

Programowanie wielowątkowe w Javie - wybrane zagadnienia

2

Java Memory Model

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- określa sposób dostępu i wykorzystania rzeczywistej pamięci RAM m.in. mówi o tym jak i kiedy różne wątki mają dostęp do wartości zapisywanych przez wątki

3

Internal Java Memory Model - Stack

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- każdy wątek ma własny stos zawierający informacje o wywołanych metodach (call stack) oraz o zmiennych lokalnych utworzonych w ramach ich wywołań
- wątek ma dostęp wyłącznie do własnych zmiennych (nawet jeśli różne wątki uruchamiają ten sam kod)

4

Internal Java Memory Model - Heap

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- sarta zawiera wszystkie obiekty utworzone w ramach aplikacji (niezależnie od tego jaki wątek je utworzył lub tego czy jest to zmienna instancyjna czy lokalna dla metody)
- sarta jest dostępna dla wszystkich wątków (mogą współdzielić dostęp do obiektów i ich zmiennych instancyjnych oraz statycznych, ale nie lokalnych)

5

Internal Java Memory Model - lokalizacja w pamięci

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- obiekty zapisywane są na stercie
- lokalne zmienne typów prymitywnych i referencje są przechowywane na stosie i kopiowane w czasie przekazywania do innego wątku (każdy działa na własnej kopii)
- zmienne instancyjne oraz zmienne statyczne przechowywane są na stercie wraz z obiektem (nawet jeśli są one prymitywne) i definicją klasy

6

Klasyczny, sprzętowy model pamięci

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- komputery mogą posiadać więcej niż jeden procesor, z których każdy może mieć wiele rdzeni co umożliwia realne działanie wielu wątków
- każdy procesor zawiera zbiór rejestrów (pamięć CPU) oraz własną pamięć podręczną (często wielopoziomową)
- operacje na rejestrach są dużo bardziej wydajne niż operacje na pamięci głównej (RAM)
- operacje na pamięci podręcznej CPU są wolniejsze niż na rejestrach, ale szybsze niż na pamięci głównej (RAM)
- dostęp do danych w RAM odbywa się najczęściej pośrednio (przez pamięć cache i rejestry)

7

Sprzętowy model pamięci vs. Java Memory Model

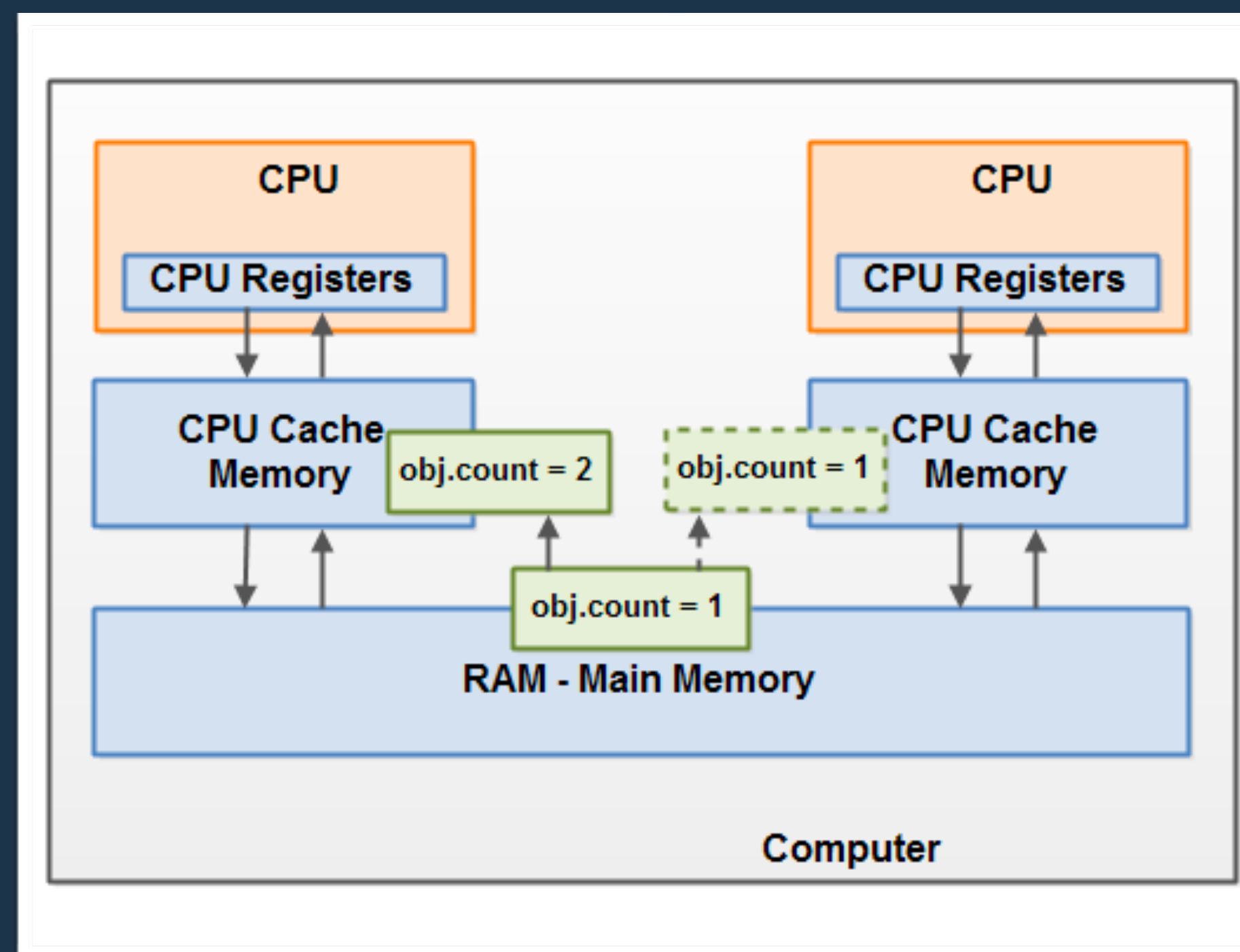
Programowanie wielowątkowe w Javie - wybrane zagadnienia

- sprzętowy model pamięci nie rozróżnia stosu i sterty (oba obszary są częścią RAM)
- fragmenty stosu i sterty mogą być przechowywane na poziomie rejestrów i pamięci podręcznej CPU
- przechowywanie zmiennych i obiektów w różnych obszarach pamięci może prowadzić do problemów
 - widoczność zmian stanu współdzielonych zmiennych
 - race condition przy odczycie, wykorzystaniu i zapisie współdzielonych zmiennych

8

Problem widoczności współdzielonych zmiennych

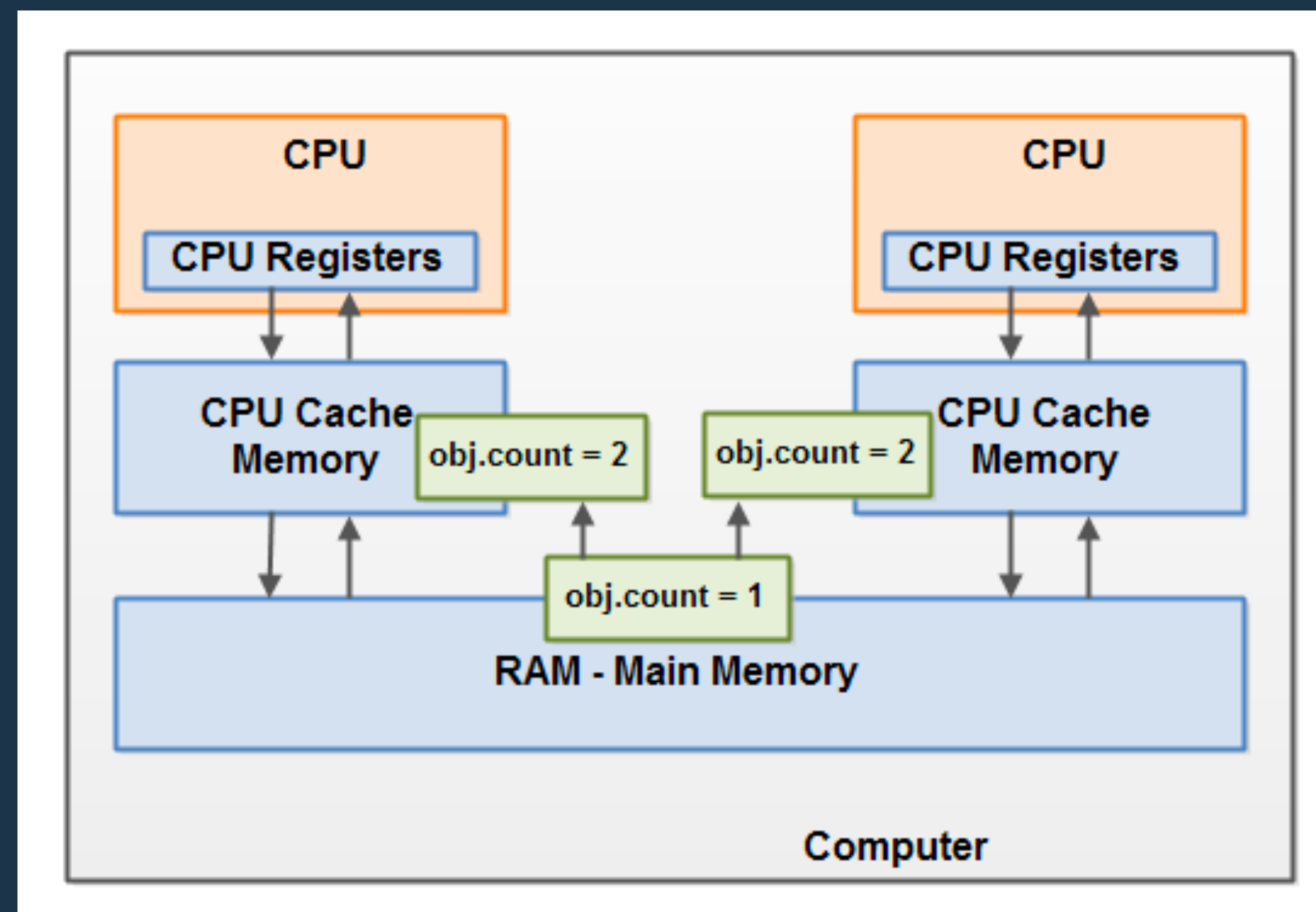
Programowanie wielowątkowe w Javie - wybrane zagadnienia



9

Problem race condition

Programowanie wielowątkowe w Javie - wybrane zagadnienia



- jeśli jeden wątek zmienia wartość zmiennej **balance**, a drugi dokonuje jej odczytu, **volatile** gwarantuje spójność (widoczność zmian). W przypadku zmiany wartości przez większą ilość wątków wymagana jest synchronizacja

```
public class Account {  
  
    volatile int balance = 0;  
  
}
```

- jeśli jeden wątek zmienia wartość zmiennej **balance**, a drugi dokonuje jej odczytu, **volatile** gwarantuje spójność (widoczność zmian). W przypadku zmiany wartości przez większą ilość wątków wymagana jest synchronizacja

```
public class Account {  
  
    volatile int balance = 0;  
  
}
```

- jeśli wątek A pisze do zmiennej oznaczonej **volatile**, a wątek B ją czyta, wtedy wszystkie zmienne widoczne dla wątku A przed zapisem będą także widoczne dla wątku B (synchronizacja z „pamięcią główną“)
- jeśli wątek A czyta zmienną oznaczoną **volatile**, wtedy wszystkie zmienne widoczne dla wątku A będą odświeżone z „pamięci głównej”

Słowo kluczowe **volatile** vs. synchronizacja

Programowanie wielowątkowe w Javie - wybrane zagadnienia

- **volatile** może być stosowane na poziomie zmiennych prymitywnych, synchronizacja (**synchronized**, typy atomowe) może odnosić się jedynie do typów referencyjnych
- **volatile** nie jest związane z blokadą (zamkiem), w związku z tym nie nadaje się do operacji typu atomic read-update-write
- zmienna volatile może mieć wartość **null** - synchronizacja zmian jest realizowana na poziomie referencji

- **Mutual Exclusion** - tylko jeden wątek może wykonywać sekcję krytyczną
- **Visibility** - zmiany realizowane przez jeden wątek są widoczne dla innych wątków
- synchronizacja z użyciem blokad spełnia oba warunki kosztem wydajności
- volatile - spełnia tylko drugi warunek, koszty wydajnościowe są mniejsze ale nadal występują (praca na „pamięci głównej”, brak wybranych optymalizacji)

- Java gwarantuje atomowość operacji odczytu i zapisu zmiennych dla typów o rozmiarze do 32-bitów
- w przypadku typów 64-bitowych, takich jak **long** odczyt/zapis zmiennej może zostać podzielony na dwie operacje 32-bitowe, co z kolei może wpłynąć na wynik działania programu współbieżnego
- zastosowanie modyfikatora **volatile** gwarantuje, że do tego nie dojdzie

- JVM oraz CPU mogą dokonywać zmian w kolejności wykonywania instrukcji kodu w celu jego optymalizacji (jeśli ich sens/semantyka nie ulega zmianie)
- w celu zagwarantowania bezpieczeństwa tych optymalizacji Java stosuje zasadę **happens-before**
- zapis/odczyt do zmiennych nie może być przeniesiony tak aby następował po zapisie do zmiennej **volatile** jeśli wcześniej występował przed nim (co nie wyklucza sytuacji odwrotnej)
- zapis/odczyt do zmiennych nie może być przeniesiony tak aby następował przed odczytem zmiennej **volatile** jeśli wcześniej występował po nim (co nie wyklucza sytuacji odwrotnej)

- scheduler w Javie (menadżer wątków) przydziela zasoby procesora bazując na priorytecie wątku stosując jednocześnie **time slicing**
- każdy wątek w Javie posiada priorytet w zakresie 1-10 (standardowo 5)
- każdy nowo utworzony wątek dziedziczy priorytet po wątku który go stworzył
- priorytet wątku może zostać zmieniony programistycznie w dowolnym momencie działania programu
- priorytety wątków w Javie są mapowane na priorytety w systemie operacyjnym, których gradacja może się różnić

- zachodzi kiedy wątek o niższym priorytecie blokuje (synchronizacja, semafony itd.) dostęp do zasobu potrzebnego do działania wątku o wyższym priorytecie przy jednoczesnej „walce” o zasoby procesora z wątkami o średnim priorytecie

Priority Inversion, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing critical section with lower priority task (L) and L is already in critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion

- tymczasowe podniesienie priorytetu wątku o niskim priorytecie, na czas wykonywania sekcji krytycznej tak, aby wątki o średnim priorytecie nie powodowały dodatkowego opóźnienia jej realizacji

Priority Inheritance, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of critical section.

- Disabling interrupts
- No blocking
- Random priority boosting

- **Mark-Sweep** - oznaczanie obszarów zajmowanych przez nieosiągalne obiekty jako gotowych do alokacji
- **Mark-Sweep-Compact** - jak wyżej, z dodatkowym kompaktowaniem sterty
- **Mark-Copy** - kopiowanie osiągalnych obiektów do nowego miejsca na sterpie

- JVM oferuje standardowo 4 mechanizmy działania Garbage Collectora
 - Serial
 - Paraller
 - Mostly-Concurrent-Mark-And-Sweep (CMS)
 - G1

- stosuje strategię Mark-Copy przy czyszczeniu młodej generacji oraz Mark-Sweep-Compact dla generacji starej
- czyszczenie starej generacji poprzedzone jest zawsze czyszczeniem młodej generacji (full gc)
- całość wykonywana jest na jednym wątku i powoduje wstrzymanie całej aplikacji
- sprawdza się dla aplikacji z niewielką stertą

- działa jak Serial z tą różnicą że wykorzystuje wiele wątków
- czyszczenie starej generacji poprzedzone jest zawsze czyszczeniem młodej generacji (full gc)
- pozwala określić cele - maksymalny czas zatrzymania aplikacji, stosunek czasu pracy gc do pracy aplikacji oraz jaki powinien być max. rozmiar sterty
- sprawdza się dla aplikacji szybko zapełniających stertę, dla których możliwe są krótkie, sporadyczne przerwy w działaniu

- minimalizuje czas trwania pauz w działaniu programu kosztem użycia procesora
- stosuje strategię Mark-Copy przy czyszczeniu młodej generacji
- dla starej generacji wykonywane są następujące etapy
 - **Initial Mark** - wstrzymuje działanie aplikacji, identyfikuje GC roots pozwalających na rozpoczęcie procesu oznaczania żywych obiektów
 - **Concurrent Mark** - przeszukiwanie wcześniej oznaczonych drzew i oznaczanie obiektów jako żywych
 - **Concurrent Preclean** - przeszukiwanie drzew referencji w obiektach które uległy zmianie od czasu poprzedniej fazy
 - **Concurrent Abortable Preclean** - wydłużenie poprzedniej fazy, aby rozpocząć kolejną mniej więcej w połowie czasu między kolejnymi czyszczeniami w edenie
 - **Final Remark** - wstrzymuje działanie aplikacji, ostateczne zlokalizowanie i oznaczenie żywych obiektów w starszej generacji
 - **Concurrent Sweep** - oznaczenie obszarów wolnych do ponownej alokacji

- lepsze wsparcie dla dużych stert (powyżej 4GB)
- wykorzystuje wiele wątków do skanowania sterty podzielonej wcześniej na mniejsze obszary w celu znalezienia wolnych wolnych obszarów (zwracanie dłużej ilości miejsca)

- większość kolekcji w języku Java nie gwarantuje bezpieczeństwa bez użycia zewnętrznej synchronizacji
- w przypadku prostych/pojedynczych operacji kolekcje można opakować stosując statyczne metody z `Collections` np. `synchronizedList()`, `synchronizedMap()`. W rezultacie dostęp do metod kolekcji jest synchronizowany (obiekt synchronizacji jest instancja kolekcji)
- w przypadku operacji złożonych takich jak jednoczesny odczyt i zapis należy użyć innych mechanizmów np. blokady jawne lub niejawne

- synchronizacja kolekcji może znacząco wpływać na wydajność realizowanych operacji (zwłaszcza jeśli są one złożone np. iteracja po wszystkich elementach)
- kolekcje współbieżne są bezpieczne wielowątkowo, ale jednocześnie nie ograniczają możliwości wykonywania operacji przez wiele wątków np.
- copy on write collections np. `CopyOnWriteArrayList`, `CopyOnWriteArraySet`
- compare and swap (CAS) np. `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`
- oparte o blokady jawne np. `ConcurrentHashMap`, `BlockingQueue`
- powyższe kolekcje nie rzucają wyjątkiem `ConcurrentModificationException`

- operacje związane obsługą UI powinny realizowane się w specjalnym wątku (event dispatch thread - EDT), który gwarantuje bezpieczeństwo w środowisku wielowątkowym
- operacje związane z logiką biznesową, przetwarzaniem powinny być realizowane w tle (worker threads), zwykle z użyciem **SwingWorker**

- nadmierna/niewłaściwa synchronizacja
- niewłaściwy sposób zarządzania wątkami (tworzenie, wykorzystanie, niszczenie)
- dobór niewłaściwych mechanizmów/implementacji w kontekście rozwiązywanego problemu
- nadmierna/niepoprawna/nieświadoma optymalizacja kodu
- niewłaściwe zarządzanie priorytetami