Singularity

OS: Rethinking Software Design In The Light Of Programming Language And Verification Advances

Divya Ojha
CSC456
University of Rochester

Table of contents

Motivation

Shortcomings in the existing OS design

- Most operating systems in use today are written in C, which is not a type safe or memory safe language.
- They depend on hardware like MMU for process isolation, this is expensive.
- They allow processes to share memory and concurrently write read with synchronization in order to be more efficient but allow bugs and attacks.
- There is no way for a program to declare what is does, or for the system to check what a program does.

Most conventional operating systems have not evolved to use the advancements in the areas of programming language, protection model, system abstractions, etc. **singularity** began with the goal of

- designing a software stack from scratch for improved dependability and trustworthiness by utilizing better programming languages and verification tools that are now available.

singularity has three key architectural features:

- *Software isolated processes* - this provides an environment for program execution protected from external interference
- *Contract based channels* - enable fast, verifiable message based communication between processes.
- *Manifest based programs* - define the code that runs within a SIP.

Software Isolated Processes

Process Isolation In Hardware

- Segmentation, paging have provided isolation of process memories.
- In order to map different process virtual addresses to different physical addresses, the OS depends on MMU.
- every page has access bits associated which tell what operations can be performed on that page, read or write or execute.
- Privileged operations are performed in kernel mode by raising an interrupt for transitioning to privileged mode.
- The cost of this implementation is high.

Software Isolated Processes

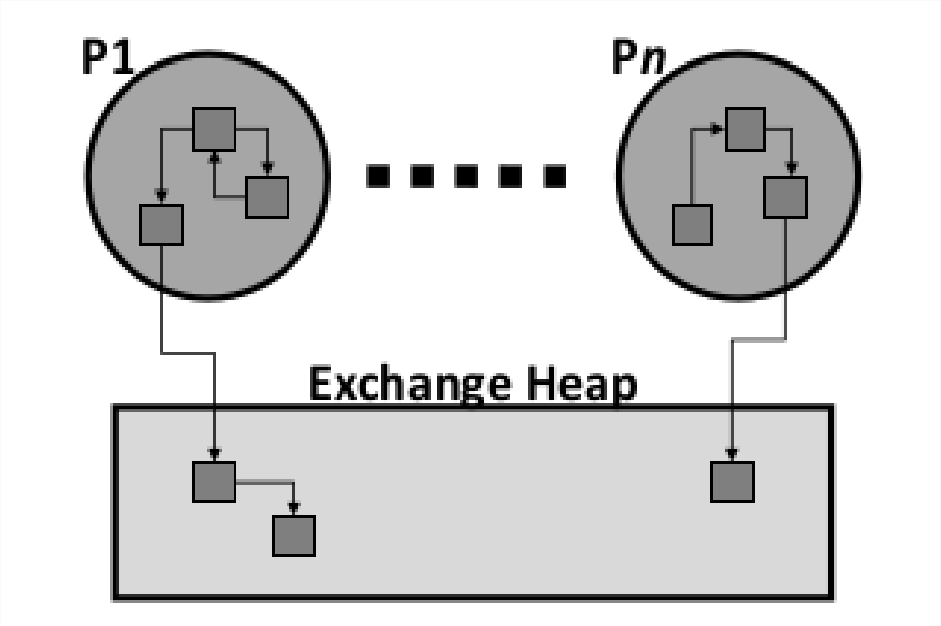
- SIP similar to processes is a holder of processing resources and provides context of execution [3][1].
- unlike conventional processes, SIP depends on programming language Sing# and its compiler to provide safe code isolation, type, memory safety. Each SIP has its own layout, garbage collector.
- SIPs are sealed code spaces, they prohibits dynamic code loading, self-modifying code, shared memory, and limits the scope of the process API [2].
- SIPs access common data over exchange heaps and communication between SIPs happens through messages and protocol for communication are specified by a channel contract and is also statically verifiable.
- SIPs access primitive kernel functions through given ABIs

Contract Based Channels

Contract Based Channels

- All communications across SIPs happen via contract based channels
- a channel is bidirectional lossless message queue with exactly two endpoints, an endpoint belongs to exactly 1 thread at a time
- the communication is described by a contract, which consists of message declarations and protocol states.
- each state specifies the possible message sequences leading to another state and message declarations have number and types of arguments and message direction.

Figure 1: Pointers in Process Heap and Exchange Heap



[3]

Contract to access network device driver

```
contract NicDevice {
out message DeviceInfo(...);
in message RegisterForEvents(NicEvents.Exp:READY c);
in message SetParameters(...);
out message InvalidParameters(...);
out message Success();
in message StartIO();
in message ConfigureIO();
in message PacketForReceive(byte[] in ExHeap p);
out message BadPacketSize(byte[] in ExHeap p, int m);
in message GetReceivedPacket();
out message ReceivedPacket(Packet * in ExHeap p);
out message NoPacket();
```

```
state START: one {
DeviceInfo! → IO_CONFIGURE_BEGIN;
}
state IO_CONFIGURE_BEGIN: one {
RegisterForEvents? →
SetParameters? → IO_CONFIGURE_ACK;
}

state IO_CONFIGURED: one {
StartIO? → IO_RUNNING;
ConfigureIO? → IO_CONFIGURE_BEGIN;
}
state IO_RUNNING: one {
PacketForReceive? → (Success! or BadPacketSize!)
→ IO_RUNNING;
GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
→ IO_RUNNING;
}}
```

- The language Sing# is an extension of C#, suited to OS communication primitives, code re-factoring, etc.
- Sing# compiler verifies that messages are not applied to wrong state.
- a separate contract verifier checks which program uses which contract.

Manifest Based Programs

Manifest Based Programs

- A manifest for a program describes required system resources, desired capabilities, dependencies etc.
- At install time the manifest is used to verify that the code meets safety property, does not interfere with previously installs MBPs.
- manifest is a machine check-able declarative expression of MBP's behaviour.
- every component is software is described by a manifest, including kernel, device drivers, applications.

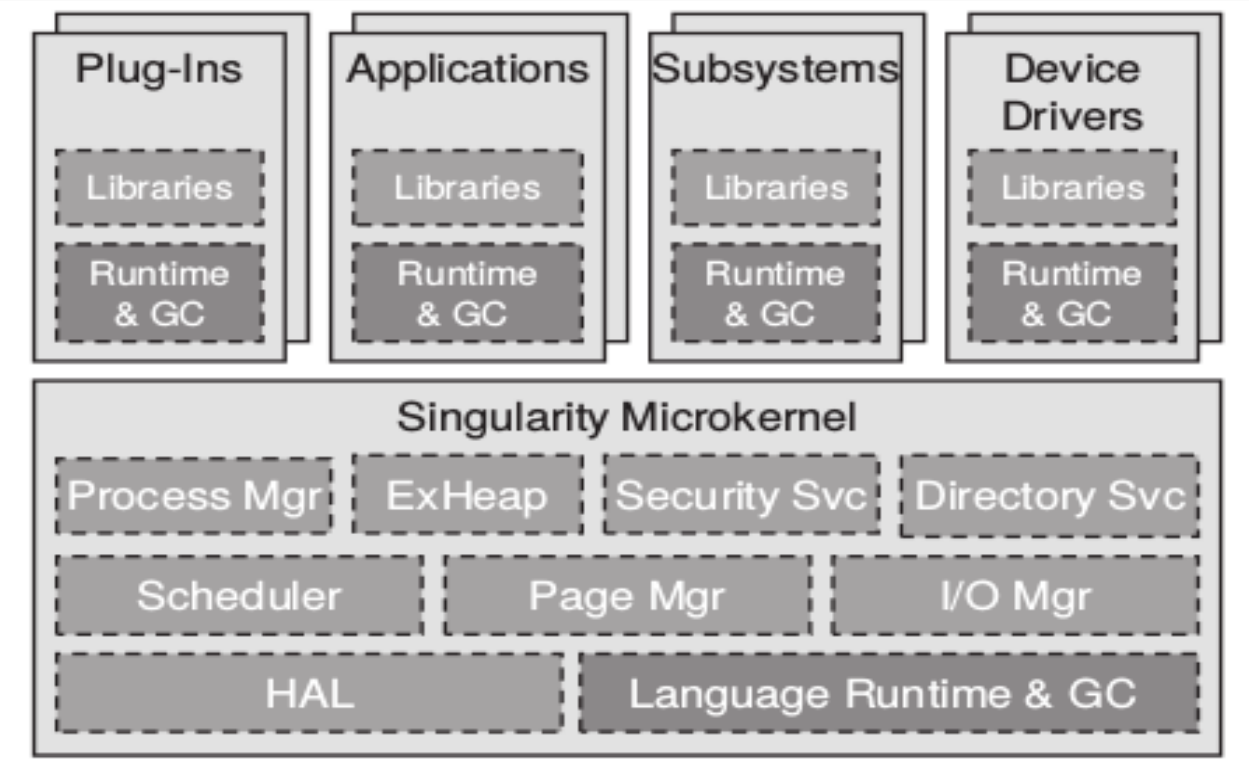
Singularity Kernel

Kernel's Role

- Kernel does resource distribution amongst competing processes. The singularity kernel is a micro-kernel, all device drivers, file-system, user application run in SIPs.. Kernel does destroying and creating SIPs and channels.
- Privileged instructions are allowed to run in SIPs to prevent illegitimate access.
- 90% of the kernel is written in Sing#, 6% in C++ and some code exists in assembly.

Singularity Kernel Architecture

Figure 2: Singularity Microkernel



[2]

Kernel's Implementation

- **ABI** - SIPs access primitive kernel facilities via ABI. higher level services are accessed via channels.
- **Memory Management** - SIPs get memory by ABI calls, has a stack per thread, can access exchange heap.
- **Threads** - all threads are kernel threads, scheduler is optimized for handling large number of threads that communicate frequently.
- **GC** - each SIP has its own GC suited to its need
- **Access Control** - each inbound channel has a single access control associated, that serves as the subject for access control decisions.

Conclusion

The benefits of tight feedback cycle among programming languages, OS architectures and verification tools can be leverages to build secure and robust systems.

References


M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus.
Deconstructing process isolation.
In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 1–10. ACM, 2006.


G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber.
Sealing os processes to improve dependability and safety.
In *ACM SIGOPS Operating Systems Review*, volume 41, pages 341–354. ACM, 2007.


G. C. Hunt and J. R. Larus.
Singularity: rethinking the software stack.
ACM SIGOPS Operating Systems Review, 41(2):37–49, 2007.