• Singularity Overview (Jim)
  – Rationale & key decisions
  – Singularity architecture

• Singularity Details (Galen)
Singularity Project

- Large Microsoft Research project with goal of more robust and reliable software
  - Galen Hunt, Jim Larus, and many others

- Started with firm architectural principles
  - software will fail, system should not
  - system should be self-describing
  - verify as many system aspects as possible

- No single magic bullet
  - mutually reinforcing improvements to languages and compilers, systems, and tools
1. Use safe programming languages everywhere
   - **safe** ⇒ type safe and memory safe (C#)
   - **everywhere** ⇒ applications, extensions, OS services, device drivers, kernel

2. Improve system resilience in the face of software errors
   - failure containment boundaries
   - explicit failure notification model

3. Facilitate modular verification
   - make system “self-describing,” so pieces can be examined in isolation
   - specify and check behavior at **many** levels of abstraction
   - facilitate automated analysis
Deemphasized Performance

• Easy to measure, but less important than dependability

• “Good enough” performance was goal
  – Singularity has very good performance
Singularity OS Architecture

- **Safe micro-kernel**
  - 95% written in C#
    - 17% of files contain unsafe C#
    - 5% of files contain x86 asm or C++
    - services and device drivers in processes

- **Software isolated processes (SIPs)**
  - all user code is verifiably safe
  - some unsafe code in trusted runtime
  - processes and kernel sealed at start time

- **Communication via channels**
  - channel behavior is specified and checked
  - fast and efficient communication

- **Working research prototype**
  - not Windows replacement
• Process contains only safe code
• No shared memory
  – communicates via *messages*
• Messages flow over channels
  – well-defined & verified
• Lightweight threads for concurrency
• Small binary interface to kernel
  – threads, memory, & channels
• Seal the process on execution
  – no dynamic code loading
  – no in-process plug-ins
• Everything can run in ring 0 in kernel memory!
Modern, safe programming languages
- prevent entire classes of (serious) defects
- easier to analyze

Singularity is written in extended C#
- Spec# (C# + pre/post-conditions and invariants)
- Sing# adds features to increase control over allocation, initialization, and memory layout

Evolve language to support Singularity abstractions
- channel communications
- factor libraries into composable pieces
- compile-time reflection

Native compiler and runtime
- no bytecodes or MSIL
- no JVM or CLR
Managed Language Runtime

- JVM & CLR not appropriate for building systems
- Rich runtime ("one size fits all")
  - monolithic, general-purpose environment
  - large memory footprint (~4 MB/process for CLR)
  - many OS dependencies (CLR PAL requires >300 Win32 APIs)
- JIT compiler
  - increases runtime size and complexity
  - unpredictable performance
- Replicate OS functionality
  - security, threading, configuration, etc.
Singularity Runtime

- Small, fast execution environment
- Ahead-of-time, global optimizing compiler (Bartok)
  - specializes runtime and libraries
  - eliminates unused language features and application or library code
- Factorable runtime and libraries
- Language runtime, garbage collector, and libraries selectable on per-process basis
  - reduce memory and computation overhead
  - enforce design discipline and system policies per process
• Cannot build software without defects
  – verification is a chimera (but we could still do a lot better)

• Software defects should not cause system failure

• A resilient system architecture should
  – isolate system components to prevent data corruption
  – provide clear failure notification
  – implement policy for restarting failed component

• Existing system architectures lack isolation and resilience
Open Process Architecture

• Ubiquitous architecture (Windows, Unix, Java, etc.)
  – DLLs, classes, plug-ins, device drivers, etc.

• Processes are not sealed
  – dynamic code loading and runtime code generation
  – shared memory
  – system API allow process to alter another’s state

• Low dependability
  – 85% of Windows crashes caused by third party code in kernel
  – interface between host and extension often poorly documented and understood
  – maintenance nightmare
Single Process Architecture

- Traditional safe language architecture
  - Xerox PARC (Cedar, Smalltalk, etc.) and Lisp Machine model
  - Java and .NET as well

- Code and data in single address space
  - language and memory safety isolates tangled data and code
  - garbage collection reclaims resources
  - dynamic code loading and runtime code generation
Java Isolates and .NET AppDomains

- Java isolates and CLR AppDomains have complex, shared runtime — single point of failure
- Shared runtime must also satisfy different applications’ requirements
Sealed Processes

- Singularity processes are sealed
  - no dynamic code loading or run-time code generation
    - all code present when process starts execution
  - extensions execute in distinct processes
    - separate closed environments with well-defined interfaces
  - no shared memory

- Fundamental unit of failure isolation
- Improved optimization, verification, security
Isolation Requires Lightweight Processes

• Existing processes rely on virtual memory and protection domains
  – VM prevents reference into other address spaces
  – protection prevents unprivileged code from access system resources

• Processes are expensive to create and schedule
  – high cost to cross protection domains (rings), handle TLB misses, and manipulate address spaces

• Cost encourages monolithic architecture
  – expensive process creation and inter-process communication
  – large, undifferentiated applications
  – dynamically loaded extensions
Software Isolated Processes (SIPs)

- Protection and isolation enforced by language safety and kernel API design
  - process owns a set of pages
  - all of process’s objects reside on its pages (object space, not address space)
  - language safety ensures process can’t create or mutate reference to other pages

- Global invariants:
  - no process contains a pointer to another process’s object space
  - no pointers from exchange heap into process
• Channels are strongly typed (value & behavior), bidirectional communications ports
  – messages passing with extensive language support

• Messages live outside processes, in exchange heap
  – only a single reference to a message

• “Mailbox” semantics enforced by linear types
  – copying and pointer passing are semantically indistinguishable

• Channel buffers pre-allocated according to contract
OS Controls Resources and Security

• OS owns, allocates, and reclaims system resources
  – conventional model

• On process termination, OS reclaims memory pages and channels
  – not dependent on finalization or garbage collection

• Clean failure notification
  – sent messages still available to other process

• Security policy on per-process
  – crux is control of channels
Would You Trust Your System to a Type System?

- Process integrity depends on type and memory safety – currently trust compiler and runtime
- TAL can remove compiler from trusted computing base
- We are working on verifying GC and runtime as well

Singularity system

TAL

MSIL+

csc

sgc

compiler verification

byte code verification

Sing# C# source

application verification

Microsoft
Still Not Convinced?

- Hardware Protection Domains
  - virtual address space
  - contains one or more SIPs
  - runs at ring 0 ("kernel domain") or ring 3
Domains: Monolithic Kernel

- App 1
- App 2
- App 3

- File System
- Net Stack

- Kernel

- SIP
- Protection Domain
- Ring 3
- Ring 0
Domains: Novel Models

- Signed Extension
- Unsigned Extension
- Ring 3
- Ring 0
- SIP
- Protection Domain
Challenge 3: Verify More

- Process internals (code):
  - type safety
  - object invariants
  - method pre- & post- conditions
  - component interfaces

- Process externals:
  - channel contracts
  - resource access & dependencies

- System:
  - communication safety
  - hardware resource conflict free
  - namespace conflict free

- Static verification: before code runs

Diagram:
- Channels: content extension, web server, TCP/IP stack, network driver
- Processes: ext. class library, server class library, tcp class library, driver class library
- Kernel API: page mgr, scheduler, chan mgr, proc mgr, i/o mgr, kernel class library, HAL
public contract IoStream {
...
  state Start : {
    Open? -> {
      OK! -> Opened;
      Error! -> End;
    }
  }

  state Opened : {
    Read? -> Data! -> Opened;
    Write? -> OK! -> Opened;
    Close? -> OK! -> End;
  }

  state End;
...
}

? = receive
! = send
Example: Contract Conformance

Contract

```java
public contract TcpSocketContract {
    ...
    state Connected : {
        Read? -> ReadResultPending;
        Write? -> WriteResultPending;
        GetLocalAddress? -> IPAddress! -> Connected;
        GetLocalPort? -> Port! -> Connected;
        DoneSending? -> ReceiveOnly;
        DoneReceiving? -> SendOnly;
        Close? -> Closed;
        Abort? -> Closed;
    }

    state ReadResultPending : {
        Data! -> Connected;
        NoMoreData! -> SendOnly;
        RemoteClose! -> Zombie;
    }
}
```

Web Server (User)

```java
... conn.SendRead();
switch receive {
    case conn.Data(readData) :
        dataBuffer.AddToTail(readData);
        return true;
    case conn.RemoteClose() :
        return false;
} ...
```

**Missing Case**

```java
    case conn.NoMoreData() :
```
Example: Configuration Specifications

```csharp
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]
class S3Trio64Config : DriverCategoryDeclaration
{
    [IoMemoryRange(0, Length = 0x400000)]
    IoMemoryRange frameBuffer;

    [IoFixedMemoryRange(Base = 0xb8000, Length = 0x8000)]
    IoMemoryRange textBuffer;

    ...

    [IoFixedPortRange(Base = 0x3c0, Length = 0x20)]
    IoPortRange control;

    [ExtensionEndpoint(typeof(ExtensionContract.Exp))]
    TRef<ExtensionContract.Exp:Start> pnp;

    [ServiceEndpoint(typeof(VideoDeviceContract.Exp))]
    TRef<ServiceProviderContract.Exp:Start> video;
    ...
}
```
Driver (Source + Spec) → Driver Manifest → Driver Manifest Conflict → System Manifest

File System:
- driver class library
- runtime

Disk Driver:
- kernel class library
- runtime

1. Load driver
2. Allocate I/O objects
3. Create channels

Conflicts:
- Disk Driver
- File System

HAL

Singularity
Can we fundamentally improve the dependability of today’s software? (yes)

Rethink language, OS, and system architecture assumptions
- OS should control application’s execution environment
- new mechanisms to enhance system integrity, verifiability, and dependability

Programming languages and runtime systems are central to new architecture

Singularity is a complete (but simple) system
- safe languages all the way down to the hardware
- OS architecture improves system integrity and verification
- many more aspects of system behavior are verifiable

We are distributing Singularity (source code)

http://research.microsoft.com/os/singularity