

Fuchsia Internals—Volume III

The Zircon Kernel: Architecture, Subsystems, and the User API

Fuchsia Internals Series

Abstract—Zircon is the microkernel at the core of Fuchsia OS. A monolithic kernel exports hundreds of unrelated system calls. Zircon instead exposes a small, uniform set of *kernel objects*. A program reaches an object only through a *handle*, which it cannot forge, and each handle holds a set of *rights*. Almost all OS policy lives in user space: drivers, file systems, the network stack and the graphics pipeline. The kernel keeps memory, scheduling, inter-process communication, and control over who may touch the hardware. This volume is a detailed tour of Zircon. It covers the object, handle and rights model and the system call interface behind the vDSO. It covers the job, process and thread hierarchy and the fair scheduler. It covers virtual and physical memory: VMOs, VMARs, the page queues and the user-space pager. It covers the IPC primitives (channels, sockets, FIFOs, ports and futexes), interrupts, exceptions and time. It covers the objects that grant hardware access: resources, MMIO, interrupts, BTIs and the IOMMU. It ends with the boot hand-off to userboot, restricted mode and the security model, diagnostics, and support for several CPU architectures. The volume is written for engineers who know hardware and RTL and want an exact model of how the kernel is built and how each part works.

Index Terms—Zircon, Fuchsia OS, microkernel, capability system, handles, virtual memory, VMO, scheduling, IPC, channels, futex, interrupts, IOMMU, restricted mode, operating system kernel.

I. INTRODUCTION: THE ZIRCON MICROKERNEL

ZIRCON is the microkernel at the heart of Fuchsia OS. It descends from LK (“Little Kernel”), a small kernel for embedded systems. Zircon has grown into a general-purpose microkernel built on object capabilities. It boots multiprocessor machines, manages virtual memory, schedules threads, controls access to hardware and routes messages between processes. It does almost nothing else. There are no drivers in the kernel, no file systems, no network stack and no graphics pipeline. Each of those runs in user space as an ordinary process. A program reaches them only through the kernel’s small set of message-passing primitives. The kernel exposes about 185 system calls, about half as many as Linux (Table I), and it treats them as a stable ABI [1], [3].

A. Design Axioms

Five design rules shape the rest of this volume.

1) *A small, stable syscall surface*: One machine-readable specification lists the whole kernel ABI. User space receives it through one shared library that the kernel provides, the vDSO (Section IV). A small surface is easier to audit. It also

lets the kernel change its internals without breaking existing binaries.

2) *Everything is a kernel object, named by a handle*: A *kernel object* is a reference-counted structure inside the kernel. It can be a block of memory, a thread, or one end of a communication link. User space never touches an object directly. Instead a process holds a *handle*: an integer, private to that process, that names the object and has a set of *rights* attached. The object is the noun, and the handle is the only way a program can refer to it [2], [4].

3) *Capabilities, not ambient authority*: In Zircon, authority is the pair (handle, rights). There is no global `root` user. The kernel has no built-in file system namespace. No permission comes from a user identity. A process can do exactly what its handles allow, and nothing more (Section V).

4) *Asynchrony everywhere*: Each object exposes a 32-bit word of *signals* that describe its state, such as “readable”, “peer closed” or “signaled”. A thread learns that the state changed by waiting on those signals. A *port* object gathers many such waits into one queue. That queue is the base of the event loops on which the rest of the system is built.

5) *Policy lives in user space*: The kernel provides mechanism, and user space decides policy. The kernel knows how to map a page or deliver an interrupt. User-space components decide *which* driver owns a device, and *what* file system format a partition holds.

lets the kernel change its internals without breaking existing binaries.

2) *Everything is a kernel object, named by a handle*: A *kernel object* is a reference-counted structure inside the kernel. It can be a block of memory, a thread, or one end of a communication link. User space never touches an object directly. Instead a process holds a *handle*: an integer, private to that process, that names the object and has a set of *rights* attached. The object is the noun, and the handle is the only way a program can refer to it [2], [4].

3) *Capabilities, not ambient authority*: In Zircon, authority is the pair (handle, rights). There is no global `root` user. The kernel has no built-in file system namespace. No permission comes from a user identity. A process can do exactly what its handles allow, and nothing more (Section V).

4) *Asynchrony everywhere*: Each object exposes a 32-bit word of *signals* that describe its state, such as “readable”, “peer closed” or “signaled”. A thread learns that the state changed by waiting on those signals. A *port* object gathers many such waits into one queue. That queue is the base of the event loops on which the rest of the system is built.

5) *Policy lives in user space*: The kernel provides mechanism, and user space decides policy. The kernel knows how to map a page or deliver an interrupt. User-space components decide *which* driver owns a device, and *what* file system format a partition holds.

B. Nouns and Verbs

You can read the whole system call list as a grammar of nouns and verbs. The nouns are the object types, such as `vmo`, `channel`, `thread` and `port`. The verbs are the system calls that act on a handle to one of them. The naming convention makes this explicit. Nearly every call is named `zx_noun_verb`, as in `zx_channel_write`, `zx_vmo_create` or `zx_thread_start`. Learning Zircon is mostly learning the set of nouns (Section III), then the few verbs that each noun accepts.

C. A Hardware Analogy

The object model maps closely onto hardware that an RTL engineer already knows. The kernel plays the part of the on-chip interconnect: the bus fabric, the MMU, and the arbiter that decides who may drive which resource. A kernel object is like the register file of an IP block: addressable state with defined behavior. A handle is like a bus-grant token that cannot be forged. Holding the token is what lets a master talk to the block, and the token’s permission bits are the handle’s rights. A `channel` is then an AXI-Stream link between two masters. A memory object (a `vmo`) mapped into a process is a region that the MMU has been told to expose at some virtual address. A master on a real bus cannot forge a grant it was never given. In the same way, a Zircon process cannot make up a handle. Only the kernel creates handles, and it hands them out only through system calls or over channels.

TABLE I
ZIRCON VERSUS A MONOLITHIC KERNEL (LINUX) AT A GLANCE.

Aspect	Zircon (microkernel)	Linux (monolithic)
Syscall surface	~185, object-oriented (<code>zx_noun_verb</code>)	~350+ POSIX calls
Drivers	User-space components	In-kernel modules
Filesystems	User-space services	In-kernel VFS
Network stack	User-space (<code>netstack</code>)	In-kernel
IPC	Channels, sockets, ports (object handles)	Pipes, sockets, signals, SysV
Security	Object capabilities: (handle, rights)	uid/gid, root, DAC
Fault isolation	Driver crash confined to its process	Driver bug can panic the kernel
Naming	Per-process handle table; no kernel-global namespace	Global VFS namespace, fds

Table I compares the two designs. The row with the largest consequence is fault isolation. A Fuchsia driver is a user process that holds only the handles it needs. So a bug in a driver crashes one process, not the machine. The rest of the system sees the crash through ordinary object signals and can recover from it.

D. A First Syscall Sequence

Listing 1 shows the smallest useful program. It allocates a page of memory, maps it into the calling process, stores to it as ordinary RAM, and releases the handle. Two objects appear: a `vmo`, which is the memory, and the process’s root `vmr`, which is its address space. Three verbs act on them. Look at the second-to-last line. Closing the VMO handle does *not* unmap the page, because the mapping in the VMAR holds its own kernel reference to the object. The object outlives the handle.

Listing 1. Create, map, write and close. Error handling is shown, and every identifier is a real Zircon call.

```
#include <zircon/syscalls.h>

zx_status_t map_and_write(void) {
    zx_handle_t vmo;
    zx_status_t st = zx_vmo_create(4096u, 0u, &vmo);
    if (st != ZX_OK) return st;

    zx_vaddr_t addr;
    st = zx_vmar_map(zx_vmar_root_self(),
                     ZX_VM_PERM_READ | ZX_VM_PERM_WRITE,
                     /*vmar_offset=*/0u, vmo,
                     /*vmo_offset=*/0u, 4096u, &addr);
    if (st != ZX_OK) { zx_handle_close(vmo); return st; }

    *(volatile uint32_t*)addr = 0xCAFEF00Du; // plain store

    zx_handle_close(vmo); // mapping still holds the VMO alive
    return ZX_OK;
}
```

Some calls never enter the kernel at all. `zx_vmar_root_self`, used above, is a C library function. It returns a cached handle to the process’s root address-space object. Time queries such as `zx_clock_get_monotonic` are vDSO calls that compute their result from a shared read-only page. Section IV describes these vDSO-local fast paths.

E. The Kernel Surface

Figure 1 draws the whole system in one picture. The LK-derived core sits in the center. Around it sit the families of object types that the kernel implements, grouped by purpose. The syscall and vDSO boundary encloses them. It is the only way for user space to enter the kernel. Outside that boundary are

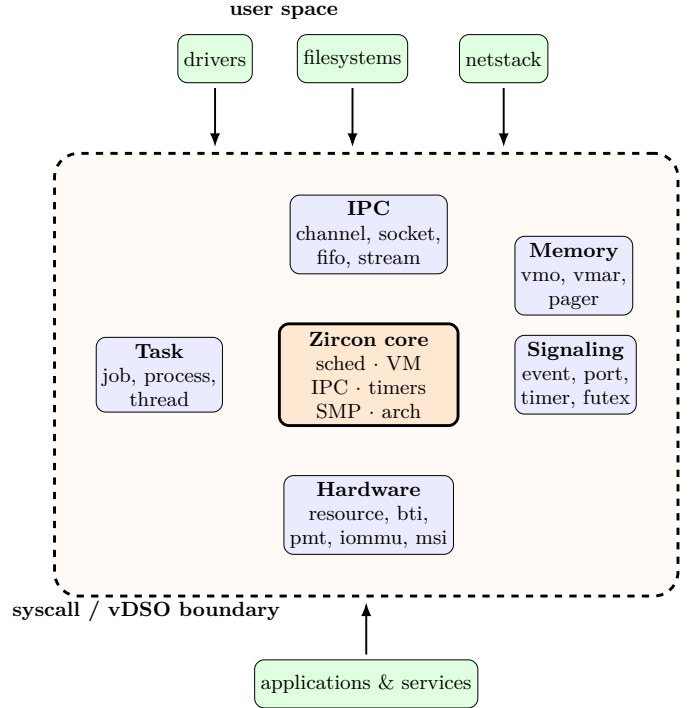


Fig. 1. The kernel surface. The LK-derived core implements five families of kernel objects: memory, IPC, task, signaling and hardware. The syscall and vDSO boundary encloses them. Drivers, file systems, the network stack and applications are user processes, and they reach the kernel only across that boundary.

the parts that a monolithic system would keep *inside* it: drivers, file systems, the network stack and applications. Each of them is a user process that uses handles. Sections II to V work inward from here. They cover the architecture and the kernel boundary (Section II), the object and handle model (Section III), the system call ABI and the vDSO (Section IV), and the security model built on rights (Section V).

II. SYSTEM ARCHITECTURE AND THE KERNEL BOUNDARY

Inside, Zircon is two layers stacked on the hardware. The lower layer is the *LK-derived core*. It holds the scheduler, the virtual memory code, timers, and the support for multiple CPUs and per-CPU state. It also holds the architecture-specific code under `arch/`, which knows about page tables, exception vectors and the instruction that traps into the kernel. The upper layer is the *object layer*. It holds the C++ classes that implement kernel objects, the per-process handle tables, and the system call implementations. Each system call turns a user request into a method call on an object. Figure 2 shows the stack. User space sits above the syscall and vDSO boundary, and everything below it is privileged.

A. The Dispatcher Pattern

Every object that user space can reach is a C++ class derived from the abstract `Dispatcher` base class. A virtual memory object is a `VmObjectDispatcher`. A channel endpoint is a `ChannelDispatcher`. A thread is a `ThreadDispatcher`. Jobs, processes, ports, timers and the other objects each have their own dispatcher subclass. The dispatcher holds the object’s state and implements the operations that the system calls invoke. Code that refers to any object uses `fb1::RefPtr<Dispatcher>`,

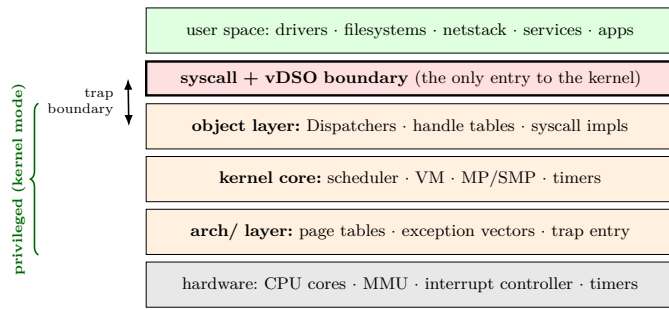


Fig. 2. Layered architecture. The syscall and vDSO boundary is the only privileged trap boundary. Below it, the object layer rests on the LK-derived core and the architecture-specific code, and that code drives the hardware.

a smart pointer that keeps the reference count inside the object [1].

Reference counting decides how long an object lives. A dispatcher stays alive exactly as long as something holds a strong reference to it. That can be a handle in some process’s handle table, or a reference inside the kernel, such as a VMAR mapping (as in Listing 1). It can also be a short-lived reference held during a system call. When the last strong reference goes away, the kernel destroys the object at once. There is no garbage collector.

When a dispatcher is built, the kernel gives it a *KOID*, a 64-bit kernel object identifier (`zx_koid_t`). A KOID is unique while the system runs and is never reused. Tools and user space use KOIDs to name objects in diagnostics. Kernel-generated KOIDs use 63 bits. Two values are reserved: `ZX_KOID_INVALID` (0) and `ZX_KOID_KERNEL`. A KOID stays the same in every process, even though the handle integers that point at the object differ from process to process.

B. The Handle Table

Each process owns a private *handle table*. To user space a handle is a 32-bit integer (`zx_handle_t`). Inside the kernel it is a table entry that binds a `Dispatcher*` to a *rights* mask. The integer encodes a table index and a per-slot *generation count*. When a handle is closed and its slot is reused, the generation changes. So a stale handle value fails to resolve, instead of silently naming a new object. The two least significant bits of a valid handle are always set, and the reserved value `ZX_HANDLE_INVALID` is 0 [4].

The table is per process, so the same integer means different things, or nothing, in different processes. A handle is meaningful only in the process that owns it. This is the kernel-level reason there is no ambient authority. A process has no way to name an object unless it holds a handle to it.

C. Syscall Flow, End to End

A system call passes through every layer of Figure 2. Take `zx_channel_write` as the example:

- 1) User code calls the vDSO function `zx_channel_write`. Only this vDSO code may issue the trap instruction (`syscall` on x86-64, `svc` on ARM64).
- 2) The trap enters the kernel’s system call entry code. That code saves the registers and dispatches on the system call number.

- 3) The kernel *validates the arguments*. It copies user pointers and lengths and checks their bounds, so a malformed request cannot read or write kernel memory.
- 4) The kernel *resolves the handle*. It looks up the handle argument in the calling process’s handle table and gets the `Dispatcher*` and the handle’s rights. It checks that the dispatcher has the expected type and that the rights include the ones the operation needs, here `ZX_RIGHT_WRITE`. If not, it returns `ZX_ERR_ACCESS_DENIED` (Section V).
- 5) The kernel calls the *dispatcher method*, `ChannelDispatcher::Write`. The method does the work: it copies the message, moves any transferred handles, and raises the peer’s `READABLE` signal.
- 6) A `zx_status_t` result returns through the entry code and the vDSO function to the caller. Any output parameters are written to user memory.

The entry check is what makes the vDSO an enforcement boundary and not only a convenience. The kernel records where each process mapped its vDSO. On every system call it checks the program counter of the trap instruction, and it rejects a `syscall` instruction executed from anywhere else. Section IV treats this in detail.

D. Process Memory Layout

Everything a process can touch is a *mapping* in its root VMAR (virtual memory address region). At startup the layout holds four things.

- The program’s code and data segments.
- The read-only vDSO mapping. The program loader places it, and the new process receives its address when it starts.
- An initial thread stack.
- A heap. The user-space allocator grows it by creating VMOs and mapping them.

Each extra thread adds a stack. Shared memory is a VMO mapped into more than one process. Each of these is just a VMAR mapping of a VMO. So a few memory calls build the whole address space: `zx_vmo_create`, `zx_vmar_map` and `zx_vmar_allocate`. Section IX covers virtual memory in full. Here it is enough to know that the address space is itself built from kernel objects.

E. Sharing Through the Handle Table

Figure 3 makes the per-process lookup concrete. Each of the two processes has its own handle table. Process A and Process B each hold a handle to one end of a channel. Those two handles point at the two peered `ChannelDispatcher` objects, which refer to each other. Both processes also map a shared VMO. Each holds a handle that resolves to the *same* `VmObjectDispatcher`, possibly with different rights. For example, A may write while B may only read. The figure also shows the entry structure. A handle integer decodes to an index and a generation, and the slot holds a `Dispatcher*` and the rights. The two tables name the same object with different integers, and possibly with different authority.

III. KERNEL OBJECTS AND HANDLES

This section defines the words that the rest of the volume uses: object, KOID, handle and dispatcher. It also lists the operations that create, copy, transfer and destroy handles.

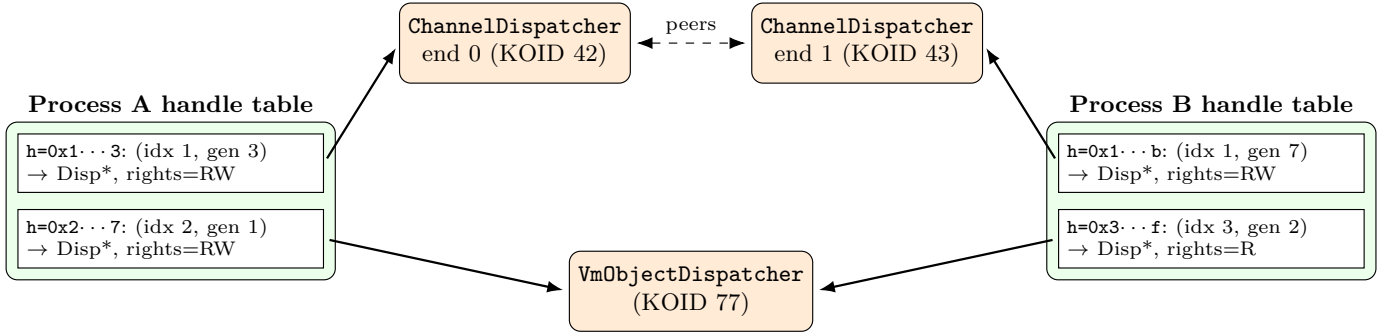


Fig. 3. Two processes share objects through their handle tables. Each handle integer decodes to an index and a generation, and the indexed slot holds a `Dispatcher*` and the rights. The two ends of a channel are separate, peered `ChannelDispatchers`. The shared VMO is one `VmObjectDispatcher` that both processes name, here with different rights: A may read and write, B may only read.

A. Definitions

A *kernel object* is a reference-counted structure that the kernel manages. As Section II described, a `Dispatcher` subclass implements it. Each object has a `KOID` (`zx_koid_t`): a 64-bit identifier that is unique across the system, allocated in increasing order, and never reused while the system runs [1], [2]. A *handle* is a 32-bit name for an object, private to one process (`zx_handle_t`). The handle table pairs it with a *rights* mask (`zx_rights_t`). Some objects come in peered pairs. A *channel* has two endpoints, and an *eventpair* has two ends. Each endpoint reports its partner's identity as a *related KOID*. So a query on a channel endpoint returns both its own `KOID` and the `related_koid` of the other end. That is how user space finds out that two handles it holds are the two sides of one link.

B. Object Lifecycle

An object's life is deterministic, and there is no garbage collection. A system call *creates* the object and returns one or more handles to it. Examples are `zx_vmo_create`, `zx_channel_create` and `zx_event_create`. The object stays alive while any strong reference to it exists. A reference can be a handle in some process's table. It can be an internal kernel reference: a VMAR mapping, a message in flight on a channel, or a wait registered against the object. It can also be a short-lived reference held during a system call. When the last reference goes away, the kernel *destroys* the object at once. Closing a handle lowers the count. When the count reaches zero, the dispatcher's destructor runs and releases any resources, such as physical pages and queued messages.

C. Handle Anatomy and Operations

Inside the kernel a handle ties together three things: a reference to its object, the rights that apply to it, and the process that owns it. While a handle is “in transit” across a channel, the kernel owns it instead [4]. The core operations are:

- `zx_handle_duplicate(h, rights, out)` makes a second handle to the same object. The handle `h` must have `ZX_RIGHT_DUPLICATE`. The new handle gets the requested subset of rights. If you pass `ZX_RIGHT_SAME_RIGHTS`, it gets a copy of the original's rights.
- `zx_handle_replace(h, rights, out)` consumes `h` and returns a new handle to the same object with the same or fewer rights. The original integer becomes invalid at the same instant, so the two never exist together.

- `zx_handle_close(h)` removes the handle from the table and drops its reference. Closing `ZX_HANDLE_INVALID` is allowed and does nothing, which keeps cleanup code simple.
- `zx_handle_close_many(handles, n)` closes an array of handles in one call.

Handles can also *move* between processes. When a process writes a handle into a channel with `zx_channel_write`, the kernel removes it from the sender's table and holds it while it is in transit. The receiver's `zx_channel_read` installs it in the receiving process's table. A transfer needs `ZX_RIGHT_TRANSFER` on the handle. It is a move, not a copy: no two processes ever name the object through that same handle at once. This is how capabilities flow through the system. To give a peer access to an object, you send it a handle. To give *reduced* access, you first duplicate the handle with fewer rights (Section V).

D. Introspection

The call for asking about an object is `zx_object_get_info`. With the topic `ZX_INFO_HANDLE_BASIC` it returns a record about one handle. The record holds the object's `koid`, the handle's `rights`, the object type (`ZX_OBJ_TYPE_CHANNEL`, `ZX_OBJ_TYPE_VMO`, and so on), and the `related_koid` for peered objects. Other topics report details for one object type, such as `ZX_INFO_PROCESS`, `ZX_INFO_VMAR` and `ZX_INFO_JOB_CHILDREN`. Objects also have properties, read with `zx_object_get_property` and written with `zx_object_set_property`. The most common one is `ZX_PROP_NAME`, a short name that makes threads, processes and VMOs easy to recognize in diagnostics. Reading a property needs `ZX_RIGHT_GET_PROPERTY`, and writing one needs `ZX_RIGHT_SET_PROPERTY`.

E. Peered Objects and Closure Signals

Channels, sockets, FIFOs and eventpairs are *peered*. They are created as a pair, and each end is a separate object that refers to the other. The `PEER_CLOSED` signal is what makes them useful. When one end is closed, the kernel asserts the matching signal on the end that survives: `ZX_CHANNEL_PEER_CLOSED`, `ZX_SOCKET_PEER_CLOSED`, `ZX_FIFO_PEER_CLOSED` or `ZX_EVENTPAIR_PEER_CLOSED`. A thread or port waiting on that end wakes up. It then knows for certain that no more messages can arrive. This is the kernel's version of a hang-up. Together with the readable and writable signals, it gives a complete, race-free way to tear down a stream or datagram link, with no extra convention.

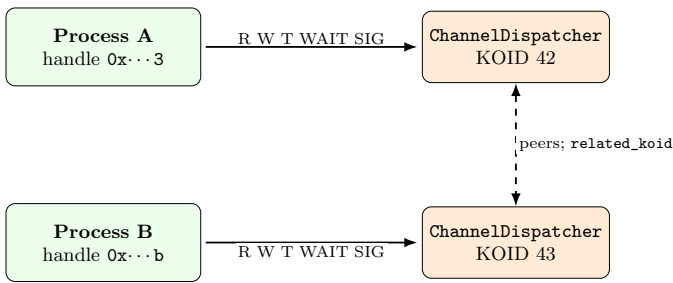


Fig. 4. Object, handle and dispatcher for a shared channel. Each process names one endpoint through a handle in its own table. The handles resolve to the two peered `ChannelDispatchers`, and each reports the other as its `related_koid`. The edge labels are the handles’ rights.

F. The Object Catalog

Table II lists the main object types by family, each with its role in one line. These are the nouns of Section I-B. The later sections cover the verbs for each.

G. Object, Handle, Dispatcher Together

Figure 4 puts the model together on the standard example: two processes that share a channel. Each process holds, in its own table, a handle to one endpoint. Each handle resolves to a `ChannelDispatcher`, and the two dispatchers are peers. The figure also shows the rights on each handle. Both sides usually hold READ, WRITE, TRANSFER, WAIT and the signal rights, but not DUPLICATE. Channel endpoints cannot be duplicated by default, so each endpoint has at most one owner. The same picture of handle to dispatcher, dispatcher to peer, and rights on the edge appears throughout Zircon.

One more property ties objects to the asynchrony rule of Section I. Most object types are *waitable*. Each has a 32-bit `zx_signals_t` word, and its bits report changes of state. Section XIV covers how a thread watches those bits, with `zx_object_wait_one`, `zx_object_wait_async` and ports. Here it is enough to know that nearly every object in the catalog is waitable.

IV. THE SYSTEM CALL ABI AND THE vDSO

User space reaches Zircon’s system calls through one shared library, the *vDSO* (virtual Dynamic Shared Object). No file system holds the vDSO. The kernel embeds its image at build time and maps it read-only into each process at startup, as a VMO. The vDSO does two jobs. It gives user space ordinary C functions to call, and it is the boundary that decides what counts as a system call [1], [3].

A. The vDSO as the Sole Gateway

A program may issue a system call only from inside the vDSO. The raw trap instruction is `syscall` on x86-64 and `svc` on ARM64. Running it from application code does not work as a back door. When a thread enters the kernel, the kernel subtracts the base address of the process’s vDSO mapping from the PC of the trap instruction. It then tests the result against a validity predicate for that system call. If the PC is not at a legitimate call site in the vDSO code, the kernel rejects the call with a synthetic exception. The thread sees the same result as if it had executed an undefined or privileged instruction. To keep this check cheap and unambiguous, the kernel allows each

process exactly one executable vDSO mapping, at the correct size and offset. It records the address of that mapping when the mapping is made.

The boundary serves three purposes:

- *ABI stability.* The trap sequence lives inside a library that ships with the kernel. Every program calls through the same stubs, so the calling convention between user and kernel can change without recompiling binaries.
- *Enforcement.* The vDSO checks the PC of the trap instruction, so “which code may talk to the kernel” is a rule the kernel checks, not a convention.
- *Fast paths.* The kernel controls the library, so it can implement some calls entirely in user space.

B. vDSO-Local Calls

Not every `zx_*` function traps. The vDSO computes some results from a read-only page that the kernel shares with every process. That page is the `vdso_constants` block, which the kernel fills in before the first process runs. Calls such as `zx_system_get_num_cpus` and `zx_ticks_per_second` simply return constants from that page. Time reads go one step further. `zx_ticks_get` reads the hardware cycle counter directly. `zx_clock_get_monotonic` scales that raw count into nanoseconds, using the ratio stored in `vdso_constants`. Neither call enters the kernel. System calls therefore fall into three performance classes:

- **vDSO-local:** no trap. The result is read or computed from the shared page (for example `zx_clock_get_monotonic` and `zx_ticks_get`).
- **Non-blocking:** a kernel trap that completes without sleeping (for example `zx_channel_write` and `zx_handle_close`).
- **Blocking:** a kernel trap that may park the thread until a condition holds (for example `zx_object_wait_one`, `zx_port_wait` and `zx_futex_wait`).

C. Definition and Code Generation

The whole ABI is written down in specification files under `//zircon/vdso`, in FIDL syntax. The system calls of each object type form one protocol, marked `@transport("Syscall")`. Each method declaration names its parameters, their types, the handle type it expects, and its error return. A generator, the `zither` toolchain, reads these files. It emits the public C and C++ declarations that user space links against, and the private entry interface on the kernel side. Methods marked `vdso_call` do not trap; hand-written C++ in the vDSO implements them. Methods marked `internal` are private entries that only the vDSO implementation uses. Listing 2 shows the definition of `zx_channel_write`, and Listing 3 shows the C signature generated from it.

Listing 2. The FIDL-style definition of the system call `Channel.Write`.

```

// //zircon/vdso/channel.fidl
@transport("Syscall")
closed protocol Channel {
  strict Write(resource struct {
    handle Handle:CHANNEL;
    options uint32;
    @voidptr @size32
    bytes vector<byte>:CHANNEL_MAX_MSG_BYTES;
    @release @size32
    handles vector<Handle>:CHANNEL_MAX_MSG_HANDLES;
  }) -> () error Status;
};
  
```

Listing 3. The C prototype that `zither` generates from Listing 2. Because of `@release`, the handles sent leave the caller’s table.

TABLE II
THE MAJOR ZIRCON KERNEL OBJECT TYPES, GROUPED BY FAMILY.

Family	Object	Role
IPC	channel	Two-way datagram link for bytes <i>and</i> handles. FIDL IPC runs on it.
	socket	Two-way byte-stream (or datagram) pipe. Bytes only, no handle transfer.
	fifo	Fixed-size, fixed-element-count ring for small low-latency messages.
	stream	Seekable read/write cursor over a VMO's contents.
	iob	I/O buffer: a shared-memory region with controlled access, for high-throughput transport.
Memory	vmo	Virtual Memory Object: a sized, resizable container of pages. It is the unit of memory.
	vmar	Virtual Memory Address Region: a recursive slice of an address space into which VMOs are mapped.
	pager	User-space pager: services page faults for pager-backed VMOs on demand.
Task	job	Container and policy domain for processes and child jobs. Jobs form the task hierarchy.
	process	Address space plus handle table. Holds one or more threads.
	thread	The schedulable unit of execution within a process.
Signaling / sync	event	A bare signalable object: a named bit for software signaling.
	eventpair	Peered events; closing one asserts PEER_CLOSED on the other.
	port	Queue that gathers asynchronous signal and packet notifications. Event loops are built on it.
	timer	One-shot/deadline object that asserts a signal at a scheduled time.
	futex	Fast user-space lock primitive. It has no handle; a user address is its key.
	interrupt	Binds a hardware IRQ to a waitable object delivered to a user-space driver.
Hardware	counter	64-bit signed counter with atomic add and read. The kernel asserts one signal when the value is positive and another when it is not.
	resource	Unforgeable capability gating access to physical address ranges, IRQs, and privileged operations.
	bti	Bus Transaction Initiator: a device's view of memory for DMA. It pins memory for DMA.
	pmt	Pinned Memory Token: proof that pages are pinned for a BTI. zx_pmt_unpin releases them.
	iommu	Models a hardware IOMMU from which BTIs are minted.
System / misc	msi	Message-Signaled Interrupt allocation for PCI-style devices.
	clock	User-controllable clock object defining a synthetic timeline.
	profile	Scheduling profile (priority/deadline) applied to threads.
	guest	A virtual machine's physical address space (hypervisor).
	vcpu	A virtual CPU belonging to a guest .
	debuglog	Handle for writing to (and, if allowed, reading) the kernel's debug log.
	exception	Object that stands for a faulting thread. The kernel delivers it to a debugger or handler.
	suspend_token	Keeps a task suspended. The task resumes when the token handle is closed.

```
// generated <zircon/syscalls.h>
zx_status_t zx_channel_write(
    zx_handle_t handle,          // requires ZX_RIGHT_WRITE
    uint32_t options,
    const void* bytes,           // num_bytes-long buffer
    uint32_t num_bytes,
    const zx_handle_t* handles, // moved out of caller
    uint32_t num_handles);
```

Each **vector** parameter becomes a pointer and a 32-bit count. The **error Status** clause makes every system call return a **zx_status_t**. The **@release** annotation on the handle vector is where move semantics come from: the call removes the listed handles from the caller's table.

D. The Error Model

Zircon has no **errno**. Every system call returns a **zx_status_t**. The value is **ZX_OK** (0) on success and a negative **ZX_ERR_*** code on failure. The status is the *only* place error information appears. Results come back through out-parameters, which the kernel writes only on success. An out-parameter of handle type passes

ownership. For example, a successful **zx_channel_read** installs the received handles in the caller's table, and from then on the caller must close them. These are the codes a programmer meets most often:

- **ZX_ERR_BAD_HANDLE**: the integer is not a live handle.
- **ZX_ERR_ACCESS_DENIED**: the handle lacks a required right.
- **ZX_ERR_WRONG_TYPE**: the handle names an object of the wrong kind for this call.
- **ZX_ERR_INVALID_ARGS**: a parameter is malformed.
- **ZX_ERR_SHOULD_WAIT**: a non-blocking call found nothing ready.
- **ZX_ERR_PEER_CLOSED**: the other end of a peered object is gone.
- **ZX_ERR_BUFFER_TOO_SMALL**: the buffer supplied cannot hold the result.

The codes mean the same thing across the whole interface, so one pattern of error handling works everywhere. The **if (st != ZX_OK)** check of Listing 1 fits every call.

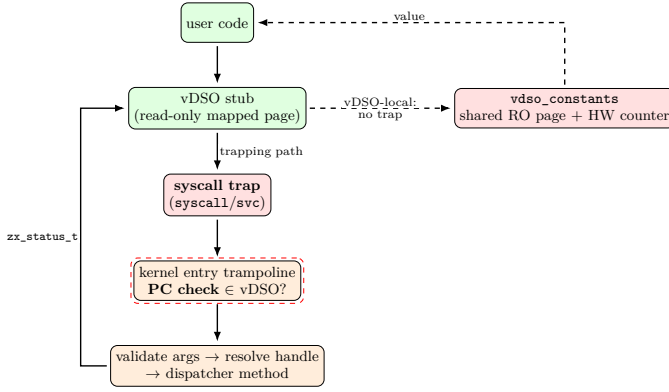


Fig. 5. The two system call paths. A trapping call goes through the vDSO stub and the trap instruction. The kernel’s entry trampoline checks that the PC of the trap lies inside the recorded vDSO mapping. Only then does the kernel validate the arguments, resolve the handle and call the dispatcher. A vDSO-local call (dashed) returns a value computed from a shared read-only page and never enters the kernel.

TABLE III
REPRESENTATIVE SYSCALLS BY FUNCTIONAL CATEGORY.

Category	Example	Action
Object	<code>zx_object_get_info</code>	Inspect a handle or object.
Handle	<code>zx_handle_duplicate</code>	Copy a handle with equal or fewer rights.
Task	<code>zx_process_start</code>	Begin executing a process.
VM	<code>zx_vmar_map</code>	Map a VMO into an address space.
IPC	<code>zx_channel_call</code>	Send a message and wait for its reply, as one call.
Port/sig	<code>zx_port_wait</code>	Take the next asynchronous packet from a port.
Time	<code>zx_clock_get_monotonic</code>	Read monotonic time (vDSO-local).
Hardware	<code>zx_bti_pin</code>	Pin VMO pages for device DMA.
System	<code>zx_system_get_num_cpus</code>	Query CPU count (vDSO-local).

E. ABI Categories and Restricted Views

The specification marks each call with a stability category. Most calls are *stable*; they form the committed ABI. Calls still under development are marked *@next*. Callers in the Fuchsia tree can use them by building against the “next” ABI, but they are not yet frozen.

A separate mechanism narrows what a thread may call. Fuchsia runs foreign binaries, such as Linux programs under the Starnix runtime, in *restricted mode*. A thread in restricted mode has a curtailed view: the kernel does not service a system call it makes directly. Instead the same thread returns to normal mode, where its supervising code handles the call. Restricted mode belongs to the security model of §XIX. Here it is enough to note that the set of calls a thread may make is a setting, not a global constant.

F. The Two Call Paths

Figure 5 compares the trapping and non-trapping paths. Table III then gives one representative call from each functional category. It shows the breadth of the interface, about 185 calls, without listing them all.

V. CAPABILITIES: RIGHTS AND THE SECURITY MODEL

Sections III and IV showed that authority moves as handles. This section covers the other half of a capability: the *rights*

attached to each handle. It gives the rule that governs how rights change, and it shows how the kernel enforces them. In Fuchsia the unit of authority is the pair (handle, rights) [4], [5].

A. The Rights Bitmask

Each handle holds a `zx_rights_t`, a 32-bit mask. The kernel gives each named right one bit. These are the rights that matter most:

- **ZX_RIGHT_DUPLICATE**: `zx_handle_duplicate` may copy the handle.
- **ZX_RIGHT_TRANSFER**: the handle may be written into a channel, and so moved to another process.
- **ZX_RIGHT_READ** and **ZX_RIGHT_WRITE**: read or write the object’s contents, such as VMO bytes or channel messages. Together with **MAP**, they decide whether a mapping may be readable or writable.
- **ZX_RIGHT_EXECUTE**: together with **MAP**, allows an executable mapping of a VMO.
- **ZX_RIGHT_MAP**: the VMO may be mapped into a VMAR at all.
- **ZX_RIGHT_GET_PROPERTY** and **ZX_RIGHT_SET_PROPERTY**: read or write object properties, such as `ZX_PROP_NAME`.
- **ZX_RIGHT_ENUMERATE**: list the children of a job or process.
- **ZX_RIGHT_DESTROY**: kill a task object.
- **ZX_RIGHT_SIGNAL** and **ZX_RIGHT_SIGNAL_PEER**: assert user signals on the object itself, or on the partner of a peered object.
- **ZX_RIGHT_WAIT**: wait on the object’s signals, with `zx_object_wait_one` or through a port.
- **ZX_RIGHT_INSPECT**: query the object with `zx_object_get_info`.
- **ZX_RIGHT_MANAGE_JOB**, **_PROCESS** and **_THREAD**: create or control the children or threads of a task.
- **ZX_RIGHT_APPLY_PROFILE**: apply a scheduling profile to a thread.
- **ZX_RIGHT_RESIZE**: resize a resizable VMO. It also needs **WRITE**.

The kernel also defines shorthand groups, which it uses when it sets default rights:

- **ZX_RIGHTS_BASIC** = **DUPLICATE** | **TRANSFER** | **WAIT** | **INSPECT**.
- **ZX_RIGHTS_IO** = **READ** | **WRITE**.
- **ZX_RIGHTS_PROPERTY** = **GET_PROPERTY** | **SET_PROPERTY**.
- **ZX_RIGHTS_POLICY** = **GET_POLICY** | **SET_POLICY**.

B. The Monotonic-Narrowing Rule

The central rule is that **rights only ever decrease**. No system call anywhere in the ABI adds a right to a handle. The two calls that produce new handles can only keep rights or drop them:

- `zx_handle_duplicate(h, r, out)` returns a new handle with rights *r*. The set *r* must be a *subset* of the rights of *h*. A request for more fails with `ZX_ERR_INVALID_ARGS`. The special value `ZX_RIGHT_SAME_RIGHTS` copies the existing rights unchanged.
- `zx_handle_replace(h, r, out)` consumes *h* and returns a handle whose rights *r* are at most those of *h*. Here too, `ZX_RIGHT_SAME_RIGHTS` means “unchanged”.

The kernel has no way to amplify rights, so a process can reason about the worst case. Any handle it hands out grants no more authority than the handle it started from. That stays true

TABLE IV

DEFAULT RIGHTS ON THE HANDLE RETURNED WHEN AN OBJECT IS CREATED. KEY: D = DUPLICATE, T = TRANSFER, R = READ, W = WRITE, X = EXECUTE, M = MAP, GP/SP = GET/SET_PROPERTY, EN = ENUMERATE, DE = DESTROY, PO = GET/SET_POLICY, S = SIGNAL, SP = SIGNAL_PEER, WT = WAIT, I = INSPECT, MJ/MP/MT = MANAGE_JOB/PROCESS/THREAD, OC = OP_CHILDREN.

Object	Default rights
channel	T R W S Sp Wt I (no D)
vmo	D T R W M gp sp S Wt I (no X)
vmr	D T I oc
process	D T R W gp sp En De S Wt I mp mt
thread	D T R W gp sp De S Wt I mt
job	D T R W gp sp En De Po S Wt I mj mp mt
event	D T S Wt I
eventpair	D T S Sp Wt I
port	D T R W I (no Wt)
timer	D T W S Wt I
socket	D T R W gp sp S Sp Wt I
fifo	D T R W S Sp Wt I
interrupt	D T R W S Wt I
resource	D T W I

after the handle passes through any number of other processes. This is what makes capability *attenuation*, handing out a weaker copy, safe.

C. Default Rights

A system call that creates an object returns a handle with a *default* set of rights for that object type (Table IV). The defaults include the rights the object’s normal operations need, and leave out rights that would surprise the caller. Two omissions show the idea:

- A channel endpoint lacks **DUPLICATE** by default. An endpoint is meant to have a single owner, so it can be moved but not copied.
- A new VMO lacks **EXECUTE** by default. Writing code into memory does not make it executable. The execute right has to be obtained separately, through a resource, and this blocks code injection at the kernel level.

D. Capabilities versus Ambient Authority

POSIX works the other way on purpose. In a POSIX system, authority is *ambient*: it comes from who you are, not from what you hold.

- A thread’s user and group identity (**uid** and **gid**) apply silently to every operation.
- The superuser, **root**, bypasses the checks entirely.
- Any process can name any path in one global file system namespace. Permission bits decide afterwards whether the access succeeds.
- A program gains privilege by changing identity, for example through **setuid**.

Zircon has none of this. It has no **uid**, no **root**, and no namespace shared across the kernel. A thread’s authority is exactly the set of handles in its process’s table, each limited by its rights. To grant authority, you hand over a handle. To deny it, you never do. So the question “what can this process do?” has an exact answer that you can list: its handle table.

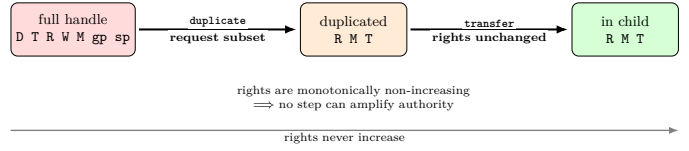


Fig. 6. Rights narrowing as a one-way pipeline. A full-rights handle is duplicated down to a chosen subset, then transferred into a child process with its rights unchanged. No step adds a right.

E. How Rights Are Enforced

The kernel enforces rights when it resolves a handle, the step in the system call flow of Section II-C. Each dispatcher method states the rights its operation requires. When the kernel resolves the handle argument, it compares the handle’s rights mask with that requirement. If a right is missing, the kernel returns **ZX_ERR_ACCESS_DENIED** before the operation runs. Some examples:

- **zx_vmo_write** requires **ZX_RIGHT_WRITE** on the VMO handle.
- Mapping that VMO writable with **zx_vmr_map** requires **ZX_RIGHT_WRITE** and **ZX_RIGHT_MAP**.
- Mapping it executable requires **ZX_RIGHT_EXECUTE** and **ZX_RIGHT_MAP**.
- Sending a handle over a channel requires **ZX_RIGHT_TRANSFER** on that handle and **ZX_RIGHT_WRITE** on the channel.

Each check is a simple bit test. Every path goes through handle resolution, so no operation escapes the checks.

F. Building Least Privilege

A parent uses the narrowing rule to give a child only what it needs. The pattern has two steps (Figure 6). First, **duplicate** a full-rights handle down to the smallest set the child needs. Then **transfer** that reduced handle to the child over a channel. Two examples:

- To share a configuration blob read-only, duplicate the VMO handle with only **READ | MAP | TRANSFER** and send the copy. The child can map and read the page, but no bug in the child can modify it.
- To let a child spawn its own processes but not create child jobs, hand it a job handle that keeps **MANAGE_PROCESS** but lacks **MANAGE_JOB**.

Each decision is local and explicit, and the kernel enforces it with the same checks as every other system call.

These kernel rights sit underneath the capability routing of the Component Framework (Volume VI). When the framework “routes a capability” from one component to another, it arranges for a handle to be transferred over a channel. That handle is narrowed to the rights the receiver should have. The component model adds declarations, names and policy on top. The authority it moves is still the (handle, rights) pair defined here. Every later subsystem in this volume builds on the same foundation: tasks (§VI), memory (§IX), IPC (§XII), interrupts (§XV) and hardware access (§XVII).

VI. TASKS: JOBS, PROCESSES, AND THREADS

Zircon groups all runnable state under three *task* object types. They form a strict containment hierarchy. *Jobs* contain processes and child jobs, *processes* contain threads, and *threads* are what the scheduler runs. All three share one set of controls. You can name a task, query it, suspend it, kill it, and give it an exception channel. Each type then adds structure of its own.

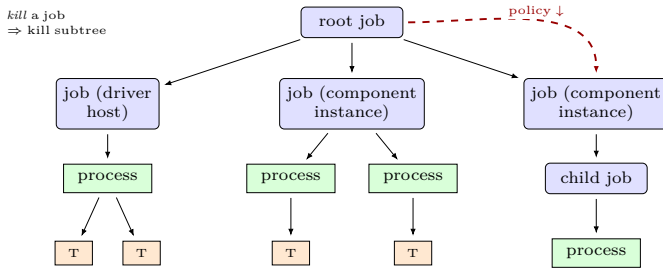


Fig. 7. The task hierarchy. A single root job anchors a tree of jobs, processes and threads. Job policy and lifetime flow downward. A child job may change an inherited policy entry only if the parent allowed it.

This section covers the hierarchy, the lifecycle of each object, and *job policy*, which turns the tree into a sandbox. It ends with the introspection that the `zxdb` debugger uses [1], [14].

A. The Task Hierarchy

Every task descends from a single *root job*. The kernel creates it at boot and hands it to `userboot` (§XVIII). From there, user space builds a tree. Each job may hold any number of child jobs and processes, and each process holds one or more threads. The tree does more than organize. Two things flow down it:

- *Job policy*: the security and resource constraints of §VI-F.
- *Lifetime*: killing a job kills everything beneath it.

A child job inherits its parent's policy. It can change an inherited entry only where the parent allowed it with `ZX_POL_OVERRIDE_ALLOW`. So the entries the root job locks bound the whole system. In the same way, the job that component manager creates for a component instance bounds that instance's subtree. Figure 7 shows the shape.

B. Jobs

A *job* is a container and a policy boundary. It has no address space and runs no code of its own. A job is created from a parent job with `zx_job_create(parent, options, &out)`. The call needs the `ZX_RIGHT_MANAGE_JOB` right on the parent job handle. A job keeps track of its children. It reports resource use summed over its subtree, such as peak memory, through `zx_object_get_info`. It is also the unit of mass termination: `zx_task_kill` on a job tears down every process and thread beneath it. Jobs also hold *critical* relationships (§VI-E). Finally, a job holds the *timer-slack* policy, which sets how timers created under it are coalesced by default.

C. Processes

A *process* is the unit of memory isolation and of capability holding. It has two defining parts. The first is a *root VMAR*, the top of its tree of virtual address regions (§IX). The second is a *handle table*, which maps the small integer handle values the process sees to kernel objects and their rights. A process is created with `zx_process_create(job, name, name_size, options, &proc, &vmar)`. The call returns a process handle and a handle to the root VMAR. It does *not* start execution. Figure 8 shows the anatomy.

The kernel does not load ELF files. Process startup happens in user space. A program loader, running in another process, does the work:

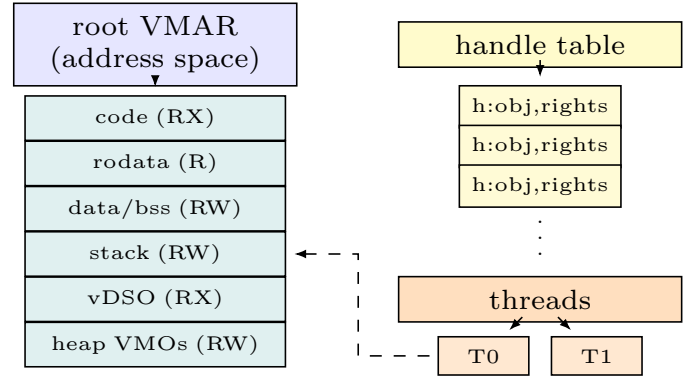


Fig. 8. Process anatomy. The root VMAR holds the code, data, stack and vDSO mappings. The handle table binds integer handles to objects and rights. One or more threads each have their own stack mapping.

- 1) It maps the program's segments (or its dynamic linker, `ld.so`), the vDSO, and an initial stack into the new VMAR.
- 2) It creates the first thread.
- 3) It calls `zx_process_start`.

That call moves a single *bootstrap* handle into the new process and starts execution at the given entry point. The bootstrap handle is usually a channel endpoint. The channel holds the program's namespace, arguments, environment and further handles. Listing 4 sketches the sequence.

Listing 4. Bringing up a process (the loader runs in user space).

```
zx_handle_t proc, vmr, thread;
// Address space + handle table, no threads yet:
zx_process_create(job, "app", 3, 0, &proc, &vmr);

// (User space) map ELF PT_LOAD segments and the vDSO
// into 'vmr', build an initial stack at 'sp', pick 'entry'.
zx_thread_create(proc, "initial", 7, 0, &thread);

// zx_process_start transfers exactly one handle ('bootstrap')
// into the new process and starts 'thread' at 'entry'.
zx_process_start(proc, thread, entry, sp, bootstrap_chan, vdso_base);
```

D. Threads

A *thread* is the unit the scheduler runs. It owns architectural register state and a user stack. It lives inside exactly one process and shares that process's address space and handle table with its sibling threads. A thread is made with `zx_thread_create(process, name, name_size, options, &out)`. It is launched with `zx_thread_start(handle, entry, stack, arg1, arg2)`, which sets the program counter and the stack pointer and passes two register-width arguments. All other registers start at zero. The first thread of a process differs in one way only: `zx_process_start` starts it, not `zx_thread_start`. Later threads follow the ordinary path, including those that a C library's `thr_create` or `pthread_create` creates.

E. Lifecycle, States, and Termination

A thread moves through these states: NEW → RUNNING ↔ BLOCKED ↔ SUSPENDED → DYING → DEAD. A blocked thread waits on a futex, a channel, a port, an interrupt, the pager, or a sleep. A process has a coarser progression: NEW → RUNNING → DYING → DEAD. It becomes RUNNING when its first thread starts. It becomes DEAD when its last thread exits or when it is killed. Termination can be observed. A task

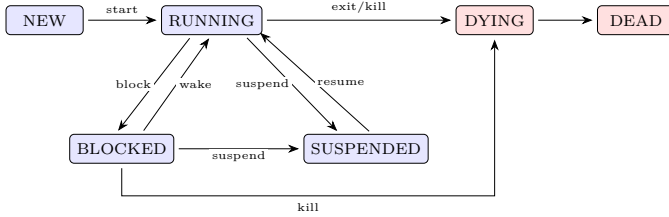


Fig. 9. Thread state machine. A thread moves between RUNNING, BLOCKED and SUSPENDED until it exits or is killed. It then passes through DYING to DEAD, and reaching DEAD asserts ZX_THREAD_TERMINATED.

asserts the ZX_TASK_TERMINATED signal when it reaches DEAD, which releases any thread blocked in `zx_object_wait_one` and any wait registered on a port. A task also has a *return code*, which `zx_object_get_info` reads with the ZX_INFO_PROCESS topic. Killing a task with `zx_task_kill` sets a synthetic return code. Figure 9 draws the thread state machine.

A *critical* relationship lets a job declare that the death of one of its processes is fatal to the whole job. `zx_job_set_critical(job, options, process)` marks process as critical to job, which must be its parent or another ancestor. If that process dies, the kernel kills the job and its subtree, as if by `zx_task_kill`. The job’s return code is then ZX_TASK_RETCODE_CRITICAL_PROCESS_KILL. With the option ZX_JOB_CRITICAL_PROCESS_RETCODE_NONZERO, the job dies only if the process exits with a non-zero code. This is how the failure of an essential service can be passed up the tree, all the way to a controlled reboot.

F. Job Policy: the Sandboxing Substrate

Job policy is Zircon’s first line of process confinement. Each basic policy entry pairs a *condition* with an *action*. The condition is an operation that a process in the job might attempt. The action is what the kernel does when a process attempts it. The conditions fall into three groups:

- Handle misuse: ZX_POL_BAD_HANDLE and ZX_POL_WRONG_OBJECT.
- Memory that is both writable and executable: ZX_POL_VMAR_WX and ZX_POL_AMBIENT_MARK_VMO_EXEC.
- Object creation: ZX_POL_NEW_VMO, ZX_POL_NEW_CHANNEL, ZX_POL_NEW_PROCESS and others, or ZX_POL_NEW_ANY for all of them.

The actions rise in severity:

- ZX_POL_ACTION_ALLOW lets the operation proceed.
- ZX_POL_ACTION_DENY fails the syscall with ZX_ERR_ACCESS_DENIED.
- The _EXCEPTION variants of both also raise a policy exception that a debugger can see.
- ZX_POL_ACTION_KILL terminates the offending process.

Policy is set with `zx_job_set_policy`, under the topic ZX_JOB_POL_BASIC_V2 or the timer-slack topic ZX_JOB_POL_TIMER_SLACK. The *options* argument chooses absolute or relative application. Each entry’s flag, ZX_POL_OVERRIDE_ALLOW or _DENY, decides whether child jobs may change that entry. Listing 5 confines a subtree. Section XIX builds this into the full sandbox. For now, the point is that the job tree *is* the policy tree.

Listing 5. Tightening policy on a child job.

```

zx_policy_basic_v2_t pol[] = {
  { ZX_POL_NEW_PROCESS, ZX_POL_ACTION_DENY, ZX_POL_OVERRIDE_DENY },
  { ZX_POL_BAD_HANDLE, ZX_POL_ACTION_KILL, ZX_POL_OVERRIDE_DENY },
  { ZX_POL_VMAR_WX, ZX_POL_ACTION_DENY, ZX_POL_OVERRIDE_DENY },
};
zx_job_set_policy(job, ZX_JOB_POL_RELATIVE,
                  ZX_JOB_POL_BASIC_V2, pol, 3);

```

G. Debugger Support

The `zxdb` debugger is built on the same task calls. A debugger suspends a thread, or a whole process, with `zx_task_suspend`, which returns a *suspend token*. The task stays suspended until every such token is closed, so several debuggers can suspend it without interfering. While the task is suspended, the debugger reads and writes its registers with `zx_thread_read_state` and `zx_thread_write_state`. A *kind* argument selects the register set: ZX_THREAD_STATE_GENERAL_REGS, _FP_REGS, _VECTOR_REGS, _DEBUG_REGS (hardware breakpoints and watchpoints), or _SINGLE_STEP. Faults and policy violations arrive through *exception channels*. `zx_task_create_exception_channel` creates one on a thread, a process or a job. A debugger therefore gets a chance to handle a fault at each level, before the fault moves up the tree.

Exceptions are delivered in a fixed order along the task hierarchy. A thread may take a hardware fault, such as a page fault or an illegal instruction, or it may break a job policy. The kernel then offers the exception to these channels in turn:

- 1) The process’s *debugger* channel.
- 2) The *thread* channel.
- 3) The *process* channel.
- 4) The debugger channel again, if the exception is second-chance.
- 5) The *job* channels, from the parent job up toward the root.

Each handler receives a message that names the faulting thread and the type of exception. The handler may inspect and rewrite registers or memory. It then either resolves the exception (ZX_EXCEPTION_STATE_HANDLED) or passes it on (_TRY_NEXT). A debugger that wants a last look before the thread dies selects the *second-chance* strategy (ZX_EXCEPTION_STRATEGY_SECOND_CHANCE). The exception then returns to it after the process handler declines, and before any job handler sees it. If no handler resolves the fault, the kernel terminates the process with the return code ZX_TASK_RETCODE_EXCEPTION_KILL, and a crash report is generated.

H. Introspection and Resource Accounting

Every task can be observed through `zx_object_get_info`, the uniform query call. A *topic* argument selects what it returns:

- ZX_INFO_PROCESS reports whether a process has started or exited, and its return code.
- ZX_INFO_PROCESS_THREADS, ZX_INFO_JOB_CHILDREN and ZX_INFO_JOB_PROCESSES return the KOIDs of a task’s children, so a supervisor can walk the tree it built.
- ZX_INFO_TASK_STATS and the maps and VMOs topics report a task’s memory footprint.
- ZX_INFO_TASK_RUNTIME returns a `zx_info_task_runtime_t` of times in nanoseconds.

The runtime record holds the time the task spent running on a CPU (`cpu_time`) and the time it spent runnable but waiting in a run queue (`queue_time`). Recent ABIs add the time lost

TABLE V
TASK-MANAGEMENT SYSTEM CALLS, GROUPED BY OBJECT.

Scope	Syscall	Purpose
Job	<code>zx_job_create</code>	make child job under a parent
	<code>zx_job_set_policy</code>	install basic/timer-slack policy
	<code>zx_job_set_critical</code>	escalate a process death to job
Process	<code>zx_process_create</code>	address space + handle table
	<code>zx_process_start</code>	start first thread; pass bootstrap
	<code>zx_process_read/</code> <code>write_memory</code>	peek/poke process memory (debugger / loader)
Thread	<code>zx_thread_create</code>	make a thread in a process
	<code>zx_thread_start</code>	set PC/SP, begin execution
	<code>zx_thread_read/</code> <code>write_state</code>	get/set register banks (general/FP/vector/debug)
Any	<code>zx_task_kill</code>	terminate task (and subtree)
	<code>zx_task_suspend</code>	suspend; returns suspend token
	<code>zx_task_create_</code> <code>exception_channel</code>	obtain exception channel for debugging/policy faults

to page faults and to lock contention. User-space profilers read these counters. So does the tuning loop that assigns scheduling profiles (§VII-E). Listing 6 reads a thread’s CPU and queue time. Table V summarizes the task-management calls.

Listing 6. Reading per-task CPU and queue time.

```
zx_info_task_runtime_t rt;
zx_object_get_info(thread, ZX_INFO_TASK_RUNTIME,
                  &rt, sizeof(rt), NULL, NULL);
// rt.cpu_time : ns actually executed on a CPU
// rt.queue_time : ns runnable but waiting to be dispatched
```

VII. THREAD SCHEDULING

The scheduler decides which thread runs on which CPU, and for how long. Zircon has two scheduling disciplines, and both run on every CPU.

- A *fair* scheduler uses weighted fair queuing in a virtual-time domain. It gives every thread a share of the CPU in proportion to its weight.
- A *deadline* (bandwidth) scheduler grants fixed, periodic CPU reservations to work with tight timing needs. It runs that work ahead of all fair threads.

User space controls both through one kernel object, the *profile*. Both also work with priority inheritance, which keeps an important thread from stalling behind a less important thread that holds a lock. This section explains each part in turn [13], [21], [22].

A. Per-CPU Run Queues and Affinity

Scheduling is *per-CPU*. Each CPU owns its run queues and makes its own dispatch decisions. There is no single global ready list for the CPUs to contend on. A thread may run only on the CPUs allowed by its *affinity mask*. Within that set, the system balances load in two ways. An idle CPU migrates, or *steals*, threads from busier CPUs. A thread that becomes runnable is placed on a CPU chosen to balance load while keeping its cache warm.

Affinity comes in two kinds. A *hard* affinity mask is a strict rule: the thread runs only on those CPUs. A *soft* affinity mask is a preference. The load balancer tries to honor it, but may override it rather than leave a thread unable to run or a CPU idle. The usual use of hard affinity is to pin a real-time thread to a dedicated set of CPUs.

run queue (ordered by virtual finish time f_i)

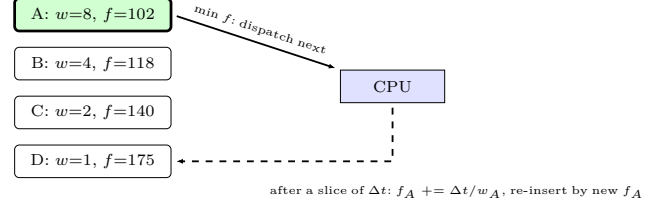


Fig. 10. Fair-scheduler dispatch. The run queue is ordered by virtual finish time f_i , and the scheduler runs the thread with the smallest f . After a slice, that thread’s f grows by an amount inversely proportional to its weight, and the thread goes back into the queue. Over time each thread gets CPU in proportion to its weight.

B. From Priorities to a Fair Scheduler

Early Zircon used a conventional priority scheduler. It had 32 priority levels (`ZX_PRIORITY_LOWEST` = 0, `ZX_PRIORITY_DEFAULT` = 16, `ZX_PRIORITY_HIGHEST` = 31), and the runnable thread with the highest priority always won. Strict priority is simple, but it has three problems. It starves low-priority threads. It folds “importance” and “latency” into one number. And it cannot share the CPU in proportion. Zircon replaced it with a *fair scheduler* modeled on weighted fair queuing (WFQ). In it, the old priority becomes a *weight*, and the scheduler compares threads in virtual time instead of by raw priority.

C. Fair-Scheduler Mechanics

Each fair thread has a *weight* w_i , derived from the base priority in its profile. Weights set the *proportion* of CPU a thread receives when threads compete. Let the runnable threads on a CPU have total weight $W = \sum_j w_j$. Then a thread of weight w_i is entitled to the fraction w_i/W of that CPU’s time.

The scheduler achieves this with a per-thread *virtual timeline*. While a thread runs, its virtual time advances at a *normalized rate* that is inversely proportional to its weight. A heavy thread’s virtual clock ticks slowly and a light thread’s ticks fast. So the same slice of wall-clock time costs a light thread more virtual time. Each runnable thread also has a *virtual finish time*: the virtual time at which it would finish its current normalized time slice. The scheduler runs the thread with the **earliest virtual finish time**. It takes that thread from a run queue kept in a balanced tree, at a cost of $O(\log n)$. Figure 10 shows the queue.

Two properties follow from this design.

- *Proportional sharing*. While the set of runnable threads stays the same, each thread’s share of CPU time converges to w_i/W . The weights set the rate at which each virtual finish time advances.
- *No starvation*. Even the lightest thread has a finite virtual finish time, which in time becomes the smallest. So that thread is sure to run. Under strict priority, a thread can be put off forever.

A *time slice* bounds how long a thread keeps the CPU before the scheduler looks at the queue again. The scheduler starts from a target scheduling latency, divides it among the runnable threads, and applies a minimum granularity so context switches stay affordable. The slice therefore also bounds how long a newly woken thread waits to run.

Here is a concrete case. Four CPU-bound threads share one CPU, with weights 8, 4, 2, 1 (total $W = 15$). Over a window

long enough to average out the slices, they receive $\frac{8}{15}, \frac{4}{15}, \frac{2}{15}, \frac{1}{15}$ of the CPU. That is roughly 53%, 27%, 13% and 7%. A strict-priority scheduler would instead give everything to the first thread. If the heaviest thread's weight doubles to 16, the total becomes $W = 23$. That thread then gets $\frac{16}{23} \approx 70\%$, and every other thread still makes progress. Weights grow with profile priority, so “a higher-priority thread” still means something. It means a larger share of the CPU and an earlier place in the virtual-time queue, not the exclusive right to run.

D. Deadline (Bandwidth) Scheduling

Some work needs a different guarantee from a fair share. An audio mixer must produce a buffer every few milliseconds. A graphics pipeline must finish before vsync. For such work Zircon offers *deadline scheduling*. A thread requests a reservation as a triple (C, D, P) : a *capacity*, a *relative deadline* and a *period*. All three are durations, with $C \leq D \leq P$. The scheduler guarantees the thread C nanoseconds of CPU in every period of P nanoseconds. It schedules those C nanoseconds to finish within D of the start of the period.

Deadline threads are dispatched **earliest deadline first (EDF)**. As a class, they take *absolute precedence over all fair threads*. A CPU runs its eligible deadline threads in order of absolute deadline. It turns to the fair run queue only when no deadline thread is eligible.

The guarantee holds only while the total demand on a CPU is feasible, that is $\sum C_k/P_k \leq 1$. The kernel does not reject reservations that break this bound. If total deadline demand on a CPU exceeds 100%, deadlines will be missed. The system then behaves as though the periods of some or all deadline threads were stretched by the overload factor.

E. Profiles: the User-Visible Control

User space never sets weights or deadlines as numbers on a thread handle. Instead it obtains a *profile* and applies it. A profile is a kernel object of type `ZX_OBJ_TYPE_PROFILE` that describes a scheduling intent. There are two ways to get one. The privileged `ProfileProvider` service makes one from a role string, over FIDL. Or a caller that holds the profile resource calls `zx_profile_create` directly.

The profile holds a `zx_profile_info_t`. Its `flags` field selects at most one scheduling discipline:

- a base *priority* (`ZX_PROFILE_INFO_FLAG_PRIORITY`), or
- a deadline reservation (`_FLAG_DEADLINE`), with the fields `capacity`, `relative_deadline` and `period`.

It can also set a CPU *affinity* mask (`_FLAG_CPU_MASK`).

A thread applies a profile with `zx_object_set_profile(handle, profile, 0)`. The target `handle` must be a thread (or a VMAR, for memory priority) and must have `ZX_RIGHT_MANAGE_THREAD`. The `profile` handle must have `ZX_RIGHT_APPLY_PROFILE`.

One service manages the profiles, so the platform can tune scheduling for each role in one place. Components do not hard-code the numbers. They request a named *role*, a string such as `fuchsia.media.audio.output`. The `ProfileProvider` looks the role up in a board-specific configuration that maps it to a priority, a deadline and an affinity. So the same binary gets suitable timing on very different hardware.

A profile may instead set a *memory* priority (`ZX_PROFILE_INFO_FLAG_MEMORY_PRIORITY`), which is applied to a VMAR. It asks the kernel to keep the region's pages resident rather than reclaim them. That is why `zx_object_set_profile`

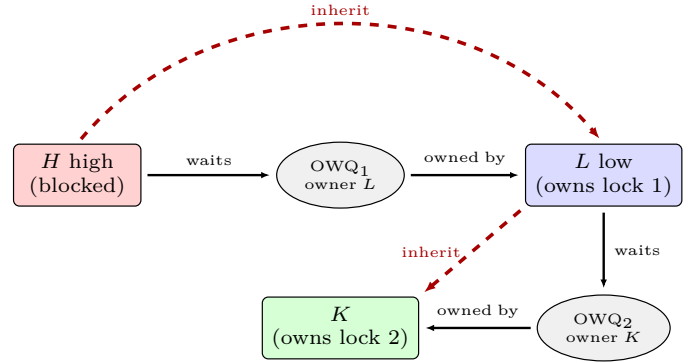


Fig. 11. Priority inheritance along a blocking chain. H blocks on an OwnedWaitQueue owned by L , and L is blocked on a queue owned by K . The kernel passes H 's profile on to L and then to K , so the thread that actually holds the lock is boosted, runs, and releases it.

accepts a VMAR handle as well as a thread handle. Listing 7 shows the two scheduling forms.

Listing 7. A priority profile and a deadline reservation.

```

// Fair: a base-priority profile.
zx_profile_info_t pi = { .flags = ZX_PROFILE_INFO_FLAG_PRIORITY };
pi.priority = 20; // above ZX_PRIORITY_DEFAULT
zx_profile_create(profile_rsrc, 0, &pi, &prof);
zx_object_set_profile(thread, prof, 0); // needs APPLY_PROFILE

// Deadline: 2 ms of CPU every 10 ms, due within 8 ms.
zx_profile_info_t dl = { .flags = ZX_PROFILE_INFO_FLAG_DEADLINE };
dl.deadline_params.capacity = ZX_MSEC(2);
dl.deadline_params.relative_deadline = ZX_MSEC(8);
dl.deadline_params.period = ZX_MSEC(10);
zx_profile_create(profile_rsrc, 0, &dl, &dprof);
zx_object_set_profile(audio_thread, dprof, 0);
  
```

F. Priority Inheritance and the OwnedWaitQueue

Both disciplines can suffer *priority inversion*. Suppose a high-importance thread H blocks on a futex or mutex held by a low-importance thread L . An unrelated medium thread M , runnable and heavier than L , then preempts L . Now H is in effect waiting for M .

Zircon's defense is the kernel's `OwnedWaitQueue`, a wait queue that knows its current *owner*. When threads block on an owned wait queue, the kernel passes the waiters' *inherited profile values* to the owner. The owner is then scheduled *as if* it had the importance of its most important waiter. This works for both disciplines. A fair waiter adds its weight to the owner's effective weight. A deadline waiter can lift the owner into the deadline class, so the owner runs ahead of fair threads long enough to release the resource.

The boost follows the *blocking chain*. If L is itself blocked on a queue owned by K , the boost flows from H through L to K . It unwinds as each lock is released. The kernel learns the owner through the same calls that block. The main one is the futex's `new_futex_owner` argument (§VIII-B), so user-space mutexes get inheritance with no extra work. Figure 11 shows a chain of two hops.

G. Preemption and Cross-CPU Reschedule

A running thread is preempted in two cases: a more eligible thread becomes runnable, or its time slice runs out. A per-CPU preemption timer bounds how long a thread runs before the scheduler decides again. The kernel sets that timer from the running thread's time slice or from the arrival of a deadline

TABLE VI
SCHEDULING SYSTEM CALLS AND PROFILE FIELDS.

Syscall / field	Meaning
<code>zx_profile_create</code>	make a profile object; needs the profile resource
<code>zx_object_set_profile</code>	apply profile to thread/VMAR
<code>zx_thread_legacy_yield</code>	(deprecated) voluntarily yield the CPU
<code>zx_nanosleep</code>	block until a monotonic deadline
<i>zx_profile_info_t (selected by flags)</i>	
<code>priority</code>	fair weight source, 0...31
<code>deadline_params</code>	{ <code>capacity</code> , <code>relative_deadline</code> , <code>period</code> } (all durations, $C \leq D \leq P$)
<code>cpu_affinity_mask</code>	allowed CPU set (hard)

thread. Most preemptions, though, are caused by events. A wake or a profile change can make another thread the new minimum-*f*, or make a deadline thread eligible, and that triggers a reschedule.

Inside the kernel, short critical sections disable preemption. Spinlocks go further and also disable interrupts. This keeps the scheduler from switching away while a per-CPU invariant is briefly broken. The reschedule then happens at the next preemption point.

Sometimes the newly runnable thread belongs on a *different* CPU, because of affinity or load balancing. The local CPU then sends an *inter-processor interrupt* (IPI) that asks the remote CPU to reschedule. Zircon has two kinds of reschedule. An *eager* one sends the IPI at once, for example for a freshly woken deadline thread that must preempt immediately. A *lazy* one defers the IPI to the next natural scheduling point and merges it with others. Lazy reschedules trade a little latency for fewer cross-CPU interrupts.

H. Idle and Power

When a CPU has no runnable thread, it runs its own *idle thread*. The idle thread puts the core into an idle, low-power state of the architecture, such as `wfi` on ARM or `hlt` and the `MWAIT` C-states on x86. The core stays there until an interrupt wakes it: a timer, a device IRQ, or a reschedule IPI from another CPU. A deeper idle state saves more power but takes longer to wake from. The choice is made together with the timer subsystem, so a CPU with no timer deadline coming soon can sleep more deeply.

The kernel can also *deactivate* a CPU entirely, for power management or hotplug. The scheduler then moves the CPU's threads to the remaining active CPUs, and reactivates the CPU when it is needed.

The per-task runtime counters `cpu_time` and `queue_time` (§VI-H) help diagnose latency. Rising queue time means the thread is starved of CPU share. Rising CPU time means the thread simply runs a long computation. Those measurements then guide which profile the thread should get. Table VI lists the scheduling system calls and profile fields.

VIII. SYNCHRONIZATION PRIMITIVES

Zircon's synchronization follows a rule seen all through the kernel: on the fast path, stay out of the kernel. The main primitive user space sees is the *futex*. When there is no contention, a futex operation is a single atomic instruction in user space. The thread enters the kernel only when it must actually block or wake another thread. The C and C++ runtimes build mutexes, condition variables and one-shot completions on

top of the futex. The kernel protects its own data structures with its own locks: spinlocks, blocking mutexes, a big reader-writer lock, and seqlocks. This section covers all of these. It also ties futex ownership back to the priority inheritance of the scheduler (§VII-F) [1], [12], [22].

A. The Futex

A *futex* ("fast userspace mutex") is not a kernel object that you allocate. It is a 32-bit, naturally aligned integer in ordinary user memory, and the kernel uses its address as a key. The work is split between user space and the kernel. User space locks and unlocks with an atomic compare-and-swap (CAS) on the integer. When there is no contention the CAS succeeds and **the kernel is never entered**. A thread makes a system call only in two cases: it finds the lock already held, or on unlock it finds that waiters may exist. The three core calls are `zx_futex_wait`, `zx_futex_wake` and `zx_futex_requeue` (Listing 8).

Listing 8. Futex syscall signatures.

```
zx_status_t zx_futex_wait(const zx_futex_t* value_ptr,
                        zx_futex_t current_value,
                        zx_handle_t new_futex_owner,
                        zx_instant_mono_t deadline);
zx_status_t zx_futex_wake(const zx_futex_t* value_ptr,
                        uint32_t wake_count);
zx_status_t zx_futex_requeue(const zx_futex_t* value_ptr,
                        uint32_t wake_count, zx_futex_t current_value,
                        const zx_futex_t* requeue_ptr,
                        uint32_t requeue_count,
                        zx_handle_t new_requeue_owner);
```

The wait is the delicate part. A thread reads the futex word, sees it locked, and decides to block. Before the kernel has queued that thread, the owner might release the lock and issue its wake. If the waiter were queued only after that, the wake would be lost. The `current_value` argument closes this race. `zx_futex_wait` checks that `*value_ptr` still equals `current_value`, and queues the caller in the same critical section. If the value has already changed, the call returns `ZX_ERR_BAD_STATE` at once without blocking, and the caller retries its CAS. This check-then-block step is what makes the lock-free fast path safe.

The other two calls act on the queue. `zx_futex_wake` releases up to `wake_count` waiters on a key. `zx_futex_requeue` does the same, and also moves up to `requeue_count` of the remaining waiters to a second futex without waking them. A condition variable uses it to move its waiters onto the mutex, which avoids a *thundering herd* of threads that all wake at once. Figure 12 compares the two paths.

B. Owner Tracking and Priority Inheritance

A futex can name the thread that currently *owns* the lock it guards. `zx_futex_wait` takes a `new_futex_owner` handle, and `zx_futex_requeue` takes a `new_requeue_owner`. Passing the owning thread's handle registers that thread as the owner of the kernel's wait queue, an *OwnedWaitQueue* (§VII-F). This is how the scheduler applies *priority inheritance* to user-space locks. Suppose a high-importance thread blocks on a futex owned by a low-importance thread. The kernel boosts the owner to the waiter's effective profile, so the owner runs and releases the lock promptly. Passing `ZX_HANDLE_INVALID` marks the futex as having no owner for now, which suits a condition variable, for example.

Two more calls work with ownership. `zx_futex_get_owner` reads the current owner. `zx_futex_wake_single_owner` wakes exactly one waiter and makes it the owner, which is how a mutex is handed to the next waiter.

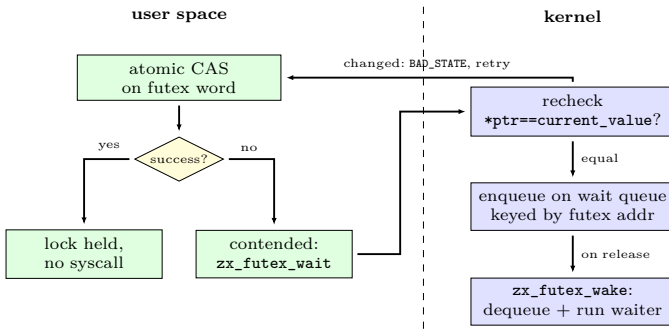


Fig. 12. Futex paths. An uncontended lock or unlock is a CAS in user space, with no system call. Under contention the waiter traps into the kernel. The kernel checks the word against `current_value` and, in the same step, queues the waiter on a wait queue keyed by the futex address. A later `zx_futex_wake` on release takes it off the queue.

C. Building Higher-Level Primitives

User-space concurrency libraries are thin layers over the futex. A C++ `std::mutex` is a futex word with three states: unlocked, locked without waiters, and locked with waiters.

- `lock()` tries a CAS from unlocked to locked. Only if that fails does it mark the word as contended and call `zx_futex_wait`, naming the holder as the owner.
- `unlock()` releases the word. Only if the contended state was set does it call `zx_futex_wake_single_owner`.

A `std::condition_variable` is a futex used for waiting. On notify it calls `zx_futex_requeue` to move its waiters onto the associated mutex, so at most one thread actually wakes to contend for the mutex.

Fuchsia's C library `libsnc` provides the same building blocks for the system's C code. They include `sync_mutex_t`, `sync_condition_t`, and the one-shot `sync_completion_t`. A completion is a latch built on a futex. Many threads can block in `sync_completion_wait` until one call to `sync_completion_signal` releases them all. It is the usual way to wait until initialization has finished. Listing 9 shows the three-state mutex. The common acquire and release are one atomic operation each, and the system calls appear only on the contended branches.

Listing 9. A futex-backed mutex (fast path is lock-free).

```

// states: 0=unlocked, 1=locked, 2=locked+waiters
// owner: handle of the thread holding the lock (for PI)
void lock(atomic_int* m, zx_handle_t owner) {
    int c = 0;
    if (atomic_compare_exchange_strong(m, &c, 1)) return; // uncontended
    if (c != 2) c = atomic_exchange(m, 2); // mark contended
    while (c != 0) { // contended slow path
        zx_futex_wait(m, 2, owner, ZX_TIME_INFINITE);
        c = atomic_exchange(m, 2);
    }
}

void unlock(atomic_int* m) {
    if (atomic_fetch_sub(m, 1) != 1) { // was 2: waiters present
        atomic_store(m, 0);
        zx_futex_wake_single_owner(m);
    }
}

```

D. Kernel-Internal Locks

Inside the kernel there is no user memory to share and no fast path to protect, so the kernel does not use futexes. It uses a small family of locks, chosen by context.

- A *spinlock* protects very short critical sections. Taking it disables interrupts and preemption on the local CPU, and

a waiting CPU busy-waits. It must never be held across a blocking operation. It is the only lock that can be used from interrupt context.

- The kernel `Mutex` is a blocking lock for longer sections. A thread that cannot take it waits on an `OwnedWaitQueue`, and the holder inherits the waiter's profile. This is the priority inheritance described above, applied inside the kernel.
- The big reader-writer lock `BrwLock` guards data that is read far more often than written. It allows many readers at once or one writer. It comes in a priority-inheriting variant, which boosts the threads a blocked writer waits for, and in a plain blocking variant.

The `BrwLock` keeps its accounting per CPU, so uncontended readers rarely touch a shared cache line. That makes it cheap enough to guard hot, read-mostly structures such as the VM's mapping tables.

Seqlocks (sequence locks) protect data that readers must see without ever blocking writers. A writer increments a sequence counter to an odd value, writes the payload, and then increments the counter again to an even value. A reader reads the counter before and after reading the payload. It retries if the two values differ or the first was odd (Listing 10). Readers never block writers and use no atomic operation on the data itself. This is what the kernel needs to publish its time and clock parameters in the read-only vDSO data page. User space reads them without a lock and computes the current time without a system call.

Listing 10. Seqlock read side (lockless, retrying).

```

uint64_t read_time_params(const seqlock_t* s) {
    uint64_t v; uint32_t seq;
    do {
        seq = atomic_load_acquire(&s->seq); // even when stable
        v = s->ticks_to_mono_scale; // read payload
        atomic_thread_fence(memory_order_acquire);
    } while (seq & 1 || seq != atomic_load(&s->seq));
    return v; // consistent snapshot
}

```

A kernel that takes many locks can deadlock if two paths take them in different orders. To catch this, a build can enable `lockdep`, a lock-ordering checker that runs in the kernel. It records the order in which each class of lock is taken and reports any cycle before it can turn into a real deadlock on a shipped device.

E. Events and Signals as Coarse Synchronization

Not all coordination needs a lock. Zircon's *event* and *event-pair* objects exist only to hold *signals*. A thread, or the kernel, raises a user signal with `zx_object_signal`. For the far end of an eventpair it uses `zx_object_signal_peer`. Waiters see the signal through `zx_object_wait_one`, or through `zx_object_wait_async` on a port.

This is coarser than a futex. There is no atomic value and no owner, and every wait and every signal enters the kernel. In exchange it fits the rest of the object system. The same wait calls that watch a channel become readable or a task terminate also watch an event. So a single port can collect wakeups of many different kinds. Signals, ports and asynchronous waits are covered fully in §XIV. Here it is enough to know that `ZX_EVENT_SIGNALED` on an event, and the user signal bits on an eventpair, give a simple meeting point run by the kernel. It works where the futex's shared-memory model does not, for example between two processes that share a handle but no memory. Table VII lists the synchronization calls.

TABLE VII
FUTEX AND RELATED SYNCHRONIZATION SYSTEM CALLS.

Syscall	Semantics
<code>zx_futex_wait</code>	block only if <code>*ptr==current_value</code> ; can name the owner for PI
<code>zx_futex_wake</code>	wake up to <code>wake_count</code> waiters
<code>zx_futex_wake_</code> <code>single_owner</code>	wake one waiter and transfer futex ownership to it
<code>zx_futex_requeue</code>	wake some, move others to a second futex (avoid thundering herd)
<code>zx_futex_get_owner</code>	read the futex's current owner
<code>zx_event_create</code>	make an event (signal carrier)
<code>zx_eventpair_create</code>	make a connected signal pair
<code>zx_object_signal</code>	raise/clear user signals on self
<code>zx_object_signal_peer</code>	raise signals on eventpair peer
<code>zx_object_wait_one</code>	block until a signal is asserted

IX. VIRTUAL MEMORY: VMOs AND VMARs

Zircon splits virtual memory into two separate objects. A *Virtual Memory Object* (VMO) is a container of physical pages. It is the unit of memory itself, with no address attached. A *Virtual Memory Address Region* (VMAR) is a span of a process's address space. It is the unit of *placement*, with no content attached. Mapping is the explicit step that binds a range of a VMO into a VMAR, at some virtual address and with some permissions. This split mirrors the handle and object split used in the rest of the kernel [1]. A handle with rights names a VMO. The same VMO handle can be mapped into many address spaces, or sent over a channel to share memory between processes.

For an RTL engineer the model is simple. A VMO is a block of addressable storage, like a BRAM or a DRAM region. A VMAR is the address decoder that places that storage on one process's bus. Two processes can decode the same storage at different addresses.

A. The VMO: a Resizable Container of Pages

A VMO is an ordered array of bytes, managed in whole pages. All sizes and offsets are multiples of the hardware page size, which is 4KiB on the supported 64-bit architectures. `zx_vmo_create(size, options, &out)` reserves a logical size but commits *no* physical pages. A fresh anonymous VMO reads as an array of zeros. The kernel takes frames from the physical memory manager only when a page is first touched or explicitly committed. So a process can create a sparse VMO of several gigabytes almost for free and fill it in lazily.

Zircon has several *kinds* of VMO [6]:

- **Anonymous or paged** (the default from `zx_vmo_create`): zero-filled RAM, committed on demand. It backs heaps, stacks, BSS and scratch buffers.
- **Physical** (`zx_vmo_create_physical`): a VMO whose pages are a fixed range of physical addresses, typically device MMIO apertures or framebuffers. It is created from a *resource* capability (§XVII), because it hands out raw physical memory. Its cache policy almost always needs changing (see below).
- **Contiguous** (`zx_vmo_create_contiguous`): anonymous RAM that the PMM guarantees to be *physically contiguous*. It is allocated against a Bus Transaction Initiator (BTI) and aligned to a power of two that the caller chooses. DMA engines without scatter-gather, or with few descriptors, need this contiguity.
- **Pager-backed**: a VMO whose pages come on demand from a user-space *pager*, instead of being zero-filled by the kernel.

`zx_pager_create_vmo` creates them. They are the basis of file-backed memory and of executable loading, and Section X covers them.

1) *VMO Operations*: `zx_vmo_read` and `zx_vmo_write` copy bytes between a caller buffer and a VMO range, with no mapping needed. They suit control structures and small transfers. They are also the only way to touch a VMO you hold but have not mapped. `zx_vmo_get_size` reports the current size. `zx_vmo_set_size` grows or shrinks the VMO. It needs the handle to hold `ZX_RIGHT_RESIZE`, which is granted only if the VMO was created with the `ZX_VMO_RESIZABLE` option. Shrinking a VMO decommits and frees the cut-off pages.

`zx_vmo_op_range(handle, op, offset, size, buffer, buffer_size)` is one call for many maintenance operations. The selector `op` picks one of these:

- `ZX_VMO_OP_COMMIT` fills in pages now, the eager opposite of demand paging.
- `ZX_VMO_OP_DECOMMIT` returns pages to the PMM and leaves them reading as zero.
- `ZX_VMO_OP_ZERO` zeroes a range, decommitting pages where it can.
- `ZX_VMO_OP_CACHE_SYNC`, `_CACHE_INVALIDATE`, `_CACHE_CLEAN` and `_CACHE_CLEAN_INVALIDATE` run the architecture's cache-line clean and invalidate instructions. They are needed before or after DMA on paths that are not cache-coherent.
- The hints `ZX_VMO_OP_ALWAYS_NEED`, `_DONT_NEED` and `_PREFETCH` bias the reclamation machinery.
- `ZX_VMO_OP_LOCK`, `_TRY_LOCK` and `_UNLOCK` pin the pages of a *discardable* VMO against reclamation.

`zx_vmo_set_cache_policy` sets how mappings of the VMO use the cache hierarchy:

- `ZX_CACHE_POLICY_CACHED`: normal write-back RAM.
- `ZX_CACHE_POLICY_UNCACHED` and `ZX_CACHE_POLICY_UNCACHED_DEVICE`: for memory-mapped registers, where every access must reach the device.
- `ZX_CACHE_POLICY_WRITE_COMBINING`: for apertures that are mostly written, such as framebuffers, where the CPU may merge stores.

The policy can be set only while the VMO is unmapped and has no committed pages. It matters mainly for physical VMOs.

2) Children and Copy-on-Write:

`zx_vmo_create_child(parent, options, offset, size, &out)` makes a new VMO from a window of an existing one. The `options` word selects one of four relationships [6]:

- `ZX_VMO_CHILD_SNAPSHOT`: a fixed point-in-time *copy* of the parent's contents, made lazily by copy-on-write (COW). Parent and child share the same physical pages until either side writes. The first write to a shared page gives the writer a private copy. So the snapshot never sees the parent's later changes.
- `ZX_VMO_CHILD_SNAPSHOT_AT_LEAST_ON_WRITE`: the widely used "lazy COW" variant. The child shares the parent's pages and gets a private copy of a page when it *writes* to it. Unlike a full snapshot, it may still see parent writes to pages that neither side has copied yet. This is the cheap, common case. The program loader uses it to give each process a private, writable view of shared read-only ELF data.
- `ZX_VMO_CHILD_SLICE`: not a copy but an *alias*. It is a window into the parent that shares its pages directly, and writes are visible in both directions. A subsystem uses a slice to hand out a bounded part of a VMO without copying storage.

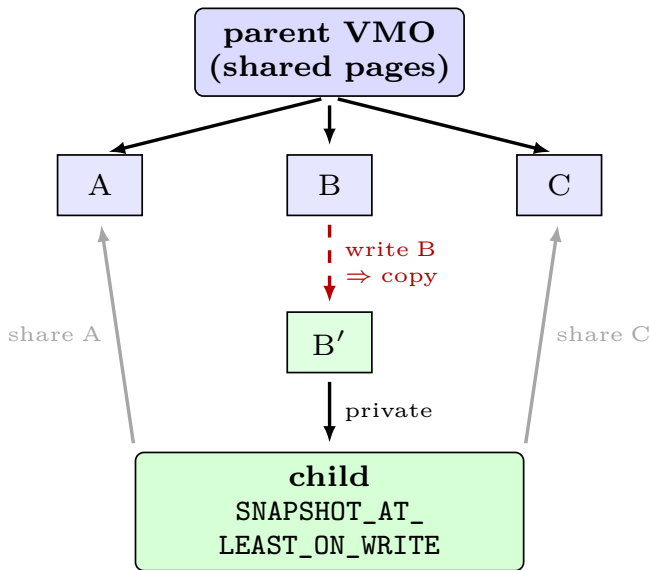


Fig. 13. A copy-on-write clone tree. The child shares the parent’s pages A, B and C by reference. A write to B traps, and the kernel gives the child a private page B’ in that one slot. A and C stay shared. Reads copy nothing; only written pages diverge.

- **ZX_VMO_CHILD_REFERENCE**: a second object, with its own handles, that refers to *all* of the parent’s pages (a full-size alias). It is useful when a separate object identity is needed over the same memory.

Modifier bits combine with these modes. **ZX_VMO_CHILD_RESIZABLE** makes the child resizable on its own, and **ZX_VMO_CHILD_NO_WRITE** makes a read-only child. Repeated cloning builds a *clone tree* (Fig. 13). A hidden parent holds the shared pages, and the leaves diverge one page at a time. The kernel tracks which leaf owns each physical page. It returns a page to the shared parent only when the last private reference to it goes away. This one mechanism serves three uses:

- **fork**-style sharing under Starnix;
- the loader’s sharing of read-only text and data across every process that runs a binary;
- cheap snapshots of large buffers in the graphics and storage stacks.

B. The VMAR: the Address-Space Tree

Every process owns a *root VMAR* that spans its whole user address range. The kernel returns it when the process is created [7]. A VMAR is a tree. **zx_vmar_allocate** carves a child VMAR out of a parent: a contiguous sub-range reserved for later use. Mappings are the leaves of the tree. Subdividing the space does two jobs:

- It groups related mappings under one handle. Destroying a child VMAR unmaps everything inside it in one step.
- It is the unit over which the kernel randomizes placement, for address-space layout randomization (ASLR). When the caller does not fix an address, the kernel picks a random offset in the parent for each child VMAR and mapping. The layout then differs from run to run and from process to process.

zx_vmar_map(vmar, options, vmr_offset, vmo, vmo_offset, len, &addr) binds memory to an address. It maps `len` bytes of `vmo`, starting at `vmo_offset`, into `vmar`,

and returns the chosen virtual address. The `options` word holds the permission bits **ZX_VM_PERM_READ**, **ZX_VM_PERM_WRITE** and **ZX_VM_PERM_EXECUTE**. It also holds modifiers for placement and behavior:

- **ZX_VM_SPECIFIC** asks for a given offset in the VMAR instead of a random one. **ZX_VM_SPECIFIC_OVERWRITE** also replaces whatever was mapped there.
- **ZX_VM_MAP_RANGE** fills in the page-table entries for the whole range at once, instead of on each fault.
- The **ZX_VM_ALIGN_*** bits ask for a larger alignment.
- **ZX_VM_OFFSET_IS_UPPER_LIMIT** bounds placement from above.

A VMAR created with **ZX_VM_CAN_MAP_SPECIFIC**, **ZX_VM_CAN_MAP_READ**, **_WRITE** or **_EXECUTE** limits what its descendants may do. **zx_vmar_protect** changes the permissions of a mapped range, and **zx_vmar_unmap** removes one. **zx_vmar_destroy** removes a whole subtree.

The rights model checks every step. Mapping needs **ZX_RIGHT_MAP** on the VMO handle. Each requested permission also needs the matching VMO right: a **ZX_VM_PERM_WRITE** mapping needs **ZX_RIGHT_WRITE**, and **ZX_VM_PERM_EXECUTE** needs **ZX_RIGHT_EXECUTE**. Only **zx_vmo_replace_as_executable** adds the execute right to a VMO. That call needs a dedicated executable-resource capability, or a job policy that allows it without one. Job policy also enforces $W \oplus X$. When **ZX_POL_VMAR_WX** denies it, no mapping can be both writable and executable. Figure 14 shows the resulting tree. It also shows how one VMO mapped into the VMARs of two processes gives shared memory.

C. Demand Paging and Faults

Mappings are lazy by default. **zx_vmar_map** only records the mapping. The page tables are filled in on first access, unless the caller asked for **ZX_VM_MAP_RANGE**. When the CPU touches an unmapped page inside a valid mapping, it raises a page fault. The kernel resolves the fault according to the backing VMO:

- For an anonymous VMO, the kernel takes a zero-filled frame from the PMM, installs it in the page tables, and resumes the faulting instruction. For a read of a page never written, it may instead share a single zero page for a while.
- For a pager-backed VMO, the kernel cannot make up the contents. It suspends the thread and sends a read request to the user-space pager that owns the VMO. The pager fetches the data and supplies the page, and then the thread continues.

This software pager is a page-miss handler that lives outside the kernel. Section X covers it in detail. An access that breaks the mapping’s permissions, or that falls outside any mapping, is not satisfied. The kernel reports it to the process as an architectural exception, through the exception channel.

D. Constructing a Process Address Space

The sections above say *what* memory objects are and how they map. This one shows *how* a real process address space is built from them, step by step. The kernel does very little of this work. It provides an empty address space and a way to start a thread in it. Everything else is ordinary user-space code that calls the system calls of Section IX-B: loading code, laying out the heap, and linking in shared libraries.

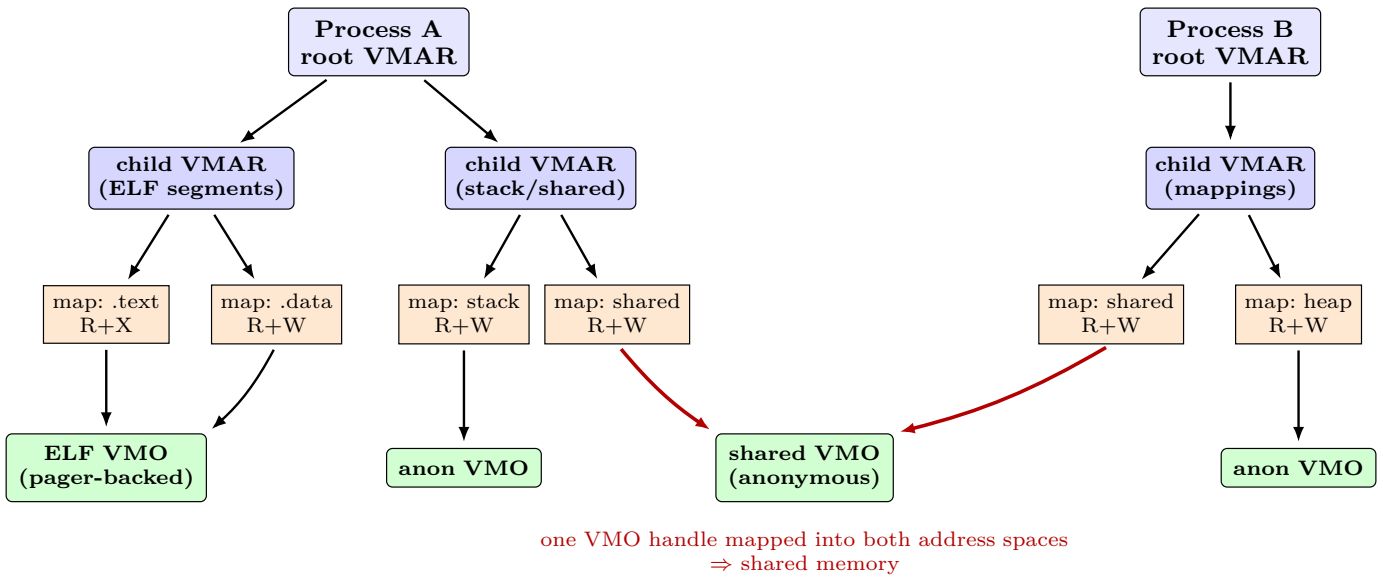


Fig. 14. The VMAR tree of two processes. Each root VMAR fans out into child VMARs that group ELF segments, stacks and shared regions. The leaves are mappings, and each mapping binds a range of some VMO. The same anonymous VMO is mapped into both Process A and Process B (heavy edges). The two processes now share physical pages, possibly at different virtual addresses. This is how Zircon shares memory.

1) *Process Creation Yields an Empty Root VMAR:* `zx_process_create(job, name, name_size, options, &proc, &root_vmar)` returns two handles. One is the process itself. The other is its *root VMAR*, which spans the whole usable user part of the address space. Below, it stops at a reserved guard region around the null address, so that a null-pointer dereference faults instead of hitting valid memory. Above, it stops at a region the kernel reserves. The new process is *empty*: not one page is mapped and no code is loaded. The kernel never reads an ELF file or chooses a layout. It only hands back the empty address space and the root VMAR handle that user space will use to fill it.

2) *Who Builds the Map: the Two-Phase Model:* The address space starts empty, so code running *outside* the new process must fill it. Fuchsia uses a bootstrap in two phases.

In the first phase a *launcher* does the work. This is the `fuchsia.process.Launcher` service, acting for whoever started the program. It maps just enough to start: the *dynamic linker* (the ELF interpreter, part of `libc`), a small bootstrap stack, and the `vDSO`. It then sends a bundle of *processargs* over a *bootstrap channel*. The bundle holds the handles and settings the new process needs, including the executable's VMO, the loader-service channel, the root VMAR, the job and the `vDSO` VMO. Finally it calls `zx_process_start` with the dynamic linker's entry point.

In the second phase the dynamic linker runs *inside* the new process. It reads the *processargs* and maps the main executable and every shared library it depends on. It applies relocations and then jumps to the program's `_start`. This split keeps the loading policy (which file, which libraries, and where) in user code that can be replaced.

3) *Mapping the ELF Segments:* The executable arrives as a VMO, served by the file system's pager from the on-disk image. The loader reads its ELF program headers and creates one mapping for each `PT_LOAD` segment (Fig. 15):

- The read-only segments, `.text` and `.rodata`, are mapped from a `ZX_VMO_CHILD_SNAPSHOT_AT_LEAST_ON_WRITE` child of

the file VMO, with `ZX_VM_PERM_READ` (and `EXECUTE` for text). Their pages stay *shared* by every process that runs the same binary. The “shared text” and unified page cache of classic systems come from plain VMO sharing, at no extra cost.

- The writable segment that holds `.data` is a copy-on-write child mapped `READ|WRITE`. It reads initial values through shared pages and gets a private copy of a page on the first write to it.
- The `.bss` tail beyond the file's content is zero-filled. It is an anonymous extension of that writable region, committed on demand.

Two rules hold throughout. No segment is both writable and executable ($W \oplus X$, which the job policy `ZX_POL_VMAR_WX` can enforce). And making *any* mapping executable needs the VMO execute right, which only an executable-resource capability can grant (§IX-B).

4) *The Stack, the vDSO, and “Growing” Memory:* The initial thread's **stack** is an anonymous VMO mapped `READ|WRITE`. It sits next to an unmapped **guard region**, a sub-VMAR left with a hole on purpose. A stack overflow then faults instead of silently corrupting a neighboring mapping. The top of the stack is passed as the stack pointer to `zx_process_start`.

The launcher maps the **vDSO** `READ|EXECUTE` and passes its base address to the new process. The optional `PA_VMO_VDSO` *processargs* handle lets the process give the `vDSO` to children it starts. The `vDSO` is the *only* executable mapping the kernel requires in every process. It is the gate through which every system call enters the kernel (§IV).

There is *no* Unix `brk`. The **heap** is just more VMO mappings. The C library's allocator (Scudo on Fuchsia) creates anonymous VMOs and maps them as it needs them, and `mmap`-style allocations are VMO mappings too. Each shared library (`.so`) that the dynamic linker loads adds another set of segment mappings, just like the executable's. The *loader service*, reached through the `PA_LDSVC_LOADER` handle, supplies each library's VMO. In short, growing a process's memory always means mapping another VMO, never asking for a privileged kernel

region.

5) *ASLR, Sub-VMARs, and the Page Tables*: The loader randomizes placement for address-space layout randomization. It usually carves one child VMAR per loaded module with `zx_vmar_allocate`. That child VMAR holds the module’s segments in one aligned, contiguous range, mapped at fixed offsets from each other. Fig. 15 shows two such module VMARs. This keeps the layout compact and bracketed by guards, and it lets a whole module be removed by destroying one VMAR.

The VMAR tree, its child VMARs and their mappings are all *bookkeeping*. The hardware version is the per-architecture page table behind the address space, called `ArchVmAspace`. It is a PML4 hierarchy on x86, translation tables rooted at `TTBR0` on ARM64, and Sv39 or Sv48 tables on RISC-V. The two layers are kept separate, and the page table is filled *lazily*. `zx_vmar_map` records a mapping but installs no page-table entries. Each entry is filled only when a fault touches its page, or at once when the caller passes `ZX_VM_MAP_RANGE`. The VMAR layer decides policy and permissions; the page table is its hardware shadow. When a mapping is removed or its permissions are narrowed, the kernel updates both the VMAR structure and the page table. It then runs the TLB shutdown of §XI-E, so that no CPU keeps a stale translation.

6) *The Startup Sequence*: Launching a program takes these steps:

- 1) `zx_process_create(job, ...)` returns a process and an empty root VMAR.
- 2) The launcher maps the dynamic linker, a bootstrap stack and the vDSO into the new root VMAR, using `zx_vmar_map`.
- 3) It creates the initial thread (`zx_thread_create`) and a bootstrap channel. It writes the processargs into the channel: handles, the executable’s VMO, and the loader-service channel.
- 4) It calls `zx_process_start(proc, thread, entry, sp, bootstrap_handle, vdso_base)`. Here `entry` is the *dynamic linker’s* entry point and `sp` is the top of the bootstrap stack. `bootstrap_handle` is the channel end the new process reads the processargs from, and `vdso_base` is the address of the vDSO.
- 5) Inside the process, the dynamic linker reads the processargs. It maps the main executable’s segments and each dependency, fetching library VMOs from the loader service, and applies relocations.
- 6) It sets up the real thread stack and jumps to the program’s `_start`.

The result is a VMAR tree of mappings over a handful of VMOs (Fig. 15, Table VIII):

- copy-on-write, file-backed code and data, shared with every other instance of the binary and its libraries;
- an anonymous, private stack and heap;
- the vDSO, supplied by the kernel.

User-space code built all of it, through the system calls of this section.

Listing 11. Create a VMO, map it, write through the mapping, then take a lazy-COW child.

```
zx_handle_t vmo;
// 64 KiB anonymous, resizable VMO: zero-filled, no pages yet.
zx_vmo_create(64 * 1024, ZX_VMO_RESIZABLE, &vmo);

// Map it read/write into the process root VMAR. Address chosen
// (and randomized) by the kernel; pages fault in on first touch.
zx_vaddr_t addr;
zx_vmar_map(zx_vmar_root_self(),
            ZX_VM_PERM_READ | ZX_VM_PERM_WRITE,
```

TABLE VIII
TYPICAL PROCESS REGIONS: BACKING, PERMISSIONS, AND WHO CREATES THEM.

Region	Backing VMO	Perm	Created by
.text	COW child of file VMO	r-x	dyn. linker
.rodata	COW child of file VMO	r--	dyn. linker
.data	COW child of file VMO	rw-	dyn. linker
.bss	anonymous (zero)	rw-	dyn. linker
shared libs	COW children (loader svc)	varies	dyn. linker
vDSO	kernel-supplied VMO	r-x	kernel / launcher
stack	anonymous	rw-	launcher / linker
heap	anonymous (on demand)	rw-	Scudo malloc
mmap	anonymous / file VMO	varies	app / malloc
guards	<i>unmapped</i> hole	none	loader (sub-VMAR)

```
/*vmar_offset=*/0, vmo, /*vmo_offset=*/0,
64 * 1024, &addr);

// Touch a page: this triggers a fault, the PMM hands back a
// zero frame, and the store commits it.
((volatile uint64_t*)addr)[0] = 0xA5A5A5A5;

// Out-of-band write without a mapping is also fine:
uint64_t word = 0x1234;
zx_vmo_write(vmo, &word, /*offset=*/4096, sizeof(word));

// Lazy copy-on-write child: shares pages until a write diverges.
zx_handle_t child;
zx_vmo_create_child(vmo, ZX_VMO_CHILD_SNAPSHOT_AT_LEAST_ON_WRITE,
/*offset=*/0, 64 * 1024, &child);
```

The VMO and VMAR pair, demand paging, COW children and the rights checks are the part of Zircon memory that user space sees. Beneath them sit the physical memory manager, which hands out the frames, and the user-space pager, which fills them. The next section covers both.

X. PHYSICAL MEMORY AND DEMAND PAGING

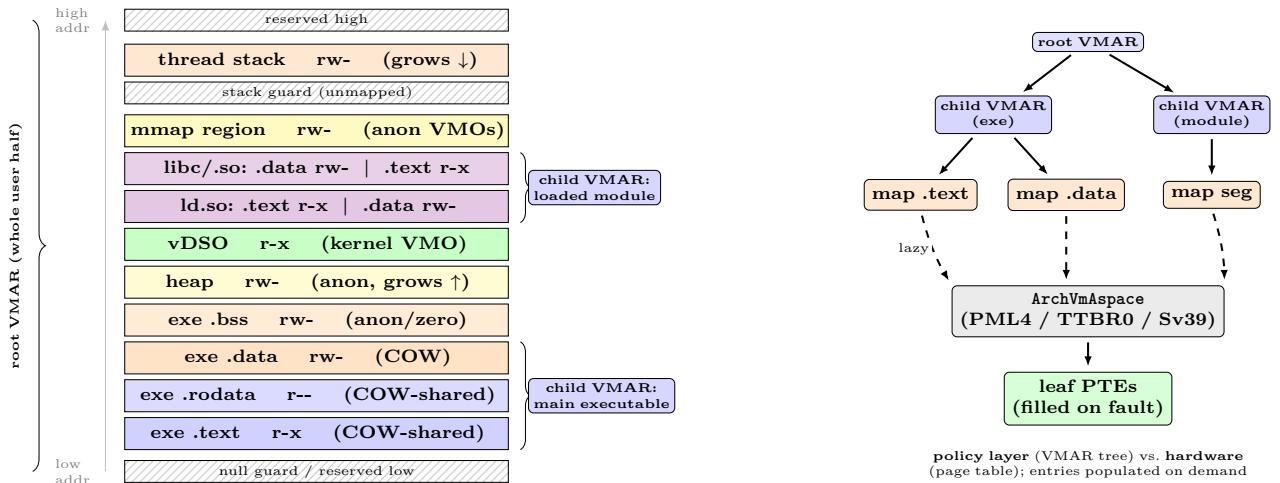
Every VMO page is backed by a physical frame of RAM. Every frame is managed by the *physical memory manager* (PMM), the kernel module that owns all of RAM. The PMM hands frames to the VM layer on demand and takes them back under pressure. Above the PMM sits the machinery that makes a finite pool of RAM behave like a larger memory:

- page queues, which rank frames for eviction;
- an in-RAM compressor;
- the user-space pager, a page-miss handler that runs entirely outside the kernel.

This section follows a frame through its life. It covers allocation, classification and reclamation, and then the demand-paging round trip that fills the frame.

A. The Physical Memory Manager

The PMM is `PmmNode` in `zircon/kernel/vm` [22]. It divides physical RAM into one or more *arenas* (`PmmArena`). Each arena describes a contiguous span of usable frames, found at boot in the bootloader’s memory map. The PMM tracks every 4 KiB frame with a fixed-size record, `vm_page_t` (`struct vm_page`).



COW-shared = SNAPSHOT_AT_LEAST_ON_WRITE child of the file VMO (shared with every instance); anon = private zero-filled VMO.

Fig. 15. Constructing a process address space. *Left*: a typical memory map, high addresses at the top. The root VMAR spans the whole user half, and child VMARs group the main executable’s segments and each loaded module. Read-only code and data are copy-on-write children of the file VMO, shared by all instances. The stack, heap, .bss and mmap regions are anonymous. The vDSO is a read-execute mapping of a VMO the kernel supplies. Guard regions bracket the stack and the null address. *Right*: the VMAR and mapping tree is the policy layer. The per-architecture ArchVmAspace page table is its hardware form: PML4 on x86, TTBR0 tables on ARM64, Sv39 or Sv48 on RISC-V. Its leaf entries are filled lazily, at fault time.

The record holds the frame’s physical address, an intrusive list node, a state byte, and a union of per-use “views”:

- an **object** view, used when the frame backs a VMO, which holds a back-pointer to the owning VmCowPages, the page’s offset, its pin count and its dirty state;
- a **zram** view, used when the frame holds compressed data.

The set of states is small, and every frame is in exactly one of them (Listing 12).

Listing 12. Physical-frame states (vm_page_state).

```
enum class vm_page_state : uint8_t {
  FREE = 0, ALLOC, OBJECT, WIRED, HEAP, MMU,
  IOMMU, IPC, CACHE, SLAB, ZRAM, FREE_LOANED, COUNT_
};
```

Free frames sit on the node’s free list (**free_list_**). Some frames are loaned out of a contiguous VMO and are unused for the moment. Those sit on a separate loaned free list (**free_loaned_list_**). The node’s lock serializes allocation and free. The current design has no per-CPU free cache. The PMM reduces contention in two other ways: it allocates in batches, and it keeps the loaned pool on its own list and lock.

The VM layer asks the PMM for a page to satisfy a fault, a commit, or growth of the kernel heap. The PMM unlinks a frame, stamps its state, and returns it. On free, the frame is relinked and zeroed lazily. WIRED, HEAP, MMU and SLAB frames belong to the kernel itself: page tables, the heap and slab caches. They are never pageable. OBJECT frames back user VMOs, and they are the candidates for reclamation.

B. Page Queues and Aging

The kernel enrolls each frame that backs a VMO in one of the *page queues* (PageQueues). The queue records the frame’s class and, for a reclaimable frame, how recently it was used. The classes include:

- **Wired**: pinned, never reclaimed;
- **Anonymous** and **AnonymousZeroFork**: RAM with no backing store;

- **PagerBackedDirty**: modified file pages that wait for write-back;
- **HighPriority**: protected from reclamation;
- a ring of *reclaim* queues, **ReclaimBase...ReclaimLast**.

The reclaim ring has eight buckets in the current source, and it approximates LRU order. A background aging thread periodically rotates the “most recently used” generation forward. Each access to a page, recorded through the access bit and **MarkAccessed**, moves the page toward the MRU end. The reclaimer never picks a victim from the newest bucket, so a running thread’s working set is not evicted from under it. Two buckets are *active*. The rest are *inactive*, and the reclaimer draws from them first. Anonymous and pager-backed reclaimable pages share this ring. So the kernel can weigh a clean file page against a compressible anonymous page on one scale of recency.

C. Reclamation: Evict, Compress, Discard

When free memory runs low, a kernel scanner walks the inactive reclaim queues. It recovers each frame by one of three routes, chosen by the kind of page (Fig. 16).

- **Evict clean pager-backed pages**. A pager-backed page is *clean* when it matches its backing store. The kernel can simply drop it, because the pager can fetch the contents again on the next fault. This is the cheapest kind of reclamation. It is why executables and file caches cost little under pressure: their pages are pure cache.
- **Compress anonymous pages**. Anonymous pages have no backing store, so the kernel cannot evict them. It can *compress* them in place instead. The VmCompression subsystem (RFC-0219, “Zircon Page Compression” [21]) passes the frame through a compression strategy. The default strategy is VmLz4Compressor, an LZ4 codec. The subsystem packs several compressed 4 KiB pages into one storage page and frees the rest. The frame’s state becomes ZRAM, and its zram view records where the compressed bytes live. A later fault decompresses the data into a fresh frame. This is the same

TABLE IX
VMO AND VMAR SYSTEM CALLS WITH THEIR PRINCIPAL OPTIONS AND FLAGS.

Call / flag group	Meaning
<i>VMO lifecycle and data</i>	
<code>zx_vmo_create</code>	Anonymous, zero-filled, demand-committed VMO.
<code>_create_physical</code>	VMO over a fixed physical (device) range.
<code>_create_contiguous</code>	Physically contiguous RAM for DMA (via BTI).
<code>_read</code> / <code>_write</code>	Copy bytes in/out without a mapping.
<code>_get_size</code> / <code>_set_size</code>	Query / resize (<code>set</code> needs <code>ZX_RIGHT_RESIZE</code>).
<code>_op_range</code>	Commit, decommit, zero, cache-maint., lock, hint.
<code>_set_cache_policy</code>	Cached / uncached / write-combining.
<code>_create_child</code>	Derive a COW / slice / reference child.
<i>VMO create options</i>	
<code>ZX_VMO_RESIZABLE</code>	Permit later <code>set_size</code> .
<code>ZX_VMO_DISCARDABLE</code>	Pages may be reclaimed unless locked.
<i>Child modes (<code>ZX_VMO_CHILD_*</code>)</i>	
<code>SNAPSHOT</code>	Immutable COW copy, isolated from parent.
<code>SNAPSHOT_AT_LEAST_ON_WRITE</code>	Lazy COW: private copy of a page on write.
<code>SLICE</code>	Shared window (alias) into parent pages.
<code>REFERENCE</code>	Full-size alias object over parent pages.
<i>Cache policy (<code>ZX_CACHE_POLICY_*</code>)</i>	
<code>CACHED</code>	Normal write-back RAM.
<code>UNCACHED</code> / <code>_DEVICE</code>	Every access reaches memory/device.
<code>WRITE_COMBINING</code>	Coalesce stores (framebuffers).
<i>VMAR calls and map options (<code>ZX_VM_*</code>)</i>	
<code>zx_vmar_allocate</code>	Carve a child VMAR (subregion).
<code>zx_vmar_map</code>	Map a VMO range into a VMAR.
<code>zx_vmar_protect</code> / <code>_unmap</code>	Change perms / tear down a range.
<code>PERM_READ/WRITE/EXECUTE</code>	Mapping permissions (need matching rights).
<code>SPECIFIC</code> / <code>[_OVERWRITE]</code>	Place at a chosen offset (replace existing).
<code>MAP_RANGE</code>	Eagerly populate page tables.
<code>CAN_MAP_*</code>	Capabilities a child VMAR may grant.

trade that Linux zram makes, often called “swap to RAM”. The kernel spends CPU cycles and a fraction of RAM so that it need not throw away data that has nowhere else to go.

- **Discard discardable VMOs.** A client can create a VMO with the `DISCARDABLE` option. While its pages are not *locked*, the kernel may throw away the whole contents at once. The next access faults the range back in as zeros. The owner learns from the return of its next lock call that it must rebuild the data. A client keeps its data live by bracketing each use with `zx_vmo_op_range(...ZX_VMO_OP_LOCK...)` and `...ZX_VMO_OP_UNLOCK...`. Caches of assets that can be rebuilt, such as decoded images and font atlases, use this to give memory back without being asked.

Each route has a clear cost:

- eviction is free, but it applies only to clean pages that the pager can fetch again;
- compression keeps the data, but it costs CPU time and some RAM;
- discarding is instant, but it destroys data that the owner must rebuild.

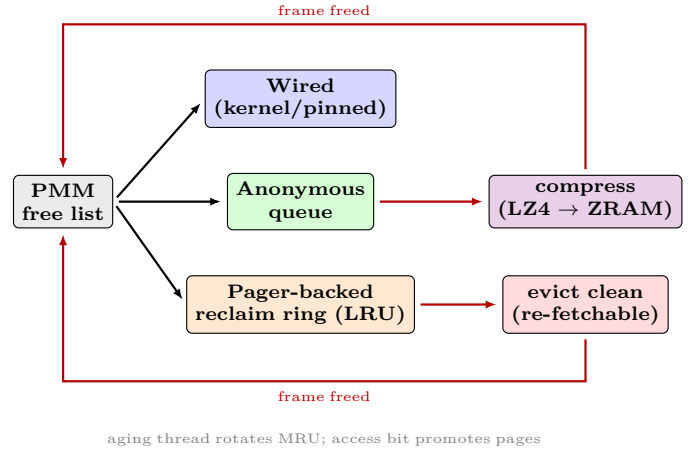


Fig. 16. Physical-page lifecycle. Frames leave the PMM free list as wired, anonymous, or pager-backed pages. Under pressure the reclaimer compresses anonymous pages (LZ4, state ZRAM) or evicts clean pager-backed pages, which the pager can fetch again. Both routes return frames to the free list. Wired frames are never reclaimed.

1) *Zero-Page Deduplication and Page Borrowing:* Two more mechanisms stretch the physical pool.

The first is *zero-page deduplication*. Freshly committed anonymous pages are all zeros, and many stay that way. A background scanner finds committed pages whose contents are still all zeros. It maps each one back to the single shared zero page and marks the slot copy-on-write, so a later write forks a private frame again. The `AnonymousZeroFork` queue tracks pages that began life as a fork of the shared zero page. Those pages are good candidates to drop again under pressure. As a result, a sparse VMO with few written pages uses almost no committed frames.

The second is *page borrowing*. A contiguous VMO reserves physically contiguous frames for DMA, but those frames are often idle between transfers. The PMM may *loan* the unused frames to the pager-backed cache instead of leaving them idle. Loaned frames appear on the separate loaned free list (`free_loaned_list`), and they can hold evictable file pages for a while. When the contiguous VMO needs a loaned frame back, the kernel reclaims the borrower’s page and returns the frame to its owner. This is safe because the borrowed page was clean, so the pager can fetch it again. Loaning lets RAM reserved for worst-case DMA do useful caching work in the common case. The cost is the bookkeeping needed to reclaim a frame on demand.

D. The User-Space Pager

The pager is the part of Zircon’s memory system that most sets it apart. Zircon keeps file system and decompression code out of the kernel. Instead, a user-space process registers as the supplier of a VMO’s contents. It then acts as a *software page-miss handler* [8]. When a thread touches a page of a pager-backed VMO that is not in memory, the kernel raises a fault. It sends a request to the user process, which fills the page, and then the kernel retries the access. The hardware analogy is close. On a TLB miss a CPU’s MMU walks the page table, and on a page-table miss it raises a fault. Zircon adds one more level: on a VMO page miss it “walks” out to a user process. The kernel provides the trap and the resume, and the policy lives in user space.

A process creates a pager with `zx_pager_create(0, &pager)`. It then creates each pager-backed VMO with

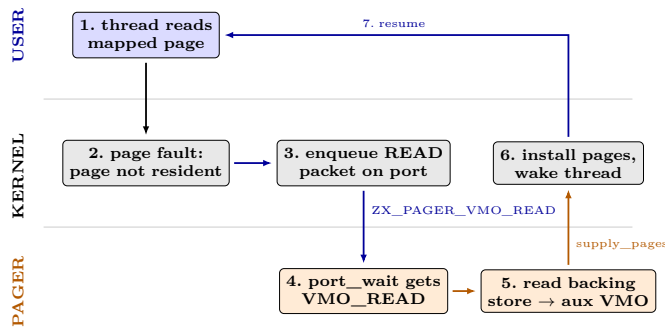


Fig. 17. Demand paging through the user-space pager. A read of a pager-backed page that is not in memory faults (2). The kernel queues a `ZX_PAGER_VMO_READ` packet on the pager’s port (3). The pager fetches the data into an auxiliary VMO (4 and 5) and calls `zx_pager_supply_pages`. The kernel installs the pages and resumes the thread (6 and 7). The kernel supplies only the trap and the resume; the code that fills the page is user code.

`zx_pager_create_vmo(pager, options, port, key, size, &vmo)`. The call binds the VMO to a *port* that receives its requests. The kernel tags every request from this VMO with *key*, so one port can serve many VMOs. When a thread faults on a page of that VMO that is not in memory, the kernel suspends the thread. It then queues a packet on the port (Listing 13).

Listing 13. The page-request packet delivered to the pager’s port.

```

typedef struct zx_packet_page_request {
    uint16_t command; // ZX_PAGER_VMO_READ / _COMPLETE / _DIRTY
    uint16_t flags;
    uint32_t reserved0;
    uint64_t offset; // byte offset within the pager VMO
    uint64_t length; // byte length of the requested range
    uint64_t reserved1;
} zx_packet_page_request_t; // packet type ZX_PKT_TYPE_PAGE_REQUEST
  
```

The pager serves the request in four steps:

- 1) It takes the packet off the port with `zx_port_wait`. The packet has `command == ZX_PAGER_VMO_READ` and an `[offset, length)` range.
- 2) It fetches that data from its backing store (a disk, the network, or a decompressor) into a scratch “auxiliary” VMO.
- 3) It calls `zx_pager_supply_pages(pager, pager_vmo, offset, length, aux_vmo, aux_offset)`. The call *moves* the pages out of the auxiliary VMO into the pager-backed VMO at the requested offset.
- 4) The kernel wakes the faulting thread. The thread runs the access again and now finds the page in memory.

If the pager cannot fill the range, it calls `zx_pager_op_range` with `ZX_PAGER_OP_FAIL` and a `zx_status_t` in the *data* argument. The kernel then turns the faulting access into an exception in the client. When a VMO is torn down, the kernel delivers `ZX_PAGER_VMO_COMPLETE` (and `zx_pager_detach_vmo` is called). This tells the pager that no more requests will arrive for that VMO. Figure 17 traces the whole round trip across the boundary between kernel and user space.

1) *Dirty Tracking and Write-Back*: A read-only file cache needs only `READ` requests. A writable file system, such as Fxfs, must also know which pages the client changed, so that it can write them back. The pager asks for this by creating the VMO with the `ZX_VMO_TRAP_DIRTY` option. The kernel then traps the first write to each clean page in memory and sends the pager a `ZX_PAGER_VMO_DIRTY` request. The pager has two answers:

- it approves the change from clean to dirty with `zx_pager_op_range(...ZX_PAGER_OP_DIRTY...)`, after it has reserved space in the backing store;
- or it rejects the write with `ZX_PAGER_OP_FAIL`.

To flush, the pager finds the modified regions with `zx_pager_query_dirty_ranges`. It brackets the write-out with `ZX_PAGER_OP_WRITEBACK_BEGIN` and `...WRITEBACK_END`. The end call returns the pages to the clean state, where they can be cheaply evicted again. The pager can also call `zx_pager_query_vmo_stats` to learn whether a VMO was modified at all. With this protocol, a user-space file system keeps the same dirty-page records that a monolithic kernel keeps internally. The difference is that here each step is an explicit message that can be audited.

E. Memory-Pressure Signaling and OOM

Reclamation buys time. When it cannot keep up, the kernel asks user space to help by giving up memory. A privileged process gets one pressure event per level with `zx_system_get_event(root_job, kind, &event)`. The levels are:

- `ZX_SYSTEM_EVENT_MEMORY_PRESSURE_NORMAL`, `_WARNING` and `_CRITICAL`;
- `ZX_SYSTEM_EVENT_IMMINENT_OUT_OF_MEMORY`;
- `ZX_SYSTEM_EVENT_OUT_OF_MEMORY`.

The kernel asserts `ZX_EVENT_SIGNED` on exactly one pressure-level event at a time. As free memory falls, it moves from `NORMAL` to `WARNING` to `CRITICAL`.

A memory monitor in user space waits on these events. It responds by dropping caches, unlocking discardable VMOs, trimming image and font caches, and asking the component framework to stop low-priority work. This recovers memory that the kernel could not reclaim on its own. If pressure still cannot be relieved, the `IMMINENT_OUT_OF_MEMORY` event gives diagnostic tools a last chance to capture state. Then, on `OUT_OF_MEMORY`, the kernel performs a controlled out-of-memory reboot. A clean shutdown and restart is preferred to unpredictable thrashing or a hard hang.

Together, the PMM, the page queues, the compressor and the pager let Zircon run a large user-space software stack on a bounded pool of RAM. File systems, loaders and graphics all live in user space. The policy for filling and freeing pages lives there too, while the kernel keeps control of the frames themselves. Section XI turns inward, to how the kernel manages *its own* memory and coordinates it across CPUs.

XI. KERNEL MEMORY, PER-CPU STATE, AND SMP

Sections IX and X covered the memory that the kernel hands to user space. This section covers the kernel’s *own* memory. It explains how the kernel maps and allocates that memory, and how it keeps fast per-CPU state. It also explains how the kernel brings up several cores, coordinates them, and keeps every CPU’s view of the page tables consistent. Applications never see these mechanisms. They still decide how well the kernel scales, and whether it is correct, on symmetric multiprocessing (SMP) hardware.

A. The Kernel Address Space and the Physmap

On a 64-bit machine the virtual address space is split into two halves. The low *user* half has a different mapping in each process. The high *kernel* half is the same in every address space,

TABLE X
PAGER SYSCALLS, REQUEST/OP CODES, AND MEMORY-PRESSURE
EVENTS.

Call / constant	Meaning
<i>Pager syscalls</i>	
<code>zx_pager_create</code>	Create a pager object (<code>options=0</code>).
<code>_create_vmo</code>	Create a pager-backed VMO bound to a port + key.
<code>_supply_pages</code>	Move pages from an aux VMO into the pager VMO.
<code>_op_range</code>	Issue <code>ZX_PAGER_OP_*</code> (below).
<code>_detach_vmo</code>	Stop serving a VMO; triggers <code>VMO_COMPLETE</code> .
<code>_query_dirty_ranges</code>	Enumerate modified ranges for write-back.
<code>_query_vmo_stats</code>	Report whether a VMO was modified.
<i>Port request commands (<code>ZX_PAGER_VMO_*</code>)</i>	
<code>READ</code>	Supply the non-resident [<code>offset</code> , <code>len</code>] range.
<code>DIRTY</code>	Client wrote a clean page (needs <code>TRAP_DIRTY</code>).
<code>COMPLETE</code>	No further requests (VMO detached).
<i>Pager ops (<code>ZX_PAGER_OP_*</code>)</i>	
<code>FAIL</code>	Cannot fill range; <code>data</code> = error status.
<code>DIRTY</code>	Approve clean→dirty transition.
<code>WRITEBACK_BEGIN/END</code>	Bracket write-out; <code>END</code> returns pages to clean.
<i>Pressure events (<code>ZX_SYSTEM_EVENT_*</code>)</i>	
<code>MEMORY_</code>	Ample free memory.
<code>PRESSURE_NORMAL</code>	
<code>..._WARNING /</code>	Shed caches / shed aggressively.
<code>..._CRITICAL</code>	
<code>IMMINENT_</code>	Last-chance diagnostics.
<code>OUT_OF_MEMORY</code>	
<code>OUT_OF_MEMORY</code>	Controlled OOM reboot.

so the kernel sees the same view whichever process is current. A single `VmAspace` describes the kernel half. Code reaches it as `VmAspace::kernel_aspace()`, and its root VMAR holds the kernel’s mappings.

The most important mapping in that half is the *physmap*. It is a linear, contiguous mapping of *all* usable physical RAM into the kernel’s address space. Every physical address therefore has a kernel virtual address that is simple to compute. The kernel can touch any frame without first setting up a mapping. Physical `[0, gPhysmapSize)` maps to virtual `[gPhysmapBase, gPhysmapBase + gPhysmapSize)`. The conversions are plain arithmetic:

- `paddr_to_physmap()` adds the base;
- `physmap_to_paddr()` subtracts it;
- `is_physmap_addr()` tests whether an address lies in the range.

Because of the physmap, the heap can use a frame the moment the PMM returns it. There is always a kernel pointer ready for any physical memory.

Other regions sit alongside the physmap:

- the kernel image itself (text, rodata, data and bss), loaded at `__executable_start` and mapped with the right permissions for each part;
- the kernel heap region;
- the per-CPU areas;
- various device and bookkeeping mappings.

Kernel pages are *wired*. Their frames are in states such as `WIRED`, `HEAP`, `MMU` or `SLAB`. The reclaimer of `$X` never touches them, because the kernel must not page itself out. Figure 18 sketches

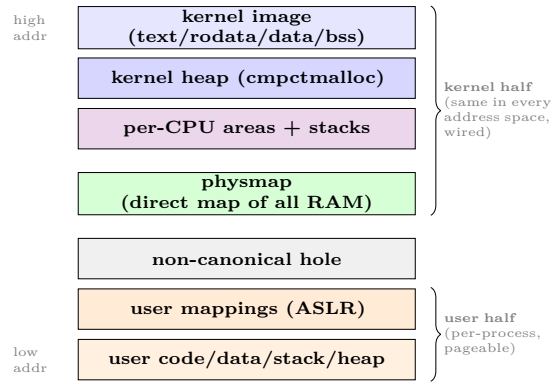


Fig. 18. Kernel address-space map. The high half is the same in every process. It holds the kernel image, the `cmpctmalloc` heap, the per-CPU areas, and the `physmap`, which maps every physical frame linearly for direct access. All of it is wired. The low half holds the current process’s user mappings, which are pageable and placed at random by ASLR.

the layout.

B. The Kernel Heap

Variable-size kernel allocations go through `cmpctmalloc` (`zircon/kernel/lib/heap/cmpctmalloc`). It is a compact allocator tuned to save space. Its interface is `cmpct_alloc`, `cmpct_free` and `cmpct_memalign`. It does not own memory of its own. When it runs short, it calls `heap_grow()`. That call pulls page-aligned chunks from the page layer, and so ultimately from the PMM, through `heap_page_alloc()`. The heap returns whole pages with `heap_page_free()`. It keeps one span cached, so that a burst of allocations and frees does not move pages back and forth each time. The heap therefore sits directly on top of the physical memory manager.

The kernel does not serve small fixed-size objects that are created and freed often from the general heap. There they would fragment the heap and pay for its metadata. The clearest example is the *handle*. The kernel allocates every open handle in the system from a dedicated arena, `HandleTableArena` (the global `gHandleTableArena`). Its `arena_.Alloc()` and `Free()` hand out fixed-size slots from a preallocated region. The region is capped at `kMaxHandleCount = 262144` handles. An arena gives $O(1)$ allocation and tight packing. Its hard ceiling also bounds kernel memory, whatever user programs do.

The dispatcher objects that handles point at are allocated on the heap. The kernel uses `fbl::AllocChecker` with placement `new` and wraps each one in a `KernelHandle`. The library also offers a general `fbl::SlabAllocator` template for subsystems that want slab-style pools of fixed-size objects. The kernel reports its memory use to user space through `zx_object_get_info` with `ZX_INFO_KMEM_STATS` and `ZX_INFO_KMEM_STATS_EXTENDED`. Part of that data comes from the per-CPU page counters described in §XI-C.

A third kind of kernel memory is the per-thread *kernel stack*. Every thread that traps into the kernel runs on a small, fixed-size kernel stack, separate from its user stack. On the supported targets each thread also has an auxiliary stack. The compiler’s safe-stack instrumentation uses it to keep return addresses apart from spilled buffers. These stacks are wired anonymous allocations that are never paged. Guard mappings surround them, so an overflow faults instead of silently corrupting a neighbouring allocation. A kernel stack stays committed for as

long as its thread is blocked. So each thread’s kernel memory, its stack plus the thread and dispatcher records, limits how far the system scales. This is one reason Zircon favours asynchronous designs built on ports over large numbers of blocked threads.

C. Per-CPU State

Kernel state that exists once per core, and is used often, lives in an array of `struct percpu` with one element per CPU [22]. Each element holds that CPU’s:

- **Scheduler**, with its run queues;
- **TimerQueue**;
- deferred-procedure-call runner (**DpcRunner**);
- CPU and guest statistics;
- page-state counters (**vm_page_counts**);
- idle-power and resume state;
- lock-up detector timer;
- memory-stall accumulator.

Because this state is per-CPU, scheduling decisions, timer expiry and statistics updates touch only the local element. They need no cross-CPU lock on the common path.

The kernel finds the *current* CPU’s element without a lock and without looking up the CPU number. It keeps a pointer to the element in a register that it reserves for the purpose. On x86 that pointer is the GS base, and `x86_get_percpu()` reads it directly through `gs::`. A standing rule in the kernel requires `gs_base` to point at the current core’s `x86_percpu` whenever interrupts are enabled. On ARM64 the pointer lives in general-purpose register `x20`, and `arm64_read_percpu_ptr()` reads it. The portable `percpu::GetCurrent()`, built on `arch_get_curr_percpu()`, wraps both. `percpu::Get(cpu_num)` addresses any other core. The kernel updates the register only at context boundaries that it controls. So a read is a single instruction with no synchronization.

Listing 14. Per-CPU access and the IPI taxonomy (sketch of the kernel interfaces).

```
// One element per core; current element via a reserved register.
struct percpu { me = percpu::GetCurrent(); // x86: gs::, arm64: x20
me.scheduler.Reschedule();

// Inter-processor interrupt kinds (scoped enum).
enum class mp_ipi : uint8_t {
    GENERIC, RESCHEDULE, INTERRUPT, HALT
};
enum class mp_ipi_target : uint8_t { MASK, ALL, ALL_BUT_LOCAL };

// Run a function synchronously on a set of other CPUs.
void mp_sync_exec(mp_ipi_target target, cpu_mask_t mask,
                 mp_sync_task_t task, void* context);
```

D. SMP Bring-Up

Zircon boots on a single *boot CPU*. That CPU sets up the core subsystems: memory, the heap, the object layer and the scheduler. It then starts the secondary cores one at a time, through `mp_init()` and the per-architecture hotplug path `arch_mp_cpu_hotplug()`.

Each secondary core starts in a tiny assembly *trampoline* written for its architecture. On ARM64 this is `start.S` and `secondary.S`. On x86 it is a real-mode bootstrap started by `x86_bringup_aps()`. The trampoline sets up a stack and the kernel page tables, then jumps into C++. On ARM64 the C++ entry is `arm64_secondary_entry()`. It runs early per-CPU setup (`arm64_init_percpu_early()` and `arch_mp_init_percpu()`) and steps through the secondary-CPU init levels. Finally it calls the common `lk_secondary_cpu_entry()`, which hands the core to the scheduler. On x86, `x86_init_percpu()` reaches

the same hand-off. From then on the core is an equal peer of the others, and it takes threads from its own run queue. For power management, cores can later be removed and added back through `mp_unplug_cpu_mask()` and `mp_hotplug_cpu_mask()`.

Running cores coordinate through *inter-processor interrupts* (IPIs). The kind of IPI is a scoped enum, `mp_ipi`, with four members (Listing 14):

- **RESCHEDULE** asks a core to run its scheduler again, for example after a thread on it became runnable;
- **GENERIC** runs a function on the target cores, and is how `mp_sync_exec` works;
- **INTERRUPT** delivers a device interrupt to a chosen core;
- **HALT** stops a core, on panic and at shutdown.

The kernel names the targets with an `mp_ipi_target`: a **MASK** of CPUs, **ALL**, or **ALL_BUT_LOCAL**. `arch_mp_send_ipi()` turns that into the hardware operation.

E. TLB Management and Shutdown

Each core’s MMU caches recently used page-table entries in a translation look-aside buffer (TLB). The kernel may unmap a page or tighten its permissions (`zx_vmar_unmap`, `zx_vmar_protect`). Any core whose TLB cached the old translation then holds a stale entry, which would still allow the old access. The *TLB shutdown* removes those stale entries from every core.

On x86 the shutdown is done in software (Fig. 19):

- 1) The unmapping core records the pending invalidation (**PendingTlbInvalidation**).
- 2) It calls `mp_sync_exec`, which sends a **GENERIC** IPI to every core that might share the mapping.
- 3) Each target runs an invalidation task that executes the local `invlpg` or `INVPCID`, then acknowledges.
- 4) The originating core continues only when every target has acknowledged.

Flushing the whole TLB on every address-space switch would be slow. So x86 uses *process context identifiers* (PCIDs) where the CPU has them, gated by `g_x86_feature_pcid_enabled`. Each address space gets a PCID, so its entries can stay in the TLB across switches. The **INVPCID** variants invalidate one PCID’s entries instead of the whole TLB. Flushes for idle or inactive cores wait until their next context switch.

ARM64 mostly avoids the IPI. Its TLB-invalidate instructions have an *inner-shareable* broadcast form: `vae1is`, `vale1is`, `vmalle1is`, and the ASID-scoped `aside1is`. The hardware passes these to every core in the shareability domain. So one core that issues the invalidation flushes all of them, with no software round trip. ARM64 also tags translations with *address space identifiers* (ASIDs). An `AsidAllocator` hands them out, and a reserved global ASID (`MMU_ARM64_GLOBAL_ASID`) covers the kernel, which is always mapped. ASIDs do the job that PCIDs do on x86. They let a process’s entries survive context switches and reduce how often a full flush is needed. Both architectures reach the same goal, no stale translations, by different means. x86 uses a software IPI and ARM64 uses a hardware broadcast.

F. Memory Ordering and Lockless Publication

On SMP hardware, correctness depends on the order in which cores see memory writes. The kernel does not rely on whatever order the hardware happens to give. It uses explicit atomics (`ktl::atomic`) and the architecture’s barrier instructions for

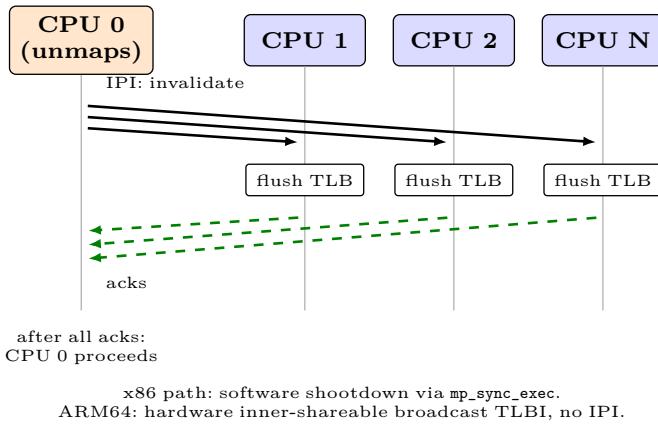


Fig. 19. TLB shootdown (x86 software model). The unmapping core sends a **GENERIC** IPI to every core that may cache the translation. Each core invalidates its TLB and acknowledges. CPU 0 reuses the freed frame only after every acknowledgement arrives. ARM64 gets the same effect with one broadcast invalidate instruction and no IPI.

shared state. On its busiest paths it uses fine-grained locks and lock-free structures.

A pattern that appears often is the *seqlock* (§VIII). A writer increments a sequence counter before and after a multi-word update. A reader retries if the counter changed during its read or was odd. So a reader never takes a lock, and readers never block a writer. Zircon provides this as `concurrent::SeqLock`, with a kernel-side variant in `lib/kconcurrent`.

Its best-known use is publishing time data to user space without a syscall. Through `SetTimeValues()`, the kernel writes a shared `TimeValues` structure into a VMO that the vDSO maps read-only. The structure holds:

- `version` and `ticks_per_second`;
- the monotonic and boot tick offsets, `mono_ticks_offset` and `boot_ticks_offset`;
- the ticks-to-time ratio, `ticks_to_time_numerator` over `_denominator`.

User code reads those fields through the `fasttime` library. It computes the current monotonic or boot time entirely in user space. If an update lands in the middle of the read, such as an offset change at suspend or resume, the *seqlock* makes the reader retry. The result is a clock read with no kernel entry and no lock. It draws on every part of the memory sections, §IX to §XI: a VMO, a read-only mapping, and SMP-safe ordering.

XII. INTER-PROCESS COMMUNICATION: CHANNELS

Most work between processes in Fuchsia is a sequence of messages on a *channel*. Opening a file, drawing a frame, querying a driver and starting a component all work this way. A channel is also the only primitive that moves *capabilities* (handles) from one process to another. If kernel objects are the nouns of Zircon, channels are its main verb [1], [9].

A channel is a **bidirectional, ordered, datagram** message pipe with exactly two *peered* endpoints. The two endpoints are equal. The kernel knows no client or server; that split is a convention of user space. `zx_channel_create` returns both endpoint handles at once. There is no “connect” step, because the channel is complete from the moment it exists.

To give one end to another process, a parent writes that endpoint handle *into a message on another channel*. That is the capability transfer described below. Every other connection a

TABLE XI
KERNEL-MEMORY AND SMP CONCEPTS.

Concept	One-line description
physmap	Linear kernel mapping of all RAM; <code>paddr_to_physmap()</code> converts.
wired	Kernel frames (WIRED/HEAP/MMU/SLAB); never reclaimed.
cmpctmalloc	Space-tuned kernel heap; grows from the PMM by the page.
handle arena	HandleTableArena : fixed-size slot pool for handles, capped.
percpu	Per-core scheduler, timer, DPC, stats; one array element per CPU.
GS base / x20	Reserved register holding the current core's <code>percpu</code> pointer.
mp_ipi	IPI kinds: RESCHEDULE , GENERIC , INTERRUPT , HALT .
mp_sync_exec	Run a function on target cores via a GENERIC IPI.
TLB shootdown	Invalidate stale translations on other cores after unmap/protect.
PCID (x86)	Per-address-space TLB tag; avoids full flush on switch.
ASID (ARM64)	Per-address-space TLB tag; AsidAllocator , global kernel ASID.
broadcast TLBI	ARM64 inner-shareable invalidate; flushes all cores, no IPI.
seqlock	Lockless publish/retry; used for vDSO TimeValues .

process has grows from the bootstrap channel that `userboot` gives the first process.

A. Messages: Bytes and Handles

A channel moves discrete *messages*, never a byte stream. Each message is one atomic unit with two parallel payloads:

- up to **64 KiB** of opaque data bytes (`ZX_CHANNEL_MAX_MSG_BYTES` = 65536), and
- up to **64 handles** (`ZX_CHANNEL_MAX_MSG_HANDLES`).

A reader receives a whole message or nothing. There are no partial reads, and the kernel never merges two messages. Message boundaries survive exactly, and each endpoint delivers its messages in FIFO order. This is the main difference from a socket (Section XIII), which is a flow-controlled byte stream with no message boundaries.

B. The Capability Transfer

The handle array is what sets a channel apart. When `zx_channel_write` succeeds, the kernel *removes* the listed handles from the sender's handle table. It *inserts* them into the peer's handle table when the peer reads the message (Fig. 20). A handle value means nothing outside its own process. So the kernel gives each transferred handle a new value in the receiver, and keeps the same object and the same rights.

Right after a successful write, the sender's handle values are invalid. This is a move, not a copy. A process cannot write a handle and keep using it, because the handle has left.

C. Reading and Writing

`zx_channel_write(handle, options, bytes, num_bytes, handles, num_handles)` queues one message on the peer. A channel write *never blocks*, because the kernel applies no flow control. If the peer is alive, the kernel queues the message and the call returns at once. If the peer is closed, the call fails with

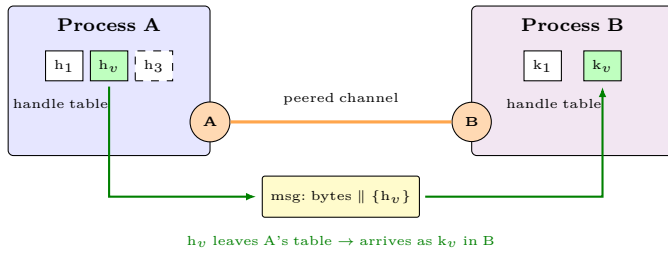


Fig. 20. Capability transfer on a channel. The message holds inline bytes and one handle, h_v . `zx_channel_write` removes the handle from process A's table. `zx_channel_read` puts it into process B's table under a new local value, k_v . The object is the same, and the rights are the same or fewer.

ZX_ERR_PEER_CLOSED. Either way the handles are consumed: the kernel discards them on success or failure.

`zx_channel_read` is the reverse. The caller supplies one buffer for the bytes and one for the handles. If either buffer is too small, the read fails with **ZX_ERR_BUFFER_TOO_SMALL**. The kernel then reports the *required* sizes through the `actual_bytes` and `actual_handles` out-parameters. The message stays at the head of the queue, unless the caller passed **ZX_CHANNEL_READ_MAY_DISCARD**. The usual pattern is a read with buffers of the maximum size, or a probe followed by a read at the reported sizes. Because messages are atomic, a reader never sees a torn or half-read message.

D. Rights Reduction in Flight: `write_etc`

The plain write and read move handles unchanged. The variants `zx_channel_write_etc` and `zx_channel_read_etc` add a per-handle *disposition*, so a sender can change each capability as it crosses. The writer supplies an array of `zx_handle_disposition_t`, with these fields:

- **operation**: **ZX_HANDLE_OP_MOVE** (consume the source) or **ZX_HANDLE_OP_DUPLICATE** (keep the source and send a copy);
- **handle**: the source handle;
- **type**: the expected `zx_obj_type_t`, or **ZX_OBJ_TYPE_NONE** to skip the check;
- **rights**: the rights the receiver gets, which may be fewer than the source's but never more, or **ZX_RIGHT_SAME_RIGHTS**;
- **result**: a status for this handle, which the kernel fills in.

The reader receives an array of `zx_handle_info_t`, which gives the object type and rights of each arriving handle. With these calls a server can hand out a read-only view of a VMO, or insist that a handle is a channel. It does so in the same atomic operation that delivers the bytes, with no extra `zx_handle_duplicate` or `zx_handle_replace` round trip.

E. The Synchronous Call: `zx_channel_call`

A request and reply built from a separate write and read has a race. After writing a request, the client must read the reply. But an unrelated message may arrive first, and the replies to two requests may arrive in either order. `zx_channel_call` removes the race. It performs the write and the matching read as one correlated operation (Fig. 21).

The key that ties the reply to the request is a **transaction id** (`zx_txid_t`) in the *first four bytes* of the message. The kernel overwrites that field in the outgoing message with a new txid between `0x80000000` and `0xFFFFFFFF`. The kernel guarantees that no other `zx_channel_call` in progress on the

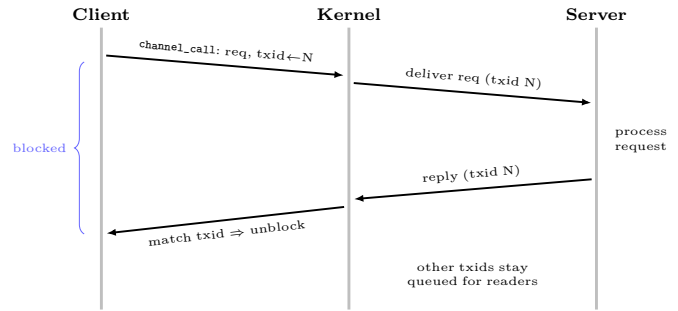


Fig. 21. The `zx_channel_call` transaction. The kernel stamps the request with txid N, blocks the caller, and returns only the reply whose header holds N. Other messages stay in the queue for ordinary `zx_channel_read` callers, so no reply can be taken by the wrong reader.

same endpoint uses that txid. The kernel then blocks the caller until a message arrives whose first four bytes hold the *same* txid, and delivers it as the reply. Messages with other txids stay queued for ordinary readers. The deadline is an absolute monotonic `zx_instant_mono_t`. If the call returns **ZX_ERR_TIMED_OUT**, it has stopped waiting, but the request was already sent.

F. Where FIDL Lives, and Where It Does Not

The kernel primitive and the protocol layer are separate. The **kernel knows nothing about FIDL**. It moves opaque bytes and handles and never parses a payload. FIDL (the Fuchsia Interface Definition Language) is a wire format that user space layers on top.

A FIDL method call is a channel message whose bytes start with a transaction header. The header holds a `zx_txid_t`, a flags and magic field, and a 64-bit method *ordinal*. The encoded arguments follow. Any `handle`, `client_end` or `server_end` arguments travel in the message's handle array. Bindings for C++, Rust and Go, generated by `fidlc`, do the encoding and decoding.

A two-way FIDL method is a `zx_channel_call`: the binding fills in the ordinal, the kernel assigns the txid, and the caller blocks for the matching reply. One-way methods and events are plain writes. Because the kernel only moves capabilities and FIDL defines the typed protocol, one channel type serves a file system protocol, a graphics protocol and a driver protocol with no kernel changes.

G. Signals, Closure, and Epitaphs

A channel endpoint has three main signals, for use with the waiting calls of Section XIV:

- **ZX_CHANNEL_READABLE**: at least one message is queued;
- **ZX_CHANNEL_WRITABLE**: the peer is open;
- **ZX_CHANNEL_PEER_CLOSED**: the other endpoint and all its handles are gone.

PEER_CLOSED stays set once asserted, and it is the standard end-of-stream signal. Messages already queued stay readable after the peer closes. So a correct reader drains the queue *and* sees **PEER_CLOSED** before it decides the channel is finished.

The FIDL *epitaph* is a convention built on this. Before a server closes a protocol, it may send one last message that holds a `zx_status_t` reason. The client then learns *why* the connection ended, not only *that* it ended. The kernel has

TABLE XII
CHANNEL SYSCALLS.

Syscall	Action
<code>zx_channel_create</code>	Make a peered pair; return both endpoints.
<code>zx_channel_write</code>	Enqueue one message (bytes + moved handles); never blocks.
<code>zx_channel_read</code>	Dequeue one whole message; <code>BUFFER_TOO_SMALL</code> returns needed sizes.
<code>zx_channel_write_etc</code>	Write with per-handle dispositions (move/dup, rights cut, type check).
<code>zx_channel_read_etc</code>	Read returning per-handle type/rights info.
<code>zx_channel_call</code>	Atomic write+blocked read correlated by header txid.

no idea of an epitaph. To the kernel it is the last message written before the handle closes, after which the client sees `ZX_CHANNEL_PEER_CLOSED`.

H. Limits and Flow Control

Writes never block, so the kernel does not slow a fast producer. There is no built-in backpressure. The hard limits are the per-message caps above (64 KiB and 64 handles) and a per-channel bound on buffered messages, whose value the kernel does not publish. A write that crosses that bound raises a policy exception in the writing thread. Flow control above that is left to FIDL and user space. Protocols bound the number of outstanding requests with hanging gets or explicit acknowledgements. The kernel’s job ends at moving bytes and handles reliably and in order.

Listing 15. A channel, a handle transfer, and a call.

```

zx_handle_t a, b;
zx_channel_create(0, &a, &b); // a,b are peers

// --- send some bytes plus a handle (capability move) ---
uint8_t payload[16] = { /* ... */ };
zx_handle_t vmo;
zx_vmo_create(4096, 0, &vmo);
// vmo is MOVED out of our table on success:
zx_channel_write(a, 0, payload, sizeof(payload),
                &vmo, 1);

// --- synchronous request/reply (what FIDL two-way uses) ---
uint8_t req[64], rep[64];
zx_channel_call_args_t args = {
    .wr_bytes = req, .wr_num_bytes = sizeof(req),
    .wr_handles = NULL, .wr_num_handles = 0,
    .rd_bytes = rep, .rd_num_bytes = sizeof(rep),
    .rd_handles = NULL, .rd_num_handles = 0,
};
uint32_t nb, nh;
// kernel stamps txid into req[0..3], blocks for the
// matching reply, then fills nb/nh:
zx_status_t st = zx_channel_call(
    b, 0, zx_deadline_after(ZX_MSEC(50)),
    &args, &nb, &nh);

```

XIII. STREAMING AND LIGHTWEIGHT IPC: SOCKETS, FIFOS, STREAMS, EVENTPAIRS

Channels suit typed protocols that pass capabilities. They suit other jobs poorly: a bulk byte pipe, a small queue of fixed-format commands, or a file with a read position. Zircon provides four more transport objects, one for each of these shapes of data movement. Table XIII compares all five, and this section explains the four new ones [2], [10].

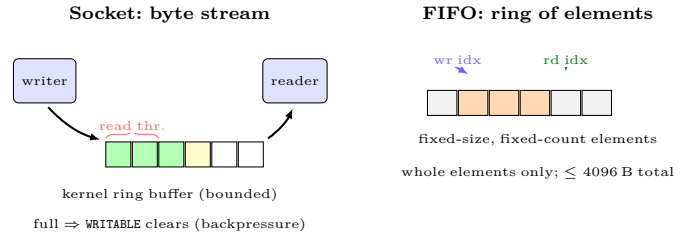


Fig. 22. Left: a socket is a bounded kernel ring with read and write thresholds. When the buffer is full, the kernel clears `ZX_SOCKET_WRITABLE`, which holds the writer back. Right: a FIFO is a fixed ring of fixed-size elements with a write index and a read index. Data moves one whole element at a time.

A. Sockets: Flow-Controlled Byte Streams

A **socket** is a peered, bidirectional *byte* transport backed by a kernel buffer (Fig. 22, left). A channel never blocks a writer and never slows it down. A socket does apply **flow control**. Each endpoint has a bounded buffer, and when the buffer fills, the kernel clears `ZX_SOCKET_WRITABLE` so that a waiting writer knows to pause. This makes sockets the natural choice for `stdio` and for byte streams in which a slow consumer must be able to hold back a fast producer.

`zx_socket_create(options, &a, &b)` returns a peered pair. The one option chooses the framing:

- `ZX_SOCKET_STREAM` (the default) is a plain byte stream: writes merge, and a read may return any prefix of the buffered bytes;
- `ZX_SOCKET_DATAGRAM` keeps each write as a separate record.

Data moves with `zx_socket_read` and `zx_socket_write`. Unlike the channel calls, these may move only part of the data and report the count they moved. **Sockets cannot move handles**. They move bytes only, so passing a capability needs a channel.

A socket has more signals than a channel, because of flow control and half-close:

- `ZX_SOCKET_READABLE`, `ZX_SOCKET_WRITABLE` and `ZX_SOCKET_PEER_CLOSED`;
- the half-close pair `ZX_SOCKET_WRITE_DISABLED` and `ZX_SOCKET_PEER_WRITE_DISABLED`;
- the tunable `ZX_SOCKET_READ_THRESHOLD` and `ZX_SOCKET_WRITE_THRESHOLD`.

A program sets the thresholds with `zx_object_set_property`. A reader can then wait for “at least *N* bytes buffered”, and a writer for “at least *N* bytes of space”. This avoids a burst of wake-ups for many tiny reads.

Half-close uses `zx_socket_set_disposition(handle, disposition, disposition_peer)`. Passing

`ZX_SOCKET_DISPOSITION_WRITE_DISABLED` shuts the write direction of an endpoint. Further writes to that endpoint fail with `ZX_ERR_BAD_STATE`. The peer drains the bytes already buffered and then sees `ZX_SOCKET_PEER_WRITE_DISABLED`. That is the clean “no more data in this direction” that a bidirectional byte protocol needs, as in a TCP shutdown. `ZX_SOCKET_DISPOSITION_WRITE_ENABLED` turns writing back on, but only while the buffer is empty.

B. FIFOs: Low-Overhead Control Queues

A **FIFO** is a deliberately small primitive. It is a fixed ring of *fixed-size elements* shared by two peered endpoints, built for the lowest cost per message (Fig. 22, right). `zx_fifo_create(elem_count, elem_size, options, &a,`

&b) fixes the geometry when the FIFO is created. Both values must be nonzero, and the total `elem_count × elem_size` may not exceed 4096 bytes, so the whole ring fits in one page. `zx_fifo_write` and `zx_fifo_read` move *whole elements only*, and report how many elements they moved. A FIFO has the usual `ZX_FIFO_READABLE`, `ZX_FIFO_WRITABLE` and `ZX_FIFO_PEER_CLOSED` signals.

A FIFO is meant to be the *control plane* beside a shared-memory data plane. Block device drivers and network drivers pair a FIFO with a VMO. The bulk payload lives in the VMO. The FIFO holds small fixed-size requests and completions, such as “transmit the buffer at offset *X*, length *Y*” and “here is completion status *Z*”. The ring is one page and the elements have a fixed size. So each operation on the hot path is a small copy and an index update, which is much cheaper than a channel message. A high-rate driver queue needs exactly that.

C. Streams: A Cursor over a VMO

A **stream** adds POSIX file behaviour to a VMO. A VMO is a flat memory object that a program addresses at any offset. A `zx_stream` adds a *seek offset*, a cursor, so that sequential `read`, `write` and `lseek` work as a C programmer expects.

`zx_stream_create(options, vmo, seek, &out)` binds a stream to an existing VMO at a starting offset. The options are `ZX_STREAM_MODE_READ`, `ZX_STREAM_MODE_WRITE` and, optionally, `ZX_STREAM_MODE_APPEND`. Each mode needs the matching right on the VMO. Data moves with the scatter/gather calls `zx_stream_readv` and `zx_stream_writev`, over arrays of `zx_iovec_t`. `zx_stream_seek` moves the cursor relative to the start, the current position or the end.

Two behaviours make streams the file primitive of Fuchsia. First, a write past the current end of the content *grows* the stream’s content size automatically, up to the size of the VMO. A file therefore grows as it is written, with no explicit `resize`.

Second, the VMO may be *pager-backed*, which is how file contents are held. Then a read through the stream makes the user-space pager fault the pages in on demand. So a file system exposes a file as a pager-backed VMO and hands clients a stream over it. Clients get demand-paged file I/O with the familiar `read`, `write` and `seek` calls, and no extra copy. The page cache *is* the VMO.

D. Eventpairs: Liveness and Cancellation Tokens

An **eventpair** is the smallest peered object. Its two endpoints share nothing but signal state. `zx_eventpair_create(0, &a, &b)` returns the pair. Either side can assert signal bits on the *other* end with `zx_object_signal_peer`: `ZX_EVENTPAIR_SIGNALED`, or any of `ZX_USER_SIGNAL_0` to `ZX_USER_SIGNAL_7`. When all handles to one end are closed, the kernel asserts `ZX_EVENTPAIR_PEER_CLOSED` on the other end automatically.

That automatic signal makes an eventpair a good *liveness* or *cancellation* token. A holder waits on `ZX_EVENTPAIR_PEER_CLOSED`. If the process at the other end dies or drops its end, for any reason including a crash, the waiter wakes. No heartbeat protocol or polling is needed.

The graphics stack uses this. A `ViewRef` and its `ViewRefControl` are the two ends of an eventpair. When a view is destroyed, its `ViewRefControl` closes, so every holder of the `ViewRef` sees `ZX_EVENTPAIR_PEER_CLOSED`. (Flatland’s view and viewport creation tokens, by contrast, are channel ends.) The

TABLE XIII
THE FIVE USER-SPACE IPC TRANSPORTS COMPARED.

Object	Transfer unit	Hdls?	Flow ctl?	Typical use
Channel	message (atomic)	yes	no	FIDL protocols, capability passing
Socket	byte or datagram	no	yes	stdio, byte streams
FIFO	fixed element	no	yes	driver control queues
Stream	cursor over VMO	no	n/a	file read-/write/seek
Eventpair	1-bit signals	no	n/a	liveness and cancel tokens

same pattern serves many “cancel this work when the requester goes away” relationships across the system.

To sum up the five transports:

- a channel is the only one that moves handles, the only one with strict message boundaries, and the only one with no flow control;
- sockets and FIFOs move bytes or elements and do apply backpressure;
- a stream is file I/O over a VMO;
- an eventpair is pure signalling between processes.

To choose among them, ask what the unit of transfer is and whether capabilities must travel with the data.

XIV. SIGNALS, WAITING, AND PORTS

The last two sections used “signals” and “waiting” without defining them. This section defines them. *Every* waitable kernel object reports its state through the same 32-bit signals word. That includes channels, sockets, FIFOs, eventpairs, timers, VMOs, processes and threads. A program waits on any of them with the same few syscalls. Once you know these calls, you can wait on anything [2], [11].

A. Object Signals

Each waitable object has a `zx_signals_t`: a 32-bit word in which each bit is one boolean condition. Most bits are *object-defined*, so their meaning depends on the object type. A few bits are reserved for user code. Some object-defined bits:

- `ZX_CHANNEL_READABLE`;
- `ZX_SOCKET_WRITABLE`;
- `ZX_VMO_ZERO_CHILDREN`;
- `ZX_TASK_TERMINATED`, asserted when a process or thread exits;
- `ZX_TIMER_SIGNALED`.

The bits `ZX_USER_SIGNAL_0` to `ZX_USER_SIGNAL_7` have no meaning to the kernel. User code may assert them freely, with `zx_object_signal` on the object itself or `zx_object_signal_peer` on the far end of a peered object. An eventpair works this way.

Signals are **level-triggered**. A bit stays asserted for exactly as long as its condition holds. `ZX_CHANNEL_READABLE` is high while a message waits, and it goes low when the queue empties. It is not a one-shot “message arrived” event. Levels make it safe to wait again. If a program asks again while the condition still holds, the kernel answers at once, so no edge can be missed between two waits.

B. Synchronous Waiting

The direct way to block is `zx_object_wait_one(handle, signals, deadline, &observed)`. The thread sleeps until *any* bit in `signals` is asserted on `handle`, or until the absolute `deadline` passes. The call returns the full set of signals observed when it woke. To wait on several objects at once, `zx_object_wait_many` takes an array of `zx_wait_item_t`. Each item is a handle and its own `waitfor` mask. The array holds at most 64 items (`ZX_WAIT_MANY_MAX_ITEMS`), and the call reports the observed signals for each item. Both calls need `ZX_RIGHT_WAIT` on every handle.

Synchronous waits are simple, and they suit a thread that serves one or two objects. They scale poorly. A thread blocked in `wait_one` can do nothing else. `wait_many` stops at 64 objects, and the program must rebuild the array whenever the set changes. A server with thousands of connections needs another mechanism.

C. Ports: The Asynchronous Dispatch Hub

That mechanism is the **port**. It is Zircon’s counterpart of `epoll` or `kqueue`, and every real event loop on the system is built on one. A port is a queue of *packets*. A thread does not block *on an object*. Threads register interest in objects ahead of time, and then block *on the port* and take the packets that have arrived.

`zx_port_create` makes an empty port. `zx_object_wait_async(handle, port, key, trigger_signals, options)` registers interest. It means: “when any bit in `trigger_signals` is asserted on `handle`, queue a packet to `port`, tagged with this 64-bit `key`.” The caller chooses the `key`. It is the only link back to application state, and is typically a pointer or a table index that names the connection that fired. When the condition holds, the kernel queues a `zx_port_packet_t` with that `key`. For a signal packet, a `zx_packet_signal_t` inside it reports the `trigger` mask and the `observed` signals. `zx_port_wait(port, deadline, &packet)` takes the next packet off the queue.

By default a registration is **one-shot**. The kernel delivers one matching packet and the registration is used up. The handler registers again with another `zx_object_wait_async` after it has done its work. This default is the safe one, because the kernel cannot deliver a second packet for an event the handler has not yet processed.

The option `ZX_WAIT_ASYNC_EDGE` asks for edge-triggered delivery instead. The kernel then sends a packet only when the signal *changes* from inactive to active, and never for a condition already true at registration. This suits non-blocking I/O loops that drain everything on each wake-up. Other options add data; `ZX_WAIT_ASYNC_TIMESTAMP`, for example, stamps the packet with the monotonic time of the trigger. `zx_port_cancel(port, handle, key)` withdraws a registration that has not fired yet. A server needs it when it tears down a connection that still has a registration outstanding.

D. One Queue, Many Packet Kinds

A port is more than a better `wait_many`, because several unrelated event sources deliver into the *same* queue. Each packet has a `type` field that names its source:

- `ZX_PKT_TYPE_SIGNAL_ONE`: an object signal, registered with `zx_object_wait_async` (the common case above);

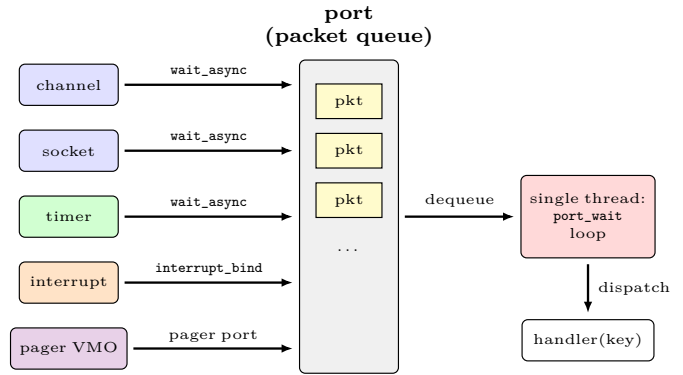


Fig. 23. A port-based reactor. Objects of many types register with the port: most with `zx_object_wait_async`, an interrupt with `zx_interrupt_bind`, and a pager-backed VMO through its pager’s port. Each ready condition queues a keyed packet. One thread takes packets with `zx_port_wait` and dispatches on the key. In hardware terms, the port is an interrupt aggregator: many sources feed one queue that one handler loop serves.

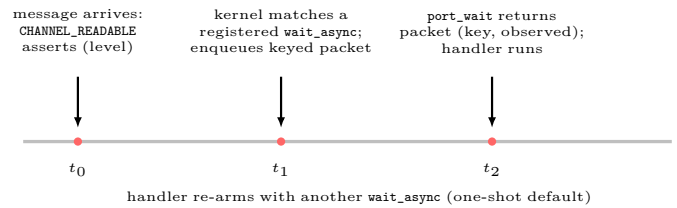


Fig. 24. One signal event through a port. A level-triggered signal is asserted on an object. The kernel queues a packet to the registered port. `zx_port_wait` returns the packet to the dispatch loop, which handles it and registers again.

- `ZX_PKT_TYPE_USER`: an application packet queued directly with `zx_port_queue`, which a loop uses as its “wake up, there is work” doorbell;
- `ZX_PKT_TYPE_INTERRUPT`: a hardware interrupt object bound to the port (Section XV);
- `ZX_PKT_TYPE_PAGE_REQUEST`: a page request from a pager-backed VMO to its user-space pager;
- exception packets, and guest packets from the hypervisor, bound to the port.

A timer signal, a channel message, an interrupt, a pager fault and a user packet all arrive as packets on one port. So a process can run its *entire* concurrency model on one `zx_port_wait` loop. Fig. 24 follows one such event from start to end.

E. The Async Runtime Tie-In

Fuchsia programs are built this way in practice. The C `libasync` dispatcher and the Rust `fuchsia_async` executor each own *one* port. Each dispatcher thread runs *one* `zx_port_wait` loop. A future that waits for a channel to become readable is, underneath, a `zx_object_wait_async` whose key names that future’s waker. When the packet arrives, the executor looks up the key, polls the matching future, and moves it forward.

Posting work to a dispatcher from another thread sends a `ZX_PKT_TYPE_USER` packet with `zx_port_queue`. So the `async/await` syntax and the port are the same mechanism seen at two levels. A process’s main loop is a `zx_port_wait` loop that dispatches keyed packets.

TABLE XIV
REPRESENTATIVE OBJECT SIGNALS AND THE WAIT/PORT SYSCALLS.

Object	Representative signals
Channel	READABLE, WRITABLE, PEER_CLOSED
Socket	READABLE, WRITABLE, READ_THRESHOLD, PEER_CLOSED
Timer	ZX_TIMER_SIGNED
Task	ZX_TASK_TERMINATED
VMO	ZX_VMO_ZERO_CHILDREN
Eventpair	SIGNED, PEER_CLOSED
Any	ZX_USER_SIGNAL_0..7 (user-assertable)
Syscall	Action
zx_object_signal	Assert/clear user signals on an object (peer variant).
zx_object_wait_one	Block on one object for any of a signal mask.
zx_object_wait_many	Block on an array of wait_items.
zx_port_create	Create a packet queue.
zx_object_wait_async	Register keyed interest; queue a packet on assert.
zx_port_wait	Dequeue the next packet (with deadline).
zx_port_queue	Inject a ZX_PKT_TYPE_USER packet.
zx_port_cancel	Withdraw a pending async wait.

XV. INTERRUPTS AND EXCEPTIONS

Two kinds of event arrive at the kernel unannounced and must reach the right handler. Devices raise *hardware interrupts*. A running thread raises an *exception*, synchronously, when it faults. Zircon exposes both through kernel objects. So an ordinary user-space driver can handle a device IRQ, and an ordinary user-space debugger can handle a page fault, with no change to kernel code. This section covers both paths, and the chain of handlers that makes exceptions debuggable [15], [16].

A. Hardware Interrupts as Kernel Objects

Almost all Fuchsia drivers run in user space. So how does a user thread learn that its device raised an IRQ? It waits on an **interrupt object**. `zx_interrupt_create(resource, vector, options, &out)` creates one and binds it to a physical vector. The `resource` argument is an IRQ *resource*, a capability that proves the caller may claim that vector. Section XVII describes this gating of hardware access. Owning an interrupt is therefore a capability, not a privilege of being root.

The `options` choose the trigger mode (`ZX_INTERRUPT_MODE_EDGE_*` or `ZX_INTERRUPT_MODE_LEVEL_*`), or ask for a software-only `ZX_INTERRUPT_VIRTUAL` interrupt.

A driver uses the object in one of two ways. In the simple way, a dedicated thread calls `zx_interrupt_wait(handle, ×tamp)`. The call blocks until the IRQ fires and returns the kernel's timestamp of the trigger. In the scalable way, the driver *binds* the interrupt to a port with `zx_interrupt_bind(interrupt, port, key, options)`. From then on each firing queues a `ZX_PKT_TYPE_INTERRUPT` packet on that port. The interrupt then sits in the same `zx_port_wait` loop (Section XIV) as the driver's channels and timers.

Either way, after the driver has serviced the device, it must call `zx_interrupt_ack` to re-arm the object for the next IRQ. The ack is backpressure, not bookkeeping. The line stays masked between the firing and the ack, whether it is level- or edge-triggered. So the kernel will not deliver the interrupt again until

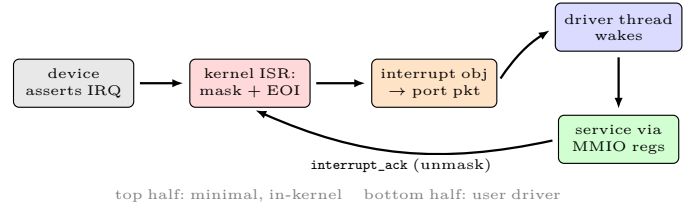


Fig. 25. The user-space interrupt path. A minimal kernel ISR masks the line, acknowledges the controller, and signals the interrupt object (or queues a port packet). The driver thread services the device through its MMIO mapping and calls `zx_interrupt_ack` to re-arm the line. This is a top-half and bottom-half split, with a kernel object between the halves.

the driver says it is ready, and a faulty device cannot livelock the system. `zx_interrupt_destroy` tears the object down, and any waiter wakes with `ZX_ERR_CANCELED`. `zx_interrupt_trigger` fires a virtual interrupt in software, which is how drivers are tested without hardware.

B. The Interrupt Flow

Fig. 25 follows a real IRQ. The device asserts its line. A *minimal* ISR in the kernel runs. It masks the line and acknowledges the interrupt controller, for example with an EOI to the APIC or GIC. Then it signals the interrupt object, or queues its port packet.

The kernel does *not* touch the device's own registers, because it does not know how. That is the job of the driver thread, which now wakes. The driver reads and writes the device's registers through its *MMIO* mapping to handle the condition: it drains a FIFO, reads a status word, or restarts DMA. Finally it calls `zx_interrupt_ack` to unmask the line for the next event.

An RTL engineer will know this two-stage handshake. A fast top half only masks the source so that the controller can move on. A deferred bottom half does the real work on the device. Here the bottom half is an unprivileged user thread, and a kernel object, not a function pointer, separates the two halves.

C. MSI and MSI-X for PCI

Modern PCI devices do not assert a shared pin. They signal an interrupt by writing a message to a special address; these are Message-Signaled Interrupts (MSI). Zircon handles a contiguous block of them with two syscalls. `zx_msi_allocate` reserves a block of MSIs whose size is a power of two from 1 to 32. It needs the platform's MSI resource, and it returns an *MSI allocation* object.

`zx_msi_create` then takes one interrupt object out of that allocation for a given MSI id, connected to the device's MSI or MSI-X capability. The result behaves like any other interrupt object: a driver can wait on it, bind it to a port and ack it. So the bus driver's dispatch loop is the same whether the source is a legacy pin or one entry in an MSI-X table.

D. Exceptions: Synchronous Thread Faults

An **exception** is the opposite of an interrupt. The faulting thread raises it *synchronously*. A thread may execute an illegal instruction, touch an address the VM cannot map, break a job policy, or hit a breakpoint. The kernel then *suspends that thread* and looks for a handler. The exception types include:

- `ZX_EXCP_FATAL_PAGE_FAULT`: a page fault the VM subsystem could not resolve;

- `ZX_EXCP_UNDEFINED_INSTRUCTION`;
- `ZX_EXCP_GENERAL`: a general or protection fault;
- `ZX_EXCP_UNALIGNED_ACCESS`;
- `ZX_EXCP_SW_BREAKPOINT` and `ZX_EXCP_HW_BREAKPOINT`, the debug events;
- `ZX_EXCP_POLICY_ERROR`, raised when a thread breaks a rule that its job set, such as calling a forbidden syscall.

Debuggers also receive the lifecycle events `ZX_EXCP_THREAD_STARTING`, `ZX_EXCP_THREAD_EXITING` and `ZX_EXCP_PROCESS_STARTING` through the same mechanism, so that they can attach at the right moment.

E. Exception Channels

A handler does not register a callback. It registers a *channel*. `zx_task_create_exception_channel(task, options, &out)` attaches an exception channel to a thread, a process or a job. Passing `ZX_EXCEPTION_CHANNEL_DEBUGGER` on a process creates the special *debugger* channel, which gets the first chance. When a matching exception occurs, the kernel writes one message to the channel. The message holds a `zx_exception_info_t`, with the faulting `pid`, `tid` and exception `type`, and a *handle to an exception object*.

From that handle, the handler gets the faulting thread and process with `zx_exception_get_thread` and `zx_exception_get_process`. It gets the architectural detail as a `zx_exception_report_t`, through `zx_object_get_info` with `ZX_INFO_THREAD_EXCEPTION_REPORT`. The report gives the faulting address, the registers and the synthetic codes.

The handler holds a real thread handle, so it can do anything a debugger does. It can read and rewrite the registers and FPU state with `zx_thread_read_state` and `zx_thread_write_state`. It can inspect memory through the process handle, or set a breakpoint. It then chooses an outcome by setting the exception object’s `ZX_PROP_EXCEPTION_STATE` property:

- `ZX_EXCEPTION_STATE_HANDLED` means “I fixed it; retry the faulting instruction”;
 - `ZX_EXCEPTION_STATE_TRY_NEXT` means “not mine; pass it on”.
- Closing the exception handle is what releases the thread. If the handler leaves the state at its default, `TRY_NEXT`, closing the handle passes the exception to the next handler in the chain.

F. The Propagation Chain

The kernel offers an exception to handlers in a fixed order. Each handler gets a chance before the fault becomes fatal (Fig. 26). The order is:

- 1) the process **debugger** channel (the first chance);
- 2) the **thread** exception channel;
- 3) the **process** exception channel;
- 4) the process debugger again, but only for an exception marked second-chance;
- 5) each **job** exception channel up the task tree, from the parent job to the **root job**.

At each stage, a handler that sets `HANDLED` stops the search and resumes the thread. The exception moves to the next stage if the handler sets `TRY_NEXT`, closes the handle without deciding, or is not registered at all.

If no handler in the chain handles the exception, the kernel kills the thread’s process. The system’s last-resort handler, `crashsvc`, is registered high in the job tree. It catches the

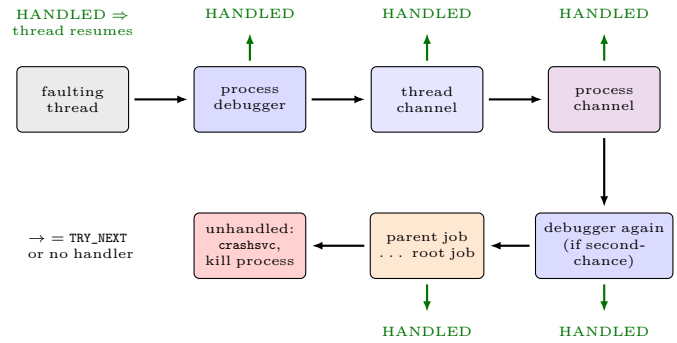


Fig. 26. The exception propagation chain. The kernel offers a fault first to the process debugger, then to the thread and process channels. A fault marked second-chance goes to the process debugger again. Then each job channel up to the root job gets it. Setting `ZX_EXCEPTION_STATE_HANDLED` resumes the thread, and `TRY_NEXT` passes the fault on. If no handler takes it, `crashsvc` reports the crash and the process is killed.

TABLE XV
EXCEPTION TYPES AND THE INTERRUPT/EXCEPTION SYSCALLS.

Exception type	Cause
FATAL_PAGE_FAULT	Page fault the VM cannot satisfy.
UNDEFINED_INSTRUCTION	Illegal/undefined opcode.
GENERAL	General protection / arch fault.
UNALIGNED_ACCESS	Misaligned memory access.
SW_/HW_BREAKPOINT	Debug breakpoint hit.
POLICY_ERROR	Job-policy violation (e.g. bad syscall).
Syscall	Action
<code>zx_interrupt_create</code>	Bind a vector (via IRQ resource) to an object.
<code>zx_interrupt_wait</code>	Block until the IRQ fires; return timestamp.
<code>zx_interrupt_bind</code>	Route firings as packets to a port.
<code>zx_interrupt_ack</code>	Re-arm the line after servicing.
<code>zx_interrupt_trigger</code>	Fire a virtual interrupt in software.
<code>zx_msi_allocate/create</code>	Reserve and carve PCI MSI/MSI-X vectors.
<code>create_exception_channel</code>	Register a handler channel on a task.
<code>zx_thread_read/write_state</code>	Inspect/modify faulting registers.

unhandled fault and prints a symbolized backtrace and a register dump to the log. It also emits a minidump for later analysis, and then lets the process die. This is why a Fuchsia crash produces a readable stack, although the kernel has no debugger in it. The debugger is one more client of an exception channel, and the crash reporter is the client at the top of the chain.

XVI. TIME, TIMERS, AND CLOCKS

Time in Zircon is not a single number. The kernel keeps several *timelines*, each with different guarantees. It lets programs read the hardware timelines cheaply. For timelines it cannot keep itself, above all UTC, it provides a *clock object*. These differences matter: a deadline measured on the wrong timeline is a bug waiting to happen [1], [20].

A. Time Bases

The basic timeline is **monotonic** time, read with `zx_clock_get_monotonic`. It counts nanoseconds since boot,

and it never goes backward or jumps. Every deadline in the system is measured against it. Monotonic time *pauses while the system is suspended*. So it measures *running* time, not elapsed wall-clock time.

Boot time, read with `zx_clock_get_boot`, is its companion. It *does* count suspended intervals: it is nanoseconds since power-on, whether or not the system slept. Code that must measure real elapsed time across a sleep uses boot time; a watchdog or a session timeout are examples. Code that times CPU-bound work uses monotonic time.

UTC is deliberately *not* a kernel primitive. The kernel does not know what year it is, and its ABI does not depend on a battery-backed clock. Instead, UTC is a *clock object* (described below) that user space maintains. A user-space *timekeeper* service adjusts it from a network or hardware time source. All wall-clock policy therefore stays out of the kernel: leap handling, choice of source, and slewing. Fuchsia Subsystems Volume VI (Time & Timekeeping) describes that service.

Programs read monotonic time, boot time and the hardware **ticks without a kernel trap**. The vDSO reads the hardware counter directly in user space. `zx_ticks_get` returns the raw counter, and `zx_ticks_per_second` returns its frequency. A nanosecond value comes from the counter through a linear scale that the kernel publishes in the vDSO. So a reading costs one counter read and a multiply, with no syscall and no mode switch. A tight loop can take millions of timestamps per second this way, which tracing and profiling need.

B. Timers

A **timer object** turns the passage of time into a signal that a program can wait on. Time then fits the same port and signal machinery as everything else. `zx_timer_create(options, clock_id, &out)` makes a timer and picks its timeline: `ZX_CLOCK_MONOTONIC` or `ZX_CLOCK_BOOT`. `zx_timer_set(handle, deadline, slack)` arms it to fire at an absolute **deadline**. When the deadline passes, the timer asserts `ZX_TIMER_SIGNALED`. A program can wait for that directly with `zx_object_wait_one`. More often it registers the timer on a port with `zx_object_wait_async` and handles it in its event loop.

`zx_timer_cancel` disarms the timer and clears the signal. Arming it again with a new `zx_timer_set` also clears any earlier signal.

The **slack** argument sets how precise the wake-up must be. The caller gives a tolerance window, and the kernel may fire anywhere inside it. The kernel uses that freedom to merge nearby timers into one wake-up. Fewer wake-ups let the CPU stay longer in a low-power idle state, which saves battery. The *slack mode*, chosen in the timer's options, sets where the window lies:

- `ZX_TIMER_SLACK_CENTER` allows firing anywhere in [deadline – slack, deadline + slack];
- `ZX_TIMER_SLACK_EARLY` allows firing only before the deadline;
- `ZX_TIMER_SLACK_LATE` allows firing only after it.

A periodic UI animation can accept a generous late slack. A hard timeout uses zero.

C. Deadlines Are Pervasive

Time is not limited to timer objects. The blocking syscalls take an absolute `zx_time_t` deadline on the monotonic timeline. Examples are `zx_object_wait_one`, `zx_port_wait`,

`zx_channel_call` and the futex waits. (`zx_interrupt_wait` is the exception: it takes no deadline.) Two special values follow one convention. `ZX_TIME_INFINITE` blocks forever. `ZX_TIME_INFINITE_PAST`, or any deadline already in the past such as 0, turns the call into a non-blocking poll. Such a call returns at once with `ZX_ERR_TIMED_OUT` if the condition does not already hold.

Deadlines are *absolute*, not relative, so a wait that is interrupted and retried does not drift. The helper `zx_deadline_after(duration)` turns a relative duration into an absolute monotonic deadline once, at the start. The scheduler runs deadline profiles (Section VII-D) against the same absolute time. So “when must this thread wake?” and “when must this thread run?” are measured in the same unit.

D. Clock Objects: UTC Without Time Jumps

The richest time primitive is the **clock object**. It is a clock that the kernel hosts and user space maintains. It maps a *reference* timeline onto a *synthetic* timeline, through a linear transform that its owner can adjust (Fig. 27). The reference is monotonic time, or boot time for a clock created with `ZX_CLOCK_OPT_BOOT`. The calls are:

- `zx_clock_create` makes a clock;
- `zx_clock_read` returns its current synthetic value;
- `zx_clock_update` adjusts the transform;
- `zx_clock_get_details` returns the full transform, and an *error bound* that estimates how far the value may be from the true time.

The transform is the affine map

$$\text{synthetic} = s_0 + \text{slope} \cdot (\text{reference} - r_0), \quad (1)$$

Here (r_0, s_0) is an anchor point: “at reference instant r_0 , the synthetic clock read s_0 ”. **slope** is the ratio of the two rates. `zx_clock_update` can change the offset s_0 and the **slope** separately.

This lets a timekeeper *discipline* UTC **without time jumps**. Suppose the timekeeper finds that the clock is slow. Stepping the clock forward would make timestamps skip, and stepping it back would make them repeat, which breaks their order. Instead the timekeeper sets the **slope** slightly above one. The synthetic clock then *slews* into agreement over seconds or minutes, and the timekeeper restores a slope of one. In hardware terms, the clock behaves like a numerically controlled oscillator whose rate and phase software trims.

Options at creation limit which adjustments are allowed:

- `ZX_CLOCK_OPT_MONOTONIC` forbids the synthetic value from ever going backward;
- `ZX_CLOCK_OPT_CONTINUOUS` also forbids *any* jump in either direction, so only slewing is allowed;
- `ZX_CLOCK_OPT_BOOT` makes boot time the reference, so the clock keeps counting while the system is suspended;
- `ZX_CLOCK_OPT_AUTO_START` makes the clock follow its reference one to one until it is first adjusted.

UTC is one of these objects, and its reference is boot time. So UTC keeps counting while the system is suspended, as a wall clock must. The platform creates the clock and hands it to the processes that need wall-clock time, and the timekeeper keeps updating it. A process reads it with `zx_clock_read`, like any other clock. When it needs to know how far to trust the value, it reads the error bound from `zx_clock_get_details`. None of this needs any notion of calendar time inside the kernel.

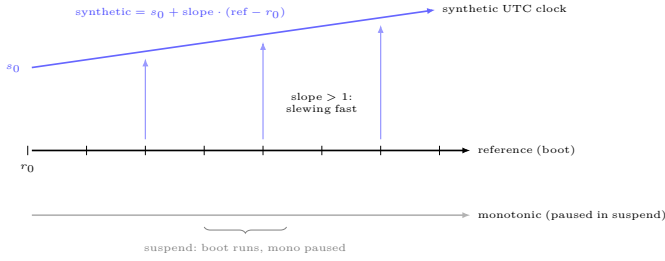


Fig. 27. The Zircon timelines. Boot time is the reference of the UTC clock. Monotonic time is boot time less the time spent suspended. A UTC clock object is a synthetic timeline, mapped from the reference by the affine transform of Eq. (1). Adjusting the slope slews UTC into agreement with no jump.

TABLE XVI
TIME, TIMER, AND CLOCK SYSCALLS AND SLACK MODES.

Syscall / constant	Meaning
<code>zx_clock_get_monotonic</code>	Monotonic ns since boot; paused in suspend.
<code>zx_clock_get_boot</code>	Boot ns since power-on; includes suspend.
<code>zx_ticks_get_per_second</code>	/ Raw hardware counter + frequency (vDSO, no trap).
<code>zx_deadline_after</code>	/ Relative duration → absolute mono deadline.
<code>ZX_TIME_INFINITE_PAST</code>	/ Block forever / poll (return at once).
<code>zx_timer_create</code>	Timer object on monotonic or boot timeline.
<code>zx_timer_set</code>	Arm at absolute deadline with slack.
<code>zx_timer_cancel</code>	Disarm; clear <code>ZX_TIMER_SIGNED</code> .
<code>SLACK_CENTER/EARLY/LATE</code>	Coalescing window: both sides / early / late.
<code>zx_clock_create</code>	User-maintained clock with affine transform.
<code>zx_clock_read</code>	Current synthetic value (e.g. UTC).
<code>zx_clock_update</code>	Adjust offset and/or slope (slew).
<code>zx_clock_get_details</code>	Transform parameters + error bound.

XVII. HARDWARE ACCESS: RESOURCES, MMIO, BTIS, AND THE IOMMU

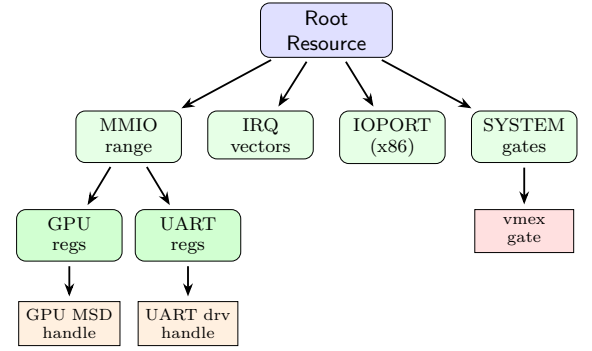
Zircon runs its drivers in user space, but the drivers must still reach real hardware: device registers, interrupt lines and DMA engines. They must do so without undoing the isolation that was the reason for moving them out of the kernel. Zircon solves this with a small family of *capability* objects:

- a driver may map a register block only if it holds an MMIO *resource* for that physical range;
- it may receive an interrupt only if it holds an IRQ resource;
- it may program a DMA engine only through a *Bus Transaction Initiator* (BTI), which the kernel binds to the platform IOMMU.

Each of these is an ordinary kernel object reached through a handle. So the rights model of Section XIX applies unchanged, and authority over hardware is as easy to audit as authority over a channel.

A. The Resource Object and the Resource Tree

A **Resource** is a capability that grants access to a restricted kernel or hardware facility. A *kind* and an address *range* identify it. At boot the kernel creates a single **root resource**. It hands



child resources narrow the parent range; kind is fixed

Fig. 28. The resource tree. The single root resource is divided into MMIO, IRQ, IOPORT and SYSTEM children. Each child is narrowed further and finally given to the one driver that needs it.

the root resource, over the bootstrap channel of Section XVIII, to `userboot`, which passes it on to `component_manager`.

The root resource is the top of a *resource tree*. From a parent resource, `zx_resource_create` makes a child that covers part of the range, for a single kind. The holder then gives the child to the one driver that needs it (Fig. 28). A child can only narrow its parent's range, and can never change its kind. So the tree is a strict delegation hierarchy. A driver that holds the MMIO window for one controller cannot reach the registers of any other device [2].

The kinds are:

- `ZX_RSRC_KIND_MMIO`: physical register ranges;
- `ZX_RSRC_KIND_IRQ`: interrupt vectors;
- `ZX_RSRC_KIND_IOPORT`: x86 I/O ports;
- `ZX_RSRC_KIND_SMC`: ARM secure monitor calls;
- `ZX_RSRC_KIND_SYSTEM`: gates on single features;
- `ZX_RSRC_KIND_ROOT`: the unrestricted top, whose base and size are both zero.

The **SYSTEM** kind is different from the others. It has no address range. Instead it holds one of a fixed set of *base* values, each of which gates one privileged facility. Examples are `ZX_RSRC_SYSTEM_HYPERVISOR_BASE`, `ZX_RSRC_SYSTEM_VMEX_BASE` (the gate on making memory executable, Section XIX), `ZX_RSRC_SYSTEM_DEBUG_BASE`, `ZX_RSRC_SYSTEM_IOMMU_BASE`, `ZX_RSRC_SYSTEM_MSI_BASE`, `ZX_RSRC_SYSTEM_POWER_BASE`, `ZX_RSRC_SYSTEM_CPU_BASE` and `ZX_RSRC_SYSTEM_MEXEC_BASE`. The call is

```
zx_resource_create(parent, options, base,
                  size, name, name_size, &child)
```

and `options` names the kind. For an MMIO or IRQ child, `base` and `size` mark the granted part of the range. For a **SYSTEM** child, `base` is one of the `*_BASE` values and `size` is 1.

B. User-Space MMIO

With an MMIO resource, a driver writes device registers directly from user space, in three steps:

- 1) It creates a *physical* VMO over the register window with `zx_vmo_create_physical(mmio_rsrc, paddr, size, &vmo)`.
- 2) It marks the VMO uncached with `zx_vmo_set_cache_policy(vmo, ZX_CACHE_POLICY_UNCACHED_DEVICE)`, so that the CPU

does not buffer or reorder stores to the registers. On AArch64 this selects the Device-nGnRE memory attributes. 3) It maps the VMO with `zx_vmar_map`.

The mapped range *is* the register block. An ordinary load or store now reads or writes a hardware register. The driver must set the cache policy before it maps the VMO, while the VMO has no mappings, children or pinned pages. The Magma system driver (MSD) of Volume I maps a GPU’s register file exactly this way. After the mapping, the driver programs the GPU with plain memory accesses.

Listing 16. User-space MMIO mapping from an MMIO resource.

```
zx_handle_t vmo;
zx_vmo_create_physical(mmio_rsrc, regs_paddr, regs_size, &vmo);
zx_vmo_set_cache_policy(vmo, ZX_CACHE_POLICY_UNCACHED_DEVICE);

zx_vaddr_t base;
zx_vmar_map(vmar, ZX_VM_PERM_READ | ZX_VM_PERM_WRITE,
            0, vmo, 0, regs_size, &base);
// base now points at the device register window:
volatile uint32_t* ctrl = (uint32_t*)(base + CTRL_OFFSET);
*ctrl = CTRL_ENABLE; // a store to a hardware register
```

Interrupts follow the same capability rule through another kind. An IRQ resource gates `zx_interrupt_create`. The driver then waits on the new interrupt object, or binds it to a port, as Section XV describes. The resource is the right to *name* a vector. The interrupt object is how the interrupt is delivered.

C. DMA and the BTI: a Per-Device Memory Firewall

Programmed I/O is enough for control registers, but bulk data moves by DMA. A device that does DMA is a bus master. Unchecked, it can read or write *any* physical address. Zircon’s answer is the BTI. A **Bus Transaction Initiator** models the view of memory that exactly one device has through the IOMMU.

A driver creates it from an IOMMU object with `zx_bti_create(iommu, options, bti_id, &bti)`. `bti_id` is the hardware identifier that the platform uses to recognize the device’s bus transactions. On Intel VT-d it is a PCI bus, device and function; on an ARM SMMU it is a StreamID.

Before a device may DMA into a buffer, the driver *pins* the buffer:

```
zx_bti_pin(bti, perms, vmo, offset, size,
          addr, num_addrs, &pmt)
```

This call does three things at once.

- It locks the VMO’s pages in RAM, so the PMM cannot move, reclaim or page them out during a transfer.
- It programs the IOMMU so that this device, and only this device, can reach those pages.
- It returns in `addr`s the **bus (device-physical) addresses** that the driver must program into the device’s DMA engine. It also returns a **PMT** (Pinned Memory Token) that stands for the pinning.

The flags `ZX_BTI_PERM_READ`, `ZX_BTI_PERM_WRITE` and `ZX_BTI_PERM_EXECUTE` limit what the device may do. By default `addr`s gets one entry per page. With `ZX_BTI_COMPRESS` it gets one entry per minimum-contiguity block, whose size `ZX_INFO_BTI` reports. With `ZX_BTI_CONTIGUOUS` it gets a single start address, for a VMO that is already contiguous. When the transfer is done, `zx_pmt_unpin(pmt)` removes the IOMMU mapping and releases the pages. The call always consumes the token.

The result is a per-device memory firewall (Fig. 29). The IOMMU checks every bus transaction from the device against

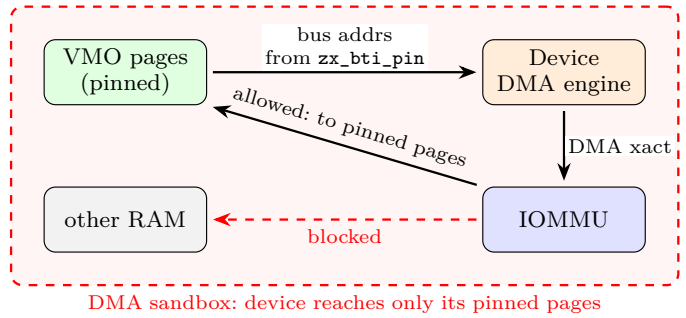


Fig. 29. Device DMA through a BTI. Pinning VMO pages gives the device its bus addresses. The IOMMU checks every transaction against those pages and faults all others, so the device can reach only its own buffers.

the pages pinned for its BTI. A transaction to any other address faults, instead of writing over unrelated RAM. A buggy or compromised device can therefore touch only the buffers its driver gave it. This is the DMA counterpart of the address-space isolation that protects one process from another.

D. The IOMMU Object and Portability

An object models the IOMMU itself. A program creates it from a `SYSTEM` resource with `zx_iommu_create(system_rsrc, type, desc, desc_size, &iommu)`, where the resource holds `ZX_RSRC_SYSTEM_IOMMU_BASE`. The `type` selects the backend. `ZX_IOMMU_TYPE_INTEL` and `ZX_IOMMU_TYPE_ARM_SMMU` are for real hardware. `ZX_IOMMU_TYPE_STUB`, once called the “dummy” IOMMU, is for platforms that have none.

The stub IOMMU gives no hardware protection and cannot fault a stray transaction. It still *pins* pages, and it returns their physical addresses as the bus addresses. So driver code written against the BTI calls is portable. The same `zx_bti_pin` and `zx_pmt_unpin` sequence works with or without translation hardware, and gains protection where the hardware exists.

Two more safety properties complete the design. First, a PMT may be destroyed *without* a proper `zx_pmt_unpin`, for example when a driver crashes during a transfer. Its pages are then not freed. They are *quarantined*: set aside so that nothing can reuse them while a device might still be writing to them. This prevents a DMA write into freed memory. Once the device is known to be quiet, the driver releases quarantined pages with `zx_bti_release_quarantine`.

Second, a device without scatter-gather needs physically contiguous buffers. `zx_vmo_create_contiguous(bti, size, align_log2, &vmo)` allocates one against a BTI. Pinned with `ZX_BTI_CONTIGUOUS`, it gives a single bus address for the whole transfer.

E. How a User-Space Driver Touches Hardware

A real driver uses all of the primitives above together: resources, physical VMOs, interrupt objects, and the BTI and IOMMU. Following that path from end to end shows why a user-space driver is both *possible* and *safe*. Take a PCI device that does DMA, such as an Ethernet NIC. The GPU register block that Volume I’s Magma system driver drives works the same way (Fig. 30).

a) *Capability acquisition*: The Driver Framework (DFv2) binds the driver to a device node and starts it in a `driver_host` process. It then *routes* to the driver exactly the handles it needs, as FIDL capabilities over the

TABLE XVII
HARDWARE NEED → KERNEL MECHANISM → SYSCALLS.

Need	Mechanism	Syscalls
Access gating	resource capability	<code>zx_resource_create</code>
Registers	physical VMO + map	<code>zx_vmo_create_physical</code> , <code>zx_vmar_map</code>
Interrupts	interrupt object	<code>zx_interrupt_create</code> / <code>wait / ack</code>
DMA	BTI + PMT	<code>zx_bti_pin</code> , <code>zx_pmt_unpin</code>

`fuchsia.hardware.platform.device` or PCI protocols. The driver gets no ambient access. It gets an MMIO resource (or a ready-made MMIO VMO for the register window), one or more interrupt objects, and a BTI handle for its device. It receives these and nothing else. It has no way to name another device's registers or to reach arbitrary RAM. This is the capability principle of Section XIX, applied to silicon.

b) Register access (MMIO): The driver turns its MMIO authority into a physical VMO with `zx_vmo_create_physical`, or receives the VMO directly. It marks the VMO `ZX_CACHE_POLICY_UNCACHED_DEVICE` and maps it with `zx_vmar_map`. A plain pointer access into the mapped range is now a register read or write (Listing 16). The kernel guarantees that the mapping covers *only* the granted physical range.

c) Interrupts: The driver waits on its interrupt object with `zx_interrupt_wait`, or binds it to a port. When the line is asserted, the kernel's minimal handler masks and acknowledges the controller and wakes the driver thread. The thread reads the device status through its MMIO mapping. It then calls `zx_interrupt_ack` to re-arm the level-triggered source.

d) DMA, the key memory step: For bulk transfers the driver allocates a buffer as a VMO. For an engine without scatter-gather it uses a contiguous VMO from `zx_vmo_create_contiguous`; otherwise an ordinary VMO will do. It then calls `zx_bti_pin`. The pin keeps the PMM from moving or reclaiming the pages during the transfer. It returns the *bus addresses* to write into the device's DMA descriptors, and a PMT. The kernel programs the IOMMU so that the device can reach *only* those pinned pages.

This step is what makes user-space DMA safe. In a monolithic design, a user-space driver that controlled a DMA engine could point it at any physical RAM and bypass all memory protection. Here a buggy or compromised device is fenced in hardware to its own buffers. When the transfer completes, the driver calls `zx_pmt_unpin`.

e) Zero-copy sharing: The buffer is a VMO, so the driver can pass it by handle to another process. A received packet's VMO goes to the network stack, and a rendered frame's VMO goes to the display compositor, with no copy in between. With `sysmem` (the buffer allocator of Volume I) coordinating the buffers, data flows from device to driver to consumer by handle passing alone.

Table XVIII lists the hardware-access objects and their syscalls.

XVIII. BOOT AND USER-SPACE BOOTSTRAP

A capability kernel grants no ambient authority. So at power-on the first question is where *any* authority comes from. Zircon's answer is a short, explicit chain. The kernel creates the root job and the root resource. It hands them to exactly one user

TABLE XVIII
RESOURCE KINDS AND HARDWARE-ACCESS SYSTEM CALLS.

Kind / object	Authorizes	Key syscall
<code>ZX_RSRC_KIND_MMIO</code>	register windows	<code>zx_vmo_create_physical</code>
<code>ZX_RSRC_KIND_IRQ</code>	interrupt vectors	<code>zx_interrupt_create</code>
<code>ZX_RSRC_KIND_IOPORT</code>	x86 I/O ports	(port in/out)
<code>ZX_RSRC_KIND_SMC</code>	ARM SMC calls	<code>zx_smc_call</code>
<code>ZX_RSRC_KIND_SYSTEM</code>	feature gates	<code>zx_iommu_create</code>
<code>ZX_RSRC_KIND_ROOT</code>	everything	<code>zx_resource_create</code>
Resource	carve a child range	<code>zx_resource_create</code>
IOMMU	model DMA translation	<code>zx_iommu_create</code>
BTI	one device's DMA view	<code>zx_bti_create</code>
(pin)	lock pages, get bus addr	<code>zx_bti_pin</code>
PMT	a specific pinning	<code>zx_pmt_unpin</code>

program over a single channel. That program hands narrower capabilities onward.

There is no global registry and no implicit access. Every later process holds exactly the handles its parent chose to give it. This section follows that chain from the bootloader to the first component [19]. Fuchsia Subsystems Volume IV (Platform Bring-up) continues the story in user space, and Subsystems Volume XIV covers the bootloader and the build of the boot image.

A. The Zircon Boot Image

The bootloader does not load a kernel and a separate ramdisk. It loads one self-describing container, the **ZBI** (Zircon Boot Image). A ZBI starts with a container header. A sequence of typed items follows, each with its own header that gives its `type`, `length`, `flags` and a CRC. The kernel image is the first item: `ZBI_TYPE_KERNEL_X64`, `ZBI_TYPE_KERNEL_ARM64` or `ZBI_TYPE_KERNEL_RISCV64`. The other items hold everything early boot needs:

- `ZBI_TYPE_CMDLINE`: the kernel command line;
- `ZBI_TYPE_STORAGE_BOOTFS`: the compressed `bootfs` ramdisk that holds the first user-space programs;
- `ZBI_TYPE_MEM_CONFIG`: the map of usable RAM;
- `ZBI_TYPE_PLATFORM_ID` and `ZBI_TYPE_CPU_TOPOLOGY`: the board and its core layout;
- `ZBI_TYPE_ACPI_RSDP` or `ZBI_TYPE_DEVICETREE`: the firmware tables;
- `ZBI_TYPE_KERNEL_DRIVER`: the serial console and other early peripherals.

The kernel item's payload starts with a `zbi_kernel_t` header. The header gives the `entry` point, and the `reserve_memory_size`: the scratch RAM the kernel needs right after its load image. In effect, the ZBI is the kernel's argument list.

B. Life of a Kernel Boot: Power-On to Scheduler

All of the above happens before any line of the kernel proper runs. This subsection follows the full sequence from reset to a running scheduler (Fig. 31). The hand-off to user space, which the rest of this section covers, is only its last step.

a) Power-on and reset: The boot CPU leaves reset at the architecture's reset vector in ROM. Firmware then initializes DRAM, the clocks and the boot core, and loads the boot image into RAM. On x86 the firmware is UEFI or a BIOS. On ARM

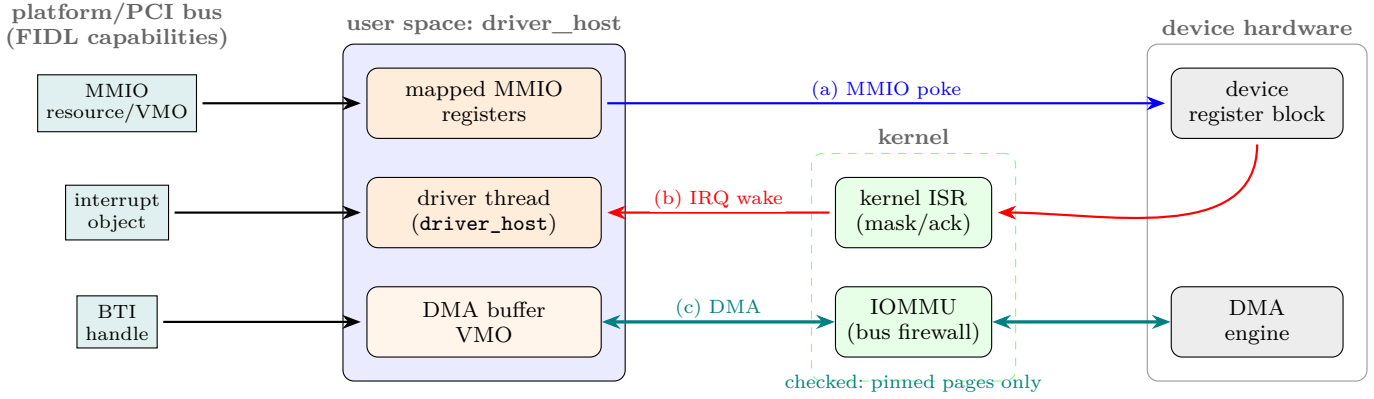


Fig. 30. How a user-space driver touches hardware. The Driver Framework routes three capabilities into the **driver_host**: an MMIO resource or VMO, an interrupt object, and a BTI. (a) The driver writes registers through the mapped MMIO window. (b) The kernel’s minimal handler wakes it on an IRQ, and it re-arms the line with `zx_interrupt_ack`. (c) It does DMA to a BTI-pinned VMO, and the IOMMU checks every bus transaction against the pinned pages. That check is what makes user-space DMA safe.

it is a boot ROM plus a bootloader such as coreboot, U-Boot or Gigaboot.

b) *Bootloader to ZBI*: The bootloader jumps to the Zircon image and passes the physical address of the ZBI. The calling convention depends on the architecture. On ARM64 and RISC-V the kernel starts with a pointer to the ZBI in a register. On x86 a small multiboot or EFI shim adapts the firmware’s hand-off. The kernel item’s `zbi_kernel_t` header tells the loader where to jump and how much memory to reserve.

c) *physboot: decompress, relocate, hand off*: The kernel item is a compressed ELF file. So a bootstrap stage that runs on physical addresses, **physboot**, runs first. It decompresses the real kernel and loads it as a position-independent ELF. It relocates the kernel to its final virtual address, which is where kernel ASLR happens. It reserves the memory the kernel asked for and builds the hand-off structures: the parsed boot options, and a `PhysHandoff` record that describes the physmap and the first mappings. Then it jumps to the kernel’s entry point.

d) *Early architecture init*: The kernel’s per-architecture entry puts the CPU in a known state before any portable code runs. On ARM64 it drops from EL2 or EL3 to **EL1**, and installs the exception vector base (`VBAR_EL1`). It programs `MAIR_EL1`, `TCR_EL1` and the `TTBR0` and `TTBR1` page-table roots. It turns on the MMU and the caches, sets the boot stack, and zeroes BSS. On x86 the matching steps are the switch to long mode, loading the GDT and IDT, and enabling paging; current builds do most of this in **physboot**. On every architecture, the kernel sets up the high-half *physmap*, a fixed mapping of all physical RAM, so that it can reach any page. Control then reaches the C entry point `lk_main()`.

e) *Kernel main and the init levels*: `lk_main` brings the system up through an ordered sequence of *init levels* (Table XIX). After the earliest hooks it runs `arch_early_init` and `platform_early_init`, which parse the ZBI and start the early UART. Then come the VM bootstrap before the heap, the kernel heap, the full VM and VMAR system, the interrupt controller, the CPU topology and the core kernel objects. By the **THREADING** level the scheduler and threads work.

Around the VM level the kernel calibrates the platform timer to a known frequency, so that monotonic time is well defined. On x86 it measures the TSC against the HPET. On ARM it reads `CNTFRQ_ELO`. On RISC-V it takes the timebase frequency from the boot configuration.

TABLE XIX
KERNEL BRING-UP INIT LEVELS (`LK/INIT.H`), IN ORDER.

Init level	What becomes available
EARLIEST	earliest debug/serial, command line
ARCH_EARLY	basic CPU/arch state
PLATFORM_EARLY	ZBI parsed, memory map, early UART
VM_PREHEAP	bootstrap VM before the heap
HEAP	kernel heap usable
VM	full VM / VMAR system
INTC	interrupt controller (GIC/APIC/PLIC)
TOPOLOGY	CPU topology known
KERNEL	core kernel objects
THREADING	scheduler / threading live
PLATFORM	fuller platform init
SMP_READY	all CPUs online
USER	just before userboot

f) *SMP bring-up*: The boot CPU then starts the secondary cores. Each architecture has its own mechanism behind one interface:

- on ARM, `PSCI CPU_ON`;
- on x86, the `INIT`, `SIPI`, `SIPI` sequence;
- on RISC-V, `sbi_hart_start` from the SBI hart state management extension.

Each secondary core runs its own per-CPU init. It waits at the `SMP_WAIT` and `SMP_READY` barrier before it joins scheduling.

g) *Threading is up*: `lk_main` creates a bootstrap thread to finish the higher init levels. The boot CPU then calls `Thread::Current::BecomeIdle()`. It becomes that CPU’s idle thread and enables interrupts. At that point the scheduler is live and the system is “up”. Each secondary core also settles into its own idle thread.

h) *Hand to user space*: Finally, at the **USER** init level, the kernel creates the root job, the root resource (Section XVII), the vDSO VMO, and the first user process, **userboot**, with its bootstrap channel. The rest of this section starts from that point, where the running kernel meets the first user program.

C. userboot: the First Program

userboot is a small user-space loader that is built *into* the kernel image, laid out like the vDSO. The kernel maps both **userboot** and the vDSO into the first process, and jumps

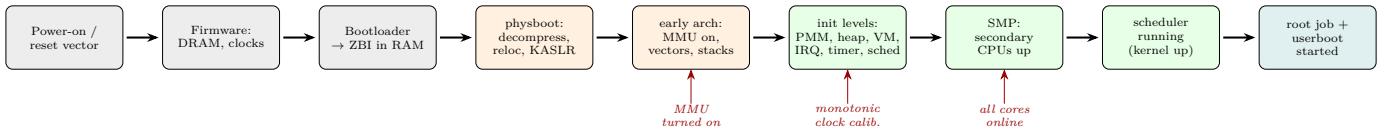


Fig. 31. Life of a kernel boot. Gray boxes are hardware and firmware, orange boxes are physboot and early architecture setup, green boxes are the kernel proper, and the teal box is the start of user space. The notes below mark where the MMU is turned on, where the monotonic clock is calibrated, and where SMP brings the other cores online.

to **userboot**'s entry point. The kernel never reads an ELF file from disk to start, because there is no disk yet. It gives **userboot** its starting state over a **bootstrap channel**, as a single **processargs** message. The message holds:

- a handle to the root job;
- the root resource, and the finer-grained MMIO, IRQ, IOPORT and SYSTEM resources;
- the process and thread handles for itself;
- the root VMAR, the initial stack VMO and the vDSO VMO;
- the ZBI itself, as a VMO tagged **PA_VMO_BOOTDATA**.

userboot finds the first **ZBI_TYPE_STORAGE_BOOTFS** item in that ZBI and decompresses it into a new VMO. This is **bootfs**, a simple read-only file system image that holds the programs and libraries of the early boot world. **userboot** then reads two options from the command line, which arrives as the message's environment strings:

- **userboot.next**: the program to launch. The default is **bin/component_manager+-boot**, where + separates the arguments.
- **userboot.root**: an optional path prefix inside bootfs.

userboot loads that program from bootfs. It serves as the program's dynamic-linker *loader service*, through the **PA_LDSVC_LOADER** handle. It sends the program a new **processargs** message with the handles it received, but with the new process's own process, thread and VMAR handles in place of its own. Then it gets out of the way. When the launched program exits, **userboot** can power the system off or reboot it (**userboot.shutdown**, **userboot.reboot**). In normal operation it simply waits.

D. The processargs Bootstrap Protocol

Every hand-off in Fig. 32 uses the **processargs** protocol. It is the convention by which a new process receives its first capabilities. The new process reads one message from its bootstrap channel. The message's *handle array* holds the handles themselves. Its byte payload is a **zx_proc_args_t** header followed by four tables (Fig. 33): a *handle-info* array, the argument strings, the environment strings, and a names table.

Each handle-info entry is a 32-bit word, built by the **PA_HND(type, arg)** macro. It tags the handle at the same index with a **PA_* type** in its low byte and a 16-bit *argument* in its high half. The type tells the new program what each handle *is*. The argument tells apart several handles of the same type, such as the *n*th file descriptor or the *n*th namespace directory.

The standard types cover the authority a program needs to start itself:

- **PA_PROC_SELF** and **PA_THREAD_SELF**: its own process and first thread;
- **PA_JOB_DEFAULT**: the job in which to create children;
- **PA_VMAR_ROOT**: its address space;
- **PA_VMO_STACK**: its first stack;

TABLE XX
SELECTED PA_* HANDLE TYPES AND ZBI ITEM TYPES.

Name	Meaning
<i>processargs handle types</i>	
PA_PROC_SELF	the new process's own handle
PA_THREAD_SELF	its first thread
PA_JOB_DEFAULT	job for spawning children
PA_VMAR_ROOT	root of its address space
PA_VMO_STACK	initial stack VMO
PA_VMO_VDSO	the vDSO to map for syscalls
PA_LDSVC_LOADER	dynamic-linker loader service
PA_NS_DIR	a namespace directory (indexed)
PA_DIRECTORY_REQUEST	where to publish outgoing services
PA_VMO_BOOTDATA	the ZBI image (privileged)
PA_SYSTEM_RESOURCE	the SYSTEM resource (privileged)
<i>ZBI item types</i>	
ZBI_TYPE_KERNEL_*	kernel image, first item (X64/ARM64/RISCV64)
ZBI_TYPE_CMDLINE	kernel command line
ZBI_TYPE_STORAGE_BOOTFS	compressed bootfs ramdisk
ZBI_TYPE_MEM_CONFIG	usable-RAM map
ZBI_TYPE_PLATFORM_ID	board identity
ZBI_TYPE_CPU_TOPOLOGY	core/cluster layout
ZBI_TYPE_ACPI_RSDP	ACPI root pointer (x86)
ZBI_TYPE_DEVICETREE	flattened device tree (ARM/RV)
ZBI_TYPE_KERNEL_DRIVER	early peripheral (e.g. UART)

- **PA_VMO_VDSO**: the vDSO to map for syscalls;
- **PA_LDSVC_LOADER**: the loader service that supplies shared libraries;
- **PA_NS_DIR**: namespace directories, named by the names table;
- **PA_FD**: file descriptors;
- **PA_DIRECTORY_REQUEST**: the server end on which the program publishes its own outgoing services.

Privileged programs also receive **PA_VMO_BOOTDATA** (the ZBI). They also receive resource handles such as **PA_MMIO_RESOURCE**, **PA_IRQ_RESOURCE**, **PA_IOPORT_RESOURCE** and **PA_SYSTEM_RESOURCE**. These kind-specific handles replace the single **PA_RESOURCE** root handle. This one table *is* the hand-off of authority from kernel to user space. A process can do exactly what its handles permit, and nothing more.

Table XX lists the most important **PA_*** handle types and ZBI item types.

XIX. THE SECURITY MODEL: POLICY, SANDBOXING, AND RESTRICTED MODE

Zircon's security model is not a layer added on top of the kernel. It follows directly from the core of objects, handles and rights. There is no ambient authority. A thread cannot name a file, a device or another process except through a handle it was given. A handle allows only the operations its rights bits permit (Section V). This section shows how Zircon builds layers of defence on that principle:

- job policy, which limits what a whole subtree of processes may even attempt;

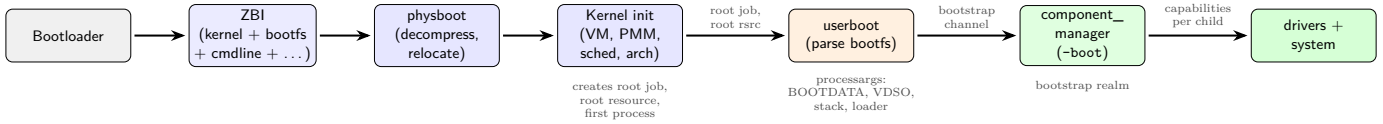


Fig. 32. The boot pipeline. The kernel creates authority once and hands it down a chain of single bootstrap channels: kernel → **userboot** → **component_manager** → the rest of the system. The label on each arrow names the handles passed in that hand-off.

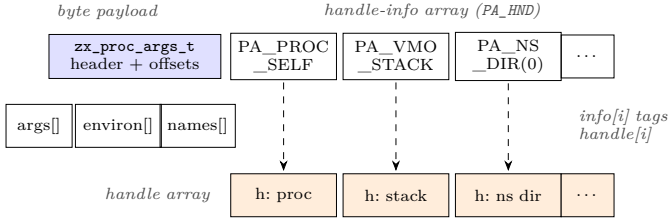


Fig. 33. The **processargs** bootstrap message. Each handle-info word tags the handle at the same index in the message’s handle array with a **PA_*** type. The arguments, the environment (the kernel command line) and the names follow in the byte payload.

TABLE XXI
CPU EXECUTION MODES UNDER ZIRCON.

Mode	Priv.	Kernel entry	Typical occupant
Kernel	EL1/ring0/S	(is the kernel)	scheduler, VM, dispatchers
Normal user	EL0/ring3/U	vDSO syscall	Fuchsia components
Restricted	EL0/ring3/U	bounce to host	Linux code under Starnix
Guest	EL0+stage2	VM-exit packet	a guest OS (VCPU)

- **W[^]X** enforcement, which puts executable memory behind a capability;
- *restricted mode*, which lets Zircon run a foreign operating system’s user programs inside one process without trusting them [1].

A. CPU Execution Modes

The security model is largely about the transitions between the CPU’s execution contexts, so it helps to list those contexts first (Table XXI). There are four.

- *Kernel* (supervisor) mode is the kernel itself: EL1 on ARM64, ring 0 on x86, S-mode on RISC-V. The scheduler, the VM system and the object dispatchers run here with full privilege.
- *Normal user* mode is where ordinary Fuchsia components run: EL0, ring 3 or U-mode. They reach the kernel only through the vDSO syscall gate.
- *Restricted* mode is a constrained user context, still EL0 or ring 3 in hardware, with a *separate* register set and a restricted view of the address space. Untrusted or foreign-ABI code runs here. On any syscall or fault, control returns to its normal-mode host instead of entering the kernel directly.
- *Guest* mode is code that runs under the hypervisor on a virtual CPU, with stage-2 (EPT) address translation. It is the strongest isolation, and Section XXI describes it.

From normal to restricted to guest mode, the kernel trusts the code less at each step, and each step adds a barrier that the hardware checks.

B. Job Policy as Sandboxing

Capabilities answer the question “may this handle do that?”. Job policy answers an earlier question: “may this process even create such a thing?”. A job holds a policy that its descendants inherit. A program installs it with

```
zx_job_set_policy(job, options, topic,
                  policy, policy_size)
```

Under the **ZX_JOB_POL_BASIC** topic, **policy** is an array of (**condition**, **action**) pairs. A *condition* names an operation that a process might attempt:

- creating an object: **ZX_POL_NEW_VMO**, **ZX_POL_NEW_CHANNEL**, **ZX_POL_NEW_PROCESS**, **ZX_POL_NEW_EVENT**, **ZX_POL_NEW_PORT**, **ZX_POL_NEW_SOCKET**, **ZX_POL_NEW_FIFO**, **ZX_POL_NEW_TIMER**, **ZX_POL_NEW_PAGER**, and the catch-all **ZX_POL_NEW_ANY**;
- misusing a handle: **ZX_POL_BAD_HANDLE** (using an invalid handle value) and **ZX_POL_WRONG_OBJECT** (an operation on the wrong object type);
- memory safety: **ZX_POL_VMAR_WX** (mapping a region writable and executable at once, which breaks **W[^]X**) and **ZX_POL_AMBIENT_MARK_VMO_EXEC** (marking a VMO executable without presenting a capability).

The *action* for each condition is one of **ZX_POL_ACTION_ALLOW**, **ZX_POL_ACTION_DENY** and **ZX_POL_ACTION_KILL**. Two more actions, **ZX_POL_ACTION_ALLOW_EXCEPTION** and **ZX_POL_ACTION_DENY_EXCEPTION**, also raise an exception that a debugger can see, so that a policy hit can be caught and diagnosed.

Policy is *inherited*: it flows from a job to all its descendants. With **ZX_JOB_POL_RELATIVE**, a child’s new rules apply only where no parent has already set a rule. With **ZX_JOB_POL_ABSOLUTE**, every listed condition must take effect, or the call fails. Whether a child job may change an inherited rule is decided by the job that set it, one rule at a time. **ZX_POL_OVERRIDE_ALLOW** lets child jobs change the rule, and **ZX_POL_OVERRIDE_DENY** forbids it. Once a job has any child job or process, its policy cannot change at all. An SoC integrator wants this from a sandbox: the enclosing domain decides which of its rules an inner block may change, and locks the rest.

C. Executable Memory Is a Capability

A classic exploit turns a data buffer into code. Zircon blocks this by making execution a right, not a default. Mapping a VMO with **ZX_VM_PERM_EXECUTE** needs a VMO handle with **ZX_RIGHT_EXECUTE**. A process cannot simply add that right. It gets it only through **zx_vmo_replace_as_executable(vmo, vmex, &out)**. Here **vmex** is a **SYSTEM** resource that holds **ZX_RSRC_SYSTEM_VMEX_BASE**, the privileged “make memory executable” gate. A process that wants to JIT-compile code must therefore be *given* a **vmex** capability. A process without one cannot create new executable pages at all.

The job-policy condition **ZX_POL_AMBIENT_MARK_VMO_EXEC** controls the exception to this rule. Where its action is **ALLOW**,

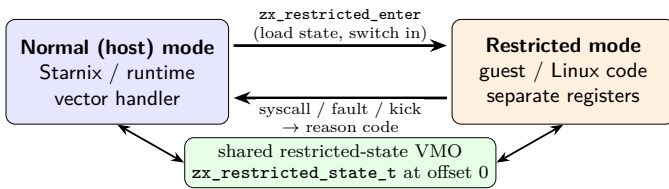


Fig. 34. The restricted-mode round trip. `zx_restricted_enter` switches the CPU into restricted mode to run guest code. A guest syscall, a fault or a kick returns control to the host’s vector handler with a reason code. Both modes share the restricted-state VMO, which holds the guest’s registers.

a process may pass `ZX_HANDLE_INVALID` as `vmex`; this ambient permission is for runtimes that really need it. Where its action is `DENY`, a real `vmex` resource is required.

So `WX` is enforced in two places. No page can be writable and executable at the same time (`ZX_POL_VMAR_WX`), and making memory executable at all needs a capability.

D. Restricted Mode

Short of full virtualization, the strongest tool is **restricted mode**. It runs untrusted or foreign-ABI code *inside* a normal process, with a separate register set and a restricted view of memory. On every syscall or fault, control returns to trusted “host” code. A host thread first calls `zx_restricted_bind_state` to get a shared *restricted-state VMO*. The guest’s registers live at offset 0 of that VMO, as a `zx_restricted_state_t`: the general registers, instruction pointer, stack pointer, flags and thread pointer. The host then calls

```
zx_restricted_enter(options, vector_table_ptr,
                  context)
```

This call loads the saved registers and switches the CPU into restricted mode, running the guest code. Restricted code runs natively at full speed until it does something the host must handle: it executes a syscall instruction, or it faults. The kernel then writes the guest’s registers back into the shared VMO. It returns control to *normal* mode at `vector_table_ptr`, with the host’s `context` and a *reason code* (Fig. 34):

- `ZX_RESTRICTED_REASON_SYSCALL` for a guest syscall;
- `ZX_RESTRICTED_REASON_EXCEPTION` for a fault;
- `ZX_RESTRICTED_REASON_KICK` when another thread called `zx_restricted_kick` to preempt it.

The host handles the event by reading and rewriting the shared registers, and then enters restricted mode again.

Starnix uses this mechanism to run unmodified Linux programs on Fuchsia (Volume IV). The Starnix kernel creates a Zircon process whose address space is split in two. The lower half is a private, restricted address space that holds the Linux program. The upper half holds the Starnix kernel itself, and is shared across all such processes through the `ZX_PROCESS_SHARED` facility. A Linux thread runs in restricted mode in the lower half, and *cannot even address* the Starnix half.

The Linux program makes a syscall with its architecture’s instruction: `syscall` on x86, `svc` on ARM. The kernel then returns to Starnix with `ZX_RESTRICTED_REASON_SYSCALL`. Starnix reads the Linux syscall number and arguments from the shared registers, and performs that Linux syscall with Zircon objects. A Linux `read` becomes operations on a Zircon channel or VMO, and a Linux `mmap` becomes a VMAR mapping. Starnix writes the result back into the registers and re-enters restricted mode.

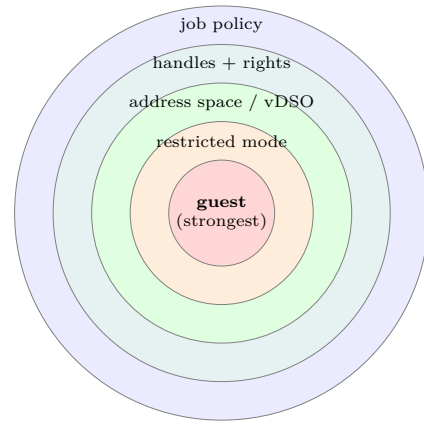


Fig. 35. The layered sandbox. Each ring is a separate barrier, and the kernel checks authority at every one. The innermost ring, a guest confined by hardware, is the strongest.

So the Linux ABI runs as a user-space personality, with no kernel changes and no loss of isolation. Each Linux process is contained exactly like any other Zircon process.

E. Defense in Depth

These mechanisms form nested barriers (Fig. 35).

- The outermost is job policy. A subtree that may not even create a channel has little room to misbehave.
- Inside that, handles cannot be forged, and rights can only be narrowed, which bounds what each capability can do.
- Inside that, each process has its own address space (VMARs and the MMU), so one process cannot read another’s memory. The vDSO check on the syscall return address makes sure syscalls enter only through the approved gate.
- Restricted mode adds a barrier inside one process, for foreign code.
- The strongest barrier is the hypervisor’s stage-2 translation (Section XXI), which confines a whole guest OS to a guest-physical sandbox.

An attacker must defeat every enclosing ring. Each ring is small enough to reason about on its own.

XX. DIAGNOSTICS, TRACING, AND DEBUGGING

Zircon moves drivers, file systems and services into user space. An engineer still needs to see inside the running system. You may need to read the log before any logger exists, or profile the scheduler at nanosecond resolution. You may need to list every mapping in a stuck process, or attach a source-level debugger to a faulting thread.

Zircon exposes each of these as ordinary operations on objects. So the same handle and rights rules that govern the rest of the kernel also govern inspection. A tool can observe only what it holds a capability for [3]. Volume XIII (Diagnostics & Observability) covers the user-space tools built on these calls.

A. The Kernel Debug Log

The lowest-level diagnostic is the **debuglog**, also called the **klog**. It is a ring buffer of timestamped records inside the kernel, copied to the serial console. It works from the first moment of boot, long before the user-space logging service starts. A program opens a handle to it with `zx_debuglog_create(rsrc, options, &log)`. The resource must be a `SYSTEM` resource

TABLE XXII
JOB-POLICY CONDITIONS AND ACTIONS (ZX_JOB_POL_BASIC).

Condition	Triggered when the process...
ZX_POL_NEW_ANY	creates any new object class
ZX_POL_NEW_VMO	creates a VMO
ZX_POL_NEW_CHANNEL	creates a channel
ZX_POL_NEW_PROCESS	creates a process
ZX_POL_NEW_PAGER	creates a pager
ZX_POL_BAD_HANDLE	uses an invalid handle value
ZX_POL_WRONG_OBJECT	acts on the wrong object type
ZX_POL_VMAR_WX	maps writable + executable (W~X)
ZX_POL_AMBIENT_MARK_VMO_EXEC	marks a VMO exec with no vmex
Action	Effect on a triggering operation
ZX_POL_ACTION_ALLOW	permit silently
ZX_POL_ACTION_DENY	fail with ZX_ERR_ACCESS_DENIED
ZX_POL_ACTION_ALLOW_EXCEPTION	permit, but raise an exception
ZX_POL_ACTION_DENY_EXCEPTION	deny and raise an exception
ZX_POL_ACTION_KILL	terminate the process

that holds `ZX_RSRC_SYSTEM_DEBUGLOG_BASE`. Without options the handle can only write. Passing `ZX_LOG_FLAG_READABLE` makes it readable as well.

`zx_debuglog_write` appends a record, and `zx_debuglog_read` takes one out. The buffer survives the transitions of early boot. So it is where kernel panics and the first messages from user-space bring-up land. When nothing else works, the serial `klog` usually still does.

B. Kernel Tracing and the Unified Timeline

For performance work, the kernel emits structured **trace records**. They cover context switches and thread state changes, syscall entry and exit, interrupt delivery, page faults, and IPC events. `zx_ktrace_control(rsrc, action, options, ptr)` arms tracing. Its actions are `KTRACE_ACTION_START`, `KTRACE_ACTION_STOP` and `KTRACE_ACTION_REWIND`. The `START` action takes a *group* bitmask that selects which kinds of record to capture: `KTRACE_GRP_SCHEDULER`, `KTRACE_GRP_SYSCALL`, `KTRACE_GRP_IRQ`, `KTRACE_GRP_VM`, `KTRACE_GRP_IPC`, and others. `zx_ktrace_read` drains the buffer.

These syscalls need a `SYSTEM` resource that holds `ZX_RSRC_SYSTEM_TRACING_BASE`. They also return `ZX_ERR_NOT_SUPPORTED` unless the kernel command line sets `kernel.enable-debugging-syscalls=true`, because tracing exposes the timing of the whole machine.

The kernel's records are only half of the picture. A user-space **trace manager** coordinates tracing across the system. It registers *trace providers*, and hands each provider a buffer VMO to write into. One provider, `ktrace_provider`, supplies the kernel's records. The trace manager then *merges* the kernel records with the events of every user-space provider into one timeline (Fig. 36).

It writes the result in the Fuchsia Trace Format (`.fxt`), which Perfetto displays. On that timeline you can see a scheduler delay, a lock convoy or a latency outlier directly, next to both kernel and application events. This combined view is the main tool for scheduler and latency analysis on Fuchsia.

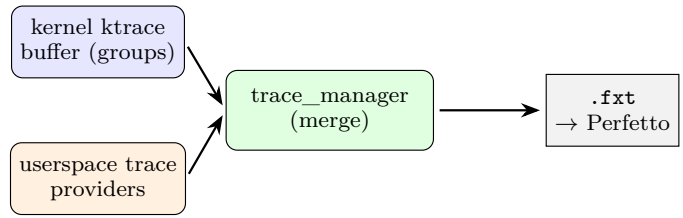


Fig. 36. Tracing data flow. The kernel's ktrace buffer and the userspace trace providers are merged by `trace_manager` into one `.fxt` timeline viewed in Perfetto.

C. Object Introspection

Most routine diagnosis uses `zx_object_get_info`, the one typed query that works on any object. Its *topic* selects the record:

- a process's address space and mappings: `ZX_INFO_PROCESS_MAPS`;
- a process's VMOs: `ZX_INFO_PROCESS_VMOs`;
- CPU and memory accounting per task: `ZX_INFO_TASK_STATS` and `ZX_INFO_TASK_RUNTIME`;
- system-wide memory: `ZX_INFO_KMEM_STATS` and `ZX_INFO_KMEM_STATS_EXTENDED`;
- the handle table: `ZX_INFO_HANDLE_TABLE`;
- load per CPU: `ZX_INFO_CPU_STATS`;
- the task tree: `ZX_INFO_JOB_CHILDREN` and `ZX_INFO_JOB_PROCESSES`.

`zx_object_get_property` and `zx_object_set_property` read and write attributes of an object that can change. Examples are `ZX_PROP_NAME`, the human label that makes a trace or a task list readable, and the debugger's hooks `ZX_PROP_PROCESS_DEBUG_ADDR` and `ZX_PROP_PROCESS_BREAK_ON_LOAD`. A debugger uses those two to find the list of loaded modules and to stop when a module loads. The kernel also has **kcounters**: named counters compiled into the kernel and shown by the `kcounter` tool. They are cheap, always on, and count events such as page faults, context switches and syscalls.

D. Debugging Faults

When a thread faults, Zircon's exception model (Section XV) delivers the fault to user space over an *exception channel*. A program gets one with `zx_task_create_exception_channel` on a thread, a process or a job. Two users of these channels matter here.

The first is **crashsvc**. It registers on the root job's exception channel and catches exceptions that no one else handled. It prints a symbolized backtrace, and can produce a minidump, before the process is killed or handed on. So even a crash that nobody watched leaves a usable report in the `klog`.

The second is **zxldb**, the Fuchsia source-level debugger. `zxldb` runs on the host and drives a `debug_agent` on the device. The agent binds exception channels and inspects threads with `zx_thread_read_state` and `zx_thread_write_state`. Those calls read and write register sets, such as `ZX_THREAD_STATE_GENERAL_REGS`, of a thread that is suspended or stopped in an exception. Single-stepping and breakpoints use the same syscalls. Fuchsia Subsystems Volume VII covers `zxldb` and the developer workflow.

For a system that is still booting or is stuck, there is also the kernel shell. Its `k` commands over serial, such as `k threads`, `k`

TABLE XXIII
SELECTED ZX_OBJECT_GET_INFO TOPICS AND DIAGNOSTIC SYSCALLS.

Topic / syscall	Yields
<i>get_info topics</i>	
ZX_INFO_PROCESS_MAPS	address-space mappings of a process
ZX_INFO_PROCESS_VMOs	VMOs a process holds/maps
ZX_INFO_TASK_STATS	memory accounting of a task
ZX_INFO_TASK_RUNTIME	CPU and queue time of a task
ZX_INFO_KMEM_STATS	system-wide memory usage
ZX_INFO_HANDLE_TABLE	all handles in a process
ZX_INFO_CPU_STATS	per-CPU load and IRQ counts
ZX_INFO_JOB_PROCESSES	processes directly under a job
<i>diagnostic syscalls</i>	
zx_debuglog_create	open the kernel debug log (klog)
zx_debuglog_write/read	append to / drain the klog
zx_ktrace_control	start/stop/rewind kernel tracing
zx_ktrace_read	drain kernel trace records
zx_object_get_property	read a property (e.g. name, debug addr)
zx_task_create_exception_channel	receive a task's exceptions
zx_thread_read_state	read a stopped thread's registers
zx_thread_write_state	write a stopped thread's registers

counters and `k oom info`, read kernel state directly when no user-space tool can run.

XXI. MULTI-ARCHITECTURE SUPPORT AND VIRTUALIZATION

Zircon runs on three 64-bit instruction set architectures from one portable kernel core:

- x86-64 (the `x64` target);
- ARM64 (`arm64`);
- RISC-V (`riscv64`), added by RFC-0211.

Everything that differs between them is confined to a thin, well-specified layer, and the bulk of the kernel does not know which one it runs on. An RTL design that is retargeted across process nodes is organized the same way. In Zircon the layer is the `arch/` tree, with one subdirectory per ISA. Above it sits a core that is written once: objects, scheduler, VM and IPC. The hypervisor, at the end of this section, uses the same hardware mechanisms to host whole guest operating systems [1], [21].

A. The Arch Abstraction Layer

The portable core depends on the architecture only through a fixed set of hooks, each implemented once per ISA (Fig. 37):

- the MMU and the page-table format;
- the context switch;
- the exception and interrupt vectors;
- access to per-CPU registers;
- atomics and memory barriers;
- cache maintenance;
- the platform timer and the interrupt controller.

Memory management is the hook with the largest effect. The VM subsystem changes address spaces only through the `ArchVmAspace` interface, which hides the page-table hardware.

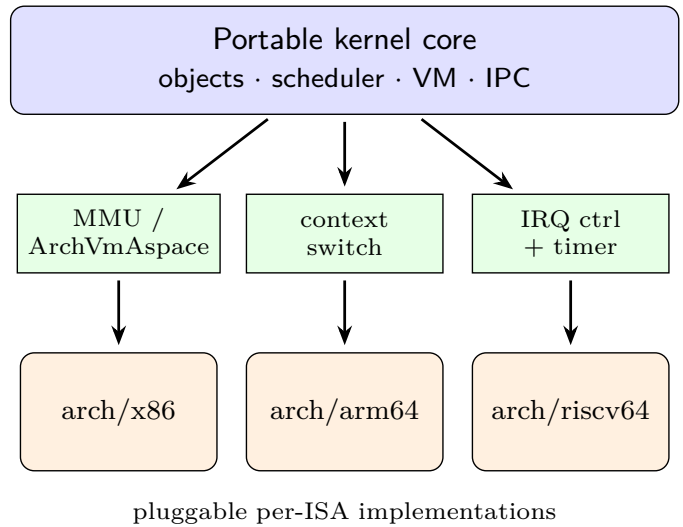


Fig. 37. The arch abstraction. The portable core reaches hardware only through fixed hooks (MMU, context switch, interrupt controller, timer). Each hook is implemented once per ISA, under `arch/x86`, `arch/arm64` and `arch/riscv64`.

- On x86-64, `ArchVmAspace` walks tables rooted at a PML4, with four-level paging, or five-level LA57 paging where the CPU has it. It tags TLB entries with **PCIDs**.
- On ARM64 it manages the split `TTBR0_EL1` and `TTBR1_EL1` roots (the user and kernel halves) in the AArch64 long-descriptor format, with **ASID**-tagged entries.
- On RISC-V it programs the `satp` CSR for **Sv39** or **Sv48** paging.

The portable VM code (Section IX) creates mappings and never sees any of this. A TLB shutdown, a context switch or the teardown of an address space calls down through the same interface on every target.

B. Interrupt Controllers and Timers

Two more facilities differ widely across ISAs, yet user space sees them the same way. Interrupts, whatever their hardware source, appear as Zircon interrupt objects (Section XV).

- On x86, the local **APIC** or **x2APIC** and the **IO-APIC** route the vectors.
- On ARM, the **GIC** (version 2 or 3) routes them.
- On RISC-V, the **PLIC** handles external device interrupts. The **CLINT** timer and IPI source belongs to M-mode, so the kernel, which runs in S-mode, reaches it through SBI firmware calls.

A driver that calls `zx_interrupt_wait` does not know which of these delivered its interrupt.

The scheduler tick and the monotonic clock (Section XVI) run on the platform timer, which is again specific to each ISA:

- on x86, the invariant **TSC**, with the HPET or the local APIC timer for one-shot deadlines;
- on ARM, the physical counter **CNTPCT** of the generic timer;
- on RISC-V, the supervisor `time` CSR.

The kernel calibrates each one at boot to a known frequency. So `zx_clock_get_monotonic` returns nanoseconds with the same meaning everywhere. Table XXIV collects these facts per architecture.

C. Virtualization: the Hypervisor and Guests

Zircon’s strongest isolation is not a software sandbox but virtualization that the hardware enforces. A program creates a *guest* with

```
zx_guest_create(rsrc, options,
               &guest, &guest_vmar)
```

from a `SYSTEM` resource that holds `ZX_RSRC_SYSTEM_HYPERVISOR_BASE`. The call returns *two* handles. One is the guest object. The other is a **guest-physical VMAR**: an address region, built like an ordinary process VMAR, that stands for the guest’s physical address space.

The host VMM fills it by mapping VMOs of guest RAM. Those mappings use *second-level* address translation: Intel **EPT** on x86, and ARM **stage-2** on arm64.

Every guest memory access is translated twice. The guest’s own page tables map guest-virtual to guest-physical addresses. Then the stage-2 tables, which the kernel controls, map guest-physical to host-physical addresses. So the guest can reach exactly the host RAM that its VMAR maps. This is an IOMMU-like firewall around a whole operating system.

`zx_vcpu_create(guest, options, entry, &vcpu)` creates a virtual CPU. `zx_vcpu_enter(vcpu, &packet)` runs it: guest code executes on the calling thread until a **VM exit** forces a return. The call returns the exit as a `zx_port_packet_t`, for example `ZX_PKT_TYPE_GUEST_MEM` for a trapped memory access or `ZX_PKT_TYPE_GUEST_IO` for port I/O. The host VMM decodes it, emulates the device the guest touched, and resumes the VCPU (Fig. 38). The VMM registers the ranges that should trap with `zx_guest_set_trap`. There are three kinds of trap:

- `ZX_GUEST_TRAP_MEM` for memory;
- `ZX_GUEST_TRAP_IO` for port I/O;
- `ZX_GUEST_TRAP_BELL`, an asynchronous doorbell, which delivers its packet to a *port* instead of through `zx_vcpu_enter`.

Another thread can force an exit at any time with `zx_vcpu_kick`; the running `zx_vcpu_enter` then returns `ZX_ERR_CANCELED`. On Fuchsia the user-space VMM is **Machina**. It builds the guest memory map, installs the traps, and emulates virtio devices in separate processes. Fuchsia Subsystems Volume V (Virtualization) describes it. Zircon supports hardware virtualization on Intel VMX and ARMv8 stage-2, but not on AMD SVM or the RISC-V H extension. The VMM is an ordinary user-space process, and the hardware confines the guest. This makes the guest boundary the firmest in the system, and the natural place to run an untrusted or legacy software stack on an FPGA-hosted SoC.

Restricted mode (Section XIX) is the other way to run foreign code, and it sits at the other end of the trade-off between isolation and cost.

- Restricted mode works at the *syscall level*. The guest shares the host process’s hardware context. Its syscalls return to a user-space host that emulates one ABI, and no second stage of translation is involved. It is cheap. But it trusts the host runtime to get the ABI right, and it works only for code that uses recognizable syscall instructions.
- Full virtualization works at the *instruction level*. The guest runs a complete, unmodified kernel under hardware trap-and-emulate, in its own stage-2 address space. Each trapped access pays for a heavier VM exit. In return, a whole operating system is isolated with no software in the trusted path.

Starnix uses restricted mode to run a Linux *userland* as a Fuchsia process. Machina uses the hypervisor to run a Linux *kernel* as

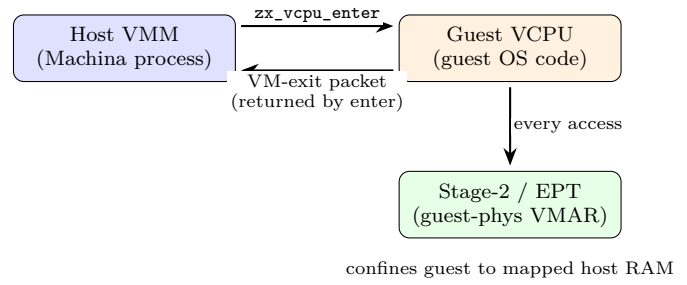


Fig. 38. The guest and VCPU model. The host VMM enters the VCPU with `zx_vcpu_enter`. The guest runs under stage-2 (EPT) translation. The call returns each VM exit to the VMM as a packet, and the VMM emulates the access.

TABLE XXIV
PER-ARCHITECTURE FACTS.

Facet	x86-64	ARM64	RISC-V
Target	x64	arm64	riscv64
Page tables	PML4 (4/5-lvl)	TTBR0/1 desc.	Sv39 / Sv48
TLB tag	PCID	ASID	ASID (satp)
IRQ ctrl	APIC/IO-APIC	GICv2/v3	PLIC (+CLINT)
Timer	TSC/HPET	CNTPTCT	time CSR
Hypervisor	VMX/EPT	stage-2	none

a guest. The two combine well. A hardened product can run foreign userland in restricted mode, and keep the hypervisor for code that must believe it owns a whole machine.

XXII. CONCLUSION

Zircon is a small, regular core of *mechanism*, and user space supplies almost all of the *policy*. The kernel exports a few dozen object classes through one uniform interface: handles that cannot be forged, each with fine-grained rights. It never grants authority that a process was not explicitly given. The rest of this volume follows from that one decision.

- Tasks (jobs, processes and threads) form a tree. The tree’s edges set both resource accounting and policy.
- The fair scheduler, with its deadline reservations, shares the CPUs among the threads.
- VMOs describe memory and VMARs place it. The page queues and a *user-space* pager decide which pages are resident.
- Signals, ports and futexes collect all asynchronous events, so that one thread can wait on thousands of sources.
- Interrupts, exceptions and time are themselves objects.
- Hardware is reachable only through resource capabilities, physical VMOs, and the BTI and IOMMU DMA firewall.
- `userboot` and the `processargs` protocol set up authority exactly once, at boot.
- The security model combines capabilities with job policy, W[^]X enforcement and restricted mode.
- Portability is confined to a thin `arch/` layer, and on top of it the hypervisor provides the strongest isolation of all.

For a hardware designer, this structure maps onto familiar RTL ideas.

- Kernel *objects* behave like IP blocks with a register interface. Each is self-contained and addressable on its own, and blocks are composed by wiring, not by shared global state.
- *Handles* are grant tokens that cannot be forged. They form a capability bus on which holding a token is the permission, and the rights are per-wire enable bits.

- *Channels* work like an AXI-Stream link. They move framed messages point to point, and they are the only transport that moves capabilities. Unlike AXI-Stream, they apply no backpressure (Section XII).
- *Ports* are an interrupt aggregator. Many edge-style and level-style sources feed one queue that one handler serves.
- The *BTI and IOMMU* pair is a bus firewall. It limits a bus master's address window, as an SoC's system MMU limits a peripheral that does DMA.
- The *scheduler* is an arbiter. It time-shares the compute fabric under a contract of bandwidth and deadlines.

A kernel built this way suits an operating system for an FPGA-hosted SoC, because its trust boundaries are explicit and local. A soft-core driver that writes a memory-mapped block does so through an MMIO resource and a mapped physical VMO. A BTI contains an accelerator that does DMA. Restricted mode or a guest confines an untrusted or foreign-ABI workload. Each boundary can be checked on its own, the way you verify a block in simulation before integration.

This document is Volume III of the fourteen-volume *Fuchsia Internals* series. Volume I, *Graphics & Display Pipeline*, shows the object model at work. Its Magma system driver (MSD) maps the GPU's register block through an MMIO resource and a physical VMO. It feeds the GPU through BTI-pinned command and data buffers, exactly as Section XVII describes. Volume II, *Build System & Toolchains*, explains how the build produces the kernel, the vDSO, `userboot` and the `bootfs` image, and assembles them into the ZBI of Section XVIII. Together the three volumes give a hardware engineer a complete path: from a Verilog block, through the kernel that mediates access to it, to the graphics stack and the build that ships it.

XXIII. APPENDIX A: SYSCALL REFERENCE

Table XXV is a representative index of the Zircon syscalls, grouped by subsystem, with a one-line purpose for each call. It is not complete. All calls return `zx_status_t` unless noted. The kernel checks every handle argument for the required rights before the call has any effect [3]. The vDSO is the only legal entry point for every one of these calls (Section IV).

TABLE XXV: Representative Zircon system calls, by subsystem.

Syscall	Purpose
<i>Handles and objects</i>	
<code>zx_handle_close</code>	Release a handle; drops one reference to the object.
<code>zx_handle_close_many</code>	Close an array of handles atomically.
<code>zx_handle_duplicate</code>	Make a new handle to the same object, optionally with reduced rights.
<code>zx_handle_replace</code>	Consume a handle, returning an equivalent one with reduced rights.
<code>zx_object_get_info</code>	Read a typed <code>ZX_INFO_*</code> record about an object.
<code>zx_object_get_property</code>	Read a <code>ZX_PROP_*</code> property (e.g. name).
<code>zx_object_set_property</code>	Write a settable property.
<code>zx_object_wait_one</code>	Block until one of a signal mask asserts on a single object.
<code>zx_object_wait_many</code>	Block on signals across several objects.
<code>zx_object_wait_async</code>	Subscribe a port to an object's signals (edge/level).
<code>zx_object_signal</code>	Assert/deassert user signals on an object.
<code>zx_object_signal_peer</code>	Signal the peer of a paired object (channel/eventpair).
<code>zx_object_get_child</code>	Look up a child task/object by KOID.
<i>Tasks: job, process, thread</i>	
<code>zx_job_create</code>	Create a child job under a parent job.
<code>zx_job_set_policy</code>	Install inherited sandboxing policy on a job.
<code>zx_job_set_critical</code>	Mark a process critical: its death kills the job.
<code>zx_process_create</code>	Create a process (and its root VMAR) in a job.
<code>zx_process_start</code>	Start a process's first thread at an entry point.
<code>zx_process_read_memory</code>	Read another process's address space (debug).
<code>zx_process_write_memory</code>	Write another process's address space (debug).
<code>zx_thread_create</code>	Create a thread in a process.
<code>zx_thread_start</code>	Start a thread at an entry with an initial stack/arg.
<code>zx_thread_read_state</code>	Read register/state sets of a suspended thread.
<code>zx_thread_write_state</code>	Write register/state sets of a suspended thread.
<code>zx_task_kill</code>	Terminate a task (and its descendants).
<code>zx_task_suspend</code>	Suspend a task, returning a resume token.
<code>zx_task_create_exception_channel</code>	Obtain an exception channel for a task.
<i>Virtual memory: VMO, VMAR, pager</i>	
<code>zx_vmo_create</code>	Create an anonymous, zero-filled VMO.
<code>zx_vmo_create_child</code>	Create a copy-on-write or slice child VMO.
<code>zx_vmo_create_physical</code>	Create a VMO over a physical (MMIO) range.
<code>zx_vmo_create_contiguous</code>	Create a physically contiguous VMO for DMA.
<code>zx_vmo_read / zx_vmo_write</code>	Copy bytes to/from a VMO.
<code>zx_vmo_get_size /</code> <code>zx_vmo_set_size</code>	Query/resize a resizable VMO.
<code>zx_vmo_op_range</code>	Commit, decommit, zero, or cache-op a range.
<code>zx_vmo_set_cache_policy</code>	Set cache policy (uncached/write-combining) on a physical VMO.
<code>zx_vmo_replace_as_executable</code>	Add <code>ZX_RIGHT_EXECUTE</code> using a vmex resource.
<code>zx_vmar_map</code>	Map a VMO range into an address region.
<code>zx_vmar_unmap</code>	Remove a mapping.
<code>zx_vmar_protect</code>	Change protections (R/W/X) of a mapping.
<code>zx_vmar_allocate</code>	Carve a sub-region of an address space.
<code>zx_pager_create</code>	Create a pager object.
<code>zx_pager_create_vmo</code>	Create a VMO whose pages are pager-backed.
<code>zx_pager_supply_pages</code>	Fulfill a page request with content.
<code>zx_pager_op_range</code>	Signal errors / writeback / dirty state on a pager VMO.
<i>IPC: channel, socket, FIFO, stream, eventpair</i>	
<code>zx_channel_create</code>	Create a connected pair of channel endpoints.
<code>zx_channel_write / write_etc</code>	Send a message (bytes + handles) on a channel.
<code>zx_channel_read / read_etc</code>	Receive a message from a channel.
<code>zx_channel_call</code>	Send a request and await its matched reply (txid).
<code>zx_socket_create</code>	Create a connected byte/datagram socket pair.
<code>zx_socket_read /</code> <code>zx_socket_write</code>	Stream bytes through a socket.
<code>zx_fifo_create</code>	Create a fixed-element-size FIFO pair.
<code>zx_fifo_read / zx_fifo_write</code>	Move elements through a FIFO.
<code>zx_stream_create</code>	Create a seekable stream over a VMO.
<code>zx_stream_readv / writev</code>	Scatter/gather I/O on a stream.
<code>zx_eventpair_create</code>	Create a pair of mutually-signaling events.
<code>zx_event_create</code>	Create a standalone signalable event.
<i>Signals and ports</i>	
<code>zx_port_create</code>	Create a port (an async packet queue).
<code>zx_port_wait</code>	Dequeue the next packet (blocking with deadline).
<code>zx_port_queue</code>	Enqueue a user packet.
<code>zx_port_cancel</code>	Cancel a pending async wait.
<code>zx_futex_wait</code>	Sleep on a futex if its value still matches.
<code>zx_futex_wake</code>	Wake up to N waiters on a futex.
<code>zx_futex_requeue</code>	Wake some waiters and requeue the rest.

(continued on next page)

Table XXV (continued from previous page)

Syscall	Purpose
<i>Time, timers, clock</i>	
<code>zx_clock_get_monotonic</code>	Read the monotonic clock (ns since boot, paused in suspend).
<code>zx_clock_get_boot</code>	Read the boot clock (includes suspend).
<code>zx_nanosleep</code>	Sleep until an absolute deadline.
<code>zx_deadline_after</code>	Compute an absolute deadline from a duration.
<code>zx_timer_create</code>	Create a one-shot timer object.
<code>zx_timer_set</code>	Arm a timer with a deadline and slack.
<code>zx_timer_cancel</code>	Disarm a timer.
<code>zx_clock_create</code>	Create a user-space synthetic clock.
<code>zx_clock_read / update / get_details</code>	Read/steer a synthetic clock.
<code>zx_ticks_get</code>	Read the raw hardware tick counter.
<i>Interrupts and exceptions</i>	
<code>zx_interrupt_create</code>	Create an interrupt object bound to a vector (needs IRQ resource).
<code>zx_interrupt_bind</code>	Bind an interrupt to a port for async delivery.
<code>zx_interrupt_wait</code>	Block until the interrupt fires.
<code>zx_interrupt_ack</code>	Acknowledge a level-triggered interrupt to re-arm.
<code>zx_interrupt_trigger</code>	Trigger a virtual interrupt (testing).
<code>zx_interrupt_destroy</code>	Tear down an interrupt object.
<code>zx_msi_allocate</code>	Allocate a block of MSI vectors (PCI).
<code>zx_msi_create</code>	Create an interrupt object for one MSI vector.
<code>zx_exception_get_thread</code>	Get the faulting thread from an exception.
<code>zx_exception_get_process</code>	Get the faulting process from an exception.
<i>Hardware: resource, BTI, PMT, IOMMU, MMIO</i>	
<code>zx_resource_create</code>	Carve a child resource for a sub-range/kind.
<code>zx_iommu_create</code>	Model a platform (or stub) IOMMU (SYSTEM resource).
<code>zx_bti_create</code>	Create a Bus Transaction Initiator from an IOMMU.
<code>zx_bti_pin</code>	Pin VMO pages; return device-physical addresses + a PMT.
<code>zx_bti_release_quarantine</code>	Release pages quarantined by an improper unpin.
<code>zx_pmt_unpin</code>	Unpin a pinned-memory token, freeing the pages.
<code>zx_smc_call</code>	Issue an ARM secure-monitor call (SMC resource).
<i>System, virtualization, restricted mode, diagnostics</i>	
<code>zx_system_get_num_cpus</code>	Number of logical CPUs.
<code>zx_system_get_physmem</code>	Total physical memory.
<code>zx_system_get_version_string</code>	Kernel/system version string.
<code>zx_system_powerctl</code>	Reboot, shutdown, or change CPU power state.
<code>zx_system_mexec</code>	Kexec-style hand-off to a new kernel image.
<code>zx_guest_create</code>	Create a guest (guest-physical address space).
<code>zx_vcpu_create</code>	Create a virtual CPU within a guest.
<code>zx_vcpu_enter</code>	Run the VCPU until a VM-exit packet is returned.
<code>zx_vcpu_kick</code>	Force a VCPU to exit <code>zx_vcpu_enter</code> .
<code>zx_restricted_bind_state</code>	Map the shared restricted-mode register state VMO.
<code>zx_restricted_enter</code>	Enter restricted mode running guest/foreign code.
<code>zx_debuglog_create</code>	Open a handle to the kernel debug log (klog).
<code>zx_debuglog_write / read</code>	Append to / drain the kernel debug log.
<code>zx_ktrace_control</code>	Start/stop/rewind kernel tracing.

XXIV. APPENDIX B: GLOSSARY

KOID Kernel Object Identifier. A 64-bit number that names a kernel object for the life of the boot. It is unique and never reused. It differs from a handle, which is a per-process *reference* to the object.

Handle A process-local reference to a kernel object that cannot be forged, with a rights mask. Holding a handle is the authority to act on the object. The kernel translates handle values to objects on every syscall.

Dispatcher The C++ base class inside the kernel behind every object class (`VmoDispatcher`, `ChannelDispatcher`, ...). It holds the object's state, signal bits and observer list. A handle points at a dispatcher.

Rights A bitmask on a handle (`ZX_RIGHT_READ`, `WRITE`, `DUPPLICATE`, `TRANSFER`, `EXECUTE`, ...) that sets which operations the handle allows. Duplication can narrow rights but never widen them.

VMO Virtual Memory Object. A resizable container of pages that the kernel manages; it is the unit of memory in Zircon. A VMO may be anonymous, copy-on-write, physical (MMIO), contiguous or pager-backed.

VMAR Virtual Memory Address Region. A recursive span of a process's address space into which VMOs are mapped. It is the kernel's record of where memory lives, and the unit of address-space randomization and isolation.

PMM Physical Memory Manager. The kernel allocator that hands out physical page frames. It keeps bookkeeping for every page, and the page queues.

Physmap A permanent kernel mapping of all, or most, physical RAM at a fixed virtual offset. With it the kernel reaches any physical page in constant time, without mapping it first.

Pager-backed VMO A VMO whose pages a *user-space* pager supplies on a fault, through a port. It allows demand paging, file caching and custom eviction outside the kernel.

PMT Pinned Memory Token. The object that `zx_bti_pin` returns. It stands for one pinning of VMO pages for DMA. Unpinning it (`zx_pmt_unpin`) releases the pages and removes the IOMMU mapping.

BTI Bus Transaction Initiator. A kernel object that stands for one device's view of memory through the IOMMU. Pinning a VMO against a BTI gives the bus addresses the device may use for DMA, and nothing else.

IOMMU Input/Output Memory Management Unit. The hardware that translates and limits device DMA, or a kernel stub in its place. Zircon programs it so that a device can reach only its pinned pages.

vDSO Virtual Dynamic Shared Object. The read-execute shared library that the kernel provides. It is the *only* legal entry point for syscalls: the kernel checks that the trap came from inside the vDSO image.

ZBI Zircon Boot Image. The container of typed items that the bootloader delivers and early boot reads. Its items include the kernel image, the command line, the `bootfs` ramdisk, the memory map and the platform id.

userboot The first user-space program, built into the kernel image. It receives the first handles over a bootstrap channel, reads `bootfs`, and launches the next program in the boot chain.

processargs The message protocol on the bootstrap channel by which a new process receives its first handles, each tagged with a `PA_*` type, and its arguments, environment and namespace. It is the hand-off of authority from kernel to user space.

Job policy Sandboxing rules attached to a job (`zx_job_set_policy`) that its descendants inherit. They set which object classes the descendants may create, whether W^X can be broken, and how violations are handled. Policy can only get stricter down the tree.

Restricted mode A per-thread execution mode with a separate register set and a restricted view of memory. It runs untrusted or foreign-ABI code, most notably Linux programs under Starnix. A syscall or fault returns control to the host handler.

Futex Fast Userspace muTEX. A 32-bit word in user memory with sleep and wake calls in the kernel (`zx_futex_wait` and `zx_futex_wake`). User-space mutexes and condition variables are built on it, and their fast path makes no syscall.

Port packet A fixed-size record taken from a port. It names its source (a signal, an interrupt, a user packet, a page request or a guest exit) and holds that source's payload. It is the unit of unified asynchronous notification.

Epitaph A final status value that a server sends on a channel just before it closes its end. It tells the other end *why* the connection ended.

Deadline scheduling The kernel's bandwidth and latency contract for a thread, given as (capacity, deadline, period). When a deadline thread is runnable, the scheduler runs it ahead of fair-weighted threads.

OwnedWaitQueue The kernel's wait-queue type that tracks an owning thread, for priority inheritance. A blocked high-importance waiter lends its scheduling weight to the thread that holds what it waits for.

TLB shutdown The protocol between CPUs that removes stale TLB entries on other cores after a mapping changes. It makes sure that no CPU keeps using a revoked translation.

REFERENCES

- [1] Google, "Zircon Kernel Concepts," *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel>
- [2] Google, "Zircon Kernel Objects," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/objects
- [3] Google, "Zircon System Calls," *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/reference/syscalls>
- [4] Google, "Handles and Rights," *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/handles>
- [5] Google, "Rights," *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/rights>
- [6] Google, "Virtual Memory Object (VMO)," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/vm_object
- [7] Google, "Virtual Memory Address Region (VMAR)," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/vm_address_region
- [8] Google, "Userspace Pager / Pager-backed VMOs," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/syscalls/pager_create
- [9] Google, "Channel," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/channel
- [10] Google, "Socket," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/socket
- [11] Google, "Port and Signals," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/port

- [12] Google, “Futex,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/futex
- [13] Google, “Zircon Fair Scheduler,” *Fuchsia RFCs / Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/kernel/fair_scheduler
- [14] Google, “Task Objects: Job, Process, Thread,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/job
- [15] Google, “Exception Handling,” *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/exceptions>
- [16] Google, “Interrupts,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/interrupts
- [17] Google, “Bus Transaction Initiator (BTI) and Pinned Memory,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/bus_transaction_initiator
- [18] Google, “RFC-0261: Fast and efficient user space kernel emulation,” *Fuchsia RFCs*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0261_fast_and_efficient_user_space_kernel_emulation
- [19] Google, “Bootting Fuchsia / userboot and the ZBI,” *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/process/userboot>
- [20] Google, “Clock and Time,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/kernel_objects/clock
- [21] Google, “Fuchsia RFCs,” *Fuchsia Documentation*, 2024. [Online]. Available: <https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs>
- [22] J. Anderson *et al.*, “LK / Zircon kernel scheduler and object internals,” public source, `zircon/kernel`, 2024.