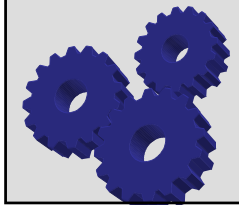
# Protection and Extension in the Microsoft Singularity Operating System

Michael Spear

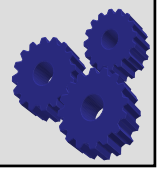
csc 256/456

25 April 2007



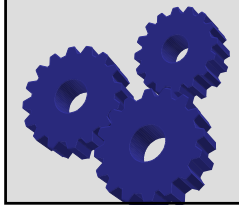
## Outline

- Background
  - Protection and Isolation
  - Recent Software Innovations
- Singularity
  - End to End System
  - Trust -vs- Verification
  - Safe Microkernel Code
  - Software Isolated Processes
  - IPC via Channels
  - Manifests and Installation
  - Applications and Device Drivers
- Summary



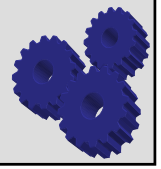
## Protection and Isolation

Why are they needed?



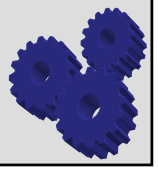
## Protection and Isolation… Why?

- Protect kernel from application code / protect applications from each other
  - Limit impact of bugs
  - Prevent malice
- Justification
  - Your application should not touch my application's memory in unexpected ways



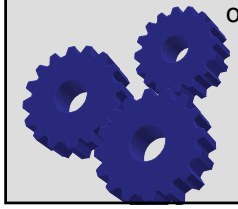
## Protection and Isolation… Why?

- Restrict access to bare hardware
  - Device drivers and kernel only
- Justification
  - Safety and security (i.e. restrict access to DMA)
  - Necessary for resource management



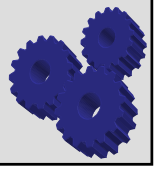
## Protection and Isolation

How are they provided
on conventional systems?



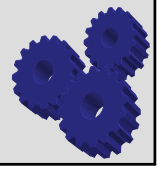
## Protection and Isolation… How?

- Hardware
  - Protected and Privileged modes
    - Certain instructions are available only to 'privileged-mode' applications (i.e. the kernel)
    - Exception if protected-mode application uses these instructions
    - ex - read/write to I/O port, set IRQ
  - Virtual addressing
    - Every application (along with its extensions) lives in a separate virtual address space
    - No matter what the app does, it can't harm anyone else



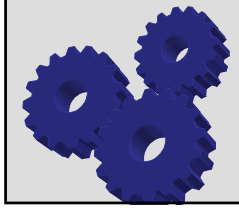
## Protection and Isolation… How?

- Software
  - The process abstraction
    - Encapsulate an application and its extensions
    - Give it its own virtual address space (memory safety)
    - Restrict its access to bare hardware (protected mode)



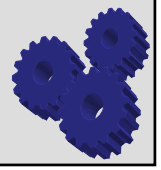
## Protection and Isolation

What are the problems?



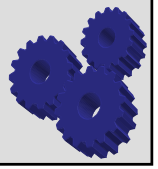
## Protection and Isolation… Problems

- Performance
  - IPC is expensive and slow
    - Requires kernel crossing, TLB remap, virtual page translation
    - Often fallback to shared memory between processes, violating process protection
  - Switching apps is slow
    - Context switch overhead is tremendous



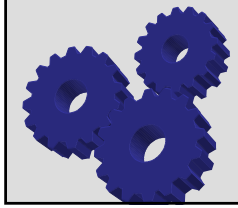
## Protection and Isolation… Problems

- Correctness
  - Extensions have access to *everything* and can see the whole process space
  - Cannot assume system states, invariants, or transitions
    - JVM: Any interrupt, thread switch, or exception can result in a new file overwriting a class and method body
    - Reflection: Can inspect class internals, get around information hiding and data abstraction
  - Drivers are known to be incorrect
    - But 85% of Windows crashes are from drivers [Swift et al. OSDI 2004]
    - Linux isn't much better [Padioleau et al. EuroSys2006]



## Software in the New Millennium

What's different now?



## Software Innovations

- Languages: Type and Memory Safety
  - Objects are interpreted and manipulated in the right ways
  - Pointers are only to valid points within live objects
  - Metadata can be part of program
- Compilers: Speed and safety
  - Compiler can ensure that privileged instructions aren't emitted except in verifiable ways
  - Compiler can output native code from intermediate code: little performance penalty even with garbage collection
  - Run-time checks prevent many errors, but static analysis limits their overheads

## Software Innovations

- Software Analysis
  - Typed Assembly Language and Typed Intermediate Language make it easy to parse and analyze code, prove properties
- Validation: end-to-end safety from code → compiler → executable → running code
- Analysis tools:
  - Sound: find all errors (plus false positives)
  - Specification-driven: no fixed collection of bugs (adapts; tests abstractions)

## Singularity Philosophy

- The whole systems software community (Theoretical, OS, Languages, Compilers, Verification and Analysis) has done an awful lot since 1970, and maybe it's time to stop exclusively using a 1970 design philosophy for operating systems
  - Use software to provide safe, uniform extensibility
  - Detect errors early (design or compile time, not execution or crash time)
  - Don't use any hardware protection
    - Everything runs in Ring0, single address space

## Key Concept: End to End System

- A safe system is more than just a kernel
  - Compiler outputs safe applications
  - Application uses safe libraries (and safe generics)
  - Installer ensures applications obey system policy
  - Boot Loader ensures nothing was changed off-line
  - Drivers are sandboxed and monitored
  - Application configuration is verified

## Key Concept: Trust -vs- Verification

- Some code is simply safe
  - Pure MSIL with no [unsafe] metadata
- Some code is trusted
  - GC (if not type safe), HAL
- Some code is verifiably safe
  - Pure MSIL + GC + call to HAL that obeys certain properties
- Native version of verified MSIL is safe
  - *If you trust the compiler…*
  - Trusted code is injected into verified code at install time... it's like inlining syscalls... but it's safe
- Typed Assembly Language and Proof-Carrying Code are safe (Future direction)
  - Key recent innovation: Garbage Collected TAL

## Key Concept: Safe Microkernel

- Written in C#
  - Most kernel code is verified C# (exceptions: 13,000 lines of C/assembly in HAL, a few runtime components)
  - Kernel has its own GC
- Kernel ABI
  - ABI is versioned
  - All parameters to ABI calls are value types
  - ABI calls are often inlined
- Modularity
  - ex: can select a scheduler at build time
- Blurred Boundary
  - With inlined ABI calls and no HW protection, boundary between kernel and applications is very blurry

## Key Concept: Software Isolated Processes

- Closed object space (not memory space)
  - All SIPs can share an address range, but can't see each other's objects (64-bit address space)
  - Linked stacks
- Closed code space
  - No reflection, no dynamic loading/generating code
- IPC via channels
- Applications are written in safe code (Sing#)
  - App should never have [unsafe] tags
  - Sing# == C# + channel support + additional safety

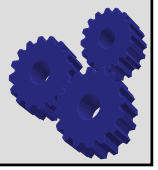
## Key Concept: IPC via Channels

- Channels are safer than pipes and shared memory
  - Messages are strongly typed
  - Communication must obey a contract (protocol)
  - IPC can be statically verified
  - Built into Sing#
  - Programmers must consider extensibility at design time
  - Only values are communicated, not objects (marshalling)

If two processes are to communicate, they had better agree on HOW

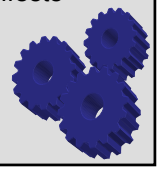
## Key Concept: IPC via Channels

- Channels are fast
  - Lots of memory tricks to transfer memory from one GC domain to another
  - No copying on IPC… each level of the network stack can be its own SIP



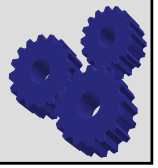
## Key Concept: Manifests

- Configuration is strongly typed, verified
  - Application == Code + Configuration (Manifest)
  - Manifest is mostly inferred from metadata tags on objects that represent resources (devices, channels) and arguments
    - Lists all dependencies, all exported channels
    - Specifies valid arguments / invocations
      - It's theoretically possible to auto-generate the --help invocation code for an application from the tags on its command-line parameters
- System manifest aggregates application manifests
  - Enables off-line analysis



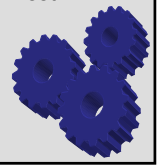
## Key Concept: Installation is a First-Class Operation

- Verify at Install Time
  - Apply standard system policy based on application classification
    - Prohibit use of certain objects or channels
  - Apply local policy too
- Optimize at Install Time
  - Apps are compiled and statically linked
  - Runtime, GC, and ABI calls are inlined
  - Whole program analysis and optimization
  - Result:  Sealed native code binary
- Reconfigure at Install Time
  - Build application manifest
  - Update system manifest



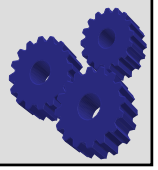
## Key Concept: Device Drivers are Applications

- Device drivers are a special class of applications
  - Permitted to use certain objects (IOObject hierarchy), but lose other privileges (interactivity)
  - Each driver is a SIP, communicates to SIPs via channels
- IOObjects
  - IOObjects are trusted: they contain native code for accessing hardware (IODma, IOIrq, IOMemory, and IOPort)
  - IOObjects can be monitored at run-time (i.e. to detect IRQ remapping)
  - IOObjects are annotated in the code to create manifest



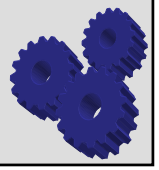
## Key Concept: Safe Boot

- Kernel uses driver manifests to control boot
  - Kernel does full resource discovery of PnP and PCI buses
  - Kernel uses a *resolution algorithm* to match drivers to resources (no etc/init.d)
  - Kernel binds drivers to devices
  - Drivers don't acquire resources at all
    - When the driver is activated, its declared resources are already configured and bound to IOObjects



## Briefly Noted: Hardware Isolated Processes

- Hardware Isolation need not be discarded entirely, but it can be selectively applied
- i.e. put all of your web SIPs in one HIP, the kernel in a HIP, and the shell in another HIP...
  - Big performance penalty
  - Why is this ‘defence in depth’ necessary?



## Summary

- Software Isolated Processes provide uniform extensibility
- As a closed, end-to-end system, Singularity does not require hardware for software protection and extension
  - Some design aspects can't just be added to Windows/Linux/MacOS

## Additional Reading


- A Garbage-Collecting Typed Assembly Language [Hawblitzel et al. TLDI 2007]
- Deconstructing Process Isolation [Aiken et al. MSPC 2006]
- Sealing OS Processes to Improve Dependability and Safety [Hunt et al. EuroSys2007]
- Language Support for Fast and Reliable Message-based Communication in Singularity OS [Fähndrich et al. EuroSys2006]
- Solving the Starting Problem: Device Drivers as Self-Describing Artifacts [Spear et al. EuroSys2006]
- Access Control in a World of Software Diversity [Abadi et al. HotOS 2005]
- Broad New OS Research: Challenges and Opportunities [Hunt et al. HotOS 2005]
- Making system configuration more declarative [DeTreville HotOS 2005]