CosmoS

Cosmos-B

GARBAGE COLLECTION

*Cosmos-B is a managed code operation system designed to run on the Cosmos kit. This document describes the garbage collection system. .*

GARBAGE COLLECTION

Ben KloosterMan

*This document describes garbage collection management as well as in application memory management*

# 0.Overview

This document describes Cosmos-B garbage collection. Garbage collection is a difficult problem for a managed operating system since pauses can’t be avoided like traditional systems. GC also increase memory usage and reduce performance. Since the whole system uses Collectors we need to ensure drivers and systems (such as the TCP/IP stack and File systems have decent performance and few if any pauses)

# 1. SINGLE OR MULTIPLE COllectOR

Large GC doesn’t work due to the number of thread stacks needed to be scanned. The Stoppage time for a concurrent Mark/Sweep is

A\*#Threads + (B +C) \* Memory \* Memory Volatility

Where A is the time to scan the stack

B is the sweep cost

C is the compacting cost

Obviously a GC serving a process (SIA) will have smaller memory and lower thread counts to scan. While a single GC can be broken up it will have to segregate via thread (in which case it is the same as a GC per process (SIA) but with much longer stoppages for compacting. (Note it can’t really be broken up in terms of memory as all threads will need to be stopped which would not be desirable) . Hence we use multiple GCs.

In terms of running a global pauseless collector, some of these have significant performance issues [Site] meaning a HW protected scheme would be superior. Running pauseless collectors for critical sub systems is viable. Even partially pauseless systems like ref counting would incur a massive overhead eg for each reference memory access you would now incur an additional 2 \* (null comparison and Memory Locked Interlocked Increment) for each reference. With the addition of increased heap size ( and traverse search time)

Note multiple processes or even the whole machine may share a GC for specific cases (say a single app educational or embedded system). It is just not viable for a generic system.

# 2. ISSUES

As some applications can be broken down into reduced services we are left with a number of problems these are….

Large object spaces

Pauseless requirements

Real time requirements.

What makes things worse is the CPU to memory performance ratio is increasing which makes GC pauses worse as they are based on memory walks.

Large object spaces are dealt with by trying to break them into smaller spaces or alternatively load balance those services.

Research into “pauseless” collectors is improving with write barrier collectors such as MS Chicken . While such a collector is trivial making one with decent performance is still a challenge so should be used as a last resort.

Real time collectors are possible but the pause is proportional to the size of the object space and hence the guaranteed time of a real time STP is limited by the size of the Heap. For small heaps or static allocations you can expect small guaranteed times.

It is important to note that pauses time, pause frequency, memory usage and CPU usage of collectors can all be traded to get the right result

# 3. ALLOCATORS, COLLECTORS, NURSERIES AND HEAPS

Allocators, Collectors, Nurseries and Heaps manage the objects. Every STP has an allocator which may be shared with other STPs; each allocator has a collector (which may be shared with other allocators). The collector determines the memory accessible to the application and memory outside of this should result in Memory violations.

The allocator creates actual objects in the Heaps (or Nursery).

# 4. Garbage COllection SCHEMES

The following GCs will be used

* Concurrent Generational Mark Sweep ( 4)
* Ref Counting (5)
* Pauseless with Write barrier (6)
* User Managed GC (7)
* Mark Sweep (8)

All GCs will fire events relating to new memory allocations or when they are about to conduct a pause. It is likely that the first GC built will be a Ref counting one followed by a Mark Sweep.

It is very important that the system allows admin and application to change the collector to get appropriate behavior.

Eg

Number crunching – Mark Sweep

Mainly static content - Ref counting

3D – Pauseless with write barrier or User Managed

Web Server – User Managed

Generic – Concurrent Generational Mark Sweep.

# 5. CONCURRENT MARK SWEEP

The default collector will be a concurrent generational mark sweep which seeks to make use of spare threads and will try to pause on idle signals from the application but will pause when certain thresholds are reached especially when the system is under memory pressure. The Mark is done by an idle thread but requires a short pause for the thread stack scans. The Sweep (and compaction if required) will require all threads to be stopped. 2 Configurations one for client one for server ( server collects more frequently resulting in smaller pauses but more frequent pauses)

# 6. REF COUNTING

The ref counting GC like pervious ref counting GCs have poor performance [Site ] and still require a mark sweep when there are cycles or for compaction both of which require pauses. The main advantage is very little memory usage is wasted and in cases where there are few loops eg device drivers where there is few loops, little change in references ( things tend to get allocated and destroyed) it is good alternative to manual management. It is provided as it is easy to implement though does require a different binary set with references and dereferences embedded in the code. .

Beside the simplicity of implementation a ref counting GC has the advantage when there is little memory churn as it provides little overhead .

# 7. pauseless With write barrier

A pauseless collector with a write barrier will be implemented. Eg The Azul system one http://www.usenix.org/events/vee05/full\_papers/p46-click.pdf or Microsoft Research Chicken Collector.

Latest research shows on modern architectures ( and with assembly tuned code) write barriers on P4 is about 1.31% and its worth noting most mark and sweeps employ an efficient read barrier anyway so checking if a collector is active on this page and pausing the current thread would be cheap. The overhead of an unconditional read barrier was about 5% and 16% for a conditional read barrier. *[Barriers: Friend or Foe? 2004 Blackburn & Hosking]*

# 8. USER MANAGED GC

The user managed GC is a modification of the Mark Sweep where the Mark algorithm calculates the amount of memory marked and hence we can determine the amount of memory to be cleared by a free and the amount defragmented by a compaction. It communicates via events and will continue allocating memory until the server memory pressure is high at which point it will notify the application and force collections. The application can request collections based on the amount of memory that would be freed and whether the program can allow the pause. Please note this is also a diagnostic GC as it will report the memory activity of an application after which the sysadmin could select a different GC.

# 9. Mark Sweep

A simple and CPU efficient Mark Sweep collector with low CPU overhead, but without a large Nursery will consume significantly more memory.

# 9. HEAP USAGE

Collectors will change heap usage depending upon the system memory conditions , so that when the machine has plenty of memory then fast but memory inefficient schemes are used yet when the machine is under memory pressure there are more compactions and slow but efficient best fit algorithms are used. This provides a similar capability to page swapping ( eg more memory but worse performance) .

# 10. PINNED AND LARGE OBJECTS

Pinned and large objects present a challenge as they will lead to fragmentation if pinned for a long time. To deal with these ( as well as permanently pinned and shared objects) these will be created in 2 secondary heaps.

Large Object Heap – is a 1k granularity heap it prevents the main system MM being flooded with small requests or prevents the waste of memory due to large granularity.

Fixed Object Heap – is a object heap with 4 byte granularity.

Both these heaps use the same code as the main memory manager and use best fit algorithms.

Each GC will keep a list of these objects and will manage these for the application.

[TODO move to memory]

# 11. Device DRIVERS & Buffers

Device drivers will often self manage their memory , objects can be placed in a buffer and reused when needed. To deal with a common variant of this , the system provides a BufferManager and Buffer classes to manage some buffers. This basically works by allocating a single continuous block of memory from the GC and then managing it via unsafe pointers. The Buffer Manager allows the device driver to release sub parts of the buffer as well eg a TCP/IP Packet would only pass a pointer to the body ( byte[]) to the application or socket stream while exposing the header to the TCP/IP stack . These buffers cover many things disk blocks, Vertex Buffers etc.

[TODO Move to memory ?]

# 12. GLOBAL NURSERY

The system will also include an optional global Nursery.

This puts 10% of System Memory into 2 Nurseries so on a 4 Gig systems that’s 2 \* 200 Meg.

[Inline]

Allocator  ( note this is all it does and is the same as the current planned alloc it will prob be 3 hand code x86 instructions )

                              Interlocked Compare alloc request to size to mem left , if empty call Nursery empty                        I

                              Interlocked Add  size of object to Nursery pointer

                              return nursery pointer

NursuryEmpty()

              Critical Sectiion

              {

              If ( all process have not finished with backup Nursury)

                              Force remaining processes to update with a stop the world.

             Set backup Nursury pointer to start. ( this is the Sweep) .

              Swap Nursury 1 and 2

              Call background update Nursury

              Return

              }

BackgroundUpdateNursury()

              Call the collector for each process active since last collection pref when they are idle.

              Reset pointer on Nursery when all collects done..

ProcessCollectGenration1

              //Normal tri colour  Mark and Sweep

              Find all references in stack and  Heap that point to the Nursery.  Note the size is at the reference location.

              Move these items to the Heap

              Signal complete to Background Update Nursery.

The whole idea is to exploit parallelism more and reduce collects, and give objects more time to expire and hence not incur any collect cost.. (e.g. avoid the situation where under memory stress you use more collects and hence make the problem worse).

Should make the new () operation about the same speed as the stack this will be very nice for strings. This is especially useful where there are lots of short term allocations (string work, functional languages, XML, Web pages).  You can allocate and use 400 Meg of strings fast! It is not useful for situations where you run a single large heap.

Another thing to watch is for contention around the allocation. While this will be an Interlocked instruction it does block memory usage, on most machines this will not be an issue as a single or a few apps will dominate the allocations. On a many core system running many applications at once it may be advisable to turn off such a Nursery.

There is a valid argument that with such a Nursery a generational Mark Sweep adds little value compared to a Mark Sweep. This needs to be investigated.

# 13. 3D Game with GC

Let’s look specifically at 3D games as these present the biggest problem due to a Large mutable object space and the need to render pauseless. Non rendering pauses of up to 100ms could be tolerated.

App Video Driver -> App 3D Engine -> 3D App

The 3D engine would probably consider of the following STPs

* Vertex Buffers ( Statically Managed buffers) so no GC pauses
* Rendering Resources (Textures, Shaders etc) also statically managed.
* A separate STP to render meshes to Vertex buffers.

The app itself will have a large object domain which can gain very significant benefits if frequently changing objects are self managed ( eg multiple positions/meshes of a “toon”) are statically controlled or moved to their own STP eg effects.

The choice of GC is also critical there are 2 alternative GCs which will be provided for the default Concurrent Mark and Sweep. These are the pauseless write barrier GC ( which is slower) and the user Managed GC.

# 14. WHEN TO USE A SINGLE GC

A single GC can be used by simply setting all the allocators to the same instance. This has the following impacts.

Pro Con

All IPC is very cheap No system memory protection

Low memory use Large heaps will reduce GC performance ( ie bigger pauses)

GC Stops will cover the whole machine

No tuning for different applications

Device drivers must tolerate pauses

Hence a single GC is useful for non real time systems where memory is measured in Meg instead of Gig eg many embedded apps. In such cases the pauses will be too small to be an issue ( eg 10 -20ms is prob about write for 10-100 Meg)