• MSR has researched, developed, and deployed programming tools for 10+ years

• Struggle to bring tools to practice
  – incomplete, informal specification
  – unsafe languages (C)
  – tenuous assumptions (e.g., code completely known and immutable)

• Verification and testing at a low level
  – language features and library APIs
  – persistent question: “are these the right bugs?”

• Software development is inherently broken
  – “first we bug the software, then we debug it”

• People and organization may be more important than bugs and testing
  – (another talk)
Singularity

- Microsoft Research project with goal of more robust and reliable software
- Rethink the software stack
- Articulated architectural principles
  - software will fail, system should not
  - system should be self-describing
  - verify as many aspects as possible
- No single magic bullet
  - mutually reinforcing improvements to languages and compilers, systems, and tools
Un fortunately, this willingness to begin with an entirely new foundation is not located within the Windows group but in Microsoft’s research arm, where scientists and their heretical thoughts are safely isolated. Last April, Microsoft publicly unveiled the five-year-old research project, called “Singularity.” It is nothing more than a neat academic exercise, not a glimpse of Windows 7. “Singularity is not the next Windows,” said Rich Rashid, the company’s senior vice president overseeing research. “Think of it like a concept car.”

Judging from recent rumours, that’s what it is preparing to do. Even though it won’t be in Windows 7, Microsoft is happy to talk about “MinWin”—a slimmed down version of the Windows core. It’s even willing to discuss its “Singularity” project—a microkernel-based operating system written strictly for research purposes. But ask about a project code-named “Midori” and everyone clams up.
Why Rethink Software Architecture?

- Design parameters
  - scarce resources
  - assembly code
  - benign environment
  - knowledgeable users
The World Changed

- Hardware and software industries were wildly successful
  - machines are fast, memory is cheap
  - computers are ubiquitous
  - safe, high-level languages

- Malicious environment
  - ubiquitous worms, viruses, scams, attacks, ...

- Few users understand computers or software
Key Singularity Tenets

1. Use safe (managed) programming languages everywhere
   - safe $\Rightarrow$ type safe and memory safe (C# or Java)
   - everywhere $\Rightarrow$ 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. Modular verification
   - design assuming automated analysis
   - seal environments so verification can be sound
   - make system “self-describing,” so pieces can be examined in isolation
   - specify and check behavior at many levels of abstraction
• Easy to measure, but less important than dependability

• “Good enough” performance was goal
  – Singularity has very good performance
• Safe micro-kernel
  – 95% written in C#
  – all services and drivers in processes

• Software isolated processes (SIPs)
  – all user code is verifiably safe
  – some unsafe code in trusted runtime
  – processes and kernel sealed at execution

• Communication via channels
  – channel behavior is specified and checked
  – fast and efficient communication

• Working research prototype
  – not Windows replacement
  – shared source download
- Process contains only safe code
  - except language runtime (GC)
- No shared memory
  - communicate via messages
- Messages flow over bi-directional channels
  - well-defined & verifiable
- Small, versioned interface to kernel
  - threads, memory, & channels
- Seal process on execution
  - no dynamic code loading
  - no in-process plug-ins
Challenge 1: Pervasive Safe Languages

- Modern, safe programming languages
  - preclude entire categories of serious defects, e.g. buffer overruns
  - easier to analyze

- Singularity is written in Spec#
  - C# + pre/post-conditions and invariants

- Language research to support Singularity abstractions
  - channel communications
  - factor libraries into composable pieces
  - compile-time reflection

- Native compiler and runtime system
  - no bytecodes or MSIL (not JVM or CLR)
Runtime System

- 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

- Minimal run-time system
- Ahead-of-time, global optimizing compiler (Bartok)
  - specializes runtime and libraries
  - eliminate 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 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 (Windows, Unix, Java, browsers, 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

- Tangled code and data
  - language safety provides some isolation
  - garbage collection must reclaim resources
  - dynamic code loading and runtime code generation
  - runtime is single point of failure
Singularity 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

- Enhance optimization, verification, security
  - “closed world” assumption is true!
### 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>2,371 KB</td>
<td>1,291 KB</td>
<td>46%</td>
</tr>
<tr>
<td>Web Server</td>
<td>2,731 KB</td>
<td>765 KB</td>
<td>72%</td>
</tr>
<tr>
<td>SPECweb99 Plug-in</td>
<td>2,144 KB</td>
<td>502 KB</td>
<td>77%</td>
</tr>
<tr>
<td>IDE Disk Driver</td>
<td>1,846 KB</td>
<td>455 KB</td>
<td>75%</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
• Move register allocator from Bartok compiler to separate process
  – one of 50 phases
  – updates IR to assign registers and insert spill operations
  – 156 shared classes (IR + ISA)
  – invoked per function
    • 6,441 calls in kernel compile
    • 50KB to 1.5MB of data
### Code Complexity

<table>
<thead>
<tr>
<th>Code</th>
<th>Lines</th>
<th>% of Orig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bartok Compiler</td>
<td>210,000</td>
<td>95.45%</td>
</tr>
<tr>
<td>Register Allocator</td>
<td>10,000</td>
<td>4.55%</td>
</tr>
<tr>
<td>Altered in Host</td>
<td>3</td>
<td>+0.00%</td>
</tr>
<tr>
<td>Limited Marshal Tags</td>
<td>107</td>
<td>+0.05%</td>
</tr>
<tr>
<td>Channel and Child</td>
<td>400</td>
<td>+0.18%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>220,510</td>
<td><strong>100.25%</strong></td>
</tr>
</tbody>
</table>

- Child process adds < .25% (508 lines) to code base
- Time to compile Singularity kernel increases by 11%
Isolation Requires Lightweight Processes

• Traditional processes rely on virtual memory and hardware domains
  – legacy of assembly language era
  – 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

• Costs 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 has exclusive access to 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
  – everything can run in ring 0 in kernel memory (without the MMU)!

• 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 language support

- Messages live outside processes ("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)

## Cost (CPU Cycles)

<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
  – messages in channel still available to other process

• Security policy on per-process
  – crux is control of channels (capabilities)
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
- Need to verify GC and runtime as well (research challenge)
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**

[Image of green leaves and trees]
Domains: Novel Models

- Novel Models
- Signed Extension
- App1
- Signed Extension
- Unsigned Extension
- App2
- Unsigned Driver
- Signed 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

0.00 0.20 0.40 0.60 0.80 1.00 1.20 1.40

Microsoft
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
Boogie and Singularity

Network device driver
• Hardware interactions
  – Registers, DMA, etc.
• Layered objects
  – Main object owns others
• Driven by WDF interface
  – Order of invocations

Scheduler (kernel)
• Global properties
  – Interrupts, lock levels, etc.
• Strongly coupled objects
  – Many object references
• Concurrent
  – Interrupts, multithreading

Use Boogie to verify interesting properties in both contexts (Kevin Bierhoff 2007)
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

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;
  }

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

  ... 

  Missing Case

  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;

    ...
}
```

- **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
• Software architecture can enhance verification
  – modern programming language with appropriate abstractions
  – sealed processes
  – explicit communications across typed channels
  – specifications throughout system

• Verification community should be involved in system design
  – your insights and contributions can make software better (e.g. TM)
  – improved reliability and robustness are achievable
  – “verify what they build” approach complicates verification and produces poor software

• Challenge: make reliability and robustness more important than performance
Conclusion

• **Can we fundamentally improve the dependability of software?** (yes)

• Reexamine and 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

• Singularity project is done
  – using Singularity in small computer (cPhone)
  – working with incubation team that is expanding Singularity
Obtaining Singularity

• Info:
  –http://research.microsoft.com/os/singularity

• Code:
  –http://www.codeplex.com/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, Quynan 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