May 2023

A tour of the Fuchsia system interface

Questions: @

Link to the presentation

The system interface (syscalls) are the building blocks of an operating system.

It’s like the alphabet of a language.

Fuchsia's system interface is different from Mac/Linux/Windows.

This talk is about understanding Fuchsia's alphabet, so we can make some sentences.

Each language implies a worldview, so does each system interface.

00

Actors

Tasks

Fuchsia has the regular execution abstractions:

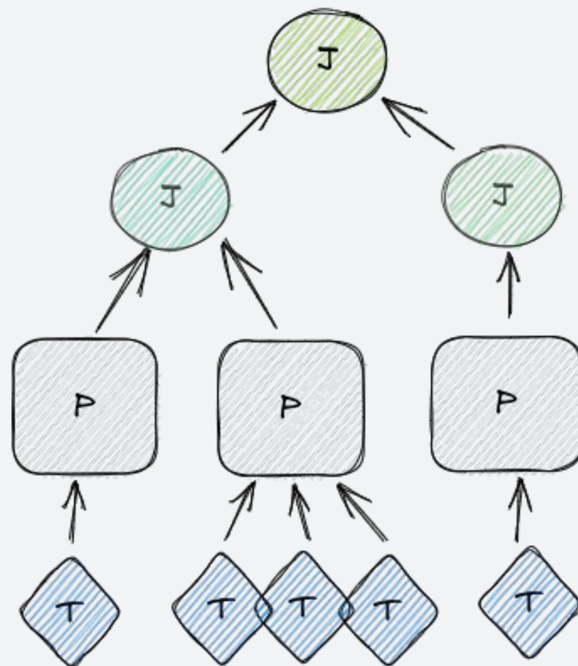
Threads: unit of execution

Processes: address space + resources + threads

Jobs: group of processes

They work as you expect them, which each process having an its own isolated address space.

The system scheduler multiplexes threads over available cores.



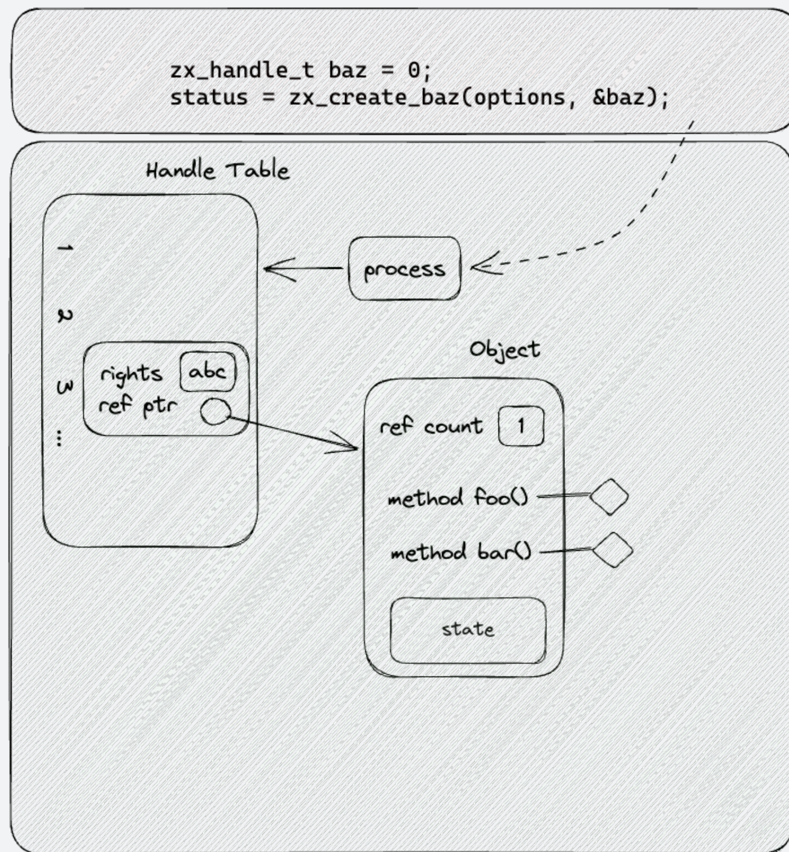
Thread, Processes and Jobs

Objects

Fuchsia has a rich object-based system interface

Each different object specializes a resource:

- Memory
- Processor time
- I/O
- Exceptions
- .....



Threads create these objects, the process holds them

Applications refer them as “**handles**”: just an **uint32_t**’s

The object type needs to be tracked by the application code.

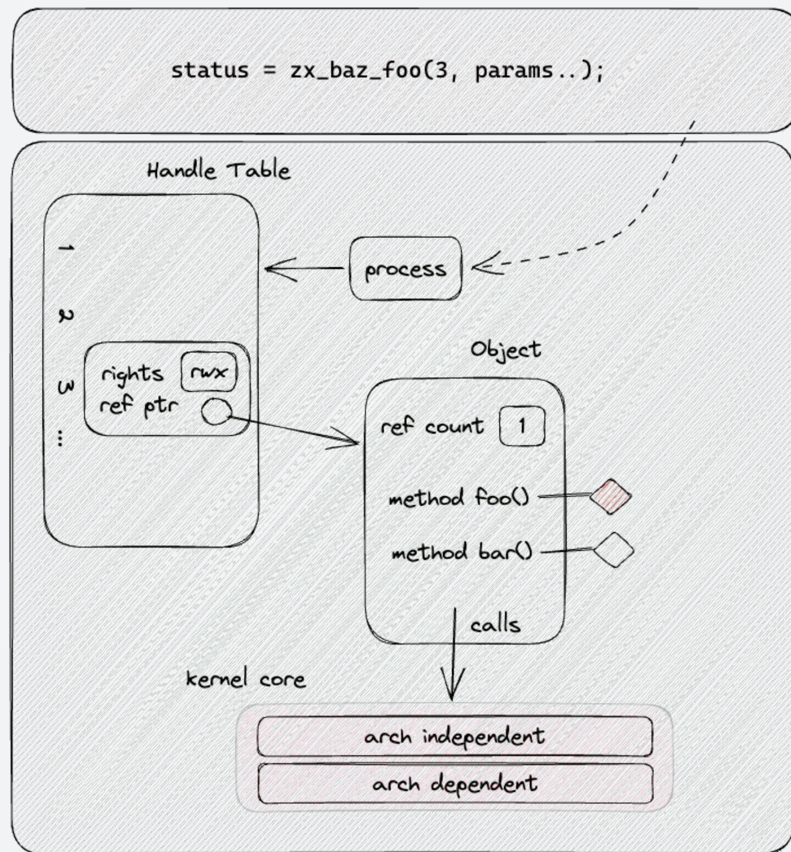
kernel object creation

Using objects

Syscall input

- Service index → object method
- Thread context → process
- Arguments in registers
  - handle value

Does it remind you of file descriptors?



kernel object use

Handle duplication

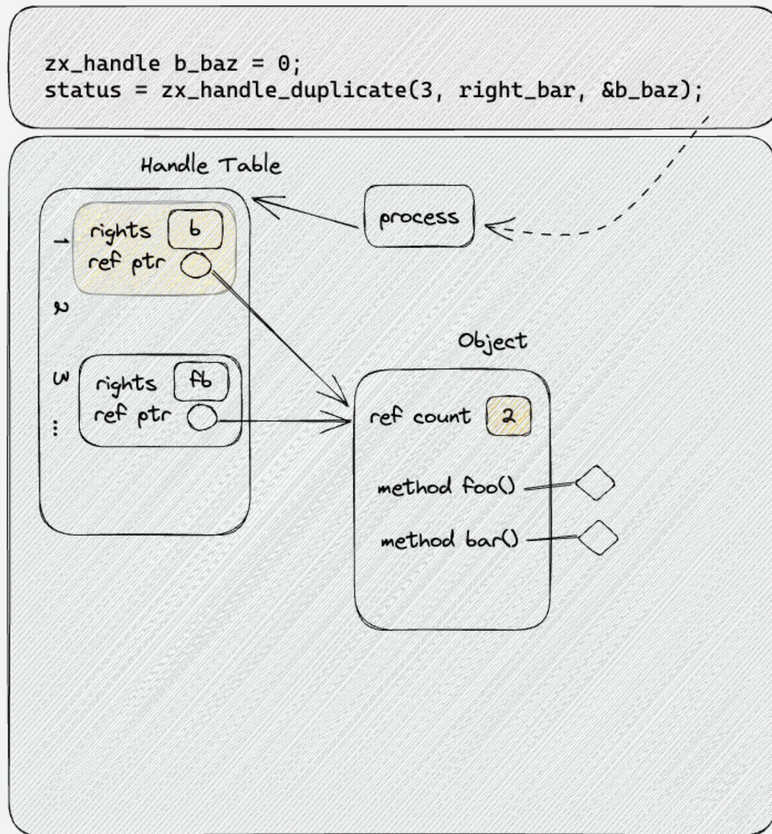
New handle “points” to the same kernel object

You can reduce the rights, or keep the same rights*.

Rights **modulate** the set of methods that are available via that handle.

```
zx_hande_close(1) ?
```

Unix FDs don’t carry rights, therefore dup() and variants only take the source FD.



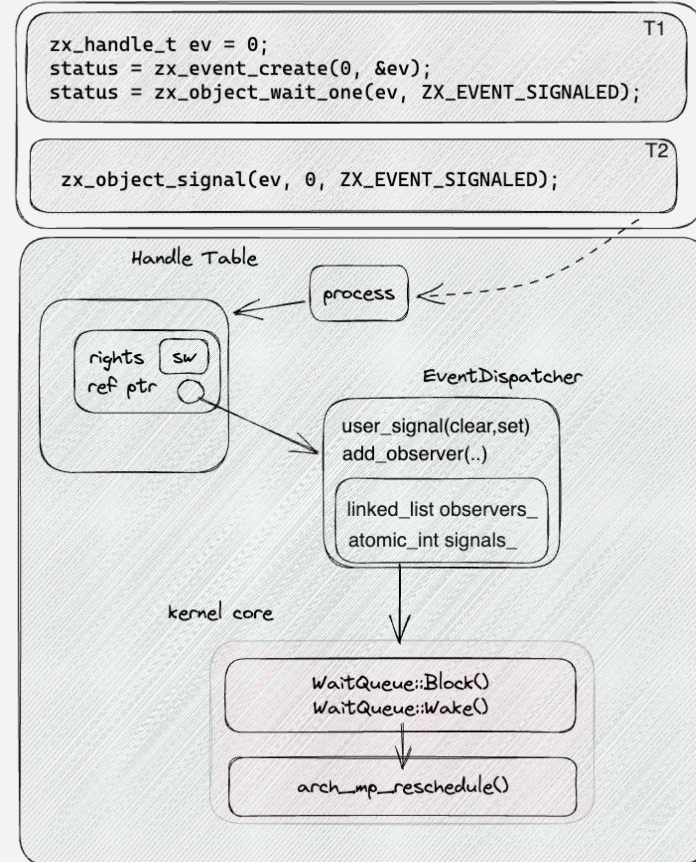
Duplication of handles

The Event

Simplest object: can be waited on or signaled.

Signaling and waiting apply to events and other objects hence zx_object rather than zx_event

Restricted form of polymorphism.



Ø1

IPC

Message passing is core to Fuchsia

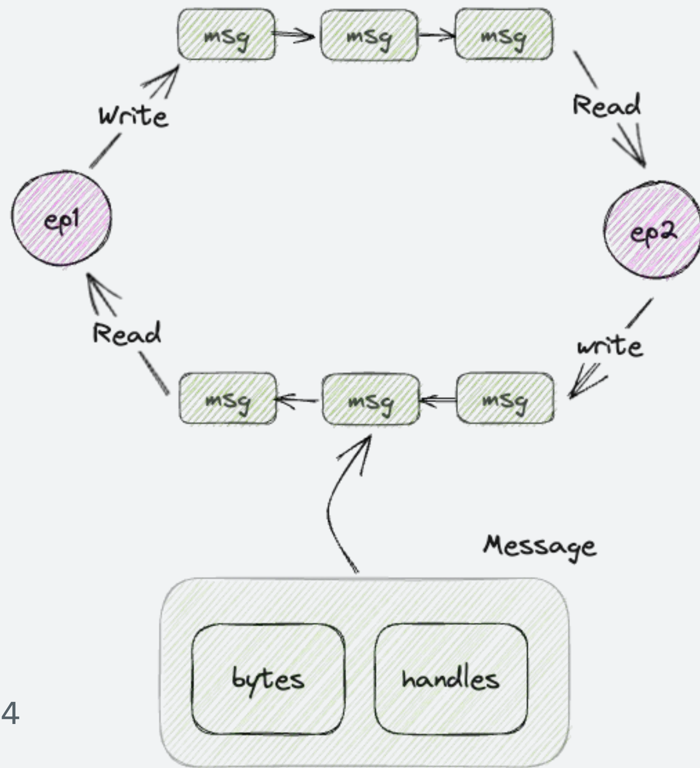
The Zircon IPC workhorse is the channel

```
zx_channel_create(0, &ep1, &ep2);

zx_channel_write(ep, bytes[], handles[]);

zx_channel_read(ep, &bytes[], &handles[]);

zx_object_wait(ep, ZX_CHANNEL_READABLE);
```

Handles “move” from process → message during write.

Max length of a message is 64KiB, max # handles is 64

Neither reading or writing blocks

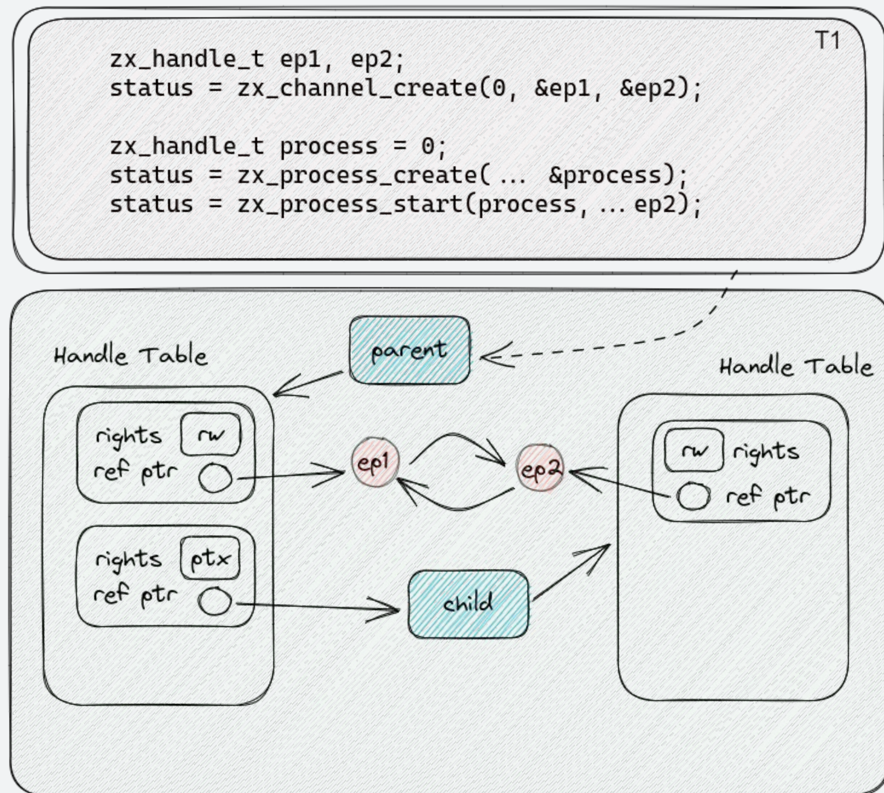


Process Creation and IPC "bootstrapping"

During the creation of the first thread, the caller gets to pass one handle.

- 99% of the times it should be a channel endpoint
- ..and the first message should be the child's environment.

Plot twist: the first message usually contains one or more channels.



Ø2

Higher Order IPC

Enter the Manager

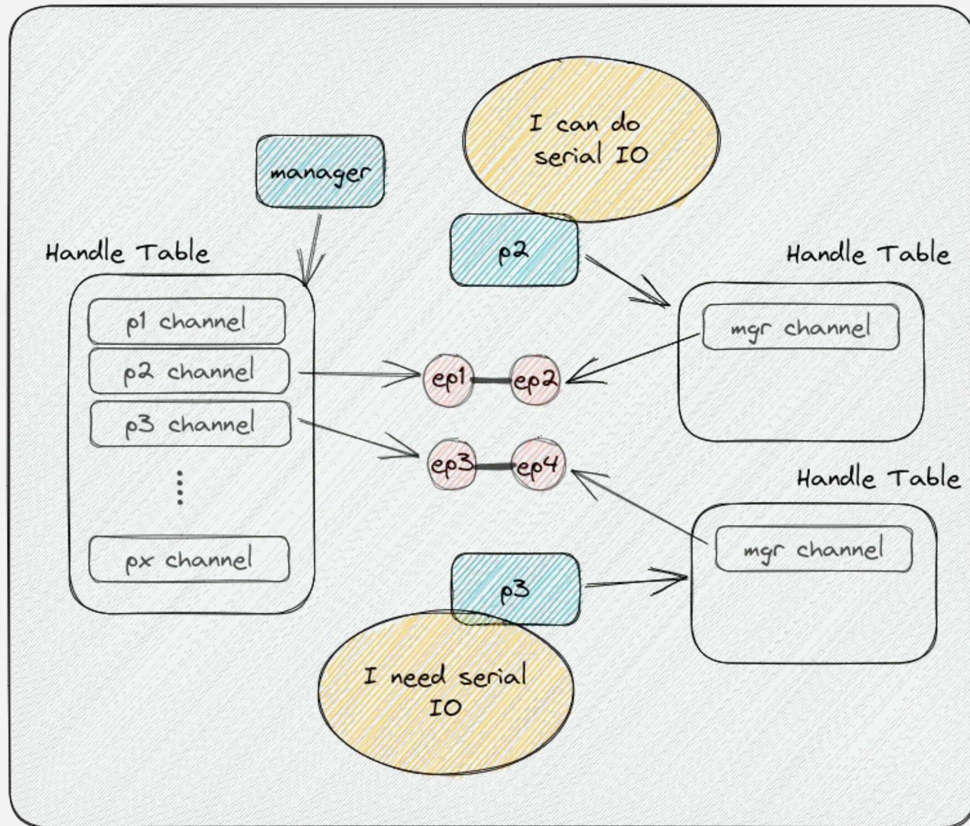
In Fuchsia there is a process that directly or indirectly starts all other processes

- Has a channel endpoint to all the children
- Keeps track of the lifetime

The manager knows what each process is meant to do

- What they need
- What they offer

Comes as a json-like metadata associated with the binary(es) of each process



Connecting client and Server

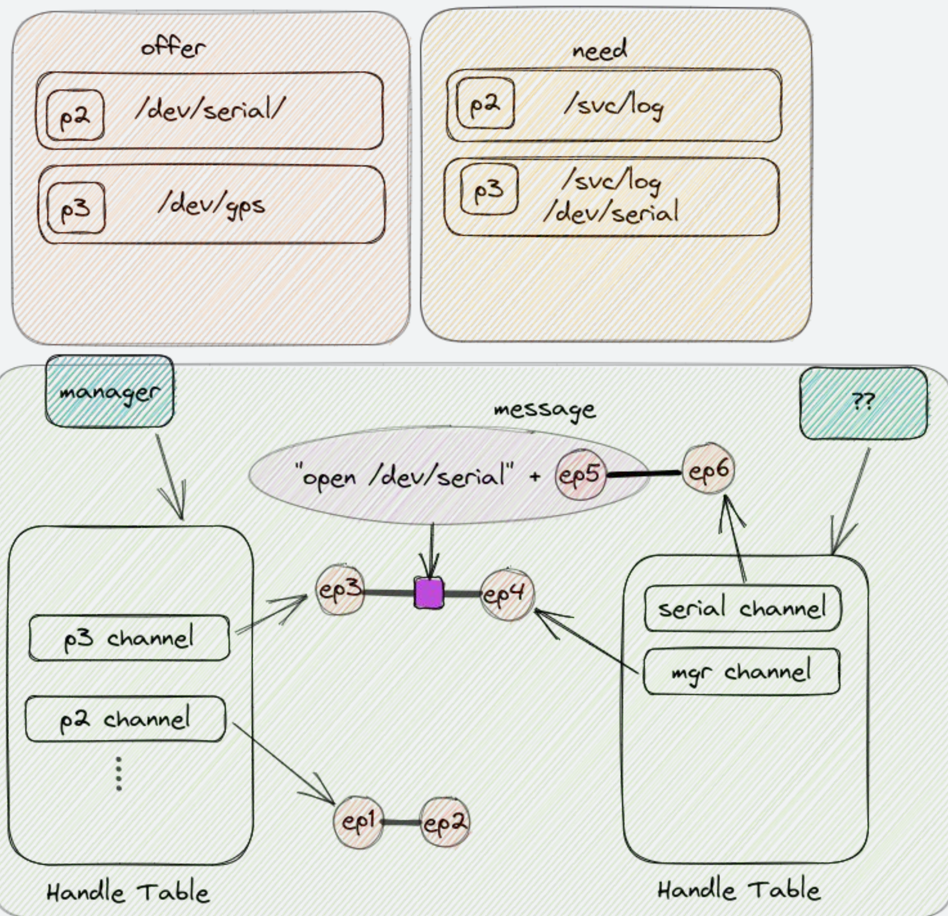
The manager has the namespaces for all the managed processes

- Routes "needs" with "offers"

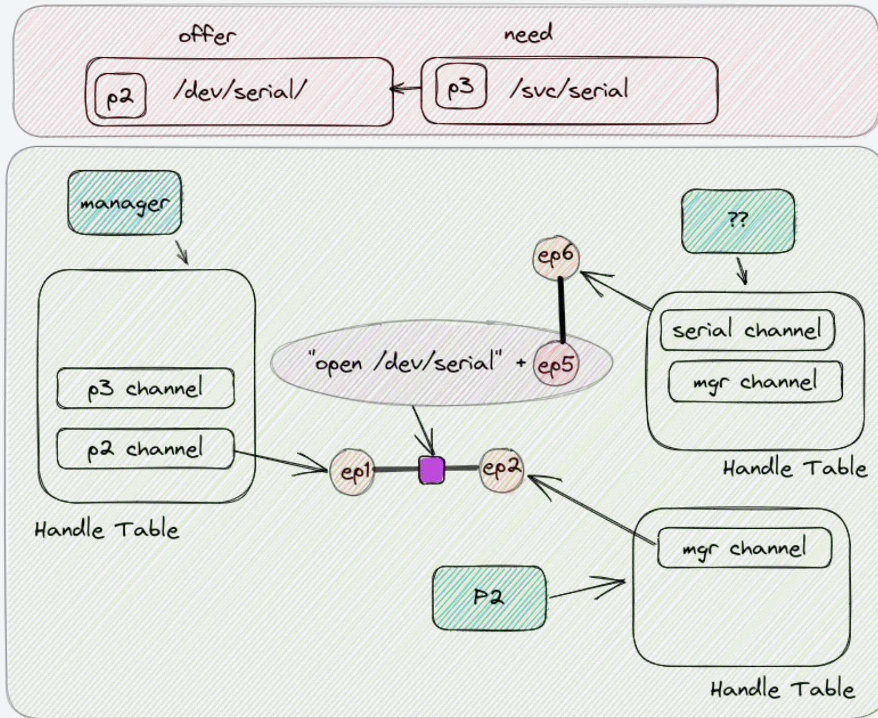
The "protocol" usually requires the client to send a channel endpoint

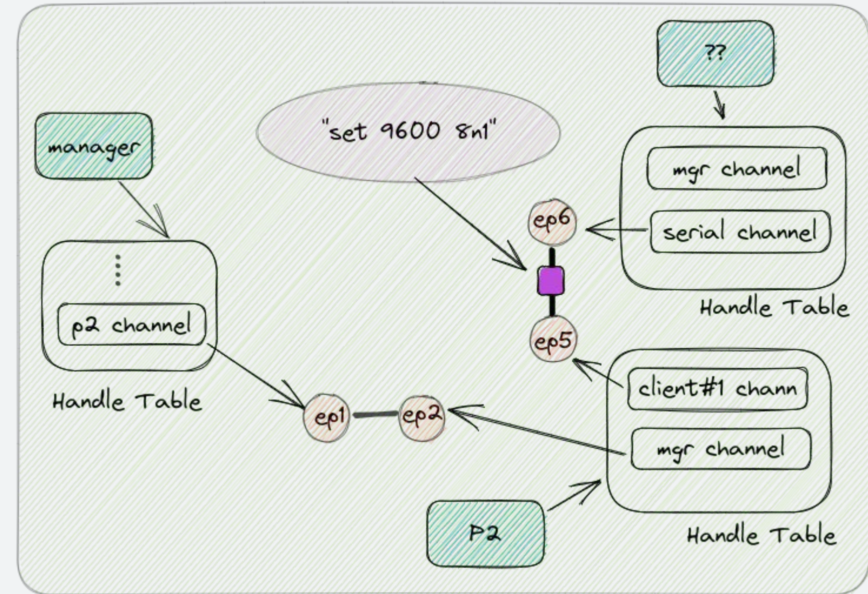
We don't need to care about which process is actually the client!

- Having the right channel endpoint and being able to write to it is all the authorization needed
- Does not need to be the original process



Connection Completed



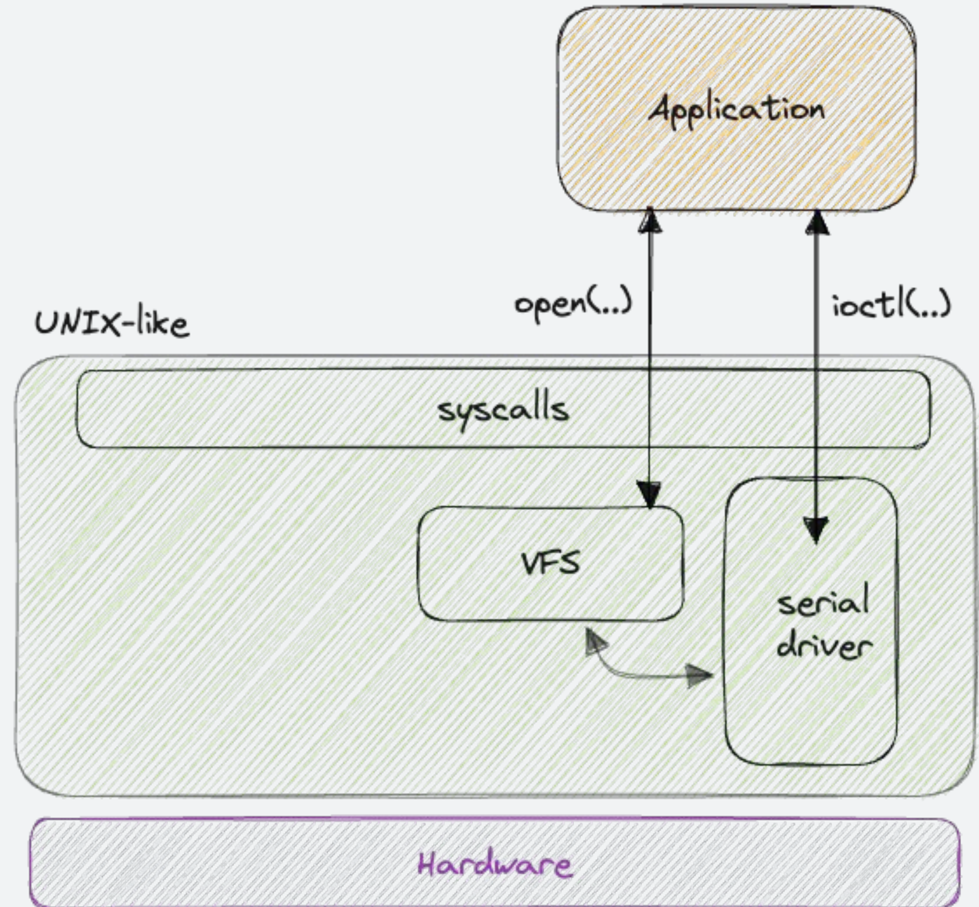


Ø3

Fuchsia Vs Others

Monolith vs Micro Kernels

Monoliths are fast: Just two Syscalls to open and configure a serial port



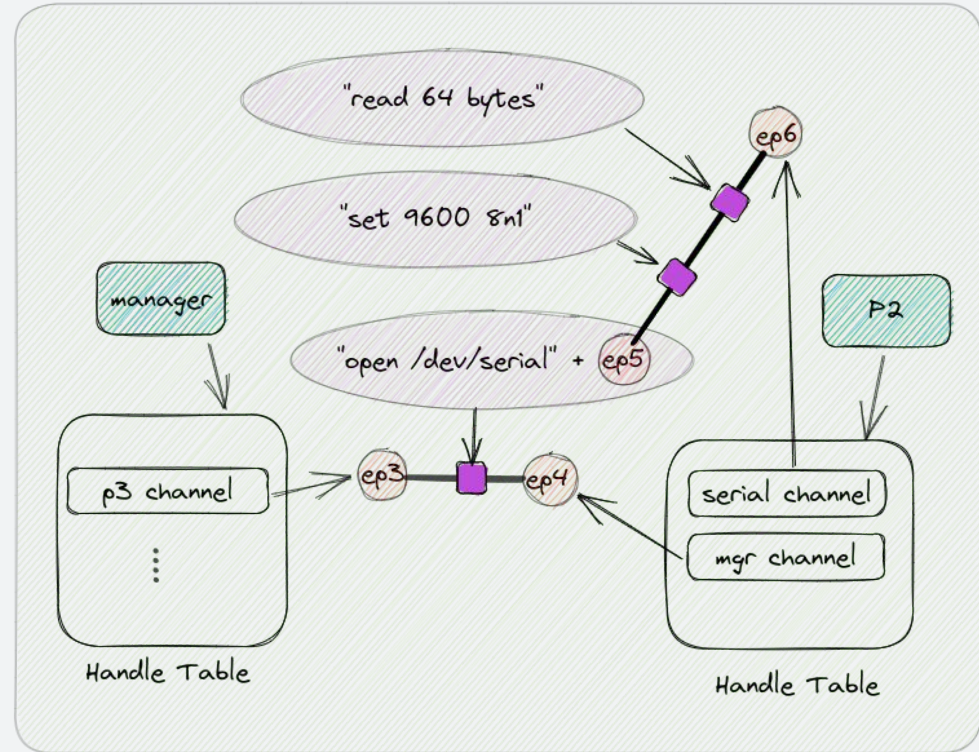
Going Faster + Saving Memory

Recall the manager has metadata for each binary from the start

- Normally starts servers lazily

The client should be optimistic and pipeline the commands:

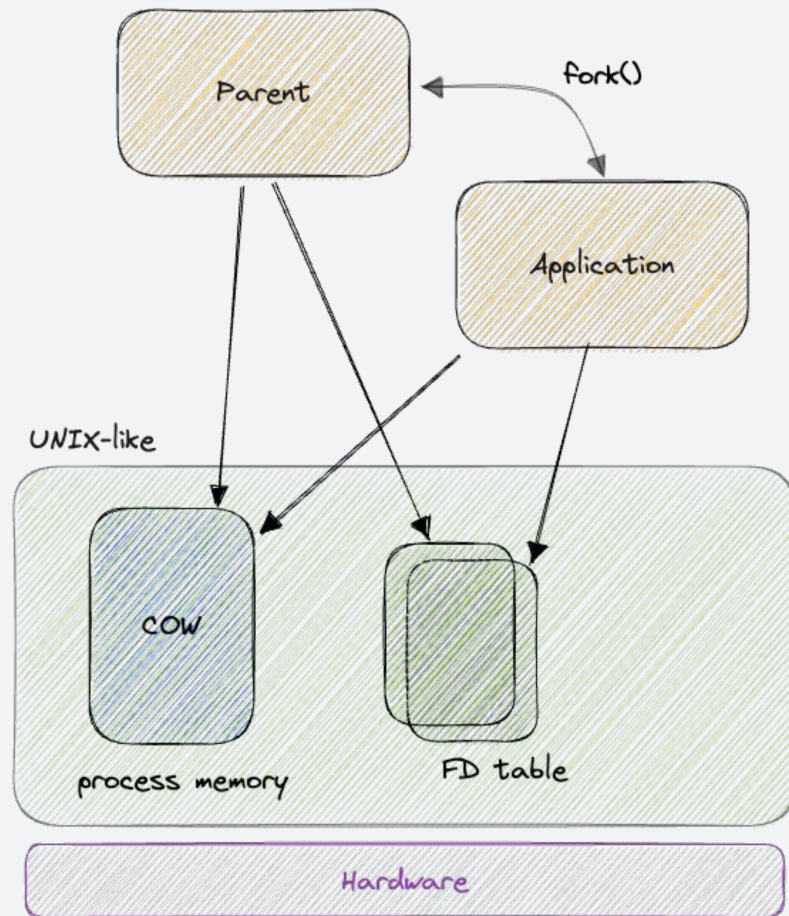
1. Open/Connect
2. Configure/Write
3. Close



Defaults are important

A fuchsia process has inherently no access to anything.

- The creator seeds it with the first channel
- The creator fully controls the namespace



Q&A

The End

Thank You!

Capabilities

Wikipedia, 2023

"A capability (known in some systems as a key) is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights. A user program on a capability-based operating system must use a capability to access an object. Capability-based security refers to the principle of designing user programs such that they directly share capabilities with each other according to the principle of least privilege, and to the operating system infrastructure necessary to make such transactions efficient and secure. Capability-based security is to be contrasted with an approach that uses traditional UNIX permissions and Access Control Lists.

Although most operating systems implement a facility which resembles capabilities, they typically do not provide enough support to allow for the exchange of capabilities among possibly mutually untrusting entities to be the primary means of granting and distributing access rights throughout the system. A capability-based system, in contrast, is designed with that goal in mind."