SLAM
  – shipping in Windows DDK

ESP
  – extensively used in Windows (CSE)

Fugue
  – shipping in FxCop

SAL (Standard Annotation Language)
  – most MS .h files annotated
  – shipping in VS2005

SpecExplorer
  – widely used within MS
Frustration, Despite Progress

• “First we bug the software, then we debug it”
  – existing code bases written without many tools

• Specification unavailable

• Program analysis difficult
  – unsafe language
  – tenuous assumptions (e.g., code completely known and immutable)

• Struggle to incorporate tools in process

• Verification and testing stuck at low level
  – language features and programming interfaces
“Modern”
OS & Applications

• Design parameters
  – scarce resources
  – benign environment
  – knowledgeable and trained users
The World Changed

• Hardware and software industries were wildly successful
  – machines are fast, memory is cheap
  – computers are ubiquitous

• Malicious environment
  – ubiquitous worms, viruses, scams, attacks, ...

• Few users understand computers or software
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# or Java)
   – 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
   – make automated analysis easier
Deemphasized Performance

• Easy to measure, but less important than dependability

• “Good enough” performance was goal
  – result had very good performance
Singularity OS

• 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 Model

- 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
- Runs in ring 0 in kernel memory!
Challenge 1: Pervasive Safe Languages

• 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
• 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
Challenge 2: Improve Resilience

• 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
### Program Optimization

<table>
<thead>
<tr>
<th>Program</th>
<th>Code Whole</th>
<th>Code w/ Tree Shake</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>2.37 MB</td>
<td>1.29 MB</td>
<td>46%</td>
</tr>
<tr>
<td>IDE Disk Driver</td>
<td>1.85 MB</td>
<td>455 KB</td>
<td>75%</td>
</tr>
<tr>
<td>Web Server</td>
<td>2.73 MB</td>
<td>765 KB</td>
<td>72%</td>
</tr>
<tr>
<td>Content Extension</td>
<td>2.14 MB</td>
<td>502 KB</td>
<td>77%</td>
</tr>
</tbody>
</table>

- Closed world allows global optimization to eliminate unused code
- Reduces process code size by up to 75%
- Fewer code paths ⇒ better optimization & error analysis
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 accessing system resources

- Processes are expensive to create and schedule
  - high cost to cross protection domains (rings), handle TLB misses, and manipulate address spaces
  - (talk in MSPC Workshop, 8:45 Sunday)

- 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
Interprocess Communications

- 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
## Performance Micro Benchmarks

- **Why?**
  - static verification replaces hardware protection
  - all SIPs run in ring 0
  - good optimizing compiler (not JIT)

<table>
<thead>
<tr>
<th>Athlon64 3000+ (1.8GHz) nForce4 SLI</th>
<th>Cost (CPU Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Singularity</td>
</tr>
<tr>
<td>Minimum kernel API call</td>
<td>80</td>
</tr>
<tr>
<td>Message request/reply</td>
<td>1,041</td>
</tr>
<tr>
<td>Process create &amp; start</td>
<td>388,000</td>
</tr>
</tbody>
</table>
- 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
Hardware Protection Domains

• A virtual address space
• Contains one or more SIPs
• Run 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

- App1
- Signed Extension
- Unsigned Extension
- Unsigned App2
- Signed Driver
- Unsigned Driver
- Kernel
- SIP
- Protection Domain
- Ring 3
- Ring 0
Hardware is Costly

Webfiles Macrobenchmark

Unsafe Code Tax

Safe Code Tax

-4.7%
+6.3%
+18.9%
+33.0%
+37.7%

No runtime checks  Physical Memory  Add VM  Add Separate Address Space  Add Ring 3  Full Microkernel
Challenge 3: Facilitate Verification

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

- Verify process externals:
  - channel contracts
  - resource access & dependencies

- Verify system:
  - communication safety
  - hardware resource conflict free
  - namespace conflict free

- Static verification: before code runs
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
missing_case conn.conn.NoMoreData() :
```
Example:

Configuration Specifications

```c
[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;

    ...
}
```

- Requires PCI device
- Requires 4MB frame buffer (from in PCI config)
- Requires system console buffer
- Requires VGA I/O ports
- Requires control by plug-and-play system
- Provides video capability to system
Specification Usable in Many Ways

1. Load driver
2. Allocate I/O objects
3. Create channels
• Can we fundamentally improve the dependability of today’s software?

• 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

• [http://research.microsoft.com/os/singularity](http://research.microsoft.com/os/singularity)
Singularity Research Team

• Lead by Galen Hunt and Jim Larus

• MSR Cambridge
  – Paul Barham, Richard Black, Tim Harris, Rebecca Isaacs, Dushyanth Narayanan

• MSR Redmond
  – Advanced Compiler Technology Group:
    • Juan Chen, Qunyan Mangus, Mark Plesko, Bjarne Steensgaard, David Tarditi
  – Foundations of Software Engineering Group:
    • Wolfgang Grieskamp
  – Operating Systems Group:
    • Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, Steven Levi
  – Security and Distributed Systems:
    • Dan Simon, Brian Zill
  – Software Design and Implementation Group:
    • John DeTreville, Ben Zorn
  – Software Improvement Group:
    • Manuel Fahndrich, James Larus, Sriram Rajamani, Jakob Rehof

• MSR Silicon Valley
  – Martin Abadi, Andrew Birrell, Ulfar Erlingsson, Roy Levin, Nick Murphy, Ted Wobber