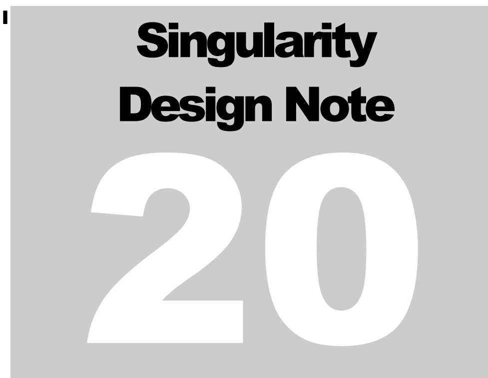


Application Binary Interface

Design and implementation of the interface between processes and the Singularity kernel

This design note captures the details of the interface between processes and the kernel in Singularity. It describes both functional and structural interfaces.

1. Design Goals

Singularity attempts to define a very strong, versioned binary interface between process code and the Singularity kernel. The construction and factory of the Singularity ABI has four important design goals:

- Enable separate GC domains for each process and for the kernel.
- Enable separate runtimes for each process and for the kernel.
- Enable independent versioning of processes from the kernel.
- Enable strong isolation of processes.

Recognizing that no class library has ever successfully been versioned independently from it applications[1], the Singularity ABI is a functional interface. In C#, Singularity ABI is exposed as a collection of static methods, never as instance methods.

2. ABI Description

The Singularity ABI includes functions to create and manipulate threads within the calling process, create and use thread synchronization objects, activate child processes, create and manipulate message content, send and receive messages via channels, securely determining the identity of another process accessed through a channel, allocate and free pages used for GC memory, grow and shrink thread stack segments, access process parameters, and terminate the calling process when it completes.

---

[1] Consider the change required to move a program from MFC 4.2 to MFC 7.0. Even though each version of MFC is packaged as a DLL, a program must be ported from one version of MFC to the next. While any library upgrade can require a port, class libraries are especially problematic because object-oriented programming practices, such as inheritance, introduce extremely intricate implementation dependencies between a program and its class library.

Copyright © Microsoft Corporation. All Rights Reserved.

Two design points of the ABI are extremely important in establishing trust and security in Singularity. **First, none of the ABI functions modify the state of other processes.** The impact of ABI functions is limited to the calling process and the kernels state related to the calling process[2]. This scoping of ABI functions isolates the impact of a process to itself, its interaction with the kernel, and the set of processes to which it communicates through channels. A parent process constrains the reach of a child process by limiting the channels accessible to the child process. For example, in hosting scenarios, a parent process can prevent an untrusted child process (an extension) from communicating with all other processes.

Second, the ABI interface between a process and the kernel cannot be intercepted or altered without explicit approval of the process' author. This system guarantee keeps a parent process from inadvertently or intentionally altering the semantics of the ABI. Thereby the OS guarantees a dependable execution environment for the child process. This guarantee also prevents an untrusted parent process from snooping on private communication between the child process and the kernel.

The Singularity kernel provides a base for establishing trust and dependability. By limiting the scope of the ABI to the state of the calling process and guaranteeing integrity of the ABI, Singularity establishes a trusted isolation boundary between parent and child process. The Singularity ABI enforces a model that allows the parent process to limit a child's access to all additional software services (accessed through channels). At the same time, the Singularity ABI gives a child process the ability to identify the providers of all software services (accessed through channels) and limit its use of services based on its trust policies.

2.1. Services

2.1.1. DebugService

The `DebugService` provides user code with access to the kernel debugger. Eventually the implementation of this ABI will be replaced with access to any debugger currently attached to this process.

```
Break
PollAndBreak
Print
WalkStack
```

2.1.2. DeviceService

The `DeviceService` provides the trusted runtime with access to hardware resources. For the most part, the trusted runtime code which wraps DMA memory, and I/O ports, and interrupt lines in access classes which are then provide to user code.

[2] While a process can communicate with another via a channel, it cannot directly change the state of the target process. A message send ABI moves a message from sender's state into the channel. The message enters the receiver's state only when it explicitly invokes a message receive ABI.

Copyright © Microsoft Corporation. All Rights Reserved.

```
GetIoDmaRange
GetIoIrqRange
GetIoMemoryRange
GetIoPortRange
GetIoRangeCount
GetIrqCount
GetPciConfig
GetPnpSignature
```

2.1.3. EndpointCore

The `EndpointCore` provides trusted runtime code with access to the kernel messaging primitives.

```
Allocate
Connect
Free
TransferBlockOwnership
TransferContentOwnership
```

2.1.4. PageTableService

The `PageTableService` provides trusted runtime garbage collectors (GCs) with access to raw memory pages.

```
Allocate
AllocateBelow
AllocateExtend
Free
GetPageCount
GetPageTable
GetProcessTag
Query
```

2.1.5. ProcessHandle

The `ProcessHandle` provides user code with access to child processes.

```
Create
Dispose
GetExitCode
GetProcessId
Join
Start
Terminate
```

2.1.6. ProcessService

The `ProcessService` provides user code with access to process activation information.

Copyright © Microsoft Corporation. All Rights Reserved.

```
GetCurrentProcessId
GetCycleCount
GetCyclesPerSecond
GetNamespaceEndpoint
GetStartupArg
GetStartupArgCount
GetStartupEndpoint
GetStartupEndpointCount
GetTracingHeaders
GetUpTime
Terminate
```

2.1.7. CommunicationHeapService

The `CommunicationHeapService` provides the trusted runtime with access to the shared communication heap.

```
Allocate
Free
GetData
GetSize
GetType
SetOwnerProcessId
SetType
Share
Split
```

2.1.8. StackService

The `StackService` provides trusted runtime code with access to linked stack segments.

```
GetUnlinkStackRange
LinkStack
```

2.2. Threads

Thread operations affect the state of the calling process only. They cannot be used, for example, to create a thread in a target process.

2.2.1. AutoResetEventHandle

The `AutoResetEventHandle` provides user code with a thread synchronization object for notifying a waiting thread that an event has occurred.

```
Create
Dispose
Reset
Set
```

2.2.2. InterruptHandle

The `InterruptHandle` provides user code with a thread synchronization object that is signaled by incoming interrupts.

Copyright © Microsoft Corporation. All Rights Reserved.

```
Ack
Create
Dispose
Wait
```

2.2.3. ManualResetEventHandle

The `ManualResetEventHandle` provides user code with a thread synchronization object to notify one or more waiting threads that an event has occurred.

```
Create
Dispose
Reset
Set
```

2.2.4. MutexHandle

The `MutexHandle` provides user code with a thread synchronization object for serializing access to a shared resource.

```
Create
Dispose
Release
```

2.2.5. SyncHandle

The `SyncHandle` provides user code with a mechanism for blocking on one or more thread synchronization objects.

```
WaitAny
WaitOne
```

2.2.6. ThreadHandle

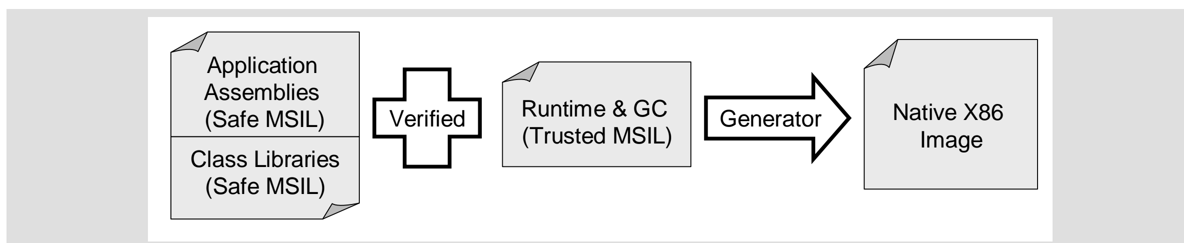
The `ThreadHandle` provides user code with access to creating, joining, and scheduling threads in the same process.

```
Create
CurrentThread
Dispose
GetExecutionTime
GetThreadLocalValue
GetThreadState
Join
SetThreadLocalValue
Sleep
SpinWait
Start
Yield
```

3. Process Construction

A Singularity process is constructed from three types of assemblies: the application assemblies, class library assemblies, and the Runtime assemblies. The application and class library assembles must be expressed as type safe MSIL. The Runtime assemblies are part of the Singularity trusted computing base (TCB), and contain unsafe, but trusted, MSIL.

Copyright © Microsoft Corporation. All Rights Reserved.



After verification, the process image generator combines application and class library assemblies with Runtime assemblies. The generated image contains native x86 instructions and accompanying metadata used by the runtime, such as VTables, GC structures, and exception structures.

For the spring 2005 release, Singularity native images are generated as part of the build process using Bartok and the Microsoft linker (`link.exe`). In future versions of Singularity, Bartok will run on the target computer and `link.exe` will be replaced with a managed code linker.

4. Implementing an ABI

To implement an ABI, the developer must edit four source files: an ABI version specific C# interface file[3], an ABI version specific trusted process code file[4], an ABI version specific kernel code file[5], and the kernel implementation file[6]. A fifth file, an ABI version specific PE module definition file[7] is created automatically as part of the build process.

4.1. The ABI C# Interface

The ABI C# interface file declares the type information relative to the ABI. An Import Link Library (`.ill`) assembly is created from the C# interface file using `csic.exe`. User programs compile against the import assembly to prevent them from creating unexpected implementation dependencies and to prevent them from accessing the implementation details of the ABI.

abi.v1.csi : Interfaces/Singularity.V1/DebugService.csi:

```
namespace Microsoft.Singularity.V1.Services
{
    public struct DebugService
    {
        public static void Print(ulong value);
    }
}
```

4.2. The ABI Trusted Process Code

The ABI trusted process code file contains any wrapper code or implementation needed for the ABI that must run in the process. In most cases, the ABI trusted process code declares that the ABI is an external function which must be resolved a link time. The ABI trusted process code typically also declares that the ABI marks the boundary between the kernel and process GC domains and that the

[3] Referred to hereafter in examples as `abi.v1.csi`.
[4] Referred to hereafter in examples as `abi.v1.cs`.
[5] Referred to hereafter in examples as `abicod.v1.cs`.
[6] Referred to hereafter in examples as `kernel.cs`.
[7] Referred to hereafter in examples as `abi.v1.def`.

Copyright © Microsoft Corporation. All Rights Reserved.

ABI may be accessed by trusted runtime code. The ABI trusted process code runs within the process GC domain.

abi.v1.cs : Libraries/Singularity.V1/DebugService.cs:

```
namespace Microsoft.Singularity.V1.Services
{
    public struct DebugService
    {
        [AccessedByRuntime]
        [OutsideGCDomain]
        [MethodImplAttribute(MethodImplOptions.InternalCall)]
        public static extern void Print(ulong value);
    }
}
```

4.3. The ABI Kernel Code

The ABI kernel code file contains the code invoked in the kernel when the ABI is called. While the ABI kernel code may contain the entire implementation of the ABI, it is more often the case that the ABI kernel code wraps lower level kernel implementation code and handles impedance mismatch between later ABI versions on new version of the kernel. The ABI kernel code runs in the kernel GC domain and can access kernel data structures as needed.

abicod.v1.cs : Kernel/Singularity/V1/Services/DebugService.cs:

```
namespace Microsoft.Singularity.V1.Services
{
    public struct DebugService
    {
        [AccessedByRuntime]
        public static void Print(ulong value)
        {
            DebugStub.Print(Thread.GetCurrentProcess(), value);
        }
    }
}
```

4.4. The Kernel Implementation

The kernel implementation files contain the most recent implementation of kernel functionality exposed via ABIs. As new versions of the kernel and operating system are produced, the kernel implementation will evolve. To increase engineering flexibility, impedance mismatches between a specific version of an ABI and the kernel implementation of the underlying objects should be resolved in the ABI kernel code. Where possible, versioning details should not appear in the kernel implementation code.

kernel.cs : Kernel/Singularity/DebugStub.cs:

```
namespace Microsoft.Singularity
{
    public struct DebugStub
    {
        public static void Print(Process process, ulong value)
        {
            ...
        }
    }
}
```

Copyright © Microsoft Corporation. All Rights Reserved.

4.5. The ABI PE Module Definition

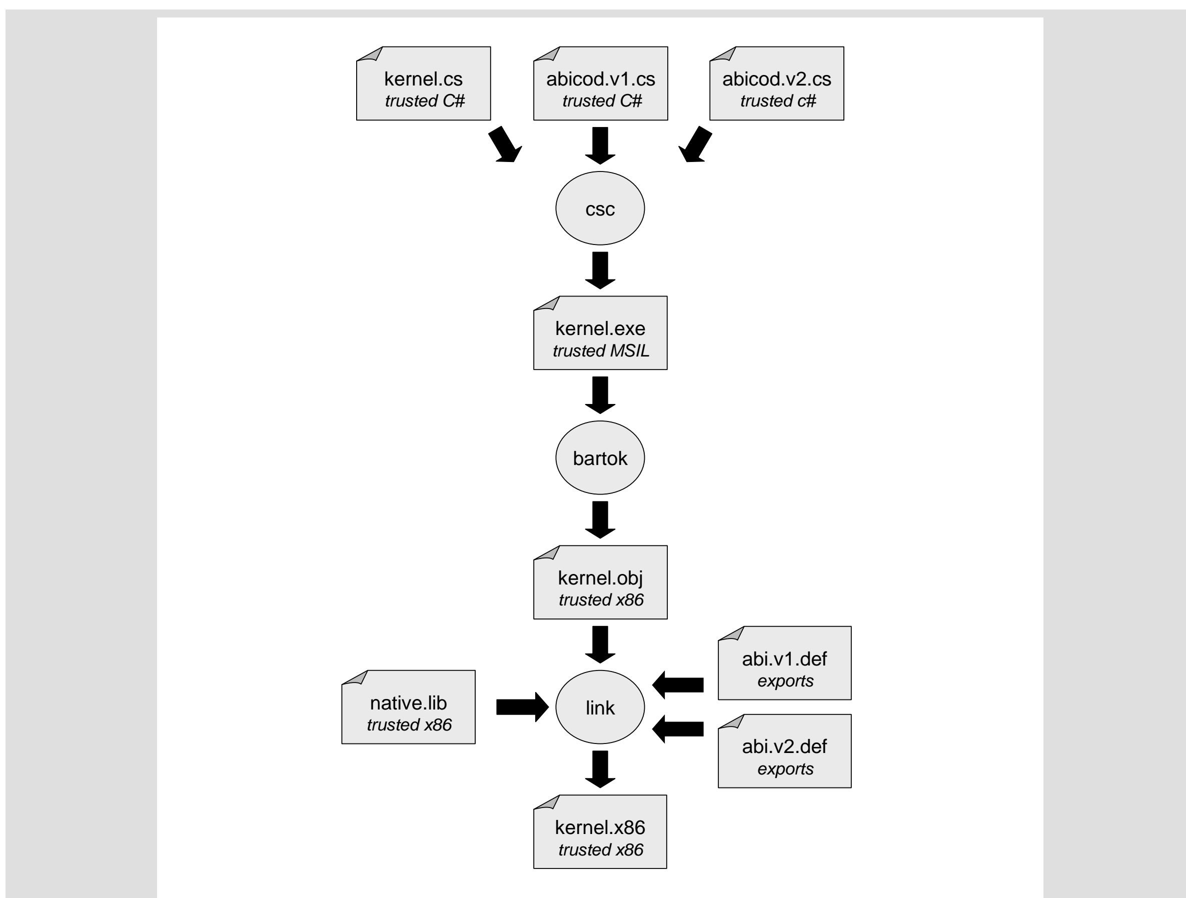
References from process code to kernel code are resolved at load time using the PE binary import/export mechanism. The kernel native binary exports entry points for all versions of ABIs currently supported. Process native binaries import entry points for the versions of ABIs against which it was built. The PE module definition files are currently generated automatically as part of the kernel build.

abi.v1.def : Kernel/obj/debug/singularity.V1.def:

```
EXPORTS
    ?g_Print@Struct_Microsoft_Singularity_V1_Services_DebugService@@SIXI@Z
```

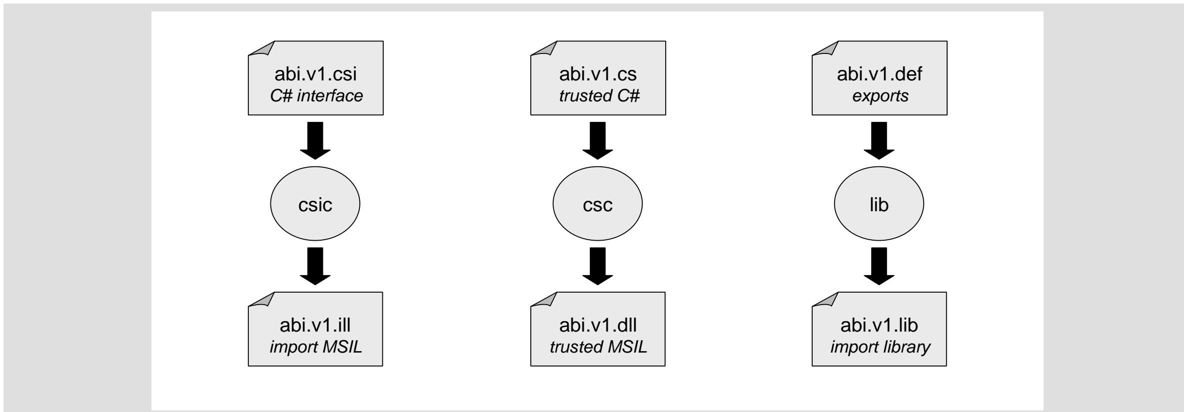
5. Building the ABI

5.1. Constructing the OS Kernel

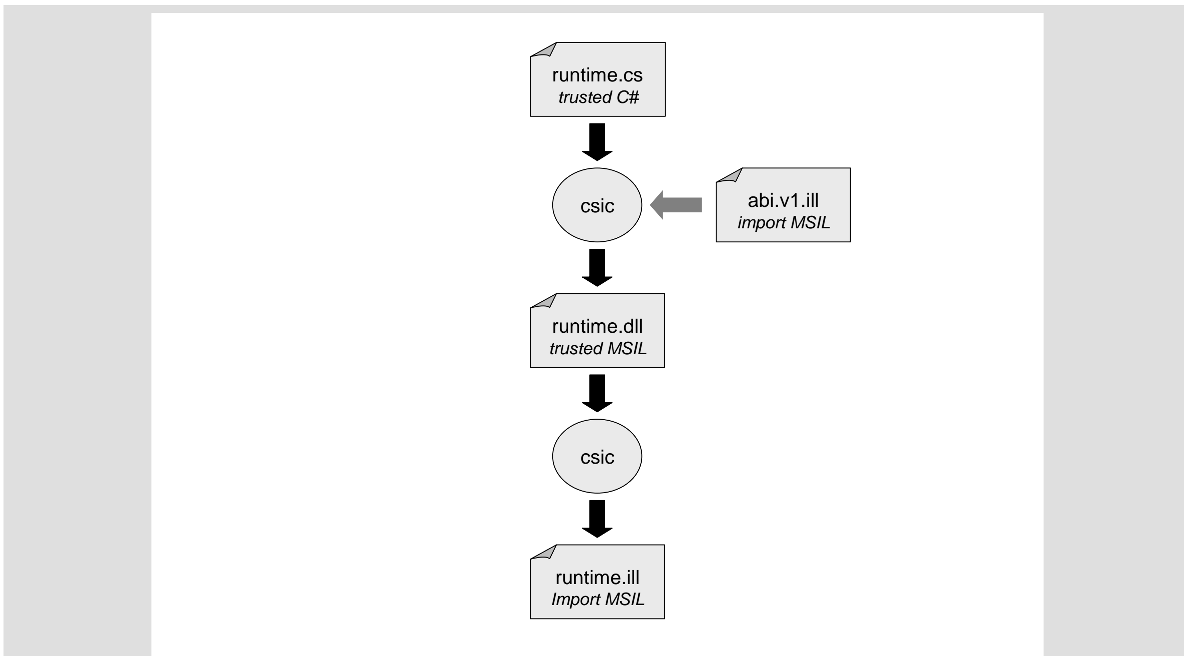


Copyright © Microsoft Corporation. All Rights Reserved.

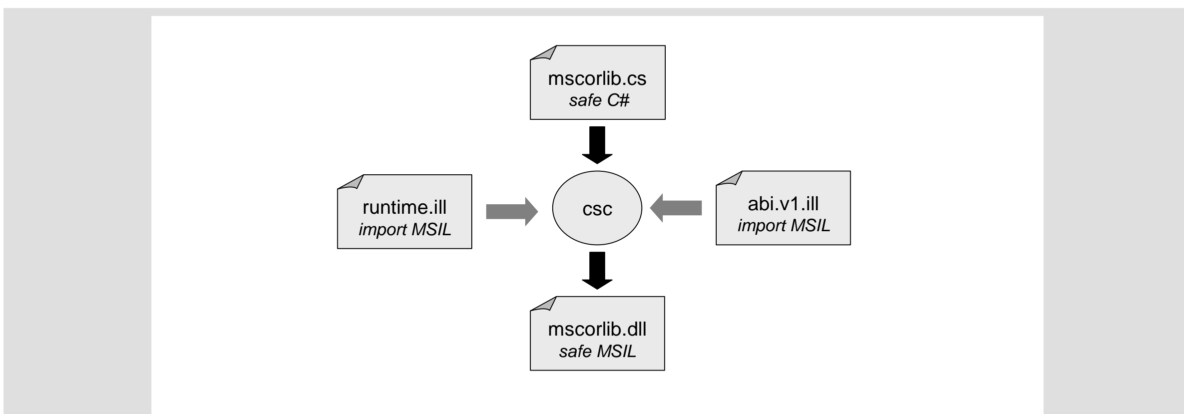
5.2. Constructing Assemblies and Libraries for an ABI



5.3. Constructing Assemblies for a Runtime

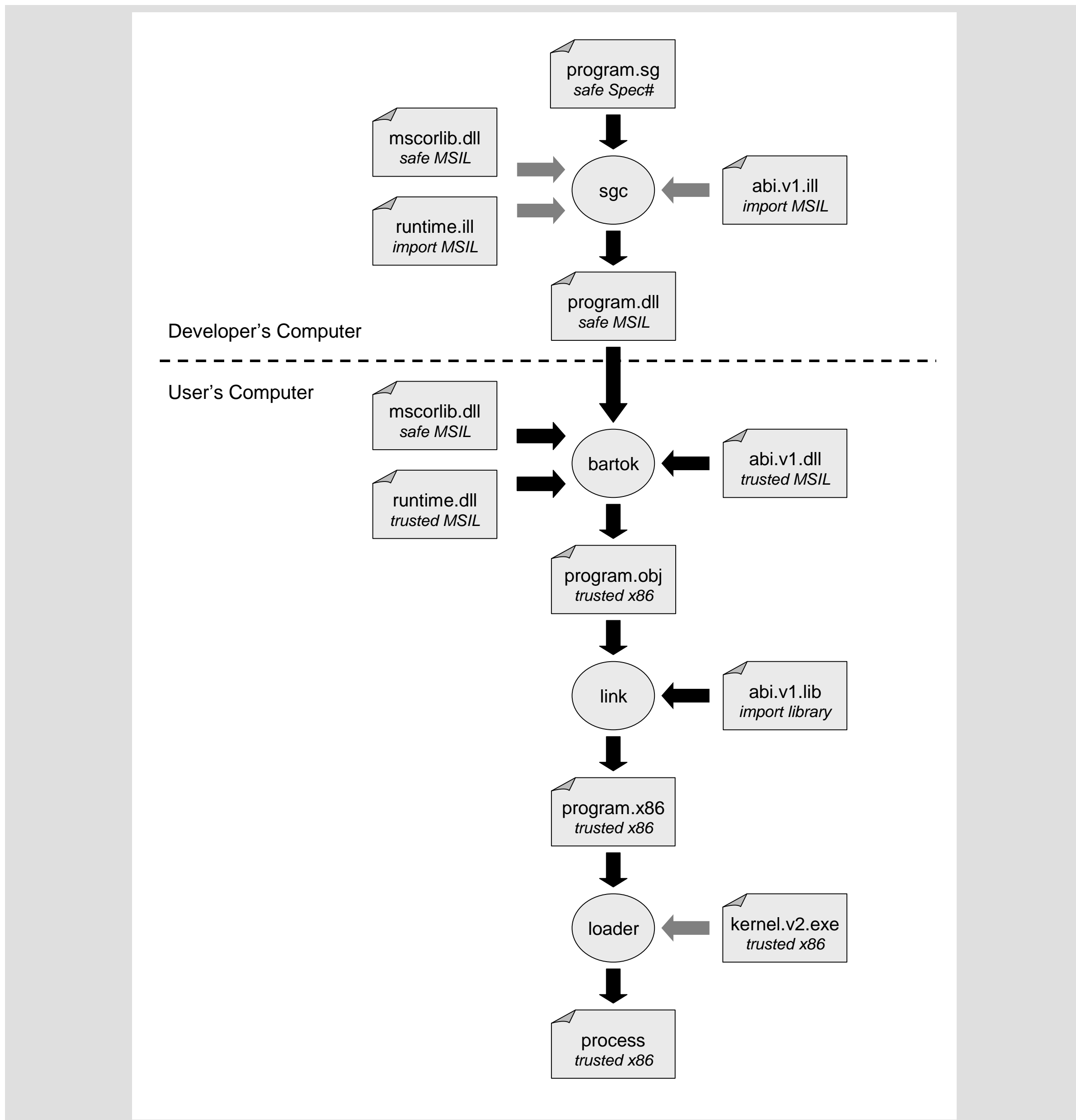


5.4. Constructing a Class Library Assembly



Copyright © Microsoft Corporation. All Rights Reserved.

5.5. Constructing a Process



6. Threads

All Singularity threads are kernel threads. Threads start life in the kernel. The thread object is encapsulated in the `Kernel/System.Threading.Thread` class. When not scheduled, the threads context is saved to the `Kernel/System.Threading.X86ThreadContext` structure.

6.1. Processor and Thread Context

Singularity is intended to support multiple processors. Each physical processor is represented by a processor object of class `Microsoft.Singularity.Processor`. In addition, each processor has an immovable processor context structure of type `Microsoft.Singularity.X86.ProcessorContext` pointed to by the processor's `fs` register.

Copyright © Microsoft Corporation. All Rights Reserved.

Native code uses offsets on the `fs` register to the `ProcessorContext` structure to quickly access processor local storage. As related to this design note the format of `ProcessorContext` structure is as follows:

```
namespace Microsoft.Singularity.X86
{
    [StructLayout(LayoutKind.Sequential)]
    internal struct ProcessorContext
    {
        ...
        [AccessedByRuntime] internal unsafe ThreadContext *threadContext;
        ...
        [AccessedByRuntime] internal UIntPtr schedulingStackBegin;
        [AccessedByRuntime] internal UIntPtr schedulingStackLimit;
        [AccessedByRuntime] internal UIntPtr interruptStackBegin;
        [AccessedByRuntime] internal UIntPtr interruptStackLimit;
        [AccessedByRuntime] internal UIntPtr exceptionStackBegin;
        [AccessedByRuntime] internal UIntPtr exceptionStackLimit;
        ...
    }
}
```

Among other items, the `ProcessorContext` structure points to the `ThreadContext` structure of the thread currently scheduled on that processor. The kernel maintains one `ThreadContext` structure for each thread, which holds the thread's registers when it is not scheduled.

As related to this design note, the format of the `ThreadContext` structure is as follows:

```
namespace Microsoft.Singularity.X86
{
    using System.GCs.CallStack;

    [StructLayout(LayoutKind.Sequential)]
    internal struct ThreadContext
    {
        ...
        [AccessedByRuntime] internal UIntPtr stackBegin;
        [AccessedByRuntime] internal UIntPtr stackLimit;
        ...
#if SINGULARITY_KERNEL
        [AccessedByRuntime] private unsafe Thread *_thread;
        [AccessedByRuntime] internal UIntPtr processThread;
        [AccessedByRuntime] internal unsafe TransitionRecord * asmStackMarkers;
        [AccessedByRuntime] internal unsafe TransitionRecord * processMarkers;
#elif SINGULARITY_PROCESS
        [AccessedByRuntime] internal UIntPtr kernelThread;
        [AccessedByRuntime] private unsafe Thread *_thread;
        [AccessedByRuntime] internal unsafe TransitionRecord * kernelMarkers;
        [AccessedByRuntime] internal unsafe TransitionRecord * asmStackMarkers;
#endif
        ...
    }
}
```

The `_thread` field of the `ThreadContext` structure contains a GC managed reference to the `System.Threading.Thread` object for this thread, not a pointer to a managed reference as is suggested by the syntax. Unfortunately, C# has no syntax to allow a managed reference to be embedded in a structure. Managed code should use the `thread` property to access this value. Native code can use the field

Copyright © Microsoft Corporation. All Rights Reserved.

directly. The `_thread` field is actually aliased to two separate objects, one for kernel code and one for process code. The former is managed by the kernel's GC, the latter is managed by the process' GC.

To load the `stackLimit` for the current thread, Bartok emits the following code:

```
mov eax,fs:[Struct_Microsoft_Singularity_X86_ProcessorContext._threadContext]
mov eax,[eax].Struct_Microsoft_Singularity_X86_ThreadContext._stackLimit
```

The `ProcessorContext` and `ThreadContext` structures are allocated either from the kernel page manager or from the system shared heap.

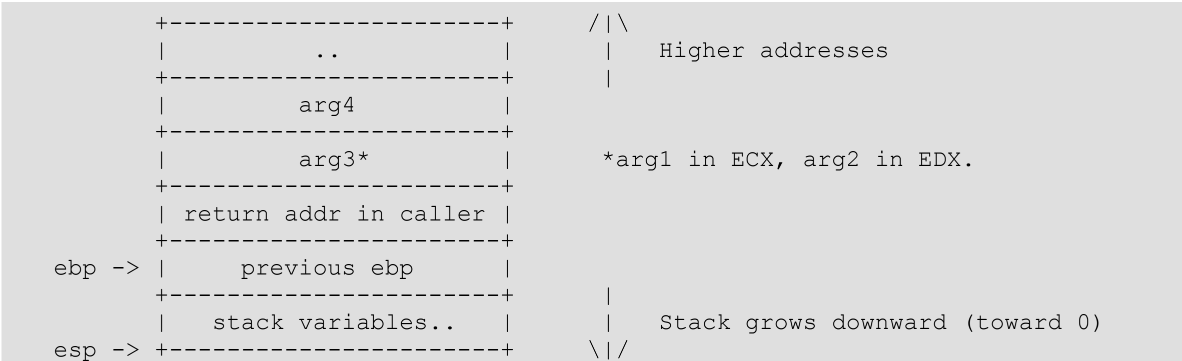
6.2. Calling Convention

On the X86, Singularity adopts the `__fastcall` calling convention. Key features of `__fastcall` are:

- **Argument passing:** The first two 32-bit or smaller arguments are passed in `ECX` and `EDX` registers; all other arguments are passed right to left.
- **Return values:** 32-bit or smaller values are returned in `EAX`, 64-bit values are returned in `EDX:EAX`, larger values are returned through a pointer passed as an argument.
- **Stack maintenance:** Called method pops the arguments from the stack.
- **Callee-saved registers:** `EBX`, `EBP`, `ESP`, `EDI`, `ESI`. `EAX` used for return values and free at callee entry. Any in-use floating pointer registers must be saved by the caller.
- **Stack-chain:** `EBP` always points to the top of the current method's stack frame. The prologue for every method issues the following sequence of instructions to maintain the `EBP` chain:

```
push  ebp
mov   ebp, esp
```

- **Stack frame:** A method's stack frame has the following format:



```
          +-----------------------+     /|\
          |          ..           |      |   Higher addresses
          +-----------------------+      |
          |         arg4          |
          +-----------------------+
          |         arg3*         |      *arg1 in ECX, arg2 in EDX.
          +-----------------------+
          | return addr in caller |
          +-----------------------+
   ebp -> |     previous ebp      |
          +-----------------------+      |
          |   stack variables..   |      |   Stack grows downward (toward 0)
   esp -> +-----------------------+     \|/
```

6.3. Fixed Stacks

Singularity threads have access to two types of stack. Fixed stacks are fixed-size stacks allocated on a per processor basis. Linked stacks are allocated on a per-thread basis and consist of one or more independently allocated segments. Linked stack segments are allocated on demand and released as soon as they are no longer needed. Linked stacks allow Singularity to create large numbers of threads with low memory overhead.

Copyright © Microsoft Corporation. All Rights Reserved.

By convention, fixed stacks may be used only when interrupts are disabled in order to avoid potential conflicts between multiple thread accessing the same fixed stack. On the x86, even relatively small sequences of code can use fixed stacks as needed because a pair of `cli` and `sti` instructions can execute in as few as 10 cycles. In addition, because of the system's abstract instruction set, Bartok can inline these instruction when needed into safe code, whether the code is part of the trusted computing base or not.

Singularity allocates three fixed stacks for each schedulable processor:

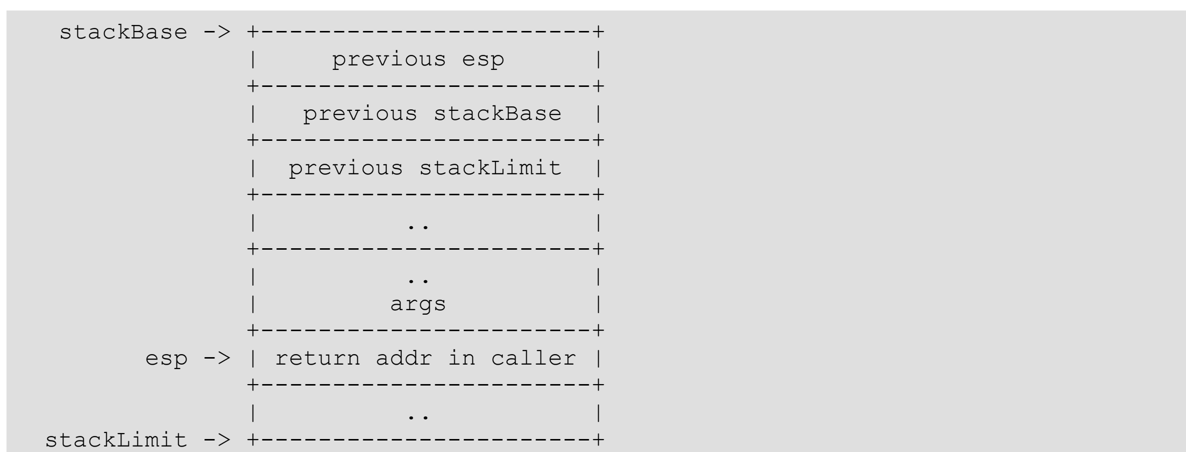
- The **Scheduling Stack** is used by the scheduler proper or scheduler support routines, such as code to determine if a message delivered through a channel will unblock the destination thread.
- The **Interrupt Stack** is used by the hardware interrupt handler to dispatch interrupts from I/O devices and other processors.
- The **Exception Stack** is used by the exception handler to dispatch processor exceptions such as debug breaks, overflow exceptions, etc.

Only trusted code may directly change from between fixed and linked stacks or from one fixed stack to another. However, Bartok may inline trusted code into other code when generating the native image for a process.

6.4. Linked Stacks

Singularity allocates a small stack initial linked stack segment for each thread then allocates and frees additional stacks segments on demand. The image generator inserts special prologues into select methods that verify at entry that sufficient stack is available. If sufficient stack space is unavailable, the stack maintenance prologue asks the kernel to allocate an additional stack segment. Each stack segment is denoted by three pointers: `stackBase`, the highest address of the segment; `stackLimit`, the lowest address of the segment; and `ESP`, the stack pointer. The `stackBase` and `stackLimit` are stored in the `ThreadContext` structure.

When a method is invoked, the layout of the current working stack segment is as follows:



```
  stackBase -> +----------------------+
               |     previous esp     |
               +----------------------+
               |   previous stackBase |
               +----------------------+
               |  previous stackLimit |
               +----------------------+
               |          ..          |
               +----------------------+
               |          ..          |
               |         args         |
               +----------------------+
        esp -> | return addr in caller |
               +----------------------+
               |          ..          |
 stackLimit -> +----------------------+
```

The method prologue emitted by Bartok for the stack check compares the current stack limit with the maximum anticipated stack limit. If the current stack limit is insufficient, the stack check invokes the appropriate `LinkStackN` method. The cost of the stack check is approximately 5 cycles on the

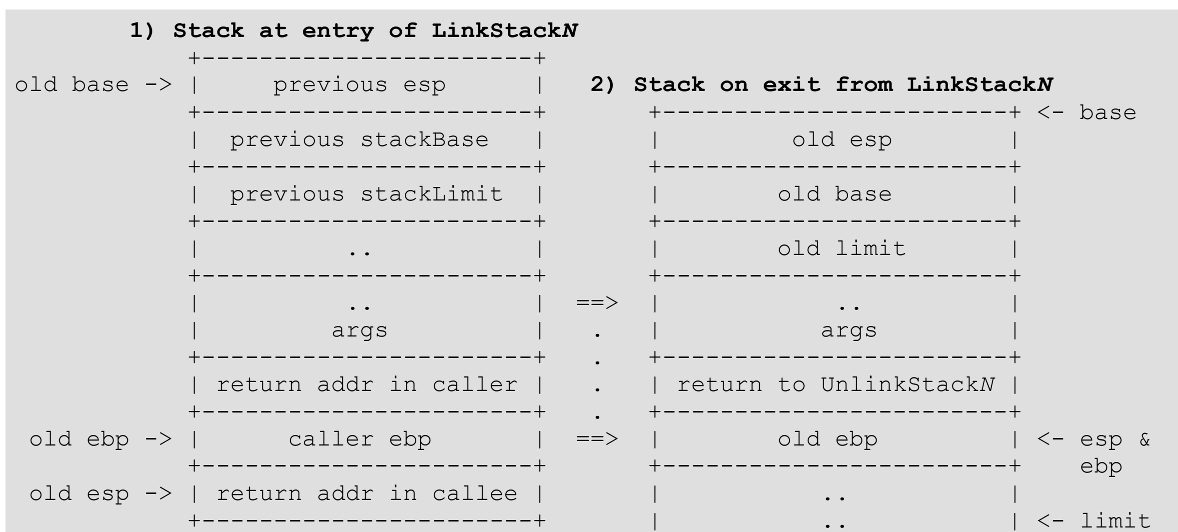
Copyright © Microsoft Corporation. All Rights Reserved.

AMD64. The `LinkStackN` methods are exported via the DLL export table from the kernel to process code. The value *N* in the name is the count of bytes of arguments on the stack that must be copied.

```
      push ebp                 ; create EBP chain.
      mov  ebp, esp            ; rest of callee code assumes EBP chain.

      mov  eax, fs:[ProcessorContext8._threadContext]
      mov  eax, [eax].ThreadContext._stackLimit
      add  eax, frame_size
      cmp  eax, esp
      jge  link                ; run link code if stack will overflow.
cont:
      ...                      ; function body
      pop  ebp                 ; undo EBP chain.
      ret  N                   ; callee pops arguments from stack.
.
link:
    ; Note: move to optimize branch prediction?
      mov  eax, stack_needed   ; load argument for LinkStackN
      call Class_Microsoft_Singularity_Memory_Stacks::LinkStackN
      jmp  cont                ; run function body.
```

In addition to allocating a new stack segment and creating a new stack frame on the segment, the `LinkStackN` method adjusts the return address in the new frame so that on exit from the method, control is transferred to one of the `UnlinkStackN` methods. The following figure shows the previous stack segment at entry and exit from `LinkStackN` and the new stack segment on exit from `LinkStackN`.



```
        1) Stack at entry of LinkStackN
             +-----------------------+
old base ->  |     previous esp      |   2) Stack on exit from LinkStackN
             +-----------------------+       +------------------------+ <- base
             |  previous stackBase   |       |        old esp         |
             +-----------------------+       +------------------------+
             |  previous stackLimit  |       |        old base        |
             +-----------------------+       +------------------------+
             |          ..           |       |       old limit        |
             +-----------------------+       +------------------------+
             |          ..           |  ==>  |           ..           |
             |         args          |   .   |          args          |
             +-----------------------+   .   +------------------------+
             | return addr in caller |   .   | return to UnlinkStackN |
             +-----------------------+   .   +------------------------+
 old ebp ->  |      caller ebp       |  ==>  |        old ebp         | <- esp &
             +-----------------------+       +------------------------+    ebp
 old esp ->  | return addr in callee |       |           ..           |
             +-----------------------+       |           ..           | <- limit
```

Bartok need not insert any code for stack unlinking as the number of arguments to be popped is coded into each `UnlinkStackN` method. The following figure shows the stack segments at the entry of `UnlinkStackN`:

[8] `Struct_Microsoft_Singularity_X86_ProcessorContext` and `Struct_Microsoft_Singularity_X86_ThreadContext` have been abbreviated here to make the text easier to read.

Copyright © Microsoft Corporation. All Rights Reserved.

```
old base -> +-----------------------+
            |     previous esp      |
            +-----------------------+       +------------------------+ <- base
            |  previous stackBase   |       |         old esp        |
            +-----------------------+       +------------------------+
            |  previous stackLimit  |       |        old base        |
            +-----------------------+       +------------------------+
            |          ..           |       |        old limit       | <- esp
            +-----------------------+       +------------------------+
            |          ..           |       |           ..           | * ebp =
            |         args          |       |           ..           |   old ebp
            +-----------------------+       |           ..           |
            | return addr in caller |       |           ..           |   ecx &
            +-----------------------+       |           ..           |   edx
     ebp -> |      caller ebp       |       |           ..           |   are free
            +-----------------------+       |           ..           |
 old esp -> | return addr in callee |       |           ..           |
            +-----------------------+       |           ..           | <- limit
```

The following figure shows the stack segment at exit from `UnlinkStackN`:

```
       2) Stack at exit from UnlinkStackN
 stackBase -> +-----------------------+
              |     previous esp      |
              +-----------------------+
              |  previous stackBase   |
              +-----------------------+
              |  previous stackLimit  |
              +-----------------------+
              |          ..           |
       ebp -> |          ..           |
              |          ..           |
       esp -> |          ..           |
              +-----------------------+    * eip = return addr in caller
```

6.5. Stack Regions

A given thread may pass back and forth from kernel to process code multiple times during its execution. Singularity must isolate stack frames created by kernel code from stack frames by process code in order to allow independent garbage collection and termination of process code. To isolate kernel and process stack frames, Singularity reuses the stack marking mechanism implemented in Bartok to isolated managed and unmanaged code on Windows.

Copyright © Microsoft Corporation. All Rights Reserved.

The transition record contains all of the callee saved registers so that the GC can fix up registers that may contain object references before execution leaves the current GC domain.

```
namespace System.GCs
{
    internal unsafe class CallStack
    {
        [StructLayout(LayoutKind.Sequential)]
        internal unsafe struct TransitionRecord
        {
            internal TransitionRecord *oldTransitionRecord;
            internal UIntPtr callAddr;
            internal UIntPtr stackBottom;
            internal UIntPtr EBX;
            internal UIntPtr EDI;
            internal UIntPtr ESI;
            internal UIntPtr EBP;
        }
    }
}
```

Kernel and process code each maintain a linked list of `System.GCs.CallStack.TransitionRecord` structures. Before transitioning to process code, the kernel creates a `TransitionRecord` on the stack and inserts it at the head of its `asmStackMarkers` linked list. Likewise before transitioning to kernel code, the process creates a `TransitionRecord` on the stack and inserts it at the head of its `asmStackMarkers` linked list.

When terminating a process the kernel walks the two lists of `TransitionRecord` structures and modifies the stack so that execution returns through kernel code skipping any stack frames created by process code.

In C#, methods whose code executes outside the current GC domain[9] are marked as `extern` methods and tagged with the `[OutsideGCDomain]` custom attribute to distinguish them from other `extern` methods.

7. Shared System Heap

[To be edited by BZill]

Channels[10] provide the primary means of inter-process communication in Singularity. The data which can be passed over channels are restricted to *exchangeable types*[11] and are allocated from a special heap. This section discusses the design of this special heap, the shared system heap.

7.1. What makes the shared system heap special?

1. It is shared among multiple processes.
2. We need to be able to identify and quickly free all the allocations “owned” by a given process (e.g. on process death).
3. The process “ownership” of a given allocation may change over time.

---

[9] For example, kernel code called from a process or process code called from the kernel.

[10] See SDN 5

[11] See SDN 6

Copyright © Microsoft Corporation. All Rights Reserved.

4. We want to be able to split a given allocation and allow the pieces to go to different processes.
5. We would like to be able to share read-only allocations between multiple processes (this would be invisible to the processes in question; to them it should be indistinguishable to having their own read-only copy).

7.2. Corresponding consequences (by number) of being special:

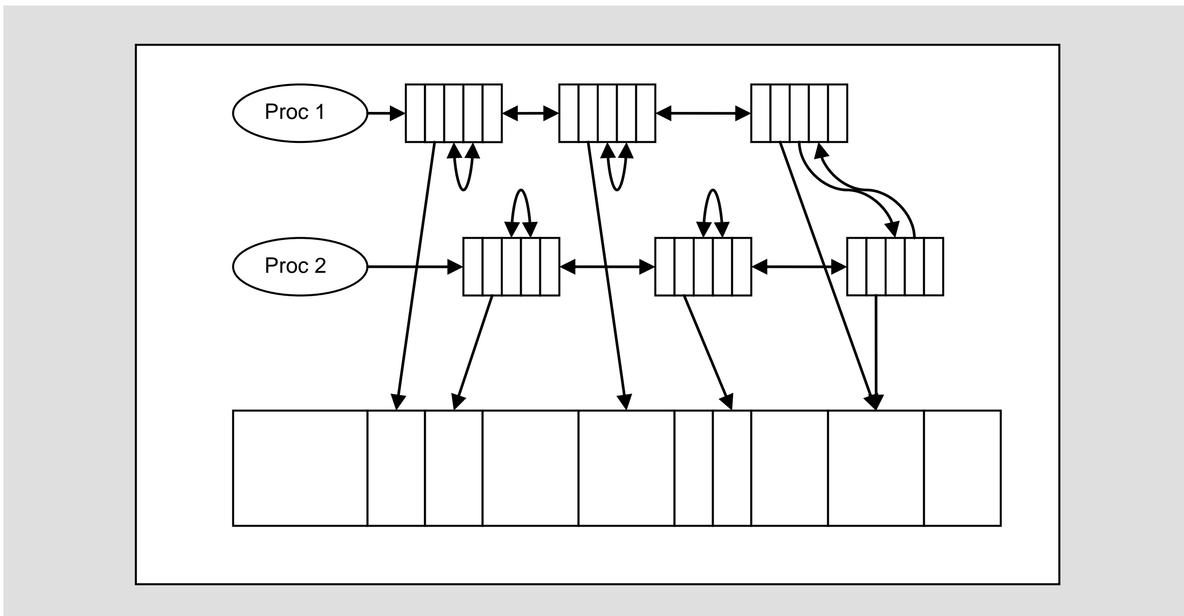
1. We will need to track process "ownership" of allocations somehow.
2. A suitably fast way to do this would be to keep a per-process linked-list of allocations "owned" by that process.
3. To quickly change ownership of an allocation, we'll likely want the above list to be doubly-linked (for quick removal of an allocation from the old owner's list).
4. This means we can't keep any heap (or type system) meta-data as part of the allocations themselves. Thus the meta-data must live off to the side.
5. Now instead of a single owner, we need a means to track multiple owners of a given allocation and only free the allocation when all of the owners go away. One way to do this would be to have a list of "other owners" of the allocation (only the allocation itself would be shared, each owner would have their own meta-data for the entry).

7.3. Strawman proposal:

Each allocation in the heap proper consists of only the bare data bytes. Any sort of type or length information is kept in a separate meta-data block off to the side. The runtime accesses the data indirectly via the meta-data block (which contains a pointer to the data). The meta-data blocks are on a doubly-linked list of such blocks belonging to their owner. Ownership transfer involves moving a block to a different owner's list. Optionally, the meta-data blocks are also on a separate single or doubly-linked ring list of "other owners". If we assume the number of processes sharing a given allocation is small, we should be able to get away with using only a singly-linked list for this and still have good performance.

Copyright © Microsoft Corporation. All Rights Reserved.

7.4. Diagram with circles and arrows:



7.5. A cost/benefit analysis

In terms of the meta-data per allocation, we're using two pointers for the doubly-linked per-process list, one or two (two shown above) pointers for the "other owners" ring, and a pointer to point to the actual data allocation. We may also need a length field. Any meta-data the type system needs to describe this data would also be stored in this block.

7.6. Meanwhile, back in the heap…

The above strawman only covers the meta-data management, and not the actual allocation and freeing of the data regions. We still need an internal representation for the free regions of the heap.

A complication is that we're allowing allocated regions to be arbitrarily split at a later date – should the split-up pieces return to the free list individually as new smaller regions, or should we wait for all the pieces to be freed before returning the single coalesced region back to the free list? The former approach has higher memory availability (since unused memory can always be immediately returned to service), but at the cost of potentially higher memory fragmentation and/or greater complexity to handle eventual coalescing of neighboring free blocks.

8. Shared System Types

Singularity implements a simple shared type system to enable global references to type information. The shared type system augments, but does not replace, the program type system created by Bartok for each process.

Copyright © Microsoft Corporation. All Rights Reserved.

Each shared system type is represented by a `SystemType` structure:

```
namespace Microsoft.Singularity.V1.Services
{
    [StructLayout(LayoutKind.Sequential)]
    public struct SystemType
    {
        internal unsafe SystemType * baseType;     // Initialized by Kernel.
        ...
    }
}
```

The shared system types are meant to be used by the kernel and trusted runtime code only. By exposing system types using pointers to structures, safe application code cannot access them.

The ABI exposes the following methods for accessing shared system types:

```
namespace Microsoft.Singularity.V1.Types
{
    public struct SystemType
    {
        public static unsafe SystemType * Register(int strongname);
        public static unsafe void IsSubtype(SystemType *base, SystemType *type);
    }
}
```

9. Channels and Endpoints

The implementation of channels and endpoints is split between the trusted runtime and the kernel. The trusted runtime is responsible for all data allocation and manipulation. The kernel is responsible for scheduling and accounting.

Copyright © Microsoft Corporation. All Rights Reserved.

A channel is realized as two endpoint structures allocated from the shared system heap. The exact structure of each endpoint structure is specific to the channel contract provided by the endpoint. However, all endpoints share a common prologue defined by the `Endpoint` structure:

```
namespace Microsoft.Singularity.V1.Channels
{
    [StructLayout(LayoutKind.Sequential)]
    public struct Endpoint
    {
        public unsafe SystemType * type;    // Initialized by kernel.
        public unsafe Endpoint * peer;      // Initialized by kernel.
        ...
        public unsafe Endpoint * waitList;
    }
}
```

The kernel exposes the following methods related to channels:

```
namespace Microsoft.Singularity.V1.Channels
{
    [StructLayout(LayoutKind.Sequential)]
    public struct Endpoint
    {
        public static unsafe Register(SystemType *contract,
                                      Endpoint *imp,
                                      Endpoint *exp);
        public static unsafe Notify(Endpoint *endpoint);
        public static unsafe Wait(Endpoint *endpoint);
        public static unsafe WaitAll(Endpoint *endpointList);
        public static unsafe Acquire(Endpoint *endpoint,
                                     Endpoint *container);
        public static unsafe Release(Endpoint *endpoint,
                                     Endpoint *container);
        public static unsafe Unregister(Endpoint *endpoint);
    }
}
```

Endpoints are allocated from the shared system heap by the trusted runtime. Once allocated, the two endpoints must be registered with the kernel as a channel using the `Endpoint.Register` ABI. When sending a message, the sender writes the message contents into the receiving endpoint, and then notifies the kernel that the message is ready using the `Endpoint.Notify` ABI. To wait for an incoming message on one endpoint, the receive uses calls the `Endpoint.Wait` ABI, and then removes the message contents from the endpoint structure. A receiver can wait for multiple channels by creating a linked list using the `Endpoint.waitList` pointers, and then calling the `Endpoint.WaitAll` ABI. To place an endpoint in a message, the sender uses the `Endpoint.Release` ABI. The receiver uses the `EndPoint.Acquire` ABI to extract the endpoint from a message. A process calls the `Endpoint.Unregister` ABI to decommission the endpoint before freeing it from the shared system heap.

Copyright © Microsoft Corporation. All Rights Reserved.