Nimrod's Garbage Collector 0.9.2"The road to hell is paved with good intentions."

Andreas Rumpf

May 21, 2013

1 Introduction

This document describes how the GC works and how to tune it for (soft) realtime systems.

The basic algorithm is *Deferrent Reference Counting* with cycle detection. References on the stack are not counted for better performance (and easier C code generation). The GC **never** scans the whole heap but it may scan the delta-subgraph of the heap that changed since its last run.

The GC is only triggered in a memory allocation operation. It it not triggered by some timer and does not run in a background thread.

To force a full collection call `GC_fullCollect`. Note that it is generally better to let the GC do its work and not enforce a full collection.

2 Cycle collector

The cycle collector can be en-/disabled independently from the other parts of the GC with `GC_enableMarkAndSweep` and `GC_disableMarkAndSweep`. The compiler analyses the types for their possibility to build cycles, but often it is necessary to help this analysis with the `acyclic` pragma (see acyclic for further information).

You can also use the `acyclic` pragma for data that is cyclic in reality and then break up the cycles explicitly with `GC_addCycleRoot`. This can be a very valuable optimization; the Nimrod compiler itself relies on this optimization trick to improve performance. Note that `GC_addCycleRoot` is a quick operation; the root is only registered for the next run of the cycle collector.

3 Realtime support

To enable realtime support, the symbol useRealtimeGC needs to be defined. With this switch the GC supports the following operations:

```
proc GC_setMaxPause*(MaxPauseInUs: int)
proc GC_step*(us: int, strongAdvice = false)
```

The unit of the parameters `MaxPauseInUs` and `us` is microseconds.

These two procs are the two modus operandi of the realtime GC:

(1) GC_SetMaxPause Mode

> You can call `GC_SetMaxPause` at program startup and then each triggered GC run tries to not take longer than `MaxPause` time. However, it is possible (and common) that the work is nevertheless not evenly distributed as each call to `new` can trigger the GC and thus take `MaxPause` time.

(2) GC_step Mode

> This allows the GC to perform some work for up to `us` time. This is useful to call in a main loop to ensure the GC can do its work. To bind all GC activity to a `GC_step` call, deactivate the GC with `GC_disable` at program startup.

These procs provide a "best effort" realtime guarantee; in particular the cycle collector is not aware of deadlines yet. Deactivate it to get more predictable realtime behaviour. Tests show that a 2ms max pause time will be met in almost all cases on modern CPUs unless the cycle collector is triggered.

3.1 Time measurement

The GC's way of measing time uses (see `lib/system/timers.nim` for the implementation):

1. `QueryPerformanceCounter` and `QueryPerformanceFrequency` on Windows.
2. `mach_absolute_time` on Mac OS X.
3. `gettimeofday` on Posix systems.

As such it supports a resolution of nano seconds internally; however the API uses microseconds for convenience.

Define the symbol `reportMissedDeadlines` to make the GC output whenever it missed a deadline. The reporting will be enhanced and supported by the API in later versions of the collector.

3.2 Tweaking the GC

The collector checks whether there is still time left for its work after every `workPackage`'th iteration. This is currently set to 100 which means that up to 100 objects are traversed and freed before it checks again. Thus `workPackage` affects the timing granularity and may need to be tweaked in highly specialized environments or for older hardware.