Електронски факултет модул Рачунарство и Информатика

*Семинарски рад*

Системско програмирање

**MEMORY PROFILING У .NET-У**

Алекса Станковић

18485

Ниш, 2024.

**Увод, проблеми и сама тема рада**

Ако би смо узели да разговорамо са тимовима који се састоје од разних програмера о профилинг-у да ли код перформанса или меморије, неко уобичајно каже, „Немамо доволје времена да профајлујемо наш код, само учитај лоад тестере“ или нешто на принципу „Ако ради споро само додај још један сервер“. Доста програмера виде профајлинг као додатан део рада, кој се додаје као вид већег оптерећена, и predstavlja nam још једнu стрмu liniju za napredak.

Велики број програмера улази у свет профајлинга перформанси и меморије само када нешто пође по злу. То нам се углавном деси током тестирања система, лоад тестера и често (нажалост) у продукцији. Програмери ће скинути једну копију профајлера и покушати да изолирају зашто апликација ради споро или се често крешује. Притисак настаје, и сата је веома незгодно време да се научи/спреми потребна вештина за ефективно профајлирање.

Користити алате за профајлинг да потражимо нека потенцијална уска решенја током израде може да знатно смањи број проблема који могу настати касније. Са правим алатима и тренингом, ово може постати регуларни део процеса израде без неког превеликог трошка.

**Начини профајлирања**

Профајлери враћају перформансе и меморијске информације из . апликације на један од ова три начина:

* Семпл (узорак): Апликацијска функција кал (call) стека је периодично снимана да би се добио мањи трошак, али са мањим трошком дође и једнако мања анализа
* Догађаји: Битна информација на извршавању, ЦПУ, меморије и скуплјача ђубра може бити скупљено користећи овај механизам, кој шаље нотификације специфичним профајлерима ДЛЛ-а.
* Инструментално: Код који мери апликацију је додат на рантајм, који може да нам обезбеди доста детаљан и прецизан резултат, али зато долази са великим трошком

**Профајлирање меморије**

Начин на кој пишемо наш код директно утиче како и када ће објекти које ми креирамо да се алоцирају и униште. Потребно је да се што „тачније“ одради, и да ће наша апликација користити меморију што ефективније, са минималним утицајем на перформансе наше апликације. Ако па не одрадимо како треба, наша апликација ће користити више меморије него потребно, што може узроковати менаџера меморије да ради више него што је потребно и овај процес ће директно утицати на перформансе.

Може бити чак горе од тог, наша апликација може да настави да алоцира меморију док је више нема, што би узроковали да се наша апликација крешује. То спада под „цурење“ меморије, што је страх свакоког програмера.

Проверити да апликација нема „цурење“ и да користи меморију ефективно, заједно са поправкама било којих проблема на које наињемо, ће наравно побољшати генералну стабилност и перформансе.

**Скупљач ђубра**

.НЕТ мемријски менаџмент модел осигурава да било који алоцирани објекти који више нису у користи од стране апликације ће бити скупљени аутоматски. Ово олакшава програмера од одговорности ослобађања меморије експлицитно, што је нешто што је често било природно Ц и Ц++ апликацијама, долазећи до „цурења“ меморије.

**Приступ профајлера**

Сви меморијски профајлери ће пратити инстанце алоцираних класа. Неке ће пратити траку алоцираног цалл (call) стека, који значи да могу да јаве за профил алокације једне функције и идентификовати функцисјке вруће пределе „ hotspots“.

Та споспобност да се виде алокације, у смислу обе класе и функције, може бити доста корисна. Али, памтити све алокације цалл (call) стека може бити веома напето за меморију и може понекад да лимитира величину и комплексност апликације која треба бити профајлирана.

**Симптоми меморијских проблема**

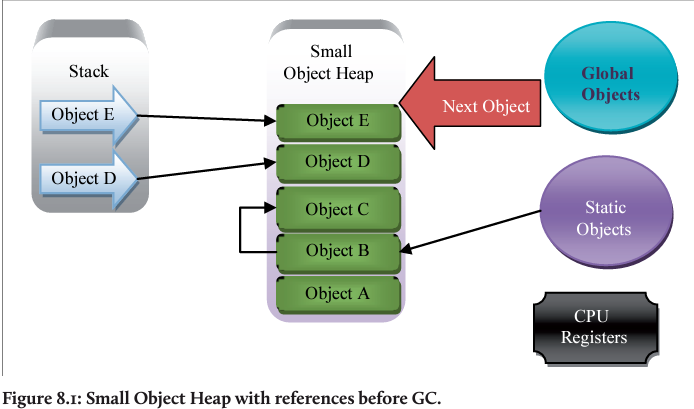
* Меморијско цурење
  + Коришћење меморије се полако повећава
  + Долази до дегредирања перформанси
  + Апликација се замрзава или крешује и захтева рестарт
* Непотребан меморијски отисак
  + Апликација је превише спора да се учита
  + После учитавања, друге апликације иду спорије него очекивано
* Недовољна алокација
  + Апликацијске перформансе се изненадно дегредирају и онд убрзо враћају у нормали.

**Мали хеап објекта**

Нови објекти су алоцирани на СОХ узастопно, референца по имену „Next Object Pointer“ (NOP) је задржана, индицира где ће следећа алокација објекта бити постављена.

Када се нови објекат креира, додат је на СОХ код НОП локације, што би, за узврат, инкрементовали за величину објекта. Алокација је брзо зато што само укључује неке адресне аритметике.

Како се извршава апликација, објекти се креирају, корисе и евентуално одбацују. Када ГЦ, покрене, одбачени објекти су скупљачи отпада.



**Велики хеап објекта**

Било који објекат од 85К или више, је алоциран на „Large Object Heap” (LOH). Насупрот СОХ, из разлога што трошкови копирања великих делова меморије, објекти на ЛОХ-у нису компактни.  
Када потпуна (генерација 2) ГЦ се деси, адресни домет било којих ЛОХ објеката није у корен листи, додаје се на „слободно место“ табли алокације.

Када се нови објекат алоцира на ЛОХ, слободан простор табеле се проверава да се види ако има адресни простор довољно велик да задржи те објекте. Ако има, објекат је алоциран на почетном биту, и празан простор се опоравља.