CosmoS

Cosmos-B

MEMORY MANAGEMENT

*Cosmos-B is a managed code operation system designed to run on the Cosmos kit. This document describes the reasons and priorities for building the OS. .*

MEMORY MANAGEMENT

Ben KloosterMan

*This document describes memory management*

# 0.Overview

This document describes Cosmos-B memory architecture. Since Cosmos only runs applications with Garbage Collectors and are verified memory safe this allows a different memory structure..

# 1.Memory Model

Cosmos-B Runs a flat 32 or 64 bit memory model.

# 2.PAGING AND VIRTUAL MEMORY

Cosmos-B will not support any paging or virtual memory. Paging/Swapping today is almost meaningless.

In particular, the following trends in the behavior of underlying hardware and user-level software has affected the performance of page replacement algorithms:

* Size of primary storage has increased by multiple orders of magnitude. With several gigabytes of primary memory, algorithms that require a periodic check of each and every memory frame are becoming less and less practical.
* Memory hierarchies have grown taller. The cost of a CPU cache miss is far more expensive. This exacerbates the previous problem.
* [Locality of reference](http://en.wikipedia.org/wiki/Memory_locality) of user software has weakened. This is mostly attributed to the spread of [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming) techniques that favor large numbers of small functions, use of sophisticated data structures like [trees](http://en.wikipedia.org/wiki/Tree_data_structure) and [hash tables](http://en.wikipedia.org/wiki/Hash_table) that tend to result in chaotic memory reference patterns, and the advent of [garbage collection](http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29) (which visits every data page but not code) that drastically changed memory access behavior of applications.

Requirements for page replacement algorithms have changed due to differences in operating system [kernel](http://en.wikipedia.org/wiki/Kernel_%28computer_science%29) architectures. In particular, most modern OS kernels have unified virtual memory and file system caches, requiring the page replacement algorithm to select a page from among the pages of both user program virtual address spaces and cached files. The latter pages have specific properties. For example, they can be locked, or can have write ordering requirements imposed by [journaling](http://en.wikipedia.org/wiki/Journaling_file_system). Moreover, as the goal of page replacement is to minimize total time waiting for memory, it has to take into account memory requirements imposed by other kernel sub-systems that allocate memory.

Typical issues on modern paging systems

* A 3G machine running about 1.5- 2 Gig of applications with little memory pressure when it swaps to an application not used for 5 minutes takes longer than  loading the application fresh Programmers create bloat , on 64Meg devices where there is no paging there are rarely out of memory errors.
* In Nearly all cases the machine should have enough memory to run the working set if you close your other windows if not you may be able to “run” it on a paging system but it will run so slow as to be unusable.
* Users don’t mind a close a window message it teaches good habits. .  Note this is more of an issue with newer browsers having multiple address spaces/ sandboxes.
* The message can be mitigated by swapping background /unused apps freeing most of the address space. ( Note here users complain to developers , developer swap background window = system less bloated)

Add this to the technical issues

* A Sequential read hibernate is about the same speed as bringing a medium to large app active.
* Power usage is high due to disk usage
* The large amount of caches in typical systems fail  as the LRU  is most likely paging defeating the whole purpose of the cache ( This goes for many caches not just disk)
* The GC issue where it visits every data page and hence will starve code pages.
* Trashing hurts practical reliability of machine ( though it may pass theoretical tests)
* Memory allocation bug or malicious apps/script can kill system unless you impose quotes which few OS use in practice. .
* Disk driver error will kill the system
* Complex interaction between disk , Scheduler and MM increasing bugs

A Garbage collector can act like a swapping pager. If plenty of memory is available it can sweep infrequently and use an inefficient but fast heap. When memory pressure rises it can use a slower best fit heap and do frequent collections. The difference should be about 100% which is similar to a swapping paging system and behavior should be better than a GC and swapping pager ( as the GC doesn’t know when the system is under memory pressure and allocates inefficiently) . Note most paging systems start thrashing around the 100% mark and combine this with the issue of how bad paging/swapping works with caches and GC it would clearly indicate there is no case for page based swapping.

# 3.SWAPPING

Processes are persisted and can be swapped to disk (or USB) . There are a number of features to assist this including developer aids and UI queues. Again the hope is process will be active or dead (note an application may be broken down and have a small active process (like a display timer) in memory and the other processes swapped out.

# 4. MEMORY BREAK DOWN

The system uses 3 Global segments (though we may add a later FS/GS segment as an optimization). These are

Null:

Code: Read + Execute

Data: Read/Write + No Execute

The OS and standard Libraries are loaded into the top of memory. The rest is data.

MAX Trusted OS libs

.NET standard libs

Other Shared Libs

Data mix of Exe and non shared DLL/Heaps and Stacks

16M (only on ISA )

Reserved for DMA

1M Reserved ( for bugs)

0

# 5. CODE - TYPE/ASSEMBLY LOADER

They system libs and frameworks are loaded into the system before the type loader. When the type loader loads it adds these dlls to its code ( so it can patch addresses) .

The type loader when requested loads all code . It treats code as data and uses the Data Segment to place the code into memory.

See process for more information the type loader sets the memory manager to prefer the top of the segment and hence most code will end up at the top of memory.

Note preloaded Shared Dlls will be unloaded if no application uses them for 5 minutes.

A separate memory pool could have guaranteed this but having a single space seemed superior especially as Dlls will be unloaded when not used fragmenting the space.

# 6. DATA

Data is all unused ( i.e. not reserved , trusted libs and Shared Lib Code) memory and merges code/text, data and stack it is divided namely:

6.1 Stack

Stacks are allocated out of data. [Via a Sub allocator ??]

6.2 Shared Memory

Type 3 Shared memory and Type 2 immutable objects will also be allocated via a sub allocator.

6.3 GC Pages

GC pages and large objects are allocated out of Data.

6.4 Large GC objects

Objects over 80K are statically allocated.

6.5 DMA

DMA must be able to be placed with certain limitations eg < 16 Meg for ISA , < 1 Gig etc.

6.6 Exe

Exe’s and not shared dlls

6.7 Cache

All unused Data is used for a read-only cache buffer when memory is needed Buffers are thrown out immediately. These buffers can be any readonly data including web pages or Network packets. For example a web server can reuse the network encode byte data and just pass it to the tcp/ip stack each time ( eg for a 200 Ok response ) .

Data uses the default memory management setting which breaks segments at the bottom and hence allocations will generally be at the bottom of memory.

# 7. RESERVED

Reserved memory is memory reserved by the Hardware or the OS , it includes hardware that uses Shared Memory such as video cards.

# 8. STACK

The system uses lots threads , threads and processes are very cheap , the only overhead is stack space which is significant to handle this threads will have a stack of only 128 Bytes which is expanded and contracted when needed. Whether this is done via Stack segments or copying the stack to a bigger or smaller block is to be decided. In a VM paged system it’s trivial to map a new page into the stack but the cannot be done in CosmosB though we do have the advantage that a stack segment can be smaller than a page.

Either

1. Some mechanism must check for the end allocate a new segment in a non contagious address and the stack must allow a disjoint stack eg increasing the SP is not a trivial addition.
2. A new stack is allocated and copied.
3. Some room for expansion is left and once exceeded ( or the memory reclaimed by the MM due to memory pressure) the other methods are used.

# 9. COSMOS ALLOCATIONS

It is worth noting that Cosmos-B has 2 distinct differences from most systems

* Allocation requests will be large blocks ( Mainly from the GC)
* Allocation requests will be infrequent. (As the GC/Sub allocator will take most of the load).

# 10. MEMORY MANAGER

The memory manager uses the Buddy algorithm, Smallest request depends on memory if

|  |  |
| --- | --- |
| Memory | Minimum request |
| <= 128M | 1K |
| <= 1G | 4K |
| >1 G | 16K |

Allocations are granted in binary powers of 2 increments eg a 17k request will be given a 32k block. Sub allocators ( eg shared memory ) can use best fit.

Each Memory Region has a pointer to the next and previous allocation. Memory Regions are stored in buckets. Bucket 0 is the minimum size, Bucket 1 is the minimum size \* 2. On a 4Gig system (with 16K minimum allocations) there will be 16 buckets.

The bucket to use is given by shifting the request size right by 14. If there are no entries the system will look in consecutively larger buckets until one is found. When one is found it is broken up to meet the request and the broken up parts inserted into the buckets.

When a block is broken Memory is allocated from the bottom unless the break from top flag is set.

A Large (eg 1 M) block size Allocation scheme with sub allocations was considered but would be wasteful on smaller systems

Note while GCs will be buddies there may be a sub allocator which takes large buddy pieces and hands it out for large memory requests as these tend to be poor fits and using a sub allocator may releave the duties

# 11. MEMORY VERIFICATION

Any pointer or IO instructions will fail verification, trusted assemblies are not checked.

CODE CHECKS

We don’t believe any runtime verification tests will be needed for static or VMT calls , jumps and farcalls since a name will be replaced with the address at load time. Delegates and other runtime modifiable call structures will need to be wrapped and when set a check whether the memory is valid.

Compile time checks will check for all branches and local jumps to ensure they cannot leave the current method. If these cant be checked at compile time they will be checked before the branch.

DATA CHECKS

All data is on the stack or the heap. References to the heap cannot be created if they are validation will fail. All references must be obtained via newObj ,box or field instructions . [TODO check!]

STACK CHECKS

. There should be no stack checks needed as threads do not retain stacks across IPC

# 12.SECONDARY MEMORY

Cosmos will assume Flash as the primary form of secondary memory storage. This is common for hand held devices and can be added to any PC. Flash will be used for disk cache and swapping.

# 13.Managed Memory

### Forever Fixed (Immovable)

C# supports a fixed statement, however it is a block construct. Cosmos will need an attribute that can mark a variable as forever fixed, i.e. unmovable. This variable would perform in a similar manner to a Mapped Variable, however in the case of a Forever Fixed the allocation would be taken care of by the memory manager. That is Cosmos decides where the variable is. The memory manager should of course treat these variables differently than movable variables and likely use a different memory space.

Forever Fixed variables are available to all types ( and fields ?) .

### Locked

Locked variables function like Forever Fixed, however are meant at some point to be unlocked. If code knows that the variable will eventually be locked it should hint using an attribute that at some point the variable will be locked. This will allow the memory manager to optimize its allocation.

For example user code may create a variable with the hint "Lockable" and initialize data. Later this variable is passed into a file system object, then a storage hardware driver. The storage driver needs to pass it as a memory block to physical hardware for DMA access. At that point it would be locked. When the DMA is completed, the variable can be unlocked.

### Endianism

Any variable that is mapped to memory must deal with the issue of [endianism](http://en.wikipedia.org/wiki/Endianism). Because of this any managed memory variable that is not byte based must also specify when endian pattern it uses. X86 uses little endian, while network byte order is big endian. So for variables on X86 platforms that are little endian, no transformation exists. For variables declared big endian, each and every read and write must perform the transformation. This transformation is done in assembly and automatically emitted by IL2CPU. Since IL is stack based, this is rather easy. On each push/pop from the IL stack proper assembly can be inserted to alter the variable as it is read and written from CPU registers. On x86 the BSWAP operation is used.

Optimizations should exist in the optimization engines to avoid going from big to little back to big in cases such as assigning one big endian variable to another. Other cases surely exist and should be examined.

### SERIALIZATION

It’s likely that Serialization will require some trusted function as it needs to break type safety.