Singularity vs. the Hard Way
Part 1

Jeff Chase
Today

• **Singularity: abstractions**
• vs. the “Hard Way” (e.g., *i*x)
  – Processes vs. SIPs
  – Protection: hard vs. soft
  – Kernel vs. microkernel
  – Extensibility: open vs. closed

• **Questions**
  – How is the kernel protected?
  – How does it control access to data?
  – How does it keep control?
Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus
Microsoft Research Redmond

galenh@microsoft.com

ABSTRACT
Every operating system embodies a collection of design decisions. Many of the decisions behind today’s most popular operating systems have remained unchanged, even as hardware and software have evolved. Operating systems form the foundation of almost every software stack, so inadequacies in present systems have a pervasive impact. This paper describes the efforts of the Singularity project to re-examine these design choices in light of advances in programming languages and verification tools. Singularity systems incorporate three key architectural features: software-isolated processes for protection of programs and system services, contract-based channels for communication, and manifest-based programs for verification of system properties. We describe this foundation in detail and sketch the ongoing research in experimental systems that build upon it.
Sealing OS Processes to Improve Dependability and Safety

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
singqa@microsoft.com

ABSTRACT
In most modern operating systems, a process is a hardware-protected abstraction for isolating code and data. This protection, however, is selective. Many common mechanisms—dynamic code loading, run-time code generation, shared memory, and intrusive system APIs—make the barrier between processes very permeable. This paper argues that this traditional open process architecture exacerbates the dependability and security weaknesses of modern systems.

As a remedy, this paper proposes a sealed process architecture, which prohibits dynamic code loading, self-modifying code, shared memory, and limits the scope of the process API. This paper describes the implementation of the sealed process architecture in the Singularity operating system, discusses its merits and drawbacks, and evaluates its effectiveness. Some benefits of this sealed process architecture are: improved program analysis by tools, stronger security and safety guarantees, elimination of redundant overlaps between the OS and language runtimes, and improved software engineering.

General Terms
Design, Reliability, Experimentation.

Keywords
Open process architecture, sealed process architecture, sealed kernel, software isolated process (SIP).

1. INTRODUCTION
Processes debuted, circa 1965, as a recognized operating system abstraction in Multics [48]. Multics pioneered many attributes of modern processes: OS-supported dynamic code loading, run-time code generation, cross-process shared memory, and an intrusive kernel API that permitted one process to modify directly the state of another process.

Today, this architecture—which we call the open process architecture—is nearly universal. Although aspects of this architecture, such as dynamic code loading and shared memory, were not in Multics’ immediate successors (early versions of UNIX [35] or early PC operating systems), today’s systems, such as FreeBSD, Linux, Solaris, and
- 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
Programs run as independent processes.

Protected OS kernel mediates access to shared resources.

The kernel code and data are protected from untrusted processes.
Processes and the kernel

• A (classical) OS lets us run programs as processes. A **process** is a running program instance (with a **thread**).
  – Program code runs with the CPU core in untrusted **user mode**.

• Processes are protected/isolated.
  – **Virtual address space** is a “fenced pasture”
  – **Sandbox**: can’t get out. **Lockbox**: nobody else can get in.

• The OS kernel controls **everything**.
  – Kernel code runs with the CPU core in trusted **kernel mode**.
Threads

- A thread is a **stream of control**…
  - Executes a sequence of instructions.
  - Thread identity is defined by CPU register context (PC, SP, …, page table base registers, …)
  - Generally: a thread’s **context** is its register values and referenced memory state (stacks, page tables).

- Multiple threads can execute independently:
  - They can run in parallel on multiple cores...
    - **physical concurrency**
  - …or arbitrarily interleaved on some single core.
    - **logical concurrency**

- A thread is also an OS abstraction to spawn and manage a stream of control.
User/kernel

Kernel code

Kernel space

User space

Safe control transfer

Kernel code

Kernel data
Threads and the kernel

- Modern operating systems have multi-threaded processes.
- A program starts with one main thread, but once running it may create more threads.
- Threads may enter the kernel (e.g., syscall).
- (We assume that) threads are known to the OS kernel.
  - Kernel has syscalls to create threads (e.g., Linux `clone`).
- Implementations vary.
  - This model applies to Linux, MacOS-X, Windows, Android, and pthreads or Java on those systems.
2.1 Software-Isolated Processes

Like processes in many operating systems, a SIP is a holder of processing resources and provides the context for program execution.

Execution of each user program occurs within the context of a SIP.

Associated with a SIP is a set of memory pages containing code and data. A SIP contains one or more threads of execution.

A SIP executes with a security identity and has associated OS security attributes.

Finally, SIPS provide information hiding and failure isolation.
3.2. Software Isolated Processes

A Singularity process is called a software isolated process (SIP):

• A SIP consists of a set of memory pages, a set of threads, and a set of channel endpoints.

• A SIP starts with a single thread, enough memory to hold its code, an initial set of channel endpoints, and a small heap.

• It obtains additional memory by calling the kernel’s page manager, which returns new, unshared pages.

• These pages need not be adjacent to the SIP’s existing address space, since safe programming languages do not require contiguous address spaces.
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
- Everything can run in ring 0 in kernel memory!
SIP safety/isolation

• Language safety ensures that untrusted code cannot create or mutate pointers to access the memory pages of another SIP.

• SIPs do not share data, so all communications occur through the exchange of messages over message-passing conduits called channels.

• The Singularity communication mechanisms and kernel API do not allow pointers to be passed from one SIP to another.
Using a safe language for protection

Singularity’s SIPs depend on language safety and the invariants of the sealed process architecture to provide low-cost process isolation. This isolation starts with verification that all untrusted code running in a SIP is type and memory safe.

Sealing OS Processes to Improve Dependability and Safety

SIPs rely on programming language type and memory safety for isolation, instead of memory management hardware.

Through a combination of static verification and runtime checks, Singularity verifies that user code in a SIP cannot access memory regions outside the SIP.

Singularity: Rethinking the Software Stack
“Lightweight” protection

• Because user code is verified safe, several SIPS can share the same address space. Moreover, SIPS can safely execute at the same privileged level as the kernel.

• Eliminating these **hardware protection barriers** reduces the cost to create and switch contexts between SIPS.

• With **software isolation**, system calls and inter-process communication execute significantly faster (30–500%) and communication-intensive programs run up to 33% faster than on hardware-protected operating systems.

• Low cost, in turn, makes it practical to use SIPS as a fine-grain isolation and extension mechanism.
MSIL

…To facilitate static verification of as many run-time properties as possible, code …is delivered to the system as compiled Microsoft Intermediate Language (MSIL) binaries.

MSIL is the CPU-independent instruction set accepted by the Microsoft Common Language Runtime (CLR) [7]….

Singularity relies on the standard Microsoft Intermediate Language (MSIL) verifier to check basic type safety properties (e.g. no casts from integers to pointers or from integers to kernel handles).

Singularity uses the Bartok compiler [13] to translate an MBP’s MSIL code to native machine language code (such as x86 code).
Hardware-based memory protection (review)

User thread (in VAS) → Translator (MMU) → Physical memory

Virtual address → Physical address

Old example: Base and Bound registers

Bound
Data
Code

Base + Bound

Base

Data
Code

Physical memory
Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel

![Diagram of virtual to physical address translation process.](Diagram.png)
Process Concept

edits

source code

compiler

executable image instructions and data

Operating System Copy

machine instructions  Data  Heap  Stack

machine instructions  Data  Heap  Stack

Process  Operating System Kernel

Physical Memory
Process Concept

• **Process**: an instance of a program, running with limited rights
  – **Process control block**: the data structure the OS uses to keep track of a process
  – **Two parts to a process**:
    • **Thread**: a sequence of instructions within a process
      – Potentially many threads per process (for now 1:1)
      – Thread aka lightweight process
    • **Address space**: set of rights of a process
      – Memory that the process can access
      – Other permissions the process has (e.g., which procedure calls it can make, what files it can access)
UNIX Process Management

```c
int main() {
...
}
```
Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run
Implementing UNIX exec

• Steps to implement UNIX fork
  – Load the program into the current address space
  – Copy arguments into memory in the address space
  – Initialize the hardware context to start execution at `start`
Hardware protection does not come for free

though its costs are diffuse and difficult to quantify.

Costs of hardware protection include maintenance of page tables, soft TLB misses, cross-processor TLB maintenance, hard paging exceptions, and the additional cache pressure caused by OS code and data supporting hardware protection.

In addition, TLB access is on the critical path of many processor designs [2, 15] and so might affect both processor clock speed and pipeline depth. Hardware protection increases the cost of calls into the kernel and process context switches [3]. On processors with an untagged TLB, such as most current implementations of the x86 architecture, a process context switch requires flushing the TLB, which incurs refill costs.
<table>
<thead>
<tr>
<th></th>
<th>Singularity</th>
<th>FreeBSD 5.3</th>
<th>Linux 2.6.11 (Red Hat FC4)</th>
<th>Windows XP (SP2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process create and start</td>
<td>353,000</td>
<td>1,030,000</td>
<td>719,000</td>
<td>5,380,000</td>
</tr>
<tr>
<td>Minimum kernel API call</td>
<td>91</td>
<td>878</td>
<td>437</td>
<td>627</td>
</tr>
<tr>
<td>Thread context switch</td>
<td>346</td>
<td>911</td>
<td>906</td>
<td>753</td>
</tr>
<tr>
<td>Message request/reply</td>
<td>803</td>
<td>13,300</td>
<td>5,800</td>
<td>6,340</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Singularity</th>
<th>FreeBSD</th>
<th>Linux</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read cycle counter</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>ABI call</td>
<td>87</td>
<td>878</td>
<td>437</td>
<td>627</td>
</tr>
<tr>
<td>Thread yield</td>
<td>394</td>
<td>911</td>
<td>906</td>
<td>753</td>
</tr>
<tr>
<td>2 thread wait-set ping pong</td>
<td>1,207</td>
<td>4,707</td>
<td>4,041</td>
<td>1,658</td>
</tr>
<tr>
<td>2 message ping pong</td>
<td>1,452</td>
<td>13,304</td>
<td>5,797</td>
<td>6,344</td>
</tr>
<tr>
<td>Create and start process</td>
<td>300,000</td>
<td>1,032,000</td>
<td>719,000</td>
<td>5,376,000</td>
</tr>
</tbody>
</table>

Figure 5 graphs the normalized execution time for the WebFiles benchmark in six different configurations of hardware and software isolation.

The WebFiles benchmark is an I/O intensive benchmarks based on SPECweb99. It consists of three SIPs: a client which issues random file read ...a file system, and a disk device driver.

Times are all normalized against a default Singularity configuration where all three SIPs run in the same address space and privilege level as the kernel and paging hardware is disabled as far as allowed by the processor.

The WebFiles benchmark clearly demonstrates the unsafe code tax, the overheads paid by every program running in a system built for unsafe code.

The unsafe code tax experienced by WebFiles may be worst case. Not all applications are as IPC intensive as WebFiles and few operating systems are fully isolated, hardware-protected microkernels

Singularity: Rethinking the Software Stack
With the TLB turned on and a single system-wide address space with 4KB pages, WebFiles experiences an immediate 6.3% slowdown. Moving the client SIP to a separate protection domain (still in ring 0) increases the slowdown to 18.9%. Moving the client SIP to ring 3 increases the slowdown to 33%. Finally, moving each of the three SIPS to a separate ring 3 protection domain increases the slowdown to 37.7%. By comparison, the runtime overhead for safe code is under 5% (measured by disabling generation of array bound and other checks in the compiler).
Figure 4a. Micro-kernel configuration (like MINIX 3). Dotted lines mark protection domains; dark domains are user-level light are kernel-level.

Figure 4b. Monolithic kernel and monolithic application configuration.

Figure 4c. Configuration with distinct policies for signed and unsigned code.
2.1. Sealed Process Invariants

1. The **fixed code** invariant: Code within a process does not change once the process starts execution.

2. The **state isolation** invariant: Data within a process cannot be directly accessed by other processes.

3. The **explicit communication** invariant: All communication between processes occurs through explicit mechanisms, with explicit identification of the sender and explicit receiver admission control over incoming communication.

4. The **closed API** invariant: The system’s kernel API respects the fixed code, state isolation, and explicit communication invariants.

The fixed code invariant does not limit the code in a process to a single executable file, but it does require that all code be identified before execution starts. A process cannot dynamically load code and should not generate code into its address space.
Channels

2.2 Contract-Based Channels

- All communication between SIPs in Singularity flows through *contract-based channels*.
- A channel is a bi-directional message conduit with exactly two endpoints.
- A channel provides a lossless, in-order message queue. Semantically, each endpoint has a receive queue. Sending on an endpoint enqueues a message on the other endpoint’s receive queue.
- A channel endpoint belongs to exactly one thread at a time. Only the endpoint’s owning thread can dequeue messages from its receive queue or send messages to its peer.
Extra slides from CPS 310
Memory model: the view from C

- **Globals:**
  - fixed size segment
  - Writable by user program
  - May have initial values

- **Text (instructions)**
  - fixed size segment
  - executable
  - not writable

- **Heap and stack**
  - variable size segments
  - writable
  - zero-filled on demand
Registers

- The next few slides give some pictures of the register sets for various processors.
  - x86 (IA32 and x86-64): Intel and AMD chips, MIPS
  - The details aren’t important, but there’s always an SP (stack pointer) and PC (program counter or instruction pointer: the address of the current/next instruction to execute).

- The system’s Application Binary Interface (ABI) defines conventions for use of the registers by executable code.

- Each processor core has at least one register set for use by a code stream running on that core.
  - Multi-threaded cores (“SMT”) have multiple register sets and can run multiple streams of instructions simultaneously.
The register model is machine-dependent. The compiler and linker must generate code that uses the registers correctly, conforming to conventions, so that separately compiled code modules will work together.
AL/AH/AX/EAX/RAX: Accumulator
BL/BH/BX/EBX/RBX: Base index (for use with arrays)
CL/CH/CX/ECX/RCX: Counter (for use with loops and strings)
DL/DH/DX/EDX/RDX: Extend the precision of the accumulator
SI/ESI/RSI: Source index for string operations.
DI/EDI/RDI: Destination index for string operations.
SP/ESP/RSP: Stack pointer for top address of the stack.
BP/EBP/RBP: Stack base pointer for holding the address of the current stack frame.
IP/EIP/RIP: Instruction pointer/program counter, the current instruction address.
Heap manager

Program (app or test)

 alloc "0xA" alloc "0xB" free "0xA" "ok"

Heap manager

sbrk system call

OS kernel

Stack

Dynamic data (heap/BSS)

"break"

4096

"Set break (4096)"
File abstraction

Program A

Library

open "a/b"
write ("abc")
read

Program B

Library

open "a/b"
read
write ("def")

system call trap/return

OS kernel
Reference counting

Used in various applications and programming language environments, and in the kernel, e.g., Unix file management.

- Keep a count of references to the object.
- Increment count when a new reference is created (shallow copy).
- Decrement count when a reference is destroyed.
- Free object when the count goes to zero.

[http://rypress.com/tutorials/objective-c/memory-management.html]
int j;
char* s = "hello\n";

int p() {
    j = write(1, s, 6);
    return(j);
}

The Birth of a Program (C/Ux)

myprogram.c

int j;
char* s = "hello\n";
int p() {
    j = write(1, s, 6);
    return(j);
}

myprogram.s

......
p: store this
    store that
    push jsr _write
    ret etc.

assembler

myprogram.o

object file

linker

program

libraries and other object files or archives

myprogram (executable file)

header files
Static linking with libraries

main2.c  vector.h

Translators (cpp, cc1, as)

main2.o

libvector.a

addvec.o

printf.o and any other modules called by printf.o

Static libraries

Linker (ld)

libc.a

Fully linked executable object file
The kernel

- The kernel is just a program: a collection of modules and their state.
- E.g., it may be written in C and compiled/linked a little differently.
  - E.g., linked with –static option: no dynamic libs
- At runtime, kernel code and data reside in a protected range of virtual addresses.
  - The range (kernel space) is “part of” every VAS.
  - VPN->PFN translations for kernel space are global.
    - (Details vary by machine and OS configuration)
  - Access to kernel space is denied for user programs.
  - Portions of kernel space may be non-pageable and/or direct-mapped to machine memory.
“Limited direct execution”

User code runs on a CPU core in user mode in a user space. If it tries to do anything weird, the core transitions to the kernel, which takes over.

The kernel executes a special instruction to **transition to user mode** (labeled as “u-return”), with selected values in CPU registers.
The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syscalls indirect through **syscall dispatch table** by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?

Use an integer index into a kernel table that points at the data object. The value is called a **handle** or **descriptor**. No direct pointers to kernel data from user space!

Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for read and write syscalls).
Linux x64 syscall conventions (ABI)

1. User-level applications use as integer registers for passing the sequence \%rdi, \%rsi, \%rdx, \%rcx, \%r8 and \%r9. The kernel interface uses \%rdi, \%rsi, \%rdx, \%r10, \%r8 and \%r9.

2. A system-call is done via the syscall instruction. The kernel destroys registers \%rcx and \%r11.

3. The number of the syscall has to be passed in register \%rax.

4. System-calls are limited to six arguments, no argument is passed directly on the stack.

5. Returning from the syscall, register \%rax contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is -errno.

6. Only values of class INTEGER or class MEMORY are passed to the kernel.

Illustration only: the details aren’t important.
MacOS x86-64 syscall example

section .data
hello_world   db   "Hello World!", 0x0a

section .text
global start

start:
mov rax, 0x2000004    ; System call write = 4
mov rdi, 1           ; Write to standard out = 1
mov rsi, hello_world ; The address of hello_world string
mov rdx, 14          ; The size to write
syscall               ; Invoke the kernel
mov rax, 0x2000001    ; System call number for exit = 1
mov rdi, 0           ; Exit success = 0
syscall               ; Invoke the kernel

Illustration only: this program writes “Hello World!” to standard output (fd == 1), ignores the syscall error return, and exits.

http://thexploit.com/secdev/mac-os-x-64-bit-assembly-system-calls/
Timer interrupts

The **system clock** (timer) interrupts periodically, giving control back to the kernel. The kernel can do whatever it wants, e.g., switch threads.
thread states

running

ready

blocked

sleep

yield preempt

dispatch

wakeup

stop

wait

thread states

running

ready

blocked

sleep

yield preempt

dispatch

wakeup

stop

wait
The kernel

- syscall trap/return
- fault/return

**system call layer**: files, processes, IPC, thread syscalls

**fault entry**: VM page faults, signals, etc.

**thread/CPU/core management**: sleep and ready queues

**memory management**: block/page cache

- sleep queue
- ready queue

**I/O completions**

**interrupt/return**

**timer ticks**
Process, kernel, and syscalls

```c
trap()
{
...
}

write()
{
...
}

copyout
copyin

user buffers

read()
{
...
}

syscall stub

syscall dispatch table

I/O descriptor table

I/O objects

kernel

process user space

Return to user mode

syscall dispatch table

I/O objects
Platform abstractions

- Platforms provide “building blocks”…
- …and APIs to use them.
  - Instantiate/create/allocate
  - Manipulate/configure
  - Attach/detach
  - Combine in uniform ways
  - Release/destroy

The choice of abstractions reflects a philosophy of how to build and organize software systems.
Plipes

Example: cat | cat

cat pseudocode (user mode)
while (until EOF) {
    read (0, buf, count);
    compute/transform data in buf;
    write (1, buf, count);
}

Kernel pseudocode for pipes: Producer/consumer bounded buffer

Pipe write: copy in bytes from user buffer to in-kernel pipe buffer, blocking if k-buffer is full.

Pipe read: copy bytes from pipe’s k-buffer out to u-buffer. Block while k-buffer is empty, or return EOF if empty and pipe has no writer.
How to plum the pipe?

1. P creates pipe.

2. P forks C1 and C2. Both children inherit both ends of the pipe, and stdin/stdout/stdio. Parent closes both ends of pipe after fork.

3A. C1 closes the read end of the pipe, closes its stdout, "dups" the write end onto stdout, and execs.

3B. C2 closes the write end of the pipe, closes its stdin, "dups" the read end onto stdin, and execs.