CosmoS

Cosmos-B

Overall Architecture

*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. .*

OVERALL Architecture

Ben KloosterMan

*This document describes the overall architecture and describes key components that will be discussed in later documents.*

# Overview

Cosmos B is a managed Operating System . What we mean by a managed operating system is an operating system that runs managed code ( ie type safe and memory safe). Programmers compile source code into a safe virtual machine (CIL/MSIL), which the trusted operating system components compile to native instructions for execution. As a result, the operating system does not need to use virtual-memory or other hardware protection features, as the trusted compiler can assure only safe code is running on the machine, safety is guaranteed by the intermediate language (CIL/MISL) , by checks the compiler makes when compiling the code and possible run time checks. Hence the compiler is also trusted code ( though this is no different from a normal OS where we trust the compiler to generate the correct code) .

The OS is written in 99% C# code with a small amount of assembler (which is actually created via C# instructions).

# Key OS Concepts

The OS consists of the following key concepts that require further explanation.

*Software Isolated Application (SIA)* Consists of a closed type space ( ie only types/classes in specified assemblies may be created , nor may new assemblies be loaded) . It also has a reference to a Collector Domain (which may be shared with other SIA’s) and an appropriate allocator. Actual execution is performed by a collection of STPs (Single Thread process). Hence an SIA provides the VM and isolation that allows applications to be independent from each other it is not an execution mechanism.

* Reference to CollectorDomain
  + Allocator
* Reference to TypeLoader
  + Assemblies
* STPs
* Capability Keyring (see below)
* MetaData

*Single Thread Process (STP)* Processes are execution units. A STP process may contain sub STPs which the parent can call directly using IPC level 0 (and the child can reply if the parent gives it a reference) . These form STP trees. A STP contains a single light weight thread.

* Reference to SIA
  + Allocator
* Light Weight thread
  + Flexible stack
* Message Pipe - Which allows communication to other Processes and contains a message pump
* Root object
* Capability Keyring (see below)
  + Object Capabilities

STPs only contain a single thread for a number of reasons primarily.

* So POLA /POLP is more closely adhered to ( see security section)
* To give threads the smallest possible data / code to work over to reduce stack size, improve cache performance, security and reliability.
* To improve scheduling and affinity on multiple core CPU’s ( esp NUMA architectures)
* To encourage an SOA style load balanced service approach rather than a single large service with 100’s of threads.
* When no work is needed the STP for that work is terminated removing the resource and thread.

While a developer may just use a traditional style with a SIA and a number of STPs over the entire address space joined at main it is hoped to encourage developers to break down applications into smaller units.

*Light Weight thread*

The system uses only light weight threads which have stacks that grow and shrink. This is important as the system will run many threads to encourage better performance. There is a perception that threads are bad for performance but this belongs in the past where HW could only execute a single thread and context switches are expensive. The only issue is with large amounts of threads is stack space.

*Object Capability*: Security management is done via Capabilities. In our case all objects are managed by the owning application or service and the caller maintains a reference. The owner is responsible for ensuring such objects can only be instantiated by itself normally by having a non public assembly and only exposing an interface (which is shared) or just a reference to Object . The capability is checked to ensure it is the correct type which guarantees it valid.

The receiver can persist the object reference in a keyring so the authority can survive. ( See the security document on the details and how the reference is reestablished)

*Service*: A restartable SIA that can be launched at boot and restarted manually when desired.

*Device Driver*: A service that registers with the HAL and manages hardware access.

*Address Space*: The operating system is a single-address space operating system where all addresses are global. Further, because the typesafe trusted core can assure only correct machine instructions are executed, hardware MMU features are not required to enforce security permissions.

*Trusted Computing Base (TCB)*: This contains assemblies which are allowed to do things SIAs are not eg manipulate pointers , perform HW IO instructions , DMA as well as things like accessing any part of memory etc . These assemblies are loaded at boot and access is controlled by who has a reference to the object.

# IS IT A Micro KERNEL, Nano KERNEL OR MONOLITHIC?

What is the Kernel in Cosmos-B? In micro/pico kernels it is defined as the messaging and task switch primitives who run at ring 0. In our case even this doesn’t apply as these are just trusted code modules. There is trusted code and un-trusted but trusted is not a good definition either eg the GC is trusted but is not the Kernel as it sits in the SIA (the app VM) .

If you have a reference to a capability OS you can do trusted things however the OS loader and type safe system ensures only certain components have access to these code assemblies.

You could argue it is Monolithic but since there is no Kernel and calls are not global it is also not appropriate.

Hence Cosmos-B OS is kernelless and best described as a Modular or Object Oriented Operating System. We will refer to trusted code as the TCB.

# Overall Design

Like a microkernel, fine-grained isolation is provided between components. However, in a microkernel, communication between these components typically has a very high cost. Using a single-address space protected by software verification eliminates these high IPC costs, making it practical to provide isolation without performance degradation.

A good example against Micro kernels is the paper “The Cost of IPC: an Architectural Analysis" which contradicts Liedtke's earlier work and argues against Micro Kernels and is based on the following issues getting worse as the CPU: Memory performance ratio increases.

* Heavy message copy costs
* Heavy context switch cost especially by mapping memory and the TLB flush cost.
* Lots of short messages and hence message overhead for a few pieces of data.

None of the issues apply to Cosmos-B.

The TCB is built by the loader of the OS **and any references to trusted objects these initial modules give out**. . There is no user/kernel layer and lastly every SIA is treated equally if the HAL decided to pass or expose a reference to a user application it could access hardware directly and break the design. There is one critical differentiation which we will use to show the system and that is we have a shared code object space and an application object space.

Hence we will show the design as dependency graph and what trusted parts are exposed.

The boot kernel is the Cosmos kernel it gets things loaded using an interim Heap it loads key assemblies. It then copies critical pages to the new GC and passes execution control to the scheduler at which point it will be over written.

1. Trust

The application loader will only load trusted assemblies.

Assemblies are checked when compiled to ensure they do no IO machine instructions, privileged instructions ( eg CLI/STI) or use pointers if they do they will fail to compile.

Trusted assemblies can do anything however they can only be loaded during the OS boot in addition the App loader cannot bind to these assemblies unless it is marked with an attribute that makes it available.

The following trusts are noted

Use pointers – Memory Manager , Collector , IPC

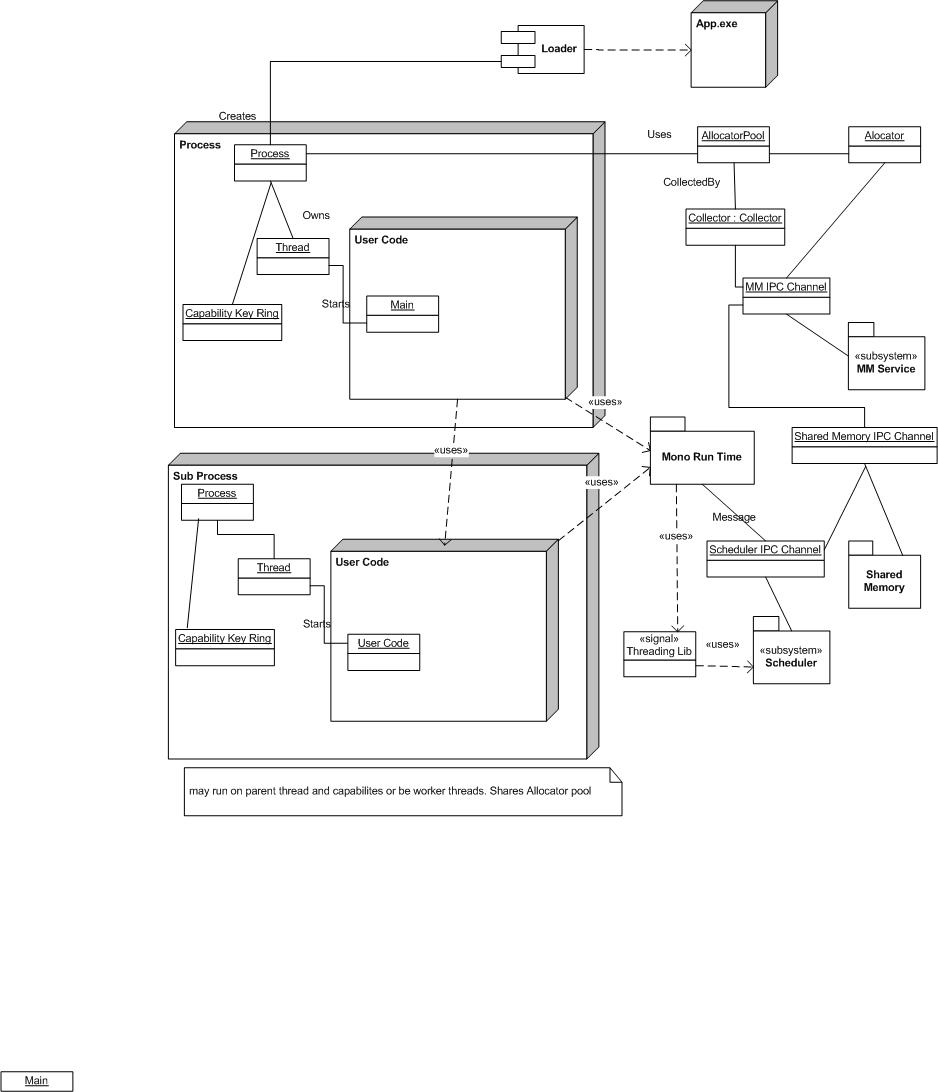
Do IO Instructions ( PIO and DMA) – Machine specific HAL.

Do privileged Hardware instructions ( eg Disable Interrupts , far jump , stack manipulation) Machine specific HAL

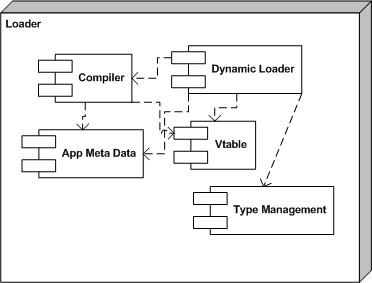
Do privileged Hardware instructions far jump: Scheduler

Unsafe apps are created at application load by the loader giving a reference to use the privileged code . Without the references it cant be used unless there is an expose attribute.

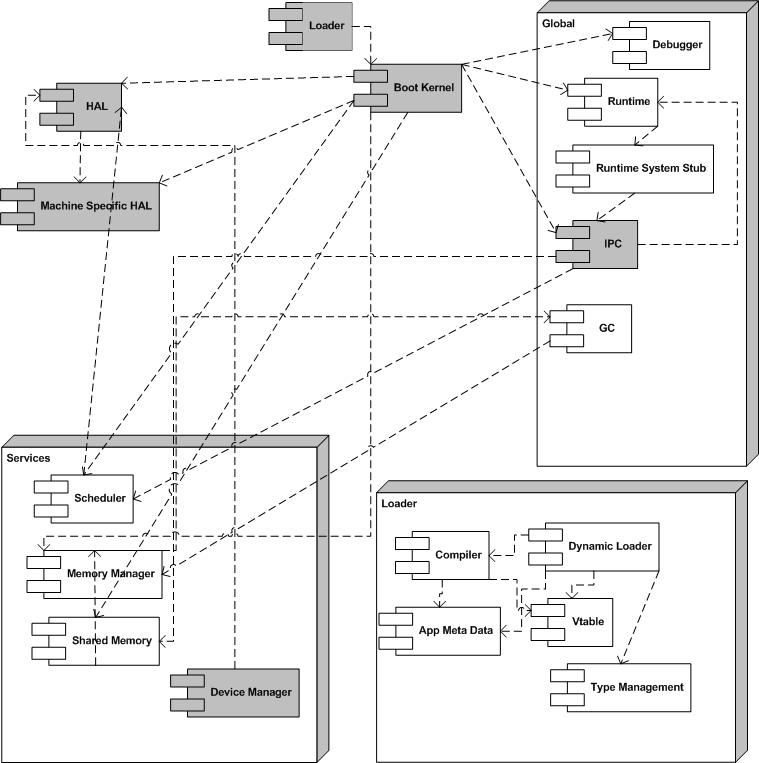
1. SIA



1. Loader



1. Rough module dependencies



# SHARED ASSEMBLIES VS SIA Assemblies

In a traditional kernel a shared-library cannot do any useful security work since anything it has permissions to can be touched by all users of the lib. This is not true in a managed OS. In a managed OS, an Assembly you are allowed to link to may have higher permission level than you. Since access is limited in a number of ways ( eg all internal class with friend assembly , or checking whether the current SIA has the right Object Capability in its keyring)

# SynchRONOUS OR 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).

Jochen Liedtke (*Improving IPC by Kernel Design*) showed that an order of magnitude improvement could be made compared to Mach 3 by using Synchronous IPC and significant other optimizations. The paper “Singularity: Rethinking the Software stack” suggests that this may not be true running when running a Managed Operating system under modern hardware which will only increase as the number of hardware threads increase.

In addition it is quite likely applications used to measure this performance were written for a Synch environment eg single thread with the occasional worker threads.

It is worth noting that high performance systems like web servers and file systems often block synchronous threads and implement an asynchronous model internally via worker threads and I/O completion ports. It has always seemed strange to me that Operating Systems favored unbuffered synchronous IPC for fast IPC yet the hot spot sub systems convert it to Asynchronous. Originally this may have been because memory malloc calls were the most frequent calls but most modern systems use in process pools.

In the past and to some extend to today Synchronous IPC matches the client – server development model. In the past with lots of malloc calls , polling IO etc and most applications were simple applications that just called the kernel for IPC and hence Synchronous calls closely matched the way the system was used. Modern systems are different, web servers serve large amounts of internet requests and can’t afford to have thousands of Synchronous requests , SOA /COM and similar products have resulted in small independent services that are called via IPC and UI and OS events are ubiquities ( and require IPC) . It is also likely that future systems will become more event driven mainly due to smaller occasionally connected GSM/3G devices.

Since we desire large amount of threads to break down the work and better use multi core hardware we have chosen an Asynchronous IPC. The receiver has the option of messages being queued or it can be immediately invoked if it is waiting for them. Each STP will have a message /event pump which the STP will dispatch.

The key advantages

* The Async-Client which does not require a completion acknowledgement are not blocked eg a clock service sending out clock ticks to event subscribers.
* Less issues with Thread starvation
* Better use of multiple Hardware Threads
* Better cache coherency.
* Better supports for multiple cores.
* Encourages applications to be Asynchronous and to create sharable services reducing bloat.
* More secure and reliable model ( *Vulnerabilities in Synchronous IPC Designs*)
* Better performance under high loads.
* Better matches the functioning of events in a modern OS eg IO interrupts , UI events , power events etc etc

We also considered whether to use an ABI for kernel calls or use a Minix style system where all kernel calls are IPC. Consider common uses for ABI calls

* Memory Allocation
* IPC ( eg Message Send)
* Locking control
* Thread Control
* Security /Process Id
* Shared Memory

Pretty much all of them are rarely used and have no performance requirement with the exception of locking control. Hence we plan to implement all calls to standard OS assemblies via Messages using a IPC Assembly which has SendMessages and takes care of things like yield.

Please note there will be a few critical cases where we will use Synch RPC style IPC, but only for a few hot spots. To do this all we need to do is expose a pointer to the appropriate destination and it can call the public methods (on the callers thread!) Note this also requires both sender and receiver to share the same Collector Domain.

# How architecture supports Key Design points

***A kernelless structure where each app is equal hence there is little reason to trust callers.***

The architecture has a kernelless structure; the structure is basically a large object space some components of which are trusted which are woven together by the boot loader. User apps are loaded into this with rigorous testing.

***Strong process isolation architecture***

Processes are strongly isolated any access must be explicitly stated and the compilation /install checks whether all code is valid. All code must be loaded and linked at load time including sub processes. It is possible to load code later as a sub process this would require its own thread and a light weight thread hand off but its undecided if this will be implemented as standard IPC is likely to be sufficient.

***A POLP /POLA Object Capability security model***

The architecture strongly adheres to the POLP and POLA principals though there is some ambient authority required for legacy applications using the System.IO namespace.

In addition NOT giving special privilege to services or the kernel means every application or service must assume its callers are non secure.

# Memory Architecture

The memory architecture by default supports a single non paged, non virtual memory address space. Idle process waiting on a message can be swapped to disk and activated when such a message is received.

. We believe 2 Gig system would run the same common applications as a 4Gig Windows systems. This is based on.

* A single very comprehensive set of shared libraries shared by all applications. A typical Windows sessions will run libraries from a large variety of SDKs resulting in a massive amount of duplication.
* Much greater reuse of code via shared services
* The lack of virtual memory will also have an impact similar to the later DOS application in terms of applications being more memory efficient. These application grew massively using the same 640K memory barrier.
* The lack of page tables.
* SIAs completely swapped out of memory when inactive.
* The lack of copying required in the OS.

Note this does not apply for large single apps like DB servers or 3D games where the memory usage would be similar.

***Shared memory***

There are 3 levels of shared memory.

***Level 1***

STPs can directly access other STP sharing the same Collection domain this is completely controlled by the STP.

***Level 2***

Level2 shares memory by sending Immutable objects to the other party and then arranging for collection.

***Level 3***

Uses indirect pointers 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 which exposes an EventWaitHandle. .