Programowanie współbieżne
Rozwiązywanie problemu wzajemnego wykluczania

Rafał Skinderowicz

Wady protokołów WW z zastosowaniem atomowych rejestrów RW

Rozwiązywanie problemu wzajemnego wykluczania za pomocą rozwiązań programowych bazujących wyłącznie na *atomowych* instrukcjach odczytu i zapisu:

- jest stosunkowo wolne
- wymaga dodatkowej pamięci o rozmiarze liniowo rosnącym wraz z liczbą procesorów

Alternatywa

- Lepszą alternatywą jest zastosowanie *silniejszych* typów rejestrów typu *odczyt-modyfikacja-zapis* (ang. read-modify-write, RMW)
- Implementowane za pomocą specjalnych rozkazów procesora
  - możemy mówić o nich jako rozwiązaniach sprzętowo–programowych

getAndSet()

Pseudokod implementacji operacji `getAndSet(v)`

```
1 public abstract class RMWRegister {
2    private int value;
3
4    public int synchronized getAndSet(int v) {
5        int prior  = this.value;
6        this.value = v; // Nowa wartość rejestru
7        return prior;   // Zwróć poprzednią
8    }
9 }
```

getAndSet()

Pseudokod implementacji operacji `getAndSet(v)`

```
1 public abstract class RMWRegister {
2    private int value;
3
4    public int synchronized getAndSet(int v) {
5        int prior  = this.value;
6        this.value = v; // Nowa wartość rejestru
7        return prior;   // Zwróć poprzednią
8    }
9 }
```

- Jak wspomniano, w praktyce realizowane za pomocą pojedynczych rozkazów procesora

getAndIncrement()

```java
public abstract class RMWRegister {
    private int value;

    public int synchronized getAndIncrement() {
        int prior  = this.value;
        this.value = this.value + 1; // Nowa wartość
        return prior;    // Zwróć poprzednią
    }
}
```

getAndAdd()

```
1 public abstract class RMWRegister {
2    private int value;
3 
4    public int synchronized getAndAdd(int v) {
5        int prior  = this.value;
6        this.value = this.value + v; // Nowa wartość
7        return prior;    // Zwróć poprzednią
8    }
9 }
```

compareAndSet()

```
1  public abstract class RMWRegister {
2  private int value;
3  public boolean synchronized compareAndSet(
4                                   int expected, int update) {
5      int prior = this.value;
6      if (this.value == expected) {
7          this.value = update;
8          return true;
9      } // else
10     return false;
11 }
12 }
```

- `expected` – oczekiwana *bieżąca* wartość
- `update` – nowa wartość
- We współczesnych procesorach (AMD, Intel) odpowiednikiem jest instrukcja porównaj-i-wymień (ang. compare-and-swap), rozkaz CMPXCHG

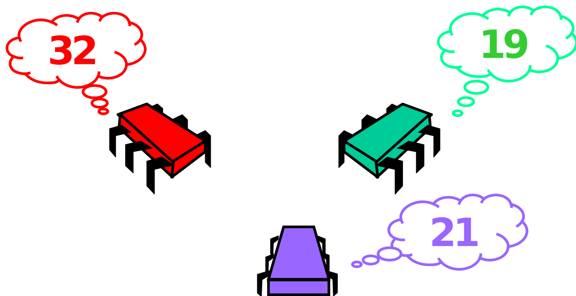
Różnice

- Przedstawione operacje RMW są przydatne do implementacji:
  - algorytmów WW
  - niewstrzymywanych (ang. lock-free) i nieczekających (ang. wait-free) struktur danych
- Różnią się *mocą* – stopniem przydatności

Różnice

- Przedstawione operacje RMW są przydatne do implementacji:
  - algorytmów WW
  - niewstrzymywanych (ang. lock-free) i nieczekających (ang. wait-free) struktur danych
- Różnią się *mocą* – stopniem przydatności
- Jako miarę „mocy” operacji RMW można użyć liczbę konsensusu

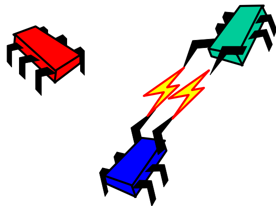
Problem konsensusu

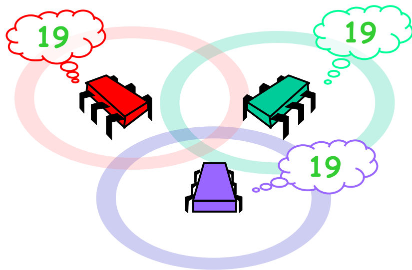


Rysunek 1: Na podstawie: "Art of Multiprocessor Programming", M. Herlihy i N. Shavit (CC BY-SA 3.0)

Każdy wątek ma swoją wartość, którą chce wpisać do rejestru.

Problem konsensusu





Wątki komunikują się...

I godzą się na jedną z zaproponowanych wartości

Konsensus – formalnie

W problemie konsensusu wątki (procesy) podejmują decyzję, która jest:

- spójna – wszystkie wątki godzą się na tę samą wartość
- poprawna – ustalona wartość jest wartością zaproponowaną przez któryś z wątków (a nie np. połączeniem wartości)

Protokół konsensusu

- Interesuje nas rozwiązanie problemu konsensusu, które jest nieczekające (ang. wait-free), tj. każde wykonanie wymaga skończonej liczby kroków – nie może się dowolnie przedłużać, bez WW
- Rozwiązanie takie nazywane jest protokołem konsensusu
- Protokół konsensusu implementowany jest za pomocą *obiektów konsensusu*

Obiekt konsensusu

- Protokół konsensusu jest realizowany przez obiekt konsensusu

```
public interface Consensus<T> {
    T decide(T value);
}
abstract class ConsensusProtocol<T>
    implements Consensus<T>{
    protected T[] proposed = new T[N]; // tab. na proponowane wartości
    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }
    abstract public T decide(T value);
}
```

Liczba konsensusu

- Mówimy, że *klasa* obiektów konsensusu ma liczbę konsensusu równą n, jeżeli pozwala na rozwiązanie problemu konsensusu dla n wątków
- Przykładowo, klasa obiektów o liczbie konsensusu równej 2 pozwala na rozwiązanie problemu konsensusu dla pary wątków

Liczba konsensusu

Tw. Liczba konsensusu rejestrów atomowych typu odczyt-zapis (RW)

Nie istnieje nieczekająca implementacja protokołu konsensusu dla n wątków, $n > 1$, za pomocą atomowych rejestrów RW

Liczba konsensusu

Tw. Liczba konsensusu rejestrów atomowych typu odczyt-zapis (RW)

Nie istnieje nieczekająca implementacja protokołu konsensusu dla n wątków, $n > 1$, za pomocą atomowych rejestrów RW

- Czyli, korzystanie tylko z atomowych operacji odczytu i zapisu zmiennych (rejestrów) nie wystarczy do implementacji *nieczekającego* protokołu konsensusu, nawet dla pary wątków
- Musielibyśmy zastosować algorytm z *oczekiwaniem* (zawierający pętle itp.)
- Mocniejsze operacje (sprzętowe) z większą liczbą konsensusu są koniecznością, aby możliwa była implementacja nieczekających struktur danych

Przykładowe liczby konsensusu

- 1: rejestry RW
- 2:
  - kolejka FIFO
  - rejestry RMW tylko z operacjami `getAndSet()`, `getAndIncrement()`, `getAndAdd()`
- ...
- ∞: rejestr RMW z operacją `compareAndSet()` (CAS)

Problem konsensusu za pomocą CAS

```java
AtomicLong czyjaKolej = new AtomicLong(0);
...
long id = Thread.currentThread().getId(); // > 0
// każdy wątek "proponuje" swój identyfikator
if (czyjaKolej.compareAndSet(0, id) == true) {
    System.out.println("Wygrałem");
}
```

Powrót do problemu WW

- Rejestry RMW z liczbami konsensusu większymi od 1 można zastosować do rozwiązania problemu WW
- Przykładem takiej operacji jest `testAndSet()`

```
1 public abstract class RMWRegister {
2    private boolean value;
3
4    public int synchronized testAndSet(boolean v) {
5        boolean prior  = this.value;
6        this.value = v; // Nowa wartość rejestru
7        return prior;   // Zwróć poprzednią
8    }
9 }
```

Blokada TAS

Przykładowa implementacja blokady Tast-and-set (TAS) w języku Java:

```
1 public class TASLock implements Lock {
2     AtomicBoolean taken = new AtomicBoolean(false);
3     public void lock() {
4         while( taken.testAndSet(true) ) { }
5     }
6     public void unlock() {
7         taken.set(false);
8     }
9 }
```

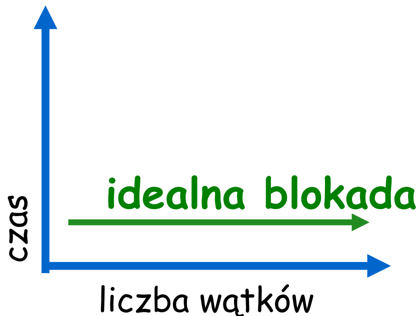
Blokada TAS

Przykładowa implementacja blokady Tast-and-set (TAS) w języku Java:

```
1 public class TASLock implements Lock {
2     AtomicBoolean taken = new AtomicBoolean(false);
3     public void lock() {
4         while( taken.testAndSet(true) ) { }
5     }
6     public void unlock() {
7         taken.set(false);
8     }
9 }
```

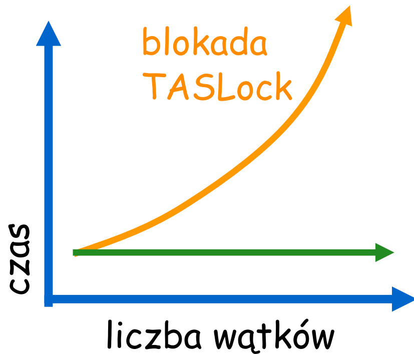
- `taken.testAndSet(true)` wykonuje się „natychmiastowo” dla dowolnej liczby wątków
- oczywiście, sam algorytm WW wymaga czekania, ponieważ dwa wątki nie mogą jednocześnie wykonywać sekcji krytycznej
- Zalety:
    - Prosta w implementacji
    - Tylko 1 rejestr RMW (`AtomicBoolean taken`)

A co z wydajnością?



- Chcielibyśmy mieć taką blokadę, aby czas wykonywania protokołów wstępnego i końcowego nie zależał od liczby wątków
  - brak dodatkowego narzutu czasowego
  - oprócz, oczywiście, koniecznego oczekiwania na zwolnienie sekcji krytycznej przez inny wątek

W praktyce



Wersja 2. – TTASLock

Wersja Test-and-test-and-set

```
class TTASlock {
    AtomicBoolean taken = new AtomicBoolean(false);

    void lock() {
        while (true) {
            while (taken.get()) {
            }
            if (taken.getAndSet(true) == false) {
                return ;
            }
        }
    }
    ...
}
```

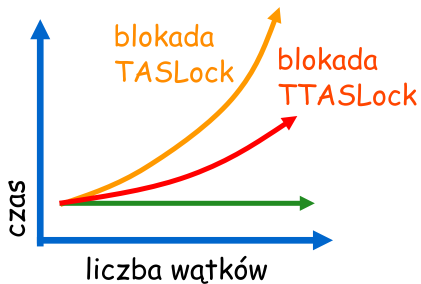
Wersja 2. – TTASLock

Wersja Test-and-test-and-set

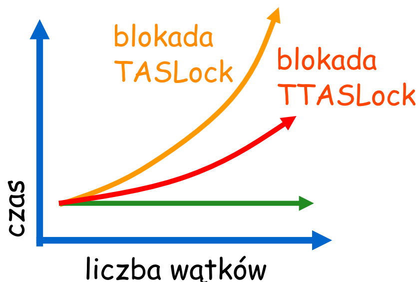
```
1  class TTASlock {
2      AtomicBoolean taken = new AtomicBoolean(false);
3  
4      void lock() {
5          while (true) {
6              while (taken.get()) {
7              }
8              if (taken.getAndSet(true) == false) {
9                  return ;
10             }
11         }
12     }
13     ...
14 }
```

- Jak widać kod bardziej *złożony*, niż blokady TASLock
- Obie są poprawne i są *logicznie* równoważne

W praktyce

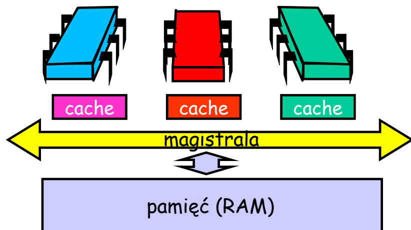


W praktyce



- Różnica w wydajności algorytmów wynika z różnego sposobu wykorzystania pamięci podręcznej (ang. *cache memory*)

Organizacja dostępu do pamięci



- We współczesnych komputerach procesory nie odczytują / zapisują danych bezpośrednio do pamięci głównej
- Odczyt / zapis za pośrednictwem pamięci podręcznych (ang. *cache memory*)
- W praktyce – pamięć podręczna może mieć kilka poziomów (L1, L2, L3)

Opóźnienia

Tabela 1: Przykładowe czasy wykonania różnych operacji wg Jeffa Deana

Operacja	Czas
Odczyt słowa z pamięci L1	0.5 ns
Odczyt słowa z pamięci L2	7 ns
Odczyt z pamięci głównej	100 ns
Założenie/zwolnienie blokady (ang. *mutex lock/unlock*)	100 ns
Przesłanie 2KB po sieci 1Gbps	20,000 ns
Sekwencyjny odczyt 1 MB z pamięci	250,000 ns
Ustawienie głowicy HDD	10,000,000 ns
Sekwencyjny odczyt 1 MB z HDD	30,000,000 ns

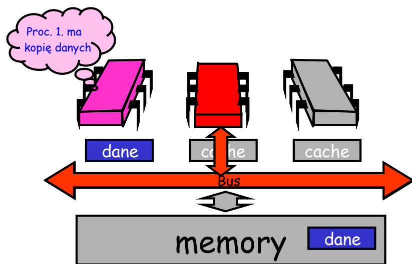
- Czasy te będą różne dla różnych modeli procesorów, zegarów taktowania itp. – chodzi tylko o *rząd wielkości*

Dostęp do pamięci

- Jeżeli procesor próbuje odczytać wartość danej, to najpierw sprawdza, czy dana ta znajduje się w pamięci podręcznej
  - tak, to nie ma potrzeby odwoływania się do pamięci wyższego poziomu – trafienie (ang. *cache hit*)
  - nie – nastąpiło chybienie (ang. *cache miss*) dana musi zostać pobrana z pamięci głównej
- Ten schemat jest uproszczony w stosunku do realnych architektur
- W przypadku architektur wieloprocesorowych (wielordzeniowych) sytuacja się komplikuje

Spójność pamięci podręcznej

- Dane mogą znajdować się zarówno w pamięci głównej, jak i pamięciach podręcznych różnych procesorów

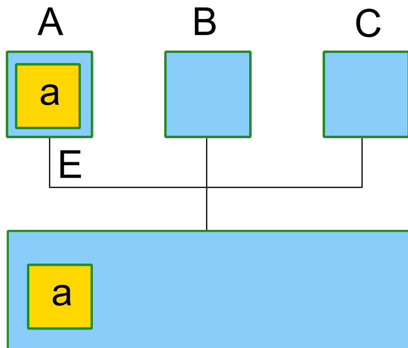


- Konieczne jest zapewnienie spójności pamięci podręcznych (ang. *cache coherence*)

Protokół MESI

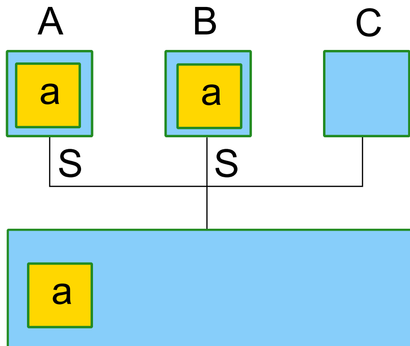
- Jeden z prostszych, ale praktycznych protokołów zapewnienia spójności pamięci podręcznych
- Z każdym wierszem pamięci podręcznej związane są dodatkowe bity *stanu*:
  - Modified – wiersz pamięci został zmodyfikowany i musi zostać zapisany do pam. głównej
  - Exclusive – wiersz nie jest zmodyfikowany i żaden inny procesor go nie buforuje
  - Shared – wiersz nie jest zmodyfikowany, ale znajduje się w kilku pamięciach podręcznych
  - Invalid – wiersz nie zawiera istotnych danych

MESI



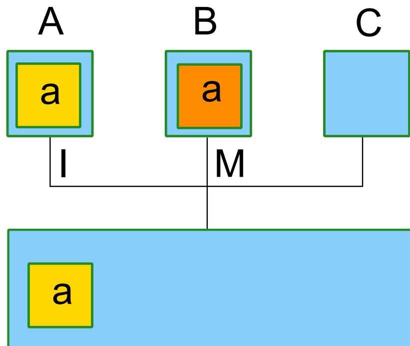
- Procesor A odczytuje dane spod adresu *a* i zapisuje w swojej pamięci podręcznej w stanie *wyłączności* (ang. *exclusive*, E)

MESI



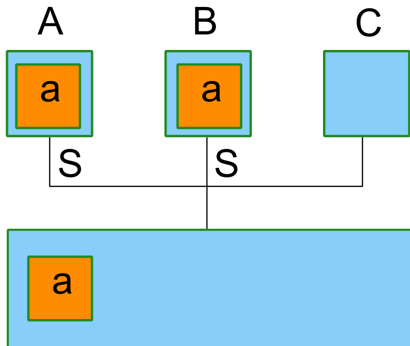
- Procesor B żąda danych spod adresu *a*
- Kontroler pamięci podręcznej proc. A wykrywa żądanie i odpowiada aktualną wartością danych
- Obie kopie danych są w stanie *współdzielenia* (ang. *shared*, S)

MESI



- Procesor B modyfikuje dane spod adresu *a*
- Rozgłasza do pozostałych informację, że dokonał modyfikacji – jego kopia danych znajduje się w stanie *zmodyfikowanym* (ang. *modified*, M)
- Pozostałe procesory *unieważniają* swoje kopie (ang. *invalid*, I)

MESI



- Próba odczytu danych przez proc. A wymusza wysłanie przez procesor B aktualnej wartości danej do pozostałych procesorów oraz pamięci głównej
- Kopie znajdują się ponownie w stanie *współdzielonym*

Opóźnienie zapisu

- Modyfikacja zmiennej nie wymusza natychmiastowego wysłania aktualnej wartości do pozostałych procesorów i pamięci podręcznej
- Efektywniejsze jest *buforowanie* zapisu, tak by zapisywać większe porcje danych
  - Buforowanie zapisu nie „psuje" spójności pamięci
  - Nie ma to nic wspólnego ze słowem `volatile` w Javie
- Procesory korzystają z jednej współdzielonej szyny pamięci
  - tylko jeden przesyła dane w danym momencie
  - potencjalne wąskie gardło

TASLock a TTASLock

```java
public class TASLock implements Lock
      {
    public void lock() {
        while( taken.testAndSet(true
     ) ) { }
    }
}
```

```java
class TTASlock {
    void lock() {
        while (true) {
            while (taken.get()) {
            }
            if (taken.getAndSet(true
    ) == false) {
                return ;
            }
        }
    }
}
```

TASLock a TTASLock

```
public class TASLock implements Lock {
    public void lock() {
        while( taken.testAndSet(true ) ) { }
    }
}
```

```
class TTASlock {
    void lock() {
        while (true) {
            while (taken.get()) {
            }
            if (taken.getAndSet(true ) == false) {
                return ;
            }
        }
    }
}
```

- Wątki oczekując unieważniają za każdym razem wartość zmiennej `taken`
- Wymaga dostępu do szyny pamięci, by rozgłosić modyfikację zmiennej do pozostałych procesorów / rdzeni
- Wątek, który chce zwolnić blokadę opóźniany przez wątki czekające

TASLock a TTASLock

```
public class TASLock implements Lock {
    public void lock() {
        while( taken.testAndSet(true) ) { }
    }
}
```

- Wątki oczekując unieważniają za każdym razem wartość zmiennej `taken`
- Wymaga dostępu do szyny pamięci, by rozgłosić modyfikację zmiennej do pozostałych procesorów / rdzeni
- Wątek, który chce zwolnić blokadę opóźniany przez wątki czekające

```
class TTASlock {
    void lock() {
        while (true) {
            while (taken.get()) {
            }
            if (taken.getAndSet(true) == false) {
                return ;
            }
        }
    }
}
```

- Wątki oczekujące nie korzystają ze współdzielonej szyny pamięci – mają aktualną wartość zmiennej `taken` w pamięci podręcznej
- Zwolnienie blokady – unieważnienie wszystkich kopii

Wady protokołów z aktywnym oczekiwaniem

- Przedstawione dotychczas algorytmy wzajemnego wykluczania implementują **aktywne oczekiwanie** – nieustanne sprawdzanie warunku w pętli `while`
  - Blokady implementowane za pomocą tych algorytmów nazywane są blokadami wirującymi (ang. spin lock)
  - Wątek zużywa czas procesora nie wykonując żadnych „produktywnych" operacji

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?
- To zależy:

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?
- To zależy:
  - Jeżeli wątków jest mniej, niż procesorów i każdy wykonuje się na odrębnym procesorze, to możemy pozwolić sobie na stratę
  - Jeżeli blokada zakładana jest na bardzo *krótki* czas i tylko niewielka liczba wątków chce ją założyć (niska rywalizacja), to blokady wirujące mogą być dobrym pomysłem

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?
- To zależy:
  - Jeżeli wątków jest mniej, niż procesorów i każdy wykonuje się na odrębnym procesorze, to możemy pozwolić sobie na stratę
  - Jeżeli blokada zakładana jest na bardzo *krótki* czas i tylko niewielka liczba wątków chce ją założyć (niska rywalizacja), to blokady wirujące mogą być dobrym pomysłem
- Alternatywa – *uśpienie* wątku oczekującego na zwolnienie blokady

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?
- To zależy:
  - Jeżeli wątków jest mniej, niż procesorów i każdy wykonuje się na odrębnym procesorze, to możemy pozwolić sobie na stratę
  - Jeżeli blokada zakładana jest na bardzo *krótki* czas i tylko niewielka liczba wątków chce ją założyć (niska rywalizacja), to blokady wirujące mogą być dobrym pomysłem
- Alternatywa – *uśpienie* wątku oczekującego na zwolnienie blokady
- Trzeba wziąć pod uwagę czas przełączania kontekstu – z uśpionego wątku na inny

Wady protokołów z aktywnym oczekiwaniem

- Czy powinno się unikać blokad z aktywnym oczekiwaniem?
- To zależy:
  - Jeżeli wątków jest mniej, niż procesorów i każdy wykonuje się na odrębnym procesorze, to możemy pozwolić sobie na stratę
  - Jeżeli blokada zakładana jest na bardzo *krótki* czas i tylko niewielka liczba wątków chce ją założyć (niska rywalizacja), to blokady wirujące mogą być dobrym pomysłem
- Alternatywa – *uśpienie* wątku oczekującego na zwolnienie blokady
- Trzeba wziąć pod uwagę czas przełączania kontekstu – z uśpionego wątku na inny
- Rozwiązania hybrydowe – „chwila” aktywnego oczekiwania, potem uśpienie

Protokół z wycofywaniem

- W alg. TTASLock może się zdarzyć, że wątek zobaczy wartość zmiennej `taken = false`, a więc może założyć blokadę
- Ale próba założenia kończy się porażką

Protokół z wycofywaniem

- W alg. TTASLock może się zdarzyć, że wątek zobaczy wartość zmiennej `taken = false`, a więc może założyć blokadę
- Ale próba założenia kończy się porażką
- Oznacza, to że inny wątek (rywal) go ubiegł
- Zjawisko dużej rywalizacji (ang. *contention*) jest niekorzystne

Protokół z wycofywaniem

- W alg. TTASLock może się zdarzyć, że wątek zobaczy wartość zmiennej `taken = false`, a więc może założyć blokadę
- Ale próba założenia kończy się porażką
- Oznacza, to że inny wątek (rywal) go ubiegł
- Zjawisko dużej rywalizacji (ang. *contention*) jest niekorzystne
- Rozwiązanie – wątek jest usypiany na *pewien* czas, dając szansę pozostałym wątkom na założenie blokady i zmniejszając tym samym rywalizację

Protokół z wycofywaniem

- Dokładniej, wątek usypiany jest na okres czasu o losowej długości
- Ponawia próbę założenia blokady
- W przypadku porażki okres czasu jest wydłużany dwukrotnie (ale, do pewnych granic)
- Algorytm taki nazywany jest blokadą z *wykładniczym wycofywaniem* (ang. *exponential backoff*)

Protokół z wycofywaniem

```
1  public class BackoffLock implements Lock {
2      ...
3      public void lock() {
4          int delay = MIN_DELAY;
5          while (true) {
6              while (taken.get()) {
7              }
8              if (!lock.getAndSet(true))
9                  return;
10             Thread.sleep(random() % delay);
11             if (delay < MAX_DELAY) {
12                 delay = 2 * delay;
13             }
14         }
15     }
16 }
```

Protokół z wycofywaniem – uwagi

- Lepsza wydajność, niż poprzednich blokad w przypadku dużej rywalizacji
- Łatwa implementacja
- Problem – jak dobrać wartości stałych `MIN_DELAY` i `MAX_DELAY`
- W praktyce stosowane są bardziej złożone algorytmy blokad:
  - kolejkowe
  - hierarchiczne
  - kompozycyjne – hybrydowe

Algorytmy nieczekające

Algorytm nieczekający

Algorytm nazywamy *nieczekającym* (ang. wait-free), jeżeli *każde* jego wykonanie przez współbieżne wątki zostanie ukończone w *skończonej* liczbie kroków

Algorytmy nieczekające

Algorytm nieczekający

Algorytm nazywamy *nieczekającym* (ang. wait-free), jeżeli *każde* jego wykonanie przez współbieżne wątki zostanie ukończone w *skończonej* liczbie kroków

- Jest to silniejsza własność niż *niewstrzymywanie*
- Gwarantuje, że każdy wątek robi postępy w obliczeniach – brak zakleszczenia i zagłodzenia
- Wada – trudne w implementacji i niestety często mniej efektywne niż alg. niewstrzymywane

Algorytmy niewstrzymywane

Algorytm niewstrzymywany

Algorytm jest *niewstrzymywany* (ang. *lock-free*), jeżeli gwarantuje, że nieskończenie często jakieś jego wykonanie zostanie ukończone w skończonej liczbie kroków

Algorytmy niewstrzymywane

Algorytm niewstrzymywany

Algorytm jest *niewstrzymywany* (ang. *lock-free*), jeżeli gwarantuje, że nieskończenie często jakieś jego wykonanie zostanie ukończone w skończonej liczbie kroków

- Dzięki alg. niewstrzymywanemu system jako całość wykonuje kolejne obliczenia, choć być może, nie wszystkie wątki „liczą"
- Innymi słowy, alg. niewstrzymywany nie musi być sprawiedliwy i któreś z wątków mogą być *głodzone* kosztem pozostałych
- Wyklucza możliwość zakleszczenia
- Algorytmy nieczekające są niewstrzymywane, ale nie odwrotnie

Algorytmy niehamowane

Algorytm niehamowany

Algorytm nazywamy *niehamowanym* (ang. *obstruction free*), jeżeli w dowolnym momencie po rozpoczęciu jego wykonywania w izolacji zostanie zakończona w skończonej liczbie kroków.

Algorytmy niehamowane

Algorytm niehamowany

Algorytm nazywamy *niehamowanym* (ang. *obstruction free*), jeżeli w dowolnym momencie po rozpoczęciu jego wykonywania w izolacji zostanie zakończona w skończonej liczbie kroków.

- Innymi słowy, jeżeli pozostałe wątki zostaną zatrzymane, to dany wątek zakończy wykonywanie w skończonej liczbie kroków
- Wyklucza użycie blokad, ale nie gwarantuje postępu obliczeń
- Metody optymistycznej kontroli współbieżności są zazwyczaj niehamowane – wykrycie konfliktu z innymi wątkami powoduje, że wątek się *wycofuje*
- Wszystkie algorytmy nieczekające i niewstrzymywane są również niehamowane, ale nie odwrotnie

Mechanizmy zamków (blokad) w Javie

- Pakiet `java.util.concurrent.locks`
- Interfejs `Lock`
- Implementacje: `ReentrantLock`, `ReentrantReadWriteLock`
- Konieczność stosowania tych samych zamków dla tych samych sekcji krytycznych i w tej samej kolejności
- Sekcje krytyczne niepowiązane ze sobą (zmienne nie zależą od zmiennych w innej sekcji bezpośrednio i pośrednio) mogą być realizowane współbieżnie jeżeli będą chronione odrębnymi zamkami

Lock c.d.

Interfejs Lock udostępnia również metodę `tryLock()`, która próbuje uzyskać zamek, ale jeżeli się to nie uda, to nie wstrzymuje wykonania wątku

```
Lock lock = ...;
if (lock.tryLock()) {
  try {
      // sekcja krytyczna
  } finally {
      lock.unlock();
  }
} else {
    // alternatywne operacje, sekcja lokalna
}
```

ReentrantLock

- Podstawowa implementacja interfejsu `Lock`
- Wydajność podobna do bloków / metod synchronizowanych za pomocą konstrukcji `synchronized`
- Wątek, który założył blokadę może ją założyć ponownie
- Dzięki konstruktorowi `ReentrantLock(boolean fair)` można stworzyć blokadę z gwarancją braku zagłodzenia i obsługą oczekujących wątków zgodnie z kolejnością zgłaszania żądań
- Można sprawdzić, czy i jakie wątki czekają na zwolnienie blokady

Interfejs ReadWriteLock

- `ReadWriteLock` zawiera parę skojarzonych blokad:
  - jedna tylko dla operacji odczytu
  - druga dla operacji odczytu i zapisu
- Przydatny, gdy dane współdzielone są często odczytywane, ale rzadko modyfikowane
  - wątki, które tylko czytają zakładają blokadę do odczytu
  - wątki modyfikujące zakładają blokadę do zapisu
- Implementacja jest bardziej złożona, niż w przypadku `ReentrantLock` – trzeba sprawdzić wydajność w praktyce

Interfejs ReadWriteLock – przykład

```
1  class CachedData {
2    Object data;
3    volatile boolean cacheValid;
4    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
5    void processCachedData() {
6      rwl.readLock().lock();
7      if (!cacheValid) {
8         rwl.readLock().unlock(); // zwolnij blokadę do odczytu
9         rwl.writeLock().lock(); // załóż do zapisu
10        if (!cacheValid) { // sprawdź ponownie
11          data = ...
12          cacheValid = true;
13        }
14        rwl.readLock().lock();  // załóż blokadę do odczytu
15        rwl.writeLock().unlock(); // zwolnij blokadę do zapisu
16     }
17     use(data); // chronione tylko blokadą do odczytu
18     rwl.readLock().unlock();
19   }
20 }
```