CosmoS

Cosmos-B

STPs Model

*Cosmos-B is a managed code operation system designed to run on the Cosmos kit. This document describes the STPs model.*

STP ( SIngle THREADED PROCESS) MODEL

Ben KloosterMan

# 1.STPs Structure

Everything in CosmosB that executes is a Single Threaded Process ( STP) , STPs may be device drivers , services , applications or even shaders. ***At its simplest a Cosmos-B STPs is an Object Space with a single thread.*** This Object Space managed by a Garbage Collectormay but doesn’t have to be shared with other STP’s.

It has long been recognized that individual services provide increased interoperability, reuse and reliability ( COM, SOA) but the IPC call overhead has meant these services must be of a minimum size or linked to other services. As we do not have this cost it means we can move to a more service oriented model within the machine or even within the application.

STPs are created and controlled by the Scheduler. There are no public members except for GetInfo the constructor is internal, thus it is un-forgeable and can be used to identify a STPs hence a STPs reference is a capability.

STPs can expose their objects either via shared memory or sharing an allocator pool and sending a reference to the other party.

# 2. Goal

The overriding goal is to guide developers into a new structure which will realize greater benefits yet still allow backwards compatibility.

# 2. Benefits OF STRUCTURE

By encouraging single threads acting over a small amount of objects the above structure promotes

* Service orientation and reuse
* Greatly increased parallelism
* Increased performance for repetitive tasks ( cache use)
* Encourages more efficient use of resources eg a heavy floating point STPs could run on a 3D video card.
* Security since STPs (threads) have access to the minimum data needed instead of the whole address space.
* Reliability since parts of applications may be restarted. Faulty or unreliable parts are also more quickly isolated and may be re written.
* Supports unalterable STPs. Yet applications may be extended such extensions ( with a little care should have little effect on the parent and can either be run in the STPs Tree or as a separate STPs.

The costs for this is a slightly performance penalty for the IPC costs and a program structure consisting of separate simple parts but with more complex interaction between these parts. Debugging whole applications is also more difficult.

Note the system still supports the traditional; model via a single SIA with multiple STPs over the same address space.

# 3. HOW APPLICATIONS ARE MODELLED

Applications are modeled as STPs Trees. The main() entry is the root and it spawns sub STPs each with its own thread. Worker threads are created by creating a number of STPs each using a reference to the work required note its best that such a thread only has a reference to data types it can validly modify. Note sub components like the UI and worker threads are isolated in their own object spaces. These sub components each have their own thread.

# 4. APPLICATION LOAD

Applications are loaded by 2 methods

1. The system load creates some privileged applications such as the Memory Manager.
2. All other applications are loaded via the Dynamic Loader ( from a variety of sources)

The Dynamic loader does the following

* It Validates the callers execute capability on the program.
* It ensures the executable is valid and compiled.
* It ensures all requirements listed in the MetaData are available this may require loading some assemblies.
* All required assemblies are loaded (and it’s checked that the caller has the capability to them). If not the load will fail. This is checked for each assembly
  + Calls in the map file are patched with the actual loaded memory address. This information is stored by the code manager.
  + Calls to existing assemblies from the new Assemblies are patched. Note this will check whether the user is allowed to run the code.
  + Each such call is checked whether the user has access to the call.
* A GC is created
* A thread with stack is created and set to execute main
* The thread is added to the dispatcher.
* The loader completes.

# 5. Code Management

All code is loaded by the Code Manager into memory, the framework and kernel code is loaded into a contiguous region of memory. Assemblies are only unloaded if no program uses them and the machine is under memory pressure Framework assemblies are never unloaded. Full signing and versioning is supported.

Code security is provided in a number of ways

* A user needs a capability to run a piece of code ( and all of its dependencies) this is normally done via the Dynamic Loader having a reference from the user and the meta data for the app requesting it. Some code is loaded but not exposed to the dynamic loader for linking.
* The IL language provides type security for private classes. These will not be patched and all references are illegal.
* System or Unsafe libraries are NEVER exposed. The only way these get loaded is by the Kernel loader and the only way to get access is for an already trusted STPs to pass a reference.

# 6. Data/ HEAP Allocation

When STPs are loaded a GC is loaded the type of allocator/collector is determined by compilation options ( ie ref counting) , memory pressure and meta data ( eg opt for memory or performance) . The Collector may change as memory pressure in a machine builds.

Initial memory allocated is also determined by the meta data. (See Memory for more details)

Care must be taken with the IPC design so that a reference passed to another STPs is not collected and hence cause security leaks/memory violations.

# Software API

The .NET API does not really map onto our system model and most of the classes should be vied as User Information . The underlying classes are things for STPs and Thread are STPsInfo and ThreadInfo.

A new Capability based API will be created though most of this will not be exposed to the user. Message based requests are sent to control STPs.

# 8.Security

The system uses capabilities for security. The STPs capability is what identifies the STPs we do not use STPs id anywhere. This is a non forgeable reference. See Cosmos 8.

# 9.IPC

See the Cosmos3 .

# 10. Non HOmogenous CPUs

The system will support non Homogenous CPUs at some point . STPs will run on a number of cores though we will use a single core to manage this.

# 11. Threads

STPs are threads and it is intended that a the system will run large amounts of them. The only resource this will take is memory due to stack space to address this we will have minimum and maximum stack values specified in MetaData. There are 2 options

1. Expand and Contract stack via memcopy when an OutOfStackMessage results or a memory pressure event.
2. Use some sort of stack based on smaller blocks which are added and subtracted.

# 12. STPS , DEVICE DRIVERS AND SERVICES

All device drivers, services are user STPs. The system will in fact convert most normal applications like word and browsers to long running persistent services.

# 13. RELIABILITY

By breaking applications down into fine grain services we allow partial failure without the dreaded crash to desktop or hang. This does not even apply to Addins failing but even for example a printing process failing will not cause the crash to desktop but just prevent printing and these sub components can even be restarted without affecting the whole.

# 14 . Impact ON GC TIMES

Since IPC is very cheap we can in most cases use a different GC for each STP , whether an STP shares its object space and hence GC with another STP can be done in the config file. By having lots of smaller STP we reduce GC pauses at the cost of slightly higher IPC overhead ( but still cheap).