Szemétgyűjtés DotNET módra

A mai korszerű programozási nyelvekhez tartozó futtatókörnyezetek többsége tartalmaz valamilyen automatikus szemétgyűjtési mechanizmust a memóriakezelés leegyszerűsítése végett. Ebben a cikkben a C# programozási nyelvhez tartozó .NET CLR futtatókörnyezetben található szemétgyűjtőt (angolul: gargabe collector, vagy röviden csak gc) fogom bemutatni a C++-os memóriakezelési megoldáshoz viszonyítva.

A szemétgyűjtés okai

Arra a kérdésre, hogy miért van szükségünk automatikus szemétgyűjtésre, a válasz egyszerű. Mert könnyebbé teszi a programozó életét, hiszen így jobban tud koncentrálni a tényleges feladatra, ha nem kell külön törődnie a memória kezelésével is. Pontosan ez a lényeg, ugyanis a memóriakezelés kapcsán két olyan tipikus hibát követhetünk le, melyek sokkal alattomosabbak, mint bármilyen más alkalmazásbeli bug, ugyanis ezek hatására a program működése nem lesz megjósolható, vagyis nemdeterminisztikussá válik. Mielőtt még továbbmennénk, nézzük is meg, mi ez a két hibalehetőség:

- Az egyik a memóriaszivárgás (memory leak) esete, amikor a programozó elfelejti felszabadítani az általa lefoglalt tárterületet. Ekkor a memóriában az ilyen objektumok csak foglalják a helyet, de senki se használja őket semmire. Ez ahhoz vezethet, hogy elfogy a szabad memória, ami miatt az alkalmazás elszáll OutOfMemomeryException-nel.

- A másik lehetőség, amikor egy már felszabadított tárterületre hivatkozik a program. Vagyis azon a mutatón keresztül, amelyet az allokációkor kapott, olyan memóriaterületet ér el, mely már nem az övé, és itt bármilyen adat lehet, aminek a következtében kiszámíthatatlanná válik az alkalmazás futása.

Ez két olyan hiba, melyet általában rendkívül nehéz fülön csípni, és az esetek nagy részében a javításuk se feltétlenül triviális feladat. Az automatikus szemétgyűjtés eme két hiba bekövetkezésének valószínűségét hivatott nullára lecsökkenteni.

Egy objektum élete, memóriakezelés szempontjából

Egy erőforrást használó objektum életciklusa során az alábbi 5 tevékenység valamelyikét végezheti:

1) Helyet foglal a szükséges erőforrás számára

2) Inicializálja az erőforrást a kezdeti állapotnak megfelelően, és előkészíti a használatra

3) Eléri és használja az erőforrást az osztálypéldány megfelelő tagján keresztül

4) „Feltakarít” maga utána

5) Felszabadítja a lefoglalt tárterületet

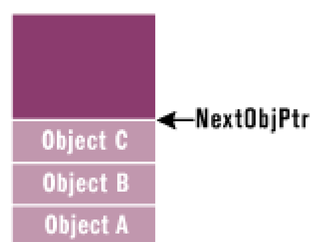
Memóriakezelés szempontjából minket most csak az első, a negyedik és az ötödik tevékenységek érdekelnek. Kezdjük először a két szélsővel, ugyanis ezeket a lépéseket képes automatikusan elvégezni a számunkra a GC! (A negyedik lépeshez a GC-nek ismernie kéne az erőforrás típusát és működését, de mivel ez nem várható el, ezért ez a feladat továbbra is a programozó feladata marad, de erről majd picit később).

C++ esetén, ha a memóriában helyet szeretnénk allokálni egy adott objektumnak, akkor azt úgy tesszük meg, hogy végigmegyünk egy láncolt listán, mely a memória szabad területeit reprezentálja, és az első méretben megfelelő szabad helyet lefoglalja. Ez így szép és jó, viszont ennek a műveletigénye a lineáris keresés műveletigényével, O(n)-nel egyenértékű. A GC ehhez képest O(1) műveletigénnyel képes nyújtani ugyanezt a szolgáltatást. Vajon miként lehetséges ez?

A memória allokáció egy olcsó művelet .NET környezetben

A GC esetén a memóriában a helyfoglalás azért lehet ilyen rendkívül olcsó, mert mindösszesen egy mutató értékének növeléséről van szó. Ez a mutató arra a tárterületre hivatkozik, amelyet a következő allokáció során az újonnan létrehozandó objektum fog majd megkapni, így nincs szükség semmilyen keresésre a helyfoglalás során. Hívjuk ezt a pointert NextObjPtr-nek. A legegyszerűbben ezt úgy tudjuk elképzelni, ha a memóriára, mint egy veremre gondolunk, és a verem legfelső elemére van egy referenciánk. (A továbbiakban erre a „veremre”, mint managed heap-re fogunk hivatkozni.)

Nézzünk egy egyszerű példát. Tegyük fel, hogy van A, B és C objektumunk. Először, amikor az A objektum számára akarunk helyet allokálni, akkor a NextObjPtr a verem legaljára mutat. Helyfoglalást követően ez az érték pontosan annyival nő meg, mint amekkora helyre szüksége volt az A objektumnak. A B és a C objektum helyfoglalása után valahogy így nézhet ki a memória.



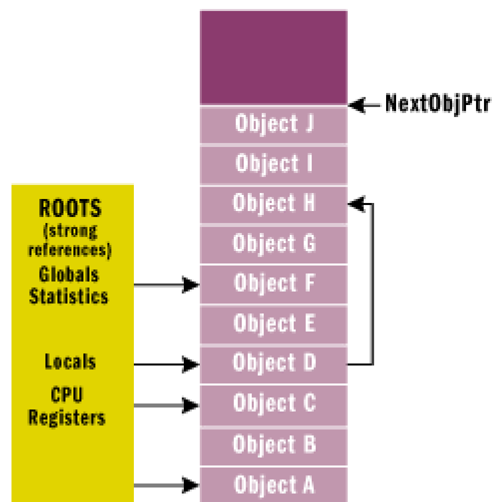
Ahhoz, hogy mindez jól és hatékonyan működhessen, két eléggé erős feltételnek kell teljesülnie: A címtér és a tárhely legyen végtelen. Nos, ezeket a feltételek eléggé nehezen lehet biztosítani, emiatt van szükség néha a memória kitakarítására. Ezt a mechanizmust hívjuk szemétgyűjtésnek.

Hogyan történik a szemét összegyűjtése?

A szemétgyűjtés menete röviden az alábbi: Ha a managed heap megtelt, akkor a szemétgyűjtő összegyűjti azokat az objektumokat, melyekre már nincs szüksége az alkalmazásnak és felszabadítja az általuk foglalt helyet, így ismét lesz szabad tárterület. Ez

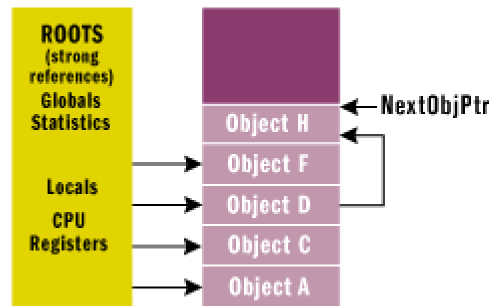
mind szép és jó, de honnan tudja a GC, hogy mire van szüksége egy alkalmazásnak és mire nincs? Onnan, hogy úgynevezett erős referenciákat, vagyis röviden root-okat használ. Ezek azok a pointerek, melyek például globális vagy statikus objektumokra hivatkoznak, vagy egy szál stack-jének lokális változói, vagy akár a CPU regiszterében lévő mutatók, tehát az alkalmazás élő komponensei. A root-ok hivatkozhatnak egy managed heap-beli tárterültre, vagy akár lehet az értékük null is.

Azok az objektumok, melyre van ilyen root hivatkozás, azok „aktív" objektumok, vagyis az alkalmazás számára elérhetőek. Az összes többi nem elérhető, vagyis szemét. Íme, egy ábra, mely erre mutat egy példát.



Maga a GC algoritmus ezek alapján a következőképpen működik. Ha megtelt a managed heap, akkor a GC elkezdi összegyűjteni a szemetet. Abból a feltevésből indul ki, hogy minden, ami a memóriában van, az szemét. Viszont, ha talál rá root hivatkozást, akkor azt elérhetőnek minősíti, és nem fogja törölni. A fenti ábrán jól látszik, hogy nem csak az A, a C, a D, és az F objektumok érhetőek el, hanem a H is. (Indirekt módon a D-n keresztül.) Azért, hogy a GC ne töröljön olyan objektumokat, melyek, ha direktbe nem is, de elérhetőek, ezért rekurzívan végignézi az összes root-ból elérhető objektumot. A rekurzív bejárás során a GC algoritmus felépít egy elérhetőségi gráfot, és azokat az objektumokat fogja majd csak törölni, amelyeket nem lehet elérni, vagyis amelyek nem szerepelnek a gráfban. A vizsgálat során, ha már egy adott objektum szerepel a gráfban, akkor azt már nem fogja továbbvizsgálni, egyrészt teljesítmény szempontból, másrészt a végtelen körhivatkozások elkerülése érdekében.

A fentebbi ábrán látható példa a GC lefutása után az alábbi módon néz ki.



Vagyis a szemétgyűjtés olyan módon valósul meg, hogy a GC végigmegy az összes a root-okból elérhető objektumokon rekurzívan, közben felépít egy gráfot, és azokat az objektumokat, melyek nem szerepelnek a gráfban, törli. A törlés után a managed heap-et ismét konzisztens állapotba hozza, vagyis memcpy-val szépen átmásolgatja az elemeket a felszabadult üres helyeknek megfelelően. Természetesen az átrendezést követően a GC feladatai közé tartozik még a pointerek értekeinek fixálása is az új helyezet alapján.

Ami a fentebbi leírásból egyből kitűnik, hogy egy GC lefutás nem egy olcsó művelet, viszont szerencsére csak akkor fut le, ha megtelt a managed heap. Köztes állapotban ellenben sokkal gyorsabb megoldást nyújt, mint a C++-os megvalósítás. Ami még szintén látszik az algoritmusból az az, hogy a fentebb említett két probléma, itt már nem fordulhat elő. Hiszen ha már nincs az adott objektumra referencia, akkor biztosan fel lesz szabadítva. A másik irány is igaz, vagyis ameddig van rá hivatkozás, addig biztosan nem kerül felszámolás alá. A nagy kérdés már csak az, hogy ha a GC ennyire jó, akkor az ANSI C++ miért nem ezt használja? A kérdésre a válasz a kasztolásban rejlik, ugyanis C++-ban a pointer által mutatott objektumok átkasztolhatóak egyik típusról a másikra, viszont így a rendszer nem tudná kideríteni, hogy a pointer mire is mutat.

Ideje feltakarítani magunk után

Most hogy már tudjuk, miként kezeli a GC az egyes (allokáció) és az ötös (felszabadítás) tevékenységét egy adott objektumnak, nézzük meg a négyes (feltakarítás) lépést is. Ebben az állapotában az objektumnak az általa használt erőforrást fel kell szabadítania, oly módon, hogy visszaállítja eredeti állapotába (pl.: fájl esetén bezárja a kommunikációs csatornát és leveszi róla az írási zárolást). Ezt a GC nem tudja automatikusan megcsinálni, ezért a fejlesztő kap egy olyan nyelvi lehetőséget, amelyen keresztül ezt a lépést megvalósíthatja. Ezt hívják Finalization-nek. Ez egy olyan függvény, melyet akkor hív meg a GC az adott objektumon, miután már szemétnek minősítette, de még mielőtt törölné. C++-os fejlesztők egyből analógiába hoznák ezt a destruktorral, de a valóságban eléggé nagy az eltérés a kettő között (részletesebben kicsit később).

Tehát kapunk a rendszertől egy Finalize elnevezésű metódust, mely segítségével feltakaríthatunk magunk után. Viszont ügyelnünk kell arra, hogy mikor használjuk ezt. Ökölszabályként elmondható, hogy szinte soha, próbáljuk meg kerülni, mert eléggé sok hátrányunk származik belőle. Íme, a teljesség igénye nélkül néhány példa:

- A Finalize metódussal rendelkező objektumok két körben kerülnek felszabadításra. (lásd később)

- Már maga az objektum allokálása is tovább tart, ugyanis speciálisan kell kezelni az objektumot, már az elejétől kezdve. (szintén lásd később)

- Hivatkozhat olyan objektumokra, melyek nem tartalmaznak Finalize-t, de emiatt mégis sokáig élnek.

- Több száz példány esetén már jelentős teljesítménycsökkentést okoz, ha egy osztály példányain egyszerre meg kell hívni a Finalize-t.

- Nincs kontrollunk afölött, hogy pontosan mikor is hívódik meg.

- Ha a program valami miatt terminál, akkor nem fut le a Finalize metódus, kivéve, ha ki nem kényszerítjük belőle (RequestFinalizeOnShutdown).

- Nem garantált a meghívásuk sorrendje, vagyis elképzelhető, hogy előbb a gyerek objektum(ok) Finalize-ja hívódik meg és csak utána a szülőé, ami probléma lehet, ha a szülőben hivatkozunk a gyerek objektum(ok)ra.

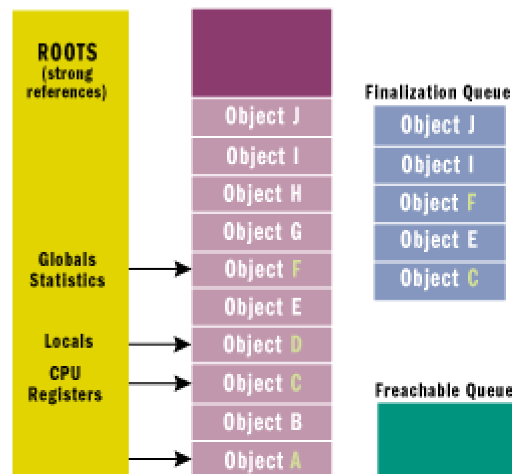
Ha ezek ellenére is szükségesnek tartjuk a Finalize metódus megírását, akkor törekedjünk arra, hogy minél gyorsabb és rövidebb legyen. Illetve ügyeljünk arra is, hogy ne tartalmazzon szál szinkronizációt és ne dobjon kivételt!

Visszatérve a Finalize != destruktor problémakörhöz, vizsgáljuk meg kicsit közelebbről a problémát. C++ és C# esetén is a fordító képes arra, hogy a származtatott osztály konstruktorába beleszerkessze az ősosztály konstruktorának hívását. C++ esetén ez a destruktorra is igaz, hiszen ott a sorrendiség garantált. Ellenben C# esetén, ahol nincs destruktor, csak Finalize, a fordító számára nem egyértelmű, hogy van-e az adott osztálynak, meg kell-e hívnia, stb. Emiatt az ősosztály Finalize metódusának a meghívása a programozó felelőssége. Ilyenkor ügyeljünk arra, hogy az ősosztály Finalize-ának hívása kerüljön a metódus legvégére.

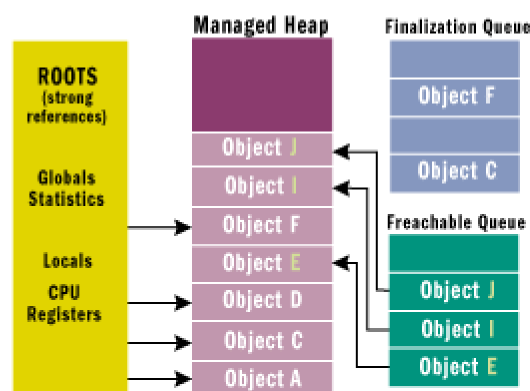
Érdekesség: C# esetén van arra lehetőség, hogy szintaktikailag „destruktort” írjunk (vagyis lehet írni ~Object() metódust), de ez a fordítás során az ősosztály Finalize-nak felüldefiniálására fordul le (protected override void Finalize()). Tehát Finalize != destruktor.

Feltakarítás két lépésben

Azon objektumokat, melyek rendelkeznek Finalize metódussal, már akkor meg kell különböztetni a többitől, amikor lefut a new operátoruk. Ezért a rendszer két segédtáblát használ arra, hogy ezt az információt nyilvántartsa. Az egyik a finalization queue névre hallgat. Ebbe a sorba bekerülnek azokra az objektumokra mutató referenciák, melyek rendelkeznek Finalize metódussal, mint ahogyan azt az alábbi ábra is mutatja.

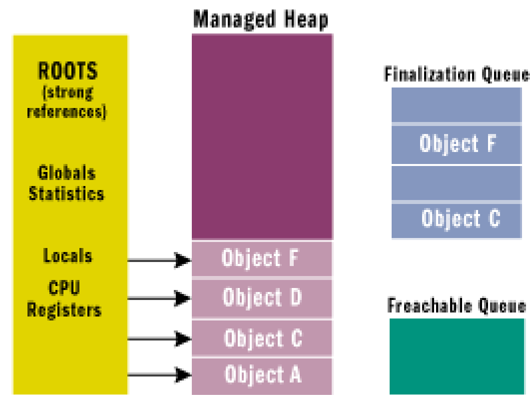


Amikor lefut a szemétgyűjtő, akkor megnézni, hogy a szemétnek ítélt objektumok közül, melyekhez tartozik finalization queue-beli bejegyzés. Amelyikhez van, azt innen törli és átteszi a másik segédtáblába, az freachable queue-ba. Tehát itt azok az objektumok vannak, amelyeknek van Finalize metódusa és szemétnek lettek minősítve. A fenti ábrán ezek az E, I és a J. (A B, G és a H objektumok ilyenkor már nincsenek benne a managed heap-ben). Ezt az állapotot szemlélteti az alábbi ábra.



Az freachable sor nevének az elején az f a finalizable-re utal, vagyis, hogy van Finalize metódusa az adott objektumnak, a reachable pedig arra, hogy újra van az adott elemre referencia, vagyis újból elérhető (van rá root), emiatt a GC nem törölheti. Azért kell újból elérhetővé tenni, hogy le lehessen futtatni a feltakarítást. Ezt egy külön szál végzi, ami mindaddig alszik, amíg üres a sor. (Emiatt a Finalize-ban kerüljük a szálkezelést!) Amint került a sorba új elem, végigmegy az összes bejegyzésen és szépen meghívogatja mindegyiken a Finalize-t, majd ezután törli a sorból a bejegyzést. Vagyis az objektum ténylegesen majd csak a következő GC lefutásakor fog felszámolódni, addig továbbra is ott lesz a managed heap-en.

Tehát a GC első lefutásakor az adott objektumra kapunk egy root-ot azáltal, hogy átpakoljuk a freachable sorba, így emiatt már nem lesz szemét, majd végrehajtatjuk egy külön szálon a Finalze metódust, töröljük a sorból, így elveszik az elérhetősége, emiatt a következő körben már az objektum szemétnek minősül. Az előző ábra az első GC lefutása utáni állapotot mutatja, az alábbi a második utánit.



Amikor a holtak életre kelnek

Ha jobban belegondolunk, hogy mi is történik valójában a Finalize metódus kezeléskor, akkor azt láthatjuk, hogy van egy objektum, ami meghal, majd a Finalize miatt újra él, majd megint (de most már végleg) meghal. Ezt a folyamatot hívjuk feltámadásnak (resurrection-nek).

A második meghalás, azért következik be, mivel a Finalize lefutása után már nincs erős hivatkozásunk az objektumra. De mi van akkor, ha a metóduson belül értékül adjuk az adott objektumot egy globális vagy statikus változónak? Nos, újra élni fog.☺ Ami több szempontból sem szerencsés. Hiszen egyrészt már feltakarított maga után, másrészt az általa hivatkozott más objektumok is már halottak lehetnek. Jön a zseniális ötlet: élesszük őket is újra. És mindjárt ott is tartunk, hogy van egy teljes zombi hadseregünk. A zombi itt nagyon találó név, hiszen él, mivel van rá root, de halott, mivel már lefutott a finalize. Így, egy olyan rendszert kapunk, amelynek viselkedése teljesen megjósolhatatlan.

Ha valamilyen okból kifolyólag mégis szükségünk lenne a holtak feltámasztására, akkor azt csináljuk okosan. Egy flag segítségével tároljuk el, hogy egyszer már halálra lett ítélve szerencsétlen, és ez alapján a flag alapján cselekedjünk az objektum metódusain belül.

Sőt még tovább megyek, ha újból feleslegessé válik az objektum, jó lenne, ha ismét lefutna a Finalize metódus. Viszont az objektum nem lett újra allokálva, így nem került bele a finalization queue-ba, vagyis nem fog meghívódni alapból. Ezen úgy tudunk segíteni, hogy meghívjuk a GC ReRegisterForFinalize() metódusát. Ilyenkor egy új bejegyzés kerül be a sorba. Ennek köszönhetően pedig sikerült eljutni a hallhatatlansághoz, hiszen ahányszor meghalunk, annyiszor újra is tudjuk éleszteni magunkat, így megtaláltuk az örök élet elixírét. ☺ Viccet félretéve, van lehetőség újraélesztésre, de csak óvatosan használjuk! (A ReRegisterForFinalize-t csak egyszer hívjuk meg, ha kell, mert különben annyiszor kerül be egy új bejegyzés a sorba, ahányszor meghívjuk a függvényt, vagyis annyiszor fog lefutni a Finalize metódus.)

Még élőként rendezzük a végrendeletünket

Az előző pár bekezdésben láthattuk a Finalize előnyeit és hátrányait, most nézzünk meg egy segédtechnikát, mely segítségével ki lehet küszöbölni a Finalize hibáinak egy részét. A technika lényege, hogy a feltakarítást ne bízzuk a GC-re, hanem mi magunk végezzük el

explicite a programból. Ehhez arra van szükségünk, hogy létrehozzunk egy Close, vagy egy Dispose metódust. Előbbre akkor van szükségünk, ha feltakarítást követően még használható az objektum >> tartozik hozzá Open is. Utóbbit pedig akkor, ha már nincs tovább szükségünk az adott objektumra.

Nézzünk egy egyszerű példát ennek a használatára. Tegyük fel, hogy van egy FileStream objektumunk, amellyel egy fájlba akarunk írni. A jobb teljesítmény érdekében az implementáció soron puffereket használ az objektum, viszont ezek tartalma csak akkor kerül ki ténylegesen az adott fájlba, ha a pufferek megteltek. Ha ezek flush műveletét a Finalize-ban hívnánk meg, akkor mindaddig zárolva lenne a fájl, ameddig fel nem szabadítja a GC a tárhelyet, amire várhatnánk egy ideig. Ezért a FileStream objektumnak van Close metódusa is. Ilyenkor felmerülhet bennünk a következő kérdés: ha meg lett hívva explicite a Close, akkor is meg fog-e hívódni a Finalize, és ha igen, akkor mit fog csinálni?

A válasz az, hogy nem, nem fog meghívódni. Egy flag a háttérben beállítódik a Close meghíváskor, így a Finalize-ban lévő kódot ki is lehet hagyni. Ettől még a Finalize továbbra is meghívódna, hiszen benne van a finalization queue-ban, teljesen feleslegesen. Ennek elkerülése érdekében a GC-nek van egy olyan metódusa, mellyel ki tudjuk innen venni (pontosabban kihagyathatjuk azt a lépést, hogy áttegye az freachable sorba). Ez pedig nem más, mint a SuppressFinalize() metódus.

Természetesen az előző példát még tovább lehet egy kicsit bonyolítani azzal, hogy egy olyan StreamWriter-t használunk, mely egy FileStream-et alkalmaz. Mindkét objektum puffert használ, így a StreamWriter-nél is szükséges a flush meghívása. Ha tehát a StreamWriter Close metódusát meghívjuk explicite, akkor az meghívja a FileStream-ét is, így azt már nem kell külön. Ez így jól is működik, de mi van, ha lemaradt a Close meghívása? Nos, ez esetben jön a GC és a Finalize metódusok. DE, van egy kis bibi. A GC nem garantálja a sorrendet, vagyis előfordulhat, hogy előbb hajtja végre a FileStream Finalize-n belül a puffer flush metódusát és majd csak aztán a StreamWriter Finalize-jának flush metódusát, ami nem túl jó nekünk, hiszen így akár fontos adatok is elveszhetnek.

Erre a Microsoft megoldása a következő: A StreamWriter-nek nincs Finalize metódusa, mivel nem garantálható a sorrend. Viszont van Close metódusa, mely explicit meghívása esetén a rendszer jól működik, meg nem hívása esetén pedig adatvesztés esélye állhat fenn, amelyet a programozó észrevesz, így a probléma feloldható.

Ez még hátha jó lesz valamire

A feltakarítás hátulütőinek tisztázása után most nézzük meg azt, hogy miként lehet egy objektumot „kómába helyezni”. A dolog lényege az, hogy van egy objektumunk, mely jelen pillanatban bevégezte a dolgát, de lehet, hogy a későbbiekben még szükségünk lenne rá, csak nem tudjuk, hogy pontosan mikor, vagy, hogy egyáltalán kell-e még. Tegyük fel továbbá azt is, hogy ezen objektum létrehozása drága memória szempontjából. Emiatt nem

szeretnénk még egyszer újrainicializálni, de ha sokáig nem kell, akkor akár törlődhet is. Ehhez az úgynevezett gyenge referenciákra van szükségünk (WeakReference, vagy röviden wr).

Vagyis, ha másfelől közelítjük meg a problémát, azt mondhatjuk, hogy a wr-k segítségével olyan objektumokra adhatunk referenciát, melyek el is érhetőek, és törölhetőek is. Ez mégis hogyan lehetséges? A válasz az időzítésben rejlik. Ha még a GC lefutása előtt a wr-en keresztül létre tudunk hozni az objektumra egy erős referenciát (Strong Reference, vagy röviden sr = root), akkor az objektum elérhető. Ha a GC lefutása után szeretnénk elérni az objektumot, akkor nem fog sikerülni, ugyanis a GC már felszabadította az adott tárterületet.

Nézzük ezt meg egy egyszerű példán keresztül. Tegyük fel, hogy van egy alkalmazásunk, mely két nagy egységből áll, melyek között a felhasználó bármikor átválthat. Az alkalmazás egyik felének el kell érnie a fájlrendszerbeli mappákat. Tegyük tovább fel azt, hogy a mappákról készített gráfot eltároljuk a memóriában a jobb teljesítmény érdekében. Ilyenkor van egy root-unk erre az objektumra. Ha a felhasználó átvált az alkalmazás másik felére, ahol nincs szükségünk a mappaszerkezetre, akkor készíthetünk erről az objektumról egy wr-t. Így, ha a felhasználó nem tér vissza egy újabb GC lefutása előtt, akkor az erőforrás felszabadítható, hiszen csak wr-ünk van az objektumra, sr-ünk nincs. Ebben az esetben sajnos újra fel kell építeni a teljes gráfot, ha a felhasználó visszatér az alkalmazás egyik feléhez. Ellenben, ha a felhasználó még a GC lefutása előtt tér vissza, akkor a wr-n keresztül el tudja úgy ismét érni az objektumot, hogy létrehoz rá egy erős referenciát.

Íme, a gyenge referenciák használatának a pszeudó-kódja:

```
LargeObject lo = new LargeObject(); //van egy sr-ünk az objektumra
//dolgozunk vele, majd átváltunk az alkalmazás másik felére
WeakReference wr = new WeakReference(lo); //van sr-ünk és wr-ünk is
lo = null; //már csak egy wr-ünk van az objektumra
// dolgozunk az alkalmazás másik felében, majd visszatérünk
lo = wr.Target;
if(lo == null)
      lo = new LargeObject(); //későn értünk vissza, lefutott a GC
```

A fenti példakód két érdekes részt tartalmaz: az egyik a wr létrehozása, a másik pedig az sr létrehozása wr-ből. Nézzük először az elsőt. Amikor egy gyenge referenciát szeretnénk létrehozni egy adott objektumra, akkor azt úgy tudjuk megtenni, hogy a beépített WeakReference típusból létrehozunk egy új példányt és átadjuk a konstruktornak paraméterként az objektumra mutató referenciát. Ezek után pedig töröljük az erős referenciát, hiszen ha ez továbbra is megmaradna, akkor az egésznek nem lenne semmi értelme.

A gyenge referenciáknak két fajtája van, a rövid és a hosszú életű. A rövid gyenge referenciák (short weak reference) olyanok, hogy nem figyelik, hogy újra lettek-e élesztve. Míg a hosszú életű gyenge referenciákat (long weak reference) igen. Ez utóbbit úgy tudjuk létrehozni, hogy a konstruktor egy másik túlterhelt változatát használjuk, amelyik vár egy bool paramétert is. Ha long wr-t akarunk, akkor ezt állítsuk be true-ra. Mellesleg megjegyezném, hogy kerüljük a long wr-eket, mert az általuk mutatott objektumok feltámadása után az objektumok kiszámíthatatlan viselkedést produkálnak!

A gyenge referenciából erős referencia készítés oly módon valósul meg, hogy a wr Target tulajdonságán keresztül elérhető objektumot értékül adjuk egy root-nak, vagy egy abból elérhető referenciának. Ha ez az érték null, akkor elkéstünk, lefutott a GC, nincs mit tenni, újra létre kell hozni az objektumot. Viszont, ha az értéke nem null, akkor minden további nélkül tovább tudjuk használni.

A gyenge referenciákkal kapcsolatban felmerülhet bennünk egy érdekes kérdés. Hogyan tudunk úgy létrehozni gyenge referenciát, hogy közben nem hozunk létre egyben erős referenciát is? Másképp megfogalmazva a kérdést, ha az objektum csak a WeakReference-n keresztül elérhető, akkor tulajdonképpen elérhető, vagy mégsem?

A gyenge referenciák misztériuma

Hogy tudunk úgy létrehozni egy objektumra referenciát, hogy az valójában ne legyen referencia? A válasz nem is annyira egyszerű. Amikor a WeakReference beépített típus konstruktorának átpasszoljuk az objektumra mutató pointert, akkor valójában nem történik a managed heap-en allokáció. De ha nincs allokáció, nem jön létre egy új objektum, mely hivatkozna rá, akkor mégis, hogy tudjuk őt elérni a későbbiekben? Nos, egy segédtábla segítségével. Pontosabban kettővel, ugyanis, amikor létrehozunk egy WeakReference-t, akkor egy bejegyzés bekerül a short weak reference táblába vagy a long weak reference táblába a referencia típusától függően.

Ezekben a táblákban a managed heap-en lévő memóriacímre vannak hivatkozások, de ezek nem tekintendőek root-oknak! Ezek alapján tekintsük meg a GC algoritmus gyenge referenciák kezelésével kibővített változatát.

1) A GC felépíti az elérhetőségi gráfot a managed heap alapján

2) Megnézi a short weak reference táblában, van-e olyan hivatkozás, mely szemétre mutat. Ha van, akkor ennek a bejegyzésnek az értékét null-ra állítja.

3) Megnézi a finalization sort, hogy van-e olyan hivatkozás benne, mely szemétre mutat. Ha igen, akkor áthelyezi a mutatót a freachable sorba, és hozzáadja az objektumot a gráfhoz, így újból elérhetővé válik az.

4) Megnézi a long weak reference táblát van-e benne olyan hivatkozás, mely olyan objektumra mutat, mely nincs a gráfban. (Ilyenkor már a gráf részét képezik az freachable objektumai is!). Ha talál ilyet, akkor ennek az értékét null-ra állítja.

5) Törli a szemetet és eltünteti a lyukakat, illetve frissíti a referenciákat.

A short és a long wr-k közötti igazi különbség a következő. Short wr esetén, ha az objektum szemétnek lett minősítve, akkor egyből ezután törlődik is a short weak reference táblából róla a bejegyzés. Viszont, ha ennek az objektumnak van Finalize-ja, ami még nem futott le, és el szeretnénk érni az objektumot, akkor azt már nem tudja, hiszen már nincsen rá mutató, pedig még mindig ott van a managed heap-en. Ezzel szemben a long wr-nál csak akkor törlődik a bejegyzés a long weak reference táblából, ha az objektum tárhelye fel lett szabadítva, vagyis, ha újra is élesztik, akkor is megmarad rá a hivatkozás.

Különböző generációk, különböző szokások

A cikk hátralévő részében néhány GC optimalizálási technikát fogok bemutatni. Ezek közül az egyik legfontosabb a generációk kezelése. Amikor több különböző generációval dolgozunk egyszerre, akkor az alábbi négy feltevés mindegyike igaz kell, hogy legyen:

1) Az új objektumok, rövid ideig fognak élni

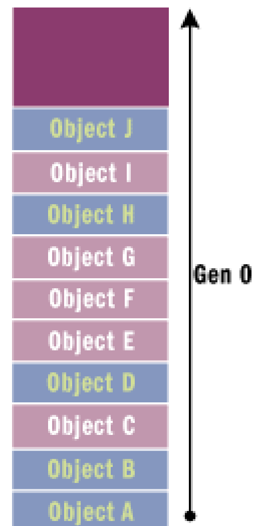
2) A régi objektumok, sokáig fognak élni

3) Az új objektumok között erős kapocs van, és elérésük gyakran azonos időben történik

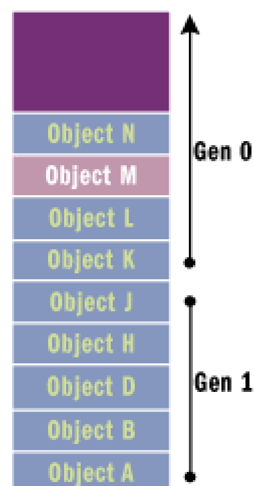
4) A heap egy adott részének a kezelése gyorsabb, mint a teljes heap menedzselése

Tanulmányok állítják, hogy ezek az állítások megállják a helyüket a manapság használt szoftverek nagy részénél. Nézzük, ezek a feltevések hogyan segítik a GC-t a teljesítményének a javításában.

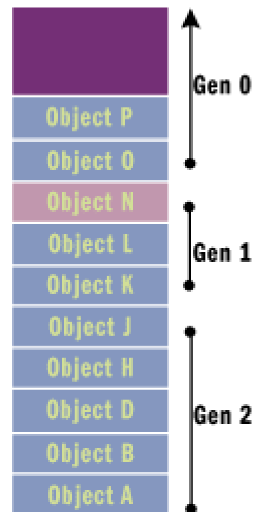
Kezdetben egy üres managed heap-pel indulunk. Létrehozunk objektumot, majd megtelik a memóriánk. Az ilyenkor a memóriában lévő objektumok a 0. generáció részét képezik. Ezek azok az új objektumok, melyeket a GC még nem vizsgált meg, mint ahogyan az az alábbi ábrán is látható.



A rózsaszínnel jelölt objektumokat a GC szemétnek minősítette, ezért felszabadítja az általuk lefoglalt helyet. A megmaradt objektumokat ezután 1. generációsaknak nevezzük, és ezek feltehetően nem is nagyon fognak változni. A GC lefutása után érkező újonnan allokált objektumok fogják képezni az új 0. generációt, mint ahogyan az alábbi ábrán is látszik. A GC az esetek többségében csak a 0. generációs objektumokat vizsgálja.



Az M törlése után a 1. generációból 2. generáció lesz, a 0. generációból pedig 1. (A legmagasabb generációs szint a 2.) Az újonnan érkező objektumok lesznek az új 0. generáció, melyet a heap betelésekor a GC-nek meg kell vizsgálnia. Ezt szemlélteti az alábbi ábra.



Tehát összefoglalva a generációk kezelését, a rendszer úgy viselkedik, hogy mindig csak a legutóbbi GC óta létrejött objektumokat vizsgálja meg, a régieket békén hagyja. Ilyenkor két kérdés merülhet fel bennünk:

- Mi van akkor, ha a 0. generációs takarítás után sincsen elég szabad hely?

- Mi van akkor, ha nem egy 0. generációs objektum valamely hivatkozása frissül egy újonnan létrehozott objektumra?

Az első kérdésre a válasz, hogy ebben az esetben a GC a 0. és az 1. generációt kezdi el vizsgálni. Ha ez se hozná meg a kellő sikert, akkor a 0., 1. és a 2. generáció elemeit fogja megvizsgálni.

A második kérdésre a válasz kicsit összetettebb. Alapból a régebbi generációk olyan elemeinek belső hivatkozásait, melyre van root, a GC nem vizsgálja a fa építésekor. Viszont ezek idővel változhatnak is, emiatt a GC igénybe veszi a rendszer write-watch támogatását, ami a kernel32.dll-beli GetWriteWatch metódushívást jelenti. Ez által megtudhatja, hogy a legutóbbi GC lefutás óta történt-e referenciafrissítés.

A kezdeti 4 felvetésünk közül eddig még csak hármat használtunk fel. Most nézzük meg a 3-as sorszámú állításból milyen előnyünk származik. Ha C++-ban új elemeket folyamatosan allokálgatunk, akkor azok nagy valószínűséggel nem fognak egymás mellé kerülni, hiszen az első méretben is megfelelő szabad helyet fogják megkapni. DotNET esetén a veremszerű szerkezet itt előnyt jelent, hiszen így biztosan közel lesznek egymáshoz az újonnan allokált elemek. Mivel ezek sűrűn lépnek kapcsolatba egymással, emiatt a folytonos elhelyezés előnyös, hiszen nagyon valószínű, hogy sok ilyen objektum befér a processzor cache-ébe, ami gyorsabb elérést biztosít, mint a Ram-ból történő adatkinyerés.

Többszálúság kezelése

Utolsó blokként még néhány szót ejtenék a többszálú programoknál lévő optimalizációs lehetőségekről. De mielőtt még belemennénk a mechanizmusokba, nézzük azt, milyen kockázatot is hordoz magában a többszálúság a GC szempontjából.

Ha van egy többszálú alkalmazásunk, akkor azt meg kell gátolnunk, hogy ezek a szálak hozzáférjenek a managed heap-hez mindaddig, amíg a GC tevékenykedik. Hiszen az átrendezéskor az egyes mutatók értekei is megváltoznak. Tehát amíg a GC egy külön szálon fut, addig a több szál működését fel kell függeszteni, és az overhead-et minimálisra kell csökkenteni. Erre sokféle mechanizmus létezik, a teljesség igénye nélkül, íme, néhány:

- Teljesen megszakítható kód: Amikor a GC elkezd dolgozni, akkor az összes többi szál felfüggesztődik. Viszont ahhoz, hogy utána folytatni lehessen őket a JIT (Just-In-Time) fordító által kezelt táblákban el kell tárolnia azt, hogy az adott szál éppen hol tartott egy adott metódusban, mely objektumokat használta és azok hogyan voltak elérhetőek (változó, regiszter, stb.). Ha ezek az infók megvannak, akkor a GC lefutása után adatvesztés nélkül folytatható tovább az alkalmazás.

- Szál-eltérítés (Hijacking): Mivel a GC eléri és módosítani is tudja a szál stack-jét, ezért az éppen aktuálisan végrehajtott függvény visszatérési pontját átirányíthatja egy speciális függvényre. Ha az éppen futó metódus végére ér a szál, meghívódik a speciális függvény, mely felfüggeszti a szál tevékenységét a GC idejére. A takarítás után pedig visszaadja a vezérlést a szálnak.

Érdekesség: Hijacking esetén unmanaged code futtatása lehetséges párhuzamosan a GC futtatásával mindaddig, míg nem akar managed objektumhoz hozzáférni.

- Mentési-pontok: A JIT fordító képes elhelyezni a metódus belsején belül olyan speciális függvényhívásokat, melyek azt ellenőrzik, hogy a GC nem várakozik-e. Ha igen, akkor altatja a szálat, majd a lefutása után felébreszti. Azokat a helyeket, ahova beszúrja ezeket a speciális függvényhívásokat a JIT, mentési pontnak nevezzük.

További lehetőségek többprocesszoros rendszerek esetén:

- Szinkronizáció-mentes allokálás: Többprocesszoros rendszerek esetén a managed heap 0. generációja felosztható annyi részre, ahány processzor található az adott rendszerben. Ebben az esetben ezek szabadon, párhuzamosan végezhetnek allokációt, mindenféle szinkronizáció és zárolás nélkül.

- Skálázható szemétgyűjtés: Többprocesszoros szerverek esetén nem csak a managed heap-et lehet megosztani, hanem magát a GC-t is. Ez azt jelenti, hogy mindegyik CPU a saját kis memória részét maga kezeli, és azon futtatgatja időről időre a nagytakarítást. Ebben az esetben a GC egy speciális változatáról beszélünk, mely a SrvGC névre hallgat és az

MSCorSrv.dll-ben található. A „normál” változat mellesleg a WksGC nevet kapta és az MSCorWks.dll-ben lakik.

Összefoglalás

A cikk elején megismerkedhettünk a memóriakezelés szemétgyűjtést nem használó módszereinél leggyakrabban előforduló hibákkal, majd megnéztük, hogy a GC ezeket hogyan képes orvosolni. Ezt követően utánajártunk annak, hogy milyen feltakarítási mechanizmusok léteznek és azoknak mik az előnyei, illetve a hátrányai. A cikk második felében pedig jó néhány trükköt megnéztünk arra, hogy miként lehet kicselezni a GC-t, például újraélesztés, gyenge referenciák, stb. segítségével. Végezetül pedig betekintést nyerhettünk abba is, hogy a GC-t miként lehet optimalizálni a jobb teljesítmény elérése érdekében.