Singularity Design Note 18

I/O Subsystem Fundamentals

Garbage in, garbage out

A closed black box isn't very interesting. Singularity will need mechanisms for input and output. Planned development occasionally results in superior results to unplanned development; thus having a plan could arguably be beneficial. This document is a humble attempt at coming up with a plan.

1. Primary Research Goals

Our preliminary discussions revealed many potential areas for research which lie along our quest for an ideal I/O subsystem. In subsequent discussions we whittled this down by concentrating on those ideas which complement Singularity's overall goals. What follows is a list of issues we intend to tackle. Discussion is deliberately kept to a high-level, although we mention mechanism when use of the mechanism is an inherent part of the goal. The real pesky details are left to later sections of this or other documents.

It should be noted that several of our goals could be lumped under the broader term of "reliable". Since reliability is an overall goal for Singularity

1.1. Run-time Fault Isolation

We want to limit the negative effects of run-time faults in device drivers. To accomplish this, we propose to isolate higher-level device drivers from the kernel (and each other) by placing them in separate processes. To further limit the potential damage a driver can cause, we will require device drivers to be written in managed code. Device drivers will not provide arbitrary device-specific code to the kernel for interrupt handling. However, device drivers will be able to register for interrupt notification messages (which they will receive over standard Channels).

In most respects, device driver processes will be like any other process in Singularity. Most communication external to a driver's process will be performed via Channels. However, driver processes for physical devices will have privileged access to *device-specific* physical address space (for memory mapped I/O) and the ability to perform port I/O.

Note that an errant or malicious driver of a DMA-capable device could potentially alter kernel memory or the memory of other processes by programming the external DMA engine to write into inappropriate regions of physical memory. We plan to explore limiting the regions of physical memory accessible by external agents on those architectures which support this feature.

1.2. Modular Architecture

We want Singularity to be extensible in a clean and predictable manner. To this end, the I/O subsystem will be comprised of an extensible number of (reconfigurable) components. A "device driver" is one or more of these components. Components will be stackable, and connections may be interposed by other components. The

interfaces to these components will be well-defined and visible prior to runtime (i.e. expressed statically, not in code).

Components will be tolerant to the failure of other components. Every component should be able to handle gracefully the disappearance/unresponsiveness of any other component (and possibly switch to using an alternative when and if one should appear). In particular, no component should fail catastrophically as a result of another component crashing.

It should be possible to dynamically start, stop, unload, reload, and restart components. This capability is needed not only to support plug-and-play devices, but is also extremely handy for debugging.

The I/O subsystem will provide some sort of management mechanism for starting appropriate components (drivers) at boot/device-insertion time, connecting components together in appropriate ways, and tracking configuration changes. This mechanism will also monitor components for crashes and possibly other forms of misbehavior. It will notify the appropriate parties of problems, as well as facilitate the reloading and restarting of crashed components.

1.3. Unified Handling of Asynchronous Events

The basic idea behind this goal is the observation that legacy I/O systems are host-driven, and that previous attempts to add remote-initiated communication or other forms of asynchronous event handling has resulted in inconsistent models of representing these things in the system. For example, a new network connection is traditionally treated quite differently than the insertion of a new plug-n-play device, although both represent the creation of a new communication endpoint of some sort. We propose to define a single model for event handling and notification which will be pervasive throughout the I/O system.

2. Other Goals

There are several other goals, which while admirable and important, didn't make it into the above section.

2.1. Help for Device Driver Writers

We want to make it easy for device driver writers to write good drivers. Cynically speaking, this will likely mean writing the drivers for them – preparing generic miniports which require only a small amount of device-specific code to operate.

2.2. Schedulable

One of Singularity's goals is to make I/O resources schedulable. Most likely this will involve reservations of bandwidth (to disk, over network, etc). Hopefully the scheduler folks have some ideas here.

2.3. Extra Reliable Content

Most traditional I/O subsystems do not perform any extra content correctness guarantees beyond that provided by the underlying hardware. Some discussion was had on the topic of adding extra checksums/hashes to disk storage blocks, for example.

3. Singularity Assumptions

We make the following assumptions about Singularity:

- Singularity will not support swapping/paging.
    - Avoids reliability problem of losing executable code for a running process due to a faulty disk driver (or broken network connection, if loading from a remote filesystem).
    - Avoids whole class of bugs.
- The transfer of a large block of data over a Channel will not result in a physical memory copy, but rather a transfer of memory ownership/access from the sending process to the receiving one.
    - Likely needed for reasonable I/O performance.
- System namespace will support non-persistent, as well as persistent entries.

  - Allows for devices to be represented in the namespace.
- System namespace will be dynamically extendable by other system components.
  - I/O subsystem may add entries for plug and play devices upon insertion.
  - Network stacks may add entries for connections upon establishment.

4. Interrupt Handling

Interrupts are non-schedulable intrusions into normal control flow. Therefore, in order to minimize their potential negative effects upon the system, we want to limit them to performing as little work as possible.

At least initially, we plan on having the kernel contain an interrupt service routine which merely squelches the interrupt at the interrupt controller (PIC or APIC) and arranges for a message to be sent to the driver (or drivers, in the case of shared interrupts) in question. There won't be any driver-specific ISR routines in the kernel.

As with other hardware resources, drivers will be allocated/reserved interrupts by the I/O management mechanism.

Some extremely low-level devices, like the timer, will be exceptions to the general rule and have their device drivers implemented in the kernel.

5. API Issues

Communication between I/O components (such as device drivers) and their clients will be conducted over Channels. Therefore, the device driver "API" will be the set of messages sent over such Channels.

5.1. Grand Unified Theory

We believe that all kinds of I/O (i.e. from/to devices, files and network connections) should be performed in the same way (or at least to the greatest extent possible). That is to say, there should be a common (likely extensible) set of messages for interacting with all types of I/O entities. This has many advantages, including ease of code reuse, interconnection, interposition, tunneling, etc. It also encourages the development of generalized tools and applications.

That said, we also want to include a generic extension mechanism to handle specialized devices – the challenge here will be to come up with the right level of uniformity across devices while still providing access to unique functionality. One possibility is to have a set of messages that devices may implement, some of which will be required for all devices, others of which will be optional. In all cases, however, the messages types will be pre-defined and the interface requirements for a given driver will be statically known (i.e. visible prior to runtime, not expressed in code).

5.1.1. Some Things Are More Equal Than Others

The Grand Unified Theory implies a single abstraction for all I/O entities (e.g. devices, files and network connections). However, some I/O entities are also collections of other entities (e.g. controllers, directories and the missing network analogy). The question arises of what abstraction to use for such collections. We observe that collections of entities are namespace entities – they affect the system namespace by adding more items to it. Rather than treat these things differently, we instead propose that they are like other I/O entities yet also are capable of extending the namespace.

5.1.2. Improved Network Integration

Existing network APIs (such as the ubiquitous Berkeley Sockets API) were mostly designed as extensions to traditional (i.e. pre-network) I/O systems. To achieve this, abstractions were often twisted in somewhat demented ways in order to shoehorn networking into the system. In particular, traditional I/O is primarily driven by a host (a host-master/device-slave perspective), whereas network I/O can be driven by either peer (requiring a more event-handling perspective). We believe we will be able to develop better abstractions for handling both host/device and peer-to-peer I/O in a consistent manner.

5.2. Naming

The I/O API is highly intertwined with the namespace used to identify I/O entities. Therefore we spend some time in this design note discussing the system namespace. Despite all evidence to the contrary, this is not an attempt to usurp other people working on the namespace design.

We propose that there be an extensible, system-wide namespace which would include all types of I/O entities. Any entity which collects/enumerates/references other entities (e.g. directories, device controllers, etc) would have the means to extend the namespace to include these other entities.

Entities in the namespace may or may not be persistent (i.e. the namespace should accommodate ephemeral things, such as network connections). Non-persistent entities may also exist outside of a namespace (i.e. it should be possible to have open files which are not named in any filesystem).

We would like the name format to handle network addresses in their native state, i.e. binary rather than text. The ability to perform logical operations (such as AND, OR, etc) upon names would be very powerful. For this reason, and also in keeping with our generic interface bias, we propose that the names in the namespace be binary blobs (length, byte array pairs). These would obviously be translated into human-readable form for presentation to the user.

An important aspect of keeping a common namespace for all I/O entities is that we will be able to handle more things in a generic fashion. For example, the insertion of a plug-and-play device is logically very similar to the establishment of a network connection. We hope to allow the system to treat these in a similar fashion.

5.3. Channel Interface

Data received from an I/O system component will appear in a new (previously unallocated) area of the receiving process's address space. Likewise, the address space holding any data sent to an I/O component will become inaccessible to the sending process after it is sent. This allows us to support so-called "zero-copy" semantics. The channel implementation can thus "take" the physical memory from the sending process and "give" it to the receiving process, avoiding a memory copy. Hopefully, our implementation of this will be fast.

6. Device Manager

This concept is mostly still TBD.

Physical system resources (memory regions for memory-mapped I/O, Ports, DMA channels, IRQs) will need to be managed by something. This thing would be responsible for allocating resources to I/O system components for their exclusive use.

Likewise, the layering of I/O system components (filesystems on top of disk partitions, network protocols on top of adapters, etc) will need to be managed by something. This thing would be responsible for arranging the connections between appropriate components, etc. It would likely also help recover functionally upon driver crashes by restarting drivers and/or re-routing connections.

We also want an entity to monitor device state and provide a generic interface to clients which need dynamic information about the machine's hardware capabilities (i.e. notify when a camera or portable music player is plugged in).

7. Random Musings

This section contains initial thoughts about things which are mostly still to be determined.

- TBD: access control of physical resources (for device drivers).
    - Dynamic? Driver processes may request access at any time?
    - Static? Access granted at driver process creation?
    - In either case, access controlled via pre-established policy.
- TBD: Provide for custom Interrupt Service Routines (ISRs)? Should only squelch their interrupt and signal their driver. Since these are non-schedulable, they need to be as minimal as possible.

    - MSIL: Vetted by system and hooked to interrupt at device start.
        - Where does code live?
    - Custom mini-language: Provides a small number of commands like read register, write register, various bit twiddlers, signal event, etc. Registered with kernel interrupt handler at device start. Common code runs these at interrupt time.

- No attempt to hide device/driver failure from clients – would require too much competence on behalf of device driver writers. Applications and other clients should contain appropriate retry code.
- Networking
    - Stacks based around layering, with each protocol (or de-multiplexing point) its own process.
        - Separation of layers is desired for extensibility.
        - Note that Channels between protocols can be used for queuing (when handling multiple packets at once), or they can provide direct hand-off of the processor (when handling a single packet). This is run-time tunable.
    - Multiple instances of a stack may occur above a de-multiplexing point.
        - For example, a machine with multiple IP addresses may assign these to different instances of the TCP implementation (or even different TCP implementations).
        - These instances may be scheduled differently.
    - Data passed between layers is tagged in some standardized fashion.
- Outstanding questions
    - Should Singularity provide data correctness guarantees?
        - Network stacks already don't trust the network not to mangle packets.
        - Yet storage subsystems trust the disks/controllers not to mangle blocks.
    - Should Singularity I/O have some sort of transactional guarantees? What would these look like?

