dup                        // Duplicate value onto the stack
5    2: getfield      #2           // Load field counter:I onto the stack
6    5: iconst_1                   // Push constant onto the stack
7    6: iadd                       // Add two values
8    7: putfield      #2           // Save result to field counter:I
9   10: return
```

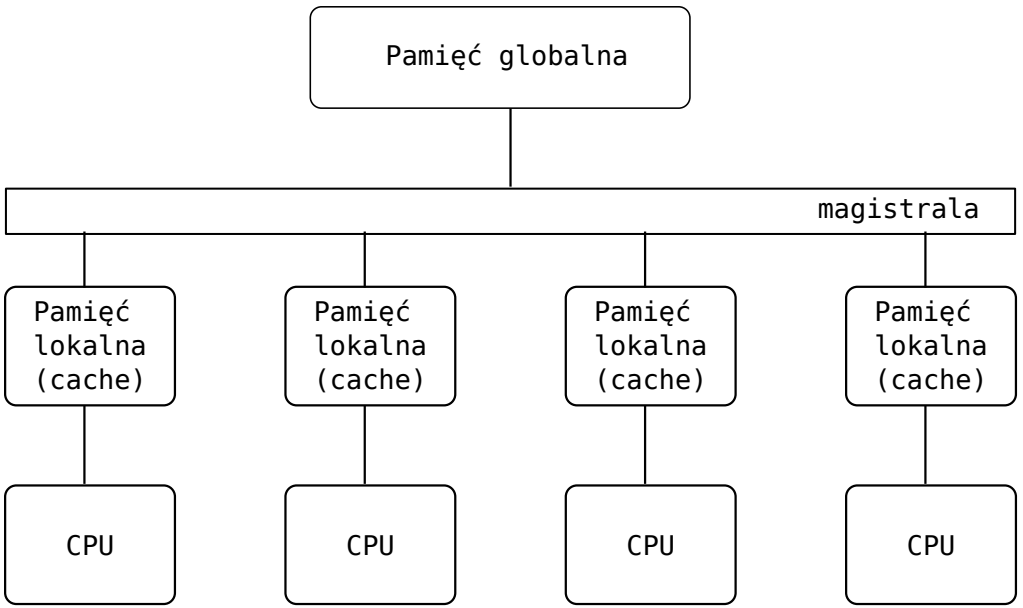
- Powyższy wynik można uzyskać za pomocą: `javap -c YourClass.class`

Rywalizacja w komputerach wieloprocesorowych

- Jeżeli więcej niż jeden procesor sięga jednocześnie do tego samego zasobu powstaje zjawisko *rywalizacji*.

Rywalizacja w komputerach wieloprocesorowych

- Jeżeli więcej niż jeden procesor sięga jednocześnie do tego samego zasobu powstaje zjawisko *rywalizacji*.
- Tym zasobem może być urządzenie I/O, ale przede wszystkim *współdzielona* pamięć



Rysunek 5: Komputer wieloprocesorowy

Poprawność programów sekwencyjnych

- Program sekwencyjny jest poprawny, jeżeli:
  - przekształca dane wejściowe w końcowe zgodnie z założeniami
  - zatrzymuje się (własność stopu)

Poprawność programów współbieżnych

- Poprawność programów współbieżnych definiuje się za pomocą własności obliczeń, a nie wyniku końcowego.
- Są dwa rodzaje własności poprawności:
  - własność bezpieczeństwa – własność, która zawsze musi być prawdziwa
  - własność żywotności – własność, która *w końcu* musi stać się prawdziwa

Własność bezpieczeństwa

- Własność bezpieczeństwa mówi, że nigdy nie dojdzie w programie do niepożądanej sytuacji
- Innymi słowy, nic złego nigdy się nie stanie w programie, np. zmienna nie otrzyma nieprawidłowej wartości
- Własnością bezpieczeństwa jest wzajemne wykluczenie w dostępie do współdzielonego zasobu

Własność żywotności

- Własność żywotności mówi, że jeżeli proces chce coś zrobić, to mu się (ostatecznie) uda

Brak żywotności może manifestować się na dwa sposoby:

- **Zakleszczanie** (ang. deadlock) – w tym przypadku w programie nic się nie dzieje, ponieważ wszystkie wątki oczekują na zajście zdarzenia, które nigdy nie zajdzie, np. zwolnienie blokady (zamka, ang. lock) posiadanej przez inny wątek
  - Często wyróżnia się zakleszczenie aktywne (ang. livelock) – stan wątków się zmienia, ale nie ma postępu obliczeń
- **Zagłodzenie** (ang. starvation) – *lokalny* brak żywotności – część procesów pracuje normalnie, ale niektóre nie mogą wykonać pracy

Poprawność programu współbieżnego

Poprawny program współbieżny

Program współbieżny jest poprawny, jeżeli dla *każdego* możliwego przeplotu ma własności bezpieczeństwa i żywotności.

- W celu pokazania, że program jest *niepoprawny* wystarczy wskazać jeden scenariusz jego wykonania, w którym własność bezpieczeństwa lub żywotności jest naruszona.

Niezagłodzenie

- Brak zagłodzenia to pożądana właściwość, ale słabsza od pozostałych
  - w praktyce przydatne, gdy występuje duża rywalizacja (ang. contention) między wątkami
  - własność ta nie określa jak długo wątek musi czekać – musi być gwarancja, że kiedyś „ruszy” z miejsca

Tworzenie wątków w Javie

- W celu utworzenia wątku w Javie mamy dwie podstawowe opcje:
  - dziedziczenie po klasie `Thread`
  - implementacja interfejsu `Runnable`

Klasa Thread

```
1  public class Watek extends Thread {
2      public Watek(String nazwa) {
3          setName(nazwa); // Ustaw nazwę wątku
4                          // domyślnie jest "Thread-nr"
5      }
6      public void run() {
7          // główna metoda wątku
8          System.out.println("Pozdrowienia z wątku: " + getName());
9      }
10     public static void main(String [] args) {
11         for (int i = 0; i < 8; ++i) {
12             Watek t = new Watek("Wątek " + i);
13             t.start();
14         }
15     }
16 }
```

Klasa Thread c.d.

Przykładowy wynik wykonania programu:

```
Pozdrowienia z wątku: Wątek 0
Pozdrowienia z wątku: Wątek 3
Pozdrowienia z wątku: Wątek 1
Pozdrowienia z wątku: Wątek 2
Pozdrowienia z wątku: Wątek 5
Pozdrowienia z wątku: Wątek 4
Pozdrowienia z wątku: Wątek 6
Pozdrowienia z wątku: Wątek 7
```

Klasa Thread c.d.

Ponowne uruchomienie zwróciło inny wynik:

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

Interfejs Runnable

```
1  package zegar;
2  class Zegar implements Runnable {
3      private volatile boolean czyZatrzymac = false;
4      public void run() {
5          int sek = 0;
6          while (!czyZatrzymac) {
7              try {
8                  Thread.sleep(1000); // uśpij wątek 1 sekundę
9                  ++sek;
10             } catch (InterruptedException e) {
11                 System.out.println("Wątek został przerwany!");
12             }
13             System.out.println("Upłynęło " + sek + " sek.");
14         }
15     }
16     public void zatrzymaj() {
17         czyZatrzymac = true;
18     }
19 }
```

Interfejs Runnable c.d.

```
1  package zegar;
2  class ZegarApp {
3      public static void main(String [] args) {
4          // wykonywane przez wątek główny aplikacji
5          Zegar z = new Zegar();
6          new Thread(z).start();
7          try {
8              Thread.sleep(5 * 1000); // uśpij główny wątek
9          } catch (InterruptedException e) {
10         }
11         z.zatrzymaj(); // bez tego program się nie zakończy
12     }
13 }
```

Różnice

- Implemetacja interfejsu, gdy klasa już dziedziczy po innej klasie
- Dziedziczenie po klasie `Thread`, jeżeli chcemy zmodyfikować jej metody
- Preferowane jest stosowanie interfejsu `Runnable`

Runnable – uwagi

Klasa implementująca `Runnable` może odwołać się do powiązanej instancji `Thread` za pomocą metody `Thread.currentThread()`, co umożliwia wykonywanie operacji jak w przypadku dziedziczenia po klasie `Thread`

```
1 class Worker implements Runnable {
2     public void run() {
3         String threadName = Thread.currentThread().getName();
4         System.out.println("Hello from " + threadName);
5         ...
```

```
1 class Worker extends Thread {
2     public void run() {
3         String threadName = this.getName();
4         System.out.println("Hello from " + threadName);
5         ...
```

Kiedy program się kończy

Program w javie kończy się, gdy:

- wszystkie wątki niebędące **demonami** (wątki usług) zakończyły działanie
  - dojście do końca wykonania metody run()
  - rzucenie wyjątku w metodzie run()
- została wykonana metoda Runtime.exit()

Kończenie pracy wątku

- Nie powinno się korzystać z metody stop() – może to doprowadzić do pozostawienia modyfikowanych przez wątek obiektów w niepoprawnym (niespójnym) stanie
- Prawidłowe kończenie pracy wątku powinno się odbywać za pomocą sprawdzania warunku opartego np. na zmiennej ulotnej:

```
1 volatile boolean czyZakonczyc = false;
2 ...
3 public void run () {
4   ...
5   while(!czyZakonczyc) {
6       ...
7   }
8 }
```

- ... lub za pomocą mechanizmu przerwań

Kończenie pracy wątku – przerwania

```
1  class Worker2 implements Runnable {
2      public void run() {
3          while (!Thread.currentThread().isInterrupted()) {
4              System.out.println("Hello :)");
5              try {
6                  Thread.sleep(10000); // 10 sek.
7              } catch (InterruptedException ex) {
8                  System.out.println("Wykonanie wątku przerwane");
9                  Thread.currentThread().interrupt(); // Przywróć flagę
       przerwania
10             }
11         }
12     }
13 }
14 ...
15 public static void main(String [] args) {
16     Thread w2 = new Thread(new Worker2());
17     w2.start();
18     Thread.sleep(1000);
19     w2.interrupt(); // Wymuś zakończenie wątku po 1 sek.
```

Przykład 2. - bankomat

- Wyobraźmy sobie system do obsługi Bankomatów
- Klasa `Konto` odpowiedzialna jest za aktualizację stanu konta
- Klasa `Bankomat` umożliwia wypłatę pieniędzy z konta
- Jest jedna instancja klasy `Konto`, ale wiele instancji klasy `Bankomat`
- Załóżmy, że możliwe jest jednoczesne wypłacanie pieniędzy z różnych bankomatów (2 karty debetowe)

Przykład 2.

```
1  class Konto {
2      private int stan = 0;
3
4      public Konto(int depozyt) {
5          stan = depozyt;
6      }
7
8      public boolean pobierz(int kwota) {
9          if (stan >= kwota) {
10             stan -= kwota;
11             return true;
12         }
13         return false;
14     }
15     public String toString() {
16         return String.valueOf(stan);
17     }
18 }
```

Przykład 2. c.d.

```
1  class Bankomat implements Runnable {
2      final Konto konto;
3      final String id;
4      public Bankomat(Konto konto, String id) {
5          this.konto = konto;
6          this.id = id;
7      }
8      public void run() {
9          int pobrane = 0;
10         final int kwota = 100;
11         for (int i = 0; i < 10; ++i) {
12             if (konto.pobierz(kwota)) {
13                 pobrane += kwota;
14             }
15             try {
16                 Thread.sleep(100);
17             } catch (InterruptedException e) { }
18         }
19         System.out.printf("Bankomat %s, pobrana kwota: %d zł\n", id,
       pobrane);
20     }
21 }
```

Przykład c.d.

```
1  public class RownoleglaModyfikacja {
2      public static void main(String [] args) {
3          final Konto konto = new Konto(10000);
4          Thread [] watki = new Thread[2];
5          for (int i = 0; i < 2; ++i) {
6              watki[i] = new Thread(new Bankomat(konto, "nr " + i));
7              watki[i].start();
8          }
9          try {
10             for (int i = 0; i < 2; ++i) {
11                 watki[i].join(); // czekaj na zakończenie
12             }
13         } catch (InterruptedException e) {
14             System.out.println("Wątek przerwany");
15         }
16         System.out.println("Stan konta: " + konto);
17     }
18 }
```

Przykładowe wykonanie programu

Wynik przykładowego wykonania programu na komputerze z dwurdzeniowym procesorem:

```
1 Bankomat nr 1, pobrana kwota: 1000 zł
2 Bankomat nr 0, pobrana kwota: 1000 zł
3 Stan konta: 8000
```

Program uruchomiony ponownie na tym samym komputerze:

```
1 Bankomat nr 0, pobrana kwota: 1000 zł
2 Bankomat nr 1, pobrana kwota: 1000 zł
3 Stan konta: 8200 (!)
```

Analiza

- Przedstawiony program ilustruje problem **wyścigu w dostępie do danych** (ang. data race)
- Oba wątki odczytują i modyfikują stan ten sam współdzielony obiekt – konto
- Nie zawsze udaje się łatwo wykryć błąd w programie współbieżnym – przy pierwszym uruchomieniu program dał poprawny wynik – **niedeterminizm**
- To czy pojawi się problem zależy od sposobu przydziału czasu procesora poszczególnym wątkom przez planistę (ang. scheduler)

Analiza

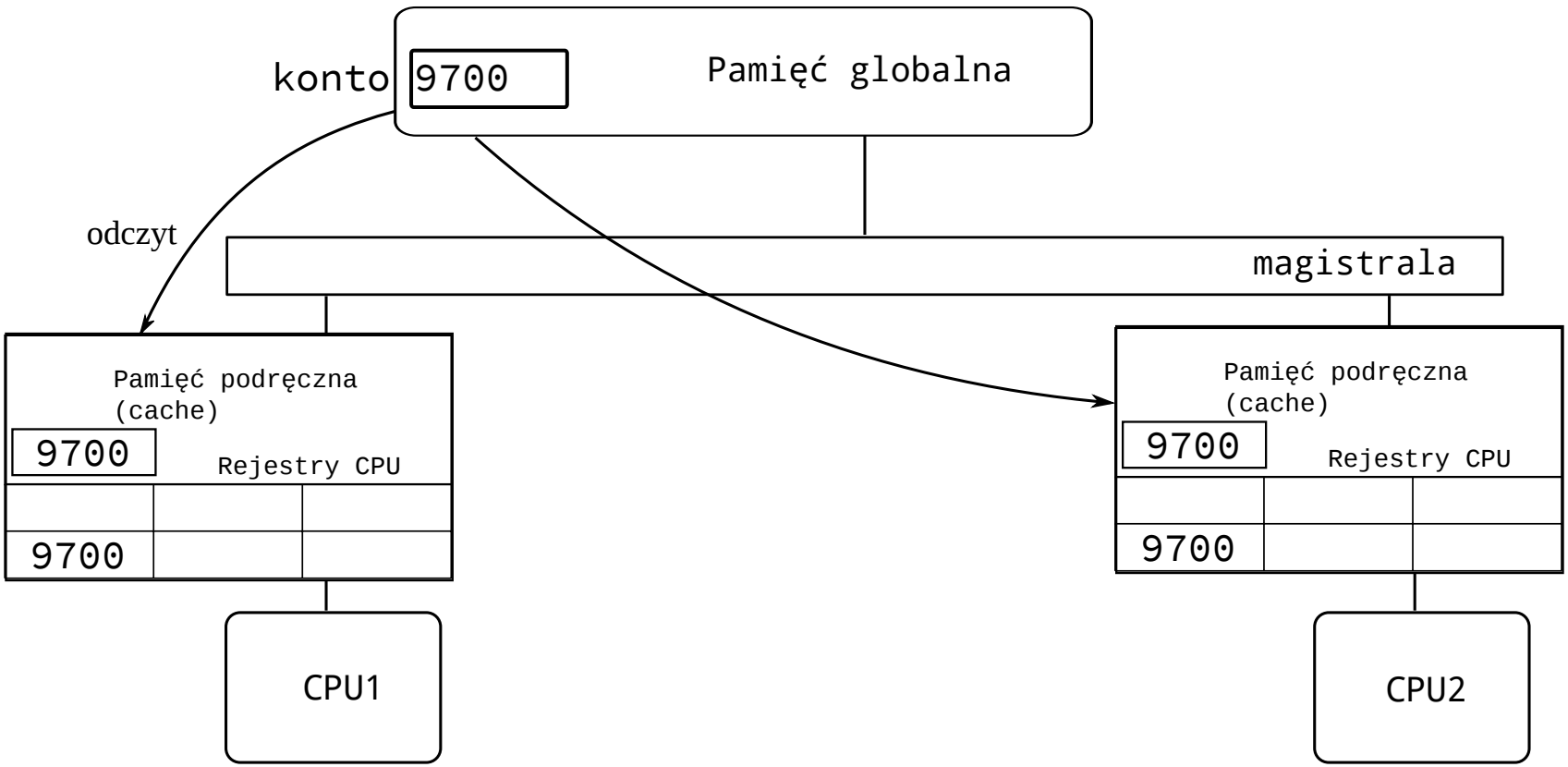
Proces 1.

```
1  Stan konta w kroku 0 = 9900
2  Stan konta w kroku 1 = 9700
3  Stan konta w kroku 2 = 9600
4  Stan konta w kroku 3 = 9500
5  Stan konta w kroku 4 = 9300
6  Stan konta w kroku 5 = 9100
7  Stan konta w kroku 6 = 8900
8  Stan konta w kroku 7 = 8800
9  Stan konta w kroku 8 = 8500
10 Stan konta w kroku 9 = 8400
```

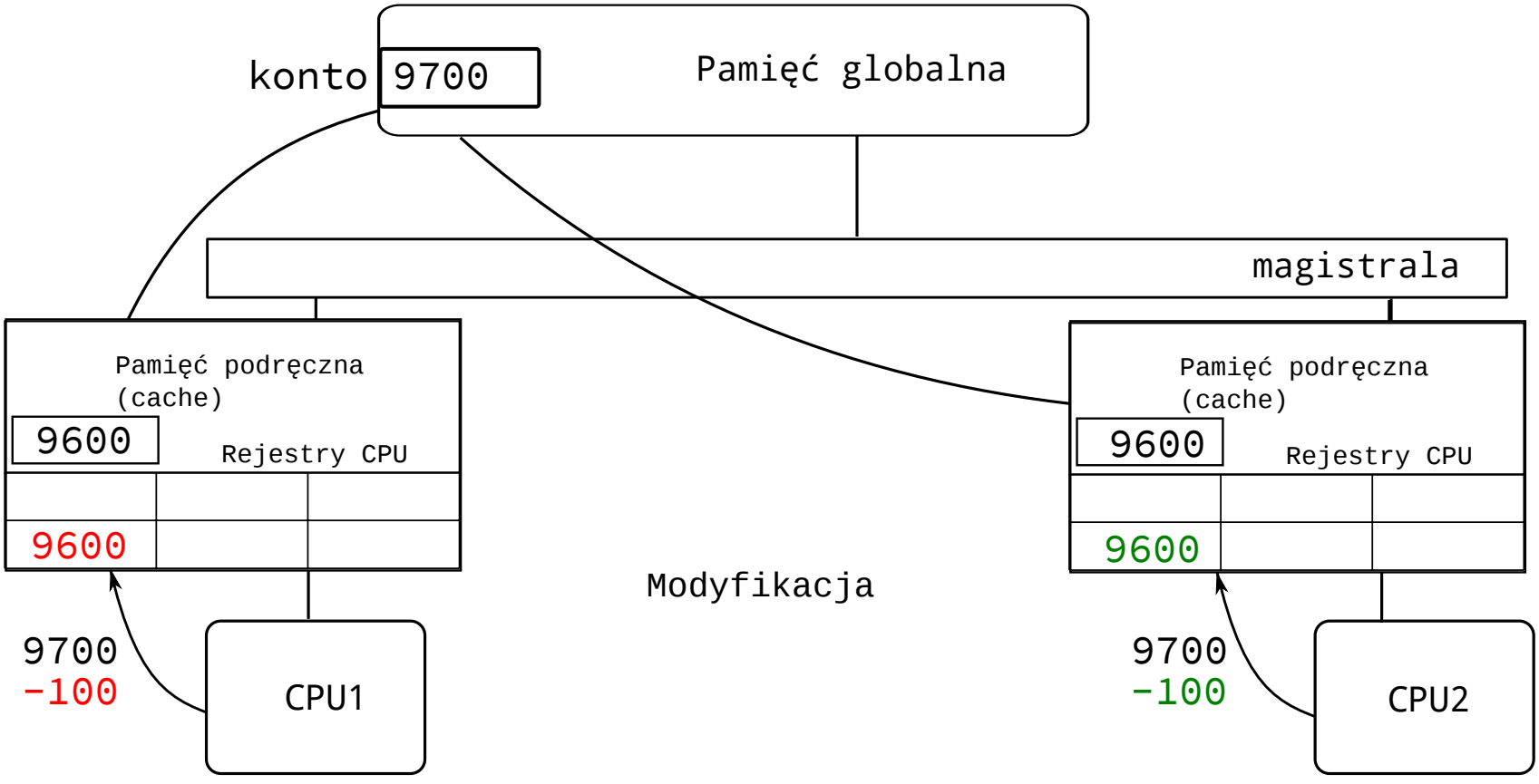
Proces 2. – rozpoczyna

```
1  Stan konta w kroku 0 = 10000
2  Stan konta w kroku 1 = 9800
3  Stan konta w kroku 2 = 9600
4  Stan konta w kroku 3 = 9400
5  Stan konta w kroku 4 = 9200
6  Stan konta w kroku 5 = 9100
7  Stan konta w kroku 6 = 8900
8  Stan konta w kroku 7 = 8800
9  Stan konta w kroku 8 = 8600
10 Stan konta w kroku 9 = 8400
```

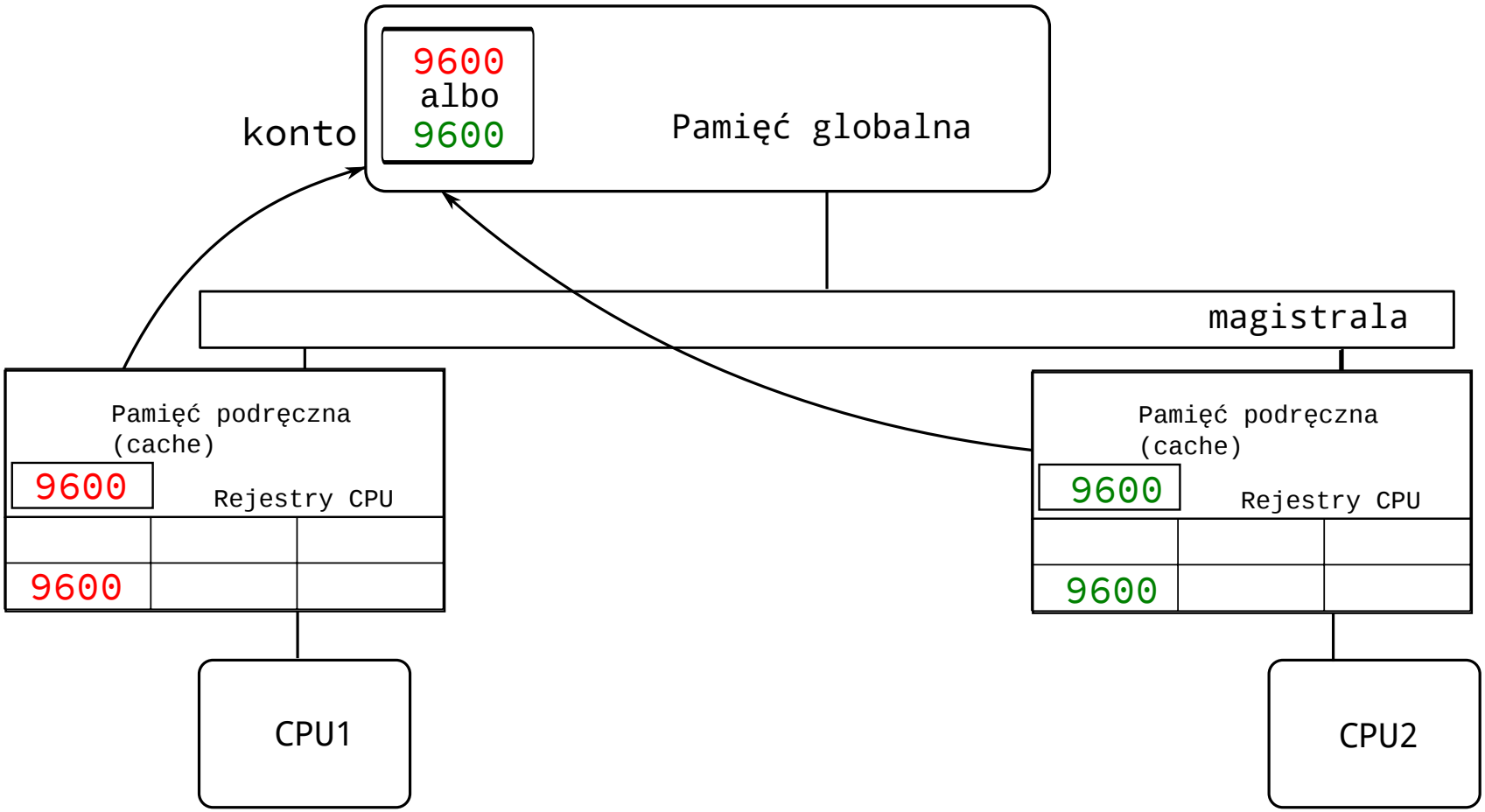
Analiza c.d.


konto
9700
Pamięć globalna
odczyt
magistrala
Pamięć podręczna
(cache)
9700
Rejestry CPU
9700
Pamięć podręczna
(cache)
9700
Rejestry CPU
9700
CPU1
CPU2

Analiza c.d.


konto
9700
Pamięć globalna
magistrala
Pamięć podręczna
(cache)
9600
Rejestry CPU
9600
Pamięć podręczna
(cache)
9600
Rejestry CPU
9600
Modyfikacja
9700
-100
CPU1
9700
-100
CPU2

Analiza c.d.


konto
9600
albo
9600
Pamięć globalna
magistrala
Pamięć podręczna
(cache)
9600
Rejestry CPU
9600
Pamięć podręczna
(cache)
9600
Rejestry CPU
9600
CPU1
CPU2

Analiza c.d.

- Problem wynika z faktu, że aktualizacja **współdzielonego** stanu konta składa się z kilku instrukcji: odczytu, modyfikacji i zapisu
- Operacje te muszą być wykonywane w sposób **wyłączny i niepodzielny** przez każdy z procesorów, czyli zgodnie z zasadą wzajemnego wykluczania

Sekcja krytyczna

- Instrukcje wykonywane na zmiennych współdzielonych tworzą **sekcję krytyczną**.
- Instrukcje wykonywane przez wątek na zmiennych lokalnych niewpływające na stan globalny noszą nazwę **sekcji lokalnej**
- W każdym momencie **tylko jeden** wątek może znajdować się w sekcji krytycznej.
- W programie może być wiele niepowiązanych ze sobą sekcji krytycznych
- Konieczna jest synchronizacja dostępu do sekcji krytycznych, tak aby zapewnić wzajemne wykluczanie wątków / procesów

Synchronizacja dostępu do współdzielonych zasobów

- Jednym ze sposobów zapewnienia synchronizacji dostępu do sekcji krytycznych w Javie jest zastosowanie konstrukcji `synchronized` (gruboziarnista synchronizacja)
- Synchronizowane mogą być zarówno całe metody, jak i wybrane bloki instrukcji (drobnoziarnista synchronizacja)

Przykład 2. – poprawiony

Synchronizacja metody:

```
1  class Konto {
2      private int stan = 0;
3      ...
4      public synchronized boolean pobierz(int kwota) {
5          boolean wynik = false;
6          if (stan >= kwota) {
7              stan -= kwota;
8              wynik = true;
9          }
10         return wynik;
11     }
12 }
```

Przykład 2. – poprawiony

Synchronizacja bloku:

```
1  class Konto {
2      private int stan = 0;
3      ...
4      public boolean pobierz(int kwota) {
5          boolean wynik = false;
6          synchronized(this) {
7              if (stan >= kwota) {
8                  stan -= kwota;
9                  wynik = true;
10             }
11         }
12         return wynik;
13     }
14 }
```

Przykład 2.

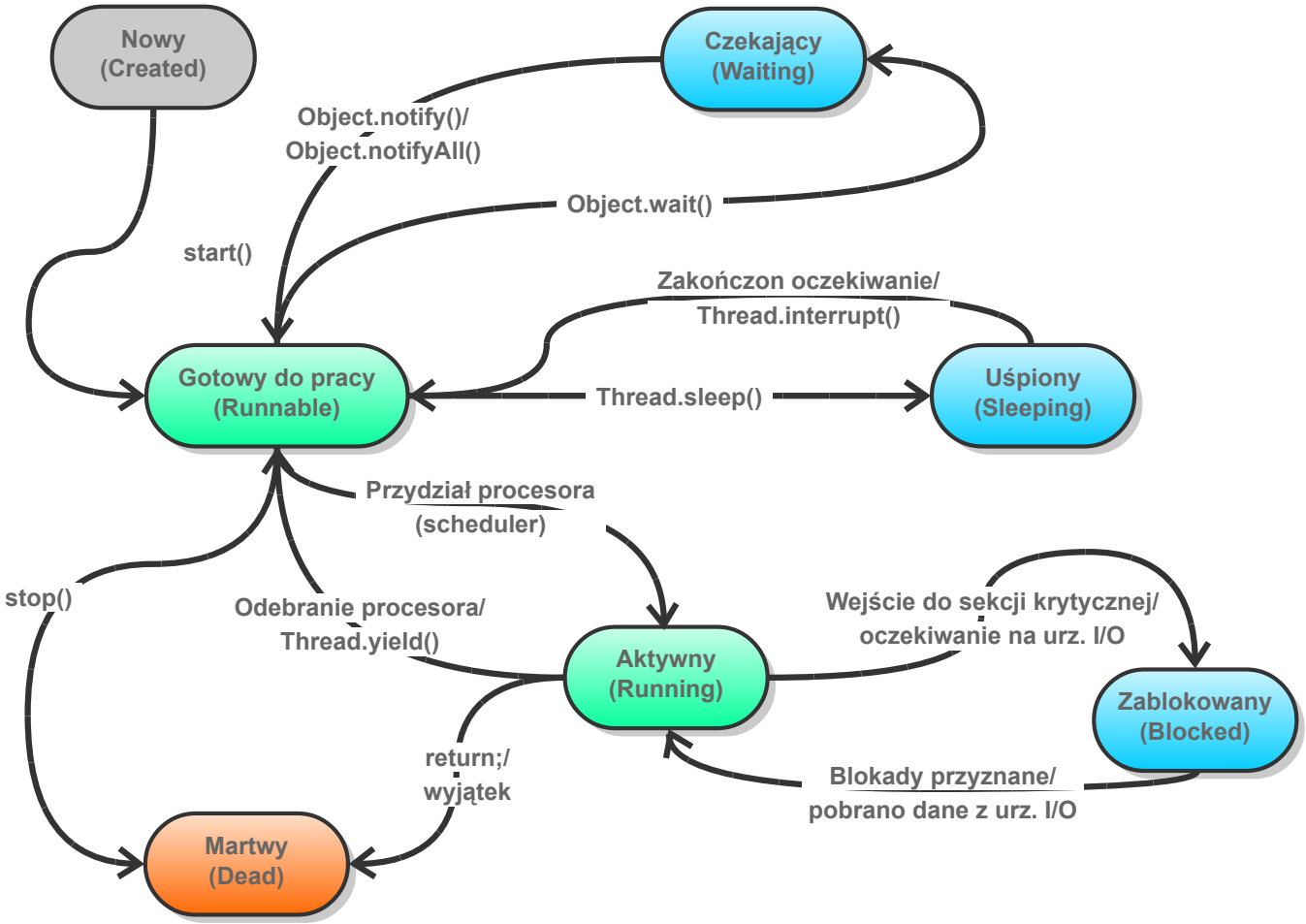
Synchronizacja bloku – wersja alternatywna:

```
1  class Bankomat implements Runnable {
2      final Konto konto;
3      final String id;
4      ...
5      public void run() {
6          int pobrane = 0;
7          final int kwota = 100;
8          for (int i = 0; i < 10; ++i) {
9              synchronized(konto) {
10                 if (konto.pobierz(kwota)) {
11                     pobrane += kwota;
12                 }
13             }
14             try {
15                 Thread.sleep(100);
16             } catch (InterruptedException e) { }
17         }
18     }
19 }
```

Zmienne i instrukcje atomowe w Javie

- Instrukcje odczytu i zapisu zmiennych typów podstawowych oprócz `long` i `double` są atomowe (dokonują się w całości)
- Instrukcje odczytu i zapisu zmiennych oraz referencji do obiektów i tablic oznaczonych jako ulotne słowem kluczowym `volatile` są atomowe
- Implementacje zmiennych (rejestrów) atomowych typu *odczyt-modyfikacja-zapis* dostępne w pakiecie `java.util.concurrent.atomic`
  - `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, `AtomicReference`
  - `AtomicIntegerArray`, `AtomicLongArray`

Cykl życia wątku



Rysunek 6: Cykl życia wątku (uproszczony).

Cykl c.d

- Nowy – obiekt wątku został utworzony, ale wątek nie został jeszcze uruchomiony (metoda start())
- Gotowy do pracy – po wykonaniu metody start(), oczekiwanie na przydział procesora przez algorytm szeregujący (inaczej planista, ang. scheduler)
- Aktywny – wątek, który uzyskał przydział procesora. Wyjście ze stanu aktywności może nastąpić przez:
  - zakończenie wykonywania metody run()
  - próbę wejścia do *sekcji krytycznej*, w której znajduje się inny wątek
  - wywołanie metod takich jak wait(), sleep(), yield()

Cykl c.d.

- Zablokowany – wątek, który był aktywny staje się zablokowany, gdy oczekuje na zwolnienie blokad przez inne wątki lub czeka na wykonanie opreacji wejścia/wyjścia
- Oczekujący – po wykonaniu metody java.lang.Object.wait() na wybranym obiekcie wątek przechdzi w stan czekania (wait), z którego może wyjść po wywołaniu przez inny wątek metody notify() lub metody notifyAll() na tym obiekcie
  - notify() wybudza jeden z oczekujących wątków, które wcześniej wywołały notify() na danym obiekcie
  - notifyAll() wybudza wszystkie oczekujące wątki (przechodzą do stanu gotowości)

Cykl c.d.

- Uśpiony – wątek, który wywołał metodę sleep() i nie upłynął jeszcze zadany kwant czasu
  - wątek może zostać wybudzony przez wywołanie metody interrupt(), co powoduje rzucenie wyjątku klasy InterruptedException
  - po zakończeniu oczekiwania wątek przechodzi do stanu gotowości, a nie aktywności
- Zakończony (martwy) – wątek, który zakończył wykonywanie metody run() lub został przerwany przez nieobsłużony wyjątek lub metodę stop()/terminate()

Zarządzanie przydziałem procesora

Cechy podstawowe:

- JVM stosuje algorytm szeregujący z wywłaszczeniem
- Kolejność przyznawania czasu procesora zależy od priorytetu wątku
- Każdy wątek ma określony priorytet, wykonywany jest wątek o najwyższym priorytecie
- Jeżeli dwa wątki mają taki sam priorytet uruchamiane są w kolejności FIFO (ang. first in, first out)

Priorytety wątków

- JVM wybiera wątek o najwyższym priorytecie do wykonania
- Priorytety mają zakres od 1 (najniższego, Thread.MIN_PRIORITY) do 10 (najwyższego, Thread.MAX_PRIORITY)
- Domyślny priorytet to Thread.NORM_PRIORITY
- Nowo tworzony wątek przyjmuje domyślnie priorytet wątku, w którym został utworzony