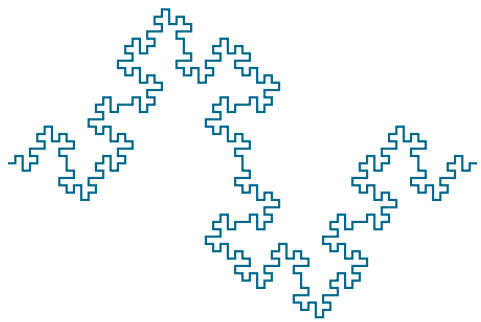
Model pamięci

Rafał Skinderowicz



Czym jest model pamięci

- Model pamięci dotyczy programów współbieżnych
- W programie współbieżnym może się zdarzyć, że dany wątek nie będzie „widział” od razu wartości zmiennej zmodyfikowanej przez inny wątek
- Model pamięci określa *kiedy* efekty operacji (na pamięci) wykonywanych w ramach danego wątku stają się widoczne dla pozostałych wątków

Motywacje

Rozbieżności mogą pojawić się z dwóch powodów

- Braku spójności pamięci ze względu na użycie pamięci podręcznej
  - W większości procesorów nie ma z tym problemu – zapewniona jest spójność pamięci podręcznych (ang. cache coherency), np. za pomocą protokołu MESI
  - Tymczasowy brak spójności ze względu na przetrzymywanie zmiennej w rejestrze procesora, buforze zapisu (ang. store buffer)
- Zmiany kolejności wykonywanych instrukcji – na poziomie programowym i sprzętowym:
  - Stosowane powszechnie przez większość kompilatorów w celu optymalizacji
  - Wewnątrz procesorów w celu ukrycia opóźnień wynikających z długiego czasu dostępu do pamięci

Zmiana kolejności instrukcji

Procesor w ramach optymalizacji może zmieniać kolejność w jakiej wykonuje instrukcje.

Wykonanie zgodnie z porządkiem (ang. *in-order execution*):

- instrukcje są ładowane, wykonywane i kończone zgodnie z porządkiem kodu wygenerowanego przez kompilator
- jeżeli instrukcja potrzebuje operandu z pamięci operacyjnej, to konieczne jest jego załadowanie – oczekiwanie
- schemat ten jest *statyczny*

Wykonanie niezgodnie z porządkiem (ang. *out-of-order execution*)

- instrukcje są ładowane zgodnie z porządkiem w skompilowanym kodzie
- jeżeli nie ma załadowanych operandów dla instrukcji, to jest dodawana do *kolejki instrukcji*
- kolejna instrukcja jest ładowana
- po załadowaniu potrzebnych danych instrukcja jest usuwana z kolejki i wykonywana
- schemat ten jest *dynamiczny*

Przykład 1

- Zmiana kolejności wykonywanych instrukcji nie może zmieniać semantyki programu – musi prowadzić do tego samego wyniku, jak źródłowa lista instrukcji
- Przykład:

```
1 int a = b + 1;
2 int c = b * 3;
3 int d = a / 2;
```

- Instrukcje 1. oraz 3. muszą zostać wykonane jedna po drugiej – występuje między nimi *zależność danych*
- Wynik instrukcji 2. nie jest powiązany z wynikami pozostałych

Przykład 1

- Kompilator w ramach optymalizacji może zmienić kolejność wykonywania instrukcji, tak by wygenerować bardziej efektywny kod
- Z punktu widzenia programu sekwencyjnego, czy danego wątku nie ma to znaczenia
- Każdy wątek widzi efekty swoich instrukcji, tak jak gdyby były one wykonywane zgodnie z porządkiem programu

Równoważne kolejności:

```
1 int a = b + 1;
2 int c = b * 3;
3 int d = a / 2;
```

.

```
1 int c = b * 3;
2 int a = b + 1;
3 int d = a / 2;
```

.

```
1 int a = b + 1;
2 int d = a / 2;
3 int c = b * 3;
```

Przykład 2

Optymalizacja pętli przez łączenie

```
1 int i, a[100], b[100];
2 for (i = 0; i < 100; i++)
3     a[i] = 1;
4 for (i = 0; i < 100; i++)
5     b[i] = 2;
```

Równoważna postać

```
1 int i, a[100], b[100];
2 for (i = 0; i < 100; i++) {
3     a[i] = 1;
4     b[i] = 2;
5 }
```

Przykład 3

```
1 boolean warunek = true;
2 /* ... */
3 {
4     while (warunek) {} // czekaj
5 }
```

Powyższa pętla w ramach optymalizacji może przez kompilator zamieniona na *równoważną* postać

```
1 {
2     while (true) {} // czekaj
3 }
```

Przykład 3

```
1 boolean warunek = true;
2 /* ... */
3 {
4     while (warunek) {} // czekaj
5 }
```

Powyższa pętla w ramach optymalizacji może przez kompilator zamieniona na *równoważną* postać

```
1 {
2     while (true) {} // czekaj
3 }
```

Gdybyśmy użyli `volatile boolean warunek = true;` taka zmiana nie byłaby możliwa

Przykład 4

- Załóżmy, że wątki A i B wykonują *jednocześnie*, odpowiednio, metody `read()` oraz `write()`

```
1  class Reordering {
2      int x = 0 , y = 0;
3      public void write() {
4          x = 1;
5          y = 2;
6      }
7      public int [] read() {
8          int r1 = y;
9          int r2 = x;
10         return new int [] { r1, r2 };
11     }
12 }
```

- Można by się spodziewać, że wątek B zobaczy jedną z dwóch konfiguracji:
  - `r1 = 0` oraz `r2 = 0`
  - `r1 = 2` oraz `r2 = 1`

Przykład 3

- Załóżmy, że wątki A i B wykonują *jednocześnie*, odpowiednio, metody `read()` oraz `write()`

```
1  class Reordering {
2      int x = 0 , y = 0;
3      public void write() {
4          x = 1;
5          y = 2;
6      }
7      public int [] read() {
8          int r1 = y;
9          int r2 = x;
10         return new int [] { r1, r2 };
11     }
12 }
```

- Jednak może się również zdarzyć, że wątek B na skutek zmiany kolejności instrukcji w wierszach 4. i 5. zobaczy wartości `r1 = 2` oraz `r2 = 0`

Model pamięci

- Model pamięci określa kiedy procesor / wątek zobaczy efekty instrukcji innego procesora / wątku
- Model pamięci można zdefiniować:
  - na poziomie CPU – niskopoziomowy, powiązany z konkretną architekturą
  - na poziomie oprogramowania, np. języka programowania – konstrukcje programowe muszą być przetłumaczone na odpowiednie rozkazy dla CPU

Model pamięci

- Modele można podzielić na dwie główne kategorie:
  - mocne (ang. *strong*) – zmiany są widoczne natychmiastowo
  - słabe (ang. *weak*) – zmiany widoczne są z pewnym opóźnieniem, np. po wykonaniu tzw. bariery pamięci (ang. *memory barrier* lub *memory fence*)

Bariery pamięci

- Bariery na poziomie sprzętu realizowane są przez specjalne rozkazy procesora
- Bariery na poziomie oprogramowania wymuszają względną kolejność wykonania instrukcji przed i po barierze

Bariery pamięci

- Podsumowując, bariera pamięci wprowadza relację poprzedzania (ang. *happens before*) między grupami instrukcji

Bariery pamięci

- Podsumowując, bariera pamięci wprowadza relację poprzedzania (ang. *happens before*) między grupami instrukcji
- Wracając do przykładu:

```
1  public void write() {
2      x = 1;
3      // Tutaj przydałaby się bariera
4      y = 2;
5  }
6  public int [] read() {
7      int r1 = y;
8      int r2 = x;
9      return new int [] { r1, r2 };
10 }
```

Bariery pamięci

- Podsumowując, bariera pamięci wprowadza relację poprzedzania (ang. *happens before*) między grupami instrukcji
- Wracając do przykładu:

```
1  public void write() {
2      x = 1;
3      // Tutaj przydałaby się bariera
4      y = 2;
5  }
6  public int [] read() {
7      int r1 = y;
8      int r2 = x;
9      return new int [] { r1, r2 };
10 }
```

- Bariery pamięciowe w językach wysokiego poziomiu zazwyczaj wstawiane są w sposób niejawny

Model pamięci na poziomie CPU

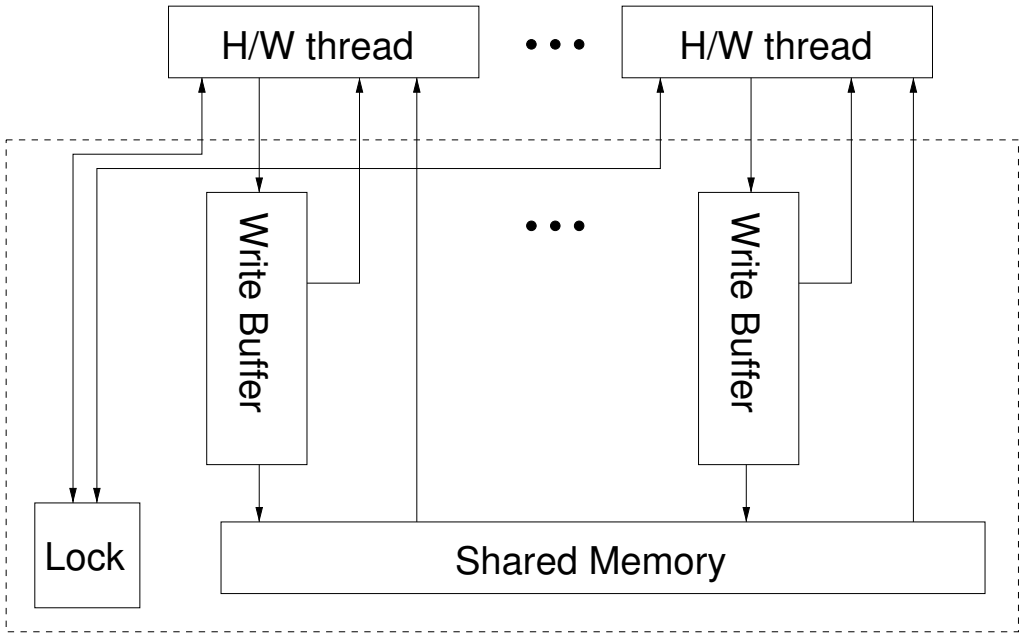
- Okazuje się, że zdefiniowanie modelu pamięci „pasującego" do rzeczywistych procesorów jest trudnym zadaniem
- Model taki musi uwzględniać m.in. wykonywanie instrukcji niezgodnie z kolejnością, ale również opóźnienia w zapisie danych do pamięci
- Jednym z modeli pasujących do współczesnych procesorów w architekturze x86 jest model x86-TSO (Total Store Ordering)[1]

[1] Sewell, Peter, et al. "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors." Communications of the ACM 53.7 (2010): 89-97.

Model x86-TSO

Model x86-TSO składa się z:

- sprzętowo realizowanych wątków (H/W threads)
- współdzielonej pamięci
- bufora zapisu dla każdego z wątków
- globalnej blokady (lock) dającej wyłączny dostęp do pamięci



Rysunek : Diagram abstrakcyjnej maszyny w modelu x86-TSO

Działanie modelu x86-TSO

1. Bufory zapisu działają jak kolejka FIFO, przy operacji odczytu wątek musi najpierw sprawdzić, czy odczytywane dane nie znajdują się w jego buforze zapisu.
2. Instrukcja bariery pamięciowej (MFENCE) opróżnia bufor zapisu danego wątku
3. Przed wykonaniem instrukcji blokowanej (LOCK [INSTR]) wątek musi najpierw zdobyć blokadę globalną – na zakończenie instrukcji wątek opróżnia swój bufor zapisu. Gdy blokada jest założona przez wątek, żaden inny nie może czytać.
4. Bufor zapisu może być opróżniony w dowolnym momencie, pod warunkiem, że inny wątek nie założył globalnej blokady

Testy lakmusowe

- W specyfikacji modelu pamięci producenci procesorów posługują się często zestawem tzw. testów *lakmusowych* (ang. litmus tests)[2]

Tabela : Przykładowy test

Proc 0	**Proc 1**
MOV [x] ← 1	MOV EAX ← [y]
MOV [y] ← 1	MOV EBX ← [x]
Zabroniony stan końcowy: Proc 1:EAX=1 oraz Proc 1:EBX=0	

- Test zapewnia, że zapisy dokonywane przez Proc 0 są widziane zgodnie z porządkiem programu dla odczytów przez Proc 1. x86-TSO zapewnia spełnienie testu, ponieważ bufor zapisu Proc 0 działa jak kolejka FIFO

[2]Intel 64 architecture memory ordering white paper, 2007. Intel Corporation. SKU 318147-001.

Model pamięci Javy

- Model pamięci języka Java określa kiedy zachodzi relacja poprzedzania dla operacji na pamięci (i nie tylko)
- W Javie bariery pamięci wstawiane są niejawnie, m.in. przy okazji synchronizacji na *monitorze*
  - za pomocą synchronized
  - za pomocą blokad / semaforów
  - przez monitor należy tu rozumieć obiekt, na rzecz którego zakładana jest blokada
- Synchronizacja gwarantuje, że efekty wykonania synchronizowanego bloku będą widoczne dla innych wątków, które będą synchronizowane na tym samym monitorze

Model pamięci Javy – przykład

```
1 int data[] = new int [] { 0, 0, 0, 0, 0 };
2 boolean is_ready = false;
3 void init_data() {
4   for( int i=0; i < 5; ++i )
5     data[i] = i;
6   is_ready = true;
7 }
```

CPU może wykonać powyższy fragment z zachowaniem różnej kolejności instrukcji

```
1 store data[0] 0
2 store data[1] 1
3 store data[2] 2
4 store data[3] 3
5 store data[4] 4
6 store is_ready 1
```

```
1 store data[3] 3
2 store data[4] 4
3 store is_ready 1
4 store data[0] 0
5 store data[1] 1
6 store data[2] 2
```

Model pamięci Javy – przykład

Jeżeli użyjemy bloku synchronized

```
1 void init_data() {
2   synchronized( this ) {
3     for( int i=0; i < 5; ++i )
4       data[i] = i;
5   }
6   is_ready = true;
7   return data;
8 }
```

to niejawnie zostanie wstawiona bariera pamięciowa

```
1 store data[3] 3
2 store data[4] 4
3 store data[0] 0
4 store data[1] 1
5 store data[2] 2
6 fence
7 store is_ready 1
```

Teraz `is\_ready = true` zawsze będzie wykonane po zakończeniu bloku synchronizowanego

Model pamięci Javy

W poniższym przykładzie wątek A wykonał metodę `put()` wstawiając nowy obiekt do kolejki. Wątek B wykonuje później metodę `get()` jednak *nie ma* gwarancji, że zobaczy element wstawiony przez wątek A, ponieważ synchronizacja wykonywana jest na *różnych* obiektach (`lock1` i `lock2`)

```
1  void put(Object o) throws InterruptedException {
2      try {
3          lock1.lock();
4          ...
5      } finally {
6          lock1.unlock();
7      }
8  }
9  Object get() throws InterruptedException {
10     try {
11         lock2.lock(); // Uwaga -- inna blokada
12         ...
13     } finally {
14         lock2.unlock();
15     }
16 }
```

Bariera pamięci – volatile

Model pamięci Javy gwarantuje, że wszystkie wartości zmiennych widoczne dla wątku przed odczytem / zapisem zmiennej `volatile` będą również widoczne dla każdego innego wątku, który dokona (później) odczytu / zapisu tej samej zmiennej

```
1  Map configOptions;
2  char[] configText;
3  volatile boolean initialized = false;
4  // Wątek A
5  configOptions = new HashMap();
6  configText = readConfigFile(fileName);
7  processConfigOptions(configText, configOptions);
8  initialized = true;
9  // Wątek B
10 while (!initialized)
11   sleep();
12 // <- mamy gwarancję, że configOptions zostało zainicjowane
```

Model pamięci Javy – relacja poprzedzania

Model pamięci Javy wprowadza relację *poprzedzania* między niektórymi operacjami, m.in.:

- Każda instrukcja w wątku *poprzedza* każdą inną instrukcję, która następuje po niej zgodnie z porządkiem programu
- Zwolnienie blokady *porzedza* kolejne założenie tej blokady
- Zapis do zmiennej ulotnej *poprzedza* kolejne odczyty z tej zmiennej
- Wywołanie `Thread.start()` *poprzedza* każdą inną operację w wątku
- Wywołanie `Thread.join()` *jest porzedzone* przez wszystkie pozostałe operacje

Model pamięci Javy – słowo kluczowe final

- Zmienna zadeklarowana jako finalna musi zostać zainicjowana przed zakończeniem działania konstruktora

```
1 public class MyClass {
2   private final int myField = 3;
3   public MyClass() {
4     ...
5   }
6 }
```

```
1 public class MyClass {
2   private final int myField;
3   public MyClass() {
4     ...
5     myField = 3;
6     ...
7   }
8 }
```

- Model pamięci gwarantuje, że odczyt zmiennych finalnych jest bezpieczny, nawet bez synchronizacji, jednak pod warunkiem...

Model pamięci Javy – słowo kluczowe final

- Nie można udostępniać referencji do konstruowanego obiektu (`this`) zanim konstruktor nie zakończy pracy

```
1  public class NumberPrinter {
2      public NumberPrinter(int number, EventSource source) {
3          this.number = number; // inicjalizacja pola finalnego
4          source.registerListener(
5              new EventListener() {
6                  public void onEvent() {
7                      printNumber(); // niejawne odniesienie do this!
8                  }
9              });
10         // EventSource może wywołać printNumber(), a konstruktor
11         // jeszcze nie zakończył pracy !
12     }
13     public void printNumber() { System.out.println(number); }
14     final int number;
15 }
```

Model pamięci Javy – statyczna inicjalizacja

- Jeżeli wartość statycznej zmiennej jest ustalana za pomocą statycznej inicjalizacji, to każdy wątek będzie widział wartość po jej ustaleniu bez konieczności dodatkowej synchronizacji

```
1 class ANumber {
2   static final Double value;
3   static {
4       value = Math.sin(Math.PI);
5   }
6   // lub po prostu
7   // static final Double value = Math.sin(Math.PI);
8 }
```

Model pamięci Javy – statyczna inicjalizacja

Przykład wykorzystania do *efektywnej* implementacji singletona

```
1  class Singleton {
2      private static class SingletonHolder {
3         public static Singleton INSTANCE = new Singleton();
4      }
5      private Singleton() { // konstruktor
6          System.out.println("Tworzę instancję");
7      }
8      public static Singleton getInstance() {
9         return SingletonHolder.INSTANCE;
10     }
11 }
```

Implementacja ta jest *leniwa*, tzn. klasa `SingletonHolder` zostanie zainicjowana dopiero, gdy będzie potrzebna, tj. gdy nastąpi pierwsze wywołanie metody `getInstance()`

Singleton w C#

Dla porównania, implementacja singletona w C# za pomocą klasy Lazy<T>

```
1  public class MySingleton {
2      private static readonly Lazy<MySingleton> mySingleton =
3          new Lazy<MySingleton>(() => new MySingleton());
4
5      private MySingleton() { }
6
7      public static MySingleton Instance {
8          get {
9              return mySingleton.Value;
10         }
11     }
12 }
```

Modele pamięci w popularnych językach programowania

- Java
- C#
- C++ dopiero od standardu C++11
- Clojure
- Erlang

Modele pamięci – uwagi

- Niestety, modele pamięci dla poszczególnych języków programowania różnią się
- Pisanie poprawnych i efektywnych programów współbieżnych wymaga zapoznania się z konkretnym modelem
- Zazwyczaj nie ma istotnych różnic między „utartymi” mechanizmami, jak np. blokady, semafory, czy monitory
- Dzięki dobrej znajomości modelu pamięci w danym języku programowania jesteśmy w stanie pisać programy nie tylko poprawne, ale i o większej wydajności

Modele pamięci – uwagi

Przykład z książki „C# 4.0 in a Nutshell”, J. Albahari

```
1  class IfYouThinkYouUnderstandVolatile {
2    volatile int x, y;
3    void Test1() {      // Executed on one thread
4      x = 1;            // Volatile write (release-fence)
5      int a = y;        // Volatile read (acquire-fence)
6      ...
7    }
8    void Test2() {      // Executed on another thread
9      y = 1;            // Volatile write (release-fence)
10     int b = x;        // Volatile read (acquire-fence)
11     ...
12   }
13 }
```

Modele pamięci – uwagi

Przykład z książki „C# 4.0 in a Nutshell”, J. Albahari

```
1  class IfYouThinkYouUnderstandVolatile {
2    volatile int x, y;
3    void Test1() {      // Executed on one thread
4      x = 1;            // Volatile write (release-fence)
5      int a = y;        // Volatile read (acquire-fence)
6      ...
7    }
8    void Test2() {      // Executed on another thread
9      y = 1;            // Volatile write (release-fence)
10     int b = x;        // Volatile read (acquire-fence)
11     ...
12   }
13 }
```

Zarówno a, jak i b mogą otrzymać wartość 0, ponieważ kolejność istrukcji odczytu i zapisu zmiennych `volatile` może zostać zmieniona