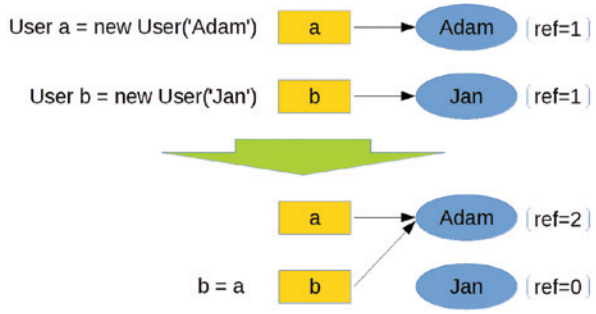
Jakub Kubryński

Co każdy programista Java powinien wiedzieć o JVM: zarządzanie pamięcią

Powód istnienia bohatera wielu javowych historii, nazywanego Odśmiecaczem Pamięci (Garbage Collector), jest zaskakująco prosty i oczywisty: nasze aplikacje do działania potrzebują pamięci, a ta z reguły ma ograniczony rozmiar i nie da się jej zapełniać w nieskończoność. A zarządzanie pamięcią jest na co dzień dla większości programistów znacznie mniej ciekawe niż zabawa nowymi frameworkami, w związku z czym zdecydowanie wolimy, aby pracami porządkowymi zajął się ktoś inny. I tę właśnie rolę strażnika porządku przejmuje od nas główna postać niniejszego artykułu.

RODZAJE ALGORYTMÓW

Algorytmy wykorzystywane przez Garbage Collector możemy podzielić na dwie zasadnicze grupy: skalarne i wektorowe. Sama technologia odśmiecania pamięci nie jest oczywiście wynalazkiem autorów Javy, a została opracowana już pod koniec lat pięćdziesiątych na potrzeby języka Lisp. Była to metoda skalarna bazująca na zliczaniu referencji. Jest to technologia stosunkowo prosta, aczkolwiek do dziś stosowana dość szeroko choćby w takich językach jak C++, Python czy PHP. W skrócie polega ona na tym, aby z każdym obiektem skojarzony był licznik wskazujących na niego odwołań (czyli referencji). Algorytm został obrazowo przedstawiony na Rysunku 1.



Rysunek 1. Zliczanie referencji

W pierwszym kroku tworzymy dwa obiekty – użytkowników „Adam" i „Jan", których przypisujemy odpowiednio do zmiennych „a" i „b". Jednocześnie związany z każdym z obiektów licznik referencji jest ustawiony na wartość 1. W drugim kroku natomiast modyfikujemy zmienną „b" tak, aby wskazywała na użytkownika „Adam". Konsekwencją wykonanej operacji jest zwiększenie licznika „Adama", na który teraz wskazują dwie zmienne, oraz wyzerowanie licznika referencji „Jana", dla którego brak odwołania za pośrednictwem jakiejkolwiek zmiennej skutkuje utratą widoczności obiektu, która to z kolei pozwala na jego usunięcie podczas procesu odśmiecania pamięci – skoro i tak nikt go nie zauważa, to można go usunąć bezstratnie dla naszej aplikacji. Jak widać, technika ta jest stosunkowo prosta, co nas, jako dociekliwych inżynierów, skłania do poszukiwania argumentu obalającego przydatność zliczania referencji i uzasadniającego potrzebę rozwinięcia artykułu. Jak to często bywa, przeczucie i tym razem nas nie zawiodło! Słabą stroną przedstawionego algorytmu jest brak możliwości wykrycia cyklicznych referencji, które mogą przecież w skrajnym wypadku skonsumować większość dostępnych zasobów pamięci. To właśnie było przyczyną bardziej zaawansowanych algorytmów wektorowych, traktujących alokowane obiekty jako grafy, a nie płaskie struktury.

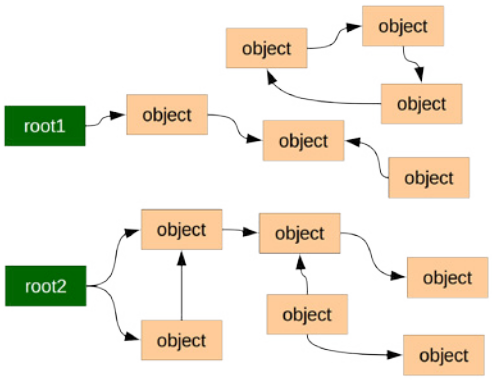
ALGORYTMY WEKTOROWE

Algorytmy wektorowe, określane także jako algorytmy śledzące, wykorzystują do analizy widoczności obiektów techniki grafowe. Na pierwszy rzut oka wygląda to bardzo prosto, gdyż wystarczy, przechodząc przez graf obiektów, zaznaczyć wszystkie, do których udało nam się dotrzeć, a całą resztę usunąć, wychodząc z założenia, że skoro my do nich nie dotarliśmy, to nie zrobi tego także aplikacja użytkownika. Sytuacja jednak komplikuje się, jak to często bywa, kiedy zaczynamy się zagłębiać w szczegóły. Pierwsze pytanie, jakie się nasuwa, to gdzie powinniśmy rozpocząć analizę grafu. Jeżeli rozpoczniemy od niewłaściwego obiektu, to cała wykonana analiza będzie zafałszowana, a w skrajnym wypadku moglibyśmy oznaczyć tylko i wyłącznie obiekty porzucone przez aplikację. Żeby do tego nie dopuścić, musimy zacząć od obiektów, które są na pewno „żywe". I właśnie takie obiekty, które niejako z definicji są wykorzystywane przez aplikację, nazywamy korzeniami (GC roots). Takim korzeniem może być na przykład uruchomiony wątek, zmienna lokalna czy też monitor wykorzystany do synchronizacji. Dzięki takiemu założeniu nasz algorytm strukturalizuje się do następującej postaci:

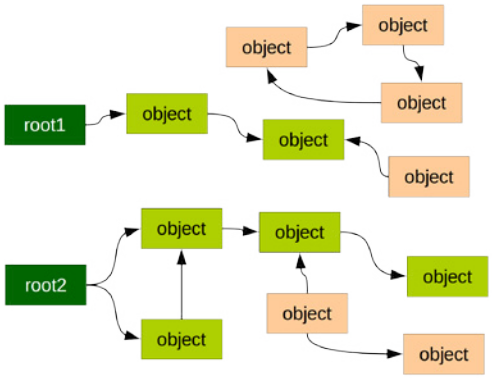
Listing 1. Pseudokod algorytmu śledzącego

```
mark_gc_roots()
for (each_root_object) {
  mark_all_referenced_objects()
}
for (each_object_in_memory) {
  if (is_marked_as_reacheable) {
    unmark_the_object_for_next_cycle()
  } else {
    remove_object_and_reclaim_memory()
  }
}
```

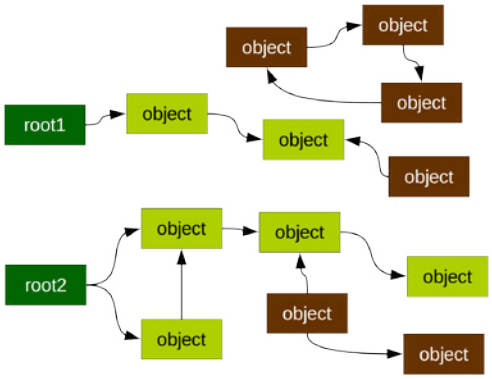
Teraz prześledzimy, jak krok po kroku przebiega to na prostym grafie obiektów. Rozpoczynamy od sytuacji, w której oznaczone zostały korzenie (Rysunek 2). Następnie od każdego z nich przechodzimy do należących do niego obiektów i oznaczamy je jako „żywe" (kolor zielony). Tę czynność powtarzamy dla każdego z właśnie oznaczonych obiektów tak długo, aż nie dotrzemy do końca gałęzi, czyli obiektu, który nie ma już żadnych składowych (Rysunek 3). Wszystkie pozostałe obiekty, niezależnie czy są z nimi powiązane referencje czy nie, są z punktu widzenia aplikacji nieużyteczne, ponieważ nie można się do nich dostać bezpośrednio lub nawet pośrednio z obiektów wykorzystywanych w chwili obecnej przez naszą aplikację (Rysunek 4). Ostatnim etapem (Rysunek 5) jest zwolnienie pamięci poprzez usunięcie „martwych" obiektów oraz wyczyszczenie markerów żyjących obiektów w celu przygotowania ich do kolejnego przebiegu algorytmu, który zostanie uruchomiony po ponownym zapełnieniu pamięci.



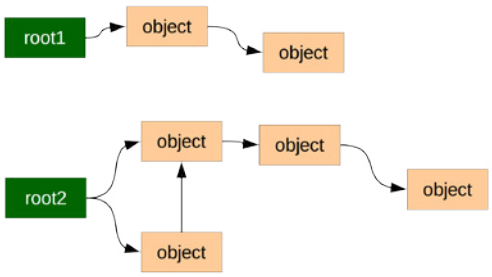
Rysunek 2. Algorytm wektorowy – oznaczenie korzeni



Rysunek 3. Algorytm wektorowy – żyjące obiekty



Rysunek 4. Algorytm wektorowy – obiekty niedostępne



Rysunek 5. Algorytm wektorowy – odzyskanie pamięci

Na początku tego akapitu wspomniałem, że przy dokładniejszej analizie sytuacja potrafi przyjąć bardziej złożoną postać niż ta, której się spodziewaliśmy. Pierwszym utrudnieniem było odpowiednie określenie punktów startowych dla algorytmu przeszukiwania, drugim natomiast jest towarzyszący nam dynamizm. Graf obiektów nieustannie się zmienia, co dodatkowo komplikuje jego analizę. Najprostszym sposobem poradzenia sobie z tym aspektem jest wykluczenie aktywności alokacji, które możemy uzyskać choćby poprzez czasowe zatrzymanie aplikacji. To właśnie te czasowe blokowania aplikacji, zwane także pauzami Garbage Collectora, spędzają nam sen z powiek, powodując niedeterministyczne zachowanie naszych systemów. Także autorzy JVM lwią część wysiłków poświęcają na optymalizację algorytmów, a w efekcie skracaniem czasu przestojów do minimum. Jedną z pierwszych linii obrony jest wdrożenie tzw. hipotezy generacyjnej.

HIPOTEZA GENERACYJNA

Jednymi z najważniejszych obserwacji poczynionych w ramach walki o skracanie czasu wymaganego przez algorytmy śledzące są hipotezy generacyjne. W Javie zastosowanie znalazła „słaba hipoteza generacyjna", która mówi:

- większość obiektów szybko staje się nieużyteczna z punktu widzenia aplikacji,
- odwołania starych obiektów do nowych są bardzo rzadkie.

Rysunek 6. Struktura sterty (heap)

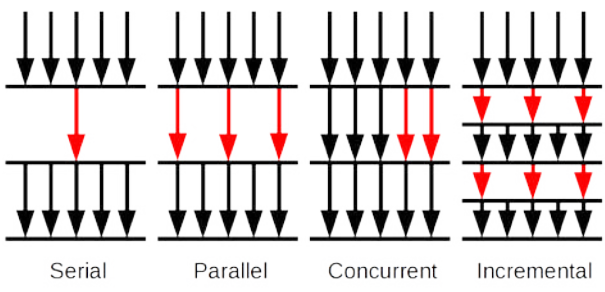
Wnioskiem nasuwającym się niemal automatycznie jest zatem podział przestrzeni dostępnej dla naszych obiektów na dwie części, przeznaczone kolejno dla nowych i starych obiektów. To pozwala nam na w miarę niezależne zarządzanie każdym z obszarów, co jest wyjątkowo istotne, jeżeli weźmiemy pod uwagę ich zgoła inną charakterystykę. Ta autonotmia obszarów może dotyczyć wielu płaszczyzn takich jak niezależne uruchamianie sprzątania pamięci czy też zastosowanie zupełnie innych algorytmów kolekcji. Strukturę segmentów pamięci przedstawiliśmy szczegółowo w pierwszym artykule tego cyklu (Programista 3/215) – cała sterta (heap) podzielona jest na dwie generacje – młodą (young) i starą (tenured, old). Co za tym idzie możemy wyróżnić kilka rodzajów kolekcji:

- małą kolekcję (minor GC) – obejmującą młodą generację,
- dużą kolekcję (major GC) – polegającą na sprzątaniu starej generacji,
- pełną kolekcję (full GC) – obejmującą całą stertę.

W dalszej części artykułu omówimy poszczególne algorytmy, które możemy wykorzystać dla wymienionych kolekcji.

RODZAJE ALGORYTMÓW

Algorytmy GC możemy podzielić na cztery kategorie w zależności od trybu ich wykonywania.

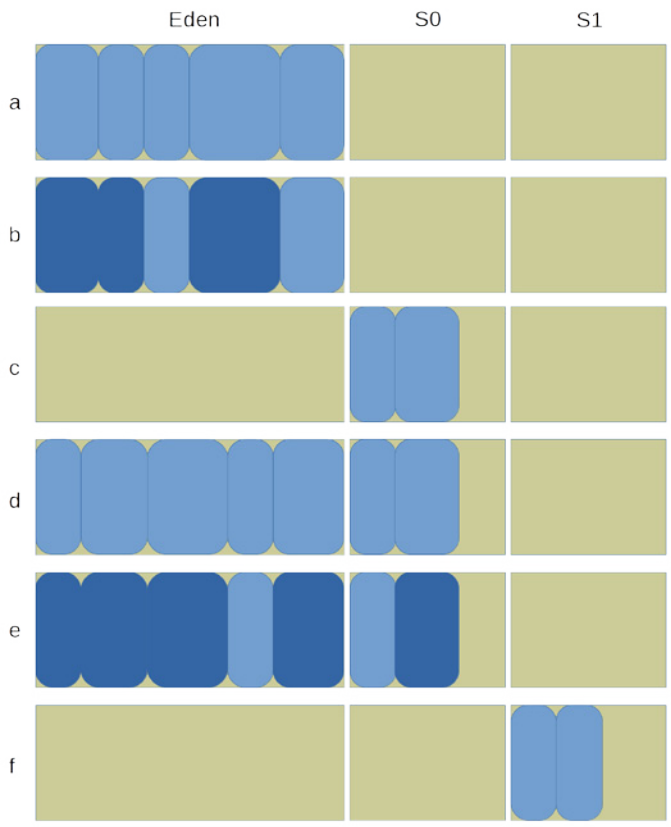


Rysunek 7. Rodzaje algorytmów GC

Najprostszym trybem jest wykonanie szeregowe (serial), w którym wszystkie wątki aplikacyjne (czarne) są zatrzymane, a wyłączny dostęp do pamięci ma pojedynczy wątek Garbage Collectora (czerwony). Jeżeli wprowadzimy modyfikację polegającą na zwiększeniu liczby wątków wykonujących kolekcję, przechodzimy w tryb równoległy (parallel). W dalszym jednak ciągu aplikacja jest zatrzymana w całym cyklu odśmiecania. Jednym z pomysłów na skrócenie czasu pauzy jest wykonywanie kolekcji przy jednoczesnym pozwoleniu na dalsze wykonywanie wątków aplikacyjnych. Działają one zatem współbieżnie (concurrent) do GC. Oczywiście z powodów wymienionych wcześniej nie jest to możliwe w 100%. Innym pomysłem wdrażanym w celu poprawy responsywności aplikacji jest wykonanie przyrostowe (incremental). Naturalnie nic nie stoi na przeszkodzie, aby łączyć ze sobą różne tryby wywołania, o ile tylko przybliża nas to do osiągnięcia założonego celu.

ALGORYTM KOPIUJĄCY

Klasyfikacje algorytmów GC zdają się nie mieć końca. Teraz poznamy jeden z ważniejszych rodzajów, czyli algorytm kopiujący. Zasada jego działania, jak nazwa wskazuje, opiera się na kopiowaniu obiektów. W momencie rozpoczęcia kolekcji wszystkie obiekty, które zostały oznaczone jako żywe, są przenoszone z jednej przestrzeni do drugiej. Jak się już można domyślić, algorytm ten wymaga przydzielenia dwa razy więcej pamięci niż ta, która realnie byłaby wykorzystana przez nasze obiekty. Natomiast podobnie jak każdy medal tak i omawiany algorytm ma dwie strony – za cenę zwiększonego wykorzystania pamięci otrzymujemy ciągłą przestrzeń nigdy nie podlegającą fragmentacji. Biorąc pod uwagę omówioną wcześniej generacyjność, można się spodziewać, iż wśród młodych obiektów kolekcje będą zachodziły znacznie częściej niż będzie to miało miejsce w przypadku starej generacji. Co za tym idzie fragmentacja pamięci mogłaby tu występować zdecydowanie szybciej, skutkiem czego to właśnie w przypadku młodej generacji warto zastanowić się nad zastosowaniem sprzątania poprzez kopiowanie. Takie też wnioski wyciągnęli programiści JVM i to właśnie ten algorytm stosowany jest do odśmiecania młodej przestrzeni.



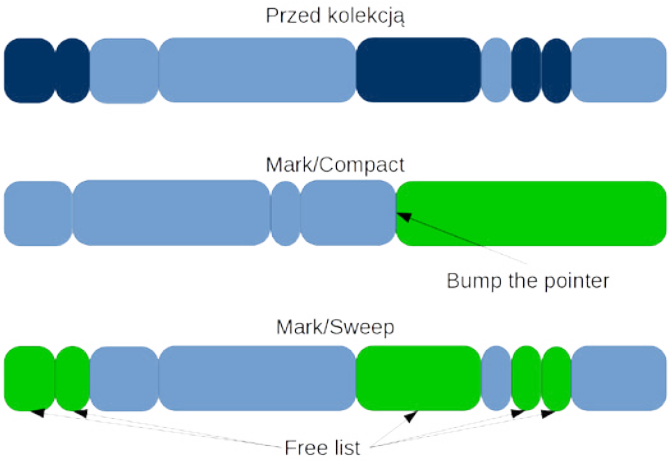
Rysunek 8. Kolekcja młodej generacji

Technicznie wygląda to w ten sposób, że wszystkie nowo utworzone obiekty alokowane są przez Wirtualną Maszynę w Edenie (Rysunek 8-a). W momencie, w którym Eden się zapełni, JVM uruchamia kolekcję. Pierwszym krokiem jest zaznaczenie żywych obiektów (Rysunek 8-b – obiekty martwe mają ciemniejszy kolor), które następnie przenoszone są do jednej z przestrzeni przetrwalnikowych (survivor space). Eden może zostać teraz wyczyszczony, co przygotowuje go do przyjęcia nowych obiektów (Rysunek 8-c). W momencie, gdy Eden ponownie się zapełni (Rysunek 8-d), wykonujemy kolejny przebieg GC. Zaznaczamy żywe obiekty (Rysunek 8-e) i kopiujemy je do drugiej przestrzeni przetrwalnikowej (Rysunek 8-f). Te czynności powtarzane są tak długo, aż algorytm nie oceni, iż dany obiekt jest już na tyle dojrzały, że można przestać przerzucać go pomiędzy przestrzeniami młodej generacji i wypromować (promote) do generacji starej (old), gdzie efektywniej będziemy mogli zarządzać pamięcią przy wykorzystaniu innych algorytmów.

ALGORYTMY WYSOKIEJ PRZEPUSTOWOŚCI

W kwestii wydajności aplikacji zawsze walczymy o jeden z dwóch celów: przepustowość lub latencję. Z tego też powodu w JVM mamy dostępny szereg różnych algorytmów cechujących sie różnymi atrybutami. W zakresie stawiania na wysoką przepustowość dostępne są dwa zbliżone do siebie algorytmy: szeregowy (serial) i będący jego wielowątkowym rozwinięciem równoległy (parallel). Oba kolektory zatrzymują aplikację na czas całej kolekcji i wykorzystują algorytm kopiujący dla młodej generacji oraz algorytm Mark-Sweep-Compact dla starej. Jak daje się łatwo zauważyć, składa się on z trzech faz:

- Mark – podczas której oznaczane są żywe obiekty,
- Sweep – kiedy czyścimy pamięć, usuwając obiekty porzucone w pierwszym kroku,
- Compact – eliminacja fragmentacji pamięci.



Rysunek 9. Sposoby wymiatania obiektów

O ile pierwsze dwa kroki zostały już omówione wcześniej, to o kompaktowaniu jeszcze nie wspomnieliśmy. Rysunek 9 przedstawia różne podejścia do sprzątania pamięci po usuniętych obiektach. Sytuacja przed kolekcją pokazuje podział na obiekty żywe (kolor jasny) i martwe (kolor ciemny). Po ich usunięciu powstaje wolna przestrzeń (kolor zielony), która, jak widzimy, w zależności od wykorzystanego podejścia może być ciągła bądź też nie. Podstawowym celem scalenia jest eliminacja fragmentacji pamięci, którą osiągamy poprzez przenoszenie obiektów tak, aby utworzyły ciągłą przestrzeń. To działanie zapewnia nam dwa ważne plusy, jednak (jak to z reguły bywa) nie bezkosztowo. Pierwszą zaletą kompaktowania jest możliwość wykorzystania całej dostępnej pamięci – wolny obszar dzięki scaleniu go w jeden region gwarantuje optymalne wykorzystanie każdego bajtu. Drugim

pozytywem jest łatwość alokacji nowych obiektów. Zawsze będziemy je „doklejać" na koniec, w związku z czym wystarczy nam przechowywać wskaźnik na początek wolnej przestrzeni i po dodaniu nowego obiektu po prostu przesunąć go o jego rozmiar. Kosztem tego podejścia jest wydłużenie czasu kolekcji – wszak obiekty trzeba przenieść, co pociąga za sobą konieczność aktualizacji ich referencji w obiektach od nich zależnych. Podsumowując, algorytmy serial i parallel są nastawione na optymalizację przepustowości. To znaczy, że czas potrzebny do odśmiecenia pamięci będzie optymalny, przy czym może to znaczyć jednorazowe zatrzymanie aplikacji na 5 minut raz na dobę, co nie zawsze będzie akceptowalne.

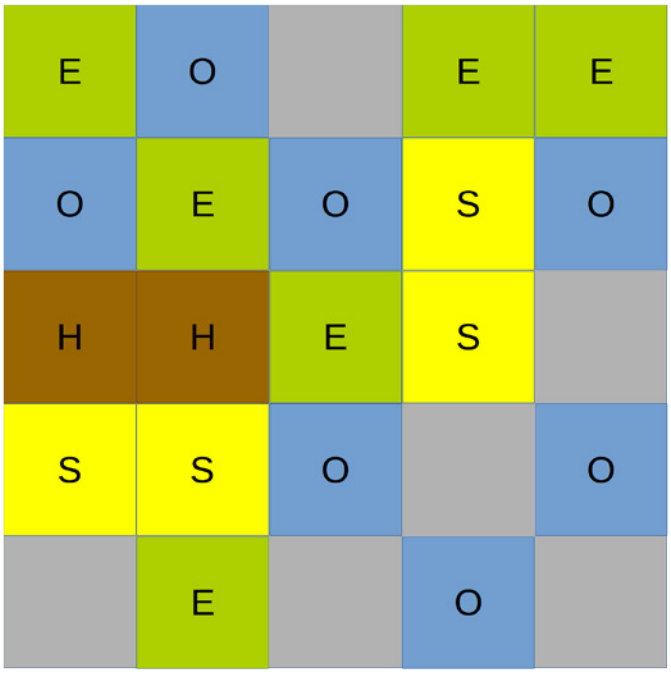
ALGORYTMY NISKIEJ LATENCJI

Jeżeli opisana powyżej sytuacja nie jest dopuszczalna ze względu na sposób wykorzystania naszej aplikacji (np. obsługa żądań z przeglądarki użytkownika), a zaczyna nam doskwierać zbyt długi czas przestojów wywołanych przez GC, musimy zainteresować się algorytmami nastawionymi na optymalizację responsywności. W JVM dostępne są obecnie dwa kolektory – CMS (Concurrent Mark Sweep) i G1. Oba algorytmy mają takie samo zastosowanie, a CMS jest po prostu prekursorem G1 i jeżeli tylko nie wykorzystujemy na co dzień Javy 5, powinniśmy wybierać młodszego kolegę (w ramach wtrącenia dodam, że nawet jeżeli wasze systemy używają Javy starszej niż wersja 8, to nie znaczy, że tę samą starszą wersję należy używać do uruchamiania IDE – to zawsze powinno działać na najnowszej dostępnej Wirtualnej Maszynie, co zdecydowanie poprawi jego wydajność).

Podstawową zasadą przyświecającą autorom tych algorytmów jest maksymalne skrócenie czasu poszczególnych pauz. Sposobem na osiągnięcie tego celu jest jak największe zrównoleglenie pracy kolektora i aplikacji. Gdyby udało się nam to w 100%, pauza nie występowałaby w ogóle, a cała kolekcja byłaby wykonana podczas normalnego działania aplikacji. Aż w takim stopniu nie jest to niestety możliwe, jednak oba algorytmy podzielone są na fazy, z których większość jest wykonywana bez przerywania pracy aplikacji. Fazy dla obu tych algorytmów są zbliżone i bez wchodzenia w nazewnictwo proces wygląda następująco: na samym początku zaznaczamy korzenie (GC roots). Jest to faza blokująca, która zatrzymuje wątki aplikacyjne (czyli tak samo jak w przypadku algorytmu parallel). Po wybraniu obiektów źródłowych następuje współbieżna faza zaznaczania żywych obiektów. Celem jej jest zaznaczenie maksymalnie wielu żyjących obiektów, których jednak liczba się cały czas zmienia, ponieważ kolekcja odbywa się przy normalnie działającej aplikacji, która, jakby nie było, cały czas alokuje nowe obiekty. Jeżeli GC oceni, że zaznaczył już wszystko, co można było w trybie współbieżnym, wchodzimy w fazę finalnego markowania, która podobnie jak wybieranie korzeni odbywa się w ramach pauzy. Po wyczerpaniu zbioru obiektów praca aplikacji jest wznawiana, a kolektor już w trybie współbieżnym dokonuje eliminacji martwych obiektów oraz wyczyszczenia statusu obiektów żywych, w celu przygotowania ich do następnej kolekcji. Część faz różni się nazewnictwem oraz szczegółami implementacyjnymi pomiędzy CMS a G1, jednak charakterystyka ich działania jest bardzo zbliżona.

Podstawową innowacją wprowadzoną przez algorytm G1 jest zmiana struktury sterty (heap). O ile w dalszym ciągu wykorzystujemy generacyjność, dzięki czemu rodzaje obszarów są bardzo zbliżone do tradycyjnego podejścia (jak występowanie Edenu, przestrzeni przetrwalnikowych czy starej generacji), to ich organizacja została całkowicie zmieniona. Cała sterta jest podzielona na ok. 2000 segmentów, które mogą być przetwarzane niezależnie od siebie. Każdy segment może przyjąć jedną z 5 ról, takich jak (Rysunek 10):

- Eden (kolor zielony),
- przestrzeń przetrwalnikowa (kolor żółty),
- stara generacja (kolor niebieski),
- humongous – do przechowywania bardzo dużych obiektów (kolor brązowy),
- wolna przestrzeń (kolor szary).



Rysunek 10. Struktura sterty w G1

PODSUMOWANIE

W artykule omówiliśmy wiele algorytmów pozwalających na automatyczne zarządzanie pamięcią. Z mojego doświadczenia wynika, że bardzo często to właśnie niezrozumienie zasad działania GC skutkuje bardzo często jego nieefektywnym tuningiem, w którym z reguły dodanie flag sterujących pogarsza jego wydajność. Najważniejszym krokiem prowadzącym do znalezienia nici porozumienia z JVM jest analiza logów. W GC możliwości logowania są bardzo szerokie, aczkolwiek już nawet samo dodanie flagi – `XX:+PrintGCDetails` dostarcza sporej dawki wiedzy o zachowaniu algorytmu. Więcej sposobów na obserwację metryk i statystyk działającej Wirtualnej Maszyny poznamy w trzecim artykule tego cyklu. Omówimy w nim narzędzia i techniki pozwalające obserwować działający system niczym pacjenta na stole operacyjnym :)

Jakub Kubryński jk@codearte.io

Od ponad 11 lat zawodowo zajmuje się oprogramowaniem. Częsty prelegent na branżowych konferencjach takich jak GeeCON, 33rd Degree, JDD, 4Developers czy Confitura, której jest organizatorem, Współzałożyciel startupu DevSKiller.com dostarczającego innowacyjną platformę oceny kompetencji programistów. Związany z software house Codearte. Trener w firmie Bottega, gdzie prowadzi szkolenia m.in.z wydajności, monitorowania i skalowalnej architektury systemów IT.