CosmoS

Cosmos-B

IPC & SHARED MEMORY

*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. .*

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 processes waiting for messages reducing latency
* Bulk processing 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 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 process 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
* 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.

# 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) process can directly pass references to other processes.

***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 processes sharing and modifying the same data.

# Level1 REFERENCE PASSING

Process can directly access sub processes memory (and could pass a pointer to itself) this is completely controlled by the process. Note it is only possible where the sender and receiver share address space (and hence GC ) I this normally occurs within a process 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 process 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 process.

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 processes 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 ASRE IMMUTABLE

The system ensures these objects to avoid strange bugs. This is done by 2 methods

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.

Access to Immutable object is via a Readon Only memory segment hence any writes will fault. This is done by changing the code from DS: Address to ES: Address where ES points to a read only segment Note newobj is done normally.

[Do we use Code segment for this ? Is it too limiting in construction ?]

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 process is killed or exists all counterparties ( i.e. process 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 process must be replaced with null. If in the sweep a dummy exists than the Process chain needs to be followed. ( This handles the issue of a reference being sent through multiple processed).

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 process ( it’s in the proxy ) . To prevent this we can do one of the following.

1. Do a sweep similar to process 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)

# Level3-Shared Memory [Optional]

To be determined. Initially we only support Level 1 and Level 2.

Shared memory managed through the IPC Shared Memory System. All such classes must be marked with SharedMemory attributes. This forces the compiler to use the shared memory allocator which manages the shared memory region. To use this shared memory a pointer is simply passed to a process after being wrapped in a control block. Locks are placed and all access should be done through locks. If a process terminates the state of the memory should be fine if it did not hold a lock. This method is used by the IPC system ( eg the messages are placed in a level 2 shared memory space and the lock is normally held by the sender ) . A control block exposes an EventWaitHandle. All pointers are indirect 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 contain a reference to a control block which exposes an EventWaitHandle. This is the recommended method it is secure and reliable. When the lock is not owned the indirect pointer will automatically be set to null ( with the control block hiding the real reference) . A write barrier is used to ensure all references are through a control block and references outside of the shared memory region will throw an exception.

# 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.

# MESSAGE QUEUES

To allow Synchronous 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.

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 processing.
* 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 process 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 for the life of the message since it is fixed.

# EVENTS

The system contains an extensive Publish/ subscribe events system (with security on the sub scription) . 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 )
* Process 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 Process

Create Immutable object ( or send if it already exists)

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

Receiving Process

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

ProcessMessage(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)

{

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