6.828: OS/Language Co-design

Adam Belay <abelay@mit.edu>

Singularity

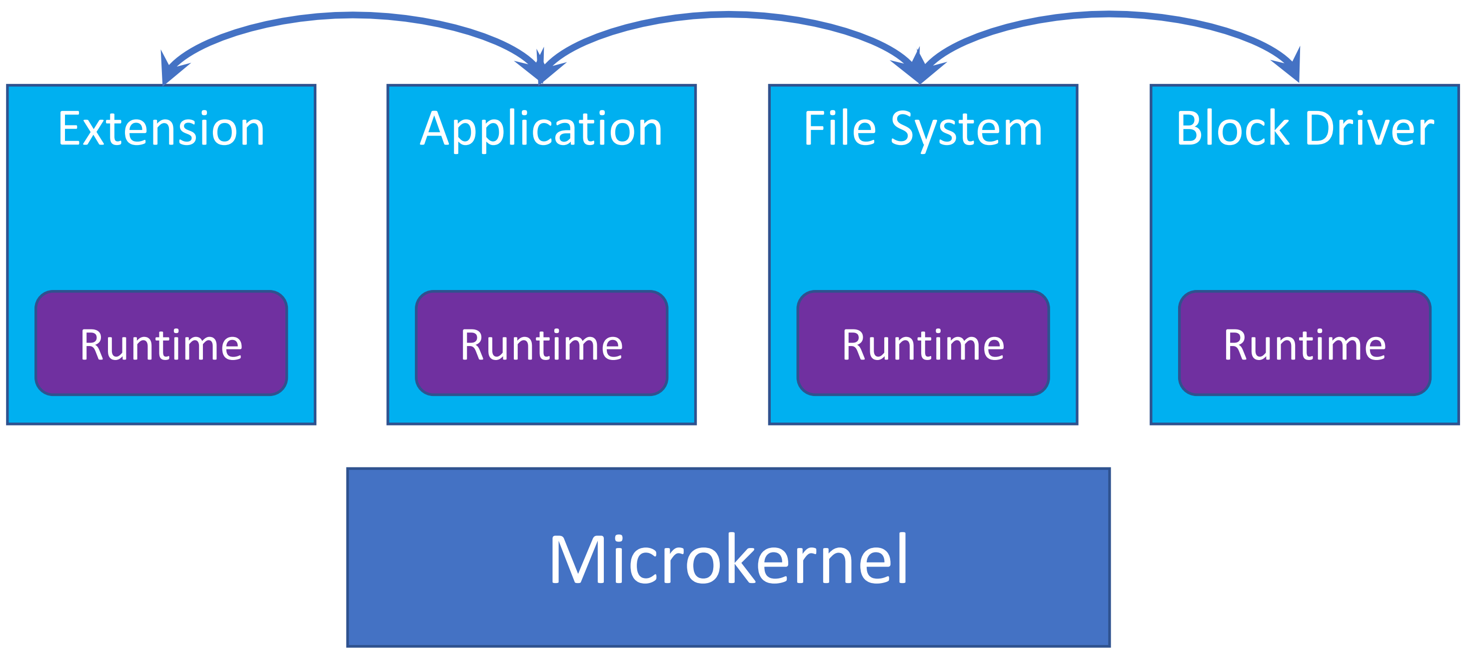
- An experimental research OS at Microsoft in the early 2000s
- Many people and papers, high profile project
- Influenced by experiences at Microsoft?
  - CLR and C#
  - Kernel stability issues from third-party device drivers

Goals

- Primary: Improved dependability and trustworthiness
  - Safe language – prevents buffer overflows, etc.
  - Program verification – detects defects at compile time
  - System architecture – prevents errors from propagating
- Modern techniques, isolation through PL
- Not a real-world system, a research testbed

High-level Structure

- Microkernel-like: kernel, processes, IPC
  - Factored OS services as userspace processes
  - UNIX compatibility is not a goal, so avoids MACH pitfalls
  - But 192 system calls!



Most radical design choice: No Paging

- Entire OS runs in a single address space
  - Both kernel and processes
- Paging HW disabled entirely, no use of segments
- User programs run in CPL 0 and can execute privileged instructions (carefully verified)

Why turn off paging?

- Performance?
- Faster process switching, no page root switch
- Faster system calls, CALL not INT
- Faster IPC, no copying needed
- Device drivers can access hardware directly
- Benefits shown in benchmarks

- But main goal isn’t performance (recall dependability and trustworthiness)

Is turning off paging consistent with robustness goal?

- A lot of unreliability comes from extensions
  - E.g. kernel modules, browser plugins, etc.
  - And those already loaded into host program's address space for convenience and performance
- So maybe VM HW is already not relevant
- Can we do without it?
- Later, the paper mentions optional support for VM

Extensions in Singularity

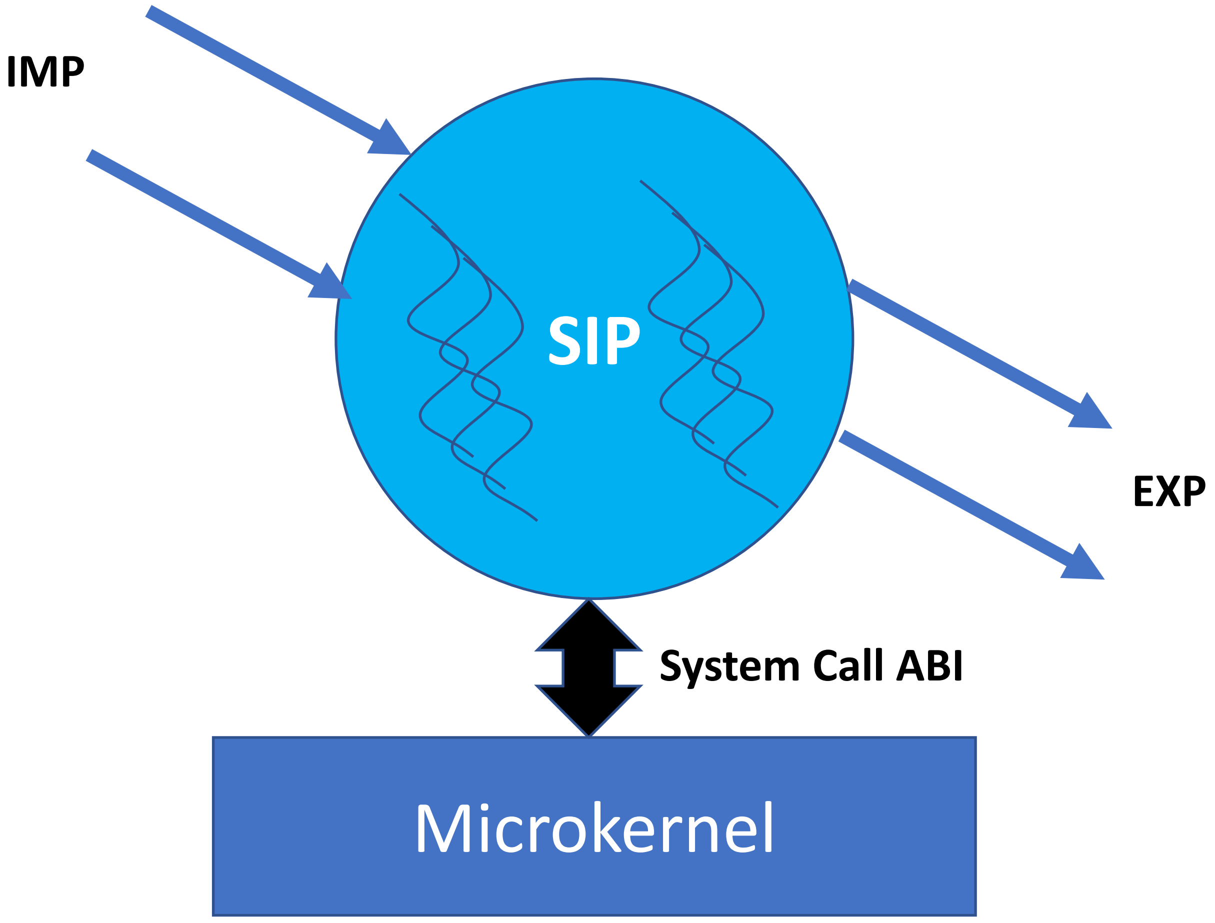
- Separate process, communication through IPC
- But lightweight, IPC overheads not burdened by address space switches

Key Concept: Software Isolated Processes (SIPs)

Each SIP is “sealed”

- No modifications from outside
  - E.g. JOS system calls that take an envid are not allowed
  - Could there be a debugger?
  - Only IPC for communication
- No modifications from inside
  - No JIT
  - No class loader
  - No dynamically-loaded libraries

SIP Communication



Rules governing SIPs

- Only pointers to own data
  - No pointers to other SIP data, no pointers to kernel
  - No sharing despite shared address space!
  - Exception: Exchange heap
- SIP can ask for pages from the kernel
  - May not be contiguous

Why can’t SIPs be modified?

Why can’t SIPs be modified?

- What are the benefits?
  - No code insertion attacks
  - Easier to reason about correctness?
  - Probably better optimization?
  - Inline? Delete unused functions?
  - Easier to verify?
- Is it worth the pain?

Why not use a single, shared runtime?

Why not use a single, shared runtime?

- IPC is the only interaction between SIPs
- More robust, enables fault isolation
- More customizable
  - Each SIP can have its own language run-time, GC scheme, etc.
  - But the runtime is a trusted component!
  - Better not have bugs!
- SIPs make it easier to clean up after kill or exit

How to keep SIPs isolated?

- Only read/write memory the kernel has given you
- Could the compiler check each access?
  - ”Does this point to memory the kernel gave us?”
  - Really slow, especially because memory is noncontiguous
  - Compiler isn’t trusted

Singularity uses PL-based protection

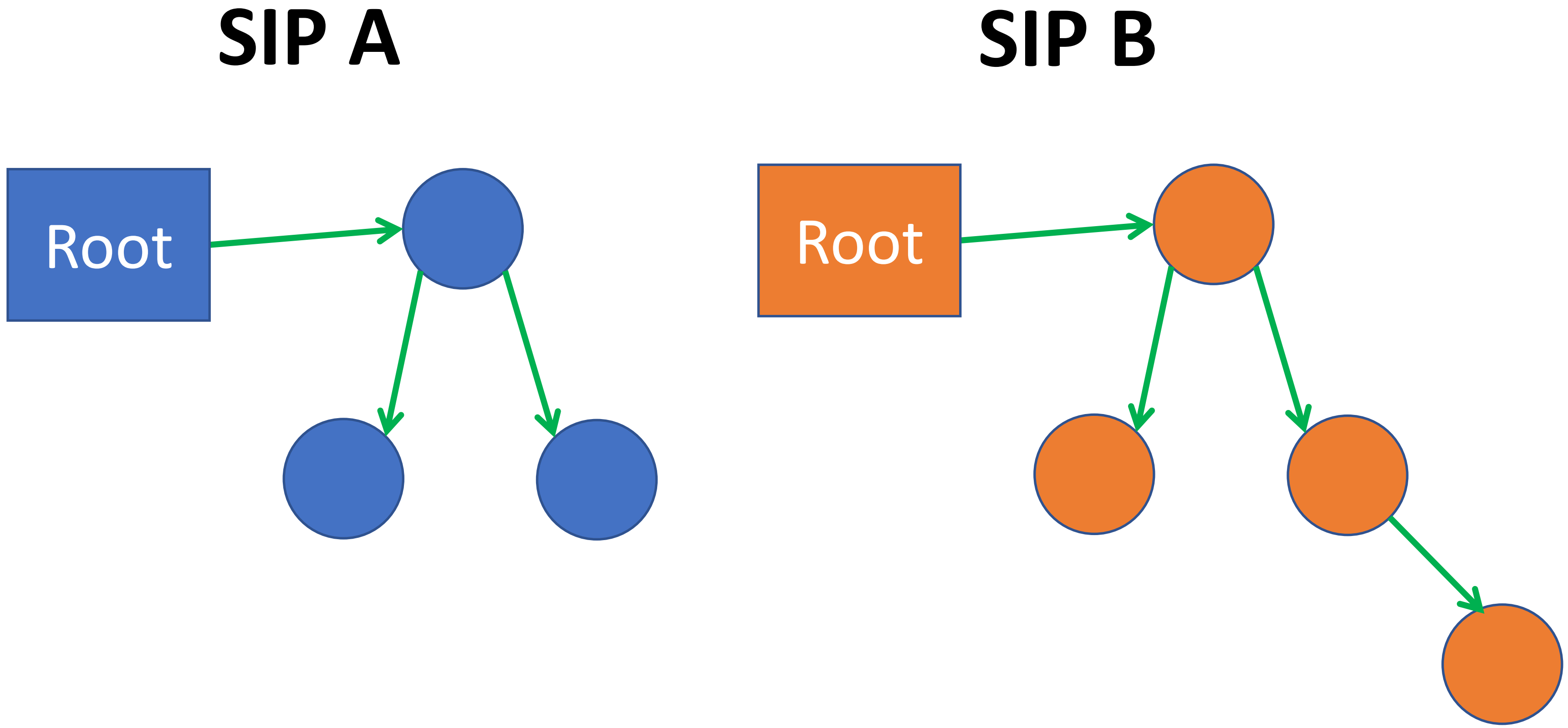
Overall plan:

1. Compile code to bytecode
2. Verify bytecode during install
3. Compile verified bytecode to machine code
4. Run machine code with trusted runtime

How does bytecode verification work?

- Does it check for “only access memory kernel gave us?”
  - Not exactly, but related!
- Plan: Each SIP can only access reachable pointers
- Only the trusted runtime can “give” new pointers
- So if kernel/runtime never supply incorrect pointers, each SIP can only access its own memory

Reachable pointer diagram



What does the verifier check?

1. Don't cook up pointers (only use pointers runtime/kernel gives you)
2. Don’t allow casts to pointers
   - E.g. int to pointer would violate rule #1
3. Don't use after free
   - Otherwise, could be used to violate rule #2
   - GC and transfer heap help guarantee this
4. Don't use uninitialized variables
   - Zero allocated memory

In general, don’t trick the verifier.

Example

```
 R0 <- new SomeClass
 jmp L1
…
 R0 <- 1000
 jmp L1
…
L1:
 mov (R0) -> R1
```

Example (continued)

- Verifier tries to deduce the type of every register
  - Pretends to execute along each code path
  - Requires that all paths to a register use result in same type
  - Check that all reg use okay for type
- In this case, R0 has type int or type *SomeClass
  - Validator would say no!

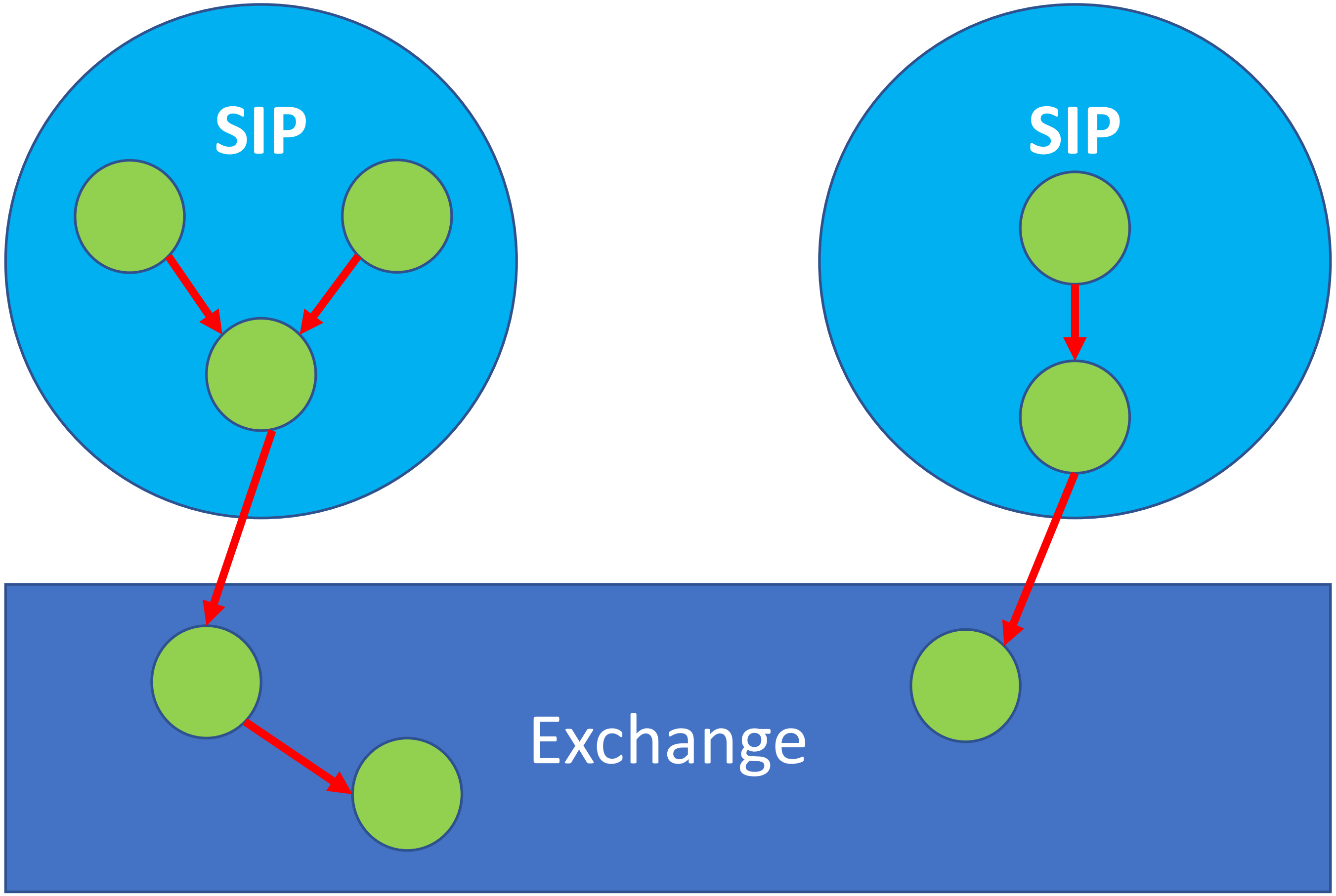
Bytecode verification may be stricter than needed

- E.g. It’s might be okay to cast pointers that are still within the SIP’s memory
- Benefits of verification:
  - Faster execution, may be able to elide runtime check!
  - Type check IPC channels
  - Need to allow R/W of exchange heap but not SIP’s memory
  - Do system calls run on SIP’s stack?
    - If so, could prevent another SIP thread from wrecking stack memory
- An interpreter could evade ban on self modifying code

Key concept: Exchange heaps

- Shared memory communication
- Message bodies are stored in exchange heap
- Possible dangers:
  - Send wrong type of data
  - Modify a sent message while it's in use by receiver
  - Modify an unrelated message
  - Use up all exchange heap memory and never free

Exchange heap diagram



How to prevent abuse

- Verifiers only allows bytecode to keep a single pointer to items in the exchange heap (i.e. linearity)
- A SIP must relinquish a pointer when it send()s it
- Verifier knows when last reference is dropped
  - e.g. send(), delete(), or pointer from another object on exchange heap
- Single pointer rule prevents modify-after-send and makes reference counting easy
- Runtime can maintain a datastructure that maps objects to the SIPs that own them
  - Why is this useful?

What about channel contracts?

- Nice to have or does singularity rely on them?
- Type signatures are clearly important
  - Verifier must check they match
- State machine guarantees finite queues
- State machine guarantees each send operation is paired with a receive
  - Queue sizes can be bounded and send doesn't block
- Receive can block, send must perform the wakeup

How do system calls work?

- No INT instruction, instead just CALL
- Or inline kernel code directly into SIP (e.g. channel send and receive)
- Kernel uses same stack as SIP
  - Stacks can grow dynamically so no size issue
  - Must delineate kernel part and user part of stack frame, otherwise how to garbage collect, etc.?

Other concepts

- Manifest files
  - Describe every detail of an application, including verified code, initial channels at startup, etc.
- Channels
  - Are like capabilities, a channel can be passed through a channel
  - Can only communicate with a SIP if you have the right channel

Does evaluation support claims?

- Robustness?
- Good model for extensions?
- Performance?