CosmoS

Cosmos-B

IPC OPTIONS

*Cosmos-B is a managed code operation system designed to run on the Cosmos kit. This document describes the IPC and shared memory.*

IPC & SHARED MEMORY

Ben KloosterMan

*This document describes the IPC architecture and shared memory*

# 0.Overview

IPC in Cosmos B is probably the area where it will diverge the most from current systems. We aim for an order of magnitude improvement in IPC time due to the fact the only dispatch in the system is a Thread dispatch which we term “Task switch” to distinguish it from the current dispatch mechanisms even though the system is pre-emptive.

Key features of Cosmos-B IPC are

* More secure messaging via pre secure point to point queues
* Less opportunities for manipulating data as messages are strongly typed and can be verified at a single point.
* No locking IPC , enhancing performance especially on mutli core systems
* IPC based APIs are easily extensible with good backward compatibility
* Asynchronous support again improving multi core performance
* Quick hand over to STPes waiting for messages reducing latency
* Bulk STPing of messages allowing for higher performance due to cache coherency and thread affinity.( At the cost of latency)
* Zero copy from Hardware to User.
* IPC and ABI ( kernel calls) are the same

The extra performance will be used to improve multi Processor scalability, security as well as reliability. The IPC works especially well with functional languages which use immutable structures and are asynchronous.

Lastly the system will contain a full publish subscribe event system.

# 1.IPC/Shared Memory in MANAGED SYSTEMS

IPC in managed systems in theory is as simple as placing a reference to an object in a predetermined address and then the receiver can use that object. In practice there are a number of difficulties. Namely

* When the 2 parties have different GC’s the objects need to be carefully managed. This creates a range of issues.
  + - When one STP is killed what happens to the reference or object.
    - How do you handle references being added to and from different GCs into the object?
    - How do you handle GC compacting?
    - How do you track stack reference to the object.
    - How do we manage these references with memory checks
    - How to prevent dangling pointers eg duplicate a reference then set the original to null or tell the creator your finished.
* Thread Safety needs to be considered like current systems, which leads to locking.
* How to ensure security in that the object may lead to a significant security breach.

# IPC and SHARED MEMORY

Let’s look at the schemes

1. Copy on send requires small compact fixed length messages ( for efficiency) which interferes with a type safe system and general code quality. Best multi Processor support.
2. Parameter passing/RPC style results in chatty tightly bound interfaces that are hard to maintain. This especially a concern for an API which should evolve. In addition there are many points of entry . Not Asynchronous friendly and poor multi Processor performance. ( The stack is normally copied to the destination)
3. Sending a message provides type safety and easy maintenance ( a derived message can be sent in a new lib with no changes to the API) . Messages may be strongly validated at a single point of entry. The main issue is the message creation cost. Requires some form of shared memory.

The goal of the system is better security and reliability and the type safe messages clearly puts a strong case forward in this area. The use of shared memory is unlikely to have a significant cost (as it is established for the life of the SIA) . We hope that messages will become larger and more infrequent.

The cost of instantiating the message should be trivial as we use the NoConstructor Option , hence the cost is merely doing an Interlocked increment on the Nursery pointer and adding the type id to the object. For non Numa systems performance is likely to be the best for Asynchronous SIAs (like web servers) and especially multi cores.

# IPC -MESSAGE BASED, QUEUED and ASYNCHRONOUS.

One of the most important decisions to be made is whether to have Synchronous or Asynchronous IPC. And what method will be used for kernel calls (ABI).. As discussed the system will have no ABI ( except for syscall) . All IPC will be point to point strongly typed messages and Asynchronous.

# SHARED MEMORY

Shared memory is the basis of Cosmos-B IPC. There are 3 levels of shared memory.

***Level 1 (Reference Passing)***

For objects sharing the same address space (and hence Garbage Collector) STP can directly pass references to other STPes.

***Level 2 ( Data Exchange and Object Exchange)***

Level2 is the main form of IPC /Shared Memory Comos uses it relies on sending Immutable references and ensuring they are not destroyed by a GC nor become dangling pointers.

There are 2 variations Data Sharing where sub trees of immutable data are sent and object sharing where an immutable object is send which can interact with data of the sender.

***Level 3 (Shared Memory)***

Level 3 is true shared memory; the shared memory has its own object space and garbage collector. References to the shared memory are controlled through a control block and references from shared memory to an outside region are illegal. It’s possible we never implement level 3 if level 2 proves adequate though level 3 promises better performance with a large amount of STPes sharing and modifying the same data.

# ISSUES WITH Shared Memory

For non copy on write Shared Memory there are a number of issues namely.

All IPC needs to handle thread safety.

When using the Senders GC ( and the receiver has a different address space/GC)

* If it is managed by a GC you need to ensure it is not prematurely collected. Any such collection will break type safety and potentially security.
* You also need to consider what happens to the references when the sender(GC) is killed.
* You need to ensure the GC is not tied to other GCs which could propagate and cause a massive machine wise pause..
* The receiver may pass the reference to further STPes.
* The original sender eventually needs to be notified. ( hence this form of Shared Memory always requires a response)

When using a Shared heap.

* Who controls the heap
* State of data when crash occurs
* Duplication of references

Its worth restating the other forms of IPC/Shared Memory

* Copy on send , not really viable with buffers and other significant data
* Copy on write requires paging and VM or complex compiler changes and a bit of issues with managing the memory.
* Monolithic 1 GC and send reference approach is not viable due to massive GC pauses (which could result in HW IO issues)
* Traditional shared memory ( a self managed byte[] ) is not viable as it breaks type safety and can cause the whole OS to crash ( note there were MANY bugs in Linux here)
* Singularity uses a shared heap and ensures ( via run time checks) there is only a single pointer into the data owned by a single owner( and the shared data has no references to and from other memory)
* Using copy on send for IPC and treating shared memory separately.

# Level1 REFERENCE PASSING

STP can directly access sub STPes memory (and could pass a pointer to itself) this is completely controlled by the STP. Note it is only possible where the sender and receiver share address space (and hence GC ) I this normally occurs within a STP trees. Thread Safety is still required.

# Level2 Data/OBJECT EXCHANGE

Level2 is the main form of communication Comos B uses and we rely on sending Immutable objects controlled by the GC of the sender. The references are sent to the destination STP they are immutable and hence so no locks are required on the object.

All such objects are IImmutable and they have some special properties

* It’s legal to have references to such objects outside of your address space (which is not the case for other user generated types).
* Objects so marked are checked at compile time to ensure they have no public fields or set properties. Method return types (including ref and out) must also be immutable objects.
* They are treated as fixed by the GC once the reference is sent to another STP.

A lib contains a number of these objects for common usage such as ImmutableArray<T> and ImmutableString. Even though strings are immutable it’s better to use a separate type for IPC.

The only objects that may be sent are IImuttable . Structs coud be passed by value but rather than checking for a struct and IImmutable everywhere its better for an IImuttable object to contain the struct since there are boxing issues.

The 2 sub types are:

**A.Data EXCHANGE**

Data Exchange sends data as sub trees of data. The immutable data trees are created via the constructor of the objects and need to be created leaf first in order. This is the best way to send data between un-trusted sources.

**B.Object EXCHANGE**

Sending Objects with methods such as eg BufferReader . In this case an owner of data can let other STPes act on data via an object provided by the owner. A good example is a BufferManager service may allow streaming reads of a buffer via a BufferReader object. The object needs to ensure all data returned via methods or events are immutable even though the Buffer does not need to be. The object also needs to ensure thread safety if it is manipulating data. Such an object allows the reading or writing of data with no dispatch.

Note many Data Exchange objects have a method that tells the caller they are finished with the data in which case the dummy reference will be released..

Methods in such an object must be careful as they are modifying data via a remote thread and GC . [Consider a using block to change GC]

# Ensuring LEVEL2 OBJECTS ARE IMMUTABLE

The system ensures these objects to avoid strange bugs

Objects marked with the attribute are checked at compile time to ensure they have no public fields or set properties. Method return types (including ref and out) must also be immutable objects.

A TCB system lib will also allow a factory construction such as Immutable.Create<GenType>(params);

# LEVEL2 OBJECTS IMMUTABLE OBJECTS CLEAN UP

When sending a reference the reference is automatically cached locally this “dummy” is needed so the GC does not collect the object.

When a STP is killed or exits all counterparties (i.e. STP which can be reached via an IPC channel) must be sweeped by the kernel and references to object in the address space of the finishing STP must be replaced with null. If in the sweep a dummy object exists than the STP chain needs to be followed. (This handles the issue of a reference being sent through multiple STPed).

The last condition that must be met is that when the receiver signals that it is finished with the object there must be no other references ,this can be achieved by a variety of means. An object can’t be released if it has been sent to another STP (it’s in the proxy) . To prevent this we can do one of the following.

1. Do a sweep similar to STP termination setting all references to null. These may be bunched up and can be synched with the receiver GC mark phase.
2. Ref count these objects and store the count in the object ( which will be set to 0 by the send)

To assist in these things Level 2 Immutable object will be created in a separate buffer ( one for each Pipe) sub allocated from the main Memory Manager to avoid issues with fixed objects in the GC. This buffer will reclaim regions of the buffer and will try to keep allocations at similar speed to a Mark Sweep Nursery. It is important that receiver’s release these objects ASAP if they do not it will additional memory to be consumed by the Pipe as the whole region of the buffer cannot be released until all parts are. [How to create string here instead of GC ? Cumbersome]

# LEVEL2 ISSUES AND OPTIONs

The outline method for 2 has some issues

1. The GC sweep required for release will result in memory being held for a longer amount of time. (Though the overhead is minimal) . This is exacerbated by immutable structures requiring more IPC.
2. Memory being released requires an IPC reply message.

This can be mitigated by a kernel base class for immutable objects which interacts with the local thread GC ( to set all references to null) and then removes the dummy reference of the sender ( directly ). [Implement]

1. The reference needs to be fixed creating issues with the GC of the sender

One option is to use the large memory model ( where the GC allocates it from the system MM) for these structures though but that puts stress on the system Memory manager especially as these objects maybe quite small this in turn can me mitigated by using a sub allocator. [Implement see]

# LEVEL2 OPTION SHARED MEMORY HEAP

A further option is for the shared objects to be created on a shared memory heap (different from option 3 above as it bypasses the sender GC). We then have to consider disposal. There are some options

* + 1. A singularity style single reference enforcement This means no dummy objects are required.
    2. ~~A walk all GC scheme ( requires big pauses)~~
    3. Use ref counting (this is quite easy as the Compiler can insert it for shared memory objects (check via attributes) ) .

Benefits:

* + - Mutable objects possible with some sort of locking mechanism ( ie Level3)
    - Quick release of memory.
    - After receiving a message the receiver can quickly destroy it.
    - Simpler
    - Long lived shared data doesn’t block GC , buffers etc.
    - Fixed objects don’t block GC

Cons:

* + - Slightly lower performance ( due to ref counting code)
    - How to handle a crashed SIA as the ref count will not be reliable ( So need to store STPes with ref count which makes ref much more expensive)
    - All memory not handled by the GC

Such a scheme though desirable needs to address how large objects such as string are created eg what decides if it is placed in the Shared Region or rhe normal GC. We could use a using region for this . .

# Level3-Shared Memory

Shared memory managed through the Shared Memory System. It allows changes to a large amount of data between 2 SIAs.

All such classes must be marked with SharedMemory attributes. Memory allocation is via a shared heap. To use this shared memory a pointer is simply passed to a STP after being wrapped in a control block.. If a STP terminates the state of the memory should be fine if it did not hold a lock, if the owner dies it becomes invalid.

All pointers to shared memory are indirect via a trusted control block and will deny access if the user is not the current owner ie there is only 1 user who has visibility of the shared memory at one time. The pointer is contained in a reference to a control block (this is the indirect pointer) This is the recommended method it is secure and reliable. When the lock is not owned the indirect pointer will automatically return null (with the control block hiding the real reference) . A write barrier or compiler check is used to ensure all references are through a control block and references outside of the shared memory region will throw an exception.

A control block exposes an EventWaitHandle , a method for creating new objects , a lock and the immutable reference to the root object in shared memory.

Control blocks are privileged objects. As the only valid reference is through a control block the only way of gaining access is to obtain a block from the system IPC (which requires the share object in its constructor) and/or cloning. The data structure is only removed from memory when all control blocks have been returned. The current lock owner may also release the memory (setting all pointers to control blocks null)

# uPGRADING FROM lVL 2 TO 1

Its important that SIAs upgrade from Type 2 IPC to type 1 automatically. The same SIA running on a desktop (and hence using type 2 IPC) may run in an embedded system with a single Collector and hence can use Type1. It’s important for SIA designers to note that while Level 1 can support any object by using a Shared/IImmutable object the system can use Level 2 when needed. Conversely if an app does not want an object to be ever exposed he should ensure it’s not a Shared/IImmutable object.

# MESSAGE PiPE (QUEUES)

To allow Asynchronous messaging the system uses fixed length queues containing references to the message. The Queues are point to point and in some cases there may not be a return queue as acknowledgements can be called via methods on the message or not required.

Messages queues have a number of benefits namely

* Messages do not need to contain the Sender information
* Security is decoupled from the messaging system ( It is coupled with the queues)
* Backward API Compatibility is trivial to maintain and API changes do not require a recompile. ( especially using inheritance)
* Choice of low latency ping pong or high latency but more cache efficient batch STPing.
* Better multi Processor support.

It should be noted if the receiver is waiting for the message it gets activated when the message is placed in the queue this allows rapid ping pong IPC for latency critical work.

Queues use Level2 Object Exchange. The receiver makes the reference available to the sender and the sender adds the items. Please note the MessageQueue objects themselves are in the TCB and use pointers to manage the queue.

When a STP starts it normally has a number of queues these are:

* IPC & Events In/Out
* Memory Manager In/Out
* Scheduling In/Out

# MESSAGEs

Message are strongly typed and inherit from IPC Message or IPCRepplyMessage . Most sub systems will implement their own Sub Class eg MMMessage . Message may be validated. Messages are Level 2 Data Exchange Objects. The Message id is equal to the memory address which is valid and unique for the life of the message since it is fixed.

# STRUCT MESSAGEs

The system will support small struct messages . These struct messages must be entirely made of unboxed value types and implement ISystemMessage. It is very useful for small calls not requiring strings. Note while you can send a message with char[] you cannot send a message with string and it is not recommended you send message with char[] unless it is very small and necessary.

These structs must not be boxed so the caller cannot use the interface to refer to them as it will cause it to be boxed. The compiler will handle this case specially ie structs sent to SendMessage (IMessage msg) are NOT boxed instead the code to copy them to receiver is inserted by the compiler.

When an unboxed struct message is sent it is copied and added to the receivers GC and hence boxed.

# EVENTS

The system contains an extensive Publish/ subscribe events system (with security on the subscription). Events work especially well with Level2 Data/Object exchange and user could even configure call backs on their objects.

Modern systems already have a large number of events namely power management and UI. Extending these seems natural and allows for greater system flexibility without any security cost.

The greatest strength of events is the loose coupling and the low performance overhead for multiple subscribers ( as compared to polling for changes) .

Some of these categories are.

* GC Events (About to collect etc)
* Power Events
* Memory Events
* UI Events
* HW Events ( eg interrupts , USB bus connections )
* Scheduling Signals

Events are based on Level 2 Data Exchange Objects and hence don’t require any locking.

It is possible Thread Events will be implemented using system events though I think a Immutable Object Exchange object is preferable. ???

# SenD MESSAGES

Note no need for a Syscall for IPC or send message.

Sending STP

Create Immutable object ( or send if it already exists)

queue.SendMessageAsynch(IImuttable) ; // or use the Synch blocking version

Receiving STP

Var message queue.RemoveNextMessage(); // blocking until message available used ManualResetEvent underneath.

STPMessage(message)

// TCB trusted code part of IPC system.

SendMessageAsych (KernelMessage message)

{

bool yield = false;

if ( multiCore == false)

yield = true;

StoreIPCDummyPointer(message); // store dummy so message is not

If (receiver is blocked waiting for message) // issue with receiver resuming after call so need to grab a lock

{

Set Receiver current message to message

Schedule Receiver to run ( via a private reference to the scheduler)

}

else

{

if receiver running

{

Lock (queue) // real solution can void lock

{

if ( high priority)

Add message to front of destination queue)

else

Add message to back of destination queue

}

}

}

if ( yield)

Sleep(0); // give receiver chance to run on single core

return;

} //SendMessage