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 manged operating system is an operating system that runs and is written in managed code ie type safe and garbage collected. The code is always compiled to native instruction and the system does not use a Virtual Machine or Sandbox , 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 trusted 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

Process Processes consist of an object space. The object space itself is immutable. A process may contain sub process which the parent can call directly without using IPC. [Can a sub process be shared by 2 parents].

Thread : A process contains a single light weight thread

Shared Memory: A process may contain links to a number of Shared Memory Regions

IPC Pipe: A process has a link to a number of IPC pipes which it uses to communicate to other processes.

AllocatorPool: A Process uses an allocator Pool which ensures objects are created on its heap. Note multiple processes can share an Allocator Pool.

Collector: Each Allocator pool has a collector for Garbage Collection

Object Capability: Security management is done via Capabilities. all such objects including the above may be persisted and are managed by the service.

Keyring: A secure persistent store for storing capability references.

Application: An exe when launched becomes a process it has extensive Meta data describing requirements as well as user and machine specific configuration , rights and data.

Service: A restartable application that can be launched at boot and restarted if failed.

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

Memory Region: A region of memory allocated to Allocator Pools in 1 Meg increments.

Please note there is no Address Space everything runs in a single unprotected address space.

# Micro KERNEL, Nano KERNEL?

What is the Kernel in a micro kernel style Asynch system? Some people define it as the messaging and task switch primitives which run at ring 0.

In our case it doesn’t even apply as these are just code modules. There is trusted code and un-trusted but this is not a good definition eg the GC is trusted but is not the Kernel as it sits in the process.

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

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

Yet none of the issues apply to Cosmos-B.

Hence Cosmos-B OS is kernelless and best described as Modular.

# Overall Design

As there is no kernel per se the system is best described by the TCB. The TCB is built by the loader of the OS **and any references to trusted objects these initial modules give out**. Note TCB components reside in each application object space eg the user Process has a reference to an Allocator that is un-trusted. There is also no user/kernel layer and lastly every process/application 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 services puts in the pipe , starts the Garbadge Collector . 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 to ensure they do no IO , privileged instructions ( eg CLI/STI) or use pointers this is checked during compilation.

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

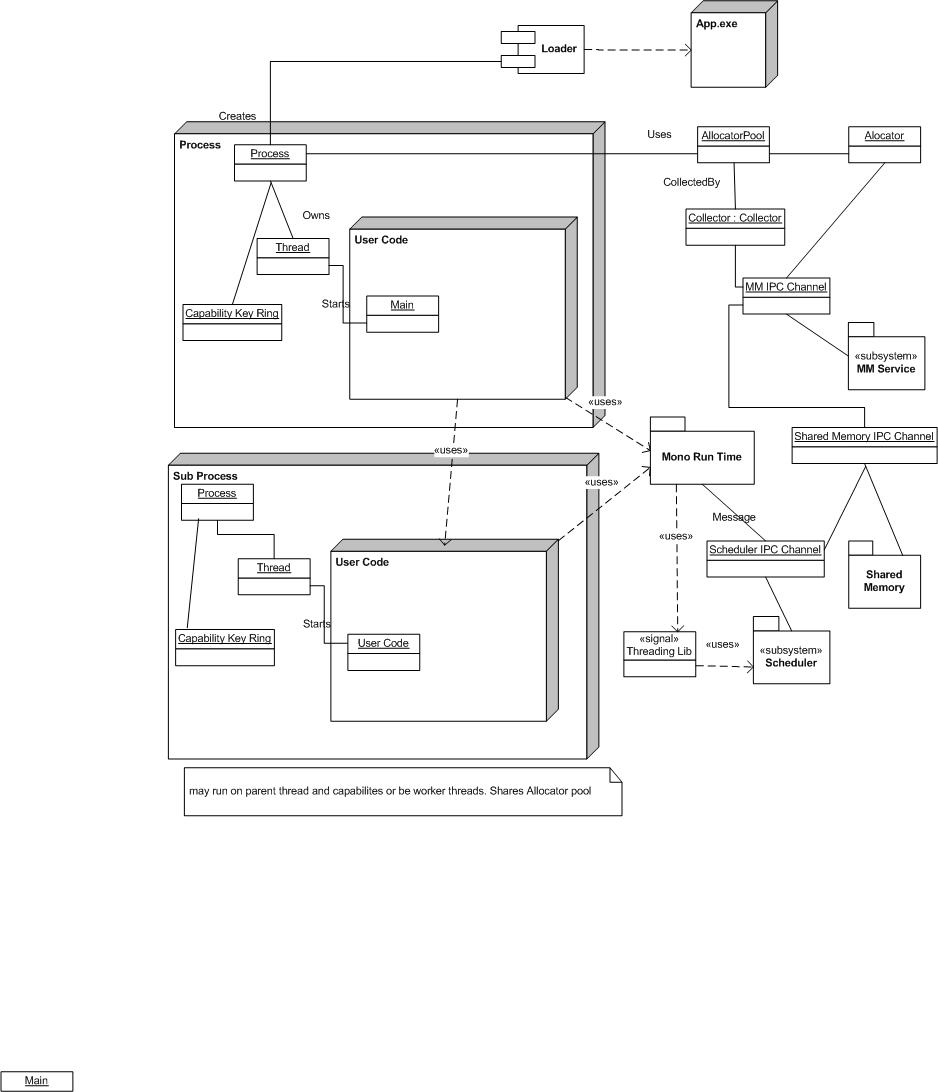
Do IO Instructions ( PIO and DMA) – Machine specific HAL.

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

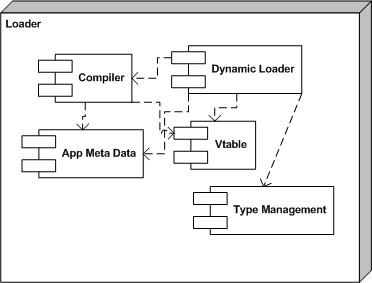
Do privileged Hardware instructions far jump: Scheduler

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.

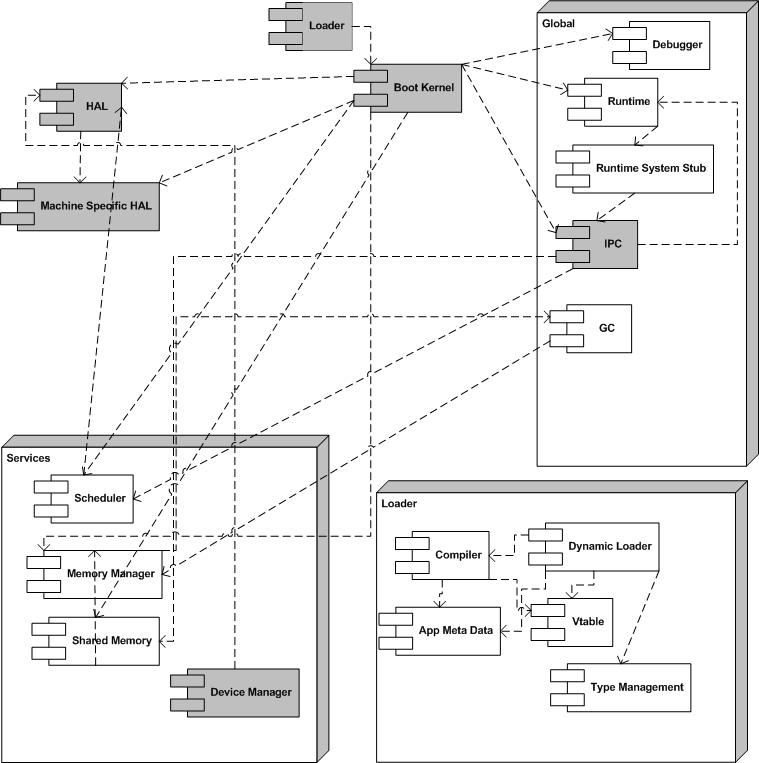
1. Application



1. Application Load



1. Rough module dependencies



Note the Memory manager uses the Garbage Collector this is possible provided spare memory is always available ( eg provided when we allocate a page its collector does not need to request a memory region) This scheme allows a very flexible code model where the Memory manager can even reside on a different machine and potentially be restarted .

# 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 server 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.

Hence we have chosen an Asynchronous IPC based on the assumption that threads in a Managed OS will be very light weight ( this is a critical requirement in the design) , any task switches are very fast and that a caller can invoke a service that has been waiting for a message. The receiver has the option of messages being queued or it can be immediately invoked if it is waiting for them. The ultimate goal is to make IPC as infrequent as possible.

The key advantages

* IPC which does not require a completion acknowledgement are not blocked eg a clock tick.
* 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 kernel calls via Messages our ABI will consist of just SendMessages and Yield though we may need to add locking primitives. This leaves us with only a single API for applications used to talk to the kernel and other applications.

Please note there will be a few critical cases where we will use Synch RPC style IPC bit 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!).

# 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. It is worth noting that Cosmos-B will be significantly more memory efficient as traditional operating systems since it supports a single very comprehensive set of libraries shared by all applications. The lack of virtual memory will also have an impact similar to the later DOS application in terms of applications being more memory efficient.. With these things in mind we believe 2 Gig system would run the same applications as a 4Gig Windows systems.

Multiple address spaces are supported for things like NUMA or managing subsidiary CPUs eg a Video Card.

***NUMA***

Numa style architectures can be supported by providing multiple unprotected address spaces the dispatcher can ensure the threads are scheduled according to the closest memory and even application that communicate frequently will be placed nearby.

***Shared memory***

There are 3 levels of shared memory.

***Level 1***

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 between process trees.

***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. To use this shared memory a pointer is simply passed to a process. 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 ) . It relies on both parties doing the right thing with the lock. It is fast and should only be used when sender and receiver are controlled ( eg with IPC it controls the sender and receiver). A control block exposes an EventWaitHandle.

***Level 3***

As level 2 but more carefully managed , the pointer 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.

.