Singularity Lecture Notes

Last lecture, we talked about a few differents kind of architectures.

First, the monolithic design:

* Protection and isolation provided by hardware.
  * Protection: Rings
  * Isolation: MMU
* Kernel <--> User communication happens via syscalls
* Kernel <--> Driver communication happens via function calls
* Driver <--> User communication happens via syscalls
* User <--> User communication happens via IPC (pipes, sockets)
  * Can also share memory, but less commonly used.

```
      RING 0                                    RING 3
 -------------------            ---------------------------------------
|                  |            |                                      |
|                  |            |  |---------|           |---------|   |
|     KERNEL       |            |  |         |           |         |   |
|     DRIVER       | <-syscall->|  | PROCESS | <--IPC--> | PROCESS |   |
|     DRIVER       |            |  |---------|           |---------|   |
|                  |            |                                      |
 -------------------            ---------------------------------------
```

Advantages:

* Tends to be fast!

Disadvantages:

* Kernel implements a lot of driver interfaces.
* Driver crashes = machine crashes.
* A lot of trusted code => exploits easier to happen.

Now, the microkernel design:

* Protection and isolation provided by hardware.
  * Protection: Rings
  * Isolation: MMU
* Two "types" of software, essentially:
  * Kernel code: kernel, IPC services
  * User code: drivers, processes, threads
* Kernel <--> User communication happens via syscalls
* Kernel <--> Driver communication happens via syscalls
* Driver <--> User communication happens via IPC
* User <--> User communication happens via IPC

```
      RING 0                                    RING 3
 -------------------            ---------------------------------------
|                  |            |                                      |
|                  |            |  |---------|           |---------|   |
|      KERNEL      | <-syscall->|  | PROCESS | <--IPC--> | PROCESS |   |
|                  |            |  |---------|           |---------|   |
|                  |            |    DRIVER                  USER      |
 -------------------            ---------------------------------------
```

Advantages:

* Kernel is much, much smaller: no driver code, minimal driver interface.
* Drivers are now untrusted! Less exploits, hopefully.
* Driver crash != kernel crash.

Disadvantages:

* IPC tends to be slow.
* Drivers now need to syscall to get into kernel.

Singularity decides enough is enough. Wants to reimagine the _entire_ stack:

```
  1) New Programming Language: Sing#
    * based on C#
    * + linear types, channels, contracts, unsafe, linked stacks, macros, more?
  2) New Architecture:
    * Microkernels with no hardware protection or isolation
    * Processes can ONLY do IPC over contract-based channel with other processes

                            RING 0
    ------------------------------------------------------------|
    |                                                           |
    |                     |---------|               |---------| |
    |   KERNEL <--f()-->  | PROCESS | <--channel--> | PROCESS | |
    |                     |---------|               |---------| |
    |                      DRIVER                      USER     |
    ------------------------------------------------------------|

  3) New way to launch programs
    * Manifest describes the program: resources, dependencies, and capabilities
    * Programs aren’t fully precompiled: MSIL (Microsoft Intermediate Language)
  4) Verification tools
    * Can verify "contracts" at bytecode level. Neat.

Q: Who are these guys?
A: Galen Hunt has done a ton of research on operating systems at Microsoft.
   Worked on Haven, Drawbridge, and Singularity.

Three core Singularity Contributions:

  1) Software-Isolated Processes
  2) Contract-Based Channels
  3) Manifest-Based Programs

All of these are enabled by their new language, sing#.

--------------------------------------------------------------------------------

# Let’s talk about Sing#.

1) Garbage Collected
  Just like Java, Go, C#, etc.

2) Has "unsafe" abilities:
  Can somehow bypass type system and memory safety, though not clear how.

3) Linear Types: important to understanding contracts/channels. (CRITICAL)

Core idea:
  An object can be actively referred to by one name at any point in time.
  This property is statically guaranteed by the type system.

Example (filesystem, sip_1, and sip_2 are channels):
  file := <-filesystem
  sip_1 <- file
  sip_2 <- file // ERROR! use of `moved` value: file was moved to `sip_1`

--------------------------------------------------------------------------------

# Software Isolated Processes are...

Written (almost) entirely in Sing#. So:
  * type safe: no need for protection (rings for protected execution) from h/w
    * function call to talk to kernel
  * memory safe: no need for memory isolation (MMU) from hardware
    * can only get valid pointers. no pointer arithmetic, of course
```

    * context switching is basically free: like user-level threads

They're sealed:
  * Code is locked down after install. Can fingerprint by code.
  * Cannot share memory with other SIPs: only channels.
  * Two SIPs protected from eachother through language-based memory isolation.

Communication by channels is protected by contract
  * Specifies the exactly protocol for communication.
  * Contracts cannot be broken by SIPs: checked by compiler at install-time.

Described and launched by manifests
  * Define code that runs within SIP and specifies verifiable properties

# Let's go through an example:

We want to launch a new SIP that sends a packet. We do this for a while. How?
What happens?

First, we need a manifest for the SIP.
  - Define code that runs within SIP and specifies verifiable properties
  - No code is allowed to run on Singularity without a manifest
    - In fact, must start program using a manifest
  1) Manifest describes the following:
    - code resources: our code
    - required system resources: need memory?
    - desired capabilities: want to send packet
    - dependencies on other programs: NIC server
  2) Manifest is machine checkable and used to verify security properties
    - Driver's manifest details the hardware it will use
      - At install time, can check that now two drivers will conflict
    - Can also be used to verify type and memory safety
      - verify that privileged instructions aren't used
      - conformance to channel contracts
      - How does this actually work!?
        - The MBP is literally MSIL: Microsoft Intermediate Language
          - Well, an extended version with Singularity specific metadata
          - Accepted by CLR: Common Language Runtime
          - Manifest defines its structure and characteristics
        - The MSIL is compiled at install time
          - How long does this take? Probably much slower than loading ELF.

We have our manifest. We ask the Singularity Kernel to execute it. How?
  - presumably some init() system launched a shell
  - we invoke our manifest in the shell
  - the shell executes an ABI call to the kernel to create child process

What's an ABI call?
  - literally just function calls
  - ABI is not for higher-level services, like files/net, which use channels
  - ABI maintains system-wide state isolation invariant
    - SIP cannot modify another SIPs state via ABI call
    - Cannot obtain reference to other SIPs state via ABI
    - Allows two SIPs to run completely independently of each-other
  - but we need to do something special when making ABI call, etc.
    - we'll come back when talking about garbage collection

Kernel reads manifest, installs program.
  - Verify manifest properties:
    1) code resources exist
    2) required system resources are available
    3) capabilities can be given to this process
    4) dependencies on other programs can be met
  - Type check and compile MSIL

- Get some memory, load it up, initialize, and start executing main().

Our SIP finally executes. We want to talk to network now. How?
- Need a channel endpoint to talk to NIC and its code.
  Q: Where does it come from?
  A: OS gave it us on initialization. Declared in manifest.
- Q: Got the endpoint. What do we send?
  A: Need to abide by the channel contract.

Channel Contracts:
  NOTE: Go through example in listing 1.
  NOTE: Nothing there for sending packets. Invent it with pointer to packet.

```
  out message SendPacket(Packet * in ExHeap p);

  state IO_RUNNING: ... {
    ...
    SendPacket? -> (Success! or Error!) -> IO_RUNNING
  }
```

Now we understand contract, want to send packet.

  0) Go through network initializaion based on contract.
  1) Create a buffer somehow, fill it with content.
  2) Call SendPacket?, pass in pointer.

    Say Packet is:

```
      struct Packet {
        byte[] contents in ExHeap p
      }
```

    Code would look like (fill this in as we go):

```
      fn main() {
        // init `network` -> IO_RUNNING

        loop {
          packet := new(Packet in ExHeap)
          fill(packet.byte with contents)
          network <- SendPacket(packet)

          status := <-network
          return if status.is_ok()
        }
      }
```

    Q: Where does packet pointer go?
    A: Into exchange heap. NOTE: Explain Memory Isolation Invariant.
      Address space is partitioned into three logical spaces:
        1) The kernel object space
        2) The SIP object space
          - All pointers in a SIP point to own object space or exchange heap
            - NEVER to another SIP's address space
            - Again to be able to GC SIPs independently
        3) An exchange heap for communication of channel data
          - SIP has access to exchange heap
          - All pointers in exchange heap point to objects in exchange heap
          - Every block of memory in e-heap is accessible by one SIP at a time

      (SIPs can only point to memory inside SIP or exchange heap.)
      (Exchange heap can only point into exchange heap.)
      (Each object in exchange heap has at most one pointer from SIP.)

    Q: What needs to be true about the pointer `byte` inside?

A: Itself and contents need to be in exchange heap. So caller must have allocated the byte[] and Packet in exchange heap. Enforced via types.

Q: How is the message actually sent? Authors says "zero-copy".
A: A write of the pointer is made directly to the other endpoint.

Q: Won't we need to allocate space for the pointer sometimes? IE, if the queue is full?
A: No. Singulartity guarantees statically that there is enough space. So the queue size is determined and allocated statically. How? Enforce that each cycle in state transitions contains at least one rx and tx. Thus, can't send unbounded amount of data...will need to wait and pull off queue to do opposite of rx/tx.

3) NIC was waiting for message. Received, processed, send message back.
* When receive message, got pointer to packet.
Q: But we can only have one reference to object in exchange heap. How do we prevent two?
A: Linear types.

4) Wait for message back from NIC. Done.

We keep doing this for a while and generate garbage: NIC, Exchange Heap, Client
* Garbage collector comes along.
Q: How do garbage collectors generally work?
A: They find all objects that are reachable by scanning stack for roots, then free all objects that are _not_ reachable.
Q: Why can garbage collector collect NIC and Client individually?
A: No pointers into each other. Impossible for one to hold a reference into another.
Q: What about the exchange heap?
A: Objects are references counted in there. Reference count goes to 0 == deallocate.
Q: What if SIP crashes? What happens to owned items in exchange heap?
Q: Can we just scan the stack/heap of crashed SIP?
A: No. Can have dangling pointers as long as they're not accessible.
A: Each block knows who its owner is. Or there's a table mapping blocks to owners. When SIP crashes, just free everything belonging to it.
Q: Since ABI calls are just function calls, won't there potentially be pointers to kernel structures on a SIPs stack?
A: Yes. So garabge collector must know where one stack ends and another begins. Special structure inserted on ABI calls delimits the two.

Say we do a bunch of function calls in our SIP. Can we exhaust our stack?
* No, effectively. Have linked/segmented stacks. Kind of like MESA.

Q: Why not verify at compile time?
A: If the verification happened only at compile time, then the compiler would have to be trusted.