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. Rather than a monolithic kernel exporting hundreds of ad-hoc syscalls, Zircon exposes a small, uniform set of *kernel objects* manipulated through unforgeable *handles* carrying fine-grained *rights*. Almost all OS policy—drivers, filesystems, the network stack, and the graphics pipeline—lives in user space, leaving the kernel responsible for memory, scheduling, inter-process communication, and the mediation of hardware access. This reference is a detailed tour of Zircon: the object/handle/rights model and the vDSO-based syscall ABI; the job/process/thread task hierarchy and the fair scheduler; the virtual- and physical-memory subsystems (VMOs, VMARs, the page queues, and the user-space pager); the IPC primitives (channels, sockets, FIFOs, ports, and futexes); interrupts, exceptions, and time; the hardware-access objects (resources, MMIO, interrupts, BTIs and the IOMMU); secure boot hand-off via `userboot`; restricted mode and the security model; and the kernel's diagnostics and multi-architecture support. The treatment targets engineers comfortable with hardware and RTL who want a precise mental model of how the kernel is structured and how its subsystems actually operate.**

***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 compact embedded kernel, but has grown into a general-purpose, object-capability microkernel that boots SMP machines, manages virtual memory, schedules threads, mediates hardware access, and routes inter-process communication—and almost nothing else. The defining decision is what Zircon does *not* contain: there are no in-kernel device drivers, no filesystems, no network stack, and no display or graphics pipeline. Every one of those subsystems runs in user space as an ordinary process, reachable only through the kernel's small, uniform message-passing primitives. The kernel exposes roughly 170 system calls—a surface deliberately kept an order of magnitude smaller than a monolithic kernel's—and treats that surface as a stable ABI [1], [3].

A. Design Axioms

Five axioms shape everything that follows.

1) A small, stable syscall surface: The complete kernel ABI is enumerated in one machine-readable specification and delivered to user space through a single kernel-provided shared library, the vDSO (Section IV). Keeping the surface small makes it auditable and lets the kernel evolve its internals without breaking binaries.

Volume III of the fourteen-volume *Fuchsia Internals* series (I: Graphics & Display Pipeline; II: Build System & Toolchains; III: The Zircon Kernel; IV: Starnix; V: Graphics FIDL APIs in Practice; VI: The Component Framework; VII: Driver Development; VIII: Storage; IX: Power Management; X: Networking; XI: FIDL; XII: Software Delivery & OTA; XIII: Diagnostics & Observability; XIV: Security & Verified Boot). Compiled 2026 from the public Fuchsia OS source tree and documentation at `fuchsia.dev` and `cs.opensource.google/fuchsia`.

2) Everything is a kernel object, named by a handle: A *kernel object* is a reference-counted entity living inside the kernel—a block of memory, a thread, a communication endpoint. User space never touches an object directly; it holds a *handle*, a per-process integer that names the object and carries a set of *rights*. The object is the noun; the handle is the only grammar by which a program may speak about it [2], [4].

3) Capabilities, not ambient authority: Authority in Zircon *is* the pair (handle, rights). There is no global `root`, no ambient filesystem namespace baked into the kernel, and no implicit permission inherited from a user identity. A process can do exactly what its handles permit and nothing more (Section V).

4) Pervasive asynchrony: Objects expose a 32-bit word of *signals* describing their state ("readable," "peer closed," "signaled"). Threads observe state changes by waiting on signals, and a `port` object multiplexes many such waits into a single queue, giving an event-loop substrate that the rest of the system is built on.

5) User-space-first policy: Mechanism lives in the kernel; policy lives in user space. The kernel knows how to map a page or deliver an interrupt, but the decision of *which* driver owns a device, or *what* filesystem format a partition holds, is made by user-space components.

B. Nouns and Verbs

A useful way to read the entire syscall list is as a grammar of nouns and verbs. The nouns are the object types—`vmo`, `channel`, `thread`, `port`—and the verbs are the syscalls that act on a handle to one of them. The naming convention makes this explicit: nearly every call has the form `zx_`*`noun`*`_`*`verb`*, as in `zx_channel_write`, `zx_vmo_create`, or `zx_thread_start`. Learning Zircon is largely a matter of learning the noun set (Section III) and then, for each noun, the handful of verbs it accepts.

C. A Hardware Analogy

For an engineer fluent in RTL, the object model has a faithful hardware reading. The kernel plays the role 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 an IP block's register file—a piece of addressable state with a defined behavior. A handle is an *unforgeable bus-grant token*: possessing it is what entitles a master to transact with that block, and the token's permission bits are exactly the object's rights. A `channel` is then an AXI-Stream link between two masters, and a memory object (a `vmo`) mapped into a process is a region the MMU has been told to expose at some virtual address. Crucially, just as a master on a real bus cannot forge a grant it was never given, a Zircon process cannot fabricate a handle: handles are minted only by the kernel and only handed out by syscalls or over channels.

Table I contrasts the two structures. The most consequential row is fault isolation: because a Fuchsia driver is a user process

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

Aspect	Zircon (microkernel)	Linux (monolithic)
Syscall surface	∼170, object-oriented (`zx_noun_verb`)	∼350+ POSIX calls
Drivers	User-space components	In-kernel modules
Filesystems	User-space services	In-kernel VFS
Network stack	User-space (`netstack`)	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

holding only the handles it needs, a driver defect crashes a process rather than the machine, and the crash is observable and recoverable through ordinary object mechanisms.

D. A First Syscall Sequence

Listing 1 shows the smallest interesting program: allocate a page of memory, map it into the calling process, store to it as ordinary RAM, and release the handle. Three objects appear—a `vmo` (the memory) and the process's root `vmar` (its address space)—and three verbs act on them. Note the teaching point on the penultimate line: closing the VMO handle does *not* unmap the page, because the mapping installed in the VMAR holds its own kernel reference to the object. The object outlives the handle.

Listing 1. A minimal create–map–write–close sequence. Error handling

```c
#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;
}
```

Two of the calls above never enter the kernel at all. `zx_vmar_root_self` returns a cached handle to the process's root address-space object, and time queries such as `zx_clock_get_monotonic` compute their result from a shared read-only page; these vDSO-local fast paths are discussed in Section IV.

E. The Kernel Surface

Figure 1 draws the whole system as one picture. At the center is the LK-derived core. Around it sit the families of object types the kernel implements, grouped by purpose. Enclosing them is the syscall/vDSO boundary—the single ring through which user space may enter the kernel. Outside that ring live the things that, in a monolithic system, would be *inside* it:

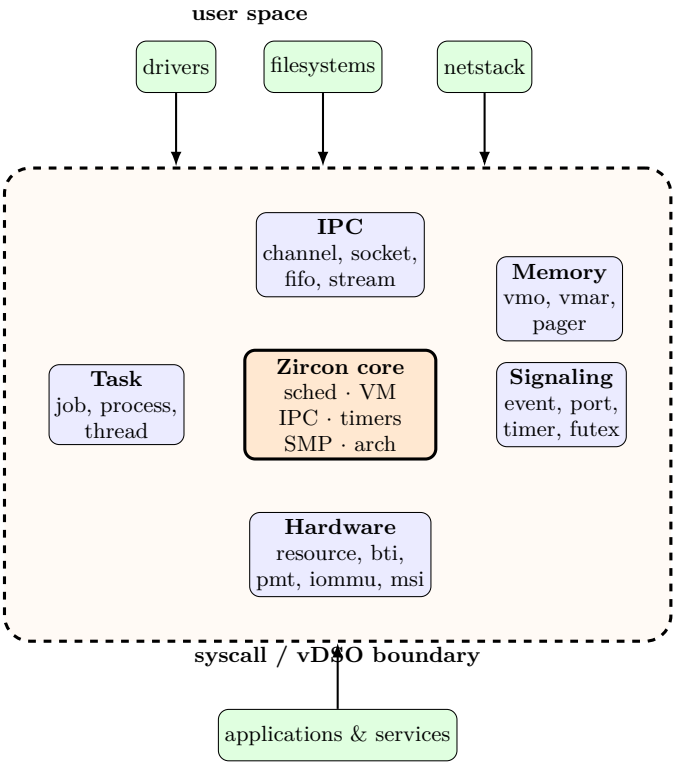


Fig. 1. The kernel surface. The LK-derived core implements the families of kernel objects (memory, IPC, task, signaling, hardware), enclosed by the single syscall/vDSO boundary. Drivers, filesystems, the network stack, and applications are user processes that reach the kernel only across that boundary.

drivers, filesystems, the network stack, and applications, each a user process wielding handles. The remainder of this Part walks inward: the architecture and kernel boundary (Section II), the object and handle model (Section III), the syscall ABI and the vDSO (Section IV), and the rights-based security model (Section V).

II. System Architecture and the Kernel Boundary

Internally Zircon is best understood as two cooperating layers stacked on the hardware abstraction. The lower layer is the *LK-derived core*: the scheduler, the virtual-memory machinery, timers, the SMP and per-CPU infrastructure, and the architecture-specific code under `arch/` that knows about page tables, exception vectors, and the instruction used to trap into the kernel. The upper layer is the *object layer*: the C++ classes that implement kernel objects, the per-process handle tables, and the syscall implementations that translate a user request into a method call on an object. Figure 2 shows the stack. User space sits above the syscall/vDSO boundary; everything below it is privileged.

A. The Dispatcher Pattern

Every object reachable from user space is a C++ class that derives from the abstract `Dispatcher` base. A virtual-memory object is a `VmObjectDispatcher`; a channel endpoint is a `ChannelDispatcher`; a thread is a `ThreadDispatcher`; jobs, processes, ports, timers, and the rest each have their own dispatcher subclass. The dispatcher holds the object's actual state and implements the operations the syscalls invoke. Code that needs to refer to an object generically does

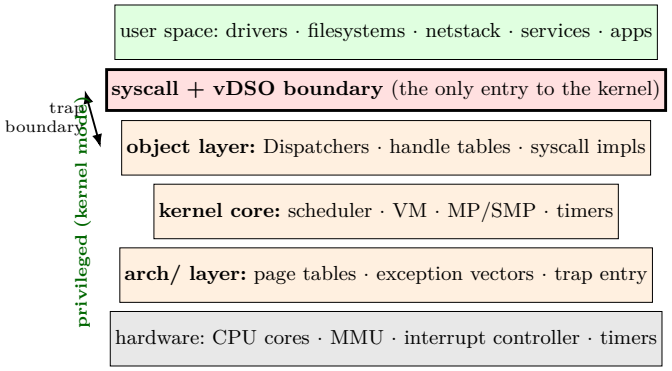


Fig. 2. Layered architecture. The single privileged trap boundary is the syscall/vDSO ring; below it, the object layer rests on the LK-derived core and the architecture-specific code, which in turn drives the hardware.

so through `fbl::RefPtr<Dispatcher>`, an intrusive reference-counted smart pointer [1].

Reference counting is the lifetime rule. A dispatcher stays alive exactly as long as something holds a strong reference to it: a handle in some process's handle table, an in-kernel reference (such as a VMAR mapping, as in Listing 1), or a transient reference held during a syscall. When the last strong reference drops, the object is destroyed deterministically. There is no garbage collector.

At construction every dispatcher is assigned a *KOID*, a 64-bit kernel object identifier (`zx_koid_t`) that is unique for the lifetime of the running system and never reused. KOIDs are how tooling and user space name objects for diagnostics; kernel-generated KOIDs occupy 63 bits, and two sentinels are reserved—`ZX_KOID_INVALID` (0) and `ZX_KOID_KERNEL`. KOIDs are stable identities even though the handle integers that point at an object differ from process to process.

B. The Handle Table

Each process owns a private *handle table*. A handle, as seen by user space, is a 32-bit integer (`zx_handle_t`); inside the kernel it is a table entry binding a `Dispatcher*` to a *rights* mask. The integer value encodes a table index together with a per-slot *generation count*; when a handle is closed and its slot is later reused, the generation field changes, so a stale handle value reliably fails to resolve rather than silently aliasing a new object. Valid handles additionally have their two least-significant bits set, and the reserved value `ZX_HANDLE_INVALID` is 0 [4].

Because the table is per-process, the same integer means different things—or nothing—in different processes. A handle is meaningful only in the address space that owns it. This is the kernel-level reason there is no ambient authority: there is simply no way to name an object you do not hold a handle to.

C. Syscall Flow, End to End

A syscall threads the whole stack of Figure 2. Consider `zx_channel_write`:

1) User code calls the vDSO stub `zx_channel_write`. The stub is the only code permitted to issue the machine trap instruction (`syscall` on x86-64, `svc` on ARM64).
2) The trap transfers control to the kernel's syscall entry trampoline, which saves register state and dispatches on the syscall number.
3) The kernel *validates arguments*: it copies and bounds-checks user pointers and lengths so that a malformed request cannot read or write kernel memory.
4) It performs *handle resolution*: the handle argument is looked up in the calling process's handle table, yielding the `Dispatcher*` and the handle's rights. The kernel checks the dispatcher is of the expected type and that the rights include those the operation requires—here `ZX_RIGHT_WRITE`—returning `ZX_ERR_ACCESS_DENIED` if not (Section V).
5) It invokes the *dispatcher method* (`ChannelDispatcher::Write`), which performs the actual work—copying the message, moving any transferred handles, raising the peer's `READABLE` signal.
6) A `zx_status_t` result propagates back through the trampoline and the vDSO stub to the caller, with any out-parameters written to user memory.

The return path is what makes the vDSO an enforcement boundary and not merely a convenience: the kernel records each process's vDSO mapping and checks the return program counter on entry, rejecting 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 contains the program's code and data segments, the read-only vDSO mapping (placed by the kernel and discoverable through the startup handle protocol), an initial thread stack, and a heap that the user-space allocator grows by creating VMOs and mapping them. Additional threads add additional stacks; shared memory appears as a VMO mapped into more than one process. Because each of these is just a VMAR mapping over a VMO, the same handful of memory verbs—`zx_vmo_create`, `zx_vmar_map`, `zx_vmar_allocate`—compose the entire address space. The virtual-memory subsystem is the subject of a later Part; here it suffices that the address space is itself built from kernel objects.

E. Sharing Through the Handle Table

Figure 3 makes the per-process indirection concrete. Two processes each have their 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 reference each other. Both processes also map a shared VMO: each has a handle resolving to the *same* `VmObjectDispatcher`, possibly with different rights—perhaps A may write while B is restricted to read. The diagram shows the entry structure: a handle integer decodes to (index, generation), and the slot stores (`Dispatcher*`, rights). The same object is named by different integers in the two tables, and may be named with different authority.

III. Kernel Objects and Handles

This section pins down the vocabulary used throughout the rest of the reference: the precise meaning of object, KOID, handle, dispatcher, and the operations that create, copy, transfer, and destroy handles.

A. Definitions

A *kernel object* is a reference-counted entity managed by the kernel and implemented, as Section II described, by a

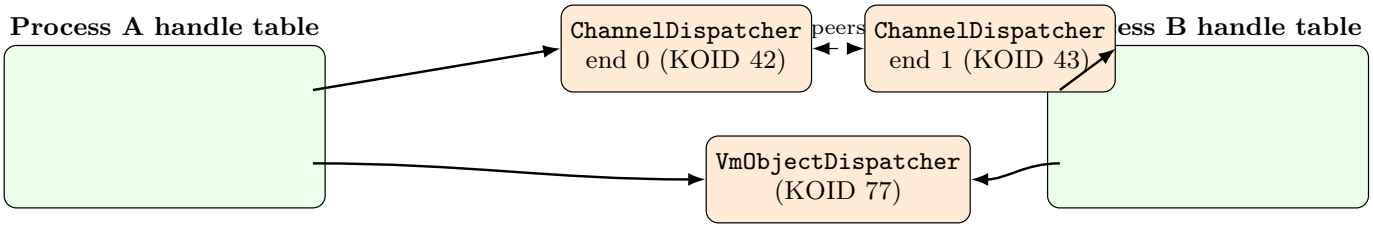


Fig. 3. Two processes sharing objects through their handle tables. Each handle integer decodes to (index, generation) and indexes a slot holding (`Dispatcher*`, rights). A channel's two endpoints are distinct peered `ChannelDispatcher`s; the shared VMO is one `VmObjectDispatcher` named by both processes, here with asymmetric rights (A read–write, B read-only).

`Dispatcher` subclass. Each object carries a *KOID* (`zx_koid_t`): a globally unique, monotonically allocated 64-bit identifier, never reused for the lifetime of the running system [1], [2]. A *handle* is a per-process 32-bit name (`zx_handle_t`) for an object, paired in the handle table with a *rights* mask (`zx_rights_t`). Some objects come in peered pairs—a `channel` has two endpoints, an `eventpair` two ends—and each endpoint exposes its partner's identity as a *related KOID*. Introspection on a channel endpoint thus reports both its own KOID and the `related_koid` of the other end, which is how user space discovers that two handles it holds are the two sides of one link.

B. Object Lifecycle

The lifecycle is deterministic and contains no garbage collection. An object is *created* by a syscall—`zx_vmo_create`, `zx_channel_create`, `zx_event_create`—which returns one or more handles to it. The object remains alive as long as any strong reference exists: a handle in some process's table, an internal kernel reference (a VMAR mapping, a queued message in transit on a channel, a wait registered against it), or a transient reference during a syscall. When the final reference drops the object is *destroyed* immediately. Closing a handle decrements the count; when the count reaches zero the dispatcher's destructor runs and any resources (physical pages, queued messages) are released.

C. Handle Anatomy and Operations

A handle bundles three things in the kernel: a reference to its object, the rights governing it, and the binding to the owning process (or to the kernel itself while the handle is "in transit" across a channel) [4]. The core verbs are:

- `zx_handle_duplicate(h, rights, out)` — produce a second handle to the same object. This requires that `h` itself carry `ZX_RIGHT_DUPLICATE`. The new handle's rights are the requested subset, or, when `ZX_RIGHT_SAME_RIGHTS` is passed, a copy of the original's rights.
- `zx_handle_replace(h, rights, out)` — atomically consume `h` and return a new handle to the same object with equal-or-lesser rights. The original integer is invalidated; there is no window in which both exist.
- `zx_handle_close(h)` — remove the handle from the table, dropping its reference. Closing `ZX_HANDLE_INVALID` is explicitly legal (a no-op), which simplifies cleanup paths.
- `zx_handle_close_many(handles, n)` — close a vector of handles in one call.

Handles also *transfer*. Writing a handle into a channel with `zx_channel_write` removes it from the sender's table and binds it to the kernel in transit; the receiver's `zx_channel_read` installs it into the receiving process's table. Transfer requires `ZX_RIGHT_TRANSFER` and is move semantics, not copy: at no instant do two processes both name the object through that same handle. This is the mechanism by which capabilities flow through the system—to grant a peer access to an object you send it a handle, and to grant *reduced* access you duplicate the handle down to a smaller rights set first (Section V).

D. Introspection

The reflective verb is `zx_object_get_info`. With the topic `ZX_INFO_HANDLE_BASIC` it returns a record describing a handle: the object's `koid`, the handle's `rights`, the object `type` (`ZX_OBJ_TYPE_CHANNEL`, `ZX_OBJ_TYPE_VMO`, ...), and the `related_koid` for peered objects. Other topics report per-type detail (`ZX_INFO_PROCESS`, `ZX_INFO_VMAR`, `ZX_INFO_JOB_CHILDREN`). Objects also carry properties accessed through `zx_object_get_property` / `zx_object_set_property`; the most common is `ZX_PROP_NAME`, a short human-readable string that makes threads, processes, and VMOs legible in diagnostics. Reading a property requires `ZX_RIGHT_GET_PROPERTY`; writing one requires `ZX_RIGHT_SET_PROPERTY`.

E. Peered Objects and Closure Signals

Channels, sockets, FIFOs, and eventpairs are *peered*: created as a pair, each end a separate object referencing the other. Their defining behavior is the `PEER_CLOSED` signal. When one endpoint is closed, the kernel asserts `ZX_CHANNEL_PEER_CLOSED` (respectively `ZX_SOCKET_PEER_- CLOSED`, `ZX_FIFO_PEER_CLOSED`, `ZX_EVENTPAIR_PEER_CLOSED`) on the surviving endpoint. A thread or port waiting on that endpoint is woken and learns, definitively, that no further messages can arrive—the kernel-level equivalent of a hang-up. Combined with the readable/writable signals, this gives a complete, race-free protocol for stream and datagram teardown without any out-of-band convention.

F. The Object Catalog

Table II enumerates the major object types grouped by family, each with its one-line role. This is the noun set of Section I-B; the verbs for each are covered in their respective later Parts.

G. Object, Handle, Dispatcher Together

Figure 4 consolidates the model on the canonical example: two processes sharing a channel. Each process holds, in its own table, a handle to one endpoint; each handle resolves to a `ChannelDispatcher`; the two dispatchers are peers. The figure also annotates the rights on each handle—both sides typically hold `READ`, `WRITE`, `TRANSFER`, `WAIT`, and the signal rights, but not `DUPLICATE` (channel endpoints are intentionally non-duplicable,

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

Family	Object	Role
IPC	`channel`	Bidirectional datagram link carrying bytes *and* handles; the substrate of FIDL IPC.
	`socket`	Bidirectional 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: shared-memory region with disciplined access for high-throughput transport.
Memory	`vmo`	Virtual Memory Object: a sized, resizable container of pages—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; the root of 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 multiplexes asynchronous signal and packet notifications; the event-loop core.
	`timer`	One-shot/deadline object that asserts a signal at a scheduled time.
	`futex`	Address-based fast user-space lock primitive; no handle, keyed by a user address.
	`interrupt`	Binds a hardware IRQ to a waitable object delivered to a user-space driver.
	`counter`	64-bit signed counter; atomic add/read, asserts a signal when its value is (non-)positive—a shared-memory signaling primitive.
Hardware	`resource`	Unforgeable capability gating access to physical address ranges, IRQs, and privileged operations.
	`bti`	Bus Transaction Initiator: represents a device's DMA view; 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.
	`msi`	Message-Signaled Interrupt allocation for PCI-style devices.
System / misc	`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`	Append-only handle to the kernel's debug log.
	`exception`	Object representing a faulting thread delivered to a debugger/handler.
	`suspend_token`	Holds a task suspended; the task resumes when the token handle is closed.

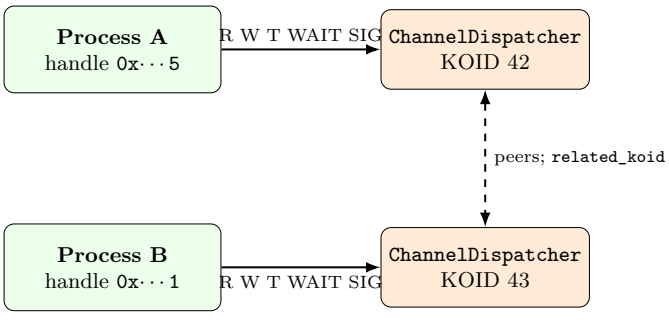


Fig. 4. The object/handle/dispatcher relationship for a shared channel. Each process names one endpoint through a handle in its own table; the handles resolve to the two peered `ChannelDispatcher`s, which expose each other through `related_koid`. Edge labels are the handle's rights.

so an endpoint has at most one owner). This single picture—handle to dispatcher, dispatcher to peer, rights on the edge—recurs throughout Zircon.

Finally, note the property that ties objects to the asynchrony axiom of Section I: most object types are *waitable*, exposing a 32-bit `zx_signals_t` word whose bits report state changes. The mechanics of observing those bits—`zx_object_wait_one`, `zx_object_wait_async`, and ports—are the subject of the later Signals and Ports Part; here it is enough that "waitable" is a near-universal property of the catalog above.

IV. THE SYSTEM CALL ABI AND THE VDSO

Zircon's system-call ABI is delivered through a single kernel-provided shared library, the *vDSO* (virtual Dynamic Shared Object). The vDSO is not loaded from any filesystem; the kernel embeds its image at build time and exposes it to each process as a read-only VMO mapped into the address space at startup. It is both the convenience layer that gives user space ordinary C functions and the enforcement boundary that defines what "a syscall" even means [1], [3].

A. The vDSO as the Sole Gateway

The vDSO is the *only* legal place from which to issue a system call. Executing the raw trap instruction (`syscall` on x86-64,

`svc` on ARM64) from application code is not a supported back door: when a thread enters the kernel, the kernel subtracts the base address of the recorded vDSO mapping from the trapping program counter and tests it against a per-syscall validity predicate. A trap whose PC does not fall within the vDSO's code at a legitimate syscall site is rejected with a synthetic exception, exactly as if the thread had executed an undefined or privileged instruction. To make this check cheap and unambiguous, the kernel permits each process exactly one executable vDSO mapping, at the correct size and offset, and records its address when the mapping is established.

Three motives justify the boundary. First, *ABI stability*: pinning the trap sequence inside a kernel-shipped library means the user/kernel calling convention can change without recompiling user binaries, since everyone calls through the same stubs. Second, *enforcement*: the return-PC check turns "which code may talk to the kernel" into a hard, kernel-checked invariant rather than a convention. Third, *fast paths*: because the kernel controls the library, it can implement some calls entirely in user space.

B. vDSO-Local Calls

Not every `zx_*` function traps. Some are computed entirely within the vDSO from a read-only page the kernel shares with the process, the `vdso_constants` block, which the kernel initializes 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, and `zx_clock_get_monotonic` scales that raw counter into nanoseconds using the conversion ratio stored in `vdso_constants`— all without entering the kernel. Syscalls therefore fall into three performance classes:

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

C. Definition and Code Generation

The entire ABI is described in FIDL-syntax specification files under `//zircon/vdso`. Each object type's syscalls are grouped into a `@transport("Syscall")` protocol; a method declaration names its parameters, their types, the handle type expected, and the error return. A generator (the `zither` toolchain) reads these files and emits both the public C/C++ declarations user space links against and the private kernel-side entry interface. Methods marked `vdsocall` are implemented by hand-written vDSO C++ rather than trapping; methods marked `internal` are private entries used only by the vDSO implementation. Listing 2 shows the definition of `zx_channel_write` and the C signature generated from it.

Listing 2. The FIDL-style syscall definition of `Channel.Write` (top) a

```
// //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 zither emits for the definition above. The `@release` annotation is why transferred handles leave the caller's t

```
// 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);
```

The `vector` parameters expand into a pointer plus a 32-bit count, and the `error Status` clause is what makes every syscall return a `zx_status_t`. The `@release` annotation on the handle vector is the formal source of move semantics: the listed handles are removed from the caller's table as part of the call.

D. The Error Model

Zircon has no `errno`. Every syscall returns a `zx_status_t`: the value `ZX_OK` (0) on success, or a negative `ZX_ERR_*` code on failure. The status is the *only* channel for error information; results are delivered through out-parameters that the kernel writes only on success. Out-parameters of handle type convey *ownership*—a successful `zx_channel_read` installs the received handles into the caller's table, and the caller is thereafter responsible for closing them. The common codes a programmer meets constantly are listed in Table-like prose here: `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` (right object kind, wrong for this call), `ZX_ERR_INVALID_ARGS` (malformed parameters), `ZX_ERR_SHOULD_WAIT` (a non-blocking call found nothing ready), `ZX_ERR_PEER_CLOSED` (the other end of a peered object is gone), and `ZX_ERR_BUFFER_TOO_SMALL` (the supplied buffer cannot hold the result). Because the codes are uniform across the whole surface, error handling composes: the same `if (st != ZX_OK)` idiom of Listing 1 works for every call.

E. ABI Categories and Restricted Views

The specification annotates calls with a stability category. Most are *stable* and constitute the committed ABI. Calls under development are marked `@next`, visible to in-tree callers building against the "next" ABI but not yet frozen. Separately, a thread running in *restricted mode*—the mechanism Fuchsia uses to host foreign (e.g. Linux) binaries under the Starnix runtime—executes with a curtailed view in which direct syscalls are trapped back to a supervising normal-mode thread rather than serviced directly. Restricted mode is part of the security architecture treated in a later Part; here it is enough to note that "what syscalls a thread may make" is itself a configurable property, not a global constant.

F. The Two Call Paths

Figure 5 contrasts the trapping and non-trapping paths. Table III then samples one representative call from each major functional category, to convey the breadth of the ∼170-call surface without enumerating it.

V. Capabilities: Rights and the Security Model

Sections III and IV established that authority flows as handles. This section gives the other half of the capability: the *rights* that qualify a handle, the monotonic rule that governs how they change, and the way the kernel enforces them. Together, (handle, rights) is the unit of authority in Fuchsia [4], [5].

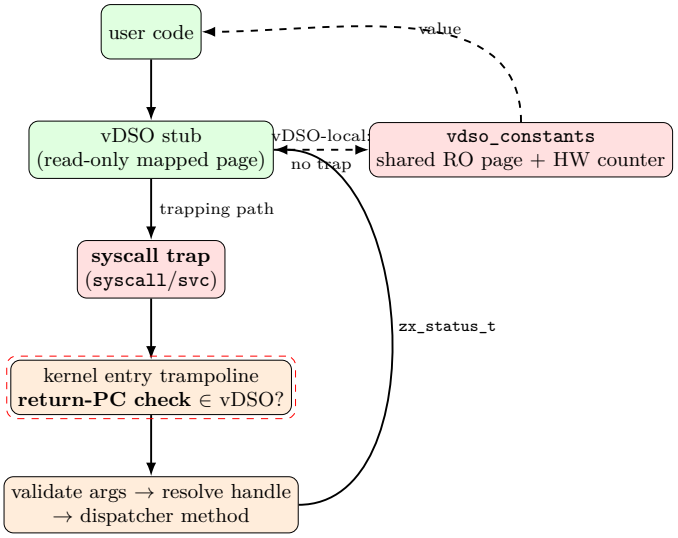


Fig. 5. The two syscall paths. Trapping calls cross the boundary through the vDSO stub and the trap instruction; the kernel's entry trampoline enforces that the return PC lies within the recorded vDSO mapping before validating arguments, resolving the handle, and invoking the dispatcher. vDSO-local calls (dashed) return a value computed from a shared read-only page without ever entering the kernel.

TABLE III
REPRESENTATIVE SYSCALLS BY FUNCTIONAL CATEGORY.

Category	Example	Action
Object	`zx_object_get_info`	Introspect a handle/object.
Handle	`zx_handle_duplicate`	Copy a handle at ≤ rights.
Task	`zx_process_start`	Begin executing a process.
VM	`zx_vmar_map`	Map a VMO into an address space.
IPC	`zx_channel_call`	Send and await a reply atomically.
Port/sig	`zx_port_wait`	Dequeue the next async packet.
Time	`zx_clock_get_monotonic`	Read monotonic time (vDSO-local).
Hardware	`zx_bti_pin`	Pin VMO pages for device DMA.
System	`zx_system_get_num_cpus`	Query CPU count (vDSO-local).

A. The Rights Bitmask

Each handle carries a `zx_rights_t`, a 32-bit mask. The kernel assigns each named right a single bit; the important ones are:

- `ZX_RIGHT_DUPLICATE` — may be duplicated by `zx_handle_duplicate`.
- `ZX_RIGHT_TRANSFER` — may be written into a channel and thus moved to another process.
- `ZX_RIGHT_READ` / `ZX_RIGHT_WRITE` — read or write the object's contents (VMO bytes, channel messages); combined with `MAP`, govern readable/writable mappings.
- `ZX_RIGHT_EXECUTE` — with `MAP`, permits an executable mapping of a VMO.
- `ZX_RIGHT_MAP` — the VMO may be mapped into a VMAR at all.
- `ZX_RIGHT_GET_PROPERTY` / `ZX_RIGHT_SET_PROPERTY` — read or write object properties (e.g. `ZX_PROP_NAME`).
- `ZX_RIGHT_ENUMERATE` — list a job's or process's children.
- `ZX_RIGHT_DESTROY` — kill a task object.
- `ZX_RIGHT_SIGNAL` / `ZX_RIGHT_SIGNAL_PEER` — assert user signals on the object itself, or on a peered object's partner.
- `ZX_RIGHT_WAIT` — wait on the object's signals (`zx_object_wait_one`, ports).
- `ZX_RIGHT_INSPECT` — introspect via `zx_object_get_info`.
- `ZX_RIGHT_MANAGE_JOB` / `_PROCESS` / `_THREAD` — create or control children/threads within a task.
- `ZX_RIGHT_APPLY_PROFILE` — apply a scheduling profile to a thread.
- `ZX_RIGHT_RESIZE` — resize a resizable VMO (with `WRITE`).

The kernel also defines composite shorthands used when minting defaults: `ZX_RIGHTS_BASIC` = `DUPLICATE | TRANSFER | WAIT | INSPECT`; `ZX_RIGHTS_IO` = `READ | WRITE`; `ZX_RIGHTS_PROPERTY` = `GET_PROPERTY | SET_PROPERTY`; and `ZX_RIGHTS_POLICY` = `GET_POLICY | SET_POLICY`.

B. The Monotonic-Narrowing Rule

The cardinal invariant is that **rights only ever decrease**. There is no syscall, anywhere in the ABI, that adds a right to a handle. The two operations that produce new handles can only preserve or shed rights:

- `zx_handle_duplicate(h, r, out)` yields a new handle whose rights are r, which must be a *subset* of `h`'s rights; requesting a superset fails with `ZX_ERR_INVALID_ARGS`. Passing the sentinel `ZX_RIGHT_SAME_RIGHTS` copies the existing rights unchanged.
- `zx_handle_replace(h, r, out)` consumes `h` and returns a handle with rights $r \leq$ `h`'s rights, again with `ZX_RIGHT_SAME_RIGHTS` meaning "unchanged."

Because amplification is simply absent from the kernel, a process can reason about the worst case: any handle it ever hands out, directly or after passing through other processes, can confer no more authority than the handle it started from. This is what makes capability *attenuation* sound.

C. Default Rights

When an object is created, the handle returned by the creating syscall is stamped with a *default* rights set chosen for that object type. The defaults (Table IV) are deliberately generous-but-bounded: they include exactly the rights the object's normal verbs need and omit ones that would be surprising. Two omissions are instructive. A channel endpoint's default *lacks* `DUPLICATE`: an endpoint is meant to have a single owner, so it can be transferred but not copied. A freshly created VMO's default *lacks* `EXECUTE`: code cannot be made executable merely by writing it into memory; an executable handle must be obtained separately (gated by a resource), which is a direct kernel-level mitigation against code injection.

D. Capabilities versus Ambient Authority

The contrast with POSIX is sharp and deliberate. In a POSIX system, authority is *ambient*: a thread's user and group identity (`uid`/`gid`) silently apply to every operation, the superuser `root` bypasses checks wholesale, and any process can name any path in a global filesystem namespace, its access decided after the fact by permission bits. Privilege is escalated by identity transitions such as `setuid`. Zircon has none of this. There is no `uid`, no `root`, and no kernel-global namespace; a thread's authority is precisely the set of handles in its process's table, each clamped by its rights. To grant authority you must hand over a handle; to

TABLE IV
DEFAULT RIGHTS STAMPED ON THE HANDLE RETURNED AT OBJECT CREATION. 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)
`vmar`	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

deny it you simply never do. The question "what can this process do?" has an exact, enumerable answer—its handle table—rather than depending on a global identity.

E. How Rights Are Enforced

Enforcement happens at handle resolution, the step in the syscall flow of Section II-C. Each dispatcher method declares the right(s) its operation requires; when the kernel resolves the handle argument it checks the handle's rights mask against that requirement and returns `ZX_ERR_ACCESS_DENIED` on a shortfall, before the operation runs. Concretely: `zx_vmo_write` requires `ZX_RIGHT_WRITE` on the VMO handle; mapping that VMO writable through `zx_vmar_map` additionally 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. The checks are simple bit tests, but because every path funnels through resolution, they are comprehensive.

F. Building Least Privilege

The narrowing rule is the tool with which a parent constructs a least-privileged child. The pattern (Figure 6) is: start from a full-rights handle, `duplicate` it down to the minimal subset the child needs, then `transfer` that reduced handle into the child over a channel. To share a configuration blob read-only, duplicate the VMO handle requesting only `READ | MAP | TRANSFER` and send that; the child can map and read but never modify the page, regardless of bugs. To let a child spawn its own processes but not reach back up the task tree, hand it a job handle deliberately stripped of `MANAGE_JOB`. Each such decision is local, explicit, and enforced by the same machinery as every other syscall.

Finally, these kernel rights are the substrate beneath Fuchsia's higher-level component capability routing: when the component framework "routes a capability" from one component to another it is, at bottom, arranging for the appropriate handle—narrowed to the appropriate rights—to be transferred over a channel. The component model adds declaration, naming, and policy on top, but the authority it moves is exactly the (handle, rights) pair defined here. The remaining Parts of this reference build every subsystem—tasks, memory, IPC, interrupts, and hardware access—on this same foundation.

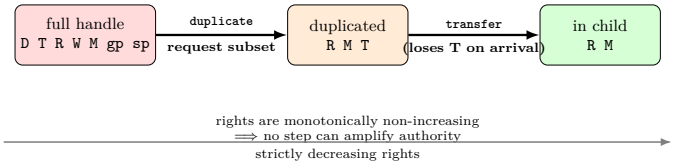


Fig. 6. Rights narrowing as a one-way pipeline. A full-rights handle is duplicated down to a chosen subset, then transferred into a child process, strictly losing authority at each step. No operation moves rightward up the lattice.

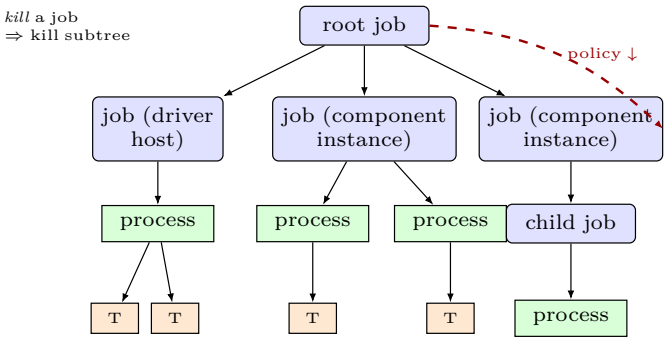


Fig. 7. The task hierarchy. A single root job anchors a tree of jobs, processes, and threads. Job policy and lifetime propagate downward; a child job may only tighten inherited policy.

VI. TASKS: JOBS, PROCESSES, AND THREADS

Zircon groups all runnable state under three *task* object types that form a strict containment hierarchy: *jobs* contain processes and child jobs, *processes* contain threads, and *threads* are the unit the scheduler dispatches. The three share a common control surface—a task can be named, queried, suspended, killed, and given an exception channel—but each adds its own structure. This section describes the hierarchy, the lifecycle of each object, the *job policy* mechanism that turns the tree into a sandboxing substrate, and the debugger-facing introspection used by `zxdb` [1], [14].

A. The Task Hierarchy

Every task descends from a single *root job* created by the kernel at boot and handed to `userboot` (see the boot section). 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 is not merely organizational. Two things flow strictly downward along it: *job policy* (security and resource constraints, §VI-F) and *lifetime* (killing a job recursively kills everything beneath it). A child job can only tighten—never loosen—the policy it inherits, so the root job's constraints bound the entire system, and each component-manager-created job for a component instance bounds that subtree. Figure 7 shows the shape.

B. Jobs

A *job* is a container and a policy boundary; it owns no address space and runs no code itself. It is created from a parent job with `zx_job_create(parent, options, &out)`,

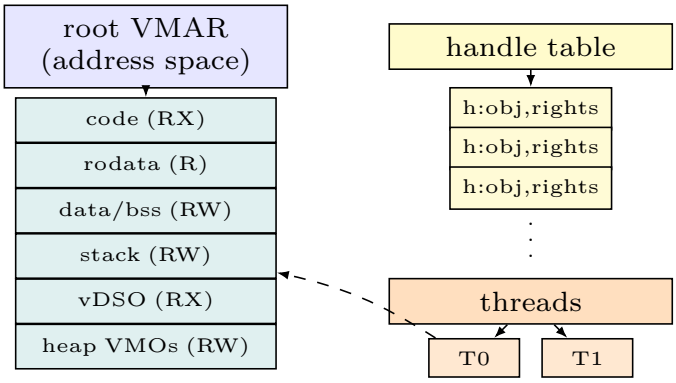


Fig. 8. Process anatomy: a root VMAR holding code, data, stack, and vDSO mappings; a handle table binding integer handles to objects and rights; and one or more threads, each with its own stack mapping.

which means job creation is itself governed by the parent's policy (the `ZX_POL_NEW_PROCESS`-class conditions). A job tracks its children, exposes aggregate resource accounting (e.g. peak memory across the subtree via `zx_object_get_info`), and serves as the unit of mass termination: `zx_task_kill` on a job tears down every descendant process and thread. Jobs also carry *critical* relationships (§VI-E) and the *timer-slack* policy that sets the default coalescing behavior for timers created under them.

C. Processes

A *process* is the unit of memory isolation and capability holding. Its two defining pieces are a *root VMAR* (the top of its virtual-address-space tree—see the virtual-memory section) and a *handle table* mapping the small integer handle values the process sees to the underlying kernel objects and their rights. A process is created with `zx_process_create(job, name, name_size, options, &proc, &vmar)`, which returns both a process handle and a handle to its root VMAR but does *not* start execution. Figure 8 shows the anatomy.

A crucial design choice is that **the kernel does not load ELF**. Process startup is a user-space dance: a loader (often the dynamic linker `ld.so` bootstrapped from the vDSO) maps the program's segments and the vDSO into the new `vmar`, constructs an initial stack, creates the first thread, and then calls `zx_process_start`. That call atomically transfers a single *bootstrap* handle—typically a channel endpoint carrying the program's namespace, arguments, environment, and additional handles—into the new process and begins execution at a given entry point. Listing 4 sketches the sequence.

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

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

// (User space) map ELF PT_LOAD segments and the vDSO
// into 'vmar', 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, 0);
```

D. Threads

A *thread* is the schedulable unit: it owns architectural register state and a user stack, and it lives inside exactly one process, sharing that process's address space and handle table with its siblings. Threads are made with `zx_thread_create(process, name, name_size, options, &out)` and launched with `zx_thread_start(handle, entry, stack, arg1, arg2)`, which sets the program counter and stack pointer and passes two register-width arguments; all other registers start zero. The very first thread of a process is special only in that it is started implicitly by `zx_process_start` rather than `zx_thread_start`. Inside a running process, additional threads (e.g. those a C library's `thrd_create`/`pthread_create` spins up) follow the same path.

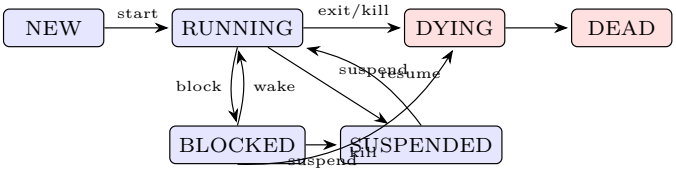


Fig. 9. Thread state machine. A thread alternates between RUNNING, BLOCKED, and SUSPENDED until it exits or is killed, passing through DYING to DEAD; reaching DEAD asserts `ZX_THREAD_TERMINATED`.

E. Lifecycle, States, and Termination

A thread moves through the states NEW → RUNNING ↔ BLOCKED (waiting on a futex, channel, port, interrupt, pager, or sleep) ↔ SUSPENDED → DYING → DEAD. A process has the coarser progression NEW → RUNNING → DYING → DEAD; it becomes RUNNING when its first thread starts and DEAD when its last thread exits or it is killed. Termination is observable: a task asserts the `ZX_TASK_TERMINATED` signal when it reaches DEAD, and watchers blocked in `zx_object_wait_one` or registered on a port are released. A task carries a *return code*, retrievable via `zx_object_get_info` with `ZX_INFO_PROCESS`; killing a task with `zx_task_kill` forces a synthetic return code. Figure 9 draws the thread state machine.

Critical relationships let 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` critical to an ancestor `job`; if that process dies, the kernel kills the job (and thus its subtree) as if by `zx_task_kill`, using the return code `ZX_TASK_RETCODE_CRITICAL_PROCESS_KILL`. With the option `ZX_JOB_CRITICAL_PROCESS_RETCODE_NONZERO` the job dies only on a non-zero exit. This is how a clean failure of an essential service can be escalated all the way up 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*—an operation a contained process might attempt—with an *action* the kernel takes when that operation occurs. Conditions cover handle misuse (`ZX_POL_BAD_HANDLE`, `ZX_POL_WRONG_OBJECT`), the write–execute memory taboo (`ZX_POL_VMAR_WX`, `ZX_POL_AMBIENT_MARK_VMO_EXEC`), and object creation (`ZX_POL_NEW_VMO`, `ZX_POL_NEW_CHANNEL`, `ZX_POL_NEW_PROCESS`, …, or the umbrella `ZX_POL_NEW_ANY`). Actions escalate from `ZX_POL_ACTION_ALLOW` through `ZX_POL_ACTION_DENY` (fail the syscall with `ZX_ERR_ACCESS_DENIED`), their `_EXCEPTION` variants (which additionally raise a debugger-visible policy exception), up to `ZX_POL_ACTION_KILL` (terminate the offending process outright). 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 field selects absolute vs. relative application, and inherited entries can never be relaxed. Listing 5 confines a subtree. The security section develops this into the full sandbox story; here it suffices to note 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 same task API underpins zxdb. 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, composing cleanly with multiple debuggers. While suspended, register state is read and written with zx_thread_read_state and zx_thread_write_state, parameterized by a *kind*: ZX_THREAD_STATE_GENERAL_REGS, _FP_REGS, _VECTOR_REGS, _DEBUG_REGS (hardware breakpoints/watchpoints), and _SINGLE_STEP. Faults and policy violations are delivered through *exception channels* obtained with zx_task_create_exception_channel on a thread, process, or job, giving a debugger a layered chance to handle a fault before it propagates upward.

Exception delivery is ordered along the task hierarchy. When a thread takes a hardware fault (a page fault, an illegal instruction) or trips a policy exception, the kernel offers it first to the process's *debugger* exception channel, then to the *thread* channel, then to the *process* channel, and finally up the chain of *job* channels toward the root. Each handler receives an exception message naming the faulting thread and the exception type; it may inspect and rewrite register or memory state and then either resolve the exception (ZX_EXCEPTION_STATE_HANDLED) or pass it to the next handler (_TRY_NEXT). A debugger that wants a last look before the thread is killed registers as *second-chance* (ZX_EXCEPTION_STRATEGY_SECOND_CHANCE), so the exception returns to it after the process and job handlers decline. If no handler resolves the fault, the thread—and, by the process's design, usually the process—is terminated and a crash report is generated.

H. Introspection and Resource Accounting

Every task is observable through zx_object_get_info, the uniform query interface, selected by a *topic*. ZX_INFO_PROCESS reports whether a process has started or exited and its return code; ZX_INFO_PROCESS_THREADS and the job topics ZX_INFO_JOB_CHILDREN and ZX_INFO_JOB_PROCESSES return the koids of children, letting a supervisor walk the tree it built. ZX_INFO_TASK_STATS and the maps/VMOs topics expose a task's memory footprint, while ZX_INFO_TASK_RUNTIME returns a zx_info_task_runtime_t accumulating, in nanoseconds, the time the task spent running on a CPU (cpu_time), the time it spent runnable but waiting in a run queue (queue_time), and—on recent ABIs—time lost to page faults and lock contention. These counters are the raw material for user-space profilers and for the scheduler-tuning feedback loop that assigns profiles (§VII-E). Listing 6 reads a thread's CPU and queue time. Table V summarizes the task-management surface.

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

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

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's design has two pillars that coexist on every CPU: a *fair* scheduler based on weighted fair queuing in a virtual-time domain, which gives every thread a share of the CPU proportional to its weight; and a *deadline* (bandwidth) scheduler that grants hard, periodic CPU reservations to latency-critical work and dispatches it ahead of all fair threads. Both are exposed to user space through a single object—the *profile*—and both interact with the priority-inheritance machinery that prevents an important thread from being stalled behind a less important lock holder. This section explains each in turn [13], [21], [22].

A. Per-CPU Run Queues and Affinity

Scheduling is *per-CPU*. Each CPU owns its run queues and makes local dispatch decisions; there is no single global ready list to contend on. A thread is eligible to run on the set of CPUs allowed by its *affinity mask*, and within that set the system load-balances by migrating or *stealing* threads from busier CPUs when one goes idle, and by placing a newly runnable thread on a CPU chosen to balance load while honoring cache locality. Affinity comes in two flavors. A *hard* affinity mask is a strict constraint—the thread will only ever run on those CPUs. A *soft* affinity mask is a preference the load balancer tries to honor but may override to avoid leaving a thread un-runnable or a CPU idle. Pinning a real-time thread to a dedicated CPU set is the canonical use of hard affinity.

B. From Priorities to a Fair Scheduler

Early Zircon used a conventional priority-based scheduler: 32 priority levels (ZX_PRIORITY_LOWEST = 0, ZX_PRIORITY_DEFAULT = 16, ZX_PRIORITY_HIGHEST = 31) with the runnable thread of highest priority always winning. Strict priority is simple but brittle: it starves low priority threads, couples "importance" to "latency" in a single number, and

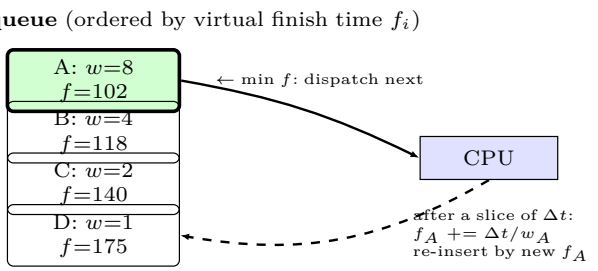


Fig. 10. Fair-scheduler dispatch. Runnable threads are kept ordered by virtual finish time f_i. The dispatcher runs the minimum-f thread; after a slice, its f advances by an amount inversely proportional to its weight and it is re-inserted, so bandwidth is shared in proportion to weight.

makes proportional sharing impossible. Zircon replaced it with a *fair scheduler* modeled on weighted fair queuing (WFQ), in which the old priority becomes a *weight* and the dispatch decision is made in a virtual-time domain rather than by raw priority comparison.

C. Fair-Scheduler Mechanics

Each fair thread carries a *weight* w_i derived from its profile's base priority. Weights set the *proportion* of CPU a thread receives when it competes: a thread with weight w_i on a CPU whose runnable set has total weight $W = \sum_j w_j$ is entitled to a fraction w_i/W of that CPU's time. The scheduler realizes this with a per-thread *virtual timeline*. When a thread runs, its virtual time advances at a *normalized rate* inversely proportional to its weight—heavy threads' virtual clocks tick slowly, light threads' tick fast—so that consuming the same wall-clock slice costs a light thread more virtual time. Each runnable thread is assigned a *virtual finish time*: the virtual time at which it would complete its current normalized time slice. The dispatcher simply runs the runnable thread with the **earliest virtual finish time**, pulling it from an ordered (balanced-tree) run queue in $O(\log n)$. Figure 10 illustrates.

Two properties follow directly. *Proportional sharing*: over any interval in which the runnable set is stable, each thread's received CPU time converges to w_i/W because virtual finish times advance at rates set by the weights. *Starvation-freedom*: even the lightest thread's virtual finish time is finite and is eventually the minimum, so it is guaranteed to run—unlike strict priority, no thread is indefinitely deferred. A *time slice* (bounded by a target scheduling latency, subdivided among the runnable threads, with a minimum granularity floor to cap context-switch overhead) bounds how long any thread monopolizes the CPU before the queue is re-examined, which bounds dispatch latency for newly-woken threads.

As a concrete instance, consider four CPU-bound threads sharing one CPU with weights $8, 4, 2, 1$ (total $W = 15$). Over any window long enough to amortize the slice quantum, they receive $\frac{8}{15}, \frac{4}{15}, \frac{2}{15}, \frac{1}{15}$ of the CPU—roughly $53\%, 27\%, 13\%, 7\%$—rather than the all-or-nothing split a strict-priority scheduler would impose. Doubling the heaviest thread's weight to 16 shifts the totals to $W = 23$ and gives it $\frac{16}{23} \approx 70\%$ while every other thread still makes progress. Because weights derive monotonically from profile priority, the familiar notion of "a higher-priority thread" is preserved—it simply means "a larger CPU share and an earlier place in the virtual-time queue," not "the exclusive right to run."

D. Deadline (Bandwidth) Scheduling

Proportional fairness is the wrong guarantee for an audio mixer that must produce a buffer every few milliseconds or a graphics pipeline that must finish before vsync. For these, Zircon offers *deadline scheduling*: a thread requests a reservation expressed as a triple (C, D, P)—*capacity*, *relative deadline*, and *period*, all durations with $C \le D \le P$. The contract is that the thread is guaranteed C nanoseconds of CPU within every P-nanosecond period, and that the scheduler will schedule it so those C units complete within D of the period's start. Deadline threads are dispatched **earliest-deadline-first (EDF)** and, as a class, take *absolute precedence over all fair threads*: a CPU runs its eligible deadline threads (ordered by absolute deadline) and only falls through to the fair run queue when no deadline thread is currently eligible. Because a reservation is a hard claim on bandwidth, the system applies *admission control*: a new or changed deadline reservation is accepted only if the CPU(s) it can run on can still satisfy the aggregate demand $\sum C_k/P_k \le 1$. If total deadline demand on a CPU were allowed to exceed 100%, deadline misses become unavoidable, so admission rejects infeasible reservations rather than silently degrading.

E. Profiles: the User-Visible Control

User space never sets weights or deadlines numerically through a thread handle. Instead it acquires a *profile*—a `ZX_OBJ_TYPE_PROFILE` kernel object describing a scheduling intent—and applies it. Profiles are minted by the privileged `ProfileProvider` service (over FIDL) from a role string, or directly with `zx_profile_create` from a profile resource; the resulting object encodes a `zx_profile_info_t` whose `flags` select one or more of: a base *priority* (`ZX_PROFILE_INFO_FLAG_PRIORITY`), a deadline reservation (`_FLAG_DEADLINE`, carrying `capacity`, `relative_deadline`, `period`), and/or a CPU *affinity* mask (`_FLAG_CPU_MASK`). A profile is applied with `zx_object_set_profile(handle, profile, 0)`; the target `handle` must be a thread (or VMAR, for memory priority) with `ZX_RIGHT_MANAGE_THREAD`, and the `profile` handle must carry `ZX_RIGHT_APPLY_PROFILE`. Centralizing scheduling parameters in profiles managed by a single service lets the platform tune roles globally without each component hard-coding numbers. In practice components do not even embed numbers: they request a named *role* (a string such as `fuchsia.media.audio.output`) and the `ProfileProvider` resolves it against a board-specific configuration that maps roles to concrete priorities, deadlines, and affinities—so the same binary gets appropriate timing on very different hardware. A profile may also carry a *memory* priority (`ZX_PROFILE_INFO_FLAG_MEMORY_PRIORITY`) applied to a VMAR, hinting that the region's pages should be kept resident rather than reclaimed, which is why `zx_object_set_profile` accepts VMAR as well as thread handles. 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);
```

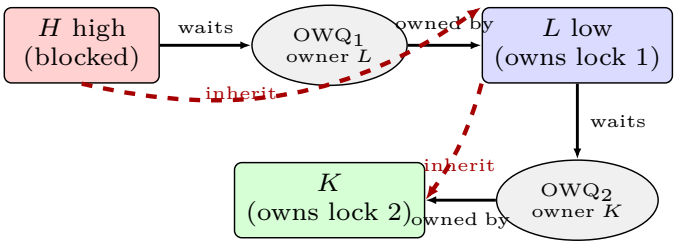


Fig. 11. Priority inheritance along a blocking chain. H blocks on an OwnedWaitQueue owned by L, which is itself blocked on a queue owned by K. H's inherited profile is propagated transitively to L and K so the actual lock holder is boosted to run and release.

F. Priority Inheritance and the OwnedWaitQueue

Both scheduling disciplines are vulnerable to *priority inversion*: a high-importance thread H blocks on a resource (a futex, a mutex) held by a low-importance thread L, and an unrelated medium thread M—runnable and heavier than L—preempts L, so H waits on M indirectly. 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 propagates the *inherited profile values* of the waiters to the owner, so the owner is scheduled *as if* it carried the importance of its most important waiter. The mechanism is general across both pillars: a fair waiter contributes its weight (boosting the owner's effective weight), and a deadline waiter can promote the owner into the deadline domain so the owner runs ahead of fair threads long enough to release the resource. Propagation follows the *blocking chain*: if L is itself blocked on a queue owned by K, the boost flows transitively from H through L to K, and unwinds as each lock is released. Owner information enters the kernel through the same APIs that block—most importantly the futex's `new_futex_owner` argument (see the synchronization section)—so user-space mutexes get inheritance for free. Figure 11 shows a two-hop chain.

G. Preemption and Cross-CPU Reschedule

A running thread is preempted when a more eligible thread becomes runnable or its time slice expires. The periodic timer tick is the backstop that bounds how long a thread runs before the scheduler reconsiders, but most preemptions are event-driven: a wake or a profile change that makes another thread the new minimum-f (or an eligible deadline thread) triggers a reschedule. Inside the kernel, short critical sections disable preemption (and, more strongly, take spinlocks with interrupts disabled) so the scheduler does not switch away while a per-CPU invariant is briefly broken; the reschedule is then taken at the next preemption point. When the newly-runnable thread belongs on a *different* CPU—because of affinity or load balancing—the local CPU sends an *inter-processor interrupt* (IPI) to ask the remote CPU to reschedule. Zircon distinguishes *eager* reschedules (send the IPI now, e.g. a freshly woken deadline thread that must preempt immediately) from *lazy* ones (defer and coalesce the IPI to the next natural scheduling point), trading a little latency for fewer cross-CPU interrupts.

H. Idle and Power

When a CPU has no runnable thread, it runs a per-CPU *idle thread*, which puts the core into an architectural idle/low-power state (e.g. `wfi` on ARM, `hlt`/MWAIT C-states on x86) until an interrupt—a timer, a device IRQ, or a reschedule IPI from another CPU—wakes it. Selecting how deeply to idle trades wake-up latency against power, and is coordinated with the timer subsystem so that a CPU with no near-term timer deadline can sleep more deeply. At a coarser grain, the kernel can *deactivate* a CPU entirely: when a CPU is taken offline (for power management or hotplug), the scheduler migrates its threads to the remaining active CPUs, refuses to admit deadline reservations that the survivors cannot satisfy, and reactivates the CPU on demand. The per-task runtime counters of §VI-H—`cpu_time` versus `queue_time`—are the diagnostic that reveals whether a latency problem is a thread starved of CPU share (rising queue time) or simply a long-running computation (rising CPU time), closing the loop between measurement and profile assignment. Table VI collects the scheduling syscalls and profile fields.

TABLE VI
SCHEDULING SYSTEM CALLS AND PROFILE FIELDS.

Syscall / field	Meaning
`zx_profile_create`	mint a profile object from a resource
`zx_object_set_profile`	apply profile to thread/VMAR
`zx_thread_legacy_yield`	(deprecated) set raw priority
	voluntarily yield the CPU
`zx_nanosleep`	block until a monotonic deadline
`zx_profile_info_t` (selected by `flags`)	
`priority`	fair weight source, $0 \ldots 31$
`deadline_params`	`{capacity, relative_deadline, period}` (all durations, $C \le D \le P$)
`cpu_affinity_mask`	allowed CPU set (hard)

VIII. Synchronization Primitives

Synchronization in Zircon is built around a principle visible throughout the kernel: stay out of the kernel on the fast path. The cornerstone user-visible primitive, the *futex*, makes the uncontended case a single atomic instruction in user space and traps into the kernel only when a thread must actually block or wake. On top of the futex, the C and C++ runtimes build mutexes, condition variables, and one-shot completions; alongside it the kernel uses its own internal locks—spinlocks, blocking mutexes, a big reader–writer lock, and seqlocks—to protect its data structures. This section covers all of these and ties futex ownership back to the priority-inheritance machinery of the scheduler [1], [12], [22].

A. The Futex

A *futex* ("fast userspace mutex") is not a kernel object you allocate; it is simply a 32-bit, naturally-aligned integer in ordinary user memory whose address the kernel uses as a key. The contract is split between user space and kernel. In user space, lock and unlock are implemented with an atomic compare-and-swap (CAS) on that integer; when there is no contention the operation succeeds and **the kernel is never entered**. Only when a thread finds the lock already held—or, on unlock, finds that waiters may exist—does it make a syscall. 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);
```

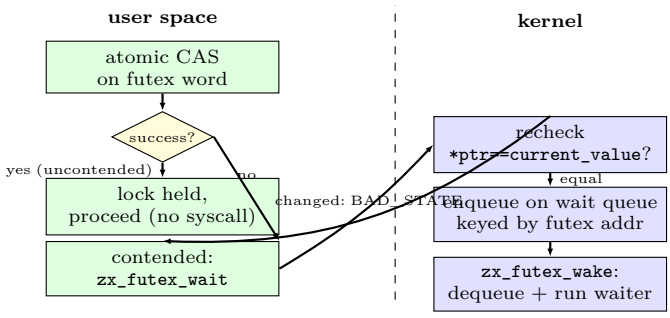


Fig. 12. Futex paths. Uncontended lock/unlock is a user-space CAS with no syscall. On contention the waiter traps into the kernel, which atomically rechecks the word against `current_value` before enqueuing on a wait queue keyed by the futex address; a later `zx_futex_wake` on release dequeues it.

```
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 subtle part is the wait. Between the moment a thread decides to block (having read the futex word and seen it "locked") and the moment it is actually enqueued in the kernel, the owner might release the lock and issue its wake—a wake that would be lost if the waiter enqueued afterward. The `current_value` argument closes this race: `zx_futex_wait` atomically verifies that `*value_ptr` still equals `current_value` and, in the same critical section, enqueues the caller. If the value has already changed, the call returns `ZX_ERR_BAD_STATE` immediately without blocking, and the caller retries its CAS. This compare-and-block is what makes the lock-free fast path safe. `zx_futex_wake` releases up to `wake_count` waiters on a key; `zx_futex_requeue` additionally moves up to `requeue_count` remaining waiters to a second futex without waking them—used to implement a condition variable's "move waiters from the condvar to the mutex" step and so avoid a *thundering herd*. Figure 12 contrasts 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` a `new_requeue_owner`; passing the owning thread's handle registers it with the kernel's wait queue (an `OwnedWaitQueue`, §VII-F). This is what lets the scheduler apply *priority inheritance* to user-space locks: when a high-importance thread blocks on a futex whose owner is a low-importance thread, the owner is boosted to the waiter's effective profile so it runs and releases the lock promptly, defeating priority inversion. Passing `ZX_HANDLE_INVALID` declares the futex temporarily unowned (appropriate for, e.g., a condition variable with no single owner). The companion call `zx_futex_get_owner` reads the current owner, and `zx_futex_wake_single_owner` wakes exactly one waiter and assigns it ownership—the common "hand the mutex to the next waiter" operation.

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 logical states (unlocked, locked-uncontended, locked-contended): `lock()` CASes unlocked→locked and, only on failure, transitions to contended and calls `zx_futex_wait` naming the holder as owner; `unlock()` CASes back and, only if the contended state was set, calls `zx_futex_wake_single_owner`. A `std::condition_variable` is a futex used for waiting plus `zx_futex_requeue` to migrate waiters from the condvar onto the associated mutex on `notify`, so that at most one thread is actually woken to contend for the mutex. Fuchsia's C `libsync` provides the same building blocks for the system's C code, including `sync_mutex_t`, `sync_condition_t`, and the one-shot `sync_completion_t`—a futex-backed latch on which many threads can `sync_completion_wait` until one `sync_completion_signal` releases them all, the idiom for "wait until initialization finished." Listing 9 makes the three-state mutex fast path concrete: the common acquire and release are a single atomic each, and the syscalls 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
void lock(atomic_int* m, zx_handle_t self) {
  int v = 0;
  if (atomic_compare_exchange_strong(m, &v, 1)) return; // uncontended
  do {                                                  // contended slow path
    if (v == 2 || atomic_exchange(m, 2) != 0)
      zx_futex_wait(m, 2, self, ZX_TIME_INFINITE);
    v = 0;
  } while (!atomic_compare_exchange_strong(m, &v, 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, futexes are irrelevant—there is no user memory to share and no fast path to protect. The kernel instead uses a small family of locks chosen by context. A *spinlock* protects very short critical sections; acquiring it disables interrupts (and preemption) on the local CPU and busy-waits, so it must never be held across a blocking operation, and is the only lock usable from interrupt context. The kernel `Mutex` is a blocking lock for longer sections: a contender that cannot acquire it enqueues on an `OwnedWaitQueue` and the holder inherits its profile (the same priority inheritance described above, applied internally). For data that is read far more often than written, the big reader–writer lock `BrwLock` allows many concurrent readers or one writer; it comes in a priority-inheriting variant (which boosts a blocked writer's would-be holders) and a plain blocking variant, and its accounting is split per-CPU so that uncontended readers rarely touch a shared cache line—the property that makes it cheap to guard hot, read-mostly structures such as the VM's mapping tables.

Seqlocks (sequence locks) protect data that readers must observe without ever blocking writers. A writer increments a sequence counter to an odd value, writes the payload, then increments it again to even; a reader snapshots the counter before and after reading and retries if the two snapshots differ or the first was odd (Listing 10). Readers never block writers and take no atomic on the data itself, which is exactly what is needed to publish the kernel's time/clock parameters into the read-only vDSO data page: user space reads them locklessly and computes the current time without a syscall.

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
```

TABLE VII
FUTEX AND RELATED SYNCHRONIZATION SYSTEM CALLS.

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

```
    atomic_thread_fence(memory_order_acquire);
  } while (seq & 1 || seq != atomic_load(&s->seq));
  return v;                              // consistent snapshot
}
```

Because a kernel that takes many locks can deadlock through inconsistent acquisition order, builds can enable *lockdep*, a runtime lock-ordering validator that records the order in which lock classes are acquired and flags any cycle before it can become a real deadlock in the field.

E. Events and Signals as Coarse Synchronization

Not all coordination needs a lock. Zircon's *event* and *eventpair* objects exist purely to carry *signals*: a thread (or the kernel) raises a user signal with `zx_object_signal` (or `zx_object_signal_peer` for the far end of an eventpair), and waiters observe it through `zx_object_wait_one`/`wait_async` on a port. This is coarser than a futex—there is no atomic value, no ownership, and every wait/raise enters the kernel—but it composes with the rest of the object system: the same wait primitives that watch a channel becoming readable or a task terminating also watch an event, so heterogeneous wakeups can be multiplexed on a single port. The full treatment of signals, ports, and asynchronous waiting belongs to the Signals and Ports section; here we note only that `ZX_EVENT_SIGNALED` on an event and the user-definable signal bits on an eventpair give a simple, kernel-mediated rendezvous when the futex's shared-memory model does not apply (for example, across two processes that share a handle but no memory). Table VII summarizes the synchronization calls.

IX. Virtual Memory: VMOs and VMARs

Zircon factors virtual memory into two orthogonal objects. A *Virtual Memory Object* (VMO) is a container of physical pages—the unit of memory itself, divorced from any address. A *Virtual Memory Address Region* (VMAR) is a span of a process's address space—the unit of *placement*, divorced from any backing content. Mapping is the explicit act that binds a range of a VMO into a VMAR at some virtual address with some permissions. This separation is the memory analogue of the handle/object split that pervades the rest of the kernel [1]: a VMO is named by a handle carrying rights, and the same VMO handle can be mapped into many address spaces or passed across a channel to share memory between processes. For an RTL engineer the mental model is clean: a VMO is a block of addressable storage (think a BRAM or a DRAM region), and a VMAR is the address decoder that places that storage into a particular process's bus—two different processes can decode the same storage at different addresses.

A. The VMO: a Resizable Container of Pages

A VMO is an ordered, page-granular array of bytes. All sizes and offsets are multiples of the hardware page size (4 KiB on the supported 64-bit architectures). Creating a VMO with `zx_vmo_create(size, options, &out)` reserves a logical size but commits *no* physical pages: a fresh anonymous VMO is conceptually an array of zeros, and frames are only drawn from the physical memory manager when a page is first touched or explicitly committed. This demand-commit discipline means a process can create a sparse multi-gigabyte VMO at negligible cost and populate it lazily.

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

- **Anonymous / paged** (the default from `zx_vmo_create`): zero-filled, demand-committed RAM. This 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 minted from a *resource* capability (§on hardware access) because it hands out raw physical memory, and its cache policy almost always needs adjusting (below).
- **Contiguous** (`zx_vmo_create_contiguous`): anonymous RAM that the PMM guarantees to be *physically contiguous*, allocated against a Bus Transaction Initiator (BTI) and aligned to a caller-chosen power of two. Contiguity is what DMA engines without scatter-gather (or with limited descriptor depth) require.
- **Pager-backed**: a VMO whose pages are supplied on demand by a user-space *pager* rather than zero-filled by the kernel. These are created with `zx_pager_create_vmo` and are the substrate for file-backed memory and executable loading; they are the subject of Section X.

1) VMO Operations: The core data path uses explicit copies rather than always mapping: `zx_vmo_read` and `zx_vmo_write` move bytes between a caller buffer and a VMO range without any mapping existing—convenient for control structures and small transfers, and 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 it but requires the handle to carry `ZX_RIGHT_RESIZE`, which is granted only if the VMO was created with the `ZX_VMO_RESIZABLE` option. Shrinking decommits and frees the truncated pages.

`zx_vmo_op_range(handle, op, offset, size, buffer, buffer_size)` is a multiplexed maintenance call. The operation selector `op` ranges over: `ZX_VMO_OP_COMMIT` (force-populate pages now, the eager opposite of demand paging), `ZX_VMO_OP_DECOMMIT` (release pages back to the PMM, leaving them logically zero), `ZX_VMO_OP_ZERO` (zero a range, decommitting where possible), the cache-maintenance operations `ZX_VMO_OP_CACHE_SYNC`, `_CACHE_INVALIDATE`, `_CACHE_CLEAN`, and `_CACHE_CLEAN_INVALIDATE` (which drive the architecture's cache-line clean/invalidate instructions, essential before or after DMA on non-coherent paths), the hint operations `ZX_VMO_OP_ALWAYS_NEED`, `_DONT_NEED`, and `_PREFETCH` (which bias the reclamation machinery), and the `ZX_VMO_OP_LOCK` / `_TRY_LOCK` / `_UNLOCK` family that pins the pages of a *discardable* VMO against reclamation.

`zx_vmo_set_cache_policy` selects how mappings of the VMO interact with 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), and `ZX_CACHE_POLICY_WRITE_COMBINING` (for write-mostly apertures such as framebuffers, where the CPU may coalesce stores). The policy may only be set while the VMO is unmapped and has no committed pages, and it is principally meaningful for physical VMOs.

2) Children and Copy-on-Write: `zx_vmo_create_child(parent, options, offset, size, &out)` derives a new VMO from a window of an existing one. The `options` word selects one of four relationships [6]:

- `ZX_VMO_CHILD_SNAPSHOT`: an immutable point-in-time *copy* of the parent's contents, realized lazily by copy-on-write (COW). Parent and child share identical physical pages until either side writes; the first write to a shared page allocates a private copy for the writer, so the snapshot never observes the parent's later mutations.
- `ZX_VMO_CHILD_SNAPSHOT_AT_LEAST_ON_WRITE`: the widely used "lazy COW" variant. The child shares the parent's pages and forks a private page on *write*; unlike a full snapshot it does not guarantee isolation from parent writes to pages neither side has yet forked. This is the cheap, common case—it is how the program loader hands each process a private, writable view of shared read-only ELF data.
- `ZX_VMO_CHILD_SLICE`: not a copy at all but an *alias*—a window into the parent that shares its pages directly, with writes visible in both directions. Slices let a subsystem hand out a bounded sub-VMO without duplicating storage.
- `ZX_VMO_CHILD_REFERENCE`: a second handle-bearing object that refers to the *entire* parent's pages (a full-size alias), useful when a distinct object identity is needed but the same backing must be shared.

Modifier bits compose with these modes: `ZX_VMO_CHILD_RESIZABLE` makes the child independently resizable, and `ZX_VMO_CHILD_NO_WRITE` produces a read-only child. Repeated cloning builds a *clone tree* (Fig. 13): a hidden parent holds the shared pages while leaves diverge one page at a time. The kernel tracks which leaf owns each physical page and reclaims a page to the shared parent only when the last private reference disappears. This single mechanism underlies `fork`-style sharing under Starnix, the loader's read-only text/data sharing across every process that runs a given binary, and cheap large-buffer snapshots in graphics and storage stacks.

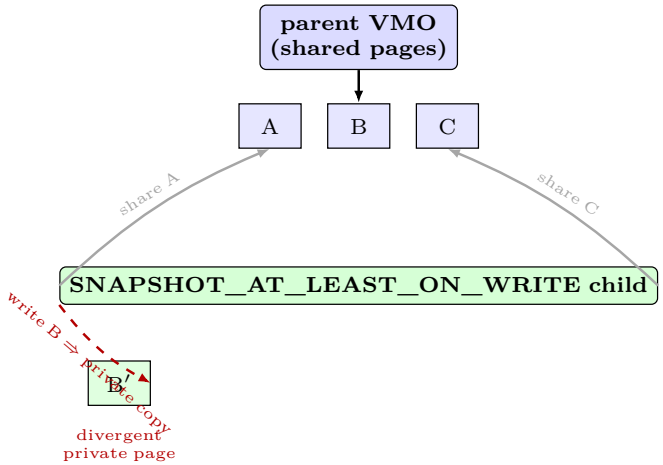


Fig. 13. A copy-on-write clone tree. The child shares the parent's pages A, B, C by reference. A write to B traps, allocates a private page B′ for the child, and rewires only that slot; A and C stay shared. Reads cost nothing; only written pages diverge.

B. The VMAR: the Address-Space Tree

Every process owns a *root VMAR* spanning its entire user address range, returned at process creation [7]. A VMAR is a tree: `zx_vmar_allocate` carves a child VMAR—a contiguous sub-range reserved for later use—out of a parent, and mappings are installed as leaves. Subdividing the space serves two ends: it groups related mappings under a single destroyable handle (tearing down a child VMAR atomically unmaps everything within it), and it provides the unit over which the kernel randomizes placement for address-space layout randomization (ASLR). When the caller does not pin an address, the kernel chooses a randomized offset within the parent for each child VMAR and mapping, so the layout differs from run to run and from process to process.

`zx_vmar_map(vmar, options, vmar_offset, vmo, vmo_offset, len, &addr)` is the operation that binds memory to an address. It maps `len` bytes of `vmo` starting at `vmo_offset` into `vmar`, returning the chosen virtual address. The `options` word carries the permission bits `ZX_VM_PERM_READ`, `ZX_VM_PERM_WRITE`, and `ZX_VM_PERM_EXECUTE`, plus placement and behavior modifiers: `ZX_VM_SPECIFIC` demands a specific offset within the VMAR (rather than a randomized one) and `ZX_VM_SPECIFIC_OVERWRITE` additionally replaces whatever was mapped there; `ZX_VM_MAP_RANGE` eagerly populates page-table entries for the whole range instead of faulting them in lazily; the `ZX_VM_ALIGN_*` bits request a larger alignment; and `ZX_VM_OFFSET_IS_UPPER_LIMIT` bounds placement from above. A VMAR created with the matching `ZX_VM_CAN_MAP_SPECIFIC` / `ZX_VM_CAN_MAP_READ` / `_WRITE` / `_EXECUTE` capabilities constrains what its descendants may do. `zx_vmar_protect` changes the permissions of an existing mapped range and `zx_vmar_unmap` tears one down; `zx_vmar_destroy` collapses a whole subtree.

The rights model gates every step. Mapping requires `ZX_RIGHT_MAP` on the VMO, and each requested permission must be matched by the corresponding VMO right: a `ZX_VM_PERM_WRITE` mapping needs `ZX_RIGHT_WRITE` on the VMO handle, and `ZX_VM_PERM_EXECUTE` needs `ZX_RIGHT_EXECUTE`. Because execute rights on a VMO are minted only from a dedicated executable-resource capability, $W \oplus X$ is enforced structurally: an ordinary writable VMO simply cannot acquire the execute right, so no mapping can be both writable and executable. Figure 14 shows the resulting tree and how one VMO mapped into two processes' VMARs yields shared memory.

C. Demand Paging and Faults

Mappings are lazy by default. `zx_vmar_map` merely records the intent—the page tables are populated on the first access unless `ZX_VM_MAP_RANGE` was requested. When the CPU touches an unmapped page within a valid mapping, it raises a page fault that the kernel resolves according to the backing VMO. For an anonymous VMO the kernel allocates a zero-filled frame from the PMM (or, for a read of a not-yet-written page, may transiently share a canonical zero page), wires it into the page tables, and resumes the faulting instruction. For a pager-backed VMO the kernel cannot manufacture the contents: it suspends

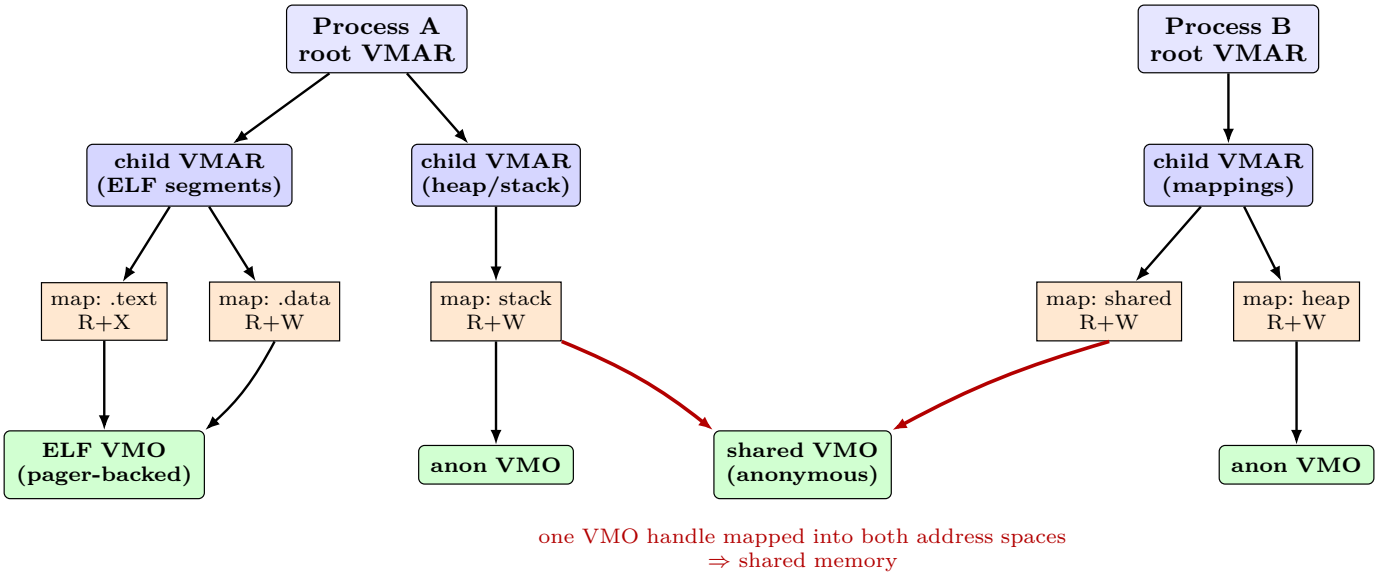


Fig. 14. The VMAR tree of two processes. Each root VMAR fans into child VMARs (grouping ELF segments, heap, stack) whose leaves are mappings, and each mapping binds a range of some VMO. The same anonymous VMO is mapped into both Process A and Process B (heavy edges): the two processes now share physical pages at possibly different virtual addresses—Zircon's shared memory.

the thread and dispatches a read request to the owning user-space pager, which fetches the data and supplies the page before the thread continues. This software-pager path—a page-miss handler implemented outside the kernel—is the centerpiece of Section X. An access that violates the mapping's permissions, or falls outside any mapping, is not satisfied at all; it is reflected to the process as an architectural exception through the exception channel.

D. Constructing a Process Address Space

The primitives above describe *what* memory objects are and how they map; this subsection shows *how* a concrete process address space is assembled from them, step by step. The striking fact is how little of this the kernel does: it provides an empty address space and a way to start a thread in it, and *everything else—loading code, laying out the heap, wiring in shared libraries—is ordinary user-space code calling the very syscalls of Section IX-B.*

1) Process Creation Yields an Empty Root VMAR: `zx_process_create(job, name, name_size, options, &proc, &root_vmar)` returns two handles: the process itself and a handle to its *root VMAR*, which spans the entire usable user portion of the address space—bounded below by a reserved guard region around the null address (so that a null-pointer dereference faults rather than hitting valid memory) and above by a region the kernel reserves. Critically, the new process is *empty*: not one page is mapped, and no code has been loaded. The kernel never reads an ELF file or chooses a layout; it only hands back the blank canvas and the root VMAR handle through which user space will paint on it.

2) Who Builds the Map: the Two-Phase Model: Because the address space starts empty, it must be populated by code running *outside* the new process. Fuchsia uses a two-phase bootstrap. In the first phase a *launcher* (the `fuchsia.process.Launcher` service, acting for whoever spawned the program) maps just enough to start: the *dynamic linker* (the ELF interpreter, part of libc) and a small bootstrap stack. It then transfers, over a *bootstrap channel*, a bundle of *processargs*—the handles and configuration the new process needs, including the executable's VMO, the loader-service channel, the root VMAR, the job, and the vDSO VMO—and calls `zx_process_start` with the dynamic linker's entry point. In the second phase the dynamic linker, now executing *inside* the new process, reads the processargs, finishes mapping the main executable and every shared library it depends on, applies relocations, and finally jumps to the program's `_start`. The split keeps the loading policy (which file, which libraries, where) entirely in replaceable user code.

3) Mapping the ELF Segments: The executable arrives as a VMO (served by the filesystem pager out of the on-disk image). The loader reads its ELF program headers and, for each `PT_LOAD` segment, materializes a mapping (Fig. 15). 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` (`|EXECUTE` for text), so the pages stay *shared* across every process running the same binary—the "shared text" and unified page cache of classical systems fall out of plain VMO sharing, at zero extra cost. The writable segment carrying `.data` is a copy-on-write child mapped `READ|WRITE`: it reads initialized values through shared pages but forks a private copy on the first write. The `.bss` tail beyond the file's content is zero-filled—an anonymous, demand-committed extension of that writable region. Two invariants hold throughout: no segment is ever both writable and executable ($W \oplus X$, enforceable as the job policy `ZX_POL_VMAR_WX`), and making *any* mapping executable requires the execute-gated VMO right that only an executable-resource capability can mint (§IX-B).

4) The Stack, the vDSO, and "Growing" Memory: The initial thread's **stack** is simply an anonymous VMO mapped `READ|WRITE`, placed adjacent to an unmapped **guard region**—a sub-VMAR deliberately left with a hole—so that a stack overflow faults instead of silently corrupting a neighboring mapping. Its top is passed as the stack pointer to `zx_process_start`. The **vDSO**, received as the `PA_VMO_VDSO` processargs handle, is mapped `READ|EXECUTE`; it is the *only* executable mapping the kernel itself insists every process carry, because it is the syscall gate through which all kernel entry flows (it is treated in the

syscall/vDSO section). There is deliberately *no* Unix `brk`: the **heap** is nothing but more VMO mappings. The C library's allocator (Scudo on Fuchsia) requests anonymous VMOs and maps them on demand, and `mmap`-style allocations are likewise VMO maps. Each shared library (`.so`) the dynamic linker pulls in—its VMO supplied by the *loader service* reached through the `PA_LDSVC_LOADER` handle—contributes another set of segment mappings exactly like the executable's. The unifying point: "growing a process's memory" always means "map another VMO," never a privileged kernel region.

5) ASLR, Sub-VMARs, and the Page Tables: The loader randomizes placement for address-space layout randomization, and it typically carves a child VMAR with `zx_vmar_allocate` per loaded module, grouping that module's segments into one aligned, contiguous range and then mapping them at fixed relative offsets within it (Fig. 15 shows two such module VMARs). This both enforces compact, guard-bracketed layout and lets an entire module be torn down by destroying one VMAR.

All of this—the VMAR tree, its child VMARs, and their mappings—is *bookkeeping*. The hardware realization is the per-architecture page table behind the address space (`ArchVmAspace`: an x86 PML4 hierarchy, ARM64 translation tables rooted at `TTBR0`, or RISC-V Sv39/Sv48 tables). The two are intentionally decoupled and the page table is populated *lazily*: `zx_vmar_map` records a mapping but installs no page-table entries, and each entry is filled only when a fault touches the page (or eagerly when the caller passes `ZX_VM_MAP_RANGE`). The VMAR layer is the policy and permission authority; the page table is its hardware shadow. When a mapping is removed or its permissions tightened, the kernel updates both the VMAR structure and the page table and then issues the TLB shootdown described in the kernel-memory section (§XI-E) so no stale translation survives on any CPU.

6) The Startup Sequence: Assembled end to end, launching a program is:

1) `zx_process_create(job, ...)` → a process and an empty root VMAR.
2) The launcher maps the dynamic linker and a bootstrap stack into the new root VMAR (using `zx_vmar_map` on the linker's VMO).
3) It creates the initial thread (`zx_thread_create`) and a bootstrap channel, and writes the processargs (handles + the executable and loader-service VMOs) into it.
4) `zx_process_start(proc, thread, entry, sp, bootstrap_handle, arg2)`—`entry` is the *dynamic linker's* entry point, `sp` the bootstrap stack top, and `bootstrap_handle` the channel end the new process will read processargs from.
5) Inside the process, the dynamic linker reads the processargs, maps the main executable's segments and each dependency (fetching library VMOs from the loader service), and applies relocations.
6) It maps the vDSO, sets up the real thread stack, and jumps to the program's `_start`.

The result (Fig. 15, Table VIII) is a VMAR tree of mappings over a handful of VMOs: copy-on-write, file-backed code and data shared with every other instance of the binary and its libraries; anonymous, private stack and heap; and the kernel-supplied vDSO—built entirely by user-space code through the syscalls of this section.

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	*unmapped* hole	—	loader (sub-VMAR)

Listing 11. Create a VMO, map it, write through the mapping, then t

```
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,
            /*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/VMAR pair, demand paging, COW children, and the rights checks together form the user-visible face of Zircon memory. Beneath it sits the physical memory manager that hands out the frames and the user-space pager that fills them—the subject of the next section.

X. Physical Memory and Demand Paging

Behind every VMO page lies a physical frame, and behind every frame lies the *physical memory manager* (PMM)—the kernel module that owns all of RAM, 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 much larger memory: page queues that rank frames for eviction, an in-RAM compressor, and—most importantly—the user-space pager, a page-miss handler implemented entirely outside the kernel. This section follows a frame through its life: allocation, classification, reclamation, and the demand-paging round trip that fills it.

A. The Physical Memory Manager

The PMM (`PmmNode` in `zircon/kernel/vm` [22]) carves physical RAM into one or more *arenas* (`PmmArena`), each describing

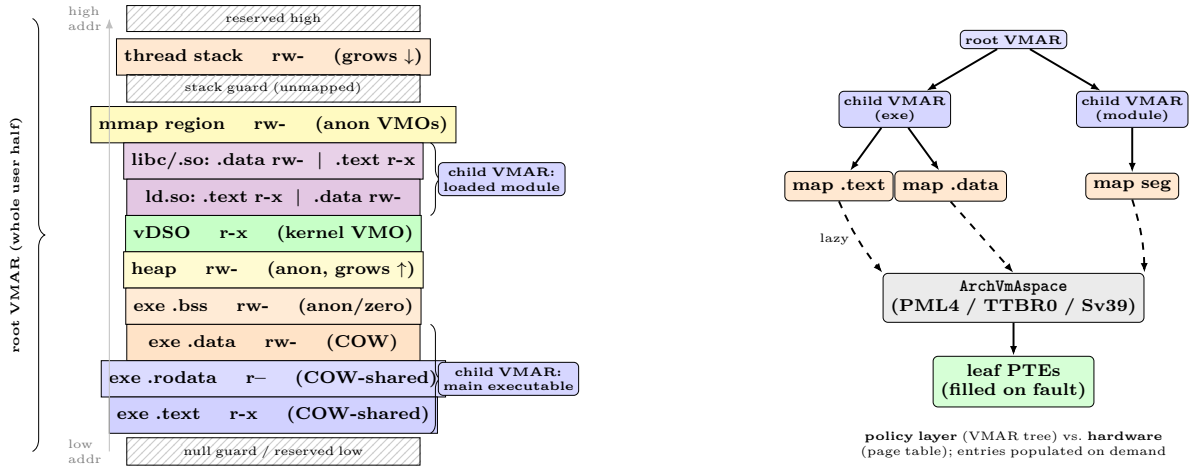


Fig. 15. Constructing a process address space. *Left*: a representative vertical memory map (high address at top). The root VMAR spans the whole user half; child VMARs group the main executable's segments and each loaded module. Read-only code/data are copy-on-write children of the file VMO (shared across all instances); stack, heap, `.bss`, and `mmap` are anonymous; the vDSO is the kernel-supplied execute-only mapping; guard regions bracket the stack and null. *Right*: the VMAR/mapping tree is the policy layer; the per-architecture `ArchVmAspace` page table (PML4 on x86, `TTBR0` tables on ARM64, Sv39/Sv48 on RISC-V) is its hardware realization, with leaf entries populated lazily at fault time.

a contiguous span of usable frames discovered at boot from the bootloader memory map. Every 4 KiB frame is tracked by a fixed-size bookkeeping record, `vm_page_t` (`struct vm_page`), holding the frame's physical address, an intrusive list node, a state byte, and a union of per-use "views"—an `object` view (back-pointer to the owning `VmCowPages`, the page's offset, pin count, and dirty state) when the frame backs a VMO, and a `zram` view when it holds compressed data. The set of states is small and exhaustive:

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 hang on the node's free list (`free_list_`); frames that have been temporarily loaned out of a contiguous VMO but are momentarily unused hang on a separate loaned free list (`free_loaned_list_`). Allocation and free are serialized by the node's lock—there is no per-CPU free cache in the current design; contention is instead reduced by allocating in batches and by keeping the loaned pool on its own list and lock. When the VM layer needs a page (to satisfy a fault, a commit, or a kernel-heap growth) it asks the PMM, which unlinks a frame, stamps its state, and returns it; on free the frame is zeroed lazily and relinked. `WIRED`, `HEAP`, `MMU`, and `SLAB` frames belong to the kernel itself (page tables, the heap, slab caches) and are never pageable; `OBJECT` frames back user VMOs and are the candidates for reclamation.

B. Page Queues and Aging

A frame backing a VMO is enrolled in one of the *page queues* (`PageQueues`), which classify it and, for reclaimable frames, rank it by recency of use. The classes include `Wired` (pinned, never reclaimed), `Anonymous` and `AnonymousZeroFork` (RAM with no backing store), `PagerBackedDirty` (modified file pages awaiting write-back), `HighPriority` (protected from reclamation), and a ring of *reclaim* queues `ReclaimBase` . . . `ReclaimLast`. That ring—eight buckets in the current source—is an LRU approximation: a background aging thread periodically rotates the "most-recently-used" generation forward, and each access to a page (recorded via the access bit and `MarkAccessed`) promotes it toward the MRU end. The newest bucket is never chosen as a victim, which keeps a working set from being evicted out from under a running thread. Two of the buckets are designated *active*; the rest are *inactive* and form the pool the reclaimer draws from 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 a single recency axis.

C. Reclamation: Evict, Compress, Discard

When free memory runs low a kernel scanner walks the inactive reclaim queues and recovers frames by one of three routes, chosen by the page's kind (Fig. 16):

- **Evict clean pager-backed pages.** A pager-backed page that is *clean*—identical to its backing store—can simply be dropped; its contents are re-fetchable from the pager on the next fault. This is the cheapest reclamation and the reason executables and file caches cost little under pressure: their pages are pure cache.
- **Compress anonymous pages.** Anonymous pages have no backing store, so they cannot be evicted—but they can be *compressed* in place. The `VmCompression` subsystem (RFC-0219, "Zircon Page Compression" [21]) passes the frame through a strategy—the default is `VmLz4Compressor`, an LZ4 codec—and packs several compressed 4 KiB pages into a single storage page, freeing the rest. The frame's state becomes `ZRAM` and its `zram` view records where the compressed bytes live; a later fault decompresses it back into a fresh frame. This is the zram-style "swap to RAM" tradeoff: spend CPU cycles and a fraction of RAM to avoid evicting data that has nowhere else to go.
- **Discard discardable VMOs.** A VMO created `DISCARDABLE` whose pages are not currently *locked* may have its entire contents thrown away at once; the next access faults the range back as zero and the owner learns, via the lock call's return, that it must regenerate

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

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

the data. Clients hold their data live by bracketing use with `zx_vmo_op_range(...ZX_VMO_OP_LOCK...)` and `...ZX_VMO_OP_UNLOCK...`; caches of regenerable assets (decoded images, font atlases) use this to donate memory back automatically.

The tradeoffs are explicit: eviction is free but only applies to clean, re-fetchable pages; compression preserves data at the cost of CPU and residual RAM; discarding is instantaneous but destroys data the owner must rebuild.

1) Zero-Page Deduplication and Page Borrowing: Two further mechanisms stretch the physical pool. The first is *zero-page deduplication*: because freshly committed anonymous pages are zero, a background scanner finds committed pages whose contents are still all-zero and collapses them back to the single canonical zero page, marking the slot copy-on-write so that a later write re-forks a private frame. The dedicated `AnonymousZeroFork` queue tracks pages that began life as a fork of the shared zero page and are thus prime candidates for being dropped again under pressure. The net effect is that sparse, lightly written VMOs cost almost nothing in committed frames.

The second is *page borrowing*. Contiguous VMOs reserve physically contiguous frames for DMA, but those frames are

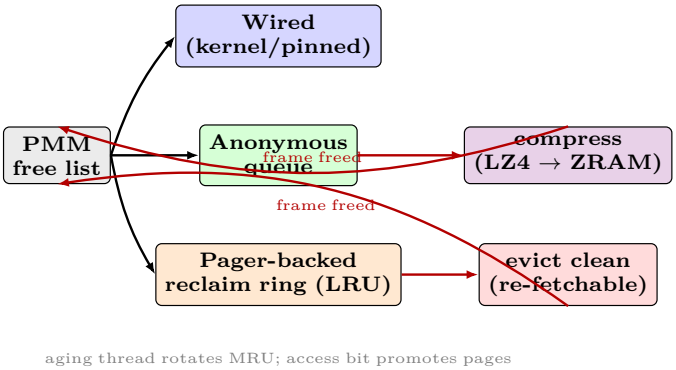


Fig. 16. Physical-page lifecycle. Frames leave the PMM free list as wired, anonymous, or pager-backed. Under pressure the reclaimer compresses anonymous pages (LZ4, state `ZRAM`) or evicts clean pager-backed pages (re-fetchable from the pager), returning frames to the free list. Wired frames are never reclaimed.

frequently idle between transfers. Rather than strand them, the PMM may *loan* a contiguous VMO's currently unused frames to the pager-backed cache—they appear on the separate loaned free list (`free_loaned_list_`) and can transiently hold evictable file pages. When the contiguous VMO needs a loaned frame back, the borrower's page is reclaimed (it was clean and re-fetchable) and the frame is returned to its owner. Loaning lets RAM that is committed for worst-case DMA still do useful caching work in the common case, at the cost of bookkeeping to reclaim a frame on demand.

D. The User-Space Pager

The pager is the most distinctive piece of Zircon's memory system. Rather than embedding filesystem and decompression logic in the kernel, Zircon lets a user-space process register as the supplier of a VMO's contents and act as a *software page-miss handler* [8]. The hardware analogy is exact: where a CPU's MMU, on a TLB miss, walks a page table and on a page-table miss raises a fault, Zircon on a VMO page miss raises a fault and—for pager-backed VMOs—"walks" out to a user process that fills the page, then retries the access. The kernel provides the trap and the resume; the policy lives in user space.

A pager is created with `zx_pager_create(0, &pager)` and bound to a *port* that will receive its requests. `zx_pager_create_vmo(pager, options, port, key, size, &vmo)` mints a pager-backed VMO; the `key` tags every request from this VMO so one port can serve many. When a thread faults on a non-resident page of that VMO, the kernel suspends the thread and enqueues a packet on the port:

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 dequeues the packet with `zx_port_wait`, sees `command == ZX_PAGER_VMO_READ` with an `[offset, length)` range, fetches that data from its backing store (disk, network, a decompressor) into a scratch "auxiliary" VMO, and installs it with `zx_pager_supply_pages(pager, pager_vmo, offset, length, aux_vmo, aux_offset)`. `supply_pages` *moves* the pages out of the auxiliary VMO into the pager-backed VMO at

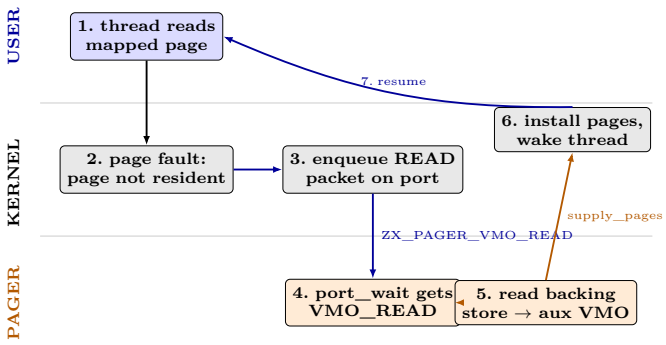


Fig. 17. Demand paging through the user-space pager. A read of a non-resident pager-backed page faults (2); the kernel enqueues a `ZX_PAGER_VMO_READ` packet on the pager's port (3); the pager fetches the data into an auxiliary VMO (4–5) and calls `zx_pager_supply_pages`; the kernel installs the pages and resumes the thread (6–7). The kernel supplies only the trap and the resume—the fill policy is user code.

the requested offset; the kernel then wakes the faulting thread, which re-executes the access and now hits a resident page. If the pager cannot satisfy the range it calls `zx_pager_op_range` with `ZX_PAGER_OP_FAIL`, carrying a `zx_status_t` in the `data` argument, and the faulting access is turned into an exception in the client. `ZX_PAGER_VMO_COMPLETE` is delivered (and `zx_pager_detach_vmo` called) when a VMO is being torn down, telling the pager no further requests will arrive. Figure 17 traces the whole round trip across the kernel/user-space boundary.

1) Dirty Tracking and Write-Back: A read-only file cache needs only `READ` requests, but a writable file-system—fxfs, for instance—must know which pages the client modified so it can write them back. Creating the VMO with the `ZX_VMO_TRAP_DIRTY` option turns each first write to a clean resident page into a trap: the kernel sends a `ZX_PAGER_VMO_DIRTY` request, and the pager either approves the clean→dirty transition with `zx_pager_op_range(...ZX_PAGER_OP_DIRTY...)` (reserving backing-store space first) or rejects it with `ZX_PAGER_OP_FAIL`. To flush, the pager discovers modified regions with `zx_pager_query_dirty_ranges`, brackets the write-out with `ZX_PAGER_OP_WRITEBACK_BEGIN` and `...WRITEBACK_END` (which returns the pages to the clean state, eligible again for cheap eviction), and can poll `zx_pager_query_vmo_stats` to learn whether a VMO was modified at all. This protocol gives user-space file systems the same dirty-page bookkeeping a monolithic kernel keeps internally—only here it is an explicit, auditable message exchange.

E. Memory-Pressure Signaling and OOM

Reclamation buys time; when it cannot keep up, the kernel asks user space to help by shedding memory. A privileged process obtains a pressure event with `zx_system_get_event(root_job, kind, &event)` for each level: `ZX_SYSTEM_EVENT_MEMORY_PRESSURE_NORMAL`, `_WARNING`, and `_CRITICAL`, plus `ZX_SYSTEM_EVENT_IMMINENT_OUT_OF_MEMORY` and `ZX_SYSTEM_EVENT_OUT_OF_MEMORY`. The kernel asserts `ZX_EVENT_SIGNALED` on exactly one pressure-level event at a time and escalates NORMAL → WARNING → CRITICAL as free memory falls. A user-space memory monitor waits on these and responds by dropping caches, unlocking discardable VMOs, trimming image and font caches, and asking the component framework to terminate low-priority work—recovering memory the kernel could not reclaim on its own. If pressure still cannot be relieved, the `IMMINENT_OUT_OF_MEMORY` event gives diagnostics a last chance to capture state before the kernel, on `OUT_OF_MEMORY`, performs a controlled out-of-memory reboot—a clean shutdown and restart in preference to unpredictable thrashing or a hard hang.

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

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

The PMM, the page queues, the compressor, and the pager together let Zircon run a rich user-space software stack—file systems, loaders, graphics—on a bounded physical pool, pushing every byte of policy out of the kernel while keeping the kernel firmly in control of the frames themselves. The next section turns inward to how the kernel manages *its own* memory and coordinates it across CPUs.

XI. Kernel Memory, Per-CPU State, and SMP

The preceding sections looked outward, at the memory the kernel hands to user space. This one looks inward: how the kernel maps and allocates its *own* memory, how it keeps fast per-CPU state, how it brings up and coordinates multiple cores, and how it keeps every CPU's view of the page tables coherent. These mechanisms are invisible to applications but decide the kernel's scalability and correctness 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 a low *user* half—a different mapping per process—and a high *kernel* half that is identical in every address space, so the kernel

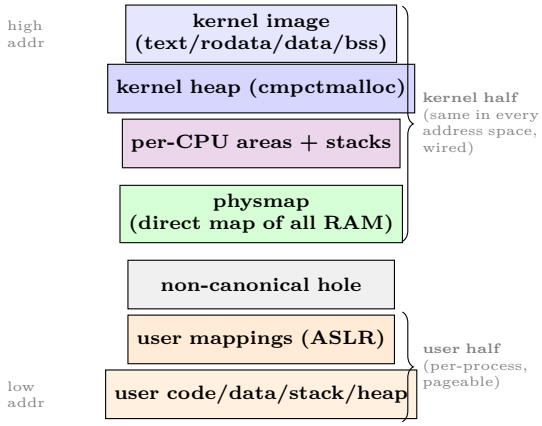


Fig. 18. Kernel address-space map. The high half—identical across all processes—holds the kernel image, the cmpctmalloc heap, per-CPU areas, and the physmap that linearly maps every physical frame for direct access; all of it is wired. The low half is the current process's pageable, ASLR-randomized user mappings.

sees the same view no matter which process is current. The kernel half is described by a single `VmAspace`, reachable as `VmAspace::kernel_aspace()`; its root VMAR holds the kernel's mappings.

The cornerstone of that half is the *physmap*: a linear, contiguous mapping of *all* usable physical RAM into the kernel's address space, so that any physical address has a trivially computable kernel virtual address and the kernel can touch any frame without first establishing a mapping. Physical `[0, gPhysmapSize)` maps to virtual `[gPhysmapBase, gPhysmapBase + gPhysmapSize)`, and the conversions are pure arithmetic: `paddr_to_physmap()` adds the base, `physmap_to_paddr()` subtracts it, and `is_physmap_addr()` tests membership. The physmap is why the PMM can return a frame and the heap can use it immediately—there is always a ready kernel pointer to physical memory. Alongside the physmap sit the kernel image itself (text, rodata, data, bss, loaded at `__executable_start` and mapped with appropriate permissions), the kernel heap region, the per-CPU areas, and assorted device and bookkeeping mappings. Kernel pages are *wired*: their frames carry states such as `WIRED`, `HEAP`, `MMU`, or `SLAB` and are never eligible for the reclamation machinery of Section X—the kernel must not page itself out. Figure 18 sketches the layout.

B. The Kernel Heap

Variable-size kernel allocations go through `cmpctmalloc`, a compact, space-tuned allocator (`zircon/kernel/lib/heap/cmpctmalloc`) exposing `cmpct_alloc`, `cmpct_free`, and `cmpct_memalign`. It is not a leaf allocator: when it runs short it calls `heap_grow()`, which pulls page-aligned chunks from the page layer (and ultimately the PMM) via `heap_page_alloc()`, and it returns whole pages with `heap_page_free()` while retaining one cached span to damp churn. The heap thus sits squarely on top of the physical memory manager.

Fixed-size, high-churn kernel objects are not served from the general heap, where they would fragment it and pay its metadata overhead. The clearest example is the *handle*: every open handle in the system is allocated from a dedicated arena, `HandleTableArena` (the global `gHandleTableArena`), whose `arena_.Alloc()` / `Free()` carve fixed-size slots from a preallocated region capped at `kMaxHandleCount` = 262144. An arena gives $O(1)$ allocation, tight packing, and a hard ceiling that bounds kernel memory regardless of user behavior. The dispatcher objects that handles point at are themselves heap-allocated (through `fbl::AllocChecker` placement `new`, wrapped in a `KernelHandle`), and the library also offers a general `fbl::SlabAllocator` template for subsystems that want slab-style fixed-size pools. Kernel memory is accounted and exposed to user space through `zx_object_get_info` with `ZX_INFO_KMEM_STATS` and `ZX_INFO_KMEM_STATS_EXTENDED`, drawing in part on the per-CPU page counters described next.

A third category 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 a thread also carries an auxiliary stack so that the compiler's safe-stack instrumentation can keep return addresses and spilled buffers apart. These stacks are wired anonymous allocations, never paged, and guard-mapped so that an overflow faults rather than silently corrupting an adjacent allocation. Because a kernel stack is committed for the life of a blocked thread, the kernel's per-thread memory footprint—stack plus the thread and dispatcher bookkeeping—is a first-order scalability concern, which is one reason Zircon favors asynchronous, port-driven designs over armies of blocked threads.

C. Per-CPU State

Hot kernel state that is logically "one per core" lives in an array of `struct percpu`, one element per CPU [22]. Each holds that CPU's `Scheduler` (its run queues), `TimerQueue`, deferred-procedure-call runner (`DpcRunner`), CPU and guest statistics, per-CPU page-state counters (`vm_page_counts`), idle-power and resume state, lock-up detector timer, and memory-stall accumulator. Keeping this state per-CPU means the scheduler tick, the timer expiry, and the statistics update touch only the local element and need no cross-CPU lock on the common path.

The trick that makes this cheap is fetching the *current* CPU's element without a lock or a CPU-number lookup, by holding a pointer to it in a register the kernel reserves for the purpose. On x86 that pointer is the `GS` base, so `x86_get_percpu()` reads it directly through `gs:`; a standing kernel invariant requires `gs_base` to point at the current core's `x86_percpu` whenever the kernel runs with interrupts enabled. On ARM64 the pointer lives in general-purpose register `x20`, read by `arm64_read_percpu_ptr()`. Both are wrapped by the portable `percpu::GetCurrent()` (atop `arch_get_curr_percpu()`), with `percpu::Get(cpu_num)` for addressing an arbitrary core. Because the register is updated only at context boundaries the kernel controls, reads are a single instruction with no synchronization.

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

```
// 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*, which initializes the core subsystems—memory, the heap, the object layer, the scheduler—and then starts the secondary cores one at a time (`mp_init()` and the per-arch hotplug path `arch_mp_cpu_hotplug()`). Each secondary begins in a tiny architecture-specific assembly *trampoline* (ARM64's `start.S`/`secondary.S`; x86's real-mode bootstrap brought up by `x86_bringup_aps()`) that establishes a stack and the kernel page tables, then jumps into C++. On ARM64 the entry is `arm64_secondary_entry()`, which runs early per-CPU initialization (`arm64_init_percpu_early()`, `arch_mp_init_percpu()`), drives the secondary-CPU init levels, and finally calls the common `lk_secondary_cpu_entry()` to hand the core to the scheduler; x86's `x86_init_percpu()` reaches the same hand-off. From that point the core is an equal scheduling citizen, pulling threads from its own run queue. Cores can later be removed and re-added through `mp_unplug_cpu_mask()` and `mp_hotplug_cpu_mask()` for power management.

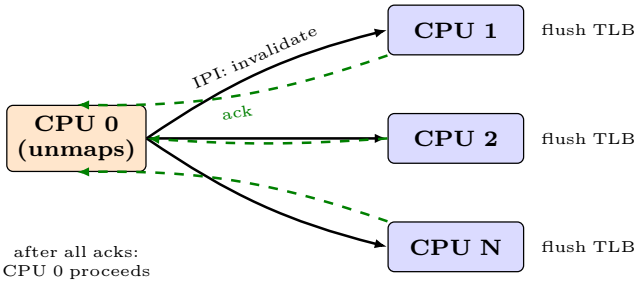
Coordination between running cores uses *inter-processor interrupts* (IPIs). The kind is a scoped enum `mp_ipi` with four members: `RESCHEDULE` (poke a core to re-run its scheduler, e.g. after making a thread on it runnable), `GENERIC` (run an arbitrary function on target cores, the vehicle for `mp_sync_exec`), `INTERRUPT` (deliver a device interrupt to a chosen core), and `HALT` (stop a core, used on panic and shutdown). A target is expressed as an `mp_ipi_target` (`MASK` of CPUs, `ALL`, or `ALL_BUT_LOCAL`) lowered to the hardware by `arch_mp_send_ipi()`.

E. TLB Management and Shootdown

Each core's MMU caches recently used page-table entries in a translation look-aside buffer (TLB). When the kernel unmaps a page or tightens its permissions (`zx_vmar_unmap`, `zx_vmar_protect`), any TLB on any core that may have cached the old translation now holds a stale, dangerous entry. Making them consistent is the *TLB shootdown*.

On x86 the shootdown is performed in software. The unmapping core records the pending invalidation (`PendingTlbInvalidation`) and calls `mp_sync_exec` with a `GENERIC` IPI to every core that might share the mapping; each target runs an invalidation task that executes the local `invlpg`/`INVPCID` and acknowledges, and only when all have acked does the originator proceed (Fig. 19). To avoid flushing the entire TLB on every address-space switch, x86 uses *process context identifiers* (PCIDs) when available (gated by `g_x86_feature_pcid_enabled`): each address space gets a PCID so its entries can coexist in the TLB across switches, and `INVPCID` variants invalidate by PCID rather than wholesale; flushes for idle or inactive cores are deferred to their next context switch.

ARM64 largely sidesteps the IPI. Its TLB-invalidate instructions have an *inner-shareable* broadcast form—`vae1is`, `vale1is`, `vmalle1is`, and the ASID-scoped `aside1is`—that the hardware propagates to every core in the shareability domain, so one core issuing the invalidation flushes all of them without a software round trip. ARM64 also tags translations with *address space identifiers* (ASIDs), allocated by an `AsidAllocator`, with a reserved global ASID (`MMU_ARM64_GLOBAL_ASID`) for the always-mapped kernel; ASIDs play the role PCIDs play on x86, letting per-process entries persist across context switches and bounding how often a full flush is needed. The two architectures thus reach the same goal—no stale translations—by opposite means: x86 by software IPI, ARM64 by hardware broadcast.



x86 path: software shootdown via `mp_sync_exec`.
ARM64: hardware inner-shareable broadcast TLBI—no IPI.

Fig. 19. TLB shootdown (x86 software model). The unmapping core sends a `GENERIC` IPI to every core that may cache the translation; each invalidates its TLB and acknowledges; only after all acks does CPU 0 reuse the freed frame. ARM64 achieves the same effect with a single broadcast invalidate instruction and no IPI.

F. Memory Ordering and Lockless Publication

On SMP hardware correctness hinges on memory ordering. The kernel expresses shared state with explicit atomics (`ktl::atomic`) and the architecture's barrier instructions rather than relying on incidental ordering, and uses fine-grained and lock-free structures on its hottest paths.

A recurring pattern is the *seqlock*: a writer bumps a sequence counter before and after a multi-word update, and readers retry if the counter changed or is odd, so a reader never needs to acquire a lock and writers are never blocked by readers. Zircon provides this as `concurrent::SeqLock` (with a kernel-side `lib/kconcurrent` variant). Its signature use is publishing time data to user space without a syscall: the kernel writes a shared, read-only `TimeValues` structure—`version`, `ticks_per_second`, the monotonic and boot tick offsets (`mono_ticks_offset`, `boot_ticks_offset`), and the ticks-to-time ratio (`ticks_to_time_numerator` / `_denominator`)—via `SetTimeValues()` into a VMO that the vDSO maps read-only. User code reads those fields through the `fasttime` library and computes the current monotonic or boot time entirely in user space, retrying under the seqlock if an update (such as a suspend/resume offset change) lands mid-read. The result is a clock read with no kernel entry and no lock—the SMP-safe culmination of the memory machinery this cluster has described.

XII. Inter-Process Communication: Channels

If the object/handle model is the noun system of Zircon, *channels* are its primary verb. Almost every cross-process interaction in Fuchsia—opening a file, drawing a frame, querying a driver, starting a component—is ultimately a sequence of messages on a channel. A channel is the only primitive that moves *capabilities* (handles) between processes, which makes it the structural backbone of the whole capability system [1], [9].

A channel is a **bidirectional, ordered, datagram** message pipe with exactly two *peered* endpoints. The two endpoints are equal: there is no client/server asymmetry at the kernel level: that distinction is a userspace convention. `zx_channel_create` returns the two endpoint handles atomically; there is no "connect" step, because a channel exists in full from the moment it is created. To hand one end to another process, a parent writes that endpoint handle *into a message on another channel*—which is exactly the capability-transfer mechanism described

TABLE XI
KERNEL-MEMORY AND SMP CONCEPTS.

Concept	One-line description
physmap	Linear kernel mapping of all RAM; `paddr_to_physmap()` 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 `percpu` 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`.

below. The bootstrap channel delivered to a new process by `userboot` is the root from which all other connectivity grows.

A. Messages: Bytes and Handles

A channel carries discrete *messages*, never a byte stream. Each message is an atomic unit consisting of 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`).

Atomicity is the defining property: a reader either receives an entire message or nothing. There is no partial read and no message coalescing; message boundaries are preserved exactly, and within one endpoint ordering is strictly FIFO. This is the sharpest distinction from a socket (Section XIII), which is a flow-controlled byte stream with no message framing.

B. The Capability Transfer

The handle array is what makes a channel special. When `zx_channel_write` succeeds, the listed handles are *removed* from the sender's handle table and *inserted* into the peer's handle table when it reads the message (Fig. 20). The integer handle values are not meaningful across process boundaries; the kernel rebinds each transferred handle to a fresh value in the receiver, preserving the underlying object reference and its rights. The instant after a successful write, the sender's handle values are invalid—move semantics, not copy. A process therefore cannot "leak" a handle by writing it and continuing to use it; the capability has genuinely departed.

C. Reading and Writing

`zx_channel_write(handle, options, bytes, num_bytes, handles, num_handles)` enqueues one message on the peer. Crucially, a channel write *never blocks*: there is no kernel-level flow control. If the peer is alive the message is queued

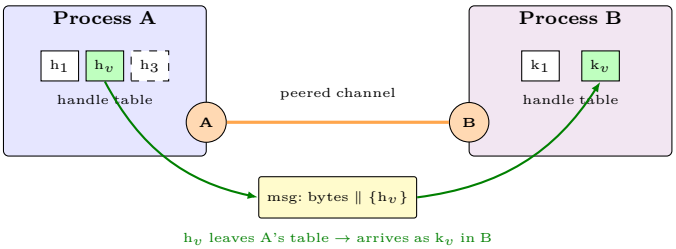


Fig. 20. Capability transfer on a channel. The message carries inline bytes plus an inline handle h_v. On `zx_channel_write` the handle is removed from process A's table; on `zx_channel_read` it reappears in process B's table under a new local value k_v, with the same object identity and (possibly reduced) rights.

unconditionally and the call returns immediately; if the peer is closed it fails with `ZX_ERR_PEER_CLOSED`. Either way the handles are consumed by a successful write.

`zx_channel_read` is the dual. The caller must present a buffer for both the bytes and the handles. If either buffer is too small the read fails with `ZX_ERR_BUFFER_TOO_SMALL` and returns the *required* sizes through the `actual_bytes`/`actual_handles` out-parameters, leaving the message intact at the head of the queue (unless `ZX_CHANNEL_READ_MAY_DISCARD` was requested). The idiomatic pattern is therefore a sized read, or a probe-then-read using the returned sizes. Because messages are atomic, there is never a torn or half-consumed message.

D. Rights Reduction in Flight: *write_etc*

The plain write/read transfers handles verbatim. The `zx_channel_write_etc`/`zx_channel_read_etc` variants attach a per-handle *disposition* so a sender can transform each capability as it crosses the boundary. The write side supplies an array of `zx_handle_disposition_t`:

- `operation` — `ZX_HANDLE_OP_MOVE` (consume the source) or `ZX_HANDLE_OP_DUPLICATE` (keep the source, send a copy);
- `handle` — the source handle;
- `type` — an expected `zx_obj_type_t`, or `ZX_OBJ_TYPE_NONE` to skip the check;
- `rights` — the rights the receiver should get, allowing *reduction* (never elevation), or `ZX_RIGHT_SAME_RIGHTS`;
- `result` — a per-handle status filled in by the kernel.

The read side receives an array of `zx_handle_info_t` reporting each arriving handle's object type and rights. This lets a server hand out a read-only view of a VMO, or assert "this must be a channel," in the same atomic operation that delivers the bytes—no extra `zx_handle_duplicate` / `zx_handle_replace` round trips.

E. The Synchronous Call: *zx_channel_call*

A naive request/reply built from separate write and read calls has a race: after writing a request, a client must read a reply, but an unrelated message may arrive first, and two outstanding requests may have their replies interleaved. `zx_channel_call` closes this race by performing the write and the matching read as one atomic, correlated operation (Fig. 21).

The correlation key is a **transaction id** (`zx_txid_t`) occupying the *first four bytes* of the message. The kernel overwrites that field in the outgoing message with a freshly generated txid in the range `0x80000000`–`0xFFFFFFFF`, guaranteed not to collide with any other in-flight `zx_channel_call` on the same

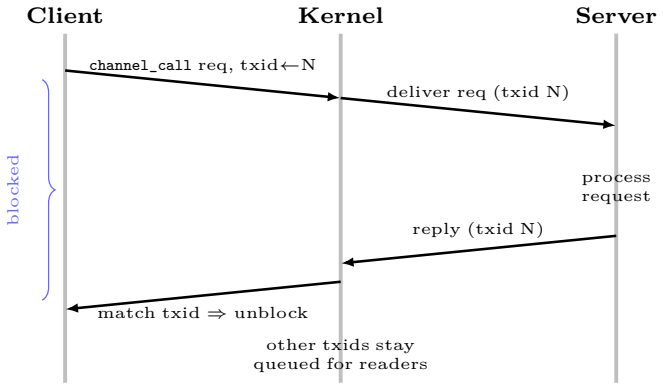


Fig. 21. The `zx_channel_call` transaction. The kernel stamps the request with txid N, blocks the caller, and routes back exactly the reply whose header echoes N. Unrelated messages remain in the queue for normal `zx_channel_read` consumers, eliminating the write/read reply race.

endpoint. It then blocks the caller until a message arrives whose first four bytes carry the *same* txid, which it delivers as the reply. Messages bearing other txids are left queued for ordinary readers. The deadline is an absolute monotonic `zx_instant_mono_t`; on `ZX_ERR_TIMED_OUT` the call is no longer waiting but the request has already been sent.

F. Where FIDL Lives—and Where It Does Not

It is essential to separate the kernel primitive from the protocol layer. The **kernel knows nothing about FIDL.** It moves opaque bytes and handles; it never parses a payload. FIDL (the Fuchsia Interface Definition Language) is a *userspace* wire format layered on top. A FIDL method call is just a channel message whose bytes begin with a transaction header—a `zx_txid_t`, a flags/magic field, and a 64-bit method *ordinal*—followed by the encoded arguments, with any `handle`/`client_end`/`server_end` fields traveling in the message's handle array. Language bindings (C++, Rust, Go, Dart) generated by `fidlc` do the encode/decode. A two-way FIDL method is precisely a `zx_channel_call`: the binding fills in the ordinal, lets the kernel assign the txid, and blocks for the correlated reply. One-way methods and events are plain writes. This clean split—kernel as a dumb, fast capability mover; FIDL as the typed protocol—is why the same channel carries a filesystem protocol, a graphics protocol, and a driver protocol with no kernel changes.

G. Signals, Closure, and Epitaphs

A channel endpoint exposes three primary signals for use with the waiting machinery of Section XIV: `ZX_CHANNEL_READABLE` (at least one message is queued), `ZX_CHANNEL_WRITABLE` (asserted while the peer is open), and `ZX_CHANNEL_PEER_CLOSED` (the other endpoint and all its handles are gone). `PEER_CLOSED` is sticky and is the universal end-of-stream indicator; note that any messages already queued remain readable even after the peer closes, so a correct reader drains to empty *and* sees `PEER_CLOSED` before concluding the channel is finished.

The FIDL *epitaph* is a userspace convention built on this: before closing a protocol, a server may send one final message carrying a `zx_status_t` reason, so the client learns *why* the connection ended rather than merely *that* it did. The kernel has no notion of an epitaph; it is simply the last message written before the handle is dropped, after which the client observes `ZX_CHANNEL_PEER_CLOSED`.

TABLE XII
Channel syscalls.

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

H. Limits and Flow Control

Because writes never block, the kernel does not throttle a fast producer: there is no built-in backpressure. The only hard limits are the per-message byte and handle caps above, plus an upper bound on the number and total size of messages a channel will buffer before the write is treated as an error condition. Higher level flow control—bounding outstanding requests, applying credit schemes—is deliberately a FIDL/userspace concern, typically expressed with hanging-get patterns or explicit acknowledgement protocols. The kernel's job ends at moving bytes and handles reliably and in order.

Listing 15. Create a channel, send a VMO handle, and issue a s

```
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 are the right tool for typed, capability-bearing protocols, but they are the wrong tool for a bulk byte pipe, for a tiny fixed-format command queue, or for a file-like cursor. Zircon supplies four more transport objects, each tuned to a shape of data movement that a channel handles poorly. Table XIII contrasts all five; this section explains the four new ones [2], [10].

A. Sockets: Flow-Controlled Byte Streams

A **socket** is a peered, bidirectional *byte* transport backed by an internal kernel buffer (Fig. 22, left). Unlike a channel, which never blocks and never throttles, a socket has genuine **flow control**: each endpoint has a bounded buffer, and once

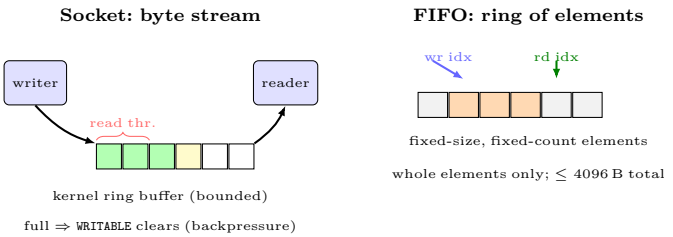


Fig. 22. Left: a socket as a bounded kernel ring with read/write thresholds; a full buffer de-asserts `ZX_SOCKET_WRITABLE`, applying backpressure. Right: a FIFO as a fixed ring of fixed-size elements with producer/consumer indices, moved a whole element at a time.

it fills, `ZX_SOCKET_WRITABLE` de-asserts so a waiter is told to pause. This makes sockets the natural substrate for stdio and for byte-stream transports where a slow consumer must be able to push back on a fast producer.

`zx_socket_create(options, &a, &b)` returns a peered pair. The single option chooses the framing discipline: `ZX_SOCKET_STREAM` (the default, a pure byte stream where writes coalesce and reads may return any prefix) or `ZX_SOCKET_DATAGRAM` (each write is a preserved record). Data moves with `zx_socket_read`/`zx_socket_write`, which—unlike a channel—may transfer a *partial* amount and report the actual count. The crucial limitation: **sockets cannot carry handles.** They move bytes only; if you need to pass a capability, you need a channel.

The signal set is richer than a channel's because of flow control and half-close: `ZX_SOCKET_READABLE`, `ZX_SOCKET_WRITABLE`, `ZX_SOCKET_PEER_CLOSED`, the half-close pair `ZX_SOCKET_WRITE_DISABLED` / `ZX_SOCKET_PEER_WRITE_DISABLED`, and the user-tunable `ZX_SOCKET_READ_THRESHOLD` / `ZX_SOCKET_WRITE_THRESHOLD`. The thresholds, set via `zx_object_set_property`, let a reader wait for "at least N bytes buffered" or a writer for "at least N bytes of space," avoiding a wake-up storm of tiny readable events.

Half-close is expressed with `zx_socket_set_disposition(handle, disposition, disposition_peer)`. Passing `ZX_SOCKET_DISPOSITION_WRITE_DISABLED` shuts down the write direction of an endpoint: further writes fail with `ZX_ERR_BAD_STATE`, the peer drains any buffered bytes and then observes `ZX_SOCKET_PEER_WRITE_DISABLED`—a clean "no more data this way" that a bidirectional byte protocol (such as a TCP-like shutdown) needs. `ZX_SOCKET_DISPOSITION_WRITE_ENABLED` can re-enable it, but only while the buffer is empty.

B. FIFOs: Low-Overhead Control Queues

A **FIFO** is a deliberately minimal primitive: a fixed-size ring of *fixed-size elements* shared between two peered endpoints, optimized for the lowest possible per-message overhead (Fig. 22, right). `zx_fifo_create(elem_count, elem_size, options, &a, &b)` fixes the geometry at creation; `elem_count` must be a power of two and the total `elem_count` × `elem_size` may not exceed 4096 bytes, so the entire ring lives in a single page. `zx_fifo_write` and `zx_fifo_read` move *whole elements only*—you cannot read half an element—and report how many elements were transferred, with the same `ZX_FIFO_READABLE` / `ZX_FIFO_WRITABLE` / `ZX_FIFO_PEER_CLOSED` signals.

The intended use is a *control plane* riding alongside a shared-memory data plane. Block-device and network (ethernet/net-stack) drivers pair a FIFO with a VMO: bulk payload lives in the VMO, while the FIFO carries tiny fixed request/response descriptors ("transmit the buffer at offset X, length Y; here is completion status Z"). Because the ring is one page and the elements are fixed, the hot path is a couple of memory copies and an index bump—far cheaper than a channel message—which is exactly what a high-rate driver queue needs.

C. Streams: A Cursor over a VMO

A **stream** grafts POSIX file semantics onto a VMO. Where a VMO is a flat, randomly addressable memory object, a `zx_stream` adds a *seek offset*—a cursor—so that sequential `read`/`write`/`lseek` work as a C programmer expects. `zx_stream_create(options, vmo, seek, &out)` binds a stream to an existing VMO at an initial offset; the `options` select `ZX_STREAM_MODE_READ`, `ZX_STREAM_MODE_WRITE`, and optionally `ZX_STREAM_MODE_APPEND`, each gated by the corresponding right on the underlying VMO. Data moves with the scatter/gather calls `zx_stream_readv` and `zx_stream_writev` over arrays of `zx_iovec_t`, and `zx_stream_seek` repositions the cursor (set/current/end).

Two behaviors make streams the file primitive of Fuchsia. First, a write past the current end of the VMO *grows* the VMO's content size automatically, so a file expands as it is written without an explicit resize. Second, when the VMO is *pager-backed* (the file-contents case of the virtual-memory subsystem), reads through the stream fault pages in lazily from the userspace pager. A filesystem therefore exposes a file as a pager-backed VMO, hands clients a stream over it, and gets demand-paged file I/O with a familiar read/write/lseek surface and no extra copy—the page cache *is* the VMO.

D. Eventpairs: Liveness and Cancellation Tokens

An **eventpair** is the smallest peered object: two endpoints sharing nothing but signal state. `zx_eventpair_create(0, &a, &b)` returns the pair. Either side can assert user-defined bits on the *other* end with `zx_object_signal_peer`—`ZX_EVENTPAIR_SIGNALED` or any of `ZX_USER_SIGNAL_0`..`7`—and, decisively, when all handles to one end are closed the kernel automatically asserts `ZX_EVENTPAIR_PEER_CLOSED` on the surviving end.

That automatic peer-closed edge makes an eventpair an ideal *liveness* or *cancellation* token. A holder waits on `ZX_EVENTPAIR_PEER_CLOSED`; the moment the counterpart process dies or drops its end—for any reason, including a crash—the waiter wakes. No heartbeat protocol, no polling. The graphics stack uses exactly this: Flatland *view tokens* are eventpair ends exchanged between a parent and child view, so each side is notified immediately if the other tears down its half of the scene graph, and the corresponding view node can be reaped. The same pattern underlies countless "cancel this work when the requester goes away" relationships throughout the system.

The throughline: a channel is the only handle-carrying transport and the only one with strict message framing and no flow control; sockets and FIFOs are the byte/element movers that *do* apply backpressure; a stream is file I/O over a VMO; and an eventpair is pure cross-process signaling. Choosing the right object is mostly a matter of asking what the unit of transfer is and whether capabilities must ride along.

XIV. Signals, Waiting, and Ports

The previous two sections kept referring to "signals" and "waiting" without defining them. That machinery is the subject

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 / datagram	—	yes	stdio, byte streams
FIFO	fixed element	—	yes	driver control queues
Stream	cursor over VMO	—	n/a	file read-/write/seek
Eventpair	1-bit signals	—	n/a	liveness / cancel tokens

here, and it is one of the most important unifying ideas in Zircon: *every* waitable kernel object—channel, socket, FIFO, eventpair, timer, VMO, process, thread—advertises its state through the same 32-bit signals word and is waited upon through the same handful of syscalls. Master this once and you can wait on anything [2], [11].

A. Object Signals

Each waitable object carries a `zx_signals_t`: a 32-bit word in which each bit is a boolean condition. Some bits are *object-defined*—their meaning depends on the object type—and a few are reserved for user code. Representative object-defined bits include `ZX_CHANNEL_READABLE`, `ZX_SOCKET_WRITABLE`, `ZX_VMO_ZERO_CHILDREN`, `ZX_TASK_TERMINATED` (asserted when a process or thread exits), and `ZX_TIMER_SIGNALED`. The bits `ZX_USER_SIGNAL_0` through `ZX_USER_SIGNAL_7` carry no kernel meaning and are freely assertable by userspace via `zx_object_signal` (on the object itself) or `zx_object_signal_peer` (on a peered object's far end), the mechanism an eventpair uses.

The single most important property is that signals are **level-triggered**: a bit is asserted for exactly as long as its condition holds. `ZX_CHANNEL_READABLE` is high while a message waits and goes low when the queue drains; it is not a one-shot "message arrived" event. This level discipline is what makes re-arming waits race-free—if you ask again and the condition is still true, you are told so immediately, so there is no window in which an edge can be missed.

B. Synchronous Waiting

The direct way to block is `zx_object_wait_one(handle, signals, deadline, &observed)`: sleep until *any* bit in `signals` is asserted on `handle`, or until the absolute `deadline` passes, returning the full `observed` signal set at wake time. To wait on several objects at once, `zx_object_wait_many` takes an array of `zx_wait_item_t` (each a handle plus a per-handle `waitfor` mask), up to a fixed limit, and reports per-item which signals were observed. Both require `ZX_RIGHT_WAIT` on every handle involved.

Synchronous waits are simple and perfect for a thread dedicated to one or two objects, but they scale poorly: a thread blocked in `wait_one` can do nothing else, and `wait_many` caps out at a small number of objects and must be torn down and rebuilt whenever the set changes. A server juggling thousands of connections needs something else.

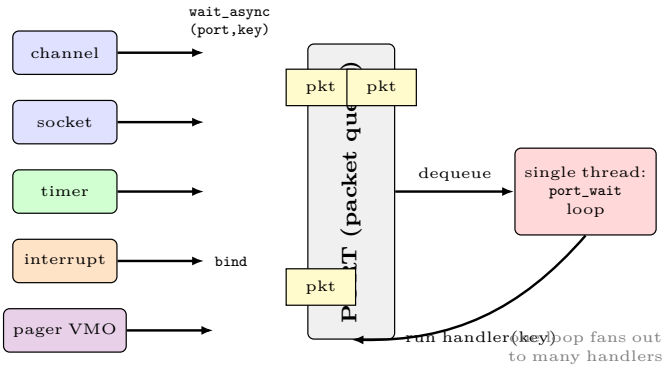


Fig. 23. A port-based reactor. Many heterogeneous objects register interest with `zx_object_wait_async` (or `zx_interrupt_bind`); each ready condition enqueues a keyed packet; a single thread drains the port with `zx_port_wait` and dispatches on the key. The port is a software interrupt aggregator: disparate sources funnel into one prioritized queue served by one handler loop.

C. Ports: The Asynchronous Dispatch Hub

A **port** is that something else—Zircon's analog of `epoll` or `kqueue`, and the keystone of every real event loop on the system. A port is a queue of *packets*. Instead of a thread blocking *on an object*, threads register asynchronous interest and then block *on the port*, draining whatever packets have accumulated.

`zx_port_create` makes an empty port. `zx_object_wait_async(handle, port, key, trigger_signals, options)` registers interest: "when any bit in `trigger_signals` asserts on `handle`, queue a packet to `port`, tagged with this 64-bit `key`." The `key` is chosen by the caller and is the only link back to application state—typically a pointer or table index identifying *which* connection fired. When the condition is met the kernel enqueues a `zx_port_packet_t` carrying that `key` and, for a signal packet, a `zx_packet_signal_t` reporting the `trigger` mask and the `observed` signals. `zx_port_wait(port, deadline, &packet)` dequeues the next packet.

The default registration is **one-shot**: one matching packet is delivered and the interest is consumed, so the handler re-arms with another `zx_object_wait_async` after servicing. This is the safe default, because it cannot deliver a second packet for an event the handler has not yet processed. Passing `ZX_WAIT_ASYNC_EDGE` instead requests edge-triggered delivery—a packet only when the signal *transitions* from inactive to active, never for a condition already true at registration time—which suits non-blocking I/O loops that drain fully on each wake. Optional flags such as `ZX_WAIT_ASYNC_TIMESTAMP` stamp the packet with the monotonic time of the trigger. A pending async wait that has not yet fired can be withdrawn with `zx_port_cancel(port, handle, key)`—essential when a connection is being torn down while its interest is still registered.

D. One Queue, Many Packet Kinds

What elevates a port from "a better `wait_many`" to "the universal event hub" is that several unrelated event sources deliver into the *same* queue, each with its own `type` tag in the packet:

- `ZX_PKT_TYPE_SIGNAL_ONE` — an object signal fired from `zx_object_wait_async` (the common case above);

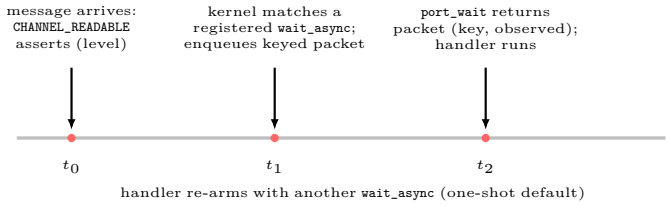


Fig. 24. Timeline of a single signal event through a port. A level-triggered signal asserts on an object; the kernel enqueues a packet to the registered port; `zx_port_wait` returns it to the dispatch loop, which services it and re-arms.

- `ZX_PKT_TYPE_USER` — an application-defined packet injected directly with `zx_port_queue`; the loop's own "wake up, I have work for you" doorbell;
- `ZX_PKT_TYPE_INTERRUPT` — a hardware interrupt object bound to the port (Section XV);
- `ZX_PKT_TYPE_PAGE_REQUEST` — a userspace pager fault request from a pager-backed VMO;
- exception and guest/hypervisor packets bound to the port.

Because a timer signal, an incoming channel message, a fired interrupt, a pager fault, and a hand-posted user packet all surface as packets on one port, a process can run its *entire* concurrency model on a single `zx_port_wait` loop. Fig. 24 traces the lifetime of one such event.

E. The Async Runtime Tie-In

This is not a theoretical capability—it is how Fuchsia programs are actually structured. The C `libasync` dispatcher and the Rust `fuchsia_async` executor each own *one* port and run *one* `zx_port_wait` loop per dispatcher thread. A future that awaits a channel becoming readable is, underneath, a `zx_object_wait_async` keyed to that future's waker; when the packet arrives, the executor matches the key, polls the corresponding future, and advances it. Posting work to a dispatcher from another thread is a `ZX_PKT_TYPE_USER` packet via `zx_port_queue`. Thus the elegant "async/await" surface and the humble port are the same thing viewed at two altitudes: a process's main loop is, in the end, a `zx_port_wait` loop dispatching keyed packets.

XV. Interrupts and Exceptions

Two kinds of events arrive asynchronously to the kernel and must be routed to the right handler: *hardware interrupts* raised by devices, and *exceptions* raised synchronously by a running thread. Zircon exposes both through kernel objects, so that—remarkably—an ordinary user-space driver can field a device IRQ, and an ordinary user-space debugger can field a page fault, with no kernel code change. This section covers both paths and the propagation "chain" that makes exceptions debuggable [15], [16].

A. Hardware Interrupts as Kernel Objects

Almost all Fuchsia drivers live in user space, which raises an obvious question: how does a user thread learn that its device asserted an IRQ? The answer is the **interrupt object**. `zx_interrupt_create(resource, vector, options, &out)` mints one, binding an abstract interrupt to a physical `vector`. The `resource` argument is an IRQ *resource* capability proving the caller is allowed to claim that vector—the same hardware-access gating described in the resources/MMIO section—so interrupt ownership is a capability, not a privilege of being root. The `options` select the trigger mode (`ZX_INTERRUPT_MODE_EDGE_*` / `ZX_INTERRUPT_MODE_LEVEL_*`) or request a purely `ZX_INTERRUPT_VIRTUAL` (software) interrupt.

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_SIGNALED`
Task	`ZX_TASK_TERMINATED`
VMO	`ZX_VMO_ZERO_CHILDREN`
Eventpair	`SIGNALED`, `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.

A driver consumes the object in one of two styles. The simple style is a dedicated thread that calls `zx_interrupt_wait(handle, &timestamp)`, which blocks until the IRQ fires and returns the kernel timestamp of the trigger. The scalable style *binds* the interrupt to a port with `zx_interrupt_bind(interrupt, port, key, options)`; thereafter each firing enqueues a `ZX_PKT_TYPE_INTERRUPT` packet onto that port, so an interrupt sits in the very same `zx_port_wait` reactor loop (Section XIV) as the driver's channels and timers.

Either way, after the device has been serviced the driver must `zx_interrupt_ack` the object to re-arm it for the next IRQ. The ack is not bookkeeping; it is back-pressure. A level- or edge-triggered line is masked between fire and ack, so the kernel will not redeliver—and a misbehaving device cannot livelock the system—until the driver explicitly says it is ready. The object is torn down with `zx_interrupt_destroy` (which also unblocks any waiter with `ZX_ERR_CANCELED`), and a virtual interrupt can be fired in software with `zx_interrupt_trigger`, the basis for testing drivers without hardware.

B. The Interrupt Flow

Fig. 25 traces a real IRQ. The device asserts its line; a *minimal* in-kernel ISR runs, just long enough to quiesce the interrupt controller—mask the line and acknowledge the controller (e.g. EOI to the APIC/GIC)—and then signal the interrupt object (or enqueue its port packet). The kernel does *not* touch the device's own registers; it does not know how. That work belongs to the user driver thread, which now wakes, reads and writes the device's registers through its *MMIO* mapping to actually handle the condition (drain a FIFO, read a status word, restart DMA), and finally calls `zx_interrupt_ack` to unmask the line for the next event.

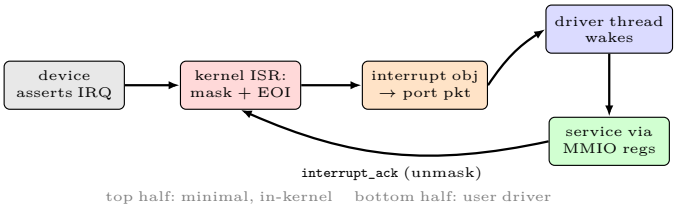


Fig. 25. The user-space interrupt path. A minimal kernel ISR masks the line and acknowledges the controller, then signals the interrupt object (or queues a port packet). The user driver thread services the device through its MMIO mapping and `zx_interrupt_ack`s to re-arm—a top-half/bottom-half split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

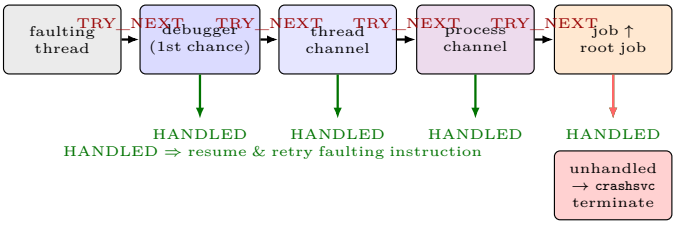


Fig. 26. The exception propagation chain. A fault is offered to the process debugger, then the thread, process, and successive job exception channels up to the root job. Setting `ZX_EXCEPTION_STATE_HANDLED` resumes the thread; `TRY_NEXT` forwards it. If nothing handles it, `crashsvc` reports and the process is terminated.

This is precisely the two-stage hardware handshake an RTL engineer knows: a fast top-half that merely de-asserts and masks the interrupt source so the controller can move on, and a deferred bottom-half that does the real device work—except here the bottom half is an unprivileged user thread, and the boundary between the halves is a kernel object rather than a function pointer.

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 (Message-Signaled Interrupts). Zircon models a contiguous block of these with two syscalls. `zx_msi_allocate` reserves a power-of-two-sized block of MSIs from the platform's MSI resource and returns an *MSI allocation* object; `zx_msi_create` then carves an individual interrupt object out of that allocation for a given MSI id, wired to the device's MSI/MSI-X capability. The resulting object behaves like any other interrupt—`wait`, `bind` to a port, `ack`—so the bus driver's dispatch loop is identical whether the underlying line is a legacy pin or one entry of a 32-vector MSI-X table.

D. Exceptions: Synchronous Thread Faults

An **exception** is the opposite of an interrupt: it is raised *synchronously* by the faulting thread itself. When a thread executes an illegal instruction, dereferences an unmapped address the VM cannot satisfy, violates a job policy, or hits a breakpoint, the kernel *suspends that thread* and goes looking for a handler. The architectural and synthetic 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/protection fault), `ZX_EXCP_UNALIGNED_ACCESS`, the debug events `ZX_EXCP_SW_BREAKPOINT` and `ZX_EXCP_HW_BREAKPOINT`, and the policy-violation exception `ZX_EXCP_POLICY_ERROR`, raised when a thread breaks a rule its job imposed (for instance, calling a forbidden syscall). The "debugger" lifecycle events `ZX_EXCP_THREAD_STARTING`, `ZX_EXCP_THREAD_EXITING`, and `ZX_EXCP_PROCESS_STARTING` ride the same mechanism so a debugger can attach at the right instant.

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 first-chance *debugger* channel. When a matching exception occurs, the kernel writes one message to the channel: a `zx_exception_info_t` (carrying the faulting `pid`, `tid`, and exception `type`) together with a *handle to an exception object*. From that handle the handler obtains the faulting thread and process via `zx_exception_get_thread` / `zx_exception_get_process`, and it fetches the architectural detail—faulting address, registers, synthetic codes—as a `zx_exception_report_t` via `zx_object_get_info` with `ZX_INFO_THREAD_EXCEPTION_REPORT`.

Because the handler holds a real thread handle, it can do anything a debugger does: read and rewrite register and FPU state with `zx_thread_read_state` / `zx_thread_write_state`, 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 to either `ZX_EXCEPTION_STATE_HANDLED`—"I fixed it; retry the faulting instruction"—or `ZX_EXCEPTION_STATE_TRY_NEXT`—"not mine; pass it on." *Closing* the exception handle is what actually releases the thread; with the state left at its default `TRY_NEXT`, the closure propagates the exception to the next handler in the chain.

F. The Propagation Chain

Exceptions are offered to handlers in a fixed order, giving each a "first chance" before the fault becomes fatal (Fig. 26). The order is: the process **debugger** channel (first chance) → the **thread** exception channel → the **process** exception channel → then, walking up the task tree, each **job** exception channel from the immediate parent job to the **root job** (the "second chance"). At each stage, a handler that sets `HANDLED` stops propagation and resumes the thread; a handler that sets `TRY_NEXT` (or simply has no channel registered, or closes the handle without deciding) forwards the exception to the next link.

If the exception reaches the end of the chain unhandled, the thread—and with it the process—is terminated as though killed. The system's last-resort handler, `crashsvc`, is registered high in the job tree; it catches the otherwise unhandled fault, prints a symbolized backtrace and register dump to the log, and emits a minidump for offline analysis before letting the process die. This is why a Fuchsia crash yields a readable stack even though the kernel itself contains no debugger: the debugger is just another exception-channel client, and the crash reporter is the one at the top of the chain.

XVI. Time, Timers, and Clocks

Time in Zircon is not a single number. The kernel distinguishes several *timelines* with different guarantees, hands out cheap read access to the hardware ones, and provides a *clock object* abstraction for the timelines it cannot maintain itself—most

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**
zx_interrupt_create	Bind a vector (via IRQ resource) to an object.
zx_interrupt_wait	Block until the IRQ fires; return timestamp.
zx_interrupt_bind	Route firings as packets to a port.
zx_interrupt_ack	Re-arm the line after servicing.
zx_interrupt_trigger	Fire a virtual interrupt in software.
zx_msi_allocate/ create	Reserve and carve PCI MSI/MSI-X vectors.
create_exception_ channel	Register a handler channel on a task.
zx_thread_read/ write_state	Inspect/modify faulting registers.

importantly UTC. Getting these distinctions right matters: a deadline measured against the wrong timeline is a latent bug [1], [20].

A. Time Bases

The foundational timeline is **monotonic** time, read with `zx_clock_get_monotonic`: nanoseconds since boot that never decrease and never jump. Monotonic time is the reference for every deadline in the system. Its one subtlety is that it is *paused across system suspend*—while the machine is suspended, monotonic time does not advance—so it measures elapsed *running* time, not wall-clock elapsed time.

Boot time, read with `zx_clock_get_boot`, is the companion that *does* include suspended intervals: nanoseconds since power-on regardless of suspend. Code that must reason about real elapsed time across a sleep—a watchdog, a session timeout—uses boot time; code timing CPU-bound work uses monotonic.

UTC is deliberately *not* a kernel primitive. The kernel has no idea what year it is and no battery-backed clock dependency baked into its ABI. Instead UTC is supplied as a user-maintained *clock object* (the clock-object subsection below) driven by a userspace *timekeeper* service that disciplines it from a network or hardware time source. This keeps wall-clock policy—leap handling, source selection, slewing—entirely out of the kernel.

Both monotonic and boot time, and the underlying hardware *ticks*, are readable **without a kernel trap**. The vDSO reads a scaled hardware counter directly in user space: `zx_ticks_get` returns the raw counter and `zx_ticks_per_second` its frequency, from which a nanosecond value is derived by an affine scale the kernel publishes into the vDSO. A tight loop can therefore timestamp itself millions of times per second at the cost of a counter read and a multiply—no syscall, no mode switch—which is essential for tracing and profiling.

B. Timers

A **timer object** turns the passage of time into a waitable signal, splicing time into the same port/signal machinery as everything else. `zx_timer_create(options, clock_id, &out)` makes one, choosing the timeline it counts against—`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`, so it can be waited on directly with `zx_object_wait_one` or, far more commonly, registered on a port with `zx_object_wait_async` and folded into the reactor loop. `zx_timer_cancel` disarms it and clears the signal; re-arming with a new `zx_timer_set` likewise clears any prior assertion.

The `slack` argument is the interesting part. Rather than demanding an exact wakeup, the caller declares a tolerance window, and the kernel is free to fire anywhere in it so that nearby timers *coalesce* into one wakeup. Fewer distinct wakeups means the CPU stays in a low-power idle state longer—a direct battery win. The slack *mode*, chosen in the timer's options, says which direction the window opens: `ZX_TIMER_SLACK_CENTER` permits firing in $[\text{deadline} - \text{slack},\ \text{deadline} + \text{slack}]$, `ZX_TIMER_SLACK_EARLY` only before the deadline, and `ZX_TIMER_SLACK_LATE` only after. A periodic UI animation can afford generous late slack; a hard timeout uses zero.

C. Deadlines Are Pervasive

Time is not confined to timer objects. *Every* blocking syscall in Zircon takes an absolute `zx_time_t` deadline on the monotonic timeline: `zx_object_wait_one`, `zx_port_wait`, `zx_channel_call`, `zx_interrupt_wait`, the futex waits—all of them. The convention is uniform: `ZX_TIME_INFINITE` blocks forever, while `ZX_TIME_INFINITE_PAST` (equivalently a deadline of `0`, already in the past) makes the call a non-blocking poll that returns immediately with `ZX_ERR_TIMED_OUT` if not already satisfied. Because deadlines are *absolute*, not relative, a wait that is interrupted and retried does not drift: the helper `zx_deadline_after(duration)` converts a relative duration to an absolute monotonic deadline once, up front. These same absolute deadlines feed the scheduler's deadline-reservation admission described in the scheduling section, so "when must this wake?" and "when must this run?" speak the same unit.

D. Clock Objects: UTC Without Time Jumps

The most sophisticated time primitive is the **clock object**: a user-maintained, kernel-hosted clock that maps a *reference* timeline (usually monotonic) onto a *synthetic* timeline through an adjustable affine transform (Fig. 27). `zx_clock_create` makes one; `zx_clock_read` returns its current synthetic value; `zx_clock_update` adjusts the transform; and `zx_clock_get_details` exposes the full transform plus an *error bound* estimating how far the synthetic value may be from true.

The transform is the affine map

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

where (r_0, s_0) is an anchoring point—"at reference instant r_0, the synthetic clock read s_0"—and `slope` is the rate ratio between the two timelines. `zx_clock_update` can change the offset s_0 and the `slope` independently. This is what lets a timekeeper *discipline* UTC **without time jumps**: rather than slamming the clock forward when it discovers it is slow—which would make timestamps go backward or skip, breaking ordering—it nudges the `slope` slightly above unity so the synthetic clock gently *slews* into agreement over seconds or minutes, then restores unit rate. Creation options constrain the allowed adjustments: `ZX_CLOCK_OPT_MONOTONIC` forbids the synthetic

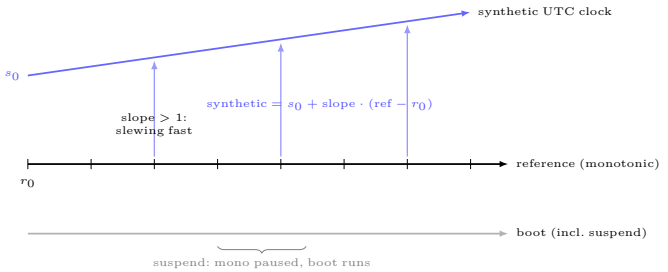


Fig. 27. The Zircon timelines. Monotonic is the reference; boot equals monotonic plus suspended intervals. A UTC clock object is a synthetic timeline mapped from the reference by the affine transform of Eq. (1); adjusting the slope slews UTC into agreement without discontinuities.

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

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

value from ever running backward, `ZX_CLOCK_OPT_CONTINUOUS` additionally forbids *any* jump in either direction (so only rate slewing is permitted), and `ZX_CLOCK_OPT_AUTO_START` has the clock track monotonic time one-to-one until first disciplined.

UTC is exactly this object: a clock created by the platform, handed to processes that need wall-clock time, and continuously updated by the timekeeper. A process reads it with `zx_clock_read` like any other clock and consults the error bound from `zx_clock_get_details` when it needs to know how much to trust the value—all without the kernel ever embedding a notion of calendar time.

XVII. Hardware Access: Resources, MMIO, BTIs, and the IOMMU

A microkernel that runs its drivers in user space must still let those drivers touch real hardware—device registers, interrupt lines, DMA engines—without surrendering the very isolation that motivated moving them out of the kernel. Zircon resolves this tension 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 with an IRQ resource; and it may program a DMA engine only through a *Bus Transaction Initiator* (BTI) that the kernel binds to the platform IOMMU. Each of these is just another kernel object reached through a handle, so the rights model of Section XIX applies unchanged, and authority over hardware is as auditable as authority over a channel.

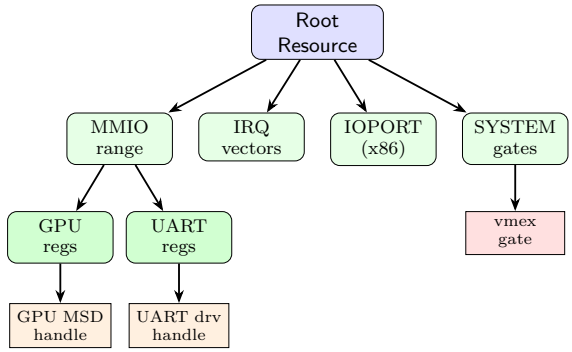


Fig. 28. The resource tree. A single root resource is sliced into MMIO, IRQ, IOPORT, and SYSTEM children, each further narrowed and finally handed to the one driver that needs it.

A. The Resource Object and the Resource Tree

A `Resource` is a capability authorizing access to a restricted kernel or hardware facility, identified by a *kind* and an address *range*. At boot the kernel mints a single **root resource** and hands it—over the bootstrap channel of Section XVIII—to one privileged process (in practice the bootstrap driver-framework process). The root resource is the apex of a *resource tree*: from a parent resource, `zx_resource_create` carves a child covering a sub-range of a single kind, which the holder then distributes to the specific driver that needs it (Fig. 28). Because a child can only ever narrow its parent's range and never change kind, the tree is a strict delegation hierarchy: a driver handed the MMIO window for one controller cannot reach any other device's registers [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` (feature gates), and `ZX_RSRC_KIND_ROOT` (the unrestricted apex, whose base and size are both zero). The `SYSTEM` kind is special: rather than an address range it carries one of a fixed set of *base* selectors, each a single-slot gate on a privileged facility—`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`, among others. The signature

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

encodes the kind in `options`; for an MMIO or IRQ child, `base` and `size` delimit the granted sub-range, while for a `SYSTEM` child `base` is one of the `*_BASE` selectors and `size` is 1.

B. User-Space MMIO

With an MMIO resource a driver pokes device registers directly, entirely in user space. It creates a *physical* VMO over the register window with `zx_vmo_create_physical(mmio_rsrc, paddr, size, &vmo)`,

marks it uncached so that stores are not buffered or reordered behind the register interface—`zx_vmo_set_cache_policy(vmo, ZX_CACHE_POLICY_UNCACHED_DEVICE)`, which on AArch64 selects Device-nGnRE attributes—and maps it with `zx_vmar_map`. The mapped range *is* the register block; an ordinary load or store now reads or writes a hardware register. The cache policy must be set before mapping and while the VMO has no mappings, children, or pinned pages. This is exactly how the companion Graphics tutorial's Magma system driver (MSD) maps a GPU's register file: an MMIO resource, a physical VMO marked uncached, and a VMAR mapping, after which the driver programs the engine 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 discipline through a different kind: an IRQ resource gates `zx_interrupt_create`, after which the resulting interrupt object is waited on or bound to a port exactly as the Interrupts section describes. The resource is the right to *name* a vector; the interrupt object is the delivery mechanism.

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

Programmed I/O suffices for control registers, but bulk data moves by DMA, and a DMA-capable device is a bus master that, left unchecked, can read or write *any* physical address. The BTI is Zircon's answer. A **Bus Transaction Initiator** models exactly one device's view of memory through the IOMMU; it is created from an IOMMU object with `zx_bti_create(iommu, options, bti_id, &bti)`, where `bti_id` is the hardware transaction identifier the platform uses to recognize that device—a PCI bus/device/function on Intel VT-d, a StreamID on an ARM SMMU.

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

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

does three things at once. It locks the VMO's pages into RAM so the PMM cannot move, reclaim, or page them out underneath an in-flight transfer; it programs the IOMMU so that this device—and only this device—may reach those pages; and it returns, in `addrs`, the **bus (device-physical) addresses** the device's DMA engine must be programmed with, together with a **PMT** (Pinned Memory Token) representing the pinning. The permission flags `ZX_BTI_PERM_READ`, `ZX_BTI_PERM_WRITE`, and `ZX_BTI_PERM_EXECUTE` bound what the device may do. By default `addrs` receives one entry per page; `ZX_BTI_COMPRESS` returns one entry per minimum-contiguity block (queryable through `ZX_INFO_BTI`), and `ZX_BTI_CONTIGUOUS` returns a single start address for an already-contiguous VMO. When the transfer is done, `zx_pmt_unpin(pmt)` tears down the IOMMU mapping and releases the pages; it always consumes the token.

The effect is a per-device memory firewall (Fig. 29). Every bus transaction the device issues is checked by the IOMMU against the set of pinned pages for that BTI; a transaction to any other address is faulted rather than allowed to scribble

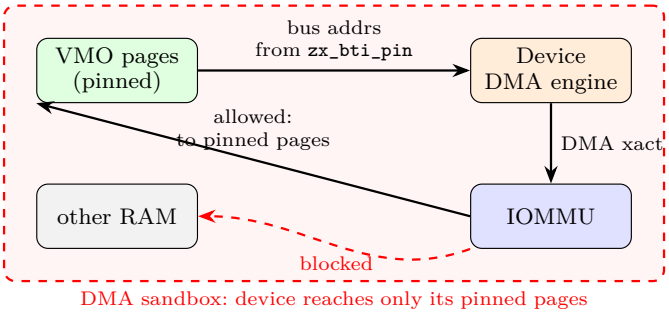


Fig. 29. Device DMA through a BTI. Pinned VMO pages yield bus addresses for the device; the IOMMU checks every transaction against those pages and faults all others, boxing the device to its own buffers.

unrelated RAM. A buggy or compromised device is therefore sandboxed to precisely the buffers its driver handed it—the DMA analogue of the address-space isolation that protects one process from another.

D. The IOMMU Object and Portability

The IOMMU itself is modeled by an object, created from the root resource with `zx_iommu_create(system_rsrc, type, desc, desc_size, &iommu)` where the `SYSTEM` resource carries `ZX_RSRC_SYSTEM_IOMMU_BASE`. The `type` selects the backend: `ZX_IOMMU_TYPE_INTEL` or `ZX_IOMMU_TYPE_ARM_SMMU` for real hardware, or `ZX_IOMMU_TYPE_STUB` (historically the "dummy" IOMMU) for platforms that lack one. The stub IOMMU offers no hardware protection—it cannot fault a stray transaction—but it still *pins* pages and returns their physical addresses identity-mapped, so that driver code written against the BTI API is portable: the same `zx_bti_pin`/`zx_pmt_unpin` sequence works whether or not real translation hardware is present, and gains protection automatically where it is.

Two safety properties round out the design. First, if a PMT is destroyed *without* a proper `zx_pmt_unpin`—a driver crash mid-transfer, say—its pages are not simply freed but *quarantined*: held aside so they cannot be reallocated while a device might still be writing them, defeating DMA-after-free. Quarantined pages are reclaimed explicitly via `zx_bti_release_quarantine` once the device is known quiesced. Second, devices that lack scatter-gather need physically contiguous buffers; `zx_vmo_create_contiguous(bti, size, align_log2, &vmo)` allocates one against a BTI, which combined with `ZX_BTI_CONTIGUOUS` on pin yields a single bus address for the whole transfer.

E. How a User-Space Driver Touches Hardware

The primitives above—resources, physical VMOs, interrupt objects, and the BTI/IOMMU pair—are not used in isolation; a real driver weaves all of them together. Tracing that path end-to-end shows the whole capability story in motion and makes concrete why a user-space driver is both *possible* and *safe*. Consider a DMA-capable PCI device—an Ethernet NIC, or equivalently the GPU register block driven by the companion Graphics tutorial's Magma system driver (Fig. 30).

a) 1) Capability acquisition: When the Driver Framework (DFv2) binds the driver to a device node and starts it inside a `driver_host` process, it *routes* to the driver—as FIDL-delivered capabilities over the `fuchsia.hardware.platform.device` or PCI protocols, never as ambient access—exactly the handles it

TABLE XVII
HARDWARE NEED → KERNEL MECHANISM → SYSCALLS.

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

needs: an MMIO resource (or a ready-made MMIO VMO for the register window), one or more interrupt objects, and a BTI handle for that device. The driver 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) 2) Register access (MMIO): The driver turns its MMIO authority into a physical VMO (`zx_vmo_create_physical`, or it receives the VMO directly), marks it `ZX_CACHE_POLICY_UNCACHED_DEVICE`, and maps it with `zx_vmar_map`. A plain pointer dereference into the mapped range is now a hardware register read or write (Listing 16); the kernel guarantees the mapping covers *only* the authorized physical range.

c) 3) Interrupts: The driver waits on its interrupt object with `zx_interrupt_wait`, or binds it to a port. When the line asserts, the kernel's minimal in-kernel handler masks and acknowledges the controller and wakes the driver thread, which inspects status through its MMIO mapping and calls `zx_interrupt_ack` to re-arm the level-triggered source.

d) 4) DMA—the key memory step: For bulk transfers the driver allocates a buffer as a VMO (contiguous via `zx_vmo_create_contiguous` for a non-scatter-gather engine, or an ordinary VMO otherwise) and calls `zx_bti_pin` to do two things at once: pin the pages so the PMM cannot move or reclaim them mid-transfer, and obtain the *bus addresses* to write into the device's DMA descriptors, plus a PMT. The kernel programs the IOMMU so the device can DMA *only* to those pinned pages. This is the crux: in a monolithic kernel a user-space driver entrusted with a DMA engine could bypass all memory protection by pointing the engine at arbitrary physical RAM, but here a buggy or compromised device is physically fenced to its own buffers. The IOMMU-plus-BTI is precisely what makes user-space DMA-capable drivers safe. When the transfer completes the driver calls `zx_pmt_unpin`.

e) 5) Zero-copy sharing: Because the buffer is just a VMO, the driver can hand it—by handle—to another process: a received packet's VMO travels to the network stack, a rendered frame's VMO to the display compositor, with no intermediate copy. Coordinated through `sysmem` (the companion documents' buffer allocator), data flows device → driver → consumer as pure handle passing.

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

TABLE XVIII
RESOURCE KINDS AND HARDWARE-ACCESS SYSTEM CALLS.

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

XVIII. Boot and User-Space Bootstrap

A capability kernel grants no ambient authority, so the very first question at power-on is where *any* authority comes from. Zircon's answer is a short, explicit chain: the kernel mints the root job and root resource, hands them to exactly one user program over a single channel, and that program hands narrower capabilities onward. There is no global registry and no implicit access; every later process holds precisely the handles its parent chose to give it. This section traces that chain from the bootloader to the first component [19].

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 begins with a container header and is followed by a sequence of typed items, each with its own header (`type`, `length`, `flags`, a CRC). The kernel image is itself the first item—`ZBI_TYPE_KERNEL_X64`, `ZBI_TYPE_KERNEL_ARM64`, or `ZBI_TYPE_KERNEL_RISCV64`—and the remaining items carry everything early boot needs: `ZBI_TYPE_CMDLINE` (the kernel command line), `ZBI_TYPE_STORAGE_BOOTFS` (the compressed `bootfs` ramdisk that holds userspace), `ZBI_TYPE_MEM_CONFIG` (the usable-RAM map), `ZBI_TYPE_PLATFORM_ID` and `ZBI_TYPE_CPU_TOPOLOGY` (board and core layout), `ZBI_TYPE_ACPI_RSDP` or `ZBI_TYPE_DEVICETREE` (firmware tables), and `ZBI_TYPE_KERNEL_DRIVER` items describing the serial console and other early peripherals. The first item is the kernel itself, whose payload is prefixed by a `zbi_kernel_t` header giving the `entry` point and the `reserve_memory_size` of scratch RAM the kernel needs immediately after its load image. The ZBI is, in effect, the kernel's argument vector.

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

Everything above happens before a single line of the kernel proper runs. For a hardware-minded reader it is worth tracing the full low-level sequence from reset to a running scheduler (Fig. 31); the user-space hand-off that the rest of this section covers is only its final step.

a) 1) Power-on and reset: The boot CPU comes out of reset at the architecture's reset vector in ROM. SoC or platform firmware—UEFI/BIOS on x86, a boot ROM plus a bootloader such as coreboot, U-Boot, or Gigaboot on ARM—initializes DRAM, clocks, and the boot core, then loads the boot image into RAM.

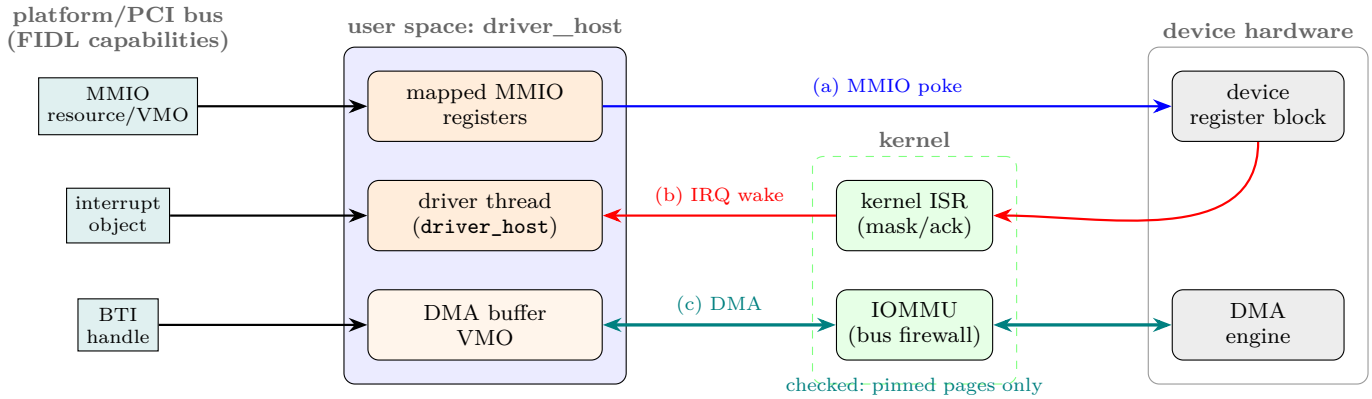


Fig. 30. How a user-space driver touches hardware. The Driver Framework routes three capabilities into the `driver_host`: an MMIO resource/VMO, an interrupt object, and a BTI. The driver (a) pokes registers through the mapped MMIO window, (b) is woken on IRQ via the kernel's minimal handler and re-arms with `zx_interrupt_ack`, and (c) DMAs to a BTI-pinned VMO whose every bus transaction the IOMMU checks against the allowed pages—the sandbox that makes user-space DMA safe.

b) 2) Bootloader to ZBI: The bootloader transfers control to the Zircon image, passing the physical base of the ZBI. The calling convention is per-arch: on ARM64 and RISC-V the kernel is entered with a pointer to the ZBI in a register; on x86 a small multiboot/EFI shim adapts the firmware handoff. The kernel item's `zbi_kernel_t` header tells the loader where to jump and how much memory to reserve.

c) 3) physboot: decompress, relocate, hand off: The kernel item is a compressed ELF, so a physical-address-mode bootstrap, **physboot**, runs first. It decompresses the real kernel, loads it as a position-independent ELF, relocates it to its final virtual address (the hook for kernel ASLR), reserves the memory the kernel claimed, builds the hand-off structures—the parsed boot options and a `PhysHandoff` record describing the physmap and initial mappings—and jumps to the kernel entry point.

d) 4) Early architecture init: The kernel's per-arch entry brings the CPU to a known state before any portable code runs. On ARM64 it drops from EL2/EL3 to **EL1**, installs the exception-vector base (`VBAR_EL1`), programs `MAIR_EL1`/`TCR_EL1` and the `TTBR0`/`TTBR1` page-table roots, turns on the MMU and caches, sets the boot stack, and zeroes BSS; on x86 the equivalent long-mode transition, GDT/IDT load, and paging enable are performed (largely within physboot on current builds). Either way the kernel high-half *physmap*—a fixed mapping of physical RAM—is established so the kernel can reach any page, and control reaches the C entry `lk_main()`.

e) 5) Kernel main and the init levels: `lk_main` drives bring-up as 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 bring up the early UART), then the pre-heap VM bootstrap, the kernel heap, the full VM/VMAR system, the interrupt controller, CPU topology, and the core kernel objects. By the `THREADING` level the scheduler and threading primitives are usable. Somewhere around the VM level the platform timer is calibrated to a known frequency—measuring the x86 TSC against the HPET, reading `CNTFRQ_EL0` on ARM, or taking the timebase frequency from boot config on RISC-V—so that the monotonic clock is well-defined.

f) 6) SMP bring-up: The boot CPU then starts the secondary cores, each through its architecture's mechanism behind a uniform interface: PSCI `CPU_ON` on ARM, the INIT–SIPI–SIPI sequence on x86, and the SBI hart-state-management `sbi_hart_start` on RISC-V. Each secondary runs its own per-CPU init and synchronizes at the `SMP_WAIT`/`SMP_READY` barrier before joining scheduling.

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`

g) 7) Threading is up: `lk_main` creates an initial bootstrap thread to finish the higher init levels, then the boot CPU executes `Thread::Current::BecomeIdle()`, becoming its per-CPU idle thread and enabling interrupts—at which point the scheduler is live and the system is "up." Each secondary likewise settles into its own idle thread.

h) 8) 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. That seam, from a running kernel to the first user program, is where the remainder of this section takes over.

C. userboot: the First Program

`userboot` is a tiny user-space loader compiled *into* the kernel image, laid out like the vDSO. The kernel maps both `userboot` and the vDSO into the first process and jumps to `userboot`'s entry point; it never parses an ELF on disk to get started, because there is no disk yet. The kernel hands `userboot` its initial state over a **bootstrap channel** as a single `processargs` message: a handle to the root job, the root resource (and the finer-grained MMIO/IRQ/IOPORT/SYSTEM resources), the process and thread self handles, the root VMAR, the initial

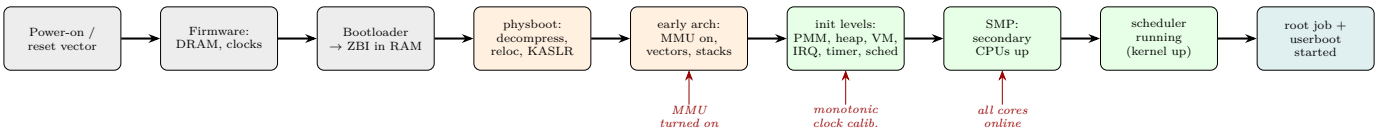


Fig. 31. Life of a kernel boot. The lanes shade hardware/firmware (gray), physboot and early arch (orange), the kernel proper (green), and the user-space seam (teal). Annotated below: where the MMU is enabled, where the monotonic clock is calibrated, and where SMP brings the remaining cores online.

stack VMO, the vDSO VMO, and—crucially—the ZBI itself as a VMO tagged `PA_VMO_BOOTDATA`.

`userboot` scans that ZBI for the first `ZBI_TYPE_STORAGE_BOOTFS` item and decompresses it into a fresh VMO: this is **bootfs**, a simple read-only filesystem image holding the binaries and libraries of the bootstrap world. It then reads the command line (delivered as the message's environment strings) for two options: `userboot.next`, the program to launch—defaulting to `bin/component_manager+-boot`, where `+` separates arguments—and `userboot.root`, an optional path prefix inside bootfs. `userboot` loads that next program from bootfs, acts as its dynamic-linker *loader service* (the `PA_LDSVC_LOADER` handle), sends it a fresh `processargs` message carrying the handles it received—now with the new process's own process/thread/VMAR handles substituted—and gets out of the way. When the launched program exits, `userboot` can power off or reboot (`userboot.shutdown`, `userboot.reboot`); in normal operation it simply waits.

D. The processargs Bootstrap Protocol

The mechanism behind every hand-off in Fig. 32 is the **processargs** protocol: the convention by which a freshly created process receives its initial capabilities. The starting process reads one message from its bootstrap channel; the message's *handle array* carries the actual handles, and its byte payload is a `zx_proc_args_t` header followed by three parallel tables—a *handle-info* array, the argument strings, the environment strings, and a names table (Fig. 33). Each entry of the handle-info array is a 32-bit word, built by the `PA_HND(type, arg)` macro, that tags the handle at the same index with a `PA_*` *type* (in the low byte) and a 16-bit *argument* (in the high half). The type tells the new program what each handle *is*; the argument disambiguates among several of the same type—for example the nth file descriptor or the nth namespace directory.

The standard types span the authority a program needs to bootstrap itself: `PA_PROC_SELF` and `PA_THREAD_SELF` (its own process and first thread), `PA_JOB_DEFAULT` (the job in which to spawn children), `PA_VMAR_ROOT` (its address space), `PA_VMO_STACK` (its initial stack), `PA_VMO_VDSO` (the vDSO to map for syscalls), `PA_LDSVC_LOADER` (the loader service that supplies shared libraries), `PA_NS_DIR` (namespace directories, indexed by the names table), `PA_FD` (file descriptors), and `PA_DIRECTORY_REQUEST` (the server end on which this program should publish its own outgoing services). Privileged programs additionally receive `PA_VMO_BOOTDATA` (the ZBI), and resource handles such as `PA_MMIO_RESOURCE`, `PA_IRQ_RESOURCE`, `PA_IOPORT_RESOURCE`, and `PA_SYSTEM_RESOURCE` (the kind-specific successors to the single `PA_RESOURCE` root handle). This one table *is* the kernel-to-user handoff of authority: a process can do exactly what its handles permit, and nothing more.

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

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

Name	Meaning
processargs handle types	
`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)
ZBI item types	
`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)

XIX. The Security Model: Policy, Sandboxing, and Restricted Mode

Zircon's security model is not a layer bolted onto the kernel; it is the direct consequence of the object/handle/rights core. There is no ambient authority: a thread cannot name a file, a device, or another process except through a handle it was given, and a handle authorizes only the operations its rights bits permit (Section on rights). This section shows how that base principle is hardened into a defense-in-depth stack—job policy that constrains what a subtree may even attempt, W^X enforcement that gates executable memory behind a capability, and *restricted mode*, the mechanism that lets Zircon run an entire foreign operating-system personality inside one process without trusting it [1].

A. CPU Execution Modes

Before the policy details it helps to fix the set of execution contexts a CPU occupies under Zircon, because the security story is largely about the transitions between them (Table XXI). There are four. *Kernel* (supervisor) mode is the kernel itself—EL1 on ARM64, ring 0 on x86, S-mode on RISC-V—running the scheduler, the VM system, and the object dispatchers with full privilege. *Normal user* mode is where ordinary Fuchsia components run—EL0, ring 3, U-mode—reaching the kernel only through the vDSO syscall gate. *Restricted* mode is a constrained user context (still EL0/ring 3 at the hardware level) with a *separate* register set and a restricted address-space view,

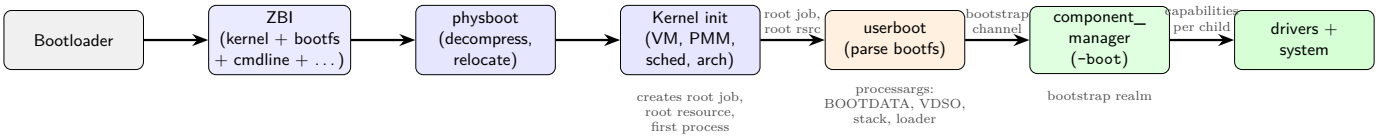


Fig. 32. The boot pipeline. Authority is created once by the kernel and handed down a chain of single bootstrap channels: kernel → `userboot` → `component_manager` → the rest of the system. Each arrow's annotation names the handles that ride in that hand-off.

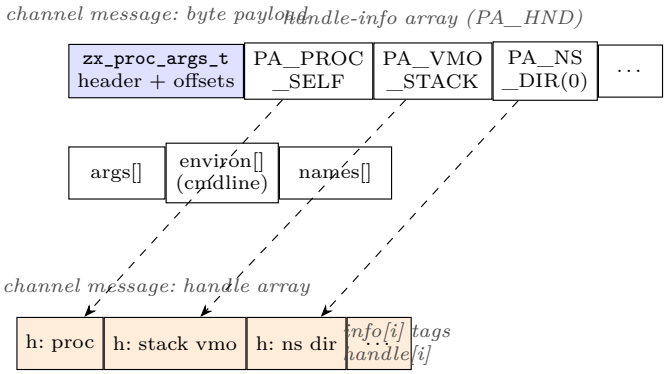


Fig. 33. The `processargs` bootstrap message. A handle-info word tags each handle in the channel message's handle array with a `PA_*` type; arguments, environment (the kernel command line), and names follow in the byte payload.

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)

in which untrusted or foreign-ABI code runs; on any syscall or fault it bounces back to its normal-mode host rather than entering the kernel directly. *Guest* mode is code running under the hypervisor with stage-2/EPT translation on a virtual CPU—the strongest isolation, detailed in the architecture section. The progression from normal to restricted to guest is a progression in how thoroughly the kernel distrusts the code it is running, and each step adds a hardware-checked barrier.

B. Job Policy as Sandboxing

Capabilities answer "may this handle do that?"; job policy answers the prior question "may this process even create such a thing?". A job carries an inherited policy installed with

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

Under the `ZX_JOB_POL_BASIC` topic the `policy` argument is an array of `(condition, action)` pairs. The *conditions* name an attempted operation: the object-creation gates `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`; the misuse conditions `ZX_POL_BAD_HANDLE` (using an invalid handle value) and `ZX_POL_WRONG_OBJECT` (an operation on the wrong object type); and the memory-safety conditions `ZX_POL_VMAR_WX` (mapping a region writable and executable at once—i.e. forbidding W^X violations) and `ZX_POL_AMBIENT_MARK_VMO_EXEC` (marking a VMO executable without presenting a capability). The *action* chosen for each condition is one of `ZX_POL_ACTION_ALLOW`, `ZX_POL_ACTION_DENY`, `ZX_POL_ACTION_KILL`, or the exception-raising variants `ZX_POL_ACTION_ALLOW_EXCEPTION` and `ZX_POL_ACTION_DENY_EXCEPTION`, which additionally generate a debugger-visible exception so a policy hit can be caught and diagnosed.

Policy is *inherited and monotonic*: it flows from a job to all its descendants and can only ever be tightened. Under `ZX_JOB_POL_RELATIVE` a child's new rules apply only where a parent has not already spoken; under `ZX_JOB_POL_ABSOLUTE` every listed condition must take effect or the call fails. In neither case can a child loosen a restriction an ancestor imposed. This is exactly the property an SoC integrator wants from a sandbox: the most restrictive enclosing domain wins, and no inner block can vote itself more privilege than its container allows.

C. Executable Memory Is a Capability

A classic exploit converts a data buffer into code. Zircon forecloses this by making executability a right, not a default. Mapping a VMO with `ZX_VM_PERM_EXECUTE` requires the VMO handle to carry `ZX_RIGHT_EXECUTE`, and that right cannot be conjured: it is obtained only through `zx_vmo_replace_as_executable(vmo, vmex, &out)`, where `vmex` is a SYSTEM resource carrying `ZX_RSRC_SYSTEM_VMEX_BASE`—the privileged "make memory executable" gate. A process that wishes to JIT-compile must therefore be *handed* a vmex capability; one that is not, cannot produce new executable pages at all. The job-policy condition `ZX_POL_AMBIENT_MARK_VMO_EXEC` controls the relaxation: where its action is `ALLOW`, a process may pass `ZX_HANDLE_INVALID` for `vmex` (ambient executability, for runtimes that legitimately need it); where it is `DENY`, a real vmex resource is mandatory. W^X is thus enforced on two fronts at once—no page is simultaneously writable and executable (`ZX_POL_VMAR_WX`), and the transition to executable is itself capability-gated.

D. Restricted Mode

The strongest tool short of full virtualization is **restricted mode**: a way to run untrusted or foreign-ABI code *inside* a normal process with a separate register set and a restricted view of memory, bouncing back to trusted "host" code on every syscall or fault. A host thread first calls `zx_restricted_bind_state` to obtain a shared *restricted-state VMO*; the guest's register file—a `zx_restricted_state_t` (general registers, instruction pointer, stack pointer, flags, thread pointer)—lives at offset 0 of that VMO. The host then calls

```
zx_restricted_enter(options, vector_table_ptr,
```

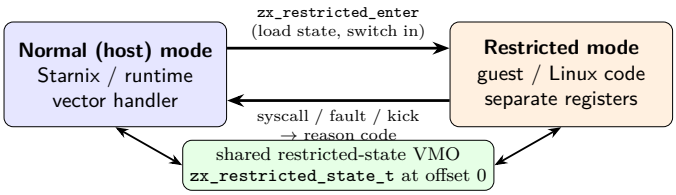


Fig. 34. The restricted-mode bounce. `zx_restricted_enter` switches the CPU into restricted mode running guest code; a guest syscall, fault, or kick returns control to the host's vector handler with a reason code. Both modes share the restricted-state VMO that holds the guest register set.

```
context)
```

which loads the saved register set and switches the CPU into restricted mode running the guest code. Restricted code executes natively at full speed until it does something that must be mediated—it executes a syscall instruction, or it faults. At that moment the kernel writes the guest's registers back into the shared VMO and returns control to *normal* mode at `vector_table_ptr`, passing the host's `context` and a *reason code*: `ZX_RESTRICTED_REASON_SYSCALL` for a guest syscall, `ZX_RESTRICTED_REASON_EXCEPTION` for a fault, or `ZX_RESTRICTED_REASON_KICK` when another thread called `zx_restricted_kick` to preempt it (Fig. 34). The host services the event by reading and rewriting the shared register set, then re-enters.

This is precisely how **Starnix** runs unmodified Linux binaries on Fuchsia. The Starnix kernel creates a Zircon process whose address space is split: the lower half is a private, restricted address space holding the Linux program, and the upper half—shared (via the `ZX_PROCESS_SHARED` facility) across all such processes—holds the Starnix kernel itself. A Linux thread runs in restricted mode in the lower half and *cannot even address* the Starnix half. When the Linux program executes its architecture's syscall instruction (`syscall` on x86, `svc` on ARM), the kernel bounces to Starnix with `ZX_RESTRICTED_REASON_SYSCALL`; Starnix reads the Linux syscall number and arguments from the shared register set, emulates that Linux syscall on top of Zircon objects (a Linux `read` becomes operations on a Zircon channel or VMO, a Linux `mmap` becomes a VMAR mapping), writes the result back into the register set, and re-enters. The Linux ABI thus runs as a user-space personality with no kernel modifications and no loss of isolation— each Linux process is boxed exactly as any other Zircon process.

E. Defense in Depth

These mechanisms compose into concentric barriers (Fig. 35). The outermost is job policy: a subtree that may not even create a channel cannot begin to misbehave. Within that, handle unforgeability and rights narrowing bound what each held capability can do. Within that, per-process address-space isolation (VMARs and the MMU) prevents one process from reading another's memory, and the vDSO PC check ensures syscalls enter only through the sanctioned gate. Restricted mode adds an intra-process barrier for foreign code, and— strongest of all—the hypervisor's stage-2 translation (the arch section) confines an entire guest OS to a guest-physical sandbox. An attacker must defeat every enclosing ring, and each ring is small enough to reason about on its own.

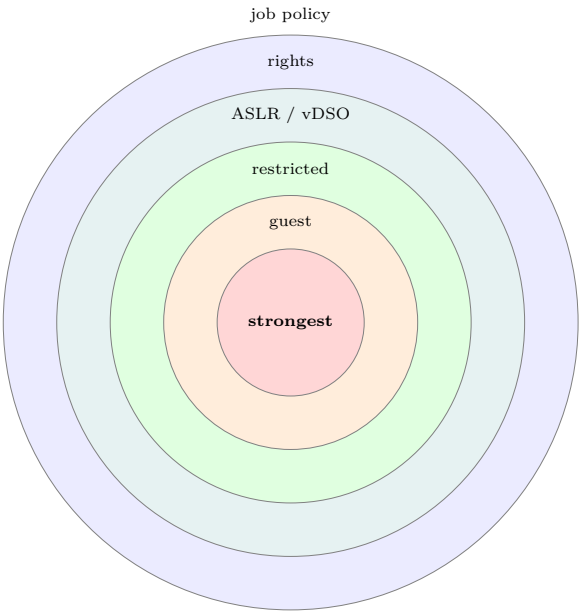


Fig. 35. The layered sandbox. Each ring is an independent barrier; authority is checked at every level, and the innermost (a hardware-enforced guest) is the strongest.

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

XX. Diagnostics, Tracing, and Debugging

A kernel that pushes drivers, filesystems, and services into user space must still give an engineer the means to see inside a running system—to read the log before any logger exists, to profile the scheduler at nanosecond resolution, to enumerate every mapping in a wedged process, and to attach a source-level debugger to a faulting thread. Zircon exposes each of these as ordinary object operations, so the same handle-and-rights discipline that governs the rest of the kernel also governs introspection: a tool can observe only what it has been handed a capability to observe [3].

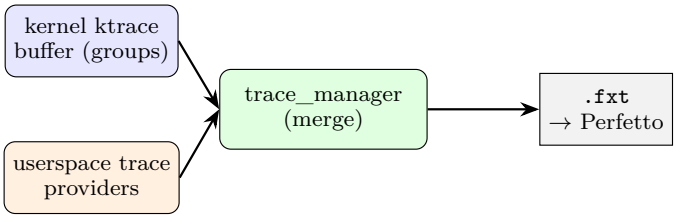


Fig. 36. Tracing data flow. The kernel's ktrace buffer and the userspace trace providers are merged by `trace_manager` into one `.fxt` timeline viewed in Perfetto.

A. The Kernel Debug Log

The lowest-level diagnostic is the **debuglog** (the `klog`): an in-kernel ring buffer of timestamped records, mirrored to the serial console and available from the very first instant of boot, long before the userspace logging service is up. A handle to it is opened with `zx_debuglog_create(rsrc, options, &log)`—the resource is a SYSTEM resource carrying `ZX_RSRC_SYSTEM_DEBUGLOG_BASE`, and passing `ZX_LOG_FLAG_READABLE` yields a readable (not merely writable) handle. `zx_debuglog_write` appends a record and `zx_debuglog_read` drains one; because the buffer survives across the early-boot transitions, it is the natural place for both kernel panics and the earliest userspace bring-up messages to land. When nothing else works, the serial `klog` usually still does.

B. Kernel Tracing and the Unified Timeline

For performance work Zircon emits structured **trace records** from inside the kernel: context switches and thread-state changes, syscall entry and exit, interrupt delivery, page faults, and IPC events. Tracing is armed through `zx_ktrace_control(rsrc, action, options, ptr)`—with `KTRACE_ACTION_START`, `KTRACE_ACTION_STOP`, and `KTRACE_ACTION_REWIND`, the `START` action taking a *group* bitmask (`KTRACE_GRP_SCHEDULER`, `KTRACE_GRP_SYSCALL`, `KTRACE_GRP_IRQ`, `KTRACE_GRP_VM`, `KTRACE_GRP_IPC`, ...) that selects which record families are captured—and the buffer is drained with `zx_ktrace_read`. These syscalls are gated on a debugging build option, since tracing exposes timing of the whole machine.

The kernel stream is only half of the picture. A userspace **trace manager** coordinates the system: it registers *trace providers*—one of which, `ktrace_provider`, surfaces the kernel's records—hands each provider a buffer VMO to write into, and *merges* the kernel records with every userspace provider's events into a single timeline (Fig. 36). The merged output is written in the Fuchsia Trace Format (`.fxt`) and visualized in Perfetto, where a scheduler hiccup, a lock convoy, or a latency outlier can be read directly off the timeline against both kernel and application events. This kernel-plus-userspace correlation is the primary tool for scheduler and latency analysis on Fuchsia.

C. Object Introspection

Most steady-state diagnosis goes through `zx_object_get_info`, the single typed query over any object. Its *topic* selects the record: a process's address space and mappings via `ZX_INFO_PROCESS_MAPS` and its VMOs via `ZX_INFO_PROCESS_VMOS`; per-task CPU accounting via `ZX_INFO_TASK_STATS` and `ZX_INFO_TASK_RUNTIME`; system-wide memory via `ZX_INFO_KMEM_STATS` (and `ZX_INFO_KMEM_STATS_EXTENDED`); the handle table via `ZX_INFO_HANDLE_TABLE`; per-CPU load via `ZX_INFO_CPU_STATS`; and the task tree via `ZX_INFO_JOB_CHILDREN` and `ZX_INFO_JOB_PROCESSES`. Mutable per-object attributes are read and written with `zx_object_get_property` / `zx_object_set_property`: `ZX_PROP_NAME` (the human label that makes a trace or a task list legible), `ZX_PROP_PROCESS_DEBUG_ADDR` and `ZX_PROP_PROCESS_BREAK_ON_LOAD` (the debugger's hooks for finding the loaded-module list and breaking on module load), and others. Complementing these are the kernel's **kcounters**—named counters compiled into the kernel and exposed through the `kcounter` tool—which give cheap, always-on tallies of events like page faults, context switches, and syscalls.

TABLE XXIII
SELECTED `ZX_OBJECT_GET_INFO` TOPICS AND DIAGNOSTIC SYSCALLS.

Topic / syscall	Yields
get_info topics	
`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
diagnostic syscalls	
`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

D. Debugging Faults

When a thread does fault, Zircon's exception model (the exceptions section) carries it to user space over an *exception channel*, obtained with `zx_task_create_exception_channel` on a thread, process, or job. Two consumers matter here. The first is **crashsvc**, which registers on the root job's exception channel, catches otherwise-unhandled exceptions, and emits a symbolized backtrace (and can produce a minidump) before the process is killed or handed off—so even an unattended crash leaves a usable report on the `klog`. The second is **zxdb**, the Fuchsia source-level debugger. zxdb runs on the host and drives an on-device `debug_agent`, which binds exception channels and inspects threads through `zx_thread_read_state` / `zx_thread_write_state`—reading and writing register sets such as `ZX_THREAD_STATE_GENERAL_REGS` on a thread that is suspended or stopped in an exception. The same syscalls underpin single-stepping and breakpoint insertion. For early or wedged systems there is also the in-kernel shell: `k` commands over serial (`k threads`, `k counters`, `k oom info`, ...) read kernel state directly when no userspace tool can run.

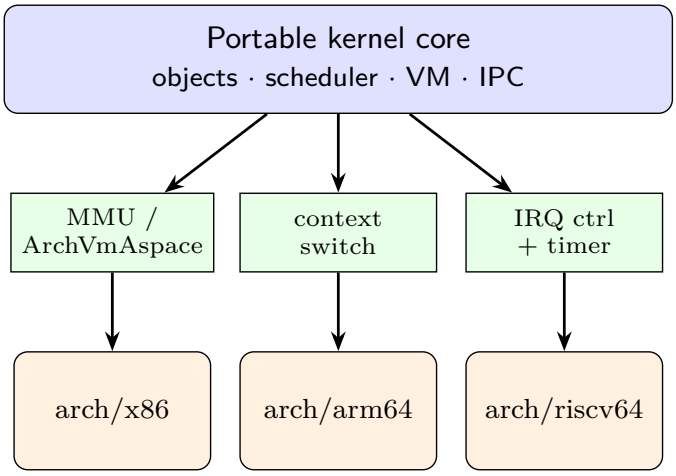


Fig. 37. The arch abstraction. A portable core reaches hardware only through fixed hooks (MMU, context switch, IRQ controller, timer), each implemented once per ISA under `arch/x86`, `arch/arm64`, and `arch/riscv64`.

XXI. Multi-Architecture Support and Virtualization

Zircon runs on three 64-bit instruction-set architectures—x86-64 (the `x64` target), ARM64 (`arm64`), and RISC-V (`riscv64`, introduced by RFC-0211)—from one portable kernel core. The trick is the same one a good RTL design uses to retarget across process nodes: confine everything that differs to a thin, well-specified layer and let the bulk of the logic remain oblivious. In Zircon that layer is the `arch/` tree, with one subdirectory per ISA, beneath a core (objects, scheduler, VM, IPC) that is written once. The hypervisor at the bottom of this section then reuses 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 with a per-ISA implementation: the MMU and page-table format, the context switch, the exception and interrupt vectors, per-CPU register access, atomics and memory barriers, cache maintenance, and the platform timer and interrupt controller (Fig. 37). The most consequential of these is memory management. The virtual-memory subsystem manipulates address spaces only through the `ArchVmAspace` interface, which hides the actual page-table hardware: an `ArchVmAspace` on x86-64 walks `PML4`-rooted tables (four-level paging, or five-level `LA57` where present) and tags TLB entries with **PCID**s; on ARM64 it manages the split `TTBR0_EL1`/`TTBR1_EL1` (user and kernel halves) in the AArch64 long-descriptor format with **ASID**-tagged entries; and on RISC-V it programs the `satp` CSR for **Sv39** or **Sv48** paging. The portable VM code (Section on VM) creates mappings and never sees any of this; a TLB shootdown, a context switch, or an address-space teardown calls down through the same interface on every target.

B. Interrupt Controllers and Timers

Two more facilities differ sharply across ISAs yet are presented uniformly to user space. Interrupts, whatever their hardware source, surface as Zircon interrupt objects (the interrupts section): on x86 the local **APIC** / **x2APIC** plus the **IO-APIC** route vectors; on ARM the **GIC** (v2 or v3) does; on RISC-V the **PLIC** handles external device interrupts, while the M-mode **CLINT** timer/IPI source is reached through SBI firmware calls since the kernel runs in supervisor mode. A driver that calls `zx_interrupt_wait` is unaware which of these delivered its interrupt.

The scheduler tick and the monotonic clock (the time section) ride on the platform timer, again per-ISA: the x86 invariant **TSC** (with the HPET or local-APIC timer for one-shot deadlines), the ARM generic timer's physical counter **CNTPCT**, and the RISC-V supervisor `time` CSR. Each is calibrated at boot to a known frequency so that `zx_clock_get_monotonic` returns nanoseconds with identical semantics everywhere. Table XXIV gathers these per-architecture facts.

C. Virtualization: the Hypervisor and Guests

The strongest isolation tier Zircon offers is not a software sandbox but hardware-enforced virtualization. A *guest* is created with

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

from a SYSTEM resource carrying `ZX_RSRC_SYSTEM_HYPERVISOR_BASE`. Notably the call returns *two* handles: the guest object and a **guest-physical VMAR**—an address region, structurally like an ordinary process VMAR, that represents the guest's physical address space. The host VMM populates it by mapping VMOs of guest RAM, but those mappings are backed by *second-level* address translation: Intel **EPT** on x86, ARM **stage-2** on arm64. Every guest memory access is walked twice—guest-virtual to guest-physical by the guest's own page tables, then guest-physical to host-physical by the stage-2 tables the kernel controls—so the guest is confined to exactly the host RAM its VMAR maps, an IOMMU-like firewall around an entire operating system.

A virtual CPU is created with `zx_vcpu_create(guest, options, entry, &vcpu)` and run with `zx_vcpu_enter(vcpu, &packet)`, which executes guest code on the calling thread until a **VM-exit** forces a return. The exit is delivered as a `zx_port_packet_t`—`ZX_PKT_TYPE_GUEST_MEM` for a trapped memory access, `ZX_PKT_TYPE_GUEST_IO` for port I/O, and so on—which the host VMM decodes to emulate the device the guest was touching, then resumes the VCPU (Fig. 38). Ranges that should trap are registered with `zx_guest_set_trap` (kinds `ZX_GUEST_TRAP_MEM`, `ZX_GUEST_TRAP_IO`, and the asynchronous-doorbell `ZX_GUEST_TRAP_BELL`, the last delivering its packet to a *port* rather than through `zx_vcpu_enter`); a second thread can force an exit at any time with `zx_vcpu_kick`, which makes the in-flight `zx_vcpu_enter` return `ZX_ERR_CANCELED`. On Fuchsia the user-space VMM that ties this together is **Machina**, which builds the guest memory map, installs the traps, and emulates virtio devices in isolated processes. (Hardware virtualization is supported on Intel VMX and ARMv8 stage-2; AMD SVM and RISC-V H-extension hosting are not.) Because the VMM is an ordinary user-space process and the guest is confined by hardware, this is the firmest boundary in the system—the natural place to run an untrusted or legacy stack on an FPGA-hosted SoC.

It is worth contrasting the hypervisor with restricted mode (Section XIX), the system's two foreign-code mechanisms at opposite ends of an isolation/cost spectrum. Restricted mode is *syscall-level*: the guest shares the host process's hardware context, its "syscalls" are bounced to a user-space host that

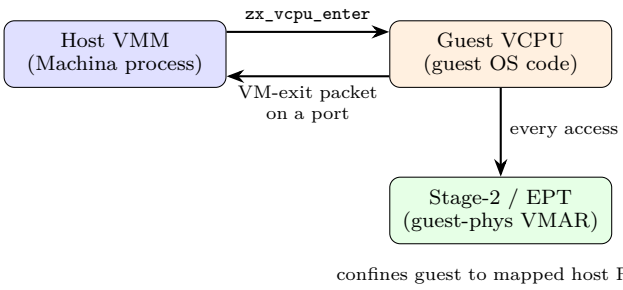


Fig. 38. The guest / VCPU model. The host VMM enters the VCPU with `zx_vcpu_enter`; the guest runs under stage-2 (EPT) translation, and a VM-exit is delivered back to the VMM as a port packet to emulate.

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	CNTPCT	`time` CSR
Hypervisor	VMX/EPT	stage-2	—

emulates a single ABI, and no second translation stage is involved—cheap, but it trusts the host runtime to get the ABI right and is limited to code that issues recognizable syscall instructions. Full virtualization is *instruction-level*: the guest runs a complete, unmodified kernel under hardware trap-and-emulate with its own stage-2-translated address space, paying for a heavier VM-exit on every trapped access but isolating an entire operating system with no software in the trusted path. Starnix uses the former to run a Linux *userland* as a Fuchsia process; Machina uses the latter to run a Linux *kernel* as a guest. The two compose: a hardened deployment can run foreign userland in restricted mode and reserve the hypervisor for code that must believe it owns a machine.

XXII. Conclusion

Zircon is best understood as a small, regular *mechanism* core over which almost all *policy* is layered in user space. The kernel exports a few dozen object classes through one uniform interface—unforgeable handles carrying fine-grained rights—and refuses to grant authority that a process was not explicitly handed. Everything else in this reference is a consequence of that single decision. Tasks (jobs, processes, threads) form a tree whose edges carry both resource accounting and policy; the fair, deadline-aware scheduler arbitrates CPUs among them; memory is described declaratively by VMOs and placed by VMARs, with the page queues and a *user-space* pager deciding what is resident; asynchrony is funneled through signals, ports, and futexes so that one thread can wait on thousands of sources; interrupts, exceptions, and time are themselves objects; hardware is reached only through resource capabilities, physical VMOs, and the BTI/IOMMU DMA firewall; authority is bootstrapped exactly once by `userboot` and the `processargs` protocol; the security model composes capabilities with job policy, W^X enforcement, and restricted mode; and portability is confined to a thin `arch/` layer beneath which the hypervisor provides the strongest isolation tier of all.

For a reader who designs hardware, the structure maps cleanly onto familiar RTL intuition. Kernel *objects* behave like IP blocks with a register interface: self-contained, individually addressable, composed by wiring rather than by shared global state. *Handles* are unforgeable grant tokens—a capability bus on which possession is permission and rights are the per-wire enable bits. *Channels* are the moral equivalent of an AXI-Stream link carrying framed messages (and, uniquely, capabilities) point-to-point with backpressure. *Ports* are an interrupt aggregator that funnels many edge- and level-style sources into one serviced queue. The *BTI/IOMMU* pairing is a bus firewall that constrains a master's address window exactly as an SoC's system MMU constrains a DMA-capable peripheral. The *scheduler* is the arbiter that time-multiplexes the compute fabric under a bandwidth-and- deadline contract. A kernel built this way suits an FPGA-hosted SoC operating system precisely because its trust boundaries are explicit and local: a soft-core driver that pokes a memory-mapped block does so through an MMIO resource and a pinned VMO, a DMA-capable accelerator is boxed by a BTI, and an untrusted or foreign-ABI workload is confined by restricted mode or a guest— each boundary checkable in isolation, exactly as one would verify a block in simulation before integration.

This document is one of three companions. The *Graphics & Display Pipeline* tutorial shows the object model in anger: the Magma system driver (MSD) maps its GPU register block through an MMIO resource and a physical VMO, and feeds the engine through BTI-pinned command and data buffers exactly as described in Section XVII. The *Build System & Toolchains* tutorial explains how the kernel, the vDSO, `userboot`, and the `bootfs` image assembled into the ZBI of Section XVIII are actually produced and assembled. Together the three give a hardware-minded engineer a complete path from a Verilog block, through the kernel that mediates it, to the graphics stack and the build that ships it.

XXIII. Appendix A: Syscall Reference

Table XXV is a representative (not exhaustive) index of the Zircon system-call surface, grouped by subsystem, with a one-line purpose for each call. All calls return `zx_status_t` unless noted; handle arguments are checked for the required rights before any effect occurs [3]. The vDSO is the only legal entry point to every one of these (Section XVIII).

TABLE XXV: Representative Zircon system calls, by subsystem.

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

(continued on next page)

Table XXV (continued from previous page)

Syscall	Purpose
Time, timers, clock	
`zx_clock_get_monotonic`	Read the monotonic clock (ns since boot, no pauses).
`zx_clock_get_boot`	Read the boot clock (includes suspend).
`zx_nanosleep`	Sleep until an absolute deadline.
`zx_deadline_after`	Compute an absolute deadline from a duration.
`zx_timer_create`	Create a one-shot timer object.
`zx_timer_set`	Arm a timer with a deadline and slack.
`zx_timer_cancel`	Disarm a timer.
`zx_clock_create`	Create a user-space synthetic clock.
`zx_clock_read / update / get_details`	Read/steer a synthetic clock.
`zx_ticks_get`	Read the raw hardware tick counter.
Interrupts and exceptions	
`zx_interrupt_create`	Create an interrupt object bound to a vector (needs IRQ resource).
`zx_interrupt_bind`	Bind an interrupt to a port for async delivery.
`zx_interrupt_wait`	Block until the interrupt fires.
`zx_interrupt_ack`	Acknowledge a level-triggered interrupt to re-arm.
`zx_interrupt_trigger`	Trigger a virtual interrupt (testing).
`zx_interrupt_destroy`	Tear down an interrupt object.
`zx_msi_allocate`	Allocate a block of MSI vectors (PCI).
`zx_msi_create`	Create an interrupt object for one MSI vector.
`zx_exception_get_thread`	Get the faulting thread from an exception.
`zx_exception_get_process`	Get the faulting process from an exception.
Hardware: resource, BTI, PMT, IOMMU, MMIO	
`zx_resource_create`	Carve a child resource for a sub-range/kind.
`zx_iommu_create`	Model a platform (or dummy) IOMMU (root resource).
`zx_bti_create`	Create a Bus Transaction Initiator from an IOMMU.
`zx_bti_pin`	Pin VMO pages; return device-physical addresses + a PMT.
`zx_bti_release_quarantine`	Release pages quarantined by an improper unpin.
`zx_pmt_unpin`	Unpin a pinned-memory token, freeing the pages.
`zx_smc_call`	Issue an ARM secure-monitor call (SMC resource).
System, virtualization, restricted mode, diagnostics	
`zx_system_get_num_cpus`	Number of logical CPUs.
`zx_system_get_physmem`	Total physical memory.
`zx_system_get_version_string`	Kernel/system version string.
`zx_system_powerctl`	Reboot, shutdown, or change CPU power state.
`zx_system_mexec`	Kexec-style hand-off to a new kernel image.
`zx_guest_create`	Create a guest (guest-physical address space).
`zx_vcpu_create`	Create a virtual CPU within a guest.
`zx_vcpu_enter`	Run the VCPU until a VM-exit packet is returned.
`zx_vcpu_kick`	Force a VCPU to exit `zx_vcpu_enter`.
`zx_restricted_bind_state`	Map the shared restricted-mode register state VMO.
`zx_restricted_enter`	Enter restricted mode running guest/foreign code.
`zx_debuglog_create`	Open a handle to the kernel debug log (klog).
`zx_debuglog_write / read`	Append to / drain the kernel debug log.
`zx_ktrace_control`	Start/stop/rewind kernel tracing.

XXIV. Appendix B: Glossary

KOID Kernel Object Identifier: a globally unique, never-reused 64-bit number naming a kernel object for the lifetime of the boot. Distinct from a handle, which is a per-process *reference* to the object.

Handle An unforgeable, process-local reference to a kernel object, carrying a rights mask. Possession of a handle is the authority to act on the object; the kernel translates handle values to objects on every syscall.

Dispatcher The in-kernel C++ base class behind every object class (`VmoDispatcher`, `ChannelDispatcher`, ...). It owns 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`, `DUPLICATE`, `TRANSFER`, `EXECUTE`, ...) that gates which operations the handle authorizes. Rights can be narrowed on duplication but never widened.

VMO Virtual Memory Object: a kernel-managed, resizable container of pages—the unit of memory in Zircon. 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; the kernel side 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, backed by per-page bookkeeping and the page queues.

Physmap A permanent kernel mapping of all (or most) physical RAM at a fixed virtual offset, giving the kernel O(1) access to any physical page without per-use mapping.

Pager-backed VMO A VMO whose pages are supplied on fault by a *user-space* pager over a port, enabling demand paging, file caching, and custom eviction outside the kernel.

PMT Pinned Memory Token: the object returned by `zx_bti_pin` that represents a specific 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 representing one device's view of memory through the IOMMU. Pinning a VMO against a BTI yields the bus addresses the device may DMA to and nothing else.

IOMMU Input/Output Memory Management Unit: the hardware (or a kernel "dummy" stand-in) that translates and bounds device DMA. Zircon programs it so a device can reach only its pinned pages.

vDSO Virtual Dynamic Shared Object: the kernel-provided, read-execute shared library that is the *only* legal entry point to syscalls; the kernel checks that the trap PC lies within the vDSO image.

ZBI Zircon Boot Image: the bootloader-delivered container of typed items (kernel image, command line, `bootfs` ramdisk, memory map, platform id, ...) consumed by early boot.

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

processargs The bootstrap-channel message protocol by which a new process receives its initial handles (each tagged with a `PA_*` type), arguments, environment, and namespace—the kernel/user-space handoff of authority.

Job policy Inherited sandboxing rules (`zx_job_set_policy`) attached to a job that constrain its descendants—which object classes they may create, whether W^X is allowed, and how violations are handled. Policy can only tighten down the tree.

Restricted mode A per-thread execution mode with a separate register set and a restricted view, used to run untrusted or foreign-ABI code (notably Linux binaries under Starnix); a syscall or fault bounces control back to the host handler.

Futex Fast Userspace muTEX: a 32-bit user word with kernel-assisted sleep/wake (`zx_futex_wait/wake`), the substrate for user-space mutexes and condition variables with no syscall on the fast path.

Port packet A fixed-size record dequeued from a port, tagging the source (signal, interrupt, user, page request, guest exit) and carrying its payload; the unit of unified asynchronous notification.

Epitaph A final status value sent on a channel just before its server end closes, conveying *why* the connection ended to the surviving peer.

Deadline scheduling The kernel's bandwidth/latency contract for a thread, expressed as (capacity, deadline, period); deadline threads are admitted ahead of fair-weighted threads when runnable.

OwnedWaitQueue The in-kernel wait-queue type that tracks an owning thread to support priority inheritance, so a blocked high-importance waiter can lend its scheduling weight to the holder it is waiting on.

TLB shootdown The cross-CPU protocol that invalidates stale translation-lookaside-buffer entries on other cores after a mapping changes, ensuring 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/reference/kernel_objects/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, “Restricted Mode and Starnix,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/kernel/restricted_mode

[19] Google, “Booting 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.