# capOS System Manual

System documentation for building, booting, operating, and reviewing the current capOS implementation.


| | |
|---|---|
| **Version** | 08dffa7 |
| **Generated** | 2026-06-11 |
| **Source** | 08dffa7 |
| **Scope** | Current system behavior, operator workflow, architecture, and security review boundaries. Project archives remain on the mdBook site and are not part of this PDF manual. |

Generated from the capOS documentation source.

# Contents

Part I: Orientation ..... 3
- Introduction ..... 4
- What capOS Is ..... 5
- Current Status ..... 7
- Build, Boot, and Test ..... 8
- Benchmarks ..... 17
- Configuration ..... 23
- Repository Map ..... 29
- Programming Languages ..... 43

Part II: Operator Demos ..... 51
- First Chat Demo ..... 52
- Aurelian Frontier — Proof Slice ..... 54
- Paperclips Terminal Demo ..... 60

Part III: Architecture ..... 67
- Current Design Authority ..... 68
- Boot Flow ..... 70
- Manifest and Service Startup ..... 74
- Process Model ..... 78
- Session Context ..... 81
- In-Process Threading Contract ..... 84
- Park Authority Contract ..... 95
- Capability Model ..... 105
- ABI Evolution Policy ..... 117
- Capability Ring ..... 121
- Error Handling ..... 128
- IPC and Endpoints ..... 131
- Authority Graph and Resource Accounting for Transfer ..... 134
- Userspace Runtime ..... 139
- Memory Management ..... 143
- Scheduling ..... 147

Part IV: Security and Verification ..... 172
- Security Verification Track Registry ..... 173
- Trust Boundaries ..... 174
- Verification Workflow ..... 179
- Trusted Build Inputs ..... 185
- Panic-Surface Inventory ..... 222
- DMA Isolation Design ..... 229

Appendix: Risk Register ..... 289
- Design Risks and Open Questions Register ..... 290

# Part I: Orientation

What capOS is, what works today, and how to build and boot the current system.

# Introduction

This book is the system manual for the current capOS implementation. It covers the implemented operating model, build and boot workflow, runnable demos, architecture, configuration surface, and security and verification boundaries.

capOS is a research operating system where kernel and userspace services are typed Cap'n Proto capabilities invoked through shared-memory rings. The manual focuses on behavior that exists or is directly reviewable in this repository; project plans, proposals, and research notes remain available as archives rather than driving the primary reading path.

## What capOS Is

A research kernel that boots on x86_64 QEMU. The rest of this page is about why it looks the way it does — the specific design bets behind the code — not a feature inventory. For the feature-by-feature matrix, see Current Status.

### What Makes capOS Different

capOS is a research vehicle for a few specific design bets. Each is unusual on its own; the combination is the point.

- **Everything is a typed capability.** System resources are accessed through Cap'n Proto interfaces defined in `schema/capos.capnp`. There is no ambient authority — no global path namespace, no open-by-name, no implicit inherit. A process can only invoke objects present in its local capability table. See Capability Model and the schema/repo map.
- **The interface IS the permission.** Instead of a parallel READ/WRITE/EXEC rights bitmask (Zircon, seL4), attenuation is a narrower capability: a wrapper `CapObject` exposing fewer methods, or an `Endpoint` client facet that cannot `RECV`/`RETURN`. The kernel just dispatches; policy lives in interfaces. See Capability Model, IPC and Endpoints, and the prior-art notes on Zircon and seL4.
- **Identity metadata is not authority.** In prose, a user is the human-facing actor, a principal is identity metadata, an account is planned durable local record state, and policy/resource profiles select bundles and quotas. Sessions receive capabilities; none of those labels become kernel subjects or bypass cap-table authority. See the local users backlog, User Identity and Policy, and Resource Accounting and Quotas.
- **io_uring-style shared-memory ring for every call.** Every process owns a submission/completion queue page. Userspace writes SQEs with a normal memory store; the kernel processes them through `cap_enter`. New operations are SQE opcodes (`CALL`, `RECV`, `RETURN`, `RELEASE`, `NOP`), not new syscalls. The remaining syscall surface is `cap_enter` and `exit`; the accepted threading contract keeps current-thread exit as a `ThreadControl` capability operation. See Capability Ring, Userspace Runtime, and In-Process Threading.
- **Release is transport, not an application method.** Dropping the last owned handle in `capos-rt` queues one local `CAP_OP_RELEASE`; acquiring or dropping a runtime ring client flushes the queue, and long-running code can call `Runtime::flush_releases()` explicitly. No `close()` method on every interface, no mutable table self-reference during dispatch. See Userspace Runtime and Capability Ring.
- **Capability transfer is first-class.** Copy and move descriptors ride sideband on `CALL`/`RETURN` SQEs. Move reserves the sender slot until the receiver accepts and preflight checks pass, then commits or rolls back atomically — no lost, duplicated, or half-inserted authority. See Authority Accounting and IPC and Endpoints.
- **Cap'n Proto wire format end-to-end.** The same encoding describes the boot manifest, runtime method calls, and future persistence/remote transparency. The debug tap records fixed, bounded SQE/CQE metadata today; authorized payload capture, replay, audit, and migration remain future transport work. See Manifest and Service Startup, Error Handling, and Storage and Naming.

- **Host-testable pure logic.** Cap-table, frame-bitmap, ELF parser, frame ledger, lazy buffers, small ABI constants, and the ring model live in `capos-lib`, `capos-abi`, and `capos-config`, and run under `cargo test-lib`, Miri, Loom, Kani, and `proptest` without any kernel scaffolding. Kernel glue stays thin. See Verification Workflow and Repository Map.
- **Schema-first boot.** `system.cue` is compiled to a Cap'n Proto `SystemManifest` embedded as the single Limine boot module. The kernel validates only the kernel-owned boot boundary and launches `initConfig.init`; `mkmanifest` and init validate the service graph under `initConfig.services` as structured data, not shell scripts or baked environment variables. See Boot Flow, Manifest and Service Startup, and Build, Boot, and Test.

### Execution Model

Each process owns an address space, a local capability table, a mapped capability-ring page, and a read-only CapSet page that enumerates its bootstrap handles. The kernel enters Ring 3 with `iretq` and returns through `cap_enter` or the timer. Ordinary capability calls progress only via `cap_enter`; timer-side polling handles non-CALL ring work and call targets that are explicitly safe for interrupt dispatch. Details in Process Model, Capability Ring, In-Process Threading, and Scheduling.

### Boot Flow

The kernel receives exactly one Limine module — a Cap'n Proto `SystemManifest` compiled from `system.cue` — validates the kernel-owned boot boundary, loads only `initConfig.init.binary`, builds that process's bootstrap capability table and CapSet page from `initConfig.init.caps`, and starts the scheduler. The default manifest now boots the standalone `init` ELF, and init validates the service graph before spawning the foreground `capos-shell`, the remote-session CapSet gateway, and the resident demo services. The shell mints an anonymous `UserSession` when it starts and the user runs `login` or `setup` as ordinary shell commands to upgrade to an operator session. Focused shell-led manifests such as `system-smoke.cue` and `system-shell.cue` still boot `capos-shell` directly as `initConfig.init` until the run-target/init-policy cleanup migrates them. Full walkthrough in Boot Flow and Manifest and Service Startup.

### Authority Boundaries

Authority is carried by cap-table hold edges with generation-tagged `CapIds`. Ring 0 ↔ Ring 3, capability table ↔ kernel object, endpoint IPC, copy/move transfer, manifest/boot-package, and process spawn are the boundaries reviewers care about; each one fails closed at hostile input. See Trust Boundaries for the boundary table and Authority Accounting for the transfer and quota invariants.

### What capOS Is Not

A POSIX clone, a microkernel-shaped Linux replacement, or a production OS. It is a place to try the above choices and see which ones survive contact with real workloads. See Build, Boot, and Test to run it.

## Current Status

capOS currently boots on x86_64 QEMU, starts a standalone init process from the default manifest, and runs the native shell plus resident demo services through typed capabilities. The operator path starts as an anonymous shell session; `login` validates the selected bootstrap account through `SessionManager` and `CredentialStore`, then upgrades the session through `AuthorityBroker`.

Implemented pieces include isolated processes, user-mode ELF loading, shared-memory capability rings, endpoint IPC, copy/move capability transfer, thread and private ParkSpace primitives, init-owned service spawning, local shell login, host-local Telnet shell wiring, a local Phase C userspace smoltcp proof that serves `TcpListenAuthority` / `TcpListener` / `TcpSocket` caps, resident chat/adventure services, and the Paperclips terminal demo.

The current selected milestone is GCE Self-Hosted Web UI. Its path is staged: the local userspace network-stack socket proof has landed, while DHCP/IPv4 configuration, the remote-session Web UI over the Phase C L4 path, private GCE reachability, and public ingress/TLS remain separate evidence levels. Public ingress and TLS are on hold for explicit authorization; the legacy in-kernel socket path is transitional rather than the production L4 direction.

Partially implemented areas also include local account storage, the SSH gateway proof path, and the single-CPU thread/park runtime surface. Durable multi-account credential storage, production SSH ingress, userspace DMA drivers, broader persistence, and broader supervision policy remain future work.

The detailed website Current Status page carries milestone commit history, recent notes, and validation pointers. This manual keeps the status summary short so build, boot, architecture, and security review material stay near the front.

## Build, Boot, and Test

The commands below are the current local workflow for the x86_64 QEMU target. The root Cargo configuration defaults to `x86_64-unknown-none`, so host tests must use the repo aliases instead of bare `cargo test`.

### Prerequisites

Expected host tools:

- Rust nightly from `rust-toolchain.toml`
- `make`
- `qemu-system-x86_64`
- `xorriso`
- `curl`, `sha256sum`, and standard build tools for pinned tool downloads
- Go, used by the Makefile to install the pinned CUE compiler when needed
- A Telnet client for the optional focused loopback shell demo
- Chromium, Chromium Browser, or Google Chrome for the optional remote-session CapSet browser UI automation
- Optional policy and proof tools for extended checks: `cargo-deny`, `cargo-audit`, `cargo-fuzz`, `cargo-miri`, and `cargo-kani`

The Makefile pins and verifies:

- Limine at the commit recorded in `Makefile`
- Cap'n Proto compiler version `1.2.0`
- CUE version `0.16.0`

Pinned repo-selected tools are installed under `CAPOS_TOOLS_ROOT`, which defaults to the per-user `$HOME/.capos-tools` cache. Override `CAPOS_TOOLS_ROOT=/path/to/cache` when a host needs a different cache location.

### Build the ISO

Use the default target when you need the current bootable capOS image.

```
make
```

This builds:

- the kernel with the default bare-metal target;
- the standalone `init` userspace binary used by focused spawn proofs;
- release-built demo service binaries under `demos/`;
- the `capos-rt` userspace binaries, including the shell proof;
- `manifest.bin` from `system.cue`;
- `capos.iso` with Limine boot files.

Relevant files: `Makefile`, `limine.conf`, `system.cue`, `tools/mkmanifest/`.

### Compare Build Provenance

Use `make build-provenance` to write the local build record at `target/build-provenance.txt`. To compare two retained records locally:

```
make build-provenance-compare \
  BASE_PROVENANCE=path/to/base-build-provenance.txt \
  CANDIDATE_PROVENANCE=path/to/candidate-build-provenance.txt
```

The comparison ignores the generated timestamp and allowed local path-root movement under worktree `target/` directories or `.capos-tools/` caches. It fails for material provenance drift, including source commit changes, manifest or artifact hash changes, embedded binary hash changes, OVMF identity/hash changes, Rust compiler date/commit changes, host-tool version or package identity changes, and operating-system identity changes.

For PR base-vs-head CI comparison, use the environment policy:

```
make build-provenance-compare \
  BUILD_PROVENANCE_COMPARE_POLICY=ci-environment \
  BASE_PROVENANCE=path/to/base-build-provenance.txt \
  CANDIDATE_PROVENANCE=path/to/candidate-build-provenance.txt
```

That mode allows expected source and artifact hash changes while still failing for runner, GitHub-hosted image, Rust, selected-tool, package-identity, OVMF selection, and OVMF hash drift. It is a PR environment gate, not a production reproducibility claim.

Local synthetic comparison checks may create scratch records under `target/provenance-fixtures/` or Python bytecode caches under `tools/`. Clean those scratch artifacts with:

```
make build-provenance-compare-clean
```

### Boot QEMU

Use the default run targets to boot either the operator-facing system or the scripted login-path smoke.

```
make run
make run-smoke
```

`make run` is the operator-facing boot path. It builds the ISO with the `qemu` feature, boots QEMU with the interactive terminal UART on stdio, attaches virtio-net with host-local remote CapSet forwarding, and writes the separate kernel/debug UART log to `target/qemu-console.log`. The run output prints the actual forwarded port as `remote CapSet: tcp 127.0.0.1 <port> -> guest :2327`.

The plaintext loopback Telnet research demo was a Phase B fixture, not part of the default operator path. `make run-telnet` and `make run-telnet-vm` are now retired because they depended on the removed qemu-only kernel TCP listener. Use the SSH gateway smokes for current remote-shell coverage and rebuild any socket-backed terminal proof on the Phase C userspace network stack before using it as validation.

The same `make run` boot starts the remote-session CapSet gateway. To run the host CLI against it, use the printed port:

```
cargo run --manifest-path tools/remote-session-client/Cargo.toml \
  --target x86_64-unknown-linux-gnu \
  --bin remote-session-client -- --host 127.0.0.1 --port <port>
```

Add `--launch-adventure` to that command when you want the CLI to start the default-manifest Adventure service graph through `serviceLaunch` and require a `running` status.

To run the trusted local web bridge against the same QEMU instance:

```
CAPOS_REMOTE_SESSION_PORT=<port> make remote-session-ui
```

Then open `http://127.0.0.1:3337/`. The Rust bridge holds the TCP stream, remote session state, and backend-held remote CapSet; the browser receives only view models, launch/status descriptors, denial diagnostics, call results, and redacted transcript rows. The former automated focused proof, `make run-remote-session-capset-ui`, is retired because it depended on the removed qemu-only kernel TCP listener. The replacement browser proof belongs to the future Phase C Web UI L4 gate.

A Tauri desktop wrapper is available as a repo-local check/dev layer over the same Rust backend. The repo-local `make remote-session-tauri` target first runs a policy preflight over the reviewed scaffold, then checks for the Tauri CLI and Linux build prerequisites, reports dependency and scaffold status, and runs a deterministic wrapper check when those prerequisites are present. It follows the official Tauri v2 Linux prerequisite shape, including WebKitGTK 4.1, `libxdo`, OpenSSL, AppIndicator, and Rsvg development packages where applicable. Missing dependencies fail with explicit diagnostics and point back to the supported local web bridge. The operator command shape is:

```
CAPOS_REMOTE_SESSION_PORT=<port> make remote-session-tauri
```

Set `CAPOS_REMOTE_SESSION_TAURI_MODE=dev` to launch `cargo tauri dev`. `CAPOS_REMOTE_SESSION_TAURI_MODE=policy tools/remote-session-tauri.sh` runs only the scaffold guardrail and does not require Tauri system packages or a desktop session. `CAPOS_REMOTE_SESSION_TAURI_MODE=package` and `CAPOS_REMOTE_SESSION_TAURI_MODE=automation` are intentionally blocked with diagnostics describing the remaining packaging and desktop-automation review work. This policy preflight proves only that the current wrapper remains a check/dev scaffold with packaging disabled, the loopback URL pinned, a single `main` window, default `core:default` permission scope, and no app-specific Tauri command/plugin authority. It is not a distributable packaging or desktop automation proof. `make remote-session-ui` remains the supported fallback host UI path and uses the same backend-held authority boundary.

Default `make run` starts chat, the remote-session gateway, and shell services. It embeds Adventure server/NPC/client binaries and the terminal Paperclips binary. The current remote-session Adventure slice makes `serviceLaunch` a real restricted backend launch in that default manifest: the trusted backend/gateway starts `adventure-server` plus simple NPC companions

through an approved service-runner profile and attaches or retains backend-held descriptors/caps for the Adventure/chat-facing services. Run it by starting `make run`, noting the printed remote CapSet forwarding port, and then using either the host CLI or `CAPOS_REMOTE_SESSION_PORT=<printed-port> make remote-session-ui`. `make run-paperclips` remains the focused authoritative Paperclips server proof; default-manifest Paperclips launch is not implemented by this slice. Raw `ProcessSpawner`, process owner handles, endpoint owner caps, local cap ids, result-cap slots, and browser-held capOS caps are non-goals for this UI path. Process handles stay backend-local.

**GCE Web UI Proof Target Map**

Use the selected-milestone proof targets below to choose the narrowest evidence gate for the GCE self-hosted Web UI ladder. Local QEMU/cloudboot targets do not prove live provider reachability, and private GCE targets do not authorize public ingress or TLS exposure.

- **Proof class:** Landed local Phase C L4 substrate
  - **Target or command shape:** `make run-cloud-prod-userspace-network-stack-smoltcp`
  - **Proves:** A non-`qemu` cloudboot kernel under QEMU starts the userspace smoltcp network-stack process and completes one hostfwd TCP request/response through a userspace-served `TcpListenAuthority`. See `cloud-prod-userspace-network-stack-smoltcp-local-proof`.
  - **Closest non-goal:** Does not prove DHCP/IPv4 configuration, `remote-session-web-ui`, live GCE reachability, or public ingress.
- **Proof class:** Landed local IPv4 configuration
  - **Target or command shape:** `make run-cloud-prod-network-stack-dhcp-ipv4-config`
  - **Proves:** The userspace network-stack process acquires the QEMU SLIRP DHCPv4 lease, serves `NetworkManager.getConfig`, installs the default route, and resolves gateway plus same-subnet ARP neighbors. See `cloud-prod-network-stack-dhcp-ipv4-config-local-proof`.
  - **Closest non-goal:** Does not prove a Web UI listener bound through that route, live GCE reachability, DNS, TLS, or public exposure.
- **Proof class:** Retired legacy local self-served Web UI
  - **Target or command shape:** `make run-remote-session-self-served-web-ui`
  - **Proves:** Pre-Phase-C proof that served the immutable full UI bundle from a focused QEMU manifest through the kernel `tcp_listen_authority` socket owner. The target is not current production L4 evidence after `cloud-prod-phase-c-kernel-smoltcp-virtio-net-removal` retires that kernel owner.
  - **Closest non-goal:** Does not prove the non-`qemu` cloudboot Phase C L4 path, and should not be used as a passing selected-milestone gate until rebuilt on the userspace network-stack substrate.
- **Proof class:** Landed cloudboot Web UI authority inventory
  - **Target or command shape:** No run target; docs-status contract
  - **Proves:** The Gate 1B inventory records the required and forbidden `remote-session-web-ui` grants, trusted listener/source metadata, browser-visible forbidden markers, and expected local L4 proof markers. See `remote-session-webui-cloudboot-authority-inventory`.

  - **Closest non-goal:** Does not prove runtime listening, browser automation, GCE reachability, or public operator access.
- **Proof class:** Landed local cloudboot Web UI L4 proof
  - **Target or command shape:** `make run-cloud-prod-remote-session-web-ui-l4` owned by `cloud-prod-remote-session-web-ui-l4-local-proof`
  - **Proves:** Proves `remote-session-web-ui` listens on guest port 8080 through the Phase C L4 path on the non-`qemu` cloudboot kernel under QEMU: the userspace network-stack process serves the scoped `TcpListenAuthority`, the Web UI serves the full fixed-name bundle, login, one backend-held capability call, logout, stale-call failure, the manual viewer, and a `cloudboot-evidence: remote-session-web-ui-l4` marker.
  - **Closest non-goal:** Local cloudboot/QEMU evidence only; it does not prove live GCE NIC reachability, private provider probing, public ingress, TLS, or production release authority.
- **Proof class:** On-hold private GCE Web UI proof
  - **Target or command shape:** Future `tools/cloudboot/run-test.sh --require-web-ui-proof` gate owned by `cloud-gce-private-self-hosted-webui-proof`
  - **Proves:** Must launch the self-hosted Web UI cloudboot image in the no-public-IP GCE posture, use a reviewed private probe that crosses the live GCE virtual network boundary, record the private endpoint and Web UI/L4 markers, and tear down all created resources.
  - **Closest non-goal:** On hold (2026-06-09): the cloudtest credential lacks the firewall IAM a private same-VPC probe needs against GCE default-deny ingress, and the live legacy virtio 0.9 GCE NIC has no reviewed userspace-stack serving story. It will not create public IPs, public firewall rules, DNS, TLS certificates, or browser-facing public operator ingress.
- **Proof class:** On-hold public ingress/TLS proof
  - **Target or command shape:** Future `tools/cloudboot/run-test.sh --require-public-web-ui-proof` gate owned by `cloud-gce-public-self-hosted-webui-ingress-tls`
  - **Proves:** After explicit authorization, must prove the selected GCE external HTTPS load-balancer ingress posture, Google-managed certificate termination, browser-session hardening, and teardown evidence.
  - **Closest non-goal:** On hold. No local target, private proof, or selected milestone status grants public exposure, broad firewall changes, certificate issuance, TLS key custody, or release authority.

`make run-smoke` preserves the focused legacy shell-led `system-smoke.cue` verification path. It drives the login and shell session through the terminal UART, captures the kernel log and terminal transcript separately, and checks that the kernel boot-launched only the first `init` service (`capos-shell`), granted only the shell bootstrap cap bundle, and then reached the expected audit, shell-bundle, child-isolation, stale-handle, and no-password-echo assertions before QEMU exits. This is distinct from the default `system.cue` path, where the kernel boot-launches standalone `init` and init starts operator-facing services.

### Spawn Smoke

Use the spawn smoke when changes affect manifest-owned process creation, ProcessSpawner behavior, or bootstrap capability wiring.

```
make run-spawn
```

This boots with `system-spawn.cue`, the focused init-owned manifest retained for ProcessSpawner checks. Only `init` is boot-launched by the kernel; `init` uses ProcessSpawner to launch endpoint, IPC, VirtualMemory, Timer, ThreadControl, the single-thread runtime checkpoint, FrameAllocator cleanup, and hostile spawn demo children, wait for ProcessHandles, and exercise hostile spawn inputs. The target captures the kernel log separately and runs `tools/qemu-spawn-smoke.sh` to assert the single-init boot markers, BootPackage validation, child spawn/exit records, Timer now/sleep and per-process sleep quota proof lines, runtime FS-base proof lines, the single-thread runtime map/protect/unmap plus park-fallback checkpoint, manifest child waits, and clean halt.

### Shell and Terminal Smokes

Use these focused QEMU smokes for shell, terminal, credential, and login paths.

```
make run-shell
make run-terminal
make run-credential
make run-login
make run-login-setup
```

- `make run-shell` boots the focused `system-shell.cue` manifest (no pre-provisioned verifier) and exercises the shell entirely in its anonymous session: CapSet listing, typed capability inspection, typed application-error display, anonymous-session metadata, the anonymous launcher rejecting `spawn-test` because its allowlist is empty, and clean exit.
- `make run-terminal` boots the focused `system-terminal.cue` manifest and exercises the `TerminalSession` substrate: visible and hidden echo input, bounded `readLine`, structured cancellation, and stale-input scrubbing between prompts.
- `make run-credential` boots the focused CredentialStore proof manifest.
- `make run-login` boots the focused password-login manifest and proves the shell's `login` command prompting for `username>` before hidden `password>`, failing generically on a wrong password, succeeding for the demo account, swapping from the anonymous bundle to the operator bundle, and performing exact-grant child launch plus stale-handle release.
- `make run-login-setup` boots the no-password first-boot setup manifest and proves that `setup` creates a volatile credential, discloses that volatility, chains into the login upgrade path, and reaches the same narrow operator shell bundle.

Durable account storage and multi-verifier local accounts are still future work; the current username-aware login path selects the manifest-seeded operator-kind account and any volatile first-boot credential record that `setup` creates.

### Focused Service Smokes

Use these targets to prove resident services and demo clients still launch through the intended shell-granted authorities.

```
make run-chat
make run-adventure
make run-paperclips
make run-revocable-read
make run-memoryobject-shared
make run-ringtap-failing-call
```

- `make run-chat` boots the focused First Chat manifest and proves a shell-spawned client can send a line through the resident singleton chat service using the broker-issued operator `chat` endpoint and observe the resident bot reply.
- `make run-adventure` boots the focused adventure manifest and proves the shell-spawned client can drive the current scripted mission through explicit `StdIO`, `adventure`, and `chat` endpoint grants.
- `make run-paperclips` boots the focused Paperclips terminal demo manifest, authenticates the shell, starts Paperclips server services, first launches the clean-room terminal client with explicit `StdIO` plus the normal `PaperclipsGame` endpoint, proves normal server authority cannot invoke `run <ms>`, rejects a forged `proof_accelerator: @timer` grant, then relaunches against the proof server endpoint with the explicit `proof_accelerator` proof authority for the accelerated transcript. The server owns generated content, game state, regular timer cadence, unlock checks, and game-rule mutation, and server-mode client help is rendered from structured server command specs. That transcript rejects an early locked autoclipper purchase, rejects an over-budget wire purchase, rejects bulk manual production, rejects a high-price sale with zero current demand, rejects manual production after automation drains wire, drives one-at-a-time manual production, explicit sales, repeatable marketing, autoclipper unlock, real-time automation, generated typed Cap'n Proto content loading, scaled business-phase production, `precision-rollers`, `design-search`, `forecast-engine`, the `survey-drones` transition to `== autonomous phase ==`, representative autonomous drone/factory scaling with local-matter conversion and additional clip production, the `mesh-coordination` and `seed-probes` cosmic transition, bounded probe replication and production, locked `final-conversion`, and clean client/shell exit.
- `make run-revocable-read` exercises the revocation transcript for endpoint and boot-package authority loss.
- `make run-memoryobject-shared` proves MemoryObject-backed parent/child sharing and cleanup.
- `make run-ringtap-failing-call` enables `debug_tap`, drives a known typed launcher failure, and runs the ringtap viewer over the captured log.

### Networking and Measurement Targets

Use these targets for the current network proof path and benchmark-only measurement image.

```
make run-net
make qemu-net-harness
make run-measure
```

- `make run-net` attaches a QEMU virtio-net PCI device and exercises current PCI enumeration, virtio transport setup, and TX descriptor completion diagnostics, plus ARP resolution and ICMP echo validation against the QEMU user-mode gateway.
- `make qemu-net-harness` runs the scripted net smoke path.
- `make run-measure` enables the separate `measure` feature for benchmark-only counters and cycle measurements. It boots `system-measure.cue`, where init spawns `ring-nop` and grants the measurement-only NullCap and ParkBench caps through ProcessSpawner. The demo prints ring/NullCap baselines plus a park-shaped comparison between compact authority-checked SQEs and generic Cap'n Proto methods. The kernel summary includes per-segment dispatch counts, total cycles, and averages for SQE processing, validation, cap lookup, capnp decode, method body dispatch, CQE posting, and waiter wake/check. Do not treat it as the normal dispatch build.

### Formatting and Generated Code

Use these local checks before claiming source formatting or generated artifacts are current.

```
make fmt
make fmt-check
make generated-code-check
```

- `make fmt` formats the kernel workspace plus standalone `init`, `demos`, and `capos-rt` crates.
- `make fmt-check` verifies formatting without modifying files.
- `make generated-code-check` verifies checked-in Cap'n Proto generated code against the repo-pinned compiler path and checks generated adventure plus Paperclips content against their CUE sources.

### Host Tests

Use these host-side checks for shared logic and userspace build surfaces that do not require a QEMU boot.

```
cargo test-config
cargo test-ring-loom
cargo test-lib
cargo test-mkmanifest
tools/check-userspace-runtime-surface.sh
make capos-rt-check
make init-capos-build
make demos-capos-build
make shell-capos-build
make capos-rt-capos-build
```

- `cargo test-config` runs shared config, manifest, ring, and CapSet tests on the host target.
- `cargo test-ring-loom` runs the bounded Loom model for SQ/CQ protocol invariants.
- `cargo test-lib` runs host tests for pure shared logic such as ELF parsing, capability tables, frame allocation, and related property tests.

- `cargo test-mkmanifest` runs host tests for manifest generation.
- `tools/check-userspace-runtime-surface.sh` verifies `capos-rt` owns the userspace entry, panic, allocator, and raw syscall surface.
- `make capos-rt-check` builds the standalone runtime smoke binary against `targets/x86_64-unknown-capos.json`, matching the userspace target used by the boot image.
- `make init-capos-build`, `make demos-capos-build`, `make shell-capos-build`, and `make capos-rt-capos-build` expose focused custom-target build wrappers for the booted userspace crates and runtime smoke binary.

### Extended Verification

Use the extended verification set for shared logic, dependency policy, fuzz targets, and bounded proof gates that are heavier than the normal host-test loop.

```
make dependency-policy-check
make fuzz-build
make fuzz-smoke
make kani-lib
cargo miri-lib
```

These require optional tools. Use them when changing dependency policy, manifest parsing, ELF parsing, capability-table/frame logic, or proof-covered shared code. `make dependency-policy-check` covers Rust deny/audit checks and the docs Node lockfile/audit gate with npm lifecycle scripts disabled. See the Security and Verification Proposal for the rationale behind the extended verification tiers. `make kani-lib` runs the bounded mandatory cap-table/frame gate.

### Validation Rule

For behavior changes, a clean build is not enough. The relevant QEMU process must exercise the behavior and print observable output that proves the path works. `make run-smoke` is the default login-path gate; `make run-spawn`, `make run-shell`, `make run-terminal`, `make run-credential`, `make run-login`, `make run-login-setup`, `make run-chat`, `make run-adventure`, `make run-paperclips`, `make run-revocable-read`, `make run-memoryobject-shared`, `make run-net`, `make qemu-net-harness`, `make run-ringtap-failing-call`, or `make run-measure` are additional gates for their specific features.

## Benchmarks

capOS benchmark rows are evidence records. Each row should say what workload ran, what was verified, how time was measured, what machine envelope was used, and where the raw artifacts were stored. A faster row whose verifier did not complete is not a performance result.

The broader benchmark model is in System Performance Benchmarks. Future parallel-pattern coverage is in HPC Parallel Processing Patterns.

### Current CPU Workloads

capOS currently has two CPU-scaling workloads:

- **Workload:** `run-smp-process-scale`
  - **Target:** Independent worker processes
  - **Timed region:** worker compute only, after setup and before result reporting
  - **Verifier:** aggregate prime count and checksum
  - **Primary use:** Exercises multiple process-owned rings running CPU work on more than one scheduler CPU.
- **Workload:** `run-thread-scale`
  - **Target:** Sibling threads in one process
  - **Timed region:** checksum work window, separate from spawn/join/shutdown totals
  - **Verifier:** deterministic root checksum and metadata checks
  - **Primary use:** Measures same-process thread scheduling, per-thread rings, and scheduler overhead.

Both workloads keep serial and harness artifacts under `target/`. The capOS rows below were collected under QEMU/KVM. The matching Linux rows use the same workload shape where possible, but units differ by harness and should not be compared directly across systems. Compare speedup ratios within a row.

### Process-Scale SMP

`make run-smp-process-scale` boots a focused manifest, runs independent worker processes, and times the CPU-bound worker window. Each worker owns its own process ring. The timed section avoids syscalls and serial output; the coordinator verifies the aggregate result after workers finish.

The current workload counts primes over `2..3_000_000` using balanced contiguous splits. capOS reports a worker-side user-mode cycle counter shifted right by 20 bits. Linux reports guest `clock_gettime` nanoseconds.

Controlled benchmark-VM reruns were recorded on GCE `n2-highcpu-8` at capOS commit `0d89a91b` (`2026-04-30 11:09 UTC`) with nested QEMU/KVM on Ubuntu `6.17.0-1012-gcp`, QEMU `8.2.2`, Rust nightly `1.97.0-nightly` (`c935696dd 2026-04-29`), and host logical CPUs `0,1,2,3` mapped to distinct physical cores with SMT siblings `4,5,6,7`.

| System | smp1 median | smp2 median | smp4 median | 1-to-2 | 1-to-4 |
|---|---|---|---|---|---|
| capOS | 1,639 scaled cycles | 875 scaled cycles | 1,111 scaled cycles | 1.873x | 1.475x |

| System | smp1 median | smp2 median | smp4 median | 1-to-2 | 1-to-4 |
|---|---|---|---|---|---|
| Linux | 1,275,187,210 ns | 659,218,025 ns | 337,877,986 ns | 1.934x | 3.774x |

The capOS 4-vCPU row improved over the 1-vCPU row but was slower than the 2-vCPU row. Linux continued improving through 4 vCPUs under the same pinning and workload. Raw capOS artifacts are under `target/smp-process-scale/pinned-20260430T1113Z/`; raw Linux artifacts are under `target/linux-smp-process-scale/pinned-20260430T1118Z/`.

### SMT Run

The same harness can run an eight-logical-CPU case on the benchmark VM. That machine exposes four physical cores and eight SMT threads, so the `smp8-smt` row is an SMT measurement on a 4-core host.

The SMT run was recorded at commit `7c15dd47` (`2026-04-30 11:45 UTC`) with QEMU pinned to logical CPUs `0,1,2,3,4,5,6,7`.

| System | smp1 median | smp2 median | smp4 median | smp8-smt median |
|---|---|---|---|---|
| capOS | 1,500 scaled cycles | 787 scaled cycles | 1,052 scaled cycles | 1,595 scaled cycles |
| Linux | 1,274,507,854 ns | 647,611,418 ns | 337,479,795 ns | 198,903,231 ns |

| System | 1-to-2 | 1-to-4 | 1-to-8 |
|---|---|---|---|
| capOS | 1.906x | 1.426x | 0.940x |
| Linux | 1.968x | 3.777x | 6.408x |

Raw capOS SMT artifacts are under `target/smp-process-scale/smt8-20260430T1148Z/`. Raw Linux SMT artifacts are under `target/linux-smp-process-scale/smt8-20260430T1151Z/`.

### In-Process Thread Scaling

`make run-thread-scale` runs sibling threads inside one process. Child threads use per-thread rings. The workload computes fixed-size checksum blocks; the default shape is a blocking parent join, 262,144 blocks (16 MiB), and `work_rounds=64`.

The harness records both a work-window time and a total time. The work window brackets the checksum computation. Total time includes thread startup, synchronization, shutdown, and join overhead. For scheduler analysis, both numbers matter: work speedup shows CPU placement and dispatch during the syscall-free section, while total speedup shows the cost of the surrounding thread lifecycle.

The old 1 MiB workload with a spinning parent is historical only because the matching Linux pthread baseline also stayed flat at four workers. The current rows use the repaired 16 MiB blocking-parent shape unless noted.

Recorded evidence:

| System / mode | Placement | Runs | 1-to-2 | 1-to-2 total | 1-to-4 | 1-to-4 total | Notes |
|---|---|---|---|---|---|---|---|
| Linux pthread baseline (benchmark VM, 2026-05-10 19:46 UTC) | physical-core logical CPUs 0,1,2,3 | 5 | 1.996x | 1.995x | 3.974x | 3.850x | Same checksum workload and pin set as the 2026-05-10 capOS row. |
| capOS (Phase D WFQ, benchmark | physical-core logical CPUs 0,1,2,3 | 5 | 1.809x | 1.774x | 3.088x | 2.700x | Per-thread weights/ latency classes, per-CPU WFQ |

| System / mode | Placement | Runs | 1-to-2 | 1-to-2 total | 1-to-4 | 1-to-4 total | Notes |
|---|---|---|---|---|---|---|---|
| VM, 2026-05-10 19:32 UTC) | | | | | | | queues, bounded steal path. |
| Linux pthread baseline (benchmark VM, 2026-05-02 21:34 UTC) | physical-core logical CPUs 0,1,2,3 | 5 | 1.988x | 1.987x | 3.963x | 3.858x | Same repaired workload before Phase D. |
| capOS (single global queue, benchmark VM, 2026-05-02 21:35 UTC) | physical-core logical CPUs 0,1,2,3 | 5 | 1.883x | 1.787x | 1.566x | 1.538x | Shows the four-worker cost of the single global runnable queue. |
| Linux pthread baseline (2026-05-01 report) | physical-core logical CPUs | 5 | 1.991x | 1.990x | 3.958x | 3.834x | Repaired-shape baseline recorded in docs/changelog.md; target artifact directory is not named in the source record. |
| capOS (pre-collapse placement, 2026-05-01 report) | physical-core logical CPUs | 5 | 1.828x | 1.687x | 3.029x | 2.386x | Commit 136b72de; per-CPU placement model later replaced by the queue-collapse cleanup; target artifact directory is not named in the source record. |
| capOS, switch logs suppressed (pre-collapse, 2026-05-01 report) | physical-core logical CPUs | 5 | 1.913x | 1.636x | 3.272x | 2.303x | Same commit and model with scheduler switch logs suppressed; target artifact directory is not named in the source record. |
| capOS (post-collapse, single global queue, 2026-05-02 10:42 UTC) | physical-core logical CPUs 0,1,2,3 on the benchmark VM | 3 | 1.890x | 1.792x | 1.504x | 1.436x | Queue-collapse row recorded in docs/backlog/scheduler-evolution.md; target artifact directory is not named in the source record. |

The 2026-05-10 Phase D WFQ row uses the same repaired shape as the 2026-05-02 pair: blocking parent join, 262,144 blocks, `work_rounds=64`, five runs, KVM-backed QEMU pinned to physical-core logical CPUs `0,1,2,3`, and a matching Linux pthread baseline on the same pin set. Raw capOS artifacts are under `target/thread-scale/20260510T193200Z/`; raw Linux artifacts are under `target/linux-thread-scale/20260510T194600Z/`.

The 2026-05-02 capOS/Linux pair used `main` commit `374f8556`; raw capOS artifacts are under `target/thread-scale/20260502T213544Z/`, and raw Linux artifacts are under `target/linux-thread-scale/20260502T213445Z/`.

The row improved the four-worker work window from `1.566x` to `3.088x` and the four-worker total window from `1.538x` to `2.700x` compared with the single-global-queue row. Linux on the same host and pin set recorded `3.974x` work and `3.850x` total at four workers. The remaining difference is the scheduler/runtime optimization target for later work.

Guest-side attribution is available with `CAPOS_THREAD_SCALE_GUEST_MEASURE=1`. It emits aggregate and per-phase measurements for `spawn_ready`, `work`, `shutdown`, and `final_total`, including scheduler choice, lock, timer, TLB, serial, shared-kernel-lock, network-poll, thread-placement, and sampled user-PC buckets. Host-side QEMU profiling is available with `CAPOS_THREAD_SCALE_PROFILE=1`.

### Interpreting CPU Counts

CPU-count rows are meaningful only with a recorded topology:

- Physical-core rows require enough physical cores for the vCPU count.
- SMT rows should say they are SMT rows and list the logical CPU set.
- Pinning QEMU with `taskset` is useful, but it is not CPU isolation by itself. Stronger runs should record `isolcpus/nohz_full/rcu_nocbs`, cpuset, or systemd affinity policy when used.
- Pinning QEMU to fewer host logical CPUs than guest vCPUs measures oversubscription behavior, not core scaling.
- Current QEMU/KVM results should stay separate from future direct cloud or bare-metal runs.

The current capOS benchmark table reaches four physical-core rows and an eight-logical-CPU SMT row on a 4-core/8-thread VM. It does not yet measure 16-core or 32-core systems.

### Next CPU-Scaling Work

The next CPU-scaling milestone should be designed around direct hardware or a dedicated perf runner rather than nested QEMU as the primary evidence source. The benchmark suite needs:

- hardware discovery records for socket/core/SMT topology, APIC mode, timer source, frequency policy, memory size, and firmware/device model;
- workload rows at 1, 2, 4, 8, 16, and 32 workers where the machine has enough physical cores, plus separately labeled SMT rows;
- at least one static map/reduce checksum workload, one uneven dynamic-task workload, one barrier-heavy phase loop, and one IPC/service-bound workload;
- work-window and total-time reporting for every workload;
- matching Linux native baselines on the same hardware where a comparable workload exists;
- scheduler/runtime counters for queue depth, migrations, steals, reschedule IPIs, TLB shootdowns, timer ticks, lock wait/hold time, blocked time, and runnable but not running time;
- raw artifacts with source commit, toolchain, kernel config, host topology, run count, warmup policy, and verifier output.

QEMU should remain useful for boot and regression coverage, but it should not be the primary source for a 16/32-core SMP scalability milestone.

### Commands

Run the capOS process-scale workload:

```
make run-smp-process-scale
```

Run the process-scale workload with QEMU pinned to selected host CPUs:

```
CAPOS_SMP_SCALE_QEMU_TASKSET_CPUS=0,1 make run-smp-process-scale
```

Run the process-scale SMT row on a host with at least eight logical CPUs:

```
CAPOS_SMP_SCALE_INCLUDE_SMT=1 \
  CAPOS_SMP_SCALE_QEMU_TASKSET_CPUS=0,1,2,3,4,5,6,7 \
  make run-smp-process-scale
```

Run the thread-scale workload:

```
CAPOS_THREAD_SCALE_RUNS=5 \
  CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS=0,1,2,3 \
  make run-thread-scale
```

Run the larger-workload Amdahl row:

```
CAPOS_THREAD_SCALE_RUNS=5 \
  CAPOS_THREAD_SCALE_TOTAL_BLOCKS=1048576 \
  CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS=0,1,2,3 \
  make run-thread-scale
```

Run a one-sample host-side QEMU profiling pass:

```
CAPOS_THREAD_SCALE_PROFILE=1 \
  CAPOS_THREAD_SCALE_RUNS=1 \
  CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS=0,1,2,3 \
  make run-thread-scale
```

Run a one-sample guest-side measurement pass:

```
CAPOS_THREAD_SCALE_GUEST_MEASURE=1 \
  CAPOS_THREAD_SCALE_RUNS=1 \
  CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS=0,1,2,3 \
  make run-thread-scale
```

Run only the host summary parser against an existing `results.csv` without booting QEMU:

```
CAPOS_THREAD_SCALE_SUMMARY_ONLY=1 \
  CAPOS_THREAD_SCALE_SUMMARY_CSV=<results.csv> \
  CAPOS_THREAD_SCALE_SUMMARY_KVM_EVIDENCE=1 \
  CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS=0,1,2,3 \
  CAPOS_THREAD_SCALE_TOTAL_BLOCKS=262144 \
  CAPOS_THREAD_SCALE_PARENT_WAIT=join \
  CAPOS_THREAD_SCALE_WORK_ROUNDS=64 \
  tools/qemu-thread-scale-harness.sh
```

Run the native Linux pthread baseline for the thread-scale checksum workload:

```
LINUX_THREAD_SCALE_TASKSET_CPUS=0,1,2,3 \
  make run-linux-thread-scale-baseline
```

Run the Linux process-scale comparison:

```
LINUX_SMP_SCALE_KERNEL=target/linux-smp-process-scale/kernel/vmlinuz \
  tools/linux-smp-process-scale-baseline.sh
```

On hosts where `/boot/vmlinuz` is not readable by the current user, copy a kernel image into ignored `target/` storage first through the host's normal administrative path, then pass it as `LINUX_SMP_SCALE_KERNEL`. The script does not invoke sudo itself.

## Configuration

The default capOS boot manifest (`system.cue` at the repo root) is layered on a shared scaffold in `cue/defaults/defaults.cue`. Operators can extend it without forking either file by dropping a `system.local.cue` overlay next to `system.cue`. The overlay is gitignored, so each developer/host can carry their own extensions without conflicting with `git pull`.

This document is the current operator-facing design for the configuration surface. The historical proposal and closeout rationale live in Proposal: System Configuration and Operator Extensibility.

### How the layering works

`mkmanifest --package capos system.cue manifest.bin` invokes `cue export .:capos --out json` against the repo root. CUE's package mode unifies every non-hidden `.cue` file in that directory that declares `package capos` — currently `system.cue` (committed) and any `system.local.cue` (gitignored) the operator drops in. The shared scaffold is imported by `system.cue`:

```
import defaults "capos.local/cue/defaults"
```

`#Manifest` (the value `system.cue` exports) inherits all defaults from `defaults.#DefaultSystem`, then applies any operator overrides declared in `system.local.cue`. The kernel decoder reads concrete fields at the document root (`schemaVersion`, `binaries`, `initConfig`, `kernelParams`); `#Manifest` is documentation-only.

The decoder rejects any other top-level field name with a typed error. For an unknown field named `kernelParameters` the rendered message is:

```
unknown field `kernelParameters` at $; expected one of `schemaVersion`,
`binaries`, `initConfig`, or `kernelParams`
```

CUE definitions (`#Foo`) and hidden fields (`_foo`) are stripped by `cue export` and never reach the decoder, so this only fires when the manifest projects an unintended visible name onto the document root — a typo such as `kernelParameters: …` instead of `kernelParams: …`, or a stale overlay field that was renamed in the defaults package. Fix it by renaming the projected field to one of the four accepted names, by moving the value under `kernelParams.…`, or by hiding the auxiliary value with a _/# prefix.

### Quick start

Copy the committed example and edit:

```
cp system.local.cue.example system.local.cue
$EDITOR system.local.cue
make run
```

The Makefile picks up the new file automatically — no flag, no include line. `make` re-evaluates the manifest because `system.local.cue` is a prerequisite of the manifest rule.

### Common Overlay Tasks

The examples below are complete `system.local.cue` fragments for common local configuration changes. Each fragment is intended to be copied as a starting point and adjusted before running `make run`.

### Override the MOTD

```
package capos

#Manifest: kernelParams: motd: """
  hello, capOS dev box.
  type 'login' to authenticate.
  """
```

The defaults package declares `motd: string | *"..."`, so a concrete overlay value wins under CUE unification (a more concrete value is strictly more specific than a default).

The system hostname is set the same way via `kernelParams.hostname` (defaults to `capos`); it is served by `SystemInfo.hostname` and shown by the shell `hostname` command. Bootstrap validation rejects whitespace, control characters, and values longer than 255 bytes.

```
#Manifest: kernelParams: hostname: "web-01"
```

### Add an authorized SSH key for the host operator

The default manifest declares a single host-operator seed account with the canonical 32-byte principal id `local-operator-principal-default`. Bind any number of authorized keys to that principal:

```
package capos

#Manifest: extraAuthorizedSshKeys: [{
  keyId:                "host-laptop-ed25519-2026-04"
  principalId:          "local-operator-principal-default"
  algorithm:            "ssh-ed25519"
  publicKey:            "<32-byte ed25519 public key as ASCII hex>"
  fingerprintSha256:    "<32-byte SHA-256 of the public key as ASCII
hex>"
  allowedShellProfiles: ["operator"]
  source:               "manifest"
  comment:              "host laptop"
}]
```

Convert an existing `~/.ssh/id_ed25519.pub` line to the manifest hex fields (Ed25519 example):

```
# extract the base64-encoded SSH wire format and decode the embedded key
ssh-keygen -e -m PKCS8 -f ~/.ssh/id_ed25519.pub | \
  openssl pkey -pubin -outform DER 2>/dev/null | \
  tail -c 32 | xxd -p -c 64
```

```
# fingerprintSha256 — SHA-256 over the same 32-byte raw public key:
ssh-keygen -e -m PKCS8 -f ~/.ssh/id_ed25519.pub | \
  openssl pkey -pubin -outform DER 2>/dev/null | \
  tail -c 32 | sha256sum | awk '{print $1}'
```

Use the printed hex as the `publicKey` and `fingerprintSha256` strings.

The proposal explicitly avoids auto-ingesting `~/.ssh/*.pub` from the Makefile. Manual conversion gives the operator control over which keys are trusted by the boot manifest.

### Add a non-operator principal

The single-account-multi-auth invariant fixes the host operator at `kind: "operator"`; slice 2 rejects manifests with multiple operator seeds. Additional principals must use `kind: "guest"` or `kind: "service"`:

```
package capos

#Manifest: extraSeedAccounts: [{
  name:            "kiosk-guest"
  displayName:     "Kiosk Guest"
  principalId:     "kiosk-guest-principal-32-bytes-x" // exactly 32
bytes
  kind:            "guest"
  credentialRefs:  []
  resourceProfile: "operator-default"
}]
```

Each seed account's `principalId` must be unique and exactly 32 bytes; each must reference an existing `resourceProfile` (either `operator-default` from the defaults package or one declared in `extraResourceProfiles`).

### Add a custom resource profile

```
package capos

#Manifest: extraResourceProfiles: [{
  name:                    "kiosk-guest-profile"
  homeQuotaBytes:          0
  tempQuotaBytes:          1048576
  processLimit:            2
  threadLimit:             4
  capLimit:                24
  memoryCommitLimitBytes:  16777216
  frameGrantLimitPages:    64
  endpointQueueLimit:      8
  inFlightCallLimit:       4
  ringScratchLimitBytes:   16384
  logQuotaBytesPerWindow:  32768
```

```
  networkProfile:            "none"
  cpuBudgetUsPerWindow:      10000
  cpuWindowUs:               100000
  timerWaiterLimit:          2
  launcherProfile:           "bootstrap-guest"
}]
```

Reference the profile name from `extraSeedAccounts[].resourceProfile`.

**Add a binary and an init-launched service**

The defaults package exposes `extraBinaries` and `extraServices` hooks. The first embeds an additional binary into `manifest.bin`; the second appends an entry onto `initConfig.services` so init launches it after the base service graph. Build the binary as part of the operator workflow — the default Make targets only build the binaries already listed in the defaults package.

```
package capos

#Manifest: extraBinaries: [{
  name: "site-monitor"
  path: "demos/target/x86_64-unknown-capos/release/capos-demo-site-
monitor"
}]

#Manifest: extraServices: [{
  name:    "site-monitor"
  binary:  "site-monitor"
  restart: "never"
  caps: [{
    name: "console"
    source: kernel: "console"
  }, {
    name: "timer"
    source: kernel: "timer"
  }],
}]
```

`extraServices` is concatenated onto `_baseServices` (base-first, then operator-extra), so the operator service starts after the defaults' chat server, remote-session gateway, and shell are launched.

**Override the console password verifier**

The defaults package ships a development-only Argon2id PHC for the plaintext "capos". Any non-research deployment should mint a fresh verifier and override it:

```
package capos
```

```
#Manifest: kernelParams: consolePasswordVerifierPhc:
  "$argon2id$v=19$m=19456,t=2,p=1$<salt-base64>$<hash-base64>"
```

Generate a verifier with the standalone argon2 tool (`argon2 "<salt>" -id -t 2 -m 19 -p 1 -e`) or from any Argon2id implementation that emits a PHC string with `m=19456,t=2,p=1`. The canonical 32-byte `local-operator-principal-default` operator principal id is unchanged; only the verifier rotates.

### Host-user injection (`@tag(user)`)

`make run` exports `CAPOS_CUE_USER=$(USER)`, and `mkmanifest` forwards it as `--inject user=...`. When `CAPOS_CUE_DISPLAY_NAME` is unset, `mkmanifest` derives `displayName` from the same account's first GECOS/comment field in `/etc/passwd` and forwards it as `--inject displayName=...`. If the passwd comment is unavailable or empty, displayName falls back to the account name. Other Make targets leave the structured tag variables unset, so untagged `system.cue` keeps the canonical `operator` account name. Focused demo and smoke manifests pin their own demo fixtures. The audit-correlatable `principalId` is fixed to the canonical 32-byte value regardless of host user, so audit history is stable across `$USER` changes.

`mkmanifest` also keeps the generic `CAPOS_CUE_TAGS` comma-separated escape hatch for additional `key=value` tags. The Makefile sets the structured variables target-scoped to `make run` only:

```
run: CAPOS_CUE_USER = $(USER)
```

Set additional tags via `make USER=alice CAPOS_CUE_DISPLAY_NAME='Alice Smith' CAPOS_CUE_TAGS=region=eu-west run` or by passing `--tag key=value` to `mkmanifest` directly. `system.cue` consumes `user` and `displayName` today; `user` must be a valid manifest seed account name. Future tags can carry hostname, locale, or other build-environment-derived values without adding new mechanisms.

### Tools-root cache

`CAPOS_TOOLS_ROOT` defaults to `$HOME/.capos-tools`. The pinned toolchain (capnp, cue, mdbook, typst, limine) lives under that path so multiple capOS clones share a single download. Override with `CAPOS_TOOLS_ROOT=/path/to/cache make ...` for non-default placement. The Makefile and `mkmanifest`'s `expected_cue_path` follow the same default; mismatched `CAPOS_CUE` / `CAPOS_CAPNP` env values are still rejected by `mkmanifest` and `make generated-code-check`.

### Schema-aware data conversion

`mkmanifest cue-to-capnp` converts CUE-authored data messages into arbitrary specified Cap'n Proto struct roots without routing them through the boot manifest ABI:

```
make cue-ensure capnp-ensure
CAPOS_CUE="$(make -s cue-path)" \
CAPOS_CAPNP="$(make -s capnp-path)" \
cargo run --manifest-path tools/mkmanifest/Cargo.toml --target "$(rustc
-vV | awk '/^host:/ {print $2}')" -- \
```

```
  cue-to-capnp --import-path schema input.cue schema/example.capnp
Example output.bin
```

The subcommand accepts the same CUE `--package`, `--tag`, and `CAPOS_CUE_TAGS` inputs as the manifest builder. It also accepts repeated `--import-path <dir>` or `-I<dir>` arguments plus `--no-standard-import`, which are passed to `capnp convert` as process arguments, not through a shell. The input CUE is first exported to JSON, then the pinned Cap'n Proto tool validates that JSON against the named schema and root struct.

This is the right path for configuration blobs, demo fixtures, or future schema-defined records that are not `SystemManifest`. It still cannot encode live capOS capability table entries or meaningful Cap'n Proto interface objects; authority transfer remains an IPC/runtime concern.

#### Limits and non-goals

- A second `kind: "operator"` seed account is rejected by the kernel in slice 2; multi-operator support is tracked in Proposal: User Identity, Sessions, and Policy.
- The slice-2 overlay is not a replacement for cloud-instance configuration; cloud-metadata-driven manifest deltas are designed in Proposal: Cloud Instance Bootstrap.
- The overlay does not auto-ingest `~/.ssh/*.pub`; conversion is manual by design (security review on which keys count).
- Focused-proof manifest migration onto the defaults package (slice 3, Task 2) is complete: every repo-root `system-*.cue` manifest declares its own CUE package and imports the defaults package, except `system-paperclips.cue` and `system-adventure.cue` (demo-owned, package-less but still importing defaults) and `system-measure.cue` (held by the measure-mode-repair plan). The Slice-3 inventory table in Proposal: System Configuration and Operator Extensibility records the per-manifest status, package, and `make run-*` target.

## Repository Map

This map names the main source locations for the current system. It is not an ownership file; use it to find the code behind architecture and validation claims.

### Root Files

- `README.md` gives the compact project overview.
- `docs/roadmap.md` records long-range stages and broad feature direction.
- `docs/tasks/state.toml` records the current selected milestone.
- `docs/tasks/README.md` defines the task-ledger schema and dispatch semantics.
- `docs/tasks/*.md`, `docs/tasks/on-hold/`, `docs/tasks/active/`, `docs/tasks/review/`, and `docs/tasks/done/` carry task lifecycle records.
- `docs/tasks/**` carries open review-finding remediation records; `REVIEW_FINDINGS.md` is a tombstone for pre-migration links.
- `REVIEW.md` defines review expectations.
- `Makefile` builds pinned tools, userspace binaries, manifests, ISO images, QEMU targets, formatting checks, generated-code checks, and policy checks.
- `rust-toolchain.toml` declares the Rust nightly channel, required targets, and `rust-src`; it does not pin an exact nightly by date or commit.
- `.cargo/config.toml` sets the default bare-metal target and useful cargo aliases.

### Schema and Shared ABIs

- `docs/abi-evolution-policy.md` defines compatibility classes, schema ordinal rules, ring-layout rules, version negotiation, and deprecation windows for externally visible ABI changes.
- `schema/capos.capnp` defines capability interfaces, manifest structures, exceptions, ProcessSpawner, ProcessHandle, and transfer-related schema.
- `capos-abi/src/lib.rs` defines small no_std ABI/policy constants shared by crates that should not depend on schema/config internals, including process quotas and credential policy limits.
- `capos-config/src/manifest.rs` defines the host and no_std manifest model.
- `capos-config/src/ring.rs` defines `CapRingHeader`, SQE/CQE structures, opcodes, flags, and transport error constants shared by kernel and userspace.
- `capos-config/src/capset.rs` defines the read-only bootstrap CapSet ABI.
- `capos-config/src/cue.rs` supports evaluated CUE-style manifest data.
- `capos-config/src/credential_policy.rs` re-exports credential policy limits; full PHC parsing is enabled by the `credential-validation` feature for bootstrap validators that need credential checks.
- `capos-config/tests/ring_loom.rs` models bounded ring protocol behavior with Loom.

Validation: `cargo test-config`, `cargo test-ring-loom`, `make generated-code-check`.

### Shared Pure Logic

- `capos-lib/src/elf.rs` parses ELF64 images for kernel loading and host tests.

- capos-lib/src/cap_table.rs implements CapId, capability-table storage, stale-generation checks, grant preparation, transfer transaction helpers, commit, rollback, and the CapTable quota constants sourced from capos-abi.
- capos-lib/src/frame_bitmap.rs implements the host-testable physical frame bitmap core.
- capos-lib/src/frame_ledger.rs contains a bounded frame-grant helper kept for host-test coverage; current MemoryObject accounting charges CapTable::ResourceLedger.
- capos-lib/src/lazy_buffer.rs provides bounded lazy buffers used by ring scratch paths.
- capos-lib/src/iso9660.rs is the pure ISO 9660 primary-volume-descriptor and directory-record parser the kernel boot-ISO driver (kernel/src/iso/) delegates to; fuzz target iso9660_volume.
- capos-lib/src/storage_format.rs holds the pure CAPOSRO1 (rofs), CAPOSST1 (disk_store), and CAPOSWF1 (writable_fs) mount parsers the kernel storage cap backers delegate to, including the shared record-layout constants the kernel writers reuse; fuzz targets storage_rofs_mount, storage_disk_store_mount, storage_writable_fs_mount.

Validation: cargo test-lib, cargo miri-lib, make kani-lib, fuzz targets under fuzz/fuzz_targets/.

### Kernel

- kernel/src/main.rs is the boot entry point, hardware setup sequence, manifest parsing path, and boot-launched service creation path. run_init resolves PID 1 from the kernel-embedded boot::INIT_ELF when initConfig.init.binary == capos_config::RESERVED_INIT_BINARY_NAME ("init") and otherwise from SystemManifest.binaries; for the embedded case it also injects the embedded image into the ProcessSpawner binary set under the reserved name so child spawns of init resolve.
- kernel/src/boot.rs exposes boot::INIT_ELF: &[u8], the PID 1 init image packaged at build time. kernel/build.rs reads the prebuilt init/ artifact (CAPOS_INIT_ELF, with a conventional-path fallback) and generates the include_bytes! static; init/ stays a standalone crate (byte packaging, not linker merging).
- kernel/src/spawn.rs loads user ELF images, creates process state, maps bootstrap pages, and enqueues spawned processes.
- kernel/src/process.rs defines Process, Thread, ThreadState, per-thread kernel stacks, park waiter storage, and userspace CPU context.
- kernel/src/sched.rs implements the single-CPU scheduler, timer-driven preemption, blocking cap_enter, direct IPC handoff, ParkSpace wait/wake, and deferred cancellation wakeups.
- kernel/src/serial.rs implements COM1/COM2 UART setup, manifest-driven console-vs-terminal routing, and kernel print macros.
- kernel/src/pci.rs implements early PCI config-space access through legacy I/O ports and ACPI MCFG/PCIe ECAM, with QEMU diagnostics for the current virtio-net and Q35 discovery paths, plus reusable memory-BAR subregion validation, kernel MMIO mapping helpers for in-kernel drivers, and MSI/MSI-X capability metadata discovery plus typed MSI-X table programming.

- `kernel/src/device_interrupt.rs` records the current kernel-owned virtio-net MSI-X config/RX/TX sources, their generation ids, route state, in-kernel driver owner, lock-free bounded device MSI vector-pool dispatch slots, and claimed-route reassignment/release without exposing userspace interrupt authority.
- `kernel/src/device_dma.rs` holds the kernel-owned, fixed-size DMA pool accounting ledgers. The net-keyed `VIRTIO_NET_DMA_POOL` backs virtio-net's `DmaPage` path; a focused single-queue `VIRTIO_BLK_DMA_POOL` (reusing the shared `ActivePage`/`QueueAccount` types, same generation-checked handle and scrub-before-free invariants) backs the virtio-blk request buffer. Each device's `VirtqueueDma` seam impl delegates to its own pool's keyed API.
- `kernel/src/dma_backend.rs` (always compiled) records the boot-time IOMMU probe verdict and resolves the fail-closed DMA backend selection (direct IOMMU remapping only with a verified probe, else kernel-owned bounce buffers) per the "Cloud DMA Backend" contract in `docs/dma-isolation-design.md`, emitting the boot proof line.
- `kernel/src/device_manager/` holds bounded in-kernel PCI device ownership records. The full DDF surface (device records, DMA pools/buffers, MSI-X interrupts, NVMe brokered controller registers, IOMMU domain ledgers, virtio ring publication, proofs) compiles only under `cfg(feature = "qemu")` in `qemu_full.rs`; the MMIO-only surface used by `cap::device_mmio` exists in both builds, dispatching to `stub.rs` (one-slot parked-region `DeviceMmio` record) in the production non-`qemu` build.
- `kernel/src/nvme_storage_backend.rs` (`cfg(not(feature = "qemu"))`) is the fail-closed activation gate for the always-built NVMe `BlockDevice` read arm: modeled on `dma_backend`, it resolves a production handle only when a brokered controller was discovered and a live `device_mmio` grant is staged, otherwise the `block_device` grant fails closed with a typed error.
- `kernel/src/virtio_transport.rs` (always compiled) is the device-agnostic virtio modern-PCI transport host surface: capability/region discovery constants and bounded volatile MMIO accessors usable outside the `qemu`-gated legacy virtio path.
- `kernel/src/virtio.rs` (`cfg(qemu)`) holds the legacy in-kernel virtio transport, now a qemu-only fixture: the non-`qemu` production build compiles `kernel/src/virtio_stub.rs` instead, whose typed negative results keep stale or fixture-only kernel networking call sites failing closed. It includes the virtqueue drivers used by the IOMMU remapping proof. Its `pub(crate) mod transport` is the device-generic layer: split-ring/common-config constants, the `MmioRegion` accessor, the `VirtqueueDescriptorTracker`, the `VirtqueueDma` DMA/notify seam, the seam-driven `Virtqueue`/`DmaPage` with their poll/submit/complete loop and the multi-descriptor `submit_request_chain`, and the device-id-parameterized `discover_modern_transport`. virtio-net is one seam caller (`VirtioNetDma`); virtio-blk is a second (`VirtioBlkDma` + `VirtioBlkDriver`, `diagnose_virtio_blk_transport`, the `block_device_*` request API behind the `BlockDevice` cap). Net-specific provider/proof methods stay in the parent module as `impl Virtqueue<VirtioNetDma>`.
- `kernel/src/iommu.rs` (`cfg(qemu)`) programs the Intel VT-d legacy-mode remapping tables, drives the hardware-DMA translation/fault proof, and runs the register-based invalidation revocation cycle.
- `kernel/src/iso/` (`cfg(boot_iso_read)` / `cfg(boot_iso)` / `cfg(qemu)`) is the boot-time ISO reader for the Boot Binary ISO Layout track. `AtapiDevice` (gate 1) locates the legacy IDE ATAPI device and exposes a bounded `read_sectors(lba, count, buf)` over polled-PIO

READ(12) packet commands with range/length validation. IsoFs (gate 2) is a read-only ISO 9660 driver layered on it: it parses the primary volume descriptor, walks directory records, and serves open_file(name) -> (lba, size) under /boot/bins/, validating every directory record and derived extent against the volume size before use (fail-closed BadVolume/NotFound/NotDirectory). boot_read_proof() reads the PVD (CD001) and boot_fs_proof() walks to /boot/bins/PAYLOAD.BIN and verifies its content, both behind boot_iso_read as the make run-boot-iso-read proof. The boot_source submodule (gate 4, cfg(boot_iso)) builds a validated (name, lba, size) registry from every declared manifest binary name (mapping each name to the ISO 9660 d-character form, e.g. capos-shell -> /boot/bins/CAPOS_SHELL) and reads ELF bytes on demand behind a device mutex; run_init and ProcessSpawnerCap consume it so the boot_iso kernel loads binaries from the ISO instead of embedded NamedBlob.data. Proofs: make run-boot-iso and the default make run-smoke. Under cfg(qemu) the always-on AtapiDevice/IsoFs surface (plus a qemu-gated block_size()/list_boot_bins() enumeration helper) also backs the read-only install-source fixture cap (kernel/src/cap/installable_image.rs).

Validation: cargo build --features qemu, make run-smoke, make run-spawn, make run-net, make run-iommu-remapping.

### Kernel Architecture

- kernel/src/arch/x86_64/gdt.rs sets up kernel/user segments and TSS state.
- kernel/src/arch/x86_64/idt.rs handles exceptions and timer interrupts; CPL3 #PF/#GP/#UD/#DB/#BP faults terminate the whole owning process through sched::exit_current_thread_terminating_process (deferred whole-process termination when sibling threads are live; proof make run-user-fault), while CPL0 faults still halt the machine.
- kernel/src/arch/x86_64/syscall.rs implements syscall MSR setup and entry.
- kernel/src/arch/x86_64/context.rs defines timer context-switch state.
- kernel/src/arch/x86_64/pic.rs and pit.rs configure legacy interrupt hardware.
- kernel/src/arch/x86_64/ioapic.rs maps MADT I/O APICs and programs masked legacy IRQ routes from interrupt-source overrides.
- kernel/src/arch/x86_64/lapic.rs programs the xAPIC LAPIC timer and IPIs.
- kernel/src/arch/x86_64/smap.rs enables SMEP/SMAP and brackets user memory access.
- kernel/src/arch/x86_64/tls.rs handles FS-base/TLS support.
- kernel/src/arch/x86_64/pci_config.rs provides legacy PCI config I/O used by the higher-level PCI module alongside its ECAM backend.
- kernel/src/arch/x86_64/percpu.rs, smp.rs, and tlb.rs provide per-CPU data, AP startup, and TLB shootdown for the SMP scheduler.

### Kernel Memory

- kernel/src/mem/frame.rs wraps the shared frame bitmap with Limine memory map initialization and global kernel access.

- kernel/src/mem/paging.rs manages page tables, address spaces, permissions, user mappings, W^X enforcement, and address-space teardown.
- kernel/src/mem/heap.rs initializes the kernel heap.
- kernel/src/mem/validate.rs validates user buffers before kernel access.

Related docs: DMA Isolation, Trusted Build Inputs.

### Kernel Capabilities

- kernel/src/cap/mod.rs initializes kernel capabilities and builds the first service's kernel-sourced bootstrap capability table.
- kernel/src/cap/table.rs re-exports shared capability-table logic and owns the kernel-global table.
- kernel/src/cap/ring.rs validates and dispatches ring SQEs.
- kernel/src/cap/transfer.rs validates transfer descriptors and prepares transfer transactions.
- kernel/src/cap/endpoint.rs implements Endpoint CALL, RECV, RETURN, queued state, cleanup, and cancellation behavior.
- kernel/src/cap/console.rs implements serial Console.
- kernel/src/cap/terminal_session.rs implements the session-scoped TerminalSession line-oriented terminal with bounded readLine, echo modes, and cancellation.
- kernel/src/cap/boot_package.rs implements the read-only BootPackage manifest-size/chunked-read capability.
- kernel/src/cap/manual.rs implements the read-only Manual capability: it parses the boot-packaged ManualCorpus blob (embedded as the manual-corpus named binary) and answers page/apropos/topics/section/describe/ buildInfo.
- kernel/src/cap/log.rs implements the Phase 1 monitoring log surface: LogSink (write) and LogReader (read) over a shared bounded, drop-oldest kernel recent-record ring. The sink drops records below the boot-seeded SystemConfig.logLevel threshold and forwards accepted records to serial; the reader returns records at/after a cursor with LogFilter (minLevel/componentPrefix), nextCursor, and dropped (docs/proposals/system-monitoring-proposal.md).
- kernel/src/cap/block_device.rs implements the BlockDevice CapObject (readBlocks/writeBlocks/info/flush). In the non-qemu production build the block_device source resolves to the userspace-brokered NVMe arm (BlockDeviceBackend::NvmeBrokered, gated by kernel/src/nvme_storage_backend.rs); the qemu build routes bounded inline-Data sector I/O to the kernel-owned virtio-blk driver in kernel/src/virtio.rs as a named fixture, not production storage (proof make run-virtio-blk). The cap is scoped to one device_index: the block_device source reaches the resolved non-target boot/storage disk, and block_device_target (KernelCapSource.blockDeviceTarget @44) reaches the manifest-selected PCI identity when it names a bound non-boot virtio-blk disk. A cap for one disk grants no authority over another. The kernel binds up to device_dma::MAX_VIRTIO_BLK_DEVICES (currently 2) virtio-blk devices, each with an independent driver/DMA-pool/interrupt-route instance (VirtioBlkDriver<const DEV> / VirtioBlkDma<const DEV> over VIRTIO_BLK_DMA_POOLS[DEV]); kernel/src/pci.rs enumerates each device with a device index

(proof make run-multi-virtio-blk). Target grants fail closed when the selector is absent, mismatched, or names the resolved boot disk. Counts are bounded to one bounce-buffer page.

- kernel/src/cap/readonly_fs.rs implements the read-only filesystem service: ReadOnlyFsDirectoryCap / ReadOnlyFsFileCap parse a fixed CAPOSRO1 on-disk layout read through the kernel-owned virtio-blk driver and serve Directory.list/open + File.read/stat; every mutating method fails closed. Granted via the read_only_fs_root KernelCapSource (returns a root Directory cap; qemu-gated, mounts at grant resolution and fails closed on a malformed/absent image). Host image builder tools/mkstore-image --readonly-fs; proof make run-storage-fs.
- kernel/src/cap/persistent_store.rs implements the disk-backed persistent Store: DiskStoreCap serves the Store interface (put/get/has/ delete) over a fixed CAPOSST1 on-disk layout read and written through a read+write BlockSource seam. put bump-allocates a data extent, writes the blob and entry record, then rewrites the superblock last as the durability commit point; delete tombstones the entry slot, and a later space-exhausting put compacts live entries through a shadow generation before recommitting the canonical front generation; the mount validates the superblock and every entry extent in-bounds and fails closed on a malformed image. The Virtio BlockSource (qemu kernel) routes to the kernel-owned virtio-blk driver byte-identically (folding in the data_region_base_lba() offset) and mounts eagerly at grant resolution; the Nvme BlockSource (built under cloud_persistent_store_over_nvme_proof) reads/writes through a granted NVMe BlockDevice window op and defers its mount-parse to the first Store call. Granted via the persistent_store KernelCapSource (virtio arm qemu-gated; the third NVMe-proof arm resolves the live device_mmio handle). Host image builder tools/mkstore-image; reboot proof make run-storage-persist (two QEMU passes on one disk image); NVMe put-then-get proof make run-cloud-provider-persistent-store-over-nvme via kernel/src/cap/persistent_store_over_nvme_proof.rs.
- kernel/src/cap/writable_fs.rs implements the disk-backed writable filesystem service: WritableDirectoryCap serves list/open/mkdir/remove/rename/ create and WritableFileCap serves read/write/stat/truncate/sync/ close over a fixed CAPOSWF1 on-disk layout (a flat node-record array with parent pointers + a bump-allocated data region) written through a BlockSource seam. The RAM tree is the working copy; each mutation write-through-commits in the order data sector → node-record sector → superblock. A filesystem-wide fail-closed single-writer policy admits one writer at a time. The Virtio BlockSource (qemu/installable kernels) routes to the kernel-owned virtio-blk driver byte-identically (folding the data_region_base_lba() offset) and mounts the singleton eagerly; the Nvme BlockSource (built under cloud_writable_fs_over_nvme_proof) reads/writes through a granted NVMe BlockDevice window op and defers the singleton mount-parse to the first Directory/File call. Granted via the writable_fs_root KernelCapSource (virtio arm qemu-gated; the third NVMe-proof arm resolves the live device_mmio handle), which mounts the process-wide singleton volume once and hands each grant a distinct writer id; fails closed on a malformed image. The NVMe write-then-read durability proof (make run-cloud-provider-writable-fs-over-nvme via kernel/src/cap/writable_fs_over_nvme_proof.rs, which supersedes and drops the persistent-store-over-NVMe proof) exercises both BlockDevice arms with the single-writer policy intact. The combined image builder tools/mkstore-image --writable co-locates the

CAPOSST1 Store sub-volume (LBA 0) and the CAPOSWF1 filesystem sub-volume on one disk; reboot proof make run-storage-writable (two QEMU passes: mutate then verify both the filesystem and the store survive). A slot becomes live on the next mount only once the superblock's bumped node_count is observed, so a poweroff in the record-written / superblock-pending window leaves an orphan slot the mount ignores. The proof-only storage_writable_recovery feature arms an induced forced poweroff in exactly that window (recovery_crash_after_record); bounded recovery proof make run-storage-writable-recovery (pass 1 commits then is kill -9d mid-allocation, pass 2 verifies recovery to a consistent tree with the interrupted allocation atomically absent). The same crash window is proven over the NVMe BlockDevice arm by make run-cloud-provider-writable-fs-over-nvme-recovery via kernel/src/cap/writable_fs_over_nvme_recovery_proof.rs (a recovery cap-waiter clone that implies and supersedes the happy-path proof module/route/init); the cloud_writable_fs_over_nvme_recovery_proof feature widens the storage_writable_recovery crash-window cfg gate, and the host-built NVMe image (tools/mkstore-image --writable-nvme, empty superblock + root-only node table) is booted twice with -device nvme (no @20 seed). writable_fs::mount_config_root (qemu-gated) scopes a writable Directory to the system/config subtree for the boot-time data-region grant below.

- kernel/src/cap/installable_image.rs implements the read-only install-source fixture (Installable System track item 5b): InstallableImageDirectoryCap serves list/open and InstallableImageFileCap serves read/stat/close over the booted CD-ROM ISO 9660 /boot/bins/ tree, reading through the kernel/src/iso/ boot_iso ATAPI/ISO 9660 driver behind a single shared-device mutex (so PIO does not interleave across CPUs). Every mutating method fails closed; a past-EOF read clamps to empty and an absent name is rejected, reusing the driver's validate_extent/read_sectors range checks. Granted via the qemu-gated installable_image_source KernelCapSource (mounts the ATAPI volume and validates /boot/bins/ at grant resolution, failing the spawn closed on an absent/malformed medium). Physically scoped to the ATAPI CD-ROM, so it cannot reach the writable virtio-blk target disk (block_device_target/writable_fs_root). Consumer demo demos/installable-image-source/; manifest system-installable-image-source.cue; proof make run-installable-image-source.
- demos/installable-system-install/ implements capos-system-install, the Installable System install flow (track item 6): under the read-only installable_image_source Directory and the target-scoped block_device_target BlockDevice selected by manifest PCI identity, it copies the packaged bootable boot-region head (BOOTHEAD.BIN) to LBA 0, writes the backup GPT (BOOTGPT.BIN) at the LBA read from the primary GPT header, and initializes an empty data region (DATAIMG.BIN, tools/mkstore-image --writable --empty-config) at the fixed cap::data_region_base_lba, validating ranges and verifying the read-back. It reads packaged files in 32 KiB windows (under the read-path reply scratch bound; see docs/tasks/done/2026-06-05/storage-file-read-reply-scratch-clamp.md) and zero-skips the FAT free space. tools/split-boot-region.py splits the mkdiskimage boot image into the head + backup GPT so only the populated prefix is packaged. Pass-1 installer manifest system-installable-install.cue; pass-2 installed manifest (baked into the boot region) system-installable-install-target.cue; harness tools/qemu-installable-install-smoke.sh; proof make run-installable-install (pass 1 installs into a second virtio-blk disk, pass 2 boots it standalone).

- kernel/src/cap/mod.rs grant_data_region (proof-only installable_data_region feature) is the Installable System boot-time data-region mount: run_init best-effort grants init a system/config Directory (data-config) plus the persistent Store (data-store) over the auto-attached data disk, failing closed wholesale to the base manifest (caps unchanged, "no data region; base floor" diagnostic) when the disk is absent, malformed, or missing system/config. No new cap type or schema change. Proof make run-installable-data-region (seeded disk prints resolved contents; no disk and zeroed-superblock disk hit the base floor).
- Installable System config-overlay compose/merge (track item 3): the SystemConfigOverlay capnp object + SystemManifest.extensionPoints (ManifestExtensionPoints) live in schema/capos.capnp; the typed decode, content-hash check, and compose_onto precedence (base-pins-win / overlay-adds-within-declared-extension-points / no-new-authority) live in capos-config/src/manifest.rs. init/src/main.rs apply_config_overlay reads system/config/overlay.bin from the granted data-config Directory, composes the overlay over the base plan, and falls closed to the base floor with [init] overlay rejected: <reason>. The tools/mkmanifest mkoverlay bin encodes overlays (filling the canonical hash) and tools/mkstore-image --writable --seed-overlay seeds them. Proof make run-installable-overlay.
- Installable System generations + rollback + failed-boot auto-fallback (track item 4): userspace-only over the already-granted Store + writable system/config Directory, no schema or kernel change. init/src/main.rs run_generation_rollback_checks (gated by a base service named generation-proof) represents system-config generations as content-addressed Store objects keyed by SHA-256, tracks the known-good active pointer and a staged/attempting candidate pointer as monotonic-epoch marker files (gen-active/gen-candidate) in the writable config region, records a boot attempt durably before applying a candidate, auto-falls-back to the known-good generation when a candidate is left unconfirmed (the brick-proofing guarantee), promotes a confirmed candidate, rolls config back to a retained prior generation, and rejects a stale/replayed (lower-or-equal-epoch) pointer. A present-but-undecodable gen-candidate marker (the torn size-0 file a poweroff inside the CREATE|TRUNCATE rewrite window leaves, or garbage bytes) is discarded with a loud diagnostic and boot falls back to the known-good generation, while a corrupt gen-active marker takes a distinct loud FATAL refuse-to-boot path (the known-good generation is genuinely unknown). Manifest system-installable-generation.cue; proof make run-installable-generation boots a --seed-config disk three times (boot 1 exercises the mechanism and leaves an unconfirmed candidate; boot 2 proves across-reboot auto-fallback to the known-good generation, then leaves a torn size-0 candidate marker; boot 3 proves torn-marker recovery).
- Installable System integrated bootable disk (track item 5, proof-only installable_disk feature, implies installable_data_region): one disk carries the boot ESP (GPT partition 1) and the co-located CAPOSST1 Store + CAPOSWF1 writable data region (GPT partition 2). kernel/src/cap/mod.rs data_region_base_lba returns the fixed partition-2 base LBA (264192) under the feature (0 otherwise), applied at the single persistent_store/ writable_fs read_range/write_range choke points so the kernel reads the region at that fixed tool/kernel-contract LBA without parsing the GPT. tools/mkdiskimage.sh --data-image/--data-offset-bytes fold the tools/mkstore-image --writable image into partition 2 and derive the ESP size from --esp-sectors (integrated disk uses the same 128 MiB ESP as the raw disk-image targets so a debug kernel fits). Manifest system-installable-disk.cue; proof make run-installable-disk boots

one virtio-blk disk and asserts the data region mounts from the boot disk and a data-region-only overlay service runs.

- kernel/src/cap/frame_alloc.rs implements FrameAllocator and MemoryObject.
- kernel/src/cap/virtual_memory.rs implements per-process anonymous memory operations.
- kernel/src/cap/timer.rs implements monotonic now and bounded sleep.
- kernel/src/cap/wall_clock.rs implements the read-only WallClock.wallTime cap: UTC over a fixed boot base layered on the monotonic timebase, reporting the fail-closed untrusted ClockProvenance (Phase 1 fixed-boot-base variant; docs/proposals/time-and-clock-proposal.md).
- kernel/src/cap/park_space.rs implements the process-local ParkSpace marker capability used by compact park (CAP_OP_PARK/CAP_OP_UNPARK) opcodes.
- kernel/src/cap/network.rs implements the qemu-only NetworkManager, TcpListener, TcpSocket, and UdpSocket fixture caps. The kernel no longer depends on smoltcp; non-qemu manifests reject the kernel network_manager / tcp_listen_authority grant sources (fail closed), and the production socket path is the Phase C userspace network-stack process. The socket-backed SocketTerminalSession shim is retired: TcpSocket.intoTerminalSession fails closed in every dispatch path.
- kernel/src/cap/process_spawner.rs implements ProcessSpawner and ProcessHandle.
- kernel/src/cap/provider_cap_waiter_proof.rs (non-qemu, cloud_provider_cap_waiter_proof Cargo feature) stages a fully-programmed-route bootstrap Interrupt grant source and the InterruptCapWaiterProof cap whose Interrupt.wait injects one device_interrupt::handle_lapic_delivery dispatch and whose Interrupt.acknowledge retires the deferred LAPIC EOI; the cap's on_release runs the masked-no-wake + reassign + stale-handle assertion chain before emitting cloudboot-evidence: provider-cap-waiter <token>. Mutually exclusive with cap::interrupt_grant_source_prod (default cloudboot path) and skips cap::provider_nic_bind_proof / cap::storage_bind_proof to keep the bound route live for the userspace cap-waiter handoff. Proof: make run-cloud-provider-cap-waiter.
- kernel/src/cap/virtio_net_device_bringup_proof.rs (non-qemu, cloud_virtio_net_device_bringup_proof Cargo feature; mutually exclusive with cloud_provider_cap_waiter_proof and the userspace selected-write handshake proof) drives the bounded virtio status sequence kernel-side over the picked virtio-net PCI function (vendor 0x1af4, device 0x1000 / 0x1041): resolves the modern virtio PCI transport regions through virtio_transport::parse_modern_pci_transport_capabilities, maps the common configuration window through pci::map_bar_region, and drives reset → ACKNOWLEDGE → DRIVER → feature discovery + driver-feature selection (VIRTIO_F_VERSION_1 only) → FEATURES_OK → DRIVER_OK with a trailing reset on every exit path. Inline assertions gate the headline cloudboot-evidence: virtio-net-device-bringup <token> on the negotiated feature set, COMMON_NUM_QUEUES >= 2, DRIVER_OK observation, and the final reset returning device_status to 0. Marker carries queue_setup=not-attempted, tx_descriptor=not-published, userspace_cap=not-issued, msix_function_enable=not-toggled, device_autonomous_raise=not-attempted, live_cloud=not-attempted. Proof: make run-cloud-provider-virtio-net-bringup.

- cloud_virtio_net_userspace_features_ok_proof (non-qemu; proof make run-cloud-prod-nic-driver-userspace-features-ok) is Phase C slice 1 of the userspace NIC relocation track. It makes cap::devicemmio_grant_source_prod stage the picked virtio-net modern common-config window as a selected-write DeviceMmio cap with registerWrite=selected-write-common-config-handshake; the userspace smoke drives reset -> ACKNOWLEDGE -> DRIVER -> FEATURES_OK over DeviceMmio.write32 and proves queue-address writes remain fail-closed. It is mutually exclusive with the kernel-owned virtio-net bringup, bundle, and queue-materialization proof chain over the same BDF/grant path, and with the cloud_nvme_readonly_bind_proof descendant chain because both stage a proof-specific production DeviceMmio grant source.
- kernel/src/cap/virtio_net_tx_authority_bundle_proof.rs, kernel/src/cap/virtio_net_tx_queue_materialization_proof.rs, and kernel/src/cap/virtio_net_msix_function_enable_proof.rs are the decomposed userspace-TX track. Each is non-qemu and gated by its own focused-proof Cargo feature (cloud_virtio_net_tx_authority_bundle_proof, cloud_virtio_net_tx_queue_materialization_proof, and cloud_virtio_net_msix_function_enable_proof respectively; the last implies the second so the bundle observer + production grant-source pickers + userspace bundle smoke stay compiled in across the chain). The bundle proof observes the three production grant sources (devicemmio_grant_source_prod, dmapool_grant_source_prod, interrupt_grant_source_prod) issuing one cap each into the spawned userspace bundle smoke and asserts same-BDF; the queue-materialization proof drives the kernel-side modern-virtio status sequence through DRIVER_OK and materializes one manager-owned TX virtqueue from three zeroed brokered frames, asserting register read-backs and post-reset clearance; the MSI-X function-enable proof extends that sequence with one canonical mask-first PCI MSI-X function-level enable (set FUNCTION_MASK, then ENABLE, then clear both) plus best-effort cleanup on every exit path. Each child emits its own headline marker (cloudboot-evidence: virtio-net-tx-authority-bundle <token>, cloudboot-evidence: virtio-net-tx-queue-materialization <token>, and cloudboot-evidence: virtio-net-msix-function-enable <token>); when the later feature is active the earlier markers are intentionally suppressed because their discipline labels would be inaccurate. Proofs: make run-cloud-provider-virtio-net-tx-authority-bundle, make run-cloud-provider-virtio-net-tx-queue-materialization, make run-cloud-provider-virtio-net-msix-function-enable.
- kernel/src/cap/virtio_net_userspace_rx_dma_proof.rs (Phase C slice 4a-ii, gated cloud_virtio_net_userspace_rx_bringup_proof) drives the first real RX DMA from the **shim-owned** vring: post_rx_descriptor writes the RX descriptor
  - avail over the shim's retained RX vring physes at DMABuffer.submitDescriptor time, and drive_rx_dma (reached from the now-live provider_notify_doorbell_write_for_cap) rings the RX doorbell, submits a kernel-half SLIRP TX ARP stimulus over the retained TX physes, polls one real device->host completion, and resets the device (clearing the retained enabled flags to release the ring-buffer pins). Self-contained byte-level vring helpers are duplicated from virtio_net_polled_provider to protect run-net. The notify region is mapped kernel-side + the per-queue notify slot offsets captured by cap::devicemmio_grant_source_prod

(rx_dma_notify_state). Proof make run-cloud-prod-nic-driver-userspace-rx-bringup (extended).

- kernel/src/cap/null.rs implements the measurement-only NullCap.
- kernel/src/cap/park_bench.rs implements the measurement-only ParkBench authority used by make run-measure.

Related docs: Capability Model, Authority Accounting.

**Userspace**

- init/ is the standalone init process. In the spawn smoke, it uses ProcessSpawner, grants initial child capabilities, waits on ProcessHandles, and checks hostile spawn inputs.
- capos-rt/src/entry.rs owns the runtime entry path and bootstrap validation.
- capos-rt/src/alloc.rs initializes the userspace heap.
- capos-rt/src/syscall.rs provides raw syscall wrappers.
- capos-rt/src/capset.rs provides typed CapSet lookup helpers.
- capos-rt/src/ring.rs implements the safe single-owner ring client, out-of-order completion handling, transfer descriptor packing, and result-cap parsing.
- capos-rt/src/client.rs implements typed clients for Console, TerminalSession, BootPackage, ProcessSpawner, ProcessHandle, and Timer. The client-side methods are generic over Transport; result-cap-adopting methods stay on the concrete RuntimeRingClient.
- capos-rt/src/transport.rs defines the Transport seam (the client-side CALL/completion/ RELEASE ring operations) and the in-system RingTransport (RingClient viewed through the seam). A host remote transport is a later slice; see docs/backlog/capos-sdk-dual-transport.md.
- capos/ is the front-door SDK facade crate: for the default ring feature it re-exports the capos-rt runtime, typed clients, the entry_point! macro, and a prelude. The remote feature is reserved. Standalone, like capos-rt.
- capos-rt/src/pollselect.rs is the pure POSIX poll/select bridge: SocketReadiness -> poll revents (POLLIN/POLLOUT/POLLHUP/POLLERR/ POLLNVAL) and select set membership, plus unsupported_request_bits for fail-closed flag handling. Shared by the libcapos-posix C surface and the posix-socket-poll-select-smoke proof. Proof: make run-posix-socket-poll-select.
- capos-rt/src/panic.rs provides the emergency Console panic output path.
- capos-rt/src/bin/smoke.rs is the runtime smoke binary used by focused runtime proofs rather than the default boot manifest.
- capos-service/src/lib.rs is the standalone no_std service lifecycle layer above capos-rt; slice 1 exposes ServiceMain, ServiceRuntime, and ordered initialize/dependency-wait/ready/ run/drain/shutdown/cleanup phases.
- shell/src/main.rs is the native capability shell, built as the standalone capos-shell crate and packaged by system.cue, system-shell.cue, and the focused login manifests.

Validation: make capos-rt-check, make run-smoke, make run-spawn, make run-shell, make run-terminal. The former Telnet fixture is retired with the qemu-only kernel TCP listener.

### Standalone C and WASI Substrates

These are standalone crates (not workspace members) built by the Makefile.

- `libcapos/` builds `libcapos.a`, a `no_std` Rust staticlib exposing the capos-rt syscall/ring/CapSet path and typed `Console`/`Timer`/ `EntropySource`/`VirtualMemory` wrappers plus C heap shims to C consumers. Public header at `libcapos/include/capos/capos.h`. No POSIX surface.
- `libcapos-posix/` builds `libcapos_posix.a`, a `no_std` Rust staticlib layering a POSIX adapter over `libcapos`: per-process fd table, errno cell, historical UDP socket wrappers over the retired qemu-only kernel `UdpSocket` cap, clock over `Timer`, `pipe`/`dup` over `Pipe`, `poll`/`select` (`poll.rs`, `<poll.h>`/`<sys/select.h>`) over the `capos-rt::pollselect` readiness bridge with fail-closed unsupported-flag / `EBADF` / `EINVAL` handling, and `fork`/`execve`/`waitpid` via the recording-shim ProcessSpawner Move-grant path, plus the libc surface the dash port needs: stdio/string/stdlib/ ctype helpers, `strerror`/`qsort`/`umask`/`abort`/ `strtoll`/`strpbrk`/`lstat`/`getgroups`/`wait3`/`vfork`, byte-order helpers (`inet.rs`), `getrlimit`/`setrlimit` (`resource.rs`), `setlocale` (`locale.rs`), `times` + `tcgetattr`, C-locale `wchar`/`wctype` multibyte (`wchar.rs`), the `environ` pointer, and the `sys_siglist` array. C headers (the namespaced source of truth) under `libcapos-posix/include/capos/posix/` – including the dash-needed `sys/types.h`, `termios.h`, `sys/resource.h`, `sys/times.h`, `wchar.h`, `wctype.h`, `locale.h`, `inttypes.h`, and the decl-only `sys/ioctl.h`/`sys/mman.h`/`arpa/inet.h`/`getopt.h`/`paths.h`/ `sys/param.h`. `libcapos-posix/sysroot/include/` is the `-nostdinc` bare-header sysroot (`<stdio.h>`, `<unistd.h>`, `<sys/stat.h>`, …) whose wrappers forward into that namespace; mirrored C ports (dash) build against it via the Makefile's `CAPOS_C_SYSROOT_INCLUDE` flags on the `capos-c-multitu-elf` rule. Focused sysroot proof `make run-c-libc-surface`.
- `capos-wasm/` is the `no_std` WASI host adapter: a `wasmi`-backed `Runtime`, the `wasm-host` userspace binary, the Preview 1 import resolver, and the manifest-supplied wasm payload reader.
- `vendor/wasmi-no_std/` and `vendor/dns-c-wahern/` are static-pinned, no-patches upstream snapshots consumed by `capos-wasm/` and the POSIX DNS smoke; do not patch them in place (refresh procedure in each `VENDORED_FROM.md`).
- `vendor/dash/` is the mirror-as-is dash `0.5.13.4` snapshot (`src/` stays byte-identical; capOS deviations live under `patches/`). Its capOS build pipeline lives outside the mirror under `vendor/dash/capos/`: the pinned `config.h` and `gen-tables.sh` (stages a patched source copy + runs the six host table generators). The Makefile `dash` target builds `target/dash/dash.elf` through `capos-c-multitu-elf` against `libcapos.a` + `libcapos_posix.a`.

Validation: `make run-c-hello`, `make run-posix-pipe-smoke`, `make run-posix-printf`, `make run-wasm-host`, `make run-wasi-hello-rust`, `make run-wasi-random`. The former POSIX DNS smoke is retired with the qemu-only kernel `UdpSocket` owner.

### Demo Services

`demos/` is a nested userspace smoke-test workspace. Each demo is a release-built service binary packaged into the boot manifest:

- `adventure-client`, `adventure-server`, `adventure-npc-shopkeeper`, `adventure-npc-wanderer`
- `capos-chat`, `chat-bot`, `chat-client`, `chat-server`

- `capset-bootstrap`, `console-paths`, `credential-store`
- `endpoint-queue-limit-smoke`, `endpoint-roundtrip`, `ipc-server`, `ipc-client`, `in-flight-call-limit-smoke`
- `frame-allocator-cleanup`, `memoryobject-shared-child`, `memoryobject-shared-parent`
- `paperclips`, `paperclips-content`
- `revocable-read`, `revocation-observer`
- `ring-corruption`, `ring-reserved-opcodes`, `ring-nop`, `ring-fairness`
- `service-common`, `shell-spawn-test`, `shell-typed-call`
- `terminal-session`, `terminal-stranger`
- `timer-smoke`, `timer-flood`
- `tls-smoke`, `unprivileged-stranger`, `virtual-memory`
- `user-fault-parent`, `user-fault-victim` (user fault containment proof, `make run-user-fault`)

Shared demo support lives in `demos/capos-demo-support/src/lib.rs` and uses `capos-rt` for entry, allocator, syscall, CapSet, and panic support while keeping raw ring helpers for low-level transport smokes.

Validation: `make run-spawn`.

### Manifest and Tooling

- `system.cue` is the default init-owned boot manifest source. It imports the shared defaults package, boot-launches standalone `init`, and lets init start the shell, remote-session CapSet gateway, and resident services.
- `system-spawn.cue` is the ProcessSpawner smoke manifest source.
- `system-smoke.cue` is the scripted focused shell-led login/shell smoke manifest source.
- `system-chat.cue`, `system-adventure.cue`, and `system-paperclips.cue` are focused resident-service and terminal-demo manifest sources.
- `system-memoryobject-shared.cue`, `system-revocable-read.cue`, and `system-measure.cue` are focused regression/measurement manifest sources.
- `system-shell.cue` is the focused anonymous-shell manifest source (no verifier, shell stays anonymous).
- `system-terminal.cue` is the focused TerminalSession proof manifest source.
- `system-credential.cue` is the focused CredentialStore proof manifest source.
- `system-login.cue` is the focused password-login proof manifest source.
- `system-login-setup.cue` is the focused first-boot setup proof manifest source.
- `tools/mkmanifest/` evaluates manifest input, embeds binaries, validates manifest shape, writes boot-manifest Cap'n Proto bytes, and provides `cue-to-capnp` for schema-aware CUE-authored data-message conversion. Its sibling `mkoverlay` bin encodes a `SystemConfigOverlay` from CUE into the `system/config/overlay.bin` bytes (filling the canonical content hash) for the installable-system config-overlay proof.
- `tools/manualc/` is the System Manual corpus compiler: it parses `schema/capos.capnp` for section-2 interface pages, reads the authored man corpus under `docs/manual/man<section>/`

*.man, and emits the boot-packaged ManualCorpus blob. It fails the build if any in-tree capability interface lacks a section-2 page (i.e. a schema doc comment).

- docs/manual/ holds the authored man-shaped corpus consumed by manualc (section 1 shell-command pages and section 7 concept pages); section-2 capability pages are generated from the schema, not stored here.
- system-manual-smoke.cue is the focused Manual proof manifest source.
- tools/agent-session-recaps/ contains private-session recap and raw-archive tooling for the agentic development experiment. The tools are tracked here; raw transcripts and generated recap stores stay outside the repo unless explicitly redacted and reviewed.
- tools/check-generated-capnp.sh verifies checked-in generated schema output.
- scripts/record_worklog.py emits per-task commit spans (from each task's commits: list, falling back to task-file history) for the development timeline/Gantt; scripts/validate_backfill_tasks.py validates backfilled task-file frontmatter against the chunk's real SHAs; scripts/check-md-links.py is the pre-commit broken-relative-link gate over all .md.
- tools/githooks/ is the repo core.hooksPath (enabled with make hooks): prepare-commit-msg stamps provenance trailers (Plan-Item/Run-Id/ Agent-Kind) onto run-driven commits, alongside the git-lfs hooks.
- tools/qemu-net-harness.sh runs the current QEMU net harness, with tools/qemu-net-smoke.sh asserting virtio-net transport, MSI-X metadata selection, kernel-owned MSI-X vector-pool allocation/programming, masked route-lifecycle proof, queue vector assignment, descriptor guards, ARP, and ICMP fixture lines.
- fuzz/ contains fuzz targets for manifest Cap'n Proto decoding (with the production reader-options envelope), mkmanifest JSON conversion/validation, ELF parsing, Telnet IAC filtering, terminal line discipline, ring SQE wire validation, ISO 9660 PVD/directory-record parsing, the CAPOSRO1/ CAPOSST1/CAPOSWF1 storage mount parsers, and the capos-tls X.509 validity walk.

Validation: cargo test-mkmanifest, make generated-code-check, make fuzz-build, make fuzz-smoke.

### Documentation

- docs/capability-model.md is the current capability architecture reference.
- docs/architecture/threading.md and docs/architecture/park.md record the accepted contracts and first implementation for in-process thread ownership and private ParkSpace authority.
- docs/*-design.md files record targeted implemented or accepted designs.
- docs/proposals/ contains accepted, future, exploratory, and rejected designs.
- docs/research/ summarizes prior art (the capability-systems-survey.md synthesis plus per-system deep-dive reports).
- docs/proposals/mdbook-docs-site-proposal.md defines the documentation site structure and status vocabulary used by the orientation pages.

## Programming Languages

capOS currently supports native Rust programs that are written for the capOS userspace runtime. Other languages are design tracks, not implemented platform support. The main rule is simple: a language runtime may expose familiar APIs, but authority still comes from the process CapSet and typed capability calls.

### Current Support

| Language or runtime | Status | Path |
| --- | --- | --- |
| Rust, capOS-native | Implemented baseline | `#![no_std]`, `alloc`, `capos-rt`, static ELF, `x86_64-unknown-capos`. Phase D best-effort fair scheduling closed at commit `77caafc0` (`2026-05-10 19:39 UTC`): per-thread weighted vruntime, per-CPU WFQ run queues, bounded steal/migration, and `SchedulingPolicyCap` weight/latency-class authority. |
| Rust `std` | Not implemented | Future Rust standard-library or adapter work over capabilities |
| C | Phase 0 in tree (libcapos C-substrate v0 + libcapos-posix v0) | `libcapos.a` exposes capos-rt syscalls, ring CALL, CapSet lookup, a heap shim, typed `Console.writeLine`, `Timer.now`, `EntropySource.fill`, VirtualMemory wrappers, and native `ProcessSpawner.createPipe` / `Pipe` wrappers through `extern "C"`; `make run-c-hello` proves baseline C wrappers and `make run-c-pipe` proves a C binary can create a Pipe, write/read a marker, close the writer, and observe EOF without using the POSIX adapter. `libcapos_posix.a` adds the POSIX adapter v0 surface above libcapos: per-process static fd table (32 fds), TLS errno via `__errno_location()`, historical UDP `socket`/`sendto`/`recvfrom`/`close` wrappers over the retired qemu-only kernel `UdpSocket` cap, `clock_gettime(CLOCK_MONOTONIC, ...)`, `gettimeofday(&tv, NULL)`, `time`, `nanosleep`, and `sleep` over the kernel `Timer` cap, fail-closed signal stubs, `pipe`/`read`/`write`/`dup`/`dup2` over the kernel `Pipe` cap, and `fork`/`execve`/`waitpid`/`_exit`/`posix_inherit_stdio` plus a direct `posix_spawn` successor via the recording-shim Move-grant path through `ProcessSpawner.createPipe` / `ProcessSpawner.spawn`. See the POSIX adapter row for shipped smokes; the old DNS smoke is retired until resolver networking is rebuilt on the userspace stack. |
| C++ | Future experiment | Depends on C startup, ABI choices, allocator, exceptions/RTTI policy, and a useful freestanding subset |
| Go | Future design | Custom `GOOS=capos` per Go Runtime proposal a separate Phase W.8 path (`docs/proposals/wasi-host-adapter-proposal.md` Task 9) targets a TinyGo / upstream Go `GOOS=wasip1` CUE evaluator binary that runs inside the WASI host adapter against a future `ScriptPackage` cap |
| Python | Future design | Native CPython or MicroPython through a POSIX-style adapter; WASI/ Emscripten for sandboxed or compute-only use |
| Lua | Phase 1 in tree (L.3 deterministic memory release) | `demos/lua-smoke/` runs a hand-written Lua-subset interpreter that exercises three capability-aware host bindings: `console:write_line`, `timer:now`, and L.3 `memory:{alloc,write,read,size,release}` over `capos-rt::VirtualMemoryClient` (kernel-mapped address never crosses to Lua; every byte access is bounds-checked host-side; release unmaps the exact rounded region and marks the userdata dead). PUC Lua dialect compatibility is deferred to the future C/libcapos port. See Lua Scripting proposal. |
| JavaScript / TypeScript | Future design | QuickJS-style native runner or WASI-hosted engine; not a browser JS shell |
| WASI / WebAssembly | Phase W.5 landed 2026-05-17 05:42 UTC (Phase W.4 closed 2026-05-07 20:09 UTC; Phase W.3 closed 2026-05-07 18:25 | Host imports backed by capabilities; useful for sandboxed code and portable tools. W.1 vendored upstream wasmi (`v1.0.9`) at `vendor/wasmi-no_std/wasmi-1.0.9/` and shipped the `capos-wasm/` standalone crate that exposes a `Runtime` value (wasmi `Engine` + `Store<HostState>`). W.2 sub-slice 1 added the `wasm-host` userspace binary in `capos-wasm/src/bin/wasm-host.rs`, the `system-wasm-host.cue` focused-proof manifest, and `make run-wasm-host`, which still asserts the empty-instantiation |

| Language or runtime | Status | Path |
| --- | --- | --- |
| | UTC; Phase W.2 closed 2026-05-07 10:53 UTC) | regression. W.2 sub-slice 2 grew the same binary with the Preview 1 import resolver in `capos-wasm/src/wasi/preview1.rs`: 46 `wasi_snapshot_preview1` imports land on the wasmi linker; `clock_time_get(CLOCKID_MONOTONIC)` is backed by the manifest-granted `Timer` cap; `proc_exit` exits via `capos_rt::syscall::exit`; `fd_write(1, …)` / `fd_write(2, …)` route through the manifest-granted `Console` cap with a fixed 4 KiB iov-total scratch ceiling and a 1 KiB per-call chunk that matches the kernel `Console` cap's `MAX_SERIAL_CAP_WRITE_BYTES`; everything else (including `random_get`, which Phase W.4 promotes against `EntropySource`) returns `ERRNO_NOSYS`. A 114-byte hand-encoded probe module imports `random_get`, calls it once, stores the returned errno in an exported global, and the host refuses to print the `[wasm-host] preview1 imports linked: clock_time_get, fd_write, proc_exit, args/environ empty; nosys=52` proof line unless that errno equals `ERRNO_NOSYS`. W.2 sub-slice 3 added `demos/wasi-hello-rust/` (a one-liner `println!` Rust crate built for the upstream `wasm32-wasip1` target), `system-wasi-hello-rust.cue` (now grants `console`, `timer`, and the optional `boot` (`BootPackage`) cap), `tools/qemu-wasi-hello-rust-smoke.sh`, and `make run-wasi-hello-rust`. The wasm-host binary keeps running the sub-slice 1 + 2 regression first; when the manifest grants `boot`, it also reads the manifest blob through `BootPackage`, decodes `binaries[]` via raw capnp readers (new `capos_wasm::payload` module), instantiates the `wasi-payload` wasm, explicitly invokes the `_start` export (wasmi's `instantiate_and_start` runs the WebAssembly `start` section, NOT WASI's `_start`), and lets the payload's `println!` reach the kernel Console cap through Preview 1 `fd_write`. capos-rt grew narrow re-exports (`capos_capnp` and `default_reader_options`) so capos-wasm keeps a single direct path-dep on capos-rt and the vendored wasmi tree. The slice also kept the W.2 sub-slice 1 userspace-image budget bump (USER_STACK_BASE `0x100_0000`) for wasmi's 3 MiB BSS. W.2 sub-slice 4 closed Phase W.2 by adding `demos/wasi-hello-c/` (a single `printf("Hello, wasi from capOS C\n")` C `main()` built directly with system clang-18 against the Ubuntu wasi-libc + `libclang-rt-18-dev-wasm32` apt packages: `clang --target=wasm32-wasi --sysroot=/usr -O2 -Wall -Wextra` produces a 46 KiB `wasm32-wasi` module), `system-wasi-hello-c.cue`, `tools/qemu-wasi-hello-c-smoke.sh`, and `make wasi-hello-c-build` / `make run-wasi-hello-c`. C runs on capOS without any `libcapos`/POSIX work in tree because the wasm-host payload-load path landed in sub-slice 3 carries the C `.wasm` payload through the same wasm-host binary unchanged. Phase W.3 backed `args_get` / `args_sizes_get` with the manifest-supplied `initConfig.init.wasiArgs` text grant: the wasm-host walks the field through raw capnp readers in `capos_wasm::payload::read_wasi_args`, validates against `WASI_ARGS_MAX_COUNT = 32` / `WASI_ARGS_MAX_ARG_BYTES = 4096` / `WASI_ARGS_MAX_TOTAL_BYTES = 8192` (rejecting interior NUL bytes), packs the bytes into a per-instance `HostState` argv buffer, and reflects them through Preview 1 to the wasm guest. A 2026-05-13 bounded environment grant mirrors that path for `initConfig.init.wasiEnv`: the wasm-host walks `capos_wasm::payload::read_wasi_env`, validates against `WASI_ENV_MAX_COUNT = 32` / `WASI_ENV_MAX_ENTRY_BYTES = 4096` / `WASI_ENV_MAX_TOTAL_BYTES = 8192` (rejecting interior NUL bytes), packs `KEY=value` entries into a per-instance environment buffer, and reflects them through Preview 1 `environ_get` / `environ_sizes_get`; absent grants remain empty. The W.2 sub-slice 2 "args/environ empty" proof line stays byte-identical because the regression module passes empty argv and no environment. The new `demos/wasi-cli-args/` Rust smoke (`println!` of `argv[1]`), `system-wasi-cli-args.cue`, `tools/qemu-wasi-cli-args-smoke.sh`, and `make wasi-cli-args-build` / `make run-wasi-cli-args` close the per-instance argv plumbing; `demos/wasi-env/`, `system-wasi-` |

| Language or runtime | Status | Path |
| --- | --- | --- |
| | | env.cue, tools/qemu-wasi-env-smoke.sh, and make wasi-env-build / make run-wasi-env prove one granted environment value reaches a Rust wasm32-wasip1 payload. Schema/schema/capos.capnp is unchanged because initConfig is already a CueValue and unknown sub-fields under initConfig.init are ignored by the existing manifest decoder. Phase W.4 wires Preview 1 random_get through the kernel EntropySource cap. The wasm-host (capos-wasm/src/bin/wasm-host.rs) looks up an optional per-instance EntropySource cap from the CapSet under the well-known name random and installs the typed EntropySourceClient on HostState AFTER the W.2 sub-slice 2 probe regression has run, keeping the closed-fail nosys=52 proof line byte-identical. Preview 1 random_get (capos-wasm/src/wasi/preview1.rs) drains arbitrary wasm-supplied byte ranges through EntropySourceClient::fill_wait, chunked at the kernel cap's MAX_ENTROPY_FILL_BYTES = 64 ceiling and capped per Preview 1 invocation at RANDOM_GET_MAX_BYTES = 65_536. RDRAND-unavailable / truncated kernel responses surface as ERRNO_IO; oversized requests as ERRNO_INVAL; out-of-bounds wasm pointer writes as ERRNO_FAULT. Manifests without the grant keep returning ERRNO_NOSYS from the closed-fail refusal branch which never enters the kernel, so an instance without an EntropySource grant cannot leak entropy. Wall-clock support stays deferred until capOS has a typed WallClock/RealTimeClock cap; clock_time_get(CLOCKID_REALTIME) keeps the W.2 sentinel ERRNO_NOSYS. The new demos/wasi-random/ Rust smoke (raw Preview 1 random_get binding reading N=64 bytes), system-wasi-random.cue (granted), system-wasi-random-ungranted.cue (ungranted), tools/qemu-wasi-random-smoke.sh, tools/qemu-wasi-random-ungranted-smoke.sh, make wasi-random-build, make run-wasi-random, and make run-wasi-random-ungranted close Phase W.4. A 2026-05-13 compatibility-import smoke adds demos/wasi-stdio-fd/, system-wasi-stdio-fd.cue, tools/qemu-wasi-stdio-fd-smoke.sh, and make run-wasi-stdio-fd; it directly imports clock_res_get(MONOTONIC), sched_yield, fd_fdstat_get(1/2), and fd_seek(1/2) and requires every promoted import to return a non-ERRNO_NOSYS result without granting filesystem, socket, or stdin authority. A 2026-05-13 harness-hardening smoke adds demos/wasi-preview1-refusals/, system-wasi-preview1-refusals.cue, tools/qemu-wasi-preview1-refusals-smoke.sh, and make run-wasi-preview1-refusals; it directly imports path_open, fd_prestat_get, fd_read, sock_send, and sock_recv and asserts the documented fail-closed errno when no Namespace/File/Store/socket authority exists. Phase W.5 (2026-05-17 05:42 UTC) wires the Preview 1 preopened-directory filesystem against the kernel Directory / File cap interface: the wasm-host looks up an optional per-instance Directory cap from the CapSet under the well-known name root and installs it as a single Preview 1 preopen at fd 3 named /preopen-0. capos-wasm/src/wasi/fs.rs implements path_open, fd_read, fd_write, fd_seek, fd_close, fd_filestat_get, fd_prestat_get, and fd_prestat_dir_name over DirectoryClient / FileClient; the resolver mirrors POSIX P1.4 Slice 4's libcapos-posix/src/path.rs – intermediate segments walk Directory.sub, the leaf mints either an existing or freshly created File via `Directory.open(flags=CREATE |
| POSIX-shaped software | Partial implementation | Compatibility adapter over explicit file, directory, socket, stdio, timer, process, and namespace caps. See POSIX Adapter proposal and plan. P1.1, P1.2, and P1.3 are closed; the former direct DNS smoke is retired with the qemu-only kernel UdpSocket owner, while make run-posix-pipe-smoke, make run-posix-spawn-smoke, and make run-posix-stdio-smoke cover pipe/fork-for-exec, direct posix_spawn, and Console-backed stdio surfaces. P1.4 file/directory fd work closed at commit f97d9833 (2026-05-23 06:23 UTC): make run-posix-file proves open(), write(), lseek(), read(), opendir(), readdir(), and closedir() through a live C process over the RAM-backed root Directory cap. Closed P1.4 successors |

| Language or runtime | Status | Path |
| --- | --- | --- |
| | | now include printf/string (`make run-posix-printf`), identity stubs (`make run-posix-identity`), and signal/time stubs (`make run-posix-signal-time`). Remaining P1.4 work is dash vendoring/patching, the multi-translation-unit C build, and `make run-posix-shell-smoke`; long-form decomposition lives in POSIX Adapter Phase P1.4: Running dash. |

### Native Rust Today

The implemented path is Rust without the standard library. Programs use `core` and may use `alloc` types such as `Vec`, `String`, `Box`, and `BTreeMap` because `capos-rt` installs a userspace allocator. They do not get `std::fs`, `std::net`, `std::thread`, `println!`, environment variables, process arguments, or a libc syscall table.

`capos-rt` owns the repeated runtime machinery:

- the `_start` entry point and `capos_rt_main` handoff;
- fixed heap initialization;
- panic output through an emergency Console path when available;
- raw `exit` and `cap_enter` syscall wrappers;
- CapSet lookup and interface-id checks;
- a single-owner ring client;
- typed clients for implemented kernel and service capabilities;
- result-cap adoption and queued local release.

Native programs should keep ordinary Rust business logic in normal modules and push OS interaction to typed capOS clients. That keeps pure logic host-testable while making authority visible at capability lookup and child-spawn sites.

### Why std Is Different

Rust `std` is not just "more Rust." It is an operating-system binding. It expects an implementation of filesystem, networking, threads, time, standard I/O, process, environment, synchronization, and platform error APIs. On Linux those calls are ambient: a process can ask the kernel to open a path or create a socket and the kernel consults global process credentials.

capOS does not have that ambient authority model. A future Rust `std` path must choose how each `std` feature gets authority:

| std area | capOS authority source |
| --- | --- |
| `std::io::{stdin, stdout, stderr}` | `StdIO`, `Console`, or `TerminalSession` caps |
| `std::fs` | scoped `Directory`, `File`, `Store`, or `Namespace` caps |
| `std::net` | socket or listener caps minted by a network service |
| `std::thread` | `ThreadSpawner`, `ThreadControl`, `ThreadHandle`, and ParkSpace support |
| `std::time` | `Timer` and future wall-clock caps |
| process spawn and wait | `ProcessSpawner`, `RestrictedLauncher`, and `ProcessHandle` caps |
| `std::env` and current directory | synthetic runtime state backed by manifest or namespace caps |

That mapping can be implemented as a capOS `std` backend, a Rust compatibility crate, or a POSIX-style adapter. The project has not selected one shared ABI for all language runtimes.

### Compatibility Terms

Use these terms instead of the vague phrase "compatibility layer":

- **Native runtime adapter**: language-specific runtime glue that talks to capOS capabilities directly. `capos-rt` is the implemented Rust example; `GOOS=capos` would be the Go example.
- **Capability-native bindings**: generated or handwritten bindings that expose Cap'n Proto interfaces as language-level APIs without POSIX names.
- **POSIX compatibility adapter**: a libc or library surface that translates `open`, `read`, `write`, `socket`, `poll`, `clock_gettime`, and similar APIs into operations on granted capabilities.
- **WASI host adapter**: a WebAssembly host implementation whose imports are backed by granted capOS capabilities.

The adapter may make code look familiar, but it cannot create authority. A process without a namespace cap still cannot open a file. A process without a network cap still cannot create a socket. A process without a launcher or spawner cap still cannot create children.

### Language Tracks

#### Rust

Rust is the only implemented userspace language. The current target is `targets/x86_64-unknown-capos.json`, which exposes `target_os = "capos"` while keeping the booted userspace baseline `no_std`, static, and `panic = "abort"`. `init`, `demos`, `shell`, and the `capos-rt` smoke binary build through this custom target.

Open work before broader Rust support:

- generated clients after the schema surface stabilizes;
- runtime ParkSpace clients and multi-threaded ring demultiplexing;
- a decision on Rust `std` over native capabilities versus a POSIX adapter;
- package/build conventions for out-of-tree capOS Rust programs.

#### C and C++

C support is in tree as a Phase `0` substrate. The `libcapos/` crate compiles to `libcapos.a`, a thin Rust staticlib that exposes the capos-rt syscall, ring CALL, CapSet lookup, typed `Console.writeLine`, `Timer.now`, `EntropySource.fill`, VirtualMemory wrappers, native `ProcessSpawner.createPipe` / `Pipe` wrappers, and the global allocator under an `extern "C"` ABI. C binaries link statically against the archive and run on the same userspace ELF layout as Rust demos; `make run-c-hello` boots a C `main()` that calls the baseline wrappers, and `make run-c-pipe` boots a C `main()` that creates a Pipe, round-trips a marker, closes the writer, and observes EOF. The substrate is intentionally narrow – no `errno`, no fd table, no POSIX surface – so the separate `libcapos-posix` layer can own those decisions without churning the substrate. The same archive is what later runtimes such as CPython, MicroPython, Lua, and QuickJS will link against.

`libcapos-posix/` builds `libcapos_posix.a` on top of `libcapos.a` and ships the `v0` POSIX surface: a 32-fd static table, `__errno_location()` TLS, UDP `socket`/`sendto`/`recvfrom`/`close` over the kernel `UdpSocket` cap, `clock_gettime(CLOCK_MONOTONIC, ...)` and `gettimeofday(&tv,`

NULL) over the kernel Timer cap, pipe/read/write/dup/dup2 over the kernel Pipe cap, file/directory fd operations (open, lseek, opendir, readdir, closedir) over the RAM-backed root Directory cap, and a recording-shim fork/execve/waitpid/_exit/posix_inherit_stdio path plus direct posix_spawn with posix_spawn_file_actions support, all routed through ProcessSpawner.createPipe / ProcessSpawner.spawn when spawning is needed. The shipped smokes are make run-posix-pipe-smoke, make run-posix-spawn-smoke, make run-posix-stdio-smoke, and make run-posix-file. The former make run-posix-dns-smoke target is retired with the qemu-only kernel UdpSocket owner. The remaining v0 phase is the dash port (Phase P1.4) over the kernel RAM-backed File/Directory/Store/Namespace caps from Storage Phase 3 slices 1-3. See docs/backlog/posix-adapter-dash-port.md for the long-form decomposition.

C++ should wait until the C substrate exists and the project decides its C++ ABI policy: exceptions, RTTI, TLS, allocation, unwind behavior, and standard library scope. A freestanding container/arena subset is plausible earlier than full hosted C++.

### Go

Go is a dedicated future design because its runtime is close to a userspace operating system. A native GOOS=capos port needs virtual memory reservation and commitment, TLS setup, OS-thread creation, park/wake, monotonic time, debug output, process exit, and eventually network polling.

The current kernel/runtime substrate already proves useful pieces: VirtualMemory, Timer, ThreadControl, ThreadSpawner, ThreadHandle, and private ParkSpace wait/wake exist at the capOS level. The missing work is the Go runtime port and the runtime-side integration contract, not a new ambient syscall namespace.

Go through WASI may be sufficient for CPU-bound tools such as CUE evaluation; that path is tracked as Phase W.8 in Proposal: WASI Host Adapter (TinyGo or upstream Go GOOS=wasip1 against a future ScriptPackage cap). Native GOOS=capos remains the path for Go network services and full runtime behavior.

### Python

Python is not currently supported on booted capOS. The practical paths are:

- **Native CPython through a POSIX compatibility adapter.** This depends on the C/libc substrate plus file, stdio, timer, networking, and process adapters. It is the likely path for trusted system scripts, configuration tooling, and Python programs that need capOS networking or storage.
- **MicroPython through the same native C substrate.** This is a smaller early scripting option with less runtime surface than CPython.
- **WASI or Emscripten-hosted Python.** This is useful for sandboxed or compute-oriented Python. It still runs a Python interpreter; WebAssembly is the sandbox/host ABI, not a way to avoid Python runtime work.

Current upstream CPython support is relevant but not sufficient by itself: PEP 11 lists wasm32-unknown-wasip1 as a Tier 2 CPython platform and wasm32-unknown-emscripten as Tier 3, while

PEP 776 records Emscripten support for Python 3.14. Those targets help the WASM path. They do not provide native capOS file, socket, thread, or capability bindings.

**Lua**

Lua is a capability-scoped scripting runner. The target is not a POSIX Lua shell. A `capos-lua` process should receive an exact CapSet, load curated standard libraries, expose capabilities as unforgeable host userdata, deny raw CapIds, and flush owned handles at script exit.

Phase 0 lives in `demos/lua-smoke/` as a hand-written Lua-subset interpreter written entirely on top of `capos-rt`. It exists to validate the long-term capability-aware host API design (typed userdata, `obj:method(args)` dispatch through a host registry, no raw SQE or method-id leak into Lua, errors surfaced as Lua runtime errors) without committing capOS to a particular Lua dialect. The interpreter accepts a strict subset (`local`, `if`/`elseif`/ `else`, numeric `for`, `while`, integer/float arithmetic, string concat, comparison, `obj:method(args)` calls); tables, closures, coroutines, metatables, and the Lua standard library are not implemented.

Upstream PUC Lua is a small C implementation, so the dialect-compatible path waits on the C/ libcapos substrate. The Phase 0 interpreter is not a promise of PUC Lua compatibility and the smoke binary is explicitly labelled `runtime = "capos-lua-subset"` rather than `lua-5.x`. When the C/libcapos port lands, the embedded interpreter is replaced or kept as a research-grade sandbox; the host binding shape stays.

**JavaScript and TypeScript**

JavaScript support means running an engine as an ordinary capOS process. A small QuickJS-style runner is the plausible early native path once C support exists. V8 or SpiderMonkey are much larger C++ runtime ports and should be treated as later experiments. TypeScript would normally compile before execution; capOS should not make a TypeScript compiler part of the kernel or base runtime.

**WASI and Browser WebAssembly**

WASI support is a host-runtime track: the host imports become capability calls. The full design is in the WASI Host Adapter proposal, and the implementation decomposition is in Proposal: WASI Host Adapter. The proposal selects wasmi for the v0 phases (`no_std + alloc` userspace runtime, fuel metering, externref support) and frames wasmtime / WAMR as the W.7+ migration targets. Each WASI import is backed by a typed capOS capability the host adapter already holds; ungranted authority is refused, not synthesised. WASI is a good fit for code that is already designed around explicit imports and sandboxed execution. It is not a replacement for native runtime ports when the language expects OS threads, signals, sockets, memory mapping, or a large POSIX surface.

The browser/WebAssembly proposal is separate. It explores running capOS concepts in a browser using worker-per-process isolation and SharedArrayBuffer-backed rings. It is a teaching and demo target, not current native userspace language support.

### Proposal Map

- Userspace Runtime documents the current `capos-rt` implementation.
- Userspace Binaries owns the native binary, language, POSIX-adapter, and WASI roadmap.

- Go Runtime owns the native Go plan.
- Go VirtualMemory Contract freezes the allocator-facing memory contract needed by Go-style runtimes.
- Lua Scripting owns the Lua runner design.
- WASI Host Adapter owns the WebAssembly host adapter design; the implementation decomposition lives in Proposal: WASI Host Adapter.
- POSIX Adapter owns the POSIX-compatibility roadmap above the libcapos C-substrate; the implementation decomposition lives in Proposal: POSIX Compatibility Adapter.
- Shell distinguishes native shell behavior from POSIX shell compatibility.
- Storage and Naming defines the `Directory`, `File`, `Store`, and `Namespace` surfaces that future POSIX and language runtimes will consume.
- Browser/WASM owns the browser-hosted WebAssembly experiment.
- LLVM Target records target-triple, Rust `std`, Go, C, TLS, and ABI grounding.

### Validation

Current language-runtime validation is Rust-only:

- `tools/check-userspace-runtime-surface.sh` verifies that `capos-rt` owns `_start`, panic handling, allocator setup, raw syscalls, and entry macros.
- `make capos-rt-check`, `make init-capos-build`, `make demos-capos-build`, `make shell-capos-build`, and `make capos-rt-capos-build` build the booted userspace artifacts against the capOS custom target.
- `make run-smoke`, `make run-spawn`, `make run-shell`, and `make run-terminal` exercise the runtime surface through QEMU.

No page should claim support for Python, Go, Lua, C, C++, JavaScript, WASI, or Rust `std` until there is a booted artifact and a validation target for that runtime.

# Part II: Operator Demos

Runnable demonstrations that show the current shell, service, and game surfaces.

## First Chat Demo

The First Chat demo is the smallest runnable multi-process service demo in capOS. It boots a resident `chat-server`, a bounded `chat-bot` actor, and a native shell that can launch `chat-client` with explicit `StdIO` plus the broker-issued operator `Chat` endpoint grant.

The chat service is not a shell builtin. The shell only launches a client process and services that client's `StdIO` endpoint while the client talks to the resident `Chat` endpoint. The focused manifest routes the kernel singleton `chat_endpoint` through init to `chat-server`, which is the same endpoint the broker facets into operator shell bundles.

### Run It

Use the focused QEMU proof:

```
make run-chat
```

The scripted proof creates a volatile shell credential, rejects an attempted client endpoint relabel, launches `chat-client` under the authenticated shell session, sends one lobby message, checks membership with `/who`, observes the resident bot reply, quits the client, and exits the shell. The terminal transcript should include:

```
[chat] /join <channel>, /leave, /who, /exit, or plain text
[chat:#lobby]> hello from shell
[chat] #lobby <member-2> hello from shell
[chat] #lobby <member-1> [chat-bot] echo-bot heard you.
```

For default manual use, boot the ordinary playground:

```
make run
```

After login:

```
run "chat-client" with { stdio: client @stdio, chat: client @chat }
```

The default playground starts the resident `chat-server` and includes `chat-client`, but it does not start the bounded `chat-bot` proof actor. Use `make run-chat` when you need the one-shot echo-bot transcript.

For lower-level manual proof work, let `make run-chat` build the focused ISO, then boot `capos-chat.iso` yourself with the terminal UART attached to stdio and the console UART written to a log.

Useful client commands:

```
/join #other
/who
/leave
/exit
plain chat text
```

The resident bot is a bounded proof actor. If the operator waits too long before joining and sending the first lobby message, the bot can time out and exit; the chat client and server remain usable, but the bot reply will no longer appear.

### What It Demonstrates

`make run-chat` and the manual terminal path described above currently show:

- `chat-server` runs as a resident service exporting only the `Chat` endpoint;
- `chat-server` keys membership by the opaque caller-session reference in the endpoint metadata, not by a caller-selected endpoint badge;
- `chat-bot` is a separate participant with a delegated chat client endpoint and its own session-bound membership record;
- `capos-shell` launches `chat-client` as an ordinary userspace process;
- the foreground client receives only explicit `StdIO` and `Chat` grants;
- caller-selected endpoint relabeling is rejected for delegated chat clients;
- the `handle` supplied to `join` is request data only; the service assigns visible `member-N` labels and the handle does not select membership authority or sender identity;
- lobby messages and bot replies are visible through the terminal transcript;
- `/who` lists current channel members from the resident service;
- client exit returns to the shell prompt, and the manifest child wait path observes clean shell and bot exits during normal completion.

### Current Limits

This is not yet a distinct-local-user chat surface over Telnet or multiple terminals.

`system.cue` and `system-chat.cue` each boot one terminal-backed shell on the QEMU terminal UART, and the shell's run command waits on the foreground client's `StdIO` endpoint. Multiple `chat-client` runs can reuse the resident service, but the current manual flow is one foreground client at a time. The demo client still sends the hard-coded join handle `shell` for compatibility; the server ignores it for visible sender labels and does not request disclosed display/profile metadata from the session broker yet.

The default `make run` foreground shell now receives its shell bundle from `AuthorityBroker`, including a profile-scoped `chat` endpoint for operator shells. Guest and anonymous shells do not receive `chat` by default. An operator shell can therefore run the same `chat-client` command after login. This is still not a distinct durable user chat surface: the demo client joins with the hard-coded handle `shell`, the server assigns its own visible member label, and multiple terminal sessions still need a multi-session terminal host or network gateway before they are a real multi-user chat model.

To make distinct local users chat through Telnet or terminals, capOS still needs a multi-session terminal host or Telnet gateway that can keep multiple shell sessions alive, grant each session a broker-authorized chat root/facet, and disclose only the bounded display/profile metadata the user or broker explicitly permits.

## Aurelian Frontier — Proof Slice

This page describes the current runnable proof slice of the Aurelian Frontier game. It is the end-to-end example of a capOS-native interactive application: a Roman-frontier text adventure with magic wards, warrior skills, wizard spells, NPC chat history, per-player state, and explicit capability grants. The wider game design lives in Aurelian Frontier this page covers what runs today and how the QEMU smoke proves it.

Unlike a shell builtin, the game runs as ordinary userspace processes:

- `capos-shell` launches `adventure-client` with only `StdIO`, `Adventure`, and `Chat` client capabilities.
- `adventure-server` owns room, inventory, writ, combat, evidence, and effect state keyed by the endpoint caller-session scoped reference and epoch, while consuming validated read-only prototype mission content generated from `adventure-content` CUE source.
- `chat-server` carries room messages and labels replayed room history so NPC actors do not treat old messages as fresh input.
- `adventure-npc-wanderer` and `adventure-npc-shopkeeper` prove that separate actors can join the shared ashen-road channel without receiving ambient game authority.
- `adventure-scenario-test` is a noninteractive capOS userspace test process with only `Console` and `Adventure` caps. It drives the custody scenario through `AdventureClient` RPCs and prints a console success marker.

### Run It

Use the focused QEMU proof:

```
make run-adventure
```

The scripted run creates a volatile shell credential, launches the interactive adventure client for representative rendering and command coverage, and also asserts the resident `adventure-scenario-test` success marker and exit status for the complex custody path.

`run "adventure-client"` starts from a fresh expedition view by default. Use the client's `resume` command to return to that session's active expedition state instead of silently continuing it on launch.

For the default init-owned boot, start `make run`, log in or run setup, then use the MOTD compatibility commands:

```
spawn "chat-server" with { console: @console, chat: @chat } -> $chat
spawn "adventure-server" with { console: @console, adventure:
@adventure, chat: client @chat } -> $adventure
spawn "adventure-npc-wanderer" with { console: @console, chat: client
@chat } -> $wanderer
spawn "adventure-npc-shopkeeper" with { console: @console, chat: client
@chat } -> $shopkeeper
run "adventure-client" with { stdio: client @stdio, adventure: client
@adventure, chat: client @chat }
```

Normal launch commands omit legacy receiver selectors; delegated client endpoint identity is preserved by default. The adventure server derives player state from live session-bound endpoint caller metadata. The focused `make run-adventure` proof is the authoritative regression path. Its manifest uses selector-free Adventure and chat endpoint grants, while hostile and lower-level smokes retain explicit legacy selector fixtures for rejection coverage.

**Current Mission**

The implemented mission starts in `fort_aurelian`, crosses `gate_yard` and `ashen_road`, and reaches `signal_tower`, with `under_vault` present as a bounded site in the generated graph. The player can request and delegate a `ward-writ`, ask actors about the mission, quote and buy Maro's route support, fight a `ward-wraith`, order Livia to expose the tower sigil, recover `eagle-standard`, record a wounded-legionary evacuation, seal the gate-yard breach, and get Iunia's witness-certified `temple-seal` custody. Room views show canonical room, exit, actor, mob, and writ ids alongside the current mission and lead. Status and inventory separate survival, location, mission, physical items, writs, relic custody, marks, evidence, effects, and the next lead; status also prints the fixed smoke seed calendar (`ashfall` day 9, `ash-wind`, `ward-static`), a bounded seasonal resource count/cap summary, and a carried seasonal-resource forecast that names the next season's degraded and expired counts. The current gameplay slice also lets active collectible seasonal resources be taken at their site; carried crops, fish, and forage participate in the next-season aging rule, while active repair-material resources can be harvested without being treated as fragile seasonal carry items. `ask quartermaster about season-transition` applies that aging rule: expired crops are removed, fish/forage degrade to explicit `-degraded` inventory tokens, and unknown or non-seasonal items stay unchanged. After the audited debrief grants Aurelian standing, the quartermaster can sell one bounded `field-ration` from the fixed-smoke per-expedition seasonal stock, spending that standing and adding the ration to inventory. Ordinary inventory is currently bounded to six slots. This is not a full seasonal economy or persistent calendar advance. Status also prints the active generated calendar event metadata for the fixed seed: the `lantern-vigil` festival's actor-location, shop, witness, route, and rumor overlays. These event fields are metadata/status only; actor movement, event-driven shop mutation, witness blocking, route safety mutation, debrief branching, quests, gifts, and affection are not implemented. Status also prints active generated actor routine metadata for named actors, selected from the fixed calendar plus the current mission and emergency state: actor id, room id, routine kind/trigger, schedule/effect text, authority stance, and metadata-only gameplay stance. These routine records do not move actors or grant/revoke authority. Status also prints a concise regional frontier summary for the generated settlement, outpost, and route metadata, plus a concise regional market order-book summary for generated market books, buy/sell orders, crossed pure matches, and receipt-ledger ids. Market-eligible items are limited to ordinary seasonal resources, construction materials, and explicit outpost produced/consumed supplies. Writs, relics, actors, mobs, spells, skills, order tasks, and artifact/authority-gated blueprint outputs are excluded. The first live regional market transaction proof is bounded to one generated order-book match at a time: Adventure owns reserve, commit, cancel/release, stale-version rejection, idempotent replay, and ordered receipt facts behind existing `quote`, `buy`, and `sell` calls for explicit `regional-market` proof actions. Fresh committed field-ration matches now debit the player-local Aurelian chit balance once, decrement the seller `ash_farm` field-ration stock once, accrue two service-owned regional market fee chits once, credit two service-owned

ash_farm seller-proceeds chits once, and deliver the committed quantity into the player expedition inventory only when ordinary inventory capacity can accept the full delivery; if capacity is full, replaying buy commit-field-ration from regional-market can apply the held delivery after ordinary items are dropped without spending, decrementing stock, accruing fees, or crediting proceeds again. Commit replay does not duplicate delivery, debit, outpost stock movement, fee accrual, or seller proceeds. Commit 29c065a9 at 2026-04-30 17:41 UTC added bounded order expiry to live matching and reserve: fixed-smoke day 65 keeps the field-ration proof active, while the scenario process proves a day-73 expired field-ration reserve releases without status, inventory, currency, outpost stock, fee, seller-proceeds, or delivery mutation. Commit 205fd6a0 at 2026-04-30 18:40 UTC added a bounded service-owned fee withdrawal proof: sell withdraw-fees to regional-market moves the two accrued regional-market fee chits into a service-owned treasury record exactly once, status exposes the treasury balance, replay is stable, and inventory, currency, outpost stock, seller proceeds, and delivery state do not mutate. Commit a547db3d at 2026-04-30 19:43 UTC adds a bounded receipt snapshot/restore proof: buy receipt-snapshot from regional-market clones the live regional market receipt facts, reconstructs a separate transaction state, replays the old field-ration commit against that reconstructed state, and returns proof success without mutating live status or inventory. Commit 4b44b32 at 2026-04-30 20:07 UTC adds a bounded settlement-side snapshot-view proof: buy settlement-snapshot from regional-market checks applied delivery, debit, stock, fee, proceeds, and withdrawal ids plus the current settlement balances, replays the committed field-ration fact and fee withdrawal as already applied, and returns proof success without mutating live status or inventory. The construction-job receipt snapshot work is scoped to pure Rust construction receipt snapshot semantics plus a size-constrained QEMU no-mutation probe. Pure adventure-content tests restore a separate ConstructionJobState from ordered field-repair job facts and validate malformed, over-capacity, and non-closed snapshot shapes. The focused QEMU path drives repair receipt-snapshot with field-engineer after the old completed repair only to check status/inventory stability and confirm live construction state and material stock are unchanged. The runtime command does not replay receipts into the live construction service and is not durable restart loading or a general construction persistence layer. It does not yet move NPC stores, broader outpost inventories, durable currency ledgers, durable seller-proceeds ledgers, profile ledger balances, fee ledgers, durable calendar advancement, durable crash-recovery state, or general economy behavior. Status also prints a construction foundation summary for generated blueprint, artifact, enchantment slot, and gate metadata; the first live construction-job proof is bounded to the field-engineer gate repair path: Adventure owns reserve/start, completion, cancel/release, stale-version rejection, idempotent replay, service-owned material holds and restores, and ordered job facts behind existing repair calls. It does not yet persist durable stock ledgers, replenish stock from outposts, update player output/currency inventories, advance job time, persist crash-recovery state, or provide general crafting/artifact gameplay. Status now also prints disabled-by-default optional fake-agent NPC metadata: budget count, supported fake-agent purpose count, aggregate session token budget, tool-call budget, and audit visibility. That is deterministic metadata for future optional chatter, hints, outpost summaries, personal routines, nonbinding shop flavor, and festival reactions, not live LLM gameplay or autonomous NPC authority. Status also prints the first local party foundation: a service-created local player label, the current party leader/members/pending invites, scoped ward-writ delegations, and recorded assists. Party labels are derived from live Adventure caller-

session keys and do not disclose global session or principal data. The same service-local labels are used by the first physical-item transfer foundation, `transfer <item> to <player>`, which mutates both player inventories atomically inside `Adventure`, requires shared party membership, and refuses relic custody such as `eagle-standard`. Currency escrow and two-client transfer proof remain future work. Valid near-miss ids such as `ward` and `wraith` return explicit suggestions. The site graph, regional metadata, visible items, actors, mobs, aliases, objectives, mission text, leads, scripted proof-path metadata, named-item inspection text, and prepared-spell inspection text are authored in `demos/adventure-content/content/prototype.cue`, generated into `demos/adventure-content/src/generated.rs`, and validated by host tests before the server consumes them.

Useful commands in the current game:

```
look
resume
status
request ward
request ward-writ
accept ward-writ
delegate ward-writ to livia
order Livia to guard
go east
go east
say hello road
take scout-marker
quote route from maro
buy route from maro
quote regional-field-ration from regional-market
buy reserve-field-ration from regional-market
buy commit-field-ration from regional-market
buy reserve-incense from regional-market
sell cancel-incense to regional-market
sell withdraw-fees to regional-market
transfer scout-marker to player-1
repair gate with field-engineer
repair retry-field-repair with field-engineer
repair complete-field-repair with field-engineer
repair stale-field-repair with field-engineer
repair reserve-cancel-field-repair with field-engineer
repair cancel-field-repair with field-engineer
go north
order livia to dispel-sigil
inspect ward-wraith
cast ember-dart ward-wraith
skill strike ward-wraith
recover eagle-standard
ask wounded-legionary about evacuation
guard
```

```
cast shield-bind self
go south
go west
seal gate
go west
ask iunia about custody
inventory
go down
```

**What It Proves**

`make run-adventure` currently asserts:

- shell-spawned game clients run with explicit `StdIO`, `Adventure`, and `Chat` grants;
- ordinary `adventure-client` launch and `look` start fresh, while the explicit `resume` command reloads active expedition state through an `Adventure` cap call;
- room joins, movement, physical item pickup, typed relic recovery, inventory, status, and representative failure messages are visible in the terminal transcript;
- `give`, `ask`, `request`, `accept`, `delegate`, `order`, `seal`, `recover`, `revoke`, `quote`, `buy`, `sell`, `trade`, `transfer`, and `repair` are wired as typed adventure calls, not shell-special strings;
- `adventure-client` exposes `party create`, `party invite`, `party accept`, `party leave`, `party delegate`, `assist`, and `transfer <item> to <player>` command paths backed by typed Adventure methods;
- the party proof covers one-client party creation, missing local-player refusal paths for invite and assist, party status output, and help/client command availability; two-client successful accept, leave, delegate, and assist calls remain future work;
- the transfer proof covers one-client unknown target, self-transfer, and missing-item refusals, with status or inventory unchanged as appropriate; successful two-player transfer remains covered by pure Rust state tests until the launcher/session harness can run two real Adventure clients;
- canonical room, exit, actor, mob, and writ ids, room-view leads, common actor casing aliases, near-miss suggestions, and improved actor-task hints are visible in the terminal transcript;
- combat status exposes hp, guard, fatigue, warrior stars, wizard circles, prepared spells, active mobs, mission state, physical items, writ authority, relic custody, marks, evidence, effects, fixed smoke seed calendar state, and objective state;
- generated actor routine metadata is visible through status as structured status-only records filtered by the fixed calendar and current mission/emergency state;
- generated regional market order-book metadata is visible through status as aggregate metadata and pure non-mutating crossed-match counts only;
- market and construction coverage proves a Maro route quote, a successful route exchange, an Iunia clean-custody trade refusal that names the `temple-seal` gate and price, a bounded regional-market reserve/commit/retry/stale/release/cancel proof where the server owns the transaction state and receipt facts, and a bounded field-repair construction job proof where the server owns job state, service-owned material hold/release facts, held-stock mutation, and

terminal facts; shell-smoke coverage also keeps the full market command-help surface, including `sell`, visible;

- delegated authority can expose the ward, repeated spell actions are idempotent, and `eagle-standard` recovery records bounded evidence in the interactive transcript;
- the `adventure-scenario-test` process covers physical-item-only `take` and `drop`, carried seasonal resource pickup, quartermaster-triggered seasonal inventory aging, post-debrief seasonal ration purchase, Iunia custody denials, witness refusal, survivor evacuation, gate sealing, `temple-seal` custody, categorized evidence tokens, and `under_vault` access through real `Adventure` cap calls, and asserts the fixed calendar, seasonal carry forecast, regional market delivery/replay, construction foundation, construction-job denial/reserve/replay/open-conflict/complete/stale/release/ reserve-after-release paths, agent NPC budget, and one-client party status lines through real `Adventure` cap calls;
- the two-client local co-op proof remains open because the current focused manifest/session launcher path does not yet provide two distinct live Adventure caller-session keys without faking them inside one process;
- replayed room messages are labeled as history, and the named NPC actor proof accepts visible replies whether the player observes them live or through room-history replay after movement;
- the read-only prototype content model rejects malformed room graphs, bad aliases, overlong text, empty proof paths, malformed construction metadata, and invalid agent NPC budget metadata in host tests.
- `make generated-code-check` fails if the checked-in generated adventure content drifts from the CUE source or generator.

### Design Context

The gameplay and future setting plan live in the Aurelian Frontier proposal. The proposal covers the Aurelian frontier setting with magic-warrior and wizard ranks, future mobs, portals, golems, logistics, campaigns, persistent shared world state, multiplayer, and how those mechanics map onto capability-native authority.

## Paperclips Terminal Demo

The Paperclips terminal demo is a small clean-room incremental game inspired by the paperclip maximizer thought experiment and by Frank Lantz's browser implementation of that premise. In the focused manifest it now runs as a Paperclips server plus terminal client launched through the native shell. The server is authoritative for generated content, resources, `GameState`, proof-command gating, unlock checks, and game-rule mutation. The terminal client owns `StdIO`, handles the transcript, renders `help` from server-provided command specs, plain `status` from server-provided `PaperclipsStatusSnapshot` data, and plain `projects` from server-provided project entries when connected to a server, and sends gameplay requests to the server through an explicit `PaperclipsGame` endpoint capability.

This is still an early client/server protocol. The server owns regular timer cadence and the current command list, while command execution still uses raw text and mostly returns transcript text rather than typed command invocations or structured UI events. In server mode, `PaperclipsGame.status` returns a `PaperclipsStatusSnapshot` for plain `status`, and the terminal client renders the familiar status text locally. `PaperclipsGame.projects` likewise returns the unlocked project list for local terminal rendering of plain `projects`, while `project <id>` still executes through the raw text/server-mutating command path. The backlog tracks broader structured state/events and moving unlocked command facets behind server-issued capabilities so a future web client or web-shell gateway can use the same game authority instead of reimplementing Paperclips logic.

No source code, CSS, images, generated tables, or copied resource files from the original browser game are checked into this repository. The implementation uses original Rust code and local CUE content in `demos/paperclips-content/content/paperclips.cue`. During development, the original site and a public mirror were inspected for license/provenance only; neither exposed a permissive license that would allow copying assets into capOS.

Reference sources:

- Original game site: https://www.decisionproblem.com/paperclips/
- Public mirror inspected for license/provenance: https://github.com/jgmize/paperclips
- Public gameplay/stage summaries: https://universalpaperclips.fandom.com/wiki/Stages
- Background overview: https://en.wikipedia.org/wiki/Universal_Paperclips

### Run It

Use the focused QEMU proof:

```
make run-paperclips
```

The scripted proof logs into the shell, launches the child process, drives the opening refusals and business loop, scales production through repeatable marketing and explicit sales, completes a business-phase project chain, asserts the transition to autonomous phase, completes a representative autonomous drone/factory scaling step, transitions into the cosmic phase, proves a bounded probe interval with replication and production, verifies that final conversion remains locked, and then checks clean child and shell exit. The accelerated proof transcript starts with an

explicit proof-capability launch, then uses ordinary player commands plus proof-only acceleration and machine-status commands:

```
run "paperclips" with { stdio: client @stdio, game: client
@paperclips_proof_game, proof_accelerator: @proof_accelerator }
status
buy autoclipper
buy wire 1000
buy marketing
make
run 10000
price 99
sell 1
price 25
sell 1
make
run 10000
sell 1
make
run 10000
sell 1
make
run 10000
sell 1
make
run 10000
sell 1
...
project autoclipper-license
project background-jobs
status
run 5000
buy wire 2
run 600000
make
projects
project survey-drones
sell 60
buy marketing
...
project precision-rollers
project design-search
run 600000
project forecast-engine
project survey-drones
project material-harvesters
run 100000
project foundry-lines
```

```
run 1000
project mesh-coordination
run 600000
project seed-probes
run 600000
status --json
status
projects
exit
```

For default manual use, boot the ordinary playground:

```
make run
```

After login:

```
run "paperclips" with { stdio: client @stdio, timer: @timer }
```

The ordinary `make run` playground command uses the standalone fallback because the default manifest does not start the Paperclips server. The focused `make run-paperclips` manifest uses `run "paperclips" with { stdio: client @stdio, game: client @paperclips_game, timer: @timer }`, where the server owns game state and the client timer drains server-generated status messages while the player is idle at the prompt. The structured command-list, status-snapshot, and project-list methods do not change the default manifest or MOTD launch command.

Useful commands inside the demo:

```
status
projects
make
sell <n>
price <cents>
buy wire [bundles]
buy autoclipper [count]
buy marketing [count]
buy processor [count]
buy memory [count]
buy drone [count]
buy factory [count]
buy probe [count]
project <id>
help
exit
```

`make` starts exactly one manual paperclip. Manual work takes 500 configured milliseconds before the clip becomes available. Repeating `make` while work is pending is refused until the player completes the Background Jobs project; after that, repeated `make` commands reserve wire and queue manual jobs behind the active one. Time advancement reports completed manual cycles

before the status update. Purchase counts are optional and default to one; explicit zero counts are rejected. Automation advances on configured millisecond intervals while the process is running. Normal player launches do not expose run <ms> or `status --json`; the focused QEMU proof passes an explicit `proof_accelerator` capability and uses those commands only as proof instrumentation. The shell rejects renaming an ordinary `@timer` grant into that proof slot. Blank input repeats the last non-empty command. The first autoclipper is granted by the Autoclipper License project, which costs cash and trust; repeatable `buy autoclipper [count]` purchases appear only after the license grants the starter autoclipper. Later-stage purchase commands such as `buy drone`, `buy factory`, and `buy probe` appear in `help` only after the corresponding automation path is unlocked. The `projects` list shows only unlocked technologies; in server mode that plain listing is rendered from structured server-provided project entries. Complete the listed projects and pay their shown costs to reveal later project chains.

The proof-only `status --json` command prints a single compact JSON object for scripted assertions when the process receives the `proof_accelerator` capability. Normal player launches do not advertise or accept it, and it stays separate from the structured plain-status snapshot used for terminal presentation. All fields are numeric and emitted in stable order. `stage` uses `0=business`, `1=autonomous`, `2=cosmic`, and `3=complete`; `design` and `strategy` are the two planning resources, and `cosmic_matter` maps to the universe-matter state.

Funds change only when clips are sold explicitly by default. Demand follows a bounded random walk during the business phase, then price modifies the current market size for sell <n>. A successful business-phase sale starts a short CUE-configured market-settlement cooldown, so repeated immediate sales are refused until timer/proof time advances. Wire is bought in CUE-configured bundles at a market price that drifts on a slower interval; repeated purchases add temporary price pressure that decays over later market updates. Repeatable marketing buys still spend funds, but each new level contributes more demand than the previous level. The CUE content owns the base marketing gain, walk thresholds, wire market thresholds, step sizes, sale cooldown, and deterministic generator parameters. It also has an `autoSellEnabled` rule for experiments that should sell during ms, but the checked-in demo keeps it disabled so market movement is visible.

### Content Pipeline

Paperclips uses the same generated-content discipline expected for larger demos, but with a stricter runtime data path:

```
demos/paperclips-content/content/paperclips.cue
  -> cue export --out json
  -> tools/paperclips-content-gen
  -> schema-validated PaperclipsContent Cap'n Proto bytes in src/
generated.rs
  -> paperclips-content deserializes the typed Paperclips schema at
startup
```

The CUE file owns the game balance: initial state, purchase costs, millisecond intervals, explicit/automatic selling policy, demand rules, trust milestones, project costs, project labels/descriptions, production cadence, later-stage matter conversion and replication caps, manual-work pacing,

unlock thresholds, and project effects. Rust owns mechanics, validation, command parsing, and the terminal adapter. `make generated-code-check` fails if the checked-in generated Cap'n Proto bytes drift from the CUE source.

### Unlock Flow

The tech progression is data-driven by the project list:

- retail phase starts with 10 wire, no cash, manual single-clip production, sales, and early wire purchases;
- Autoclipper License grants the first autoclipper for cash plus trust, then Background Jobs enables queued manual `make` commands;
- repeatable marketing investment raises dynamic demand, while later business projects improve autoclippers, unlock design search, and unlock Strategy generation;
- Survey Drones moves the game into autonomous matter conversion;
- harvester/foundry/mesh projects scale harvesting, production, and compute;
- Seed Probes move the game into cosmic replication;
- Final Conversion completes the run once reachable matter is exhausted.

### What It Demonstrates

`make run-paperclips` currently shows:

- `capos-shell` launches `paperclips` as a normal child process;
- init launches Paperclips server services before the shell starts;
- the terminal client receives only explicit `StdIO` and `PaperclipsGame` endpoint grants;
- server-mode `help` is rendered from the Paperclips server's structured command specs, so the visible command list follows server-side unlock/proof authority;
- server-mode plain `status` is rendered by the terminal client from the Paperclips server's `PaperclipsStatusSnapshot`, while proof-only `status --json` stays a separate server-gated command;
- server-mode plain `projects` is rendered by the terminal client from the Paperclips server's structured project list, while `project <id>` execution stays on the server-mutating text command path;
- the normal and proof launches use separate server endpoints, so proof-only commands are decided by server-side authority rather than by client text;
- the foreground shell services the child's stdio bridge while the game runs, so the demo exercises real endpoint IPC between shell and child process;
- the server's timer capability drives regular automation without ambient clock access by the terminal client;
- the Paperclips server owns generated content, `GameState`, unlock checks, proof-command gating, and game-rule mutation for the focused manifest;
- a repeatable economic choice (`buy marketing`) changes the early business loop before automation is purchased;

- representative Stage 1 refusal output remains legible: early locked `buy autoclipper`, insufficient-funds `buy wire 1000`, pending manual work, bulk manual rejection, a high-price `sell 1` demand refusal, a no-wire manual production refusal, and a locked `project survey-drones` attempt are asserted in the focused transcript;
- manual production and explicit sales fund Autoclipper License, which grants the first autoclipper and unlocks repeatable `buy autoclipper`;
- repeatable demand investment remains a purchase path rather than a one-shot project, and the smoke asserts at least five marketing purchases before the phase transition path completes;
- business-phase sales are paced by a timer-backed market-settlement cooldown, and the smoke asserts an immediate repeat sale is refused;
- scaled business-phase production reaches the 10,000-clip trust threshold, then completes `autoclipper-license`, `precision-rollers`, `design-search`, `forecast-engine`, and `survey-drones`;
- completing the chain is asserted by `[done]` project entries, the visible `== autonomous phase ==` status line, `Automation: 14 autoclippers, 1 drones, 0 factories, 0 probes`, and the local matter grant;
- the autonomous follow-up completes `material-harvesters` and `foundry-lines`, runs milliseconds, then asserts `Automation: 14 autoclippers, 5 drones, 2 factories, 0 probes`, lower local matter, and additional clip production;
- the late-game follow-up completes `mesh-coordination`, then `seed-probes`, asserts `== cosmic phase ==`, visible probe replication, lower cosmic matter, and additional clip production, then asserts `final-conversion` remains locked;
- the late-game proof also asserts a proof-only `status --json` line with compact, machine-readable numeric state while preserving the human transcript checks;
- the Paperclips server maintains game state without ambient authority;
- the pure rules layer in `paperclips-content` is host-testable separately from the terminal adapter and reads generated Cap'n Proto content data;
- exiting the game closes stdio, returns to the shell, and lets the focused manifest halt through the normal debug-exit path.

This is now a coarse client/server game-state demo. It is not yet the final capability-management showcase: help, plain status, and plain projects are structured, command execution is still raw text, broader state/events remain future work, and unlocks are reflected in server-owned command specs/project lists/output rather than transferred command facets. That split is the intended path for a later web client or gateway that uses the same game capabilities.

### Current Limits

The demo intentionally implements a compact terminal adaptation, not a browser-accurate port. It has no original artwork, CSS, JavaScript, exact project list, exact balancing, save file, market UI, tournament model, or complete original event text. The host tests cover early mechanics, project locking, the deterministic business-to-autonomous project chain, autonomous resource conversion caps, factory/drone scaling, cosmic probe replication, and completion gating, including a one-real-time-hour upper bound for normal creativity generation. The focused

QEMU proof covers launch, the first production loop, one early automation purchase, representative Stage 1 refusal output, business-phase project chaining, the autonomous transition, one timer-driven autonomous scaling action, and a bounded cosmic probe interval. It is representative transcript coverage rather than an exhaustive full playthrough.

Future rule/content expansion is tracked in Paperclips Terminal Demo Backlog. New data-heavy content should migrate through `mkmanifest cue-to-capnp`: author bounded CUE, convert it to a specified Cap'n Proto root with pinned host tools, validate the result on the host, and keep runtime CUE parsing out of the demo.

# Part III: Architecture

Boot flow, process structure, capability transport, IPC, authority accounting, memory, and scheduling.

## Current Design Authority

The current capOS design lives in reader-facing architecture, capability, security, device, configuration, and status pages. Proposal documents remain important design history, but they stop being the primary place to patch a design after that design is implemented or accepted as the working baseline.

### Stable Homes

Use these homes for current behavior and accepted contracts:

- **Area:** Boot, manifest, init, processes, rings, IPC, session context, memory, scheduling
  - **Current-design home:** `docs/architecture/`
- **Area:** Capability model, authority accounting, ABI policy
  - **Current-design home:** `docs/capability-model.md`, `docs/authority-accounting-transfer-design.md`, `docs/abi-evolution-policy.md`
- **Area:** Operator configuration and CUE overlays
  - **Current-design home:** `docs/configuration.md`
- **Area:** DMA isolation, device authority, trusted inputs, panic surfaces
  - **Current-design home:** `docs/dma-isolation-design.md`, `docs/devices/`, `docs/trusted-build-inputs.md`, `docs/panic-surface-inventory.md`
- **Area:** Current status, roadmap, backlog, task lifecycle
  - **Current-design home:** `docs/status.md`, `docs/roadmap.md`, `docs/backlog/`, `docs/tasks/`
- **Area:** Proposal status and archival decision records
  - **Current-design home:** `docs/proposals/index.md` and individual files under `docs/proposals/`

When a current-design home already exists, future implementation slices update that page. When none exists and the proposal has become the working design, create or extend a stable page in the nearest existing area instead of leaving the proposal as the only current reference.

### Proposal Lifecycle

The proposal index classifies proposals with these roles:

- **Active design**: near-term design work still being changed before or during implementation. It may remain the primary working document while the design is not stable.
- **Accepted design**: selected direction. It can guide implementation, but any implemented subset needs a stable current-design page or an explicit pointer to the page that already owns the current contract.
- **Partially implemented**: some behavior is in tree. The proposal must distinguish present behavior from planned behavior, and current pages should describe the implemented subset.
- **Implemented**: the proposal is an archival decision record. Future changes update the stable current-design docs and code references first; the proposal changes only for archival status, links, or corrected history.

- **Superseded** or **Rejected**: historical records. They should point at the replacement or rejection rationale and must not be cited as current behavior.

### Initial Promotions

This repository already had stable homes for several implemented or accepted designs. The initial promotion set makes the weakest current-authority links explicit:

- **Proposal or decision:** Session-Bound Invocation Context
  - **Current-design authority:** `docs/architecture/session-context.md`, with endpoint transport details in `docs/architecture/ipc-endpoints.md`
  - **Disposition:** Implemented proposal becomes archival history.
- **Proposal or decision:** Error Handling
  - **Current-design authority:** `docs/architecture/error-handling.md`, with ring transport details in `docs/architecture/capability-ring.md`
  - **Disposition:** Implemented proposal becomes archival history.
- **Proposal or decision:** System Configuration
  - **Current-design authority:** `docs/configuration.md` and `docs/architecture/manifest-startup.md`
  - **Disposition:** Implemented proposal stays as rationale and closeout history.
- **Proposal or decision:** DMA Assurance Model
  - **Current-design authority:** `docs/dma-isolation-design.md`
  - **Disposition:** Accepted design remains grounded in the stable DMA design page.
- **Proposal or decision:** SMP and Scheduler Evolution
  - **Current-design authority:** `docs/architecture/threading.md` and `docs/architecture/scheduling.md`
  - **Disposition:** Accepted design feeds current scheduler and threading contracts.

Follow-up promotions should focus on proposals whose implemented slices are large enough that readers still have to mine proposal text for current behavior. Good candidates include storage/naming, installable system, SystemInfo/System Manual, and userspace driver relocation once their current contracts settle further.

## Boot Flow

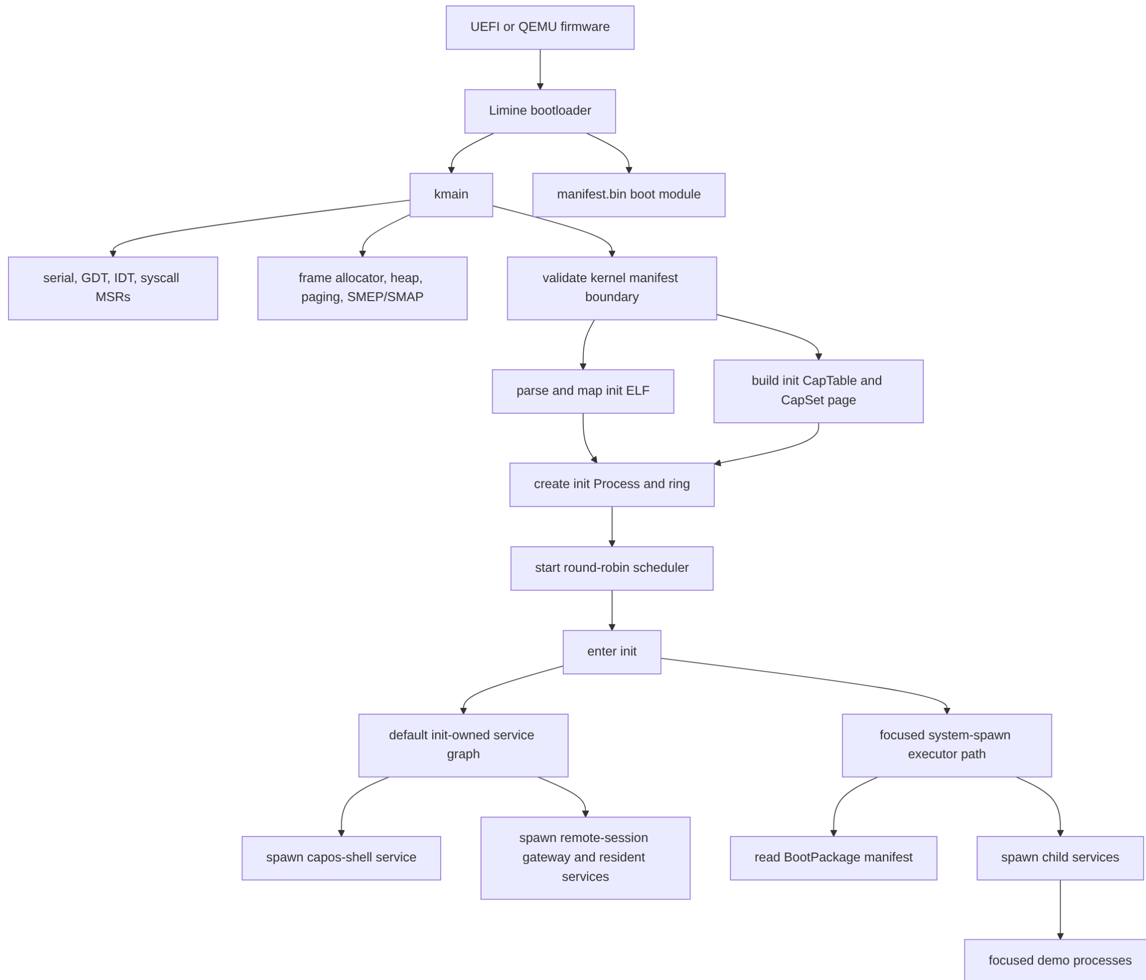
Boot flow defines the trusted path from firmware-owned machine state to the first user processes. It establishes memory management, interrupt/syscall entry, capability tables, process rings, and the boot manifest authority graph.

### Current Behavior

Firmware loads Limine, Limine loads the kernel and exactly one module, and the kernel treats that module as a Cap'n Proto `SystemManifest`. The kernel rejects boots with any module count other than one.

`kmain` initializes serial output, x86_64 descriptor tables, memory, paging, SMEP/SMAP, the kernel capability table, the idle process, PIC, and PIT. It then parses the manifest, validates the kernel-owned boot boundary, loads only `initConfig.init.binary` into a fresh `AddressSpace`, builds init's bootstrap capability table and read-only CapSet page from `initConfig.init.caps`, enqueues init, and starts the scheduler.

Default boot uses the standalone `init` ELF as that init process. It receives the bootstrap authority needed to read BootPackage, validate the service graph, spawn child services, and supervise them. The foreground `capos-shell` is now an init-started service with the terminal, credential, session, audit, and broker capabilities needed for the local shell flow; it does not receive BootPackage or broad ProcessSpawner authority. Focused shell-led manifests such as `system-smoke.cue` and `system-shell.cue` still boot `capos-shell` directly as `initConfig.init` for narrow login/shell proofs until the run-target/init policy cleanup migrates them.




The invariant is that the kernel starts only `initConfig.init` after validating the kernel-owned manifest boundary, and no child service starts until mkmanifest/init validation has accepted service binary references, authority graph structure, and bootstrap capability source/interface checks.

### Design

The boot path is deliberately single-shot. The kernel receives a single packed manifest and validates only the kernel-owned boot contract before creating init. Init then performs the userspace execution step: it reads manifest chunks from BootPackage, validates a metadata-only `ManifestBootstrapPlan`, resolves kernel and service cap sources, and asks ProcessSpawner to load each child ELF into its own address space with its own user stack, TLS mapping if present, ring page, and CapSet mapping.

The default manifest (`system.cue`) now boots an init-owned local path: the kernel launches the standalone `init` binary described by `initConfig.init`, and init spawns the shell, remote-session CapSet gateway, and resident services from `initConfig.services`. The shell mints an anonymous `UserSession` on startup through `SessionManager.anonymous()`, receives an empty-allowlist `anonymous` launcher from the broker, and waits at its own interactive prompt. The user

types `login` (or `setup` on a fresh image) to upgrade in place. The smoke and shell manifests still provide focused shell-led proofs, while `system-spawn.cue` remains the focused init-owned graph retained for ProcessSpawner validation.

### Invariants

- Limine must provide exactly one boot module, and that module is the manifest.
- Kernel manifest validation must complete before init is enqueued, and init BootPackage validation must complete before any child service is spawned.
- Service ELF load failures roll back frame allocations before boot continues or fails.
- Kernel page tables are active and HHDM user access is stripped before SMEP/SMAP are enabled.
- The kernel passes `_start(ring_addr, pid, capset_addr)` in RDI, RSI, and RDX.
- CapSet metadata is read-only user memory; the ring page is writable user memory.
- QEMU-feature boots halt through `isa-debug-exit` when no runnable processes remain.

### Code Map

- `kernel/src/main.rs` - `kmain`, manifest module handling, validation, boot-only-init loading, process enqueue, halt path.
- `kernel/src/spawn.rs` - ELF-to-address-space loading, fixed user stack, TLS mapping, `Process` construction helpers.
- `kernel/src/process.rs` - process bootstrap context, ring page mapping, CapSet page mapping.
- `kernel/src/cap/mod.rs` - bootstrap capability resolution and CapSet entry construction for init.
- `capos-config/src/manifest.rs` - manifest decode and schema-version storage.
- `capos-config/src/validation.rs` - graph/source/binary validation policy.
- `tools/mkmanifest/src/lib.rs` - host-side manifest validation and binary embedding.
- `system.cue` and `system-spawn.cue` - default and spawn-focused boot graphs.
- `limine.conf` and `Makefile` - bootloader config, ISO construction, QEMU targets.

### Validation

- `make run-smoke` validates the scripted focused shell-led login path: single `capos-shell` init boot from `system-smoke.cue`, password prompt, failed-auth redaction, successful shell launch, narrow shell bundle, and clean QEMU halt.
- `make run` is the operator-facing interactive boot path with the terminal UART on stdio and console/debug output logged separately.
- `make run-spawn` validates that the kernel boot-launches only the standalone `init` with Console, BootPackage, and ProcessSpawner, and that init validates BootPackage metadata before running the focused ProcessSpawner, Timer, IPC, and memory smokes.
- `cargo test-config` covers manifest decode, roundtrip, and validation logic.
- `cargo test-mkmanifest` covers host-side manifest conversion and embedding checks.

- `make generated-code-check` verifies checked-in Cap'n Proto generated output.

**Open Work**

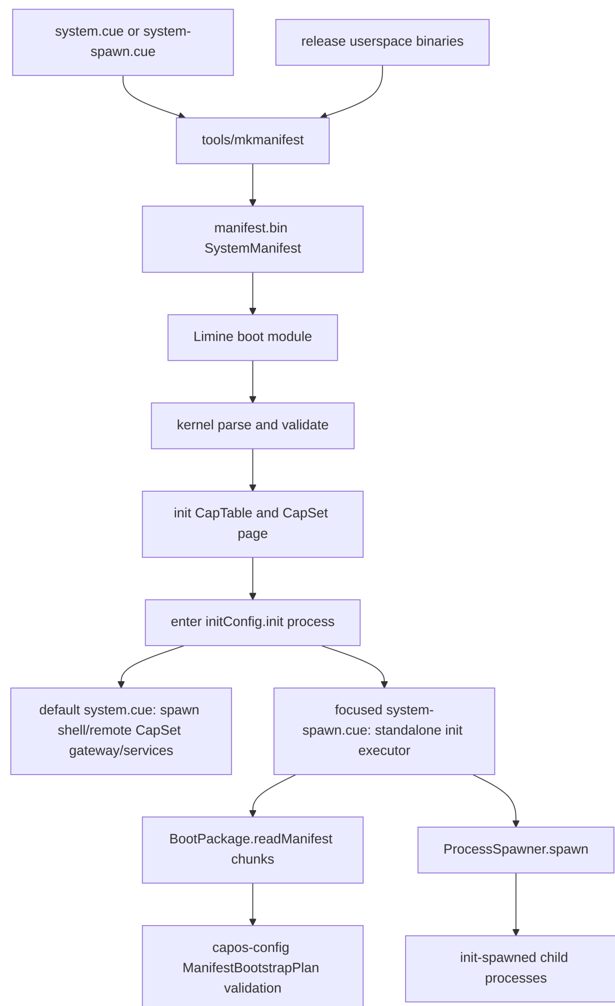
- The run-target/init-policy backlog still needs to migrate remaining focused shell-led manifests onto standalone init or explicitly preserve them as compatibility smokes.
- A future manifest-loader or mkmanifest gate should reject accidental non-`init` default boot graphs once all focused exceptions are reconciled.

## Manifest and Service Startup

The manifest is the boot package and init configuration. It names embedded binaries, the single kernel-launched init process, kernel boot parameters, and the init-owned service graph used by focused executor manifests.

### Current Behavior

`tools/mkmanifest` requires the repo-pinned CUE compiler, evaluates `system.cue`, embeds declared binaries, validates binary references and the init-owned authority graph under `initConfig`, serializes `SystemManifest`, and places `manifest.bin` into the ISO. The kernel receives that file as the single Limine module. The diagram below is intentionally large: it separates the default init-owned boot path from the focused spawn-proof path.




The default manifest starts only `initConfig.init` from the kernel, and that process is now the standalone `init` ELF. Init receives the bootstrap authority needed to read BootPackage, validate `initConfig.services`, spawn the foreground shell, remote-session CapSet gateway, resident chat service, and other default services, then wait according to the manifest policy. The shell is an init-started service; it receives terminal, credential-store, session-manager, audit-log, and authority-

broker caps, mints its own anonymous `UserSession`, and waits for an explicit `login` or `setup` command before upgrading. It never holds BootPackage or broad ProcessSpawner authority.

Focused shell-led manifests such as `system-smoke.cue` and `system-shell.cue` still put `capos-shell` directly in `initConfig.init` for narrow login/shell proofs. That compatibility path is tracked by the run-target/init-policy backlog and should not be confused with the default `system.cue` boot path.

The focused `system-spawn.cue` manifest still puts the standalone `init` ELF in `initConfig.init`. There, init receives `ProcessSpawner`, a read-only `BootPackage` cap, and Console. It reads bounded manifest chunks into a metadata-only `capos-config::ManifestBootstrapPlan`, validates binary references, authority graph structure, exports, cap sources, and interface IDs, then spawns the focused smoke services. Low-level spawn grants still model receiver selectors for hostile and compatibility proofs, but normal shell `client @...` grants omit selector syntax and preserve delegated client endpoint identity. Raw parent-capability grants must preserve the source hold metadata, endpoint-client grants may mint selectors only from an endpoint owner or a `ProcessSpawner`-returned parent endpoint facet without widening it to server authority, and kernel-source Endpoint, FrameAllocator, VirtualMemory, Timer, ThreadControl, ThreadSpawner, and EntropySource grants mint fresh child-local caps without receiver selectors. QEMU-only PersistentStore grants mount the root store through the same child-local kernel-source path when a focused proof manifest names that source. Endpoint kernel grants also return parent-side client facets as `ProcessSpawner` result caps so init can wire later service-sourced imports without ever holding child endpoint owner caps.

`mkmanifest cue-to-capnp` is the adjacent general conversion path for CUE-authored data that should not become part of `SystemManifest`. It evaluates the input with the same pinned CUE compiler, package mode, tag injection, and `CAPOS_CUE_TAGS` handling as the manifest path, then passes the exported JSON to the pinned Cap'n Proto compiler through `capnp convert json:binary`. The caller supplies the `.capnp` schema file, root struct type, output path, and optional Cap'n Proto import paths. This is schema-aware serialization for data messages rooted at arbitrary specified structs; it is not a live capability or interface-object serialization path.

### Design

Manifest validation has three layers:

- Kernel bootstrap references: binary names are unique, `initConfig.init.binary` resolves, referenced payloads are non-empty, and init kernel cap sources match their expected interface IDs.
- Init-owned binary references: `initConfig.services[*].binary` references resolve before the executor spawns children.
- Init-owned authority graph: service names, cap names, export names, and service-sourced references are unique and resolvable; re-exporting service-sourced caps is rejected.
- Init-owned cap sources: expected interface IDs match kernel sources or declared service exports.

Kernel startup now resolves only `initConfig.init.caps`. Init performs service execution in two userspace passes. The preflight pass walks `initConfig.services` in manifest order, resolves

kernel and service-sourced caps against init grants and prior exports, and rejects an unstartable graph before spawning children. The spawn pass grants caps in declaration order, records declared exports, keeps owned parent client facets for exported child endpoints, and attenuates endpoint exports to client-only facets for importers. After every child is spawned, init drops and flushes those parent facets before waiting on children; a dropped init facet therefore cannot owner-cancel queued, pending, or in-flight child endpoint state.

### Invariants

- The manifest is schema data plus an init config tree, not shell script or ambient namespace.
- Omitted cap sources fail closed.
- Cap names within one service are unique and are the names userspace sees in CapSet.
- Service exports must name caps declared by the same service.
- Service-sourced imports must reference a declared service export.
- Endpoint exports to importers must be attenuated to client-only facets.
- Init must not hold endpoint owner caps for child-local manifest endpoints.
- `expectedInterfaceId` checks compatibility; it is not the authority selector.
- Legacy receiver metadata travels with cap-table hold edges and endpoint invocation metadata. Spawn-time client endpoint minting may carry the requested child selector only from owner or trusted parent endpoint result sources instead of copying the parent's hold selector. Client facets received through ordinary spawn grants are not selector-minting authority for later spawns. Caller-selected endpoint badges are transitional compatibility state; session-bound invocation context plus broker-granted service roots/facets is the target shared-service authority model.

### Code Map

- `schema/capos.capnp` - `SystemManifest`, `NamedBlob`, `SystemConfig`, `KernelCapSource`, and generic `CueValue` storage for `initConfig`.
- `capos-config/src/manifest.rs` - manifest structs, `initConfig` CUE parsing, capnp encode/decode, metadata-only `ManifestBootstrapPlan`, and schema-version storage.
- `capos-config/src/validation.rs` - kernel bootstrap, init-owned graph, binary-reference, and capability-source validation policy.
- `tools/mkmanifest/src/lib.rs` and `tools/mkmanifest/src/main.rs` - host-side manifest build pipeline, binary embedding, and general CUE-to-Cap'n Proto data-message conversion.
- `kernel/src/main.rs` - kernel manifest module parse and validation.
- `kernel/src/cap/mod.rs` - bootstrap cap creation and CapSet entry construction for init.
- `kernel/src/cap/boot_package.rs` - read-only manifest-size and chunked manifest-read capability.
- `kernel/src/cap/process_spawner.rs` - init-callable spawn path for packaged boot binaries.
- `capos-rt/src/client.rs` - typed BootPackage and ProcessSpawner clients.
- `init/src/main.rs` - BootPackage manifest reader, graph preflight, generic spawn loop, hostile spawn checks, and child waits.

- `system.cue` and `system-spawn.cue` - default init-owned login/service graph and focused init-owned spawn manifests using `initConfig`.

**Validation**

- `cargo test-config` validates manifest decode, CUE conversion, graph checks, source checks, and binary reference checks.
- `cargo test-mkmanifest` validates host-side manifest conversion, embedded binary handling, pinned CUE path/version checks, pinned Cap'n Proto path/version checks, and schema-aware JSON-to-binary conversion through `capnp convert` when `CAPOS_CAPNP` is available.
- `make run-smoke` validates the focused shell-led scripted login manifest: single `capos-shell` init boot from `system-smoke.cue`, failed-auth redaction, successful password auth, broker-issued shell launch, terminal isolation, and clean halt.
- `make run` is the operator-facing interactive boot path with the terminal UART on stdio and console/debug output logged separately.
- `make run-spawn` validates the narrower `system-spawn.cue` graph: the kernel boot-launches only standalone `init`, init validates BootPackage metadata, ProcessSpawner launches each focused child service, grants Timer to the timer smokes, and init waits for them.
- `make generated-code-check` validates schema-generated Rust stays in sync.

**Open Work**

- The run-target/init-policy backlog still needs to migrate remaining focused shell-led manifests or preserve them as explicit exceptions, then add a manifest-loader or mkmanifest guard against accidental non-`init` default boot graphs.
- Service object identity migration still needs to retire caller-selected endpoint badge syntax from normal manifest paths. Normal shell paths already reject explicit client-grant selector syntax; low-level hostile fixtures and manifest-scoped non-identity encodings such as TCP listen ports remain separate cases.

## Process Model

The process model defines how capOS represents isolated user programs, how they receive authority, how they enter and leave the scheduler, and how a parent can observe a child.

### Current Behavior

A `Process` currently owns a user address space, a per-process capability table, a ring scratch area, a mapped capability ring, an optional read-only CapSet page, private thread/kernel-stack ledgers, and one or more `Thread` records. Process IDs are assigned by an atomic counter. The scheduler names current execution, run queues, direct IPC handoff, and blocking waiters with generation-checked `ThreadRef` values. Each thread owns its kernel stack, saved CPU context, FS base, and `cap_enter` blocking state, while address space, capability table, ring, CapSet, and resource accounting stay process-owned.

ELF images are loaded into fresh user address spaces. PT_LOAD segments are mapped with page permissions derived from ELF flags, the user stack is fixed at `USER_STACK_BASE` (`0x100_0000` as of WASI Phase W.2 sub-slice 1; see `capos-config/src/process_layout.rs` for the canonical layout) with a linker-enforced image limit below it, and PT_TLS data is mapped into a per-process TLS area below the ring page. The process starts from a synthetic `CpuContext` that returns to Ring 3 with `iretq`.

`ProcessSpawner` lets a holder spawn packaged boot binaries, grant selected caps to the child, and receive result caps. Every successful spawn returns a non-transferable `ProcessHandle`; child-local endpoint kernel grants also return parent-side client facets so a supervisor can wire imports without sharing endpoint owner authority. `ProcessHandle.wait` either completes immediately for an already-exited child or registers one waiter. Child-local `ThreadControl` grants give runtimes ownership of their current FS base and current-thread exit. Child-local `ThreadSpawner` grants let a process create additional in-process threads and receive process-local `ThreadHandle` result caps for join, detach-on-release, and exit-code observation.

### Design

Process construction separates image loading from capability-table assembly. Default boot maps only init in the kernel and gives it a bootstrap CapSet. Spawned children use the same image loading and `Process` creation helpers, but their grants are supplied by the calling process through `ProcessSpawner`. Init resolves service-sourced manifest imports against previously recorded exports before asking ProcessSpawner to create each child.

Each process starts with three machine arguments:

- `RDI` - fixed ring virtual address (`RING_VADDR`).
- `RSI` - process ID.
- `RDX` - fixed CapSet virtual address, or zero if no CapSet is mapped.

Exit releases authority before the `Process` storage is dropped. The scheduler switches to the kernel page table before address-space teardown, cancels endpoint state for the exiting pid, completes any pending process waiter, and defers the final process drop until execution is on another kernel stack.

Future process lifecycle work should keep authority transfer explicit: parents should not gain ambient access to child internals, and child grants should come from named caps plus interface checks.

The 7.1.0 in-process threading contract is documented in In-Process Threading. It defines `ThreadSpawner` and `ThreadHandle` as process-local authorities, preserves `ProcessHandle` as the parent-facing whole-process lifecycle handle, and keeps process exit as the operation that releases shared capability authority.

### Invariants

- A process cannot access a resource unless its local `CapTable` holds a cap.
- Bootstrap CapSet metadata is immutable from userspace.
- A stale `CapId` generation must not name a reused cap-table slot.
- `ProcessSpawner` raw grants require a copy-transferable cap or an endpoint owner cap; client-endpoint grants require an endpoint owner or `ProcessSpawner` endpoint result source and never add receive or return authority.
- `ProcessSpawner` kernel-source Endpoint, FrameAllocator, VirtualMemory, ThreadControl, ThreadSpawner, and EntropySource grants are fresh child-local caps and cannot be badged. QEMU-only PersistentStore grants mount a Store cap through the child-local kernel-source path for focused persistence proofs. Endpoint kernel grants are exportable only through returned parent client facets, not through a shared owner cap in init.
- `ProcessHandle` caps are non-transferable.
- `ThreadHandle` caps are process-local, non-transferable, and observe only one thread in the same process.
- At most one waiter may be registered on a `ProcessHandle`.
- Process exit releases cap-table authority before the kernel stack frame is freed.

### Code Map

- `kernel/src/process.rs` - `Process`, bootstrap CPU context, ring/CapSet mapping, exit capability cleanup.
- `kernel/src/spawn.rs` - ELF mapping, stack mapping, TLS mapping, process construction helpers.
- `kernel/src/sched.rs` - process table, process handles, wait completion, exit path.
- `docs/architecture/threading.md` - frozen 7.1.0 contract for process-owned versus thread-owned state, creation, FS-base, and join/exit behavior.
- `kernel/src/cap/process_spawner.rs` - `ProcessSpawnerCap`, `ProcessHandleCap`, spawn grant validation, child-local kernel grants, child CapSet construction.
- `capos-lib/src/cap_table.rs` - `CapId` generation and cap-table operations.
- `capos-config/src/capset.rs` - fixed CapSet page ABI.
- `schema/capos.capnp` - `ProcessSpawner`, `ProcessHandle`, and `CapGrant`.
- `init/src/main.rs` - BootPackage manifest validation, generic spawn loop, child waits, and hostile spawn checks.

### Validation

- `make run-smoke` validates init-owned default service startup, `ProcessSpawner`, `ProcessHandle.wait`, child grants, exit cleanup, and clean halt.
- `make run-spawn` validates the narrower ProcessSpawner graph for endpoint, IPC, VirtualMemory, FrameAllocator cleanup, and hostile spawn failures.
- `cargo test-lib` covers `CapTable` generation, stale-slot, and transfer primitives.
- `cargo test-config` covers CapSet and manifest metadata used to build process grants.
- `cargo build --features qemu` verifies the kernel and QEMU-only paths compile.

### Open Work

- Add lifecycle operations such as kill and post-spawn grants only after their authority semantics are explicit.
- Implement restart policy outside the kernel-side static boot graph.

## Session Context

Session-bound invocation context is the current shared-service identity model. Capabilities decide what a process may invoke. The process session supplies the privacy-preserving subject context for the invocation. Request payloads, manifest strings, and legacy endpoint receiver metadata do not identify the caller and must not authorize service behavior by themselves.

### Current Behavior

Every normal workload process has one immutable `SessionContext` installed through trusted spawn, session-manager, or broker paths. Endpoint CALL delivery includes a scoped caller-session reference plus freshness metadata by default. The server does not receive a global principal, account, profile, display name, auth source, or tenant field unless the call explicitly requests disclosure and the invoked service/facet has a matching disclosure scope.

The current endpoint ABI carries:

- `scoped_ref` and `scoped_ref_hi`: a 128-bit opaque caller-session reference derived from a boot secret, endpoint service scope, and kernel session id;
- `epoch`: a domain-separated freshness/audit value for the same service scope;
- liveness/freshness state used to fail closed for stale ordinary sessions.

The reference is service-scoped and non-portable. A value observed by one service is not authority and is not a stable global identity in another service.

### Authority Split

Capability possession answers whether a process may invoke a target. Session context answers who the invocation is attributable to, whether the session is fresh enough, which resource/accounting bucket should be used, and which subject facts may be disclosed.

The service decision is therefore layered:

1. capability authority;
2. invocation subject context;
3. service-local policy and state.

For example, holding `ChatRoot` lets a process ask chat to join. The caller's live session supplies the subject context. The chat service may key its per-session state by the opaque reference and may request bounded disclosure when its method contract and broker policy allow it.

### Disclosure

Disclosure is opt-in and field-granular. A service receives broader subject facts only when both conditions hold:

- the method or call shape explicitly requests the fields;
- the invoked capability or broker-granted facet carries a service-scoped disclosure scope allowing those fields.

Without both, endpoint metadata stays opaque. Services that need display names, profile classes, or audit labels should request only those fields and treat them as service-local policy input, not as independent authority.

### Transfer And Liveness

Cross-session capability transfer is allowed only when the transferred cap's transfer scope permits it. A transferred cap carries invoke authority; the receiver's session remains the invocation subject. `service_regrant_only` caps cannot cross sessions through raw copy, move, endpoint IPC, or spawn grants; a trusted service or broker regrant path must mint target-session authority.

Ordinary endpoint calls from logged-out or expired sessions fail closed. The current liveness implementation is a `Live`/`LoggedOut` state cell plus expiry; administrator revocation, recovery-only session modes, and renewal/recovery caps remain future lifecycle work. Fixed wall-clock expiry remains a bounded guardrail, not complete production interactive-session lifecycle UX.

### Code Map

- `kernel/src/session_context.rs` - kernel session records, liveness state, and scoped reference derivation.
- `kernel/src/cap/endpoint.rs` - endpoint caller-session delivery and stale endpoint/session checks.
- `kernel/src/cap/transfer.rs` and `capos-lib/src/cap_table.rs` - transfer-scope validation and rollback.
- `kernel/src/cap/session_manager.rs` - session creation and `UserSession` result-cap minting.
- `kernel/src/cap/user_session.rs` - `UserSession` capability behavior.
- `kernel/src/cap/restricted_launcher.rs` and `kernel/src/cap/authority_broker.rs` - broker and launch surfaces that mint session-scoped bundles.
- `capos-rt/src/client.rs` - runtime clients that observe session, endpoint, and logout behavior.
- `docs/architecture/ipc-endpoints.md` - endpoint transport and transfer rules.
- `docs/architecture/process-model.md` - spawn and process ownership model.

### Validation

- `make run-session-context` covers process-session invariants, default endpoint caller-session metadata, stale normal endpoint rejection, transfer scopes, and disclosure gating.
- `make run-capnp-chat-interop` and the chat/adventure smokes cover ordinary service state keyed by live caller-session metadata instead of caller-chosen selectors.
- `make run-remote-session-capset-interop` and focused remote-session UI smokes cover DTO gateway logout/close propagation.
- `make run-ssh-public-key-session` covers `UserSession.auditContext`, explicit logout idempotence, and post-logout fail-closed reads.

### Open Work

- Administrator revocation, renewal/recovery, live proxy cleanup, and audit reason separation remain future lifecycle work.
- Stable service-audit identity across endpoint replacement or service upgrade needs a future service-audit scope.
- Delegated act-on-behalf-of subject context is a separate future design, not part of the completed session-bound invocation context milestone.
- A dedicated result-cap move-source rollback proof is still needed before fixed expiry is treated as the whole production session lifecycle.

### Design Grounding

The archival decision record is Proposal: Session-Bound Invocation Context. The superseded direction is Proposal: Service Object Capabilities. Capability-system precedent is summarized in Research: Capability-Based and Microkernel Operating Systems.

## In-Process Threading Contract

This page records the implemented contract for kernel-managed threads inside one process. The park authority contract is frozen separately in Park Authority. These pages are the handoff from the initial single-thread runtime checkpoint to same-process SMP work. The current slice has per-thread completion rings for spawned child threads, per-CPU WFQ run queues with bounded stealing, a caller-thread-bound `SchedulingPolicyCap`, and a `SchedulingContext` cap that records identity, bind/revoke, dispatcher budget charging/replenishment, bounded endpoint donation/return, and fixed depletion/deadline notification cells. Same-process sibling scheduling has formal accepted 1-to-2 evidence on `capos-bench` 2026-05-02 21:38 UTC against `main` commit `374f8556` (capOS work `1.883x` / total `1.787x`, both clearing the configured `1.6x` gates; matching Linux pthread baseline `1.988x`/`1.987x` on the same physical-core pin set). The 2026-05-02 1-to-4 row was the diagnostic that justified Phase D's fair-share enqueue policy: capOS sat at `1.566x`/`1.538x` while Linux scaled to `3.963x`/`3.858x`. Phase D now runs per-CPU WFQ queues with bounded stealing and manually accepted the 2026-05-10 1-to-4 diagnostic row (`3.088x`/`2.700x`) while the harness-enforced gate remains 1-to-2 work/total speedup; see `docs/benchmarks.md` for the full evidence table including historical pre-collapse rows. Phase F has landed the one-SQ-consumer prerequisite, nohz telemetry, housekeeping/deferred-work placement, the clockevent/deadline substrate, and bounded SQPOLL ring mode including the non-periodic SQPOLL producer-wake progress path; the first automatic nohz activation increment is closed via `docs/tasks/done/2026/scheduler-phase-f-auto-nohz-activation.md` and SQPOLL-driven auto-nohz activation is also closed via `docs/tasks/done/2026/scheduler-phase-f-auto-nohz-sqpoll.md`; generic full-nohz for ordinary budgeted compute leases and timeout-based auto-revoke are landed; policy-service AutoNoHz issuance remains future work.

### Scope

The threading milestone changes the scheduler's unit of execution from process to thread while keeping the process as the authority, address-space, and resource-accounting boundary. Same-process sibling scheduling on multiple CPUs is functional for per-thread-ring processes. The accepted 1-to-2 performance claim is now the formal `capos-bench` 5-run pair recorded on 2026-05-02 21:38 UTC against `main` commit `374f8556`: capOS work `1.883x` and total `1.787x` clear the configured `1.6x` gates; the matching Linux pthread baseline on the same physical-core pin set (`0,1,2,3`) records `1.988x`/`1.987x`, validating the workload shape. The 2026-05-02 1-to-4 row was the diagnostic that justified Phase D: capOS sat at `1.566x`/`1.538x` while Linux scaled to `3.963x`/`3.858x`. Phase D now runs per-CPU WFQ queues with bounded stealing and its 2026-05-10 1-to-4 row (`3.088x`/`2.700x`) was manually accepted from recorded diagnostics; the harness-enforced gate remains 1-to-2 work/total speedup. Historical pre-collapse rows and the post-collapse 3-run diagnostic remain in `docs/benchmarks.md` for reference. Phase E adds the `SchedulingContext` cap (identity, caller-thread bind, revoke, budget charging/replenishment, bounded synchronous endpoint donation/return, and fixed depletion/deadline notification cells with drain observer results), and Phase F has landed the bounded SQPOLL ring mode plus the clockevent/deadline substrate. Automatic nohz activation, realtime admission, and privileged userspace scheduler-policy services remain later work.

This contract covers:

- process-owned versus thread-owned state;
- the initial thread creation ABI;
- per-thread FS-base/TLS rules;
- thread exit and join semantics;
- the per-thread ring blocking and completion-routing contract;
- the caller-thread-bound `SchedulingPolicyCap` and `SchedulingContext` surfaces that mutate per-thread WFQ weight/latency-class and per-thread scheduling-context binding;
- the handoff to the 7.1.1 park authority design.

### Ownership Split

The process remains the security boundary. All threads in one process share the same address space and capability table, so a thread has the same authority as its sibling threads.

- **Process-owned state:** Process id and process generation
  - **Thread-owned state:** Thread id and thread generation
- **Process-owned state:** User address space and CR3
  - **Thread-owned state:** Saved CPU context and user register state
- **Process-owned state:** Capability table and resource ledger
  - **Thread-owned state:** Kernel stack and syscall stack top
- **Process-owned state:** Initial compatibility ring and ring arena ownership
  - **Thread-owned state:** Per-thread ring endpoint, scratch, and FS base
- **Process-owned state:** Read-only CapSet page
  - **Thread-owned state:** Scheduling/blocking state
- **Process-owned state:** ProcessHandle exit state
  - **Thread-owned state:** ThreadHandle join/exit state
- **Process-owned state:** Endpoint owner state and process-wide cleanup hooks
  - **Thread-owned state:** WFQ weight, latency class, virtual runtime, and `virtual_finish_ns` enqueue tag
- **Process-owned state:** Process-wide resource ledgers (thread records, kernel stacks, cap-table slots)
  - **Thread-owned state:** `SchedulingContext` binding (identity/generation, remaining budget, replenish/deadline timestamps, donation/return slot, notification recorder)

The implementation migrated incrementally. The 7.2.0 slice made each process contain a single initial `Thread`, with saved context, kernel stack, FS base, and blocking state stored on that thread. Later slices changed scheduler-owned queues, current execution, direct IPC handoff, and wake records to generation-checked `ThreadRef` values, added creation and lifecycle caps, and then assigned per-thread rings to spawned children.

### Scheduler Contract

`Scheduler` stores runnable execution contexts as thread references, not process ids. A thread reference is `(pid, process_generation, tid, thread_generation)`. The process generation

keeps handles from naming a reused process; the thread generation keeps handles from naming a reused thread slot inside a live process.

This identity applies to `Scheduler.current`, run queues, direct IPC targets, Timer sleep waiters, process/terminal waiters, endpoint caller/receiver wake records, and deferred cancellation state.

Runnable ownership is split across per-CPU run queues (`SCHEDULER_CPUS = 4`). Each queue is ordered ascending by `virtual_finish_ns`, which is recomputed per enqueue from `virtual_runtime_ns`, the thread's WFQ `weight` (clamped to `[MIN_WEIGHT, MAX_WEIGHT]` in `capos-abi::scheduler`), and a per-class slice scaled by `LatencyClass` (`Interactive` divides the slice, `Batch` multiplies it, `Normal`/`IpcServer` pass it through). Default placement targets the current CPU; a bounded steal path balances when a CPU's local queue is empty, recomputes the WFQ tag at the destination, and records placement-spread / steal migrations under the `measure` feature. Each per-CPU queue is reserved at thread-create time to the live runnable-capable thread count so timer-tick, unblock, direct-IPC fallback, and steal-requeue paths never allocate.

The run queue, `current`, direct IPC target, and blocked waiter scans are thread-oriented. Address-space switches happen only when the next runnable thread belongs to a different process. TSS.RSP0, the syscall kernel stack, and FS base are updated on every thread switch because those are thread-local machine resources. Per-thread `runtime_ns` advances 1:1 with elapsed CPU time; `virtual_runtime_ns` advances by `elapsed_ns * REFERENCE_WEIGHT / weight` so weight changes the cumulative WFQ share rather than just an enqueue tie-breaker.

`SchedulingContext` bindings layer dispatcher budget on top of WFQ. A thread may carry at most one `SchedulingContextThreadBinding`. While bound, the dispatcher charges elapsed time against the binding's `remaining_budget_ns`, replenishes from `period_ns` at the next replenish boundary, records `deadline_or_timeout` and `budget_depleted` notifications in the per-context fixed cells, and routes synchronous endpoint donation/return for passive receiver threads (`donated_holder` in the notification snapshot tracks whether the holder is the donor or the receiver). Stale-generation or revoked caps fail closed before mutating scheduler state. Realtime-island admission, CPU placement enforcement, and overrun-fault policy remain deferred.

The idle path is a per-CPU CPL0 (kernel-mode) idle thread; the former special user-mode idle process has been removed. Each CPU's idle thread is a kernel-owned execution context — it runs on the kernel PML4 with a dedicated idle kernel stack and cannot block, exit, or hold ordinary caps. A lightweight synthetic idle `Process` record is retained per CPU only so the idle `ThreadRef` resolves through scheduler bookkeeping; it maps no user code, stack, or cap ring. See the "Idle paths" section of `docs/architecture/scheduling.md`.

Phase F has landed the one-SQ-consumer prerequisite, nohz telemetry, housekeeping/deferred-work placement, the clockevent/deadline substrate, and a bounded SQPOLL ring-mode worker (`MAX_SQPOLL_WORKERS = 16`, `request_sqpoll_start_for_thread` / `finalize_pending_sqpoll_start_for_thread` with stale-owner rollback). Tick suppression now exists behind explicit `CpuIsolationLease` admission, including ordinary budgeted compute leases that target a live `SchedulingContext`; policy-service AutoNoHz issuance and generic SQPOLL nohz for arbitrary rings remain future work.

### Thread Creation ABI

Thread creation is exposed through a process-local `ThreadSpawner` capability. It creates threads only in the caller's current process. It does not grant authority to another process and is non-transferable across IPC in the initial implementation.

The initial control-plane shape is:

```
interface ThreadSpawner {
    create @0 (
        entry :UInt64,
        stackTop :UInt64,
        arg :UInt64,
        fsBase :UInt64,
        flags :UInt64
    ) -> (handleIndex :UInt16);
}

interface ThreadHandle {
    join @0 () -> (exitCode :Int64);
    exitCode @1 () -> (exited :Bool, exitCode :Int64);
}

interface ThreadControl {
    getFsBase @0 () -> (fsBase :UInt64);
    setFsBase @1 (fsBase :UInt64) -> ();
    exitThread @2 (code :Int64) -> ();
}
```

Any 7.2 schema adjustment must update this page in the same branch before implementation review. The stable semantics are that creation is in-process, the returned handle is an observed result cap, `ThreadHandle` observes one thread rather than the whole process, and current-thread exit is available through both `ThreadControl.exitThread` and the raw `exit(code)` syscall.

The new thread starts in Ring 3 at `entry` with:

- `RDI = arg`;
- `RSI = tid`;
- `RDX = pid`;
- `RCX = the current thread's ring address`;
- `R8 = CAPSET_VADDR`, or zero if the process has no CapSet.

The runtime supplies the user stack and TLS block. The kernel validates that `entry`, `stackTop`, and `fsBase` are user-canonical, that `stackTop` is 16-byte aligned at entry, and that reserved `flags` bits are zero. Page presence and stack-growth policy remain process address-space questions; before a page-fault subsystem exists, an invalid thread stack can fault the process.

### Resource Accounting

Thread creation allocates kernel memory and is quota-backed by process-owned ledger state, not per-capability helper counters. The 7.2.0 checkpoint charges the initial thread during process creation; `ThreadSpawner.create` extends the same ledgers to additional threads. The ledger of record is:

- `PROCESS_THREAD_LIMIT`, the maximum live or retained thread records in one process, initially 16;
- `PROCESS_THREAD_KERNEL_STACK_PAGES`, initially matching the current per-thread kernel stack allocation size of 32 pages;
- `thread_records_used` / `thread_records_max`;
- `thread_kernel_stack_pages_used` / `thread_kernel_stack_pages_max`.

The initial process thread charges one thread record and one kernel-stack allocation during process creation. `ThreadSpawner.create` reserves a thread record and kernel-stack page budget before allocating the stack or publishing a `ThreadHandle`; every later failure rolls both reservations back before returning. Cap-slot reservation for the result handle remains charged to the existing process cap-table ledger.

Creation failures are controlled application exceptions. Thread count, kernel-stack budget, handle cap-slot exhaustion, and kernel stack allocation failure return `Overloaded` with a specific message and no partially runnable thread. Invalid entry, stack, FS base, or flags return `Failed`.

Thread exit releases the kernel stack only after the scheduler is running on a different kernel stack. The thread record remains charged while a live `ThreadHandle`, pending join waiter, or unjoined exit status can still observe it. Once the handle is released without a pending join, or once a one-shot join has consumed the status and no wait record pins it, the retained record charge is released. Process exit releases all thread records and stack charges once.

The off-stack property is enforced by an `OffStackToken` witness on every stack frame release path: the deferred per-thread drain calls `Process::release_thread_kernel_stack`, whole-process teardown calls `Process::release_all_thread_kernel_stacks`, and pre-publication rollback calls `Process::rollback_created_thread`. The token constructor is private to the scheduler module. Implicit `Thread::Drop` is deliberately not a release path; if a `Thread` value reaches its destructor with a nonzero stack, it fails closed by leaving the frames allocated instead of freeing a stack without an off-stack witness.

### FS Base And TLS

FS base is thread-owned. The existing `ThreadControl.getFsBase` and `ThreadControl.setFsBase` operations keep their names, but after threading they refer to the current thread, not the whole process. `setFsBase` continues to reject non-user-canonical values and writes the CPU FS-base MSR immediately when called by the running thread. Both methods route through context-aware dispatch (`CapCallContext::caller_thread`) so the operation always targets the caller, never a different thread; calling `ThreadControl` from a non-live caller returns `ProcessFsBaseError::CallerNotLive`.

The initial process thread uses the PT_TLS block installed by ELF loading. Additional threads receive an FS base from `ThreadSpawner.create`; the runtime is responsible for allocating and

initializing each thread's TLS/TCB data. There is no process-global FS base. Current-thread FS-base operations are useful for the single-thread runtime checkpoint, but they must not be treated as the final threading ABI for language runtimes. True multi-threaded Go or C/POSIX-like runtime support requires each `ThreadRef` to own a distinct TLS block and FS base.

Context switching must save the outgoing thread's FS base and restore the next thread's FS base even when both threads belong to the same process and no CR3 switch is needed.

### Thread Identity In Waiters And Dispatch

The concrete identity type for in-process scheduling is:

```
ThreadRef {
    pid,
    process_generation,
    tid,
    thread_generation,
}
```

Process identity still governs authority and accounting, but wakeup and blocking state must name a thread. 7.2 changes context-aware capability dispatch so `CapCallContext` carries both the caller process id for authority checks and the caller `ThreadRef` for wake/cancel decisions. Existing pid-only records that can resume execution or write a caller CQE must be widened before multiple threads can run in one process.

The migration target is:

- `TimerSleepWaiter` stores the sleeping `ThreadRef` and validates the generation before waking it;
- endpoint CALL, RECV, RETURN target, deferred-cancel, current-caller, and direct IPC handoff records store the blocked or target `ThreadRef`;
- terminal line input and any other `ProcessWaiter` consumer store the waiting `ThreadRef` and validate the generation before writing a CQE;
- `ProcessHandle.wait` records the waiting `ThreadRef` while the handle still names the child process;
- `ThreadHandle.join` records the waiting `ThreadRef` and the target `ThreadRef`;
- `cap_enter` blocks the current `ThreadRef` on that thread's ring endpoint;
- process-exit cleanup cancels every waiter whose `pid` and `process_generation` match the exiting process, regardless of thread id.

A generation mismatch on wake or completion is a stale waiter and must be drained without writing to userspace. This mirrors current process-generation behavior and prevents one thread slot reuse from receiving another thread's Timer, endpoint, join, or ring completion.

### Exit And Join

The current `exit(code)` syscall terminates the current thread. This preserves single-thread process exit because the process exits when its last non-idle thread exits, and it avoids tearing down a shared address space while sibling threads are still current on other CPUs.

Thread exit does not add a new syscall. The initial implementation added `ThreadControl.exitThread(code)` as a terminal capability-ring operation on the current thread, with the same current-thread termination semantics as the raw syscall. A successful invocation does not post a CQE back to the exiting thread, because `cap_enter` will not return to that execution context. It records the exit code, wakes or completes any valid join waiter, and removes only the current thread from scheduling. If the last non-idle thread in a process exits through `exit(code)` or `exitThread`, the process exits with that thread's code and completes the parent-facing `ProcessHandle`.

Whole-process termination remains a `ProcessHandle` operation. It releases the shared capability table, cancels process-owned endpoint state, removes timer/park/ring waiters for every thread in the process, and completes the parent-facing `ProcessHandle` after the process is no longer current on any CPU.

`ThreadHandle.join` is process-local and one-shot. If the target thread already exited and its status is retained, join returns its code immediately and marks the status joined. If it is still live, join blocks the caller's thread until the target exits. Self-join returns `Failed`. A second waiter, join after a successful join, or join after detach returns `Failed`; it must not park an ambiguous waiter. `ThreadHandle.exitCode` is nonblocking and may observe the retained status while the handle is live, but it does not consume the one-shot join right.

Releasing the last `ThreadHandle` before the target exits detaches the target: the thread continues to run, but no exit status is retained after it exits unless a join waiter already pins the state. Releasing the handle after exit but before join drops the retained status and releases the thread-record charge. A pending join waiter pins the handle state until completion or process exit, so cap release cannot create a use-after-free. The exiting thread's kernel stack must not be freed while it is still executing on that stack; final process teardown performs an explicit token-gated stack release after another kernel stack is active, before the deferred `Process` value is dropped.

Fatal user faults remain process-fatal in the first implementation. Per-thread fault isolation can be designed later, after the basic scheduler and futex paths are stable.

### Capability Ring And Blocking

The first Ring v2 implementation keeps the initial thread's compatibility ring at `RING_VADDR` and gives each spawned child thread a kernel-chosen ring mapping inside the reserved process ring arena. Runtime-selected ring address ranges remain a later `VirtualMemory` reservation extension.

`ThreadSpawner.create` allocates a ring record and user mapping for the new thread, stores that mapping on the child `ThreadRef`, and passes the ring address in the child start registers. `cap_enter` blocks the current thread against that thread's own CQ, so same-process sibling threads may block in `cap_enter` independently. Timer, endpoint, join, park, and cancellation paths must route completions by generation-checked `ThreadRef` to the target thread's ring endpoint.

The runtime's single-owner ring-client invariant remains local to each ring client. Well-formed userspace serializes submission and completion matching per thread ring through `capos-rt`; it must not have two consumers racing on the same SQ/CQ. The scheduler still refuses to run the

exact same `ThreadRef` on two CPUs at once, but it no longer treats every multithreaded pid as tied to one scheduler CPU.

This is sufficient for functional same-process sibling scheduling. The formal accepted 1-to-2 `make run-thread-scale` capOS evidence is the `capos-bench` 2026-05-02 21:38 UTC pair (work `1.883x`, total `1.787x`, both clearing the configured `1.6x` gates). The guest result row's `accepted` field remains diagnostic; the host summary enforces the work-window and total-time gates, and refuses speedup enforcement unless `CAPOS_THREAD_SCALE_QEMU_TASKSET_CPUS` records the QEMU CPU pin set. Linux validates the repaired benchmark shape through four workers on physical cores (`3.963x`/`3.858x`). That capOS 4-worker row was diagnostic (`1.566x`/`1.538x`) and justified Phase D's per-CPU WFQ queues plus bounded stealing. The 2026-05-10 Phase D rerun recorded 1-to-4 work/total diagnostics `3.088x`/`2.700x`, manually accepted for closeout; remaining risks are the shared scheduler lock, temporary CPU pinning, CQ/join/exit/block/schedule overhead, broader workload classes, and higher-thread-count evidence.

### Scheduling Policy And Context Authority

`SchedulingPolicyCap` is the caller-thread-bound surface for WFQ knobs. Every method routes through `CapCallContext::caller_thread`; there is no per-cap-object `ThreadHandle`, no badge-encoded thread id, and no cross-thread mutation in this slice. Cross-thread authority is deferred to the privileged scheduler-policy service plan. The schema shape is:

```
interface SchedulingPolicyCap {
    setWeight @0 (weight :UInt16) -> ();
    setLatencyClass @1 (class :LatencyClass) -> ();
    snapshot @2 () -> (
        weight :UInt16,
        class :LatencyClass,
        runtimeNs :UInt64,
        virtualRuntimeNs :UInt64,
    );
}
```

`setWeight` validates against `[MIN_WEIGHT, MAX_WEIGHT]` at the cap boundary and updates the caller thread's WFQ weight; the new weight applies to the next enqueue's `virtual_finish_ns` tag and to subsequent `virtual_runtime_ns` accounting. `setLatencyClass` swaps the per-thread `LatencyClass` (`Normal`, `Interactive`, `IpcServer`, `Batch`) used to scale the dispatcher slice. `snapshot` is a read-only observer over the core WFQ state and does not expose the `measure`-only counters.

`SchedulingContext` is the schema-typed cap for dispatcher budget authority:

```
interface SchedulingContext {
    info @0 () -> (info :SchedulingContextInfo);
    create @1 (spec :SchedulingContextSpec) -> (
        contextIndex :UInt16,
        identity :SchedulingContextIdentity,
        result :SchedulingContextOperationResult,
```

```
        dispatchEffect :SchedulingContextDispatchEffect,
    );
    bindCallerThread @2 () -> (
        identity :SchedulingContextIdentity,
        binding :SchedulingContextBinding,
        result :SchedulingContextOperationResult,
        dispatchEffect :SchedulingContextDispatchEffect,
    );
    revoke @3 () -> (
        identity :SchedulingContextIdentity,
        previousGeneration :UInt64,
        result :SchedulingContextOperationResult,
        dispatchEffect :SchedulingContextDispatchEffect,
    );
    drainNotifications @4 () -> (
        notifications :SchedulingContextNotificationSnapshot,
    );
}
```

`create` returns a same-interface child context as transferred result cap 0 and becomes chargeable only after `bindCallerThread`. `revoke` bumps the generation and clears any matching thread binding; later calls through the stale cap generation report `staleGeneration` or fail closed before mutating scheduler state. `drainNotifications` reads the fixed per-context budget-depleted and deadline-or-timeout slots; the scheduler updates these in place from hard paths without allocation, including the holder identity and a `donatedHolder` bit for endpoint donation/return. The bootstrap manifest grants `SchedulingPolicyCap` and `SchedulingContext` only to focused-proof manifests; the default boot manifest does not grant them.

### Userspace API Surface

The `capos-rt` runtime exposes the threading caps as typed clients on top of the per-thread ring:

- `ThreadControlClient` – `get_fs_base`/`set_fs_base`/`exit_thread`, including `*_wait` blocking variants over `RuntimeRingClient`.
- `ThreadSpawnerClient::create` – submits the `entry`/`stackTop`/`arg`/ `fsBase`/`flags` ABI and returns an `OwnedCapability<ThreadHandle>` delivered as transferred result cap 0 in the CQE.
- `ThreadHandleClient` – `join`, `exit_code` (nonblocking observer), and their `finish_*` helpers; `finish_join` decodes the one-shot exit code.
- `SchedulingPolicyClient` – `set_weight`, `set_latency_class`, and `snapshot`, all caller-thread-bound.
- `SchedulingContextClient` – `info`, `create`, `bind_caller_thread`, `revoke`, and `drain_notifications`.

A typical spawn/join pseudocode against these clients is:

```
let handle = thread_spawner.create_wait(
    &mut ring,
```

```
    entry_addr,
    user_stack_top,
    arg,
    fs_base,
    /* flags */ 0,
    timeout_ns,
)?;
// ... runtime work on the parent thread ...
let exit_code = thread_handle
    .join_wait(&mut ring, timeout_ns)?;
```

The userspace runtime is responsible for the user stack, TLS/TCB, and any free-list bookkeeping for retired handles; the kernel only validates the ABI fields and charges the per-process ledgers.

### Park Handoff

Park authority is defined in Park Authority. The scheduler changes above must leave room for a thread block reason that is not tied to the process ring CQ. The frozen handoff is:

- park wait blocks the current thread, not the whole process;
- park wake makes selected generation-checked `ThreadRef` values runnable;
- timeouts use the same monotonic time base as `Timer`;
- private park keys are based on address-space identity plus user virtual address;
- shared-memory park keys are MemoryObject-derived identity plus offset;
- the first implementation starts with compact `CAP_OP_PARK` and `CAP_OP_UNPARK` operations rather than generic Cap'n Proto methods;
- park wait SQEs are thread-owned so ring dispatch cannot park a sibling thread under the waiter's `user_data`;
- blocking park wait is a syscall-context operation that releases runtime ring-client ownership before the thread parks, while `capos-rt` demultiplexes reserved park CQEs back to the waiting thread.

Pre-thread 4.5.4 measurement chose the compact capability-authorized shape for failed wait and empty wake. 4.5.5 measured the real blocked/resume path through `thread-lifecycle` under `make run-measure`, so the compact ParkSpace opcodes remain the runtime ABI target for this slice.

### Security Invariants

- A thread never owns a separate capability table in the initial model.
- A thread cannot escape the authority of its containing process.
- A `ThreadHandle` names only a thread in the same process and is non-transferable in the first implementation.
- Thread creation is charged to one process-owned thread/kernel-stack ledger of record before the thread can become runnable.
- Process exit releases shared authority once, after all live threads are removed from scheduling.
- Per-process resource quotas are shared by all threads.

- `ThreadControl` changes only the current thread's FS base.
- `ThreadControl.exitThread` terminates only the current thread and is a capability-ring operation, not a syscall.
- Every waiter or direct handoff that can resume execution stores a generation checked `ThreadRef`.
- Process-owned user-buffer validation/copy/read paths hold the process `AddressSpace` lock; future shared-memory thread primitives still need mapping provenance or object pins when they derive keys from shared backing.

### Implementation Order

1. Add internal `Thread` state, make each process own one initial thread, move saved context / kernel stack / FS base / block state onto that thread, and charge the initial thread against private process ledgers. Done 2026-04-24 23:09 UTC.
2. Change scheduler queues, blocking, exit cleanup, and direct IPC targets from pid-oriented state to thread references while preserving one thread per process. Done 2026-04-24 23:33 UTC.
3. Add `ThreadSpawner`, `ThreadHandle`, and `ThreadControl.exitThread` with a QEMU smoke for create, join, detach, self-join rejection, second join rejection, and last-thread process exit. Done 2026-04-25.
4. [x] Implement the ParkSpace private wait/wake path from Park Authority after the scheduler can block and wake individual threads, then run 4.5.5 blocked/resume measurements before declaring the park ABI stable. Done 2026-04-25.

### Validation

The `thread-lifecycle` proof creates multiple threads in one process, proves they share the address space and CapSet, proves each has an independent FS base, rejects invalid join cases, joins one thread from another, and lets the last thread exit the process. The existing `make run-spawn` path keeps covering `runtime-fs-base` and `single-thread-runtime` so regressions in the pre-thread runtime contract stay visible. `make run-measure` additionally records the private ParkSpace blocked/resume timings and proves process exit with a parked park waiter. Phase D fairness/Interactive/weight-change smokes (`make run-thread-fairness`, `make run-thread-fairness-interactive`, `make run-thread-fairness-weight-change`) exercise the `SchedulingPolicyCap` caller-thread-bound surface; the `thread-scale` proof carries the recorded WFQ scaling evidence. The recorded 1-to-2 work/total speedup gate is the host-enforced Phase D acceptance criterion; the 1-to-4 row remains a manually accepted diagnostic. Safe runtime park wrappers and a focused `SchedulingContext` budget/donation/notification smoke remain future capos-rt and harness work.

## Park Authority Contract

This page freezes the 7.1.1 design contract for thread-park (`park`/`unpark`) authority. It is the handoff from the in-process threading contract to the 7.2 implementation work and records the first 7.2.3 implementation status.

**Linux prior art.** `Park` solves the same problem as Linux `futex(2)`: userspace owns the uncontended fast path through atomic operations on a 32-bit word, and the kernel parks/wakes threads only on contention. capOS uses the distinct name `Park` because the contract differs in important ways from Linux's: it is capability-gated (no ambient authority), there is no priority inheritance, no requeue, no robust lists, and the shared variant is keyed by `MemoryObject` identity rather than `(inode, pgoff)`. References to "Linux futex" in this page point to that prior art, not to the capOS API surface.

### Scope

The first park milestone stays single-CPU and in-process. It gives a multi-threaded runtime one kernel primitive: park the current thread when a userspace word still has an expected value, and wake parked threads associated with that word. Userspace owns the uncontended path through ordinary atomic operations; the kernel owns only the contended sleep/wake path and timeout integration.

This contract covers:

- production park authority objects;
- private and shared park key identity;
- the provisional compact wait/wake transport ABI;
- scheduler, timeout, and process-exit interactions;
- resource-accounting and security invariants;
- the 4.5.5 measurement loop after real thread blocking exists.

This is not a Linux `futex(2)` compatibility surface. Priority inheritance, requeue, robust lists, shared-memory park-words before MemoryObject mapping identity is exposed, and SMP-safe user-buffer pinning remain later work.

### Implementation Status

The 2026-04-25 7.2.3 slice implements:

- schema marker interfaces for `ParkSpace` and `SharedParkSpace`;
- compact `CAP_OP_PARK` and `CAP_OP_UNPARK` opcodes;
- process-local, non-transferable ParkSpace grants through boot/spawn manifests;
- private wait/wake keyed by the caller process address space and user virtual address;
- per-thread `Park` block state with finite timeout integration;
- one reserved CQE credit per parked waiter so wake/timeout delivery cannot be crowded out by ordinary completions;
- QEMU correctness coverage in `thread-lifecycle` for mismatch, immediate timeout, wake-one, wake-many, anonymous VirtualMemory multi-waiter unmap range cleanup with stale wake-

after-reuse checks, anonymous VirtualMemory.decommit reuse stale waiter cleanup, and MemoryObject.unmap borrowed-mapping reuse stale waiter cleanup;

- 4.5.5 QEMU timing coverage in `run-measure`.

`SharedParkSpace` is a marker only. `capos-rt` has the marker type but no safe park client wrapper yet; the current correctness and measurement demos use raw compact SQEs so the ABI can settle before runtime synchronization wrappers claim the `user_data` namespace.

### Design Grounding

The reviewed project documents for this contract are:

- `docs/tasks/README.md`;
- `docs/roadmap.md`;
- `REVIEW.md`;
- `docs/architecture/threading.md`;
- `docs/architecture/scheduling.md`;
- `docs/architecture/userspace-runtime.md`;
- `docs/proposals/go-runtime-proposal.md`.

The relevant research grounding is:

- `docs/research/out-of-kernel-scheduling.md` for the kernel-assisted wait/wake split used by language runtimes;
- `docs/research/llvm-target.md` for the Go/runtime syscall surface that needs thread creation, per-thread TLS, and futexes;
- `docs/research/genode.md` for typed capability precedent and resource-accounted session state.

### Authority Objects

`ParkBench` remains measurement-only. It is not a production authority and must not be granted by normal boot manifests.

The first production model has two authority objects:

```
interface ParkSpace {}
interface SharedParkSpace {}
```

These schema interfaces are marker interfaces for typed CapSet/result-cap identity. The wait and wake operations use compact ring opcodes rather than Cap'n Proto methods, because the pre-thread 4.5.4 measurement showed the generic Cap'n Proto path is not the right default for the park hot path.

`ParkSpace` is minted for a process by the same bootstrap/spawn path that grants `ThreadControl` and `ThreadSpawner`. It is process-local and non-transferable in the initial implementation. Holding it authorizes private park wait/wake only in the caller's own address space; it does not grant memory access, cross-process wake authority, or the right to name arbitrary kernel wait queues.

`SharedParkSpace` is the shared-park object for a later MemoryObject-derived slice. A MemoryObject holder can derive a SharedParkSpace scoped to that MemoryObject's backing identity. Shared park operations through that SharedParkSpace are keyed by object offset, not by one process's virtual address. The first 7.2 implementation may leave `SharedParkSpace` unimplemented, but it must not choose a private-key ABI that prevents this shared-key model.

**Park Keys**

Private park keys are address-space scoped:

```
ParkKey::Private {
    address_space_id,
    address_space_generation,
    uaddr,
}
```

The first implementation can derive `address_space_id` and generation from the process id/generation while each process owns exactly one address space. The contract names address-space identity deliberately so a later fork/shared-AS model does not inherit a pid-shaped key.

Private parks are synchronization inside one address space. `wake` for a private key may wake only waiters in the same address space generation; a raw virtual address alone is never cross-process synchronization authority.

Shared park keys are MemoryObject scoped:

```
ParkKey::Shared {
    memory_object_id,
    memory_object_generation,
    offset,
}
```

Shared keys are disabled until the kernel can prove, while handling a park operation, that the submitted user address maps the MemoryObject backing the SharedParkSpace and can compute the byte offset in that backing object. Virtual aliases of the same shared page must converge on the same shared key. Private aliases within one address space do not converge unless they use the same user virtual address.

Shared parks require explicit shared-memory authority through the MemoryObject-derived `SharedParkSpace`. Never use raw virtual address alone for cross-process park/futex keys.

All park words are 32-bit and must be 4-byte aligned. `wait` validates the word as a readable user mapping before reading it. `wake` validates that the address is user-canonical and aligned; shared `wake` additionally validates the MemoryObject mapping identity so a caller cannot wake an unrelated object by guessing an offset.

Private-key cleanup is part of the ParkSpace contract, not an implementation detail of the Go runtime. Unmap, revoke, address-space generation change, and address-space teardown must drain or fail waiters for the old private key before the same virtual address can be reused as

unrelated state. A stale private waiter may complete only against the address-space generation it was registered under; it must not observe or wake a later mapping with the same numeric `uaddr`.

Current implementation status: process/thread-exit cleanup exists. Anonymous `VirtualMemory.unmap`, `VirtualMemory.decommit`, and `MemoryObject.unmap` for borrowed mappings drain private waiters whose `uaddr` lies in the affected range by posting `PARK_INTERRUPTED` through the waiter's reserved completion credit before making the blocked thread runnable. Cleanup removes the waiter from the address-keyed private wait table before attempting the completion. If that completion cannot be posted immediately, the thread remains blocked in a pending park-completion state with the exact completion status and reserved completion credit still charged, and scheduler wake processing retries the stored status; the waiter is not restored to the `uaddr` table while the virtual address can be reused. Shared park-word cleanup and explicit address-space generation teardown remain open. Until those land, the implemented private path is suitable for process-lifetime park words, anonymous VirtualMemory regions that use these unmap/decommit paths, and borrowed MemoryObject mappings that are explicitly unmapped with `MemoryObject.unmap`.

The ordinary QEMU proof covers wake-one, wake-many, handoff wake retry, multi-waiter private range cleanup, and stale wake-after-reuse. It does not deterministically force the transient unmap interruption ring-scratch contention race that can make the first interruption completion post fail: from userspace, waiter submission is observable before the kernel registers the waiter or after ring dispatch has released the scratch buffer. The production cleanup path therefore treats that race as a retry state outside the address-keyed waiter table rather than restoring the waiter.

#### Provisional Ring ABI

The 7.2 implementation starts with compact capability-authorized operations:

- `CAP_OP_PARK`;
- `CAP_OP_UNPARK`.

The numeric opcode values are assigned when the implementation edits `capos-config/src/ring.rs`. `CAP_OP_PARK_BENCH` remains reserved for measurement-only kernels and must not be repurposed.

`CAP_OP_PARK` uses the existing 64-byte SQE fields as:

- **SQE field:** `cap_id`
  - **Meaning:** `ParkSpace` for private wait, or `SharedParkSpace` for shared wait
- **SQE field:** `user_data`
  - **Meaning:** returned in the wait completion CQE
- **SQE field:** `addr`
  - **Meaning:** user virtual address of the 32-bit park word
- **SQE field:** `len`
  - **Meaning:** expected 32-bit value
- **SQE field:** `pipeline_dep`
  - **Meaning:** relative timeout in monotonic nanoseconds; `u64::MAX` means no timeout

- **SQE field:** `flags`
  - **Meaning:** must be `CAP_SQE_THREAD_OWNED`
- **SQE field:** `call_id`
  - **Meaning:** owning thread id; a different thread leaves the SQE at the ring head

`CAP_OP_UNPARK` uses:

- **SQE field:** `cap_id`
  - **Meaning:** `ParkSpace` for private wake, or `SharedParkSpace` for shared wake
- **SQE field:** `user_data`
  - **Meaning:** returned in the wake caller's completion CQE
- **SQE field:** `addr`
  - **Meaning:** user virtual address of the 32-bit park word
- **SQE field:** `len`
  - **Meaning:** maximum number of waiters to wake; zero is malformed

Both operations require `method_id`, `result_addr`, `result_len`, `pipeline_field`, `xfer_cap_count`, and `_reserved0` to be zero. `CAP_OP_UNPARK` also requires `flags == 0`, `pipeline_dep == 0`, and `call_id == 0`. Park operations are not promise-pipelineable in this slice. `pipeline_dep` is used as the wait timeout storage only for `CAP_OP_PARK`; future promise pipelining must keep rejecting `CAP_SQE_PIPELINE` on park opcodes or replace the park ABI in a reviewed branch.

Wait completions use non-negative `CQE.result` statuses:

- **Result:** `PARK_WOKEN = 0`
  - **Meaning:** a wake operation made the thread runnable
- **Result:** `PARK_VALUE_MISMATCH = 1`
  - **Meaning:** the loaded word did not equal `expected`
- **Result:** `PARK_TIMED_OUT = 2`
  - **Meaning:** the timeout expired before a wake
- **Result:** `PARK_INTERRUPTED = 3`
  - **Meaning:** a future cancellation/interrupt path aborted the wait

Wake completions return the non-negative number of threads woken. Malformed SQEs, invalid caps, unreadable wait words, unsupported cap object types, and stale authority use the existing negative transport errors until a later ABI adds a more specific compact-error namespace.

### Ring Ownership And Dispatch Context

Park operations use the process capability ring for submission and CQE delivery, but blocking wait is not an ordinary long-lived runtime call. A runtime must not hold `RuntimeRingClient` while the thread is parked in `CAP_OP_PARK`; otherwise no sibling thread in the same process can borrow the same ring client to submit `CAP_OP_UNPARK`.

The runtime contract for park operations is:

- `capos-rt` owns a process-wide park submission/completion path separate from the generic request-buffer `RuntimeRingClient` pending-call list;
- park wait reserves a unique `user_data` value, writes the SQE while holding the runtime's ring-submission lock, records a park-wait completion slot in runtime-owned memory, and releases the ring-submission lock before entering `cap_enter`;
- park wait sets `CAP_SQE_THREAD_OWNED` and `call_id` to the current thread id so a sibling thread cannot drain the wait and park the wrong `ThreadRef`;
- the park `user_data` namespace is reserved by the runtime so ordinary generic clients cannot accidentally claim a park completion;
- all runtime CQ draining must route reserved park `user_data` completions to the park-wait slot instead of treating them as generic client completions;
- if another thread drains the waiter CQE before the waiting thread returns from `cap_enter`, the waiting thread reads the already-recorded status from that park-wait slot;
- park wake may use the ordinary serialized ring submission path because it completes without parking the caller's thread.

`CAP_OP_PARK` is syscall-context only. Timer ring polling and any future interrupt-context ring drain must leave it unconsumed because consuming it can block the current thread and mutate scheduler state. `CAP_OP_UNPARK` also starts as syscall-context only; widening wake to timer polling would need a separate review of scheduler locking and completion delivery.

This design preserves one process ring and the single blocked `cap_enter` waiter rule. A thread blocked in `Park` is not the process ring's `CapEnter` waiter, so a sibling can still enter the kernel to submit wake, Timer, IPC, or ordinary capability work through the same process ring.

### Wait And Wake Semantics

`wait` is atomic with respect to `wake` for the same key:

1. validate the SQE shape, including thread ownership, and authority cap;
2. verify `call_id` names the current thread so a sibling cannot park on behalf of the waiter;
3. validate the user address shape and derive the private or shared park key;
4. lock the current process `AddressSpace` across validation and the user-word read for private keys; future shared keys must additionally prove mapping identity or pin the backing object;
5. take the park bucket lock;
6. read the 32-bit user word while the bucket lock is held;
7. compare the loaded value with `expected`;
8. if the value differs, post `PARK_VALUE_MISMATCH` without blocking;
9. if the value matches and the timeout is zero, post `PARK_TIMED_OUT` without blocking;
10. otherwise, record the current `ThreadRef`, key, timeout deadline, and `user_data`, then block only the current thread.

The user-word read, comparison, and enqueue are serialized with `wake` by the park scheduler path, and the read itself occurs while the process `AddressSpace` mutex is held. This prevents a page-table validation/use race and the classic lost wake where a waiter reads the old value, a

sibling stores the new value and wakes no one, and the waiter then parks based on the stale read. Shared park-words still need mapping provenance or object pinning so a MemoryObject-derived key cannot be swapped out from under key derivation. The user word is not a kernel-owned mutex. Runtime code must use normal atomic load/store and memory-ordering rules around the park word.

`wake` derives the same key, removes up to `maxWake` valid waiters from that key's FIFO list, posts `PARK_WOKEN` completions to the waiting process ring using the completion credits reserved when those waiters parked, and marks those `ThreadRef` values runnable after generation checks. A wake SQE is consumed only when the kernel can also post the wake caller's own CQE; if that ordinary CQ slot is not available, no waiters are removed and the SQE remains pending like other uncompletable ring work. Stale waiters caused by thread or process generation mismatch are drained without writing to userspace, release their reserved completion credits, and do not count as successfully woken. If a valid waiter is still in a current or handoff CPU slot when the wake path removes it from the address-keyed table, the wake still counts that waiter as woken and records a pending `PARK_WOKEN` completion for scheduler retry.

Timeouts use the same monotonic time base as `Timer`. The kernel may convert nanoseconds to scheduler ticks internally, but the ABI remains nanoseconds. Finite deadlines post `PARK_TIMED_OUT` through the waiting process ring using the waiter's reserved completion credit and wake the blocked thread if the thread generation still matches.

An explicit wake, timeout, cancellation, process exit, and unmap/revoke cleanup race must produce exactly one waiter completion or cleanup-consumption path. Once any path consumes the waiter record, the other racing paths must observe it as gone and must not post a second CQE or wake a later `ThreadRef`.

Process exit removes every park waiter whose pid/process generation matches the exiting process. Thread exit removes that thread's own park waiter before the thread record can be retained for join observation. These cleanup paths must not allocate.

Unmap, mapping revoke, and address-space teardown remove or fail private waiters for the affected key/generation before the old virtual address range is made reusable for unrelated mappings. A wake or timeout racing with cleanup must either complete the old waiter under its original generation or observe that cleanup already consumed it; it must not post a completion to a new owner of the same numeric address.

### Resource Accounting

Park waits are bounded by the process thread ledger. A thread can be in only one scheduler block reason, so live park waiters cannot exceed live threads. The first private ParkSpace implementation stores the wait node in thread-owned block state and links it into a fixed process-owned waiter table. That is valid only because private ParkSpace caps are process-local and the first key is the process address space plus user virtual address. Shared SharedParkSpace support must move to object-owned fixed buckets scoped to MemoryObject identity. Wait, wake, timeout, and process-exit cleanup must not allocate. Registering a blocking wait reserves one deferred CQE credit in the waiting process. Ordinary completion posting treats reserved credits as unavailable, so wake and timeout paths can always post the waiter completion without losing the waiter. If the kernel cannot reserve that credit, it must not enqueue or block the wait; it either

leaves the SQE pending until capacity exists or posts a negative completion for the wait attempt without consuming a waiter slot.

`ParkSpace` creation is charged as ordinary process capability/table state. If the first implementation needs per-process bucket storage beyond the cap object itself, that storage must be reserved before the ParkSpace is published and released when the process exits or the cap is finally dropped.

In the first private implementation, the waiter table is process-owned and survives release of the ParkSpace handle. `CAP_OP_RELEASE` of the last capability handle removes submit authority but cannot free a parked waiter's storage. A waiter can still receive a `PARK_WOKEN` CQE from a wake operation that already resolved the authority object, a `PARK_TIMED_OUT` CQE from a finite deadline, or a future `PARK_INTERRUPTED` CQE from an explicit cancellation path. Thread or process exit drains the wait node without posting a CQE to the exiting thread/process and releases the reserved completion credit. If a runtime drops the last ParkSpace while it has indefinite waiters, it can deadlock its own process, but it cannot create a use-after-free or leak authority outside that process. Future shared SharedParkSpace storage must use explicit non-cap-table waiter pins so object-owned buckets are not freed while parked waiters remain.

`SharedParkSpace` storage is charged to the MemoryObject-derived object when shared parking lands. It must not create a second unbounded resource path where a holder can allocate wait queues by touching many offsets.

**Security Invariants**

- Holding a ParkSpace or SharedParkSpace authorizes blocking/waking, not memory access. Wait still requires a readable user word.
- Private ParkSpace caps are process-local and non-transferable in the first implementation.
- Shared park authority must be derived from MemoryObject identity and offset, not from another process's virtual address.
- Park wait blocks the current thread, not the whole process.
- Park wait SQEs are thread-owned; a non-owner `cap_enter` leaves the SQE at the ring head instead of parking the wrong thread.
- Park wake can only make generation-checked ThreadRef values runnable.
- Park completions are posted to the waiting process ring using the waiter SQE's `user_data`.
- Blocking wait registration reserves one CQE credit for the eventual waiter completion, and wake must not remove a waiter unless that credit exists.
- `CAP_OP_PARK` is dispatched only from syscall-context `cap_enter` and never from timer or interrupt-context ring polling.
- A parked private ParkSpace waiter is stored in process-owned fixed storage; future shared SharedParkSpace waiters must pin the authority object backing their bucket table until wake, timeout, thread exit, or process exit removes the waiter.
- One process ring still has at most one blocked `cap_enter` waiter in 7.2; park wait does not create an extra blocked ring waiter.

- Private ParkSpace wait reads hold the process `AddressSpace` lock across validation and the user-word read. SharedParkSpace park-words remain blocked until MemoryObject mapping provenance or explicit object pins cover shared key derivation.

### Measurement Handoff

4.5.4 measured failed wait and empty wake before real threads existed. That result chooses a compact capability-authorized operation as the starting ABI for 7.2 rather than a generic Cap'n Proto `wait`/`wake` method pair.

4.5.5 is closed for the first real thread-blocking path. It measures:

- value-mismatch wait;
- empty wake;
- wait-to-block;
- wake-to-runnable;
- wake-to-resume through `cap_enter`.

The 2026-04-25 QEMU sample printed:

```
[thread-lifecycle] park path avg cycles: failed_wait=6778
empty_wake=6840 wait_to_block=55994326 wake_to_runnable=28219
wake_to_resume=28000684
```

The compact shape still holds for this slice: `CAP_OP_PARK` and `CAP_OP_UNPARK` remain the production runtime ABI target, while `ParkBench` remains measurement-only.

### Implementation Order

1. [x] Add `ParkSpace` and `SharedParkSpace` marker interfaces plus compact opcode constants.
2. [x] Add a process-local ParkSpace grant path next to `ThreadControl` and `ThreadSpawner`; keep it non-transferable.
3. [x] Add thread-owned `Park` block state and fixed private waiter storage with no wait/wake allocation.
4. [x] Dispatch `CAP_OP_PARK` and `CAP_OP_UNPARK` against ParkSpace for private address-space keys.
5. [x] Add QEMU smoke coverage for mismatch, timeout, wake-one, wake-many, and handoff wake retry. Safe runtime park wrappers remain a later capos-rt slice.
6. [x] Run 4.5.5 blocked/resume measurements and fold the result into the final ABI decision.
7. [ ] Drain or fail private waiters before the affected virtual address range can be reused. Anonymous `VirtualMemory.unmap` and `VirtualMemory.decommit`, plus `MemoryObject.unmap` for borrowed mappings, are covered; shared park-word cleanup and address-space generation teardown remain open.
8. [ ] Add MemoryObject-derived SharedParkSpace support only after mapping provenance or object pins cover shared key derivation under the same validation/use discipline.

### Validation

The `thread-lifecycle` proof creates multiple threads in one process, parks threads on a userspace park word, wakes them through the same ParkSpace, proves timeout and value-mismatch paths, and shows that process exit drains pending waits. `make run-measure` records failed-wait, empty-wake, wait-to-block, wake-to-runnable, and wake-to-resume timings for the implemented private path. Safe capos-rt park wrappers remain future runtime work.

## Capability Model

How capabilities work in capOS.

### What is a Capability

A capability in capOS is a reference to a kernel object that carries:

- An **interface** (what methods can be called), defined by a Cap'n Proto schema
- A **permission** (the object it references, enforced by the kernel)
- A **wire format** (Cap'n Proto serialized messages for all invocations)

A process can only access a resource if it holds a capability to it. There is no ambient authority – no global namespace, no "open by path" syscall, no implicit resource access.

### Identity Terms and Authority

capOS documentation uses identity terms as policy metadata, not as kernel authorization primitives. A **user** is human-facing prose. A **principal** is the stable identity metadata used by authentication, policy, audit, and ownership records. An **account** is planned durable local record state for a principal, including credential references, roles, attributes, storage-root references, and default profile names. A **session** is the live context that receives a concrete CapSet. **Policy profiles** and **resource profiles** select bundle fragments, approval eligibility, and quotas that a trusted broker may use when minting capabilities.

None of those terms is kernel authority: the kernel dispatches through generation-tagged `CapId` entries, not users, roles, accounts, groups, UIDs, or profile names. Account-store behavior, durable profile records, and broader quota policy remain future work tracked in the local users backlog.

### Session-Bound Invocation Context

Services should not infer authority from caller-supplied identity fields. A request parameter such as `user`, `principal`, `client`, or `role` is data. The active model is one immutable session context per process plus explicit capabilities granted by a broker or supervisor.

The general pattern is:

- authentication or admission creates a live `SessionContext`;
- process spawn installs exactly one immutable session context in the child;
- `AuthorityBroker` grants service roots/facets appropriate to that session;
- endpoint calls carry privacy-preserving caller-session metadata by default;
- subject details such as global principal id, display name, profile class, or external claims are disclosed only through explicit client disclosure and a matching broker/service disclosure scope. The current endpoint CALL path implements this as a disclosure request mask intersected with cap-held disclosure scope.

The kernel role is narrower. It verifies that a process holds a live cap-table entry, that the process session is live, and that transfer/spawn obey session scope. It may deliver an opaque service-scoped caller-session reference and freshness result to endpoint servers, but it must not disclose

broader subject details by default. It does not decide that a process is Alice, an operator, a moderator, or an NPC. Those are policy facts maintained by session, broker, account, and application services.

Opaque receiver selectors may still exist in the IPC implementation and in historical service-object routing tests. A receiver selector is not identity metadata, not shell syntax, not a user field, not a disclosure channel, and not a role bit. New shared-service identity should use the caller session context and broker-granted service facets, not caller-selected numeric labels. The chat demo now follows this rule for membership: the server receives the endpoint caller metadata and keys member records by an opaque live caller-session reference, while chat handles remain request data and visible member labels are assigned by the service. The shared chat/adventure endpoint helper now exposes caller-session metadata through `EndpointCaller` instead of a badge field; the old badge-named user-data type remains only as a source-compatible alias. Terminal output and shell-serviced stdio bridges are also gated by live caller-session metadata.

### Schema as Contract

Capability interfaces are defined in `.capnp` schema files under `schema/`. The schema is the canonical interface definition. Currently defined:

```
interface Console {
    write @0 (data :Data) -> ();
    writeLine @1 (text :Text) -> ();
}

interface TerminalSession {
    write @0 (data :Data) -> ();
    writeLine @1 (text :Text) -> ();
    readLine @2 (request :LineRequest) -> (status :LineStatus,
line :Data);
}

interface FrameAllocator {
    allocFrame @0 () -> (handleIndex :UInt16);
    allocContiguous @1 (count :UInt32) -> (handleIndex :UInt16);
}

interface MemoryObject {
    info @0 () -> (pageCount :UInt32, sizeBytes :UInt64);
    map @1 (hint :UInt64, offset :UInt64, size :UInt64, prot :UInt32) ->
(addr :UInt64);
    unmap @2 (addr :UInt64, size :UInt64) -> ();
    protect @3 (addr :UInt64, size :UInt64, prot :UInt32) -> ();
}

interface VirtualMemory {
    map @0 (hint :UInt64, size :UInt64, prot :UInt32) -> (addr :UInt64);
    unmap @1 (addr :UInt64, size :UInt64) -> ();
```

```
    protect @2 (addr :UInt64, size :UInt64, prot :UInt32) -> ();
}

interface Endpoint {}

interface ProcessSpawner {
    spawn @0 (name :Text, binaryName :Text, grants :List(CapGrant)) ->
(handleIndex :UInt16);
}

interface ProcessHandle {
    wait @0 () -> (exitCode :Int64);
}

interface BootPackage {
    manifestSize @0 () -> (size :UInt64);
    readManifest @1 (offset :UInt64, maxBytes :UInt32) -> (data :Data);
}

# Management-only introspection. Ordinary handle release uses the system
# transport opcode CAP_OP_RELEASE, not a method here.
interface CapabilityManager {
    list @0 () -> (capabilities :List(CapabilityInfo));
    revoke @1 (capId :UInt32) -> ();
    # grant is planned for a later Stage 6 management slice
}
```

Each interface has a unique 64-bit `TYPE_ID` generated by the Cap'n Proto compiler. `TYPE_ID` is the schema constant. `interface_id` is the runtime metadata used by CapSet/bootstrap descriptions and endpoint delivery headers. Method dispatch uses the interface assigned to the capability entry plus `method_id`; `method_id` selects a method inside that schema.

This is not capability identity. A `CapId` is the authority-bearing handle in a process table, analogous to an fd. Multiple capabilities can expose the same interface:

- `cap_id=3` -> serial-backed `Console`
- `cap_id=4` -> log-buffer-backed `Console`
- `cap_id=5` -> `Console` proxy served by another process

All three use the same `Console` `TYPE_ID`, but they are different objects with different authority. The manifest/CapSet should record the expected schema `TYPE_ID` as interface metadata for typed handle construction. Normal CALL SQEs do not need to repeat it because the kernel or serving transport can derive it from the target capability entry. `CapSqe` keeps reserved tail padding for ABI stability.

The kernel exposes the initial CapSet to each process as a read-only 4 KiB page mapped at `capos_config::capset::CAPSET_VADDR` and passes its address in RDX to `_start`. The page starts with a `CapSetHeader { magic, version, count }` and is followed by `CapSetEntry { cap_id,`

name_len, interface_id, name: [u8; 32] } records in manifest declaration order. Userspace looks up caps by the manifest name rather than by numeric index (capos_config::capset::find), so grants can be reordered in system.cue without breaking clients. The mapping is installed without WRITABLE so userspace cannot mutate its own bootstrap authority map.

Security invariant: a CapTable entry exposes one public interface. If the same backing state must be available through multiple interfaces, mint multiple capability entries, each wrapping the same state with a narrower interface. Do not grant one handle that accepts unrelated interface_id values; that makes hidden authority easy to miss during review.

### Invocation Path

Capabilities are invoked via a shared-memory **capability ring** (io_uring- inspired). Each process has a submission queue (SQ) and completion queue (CQ) mapped into its address space. Two invocation paths exist:

```
Caller builds capnp params message
    → serialize to bytes (write_message_to_words)
    → write CALL SQE to SQ ring (pure userspace memory write)
    → advance SQ tail
    → caller invokes cap_enter for ordinary capability methods
      (timer polling only runs explicitly interrupt-safe CALL targets)
    → kernel reads SQE, validates user buffers
    → CapTable.call(cap_id, method_id, bytes)
    → kernel writes CQE to CQ ring
    ... caller reads CQE after cap_enter, or spin-polls only for
        interrupt-safe/non-CALL ring work ...
    → caller reads CQE result
```

CapObject::call does not receive a caller-supplied interface ID. The cap table derives the invoked interface from the target entry before invoking the object. The SQE carries only the capability handle and method ID because each capability entry owns one public interface:

```rust
pub trait CapObject: Send + Sync {
    fn interface_id(&self) -> u64;
    fn label(&self) -> &str;
    fn call(
        &self,
        method_id: u16,
        params: &[u8],
        result: &mut [u8],
        reply_scratch: &mut dyn ReplyScratch,
    ) -> capnp::Result<CapInvokeResult>;
}
```

All communication goes through serialized capnp messages, even when caller and callee are in the same address space. This ensures the wire format is always exercised and makes the transition to cross-address-space IPC seamless.

The result buffer is supplied by the caller (the user-validated SQE result region). Implementations serialize directly into it and return the number of bytes written, so the kernel's dispatch path does not allocate an intermediate `Vec<u8>` per invocation.

### Capability Table

Each process has its own capability table (`CapTable`), created at process startup. The kernel also maintains a global table (`KERNEL_CAPS`) for kernel-internal use. Each table maps a `CapId` (`u32`) to a boxed `CapObject`.

CapId encoding: `[generation:8 | index:24]`. The generation counter increments when a slot is freed, so stale CapIds (from a previous occupant of the slot) are rejected with `CapError::StaleGeneration` rather than accidentally referring to a different capability.

Generation wrap must not resurrect old authority. The implemented table retires a slot permanently when its 8-bit generation would wrap from `255` back to `0`; that slot is not returned to the free list. Heavy churn can therefore exhaust a table even when many retired slots are empty, but the failure mode is `CapError::TableFull`, not stale-cap revalidation. Future widening of `CapId` generation bits is an ABI change and belongs in the schema/ring ABI evolution track.

Operations:

- `insert(obj)` – register a new capability, returns its CapId
- `get(id)` – look up a capability by ID (validates generation)
- `remove(id)` – revoke a capability, bumps slot generation
- `call(id, method_id, params)` – dispatch a method call against the interface assigned to the capability entry

Every current boot manifest gives only `initConfig.init` a kernel-built capability table. The default `system.cue` manifest boots the standalone `init` binary, which reads BootPackage, validates `initConfig.services`, and spawns `capos-shell`, the remote-session CapSet gateway, and resident demo services through ProcessSpawner. The Telnet gateway fixture is retired with the kernel socket owner. Focused shell-led manifests such as `system-smoke.cue` and `system-shell.cue` still boot `capos-shell` directly as `initConfig.init` for narrow login/shell proofs. Focused init-executor manifests such as `system-spawn.cue` also boot the standalone `init` binary with Console, BootPackage, and ProcessSpawner for isolated ProcessSpawner coverage. Child capabilities are assembled from explicit spawn grants in declaration order: raw grants preserve the source capability metadata, legacy endpoint-client grants attenuate an endpoint owner or `ProcessSpawner` endpoint result source to a client facet while preserving delegated receiver metadata, and child-local Endpoint, FrameAllocator, and VirtualMemory grants are minted for the child's process. Endpoint kernel grants return parent-side client facets as result caps; init uses those facets for later service imports and releases them before waiting on children. Kernel bootstrap now builds only `initConfig.init` kernel-sourced caps; `CapSource::Service` resolution stays in init's BootPackage executor path. `CapRef.source` is structured CUE inside `initConfig.services`, not an authority string:

```
{
    name:                "client"
```

```
    expectedInterfaceId: 0xacf0c15a7b2e0041
    source: service: {
        service: "endpoint-server"
        export:  "client"
    }
}
```

The source selector chooses the object or authority to grant. The `expectedInterfaceId` value is a schema compatibility check against the constructed object, not the authority selector itself. This distinction matters because different objects can implement the same interface.

#### Transport-Level Capability Lifetime

Cap'n Proto applications do not usually model capability lifetime as an application method on every interface. The RPC transport owns capability reference bookkeeping.

The standard Cap'n Proto RPC protocol is stateful per connection. Each side keeps four tables: questions, answers, imports, and exports. Import/export IDs are connection-local, not global object names. When an exported capability is sent over the connection, the export reference count is incremented. When the importing side drops its last local reference, the transport sends `Release` to decrement the remote export count. Implementations may batch these releases. If the connection is lost, in-flight questions fail, imports become broken, and exports/answers are implicitly released. Persistent capabilities, when implemented, are a separate `SturdyRef` mechanism and should not be treated as owned pointers.

References:

- Cap'n Proto RPC Protocol: Handling disconnects
- Cap'n Proto `rpc.capnp`: four tables and `Release`

This distinction matters for capOS:

- `close()` is application protocol. A `File.close()` method can flush dirty state, commit metadata, or tell a server that a session should end.
- `Release` / cap drop is transport protocol. It removes one reference from the caller's local capability namespace and eventually lets the serving side reclaim the object if no references remain.
- Process exit is bulk transport cleanup. Dropping the process must release all caps in its table, cancel pending calls, and wake peers waiting on those calls.

capOS therefore needs a system transport layer in the userspace runtime (`capos-rt` / later language runtimes), not just raw SQE helpers. That transport should own typed client handles, local reference counts, promise-pipelined answers, and broken-cap state. When the last local handle is dropped, it should queue a transport-level release operation that is flushed through the kernel ring at an explicit runtime boundary.

Ordinary handle release is a transport concern, not an application method. The target design: the generated client drops the last local handle (RAII / GC / finalizer), the runtime transport queues `CAP_OP_RELEASE`, an explicit runtime flush or later ring-client boundary submits it, and the kernel

removes the caller's `CapTable` slot with mutable access to that table. Encoding ordinary local release as a regular method call on `CapabilityManager` was rejected because it would mutate the same table used to dispatch the call; `CapabilityManager` is therefore management-only (`list()` plus child-scoped `revoke(capId)`, later `grant()`), not the default release path. `CAP_OP_FINISH` remains reserved in the same transport opcode namespace for application-level "end of work" signals that the transport must deliver reliably, so the kernel can tell them apart from a truly malformed opcode.

Current status: the kernel dispatches `CAP_OP_RELEASE` as a local cap-table slot removal and fails closed for stale or non-owned cap IDs. `capos-rt` bootstrap handles remain explicitly non-owning, while adopted owned handles queue `CAP_OP_RELEASE` on final drop and expose `Runtime::flush_releases()` for callers that need to force the queued releases. Result-cap adoption validates the kernel-supplied interface ID before producing an owned typed handle. `CAP_OP_FINISH` remains reserved and returns `CAP_ERR_UNSUPPORTED_OPCODE`. Process exit remains the fallback cleanup path for unreleased local slots.

Queued release is not immediate revocation. A dropped runtime handle no longer provides local typed access in that runtime, but the kernel cap-table slot is removed only after the release SQE is flushed and processed, or during process exit cleanup. Security-sensitive flows that need to invalidate authority for other holders or peers must use explicit revoke/epoch semantics such as `CapabilityManager.revoke`, session expiry, object epochs, or service-specific close/revoke methods; they must not rely on destructor timing.

Session expiry is also not a substitute for every revocation shape. The target session lifecycle model has separate layers:

- a mutable session liveness cell for `live`, `logged_out`, `revoked`, `expired`, and `recovery_only` state behind the immutable process `SessionContext`;
- broker grant leases for bundle fragments and elevated or temporary caps;
- object/facet epochs for invalidating a live target generation.

Renewal acts on the first two layers. It may extend session liveness or mint fresh grant leases, but it must not make old ordinary grants fresh merely because the session renewed. Object/facet revocation remains an independent target-side operation.

Service authors should make this distinction explicit in protocol design:

- Use ordinary handle drop or runtime `flush_releases()` only to stop this process from using one local cap slot.
- Use a service `close` method when the service must observe application-level shutdown, flush durable state, or publish an orderly end-of-session result.
- Use `CapabilityManager.revoke`, session expiry, object epochs, or a service-specific revoke method when existing peers or delegated holders must lose authority before the service proceeds.
- Treat destructor/finalizer timing as advisory cleanup. It is not a security boundary, and it is not proof that another process has stopped using a cap.

**Stale-Handle and Revoke Patterns**

Not all kernel cap families use the same model for handling stale or revoked capabilities. The correct pattern depends on the semantics of the object, not on a blanket epoch test. Using the wrong model produces incorrect tests or incorrect behavior expectations.

**Category A — Exception-based stale guard**

The cap exposes an `ensure_*_live` guard or an equivalent consumed-state check that returns a stable typed exception (not a silent success) on a stale or consumed cap.

- `UserSession` (`kernel/src/cap/user_session.rs`): `info()`/`auditContext()` fail closed with a stable exception message after `logout()`; second `logout` is idempotent. Proved by `run-ssh-public-key-session`.
- `SchedulingContext`, `CpuIsolationLease`: expose an explicit `revoke` method returning `staleGeneration`. Subsequent `info`, `bind_caller_thread`, `activation_preflight`, `create`, and `drain_notifications` calls fail closed on the staled cap. Proved by `run-scheduling-context` (`demos/scheduling-context-smoke/src/main.rs:285-313, 1129-1141`) and `run-scheduler-cpu-isolation-lease` (`demos/cpu-isolation-lease-smoke/src/main.rs:201-237`).
- `ThreadHandle` (`kernel/src/cap/thread_handle.rs`): `join` (`sched.rs:1038-1057`) returns `AlreadyJoined` on the second call (hard fail, not silent success) and returns `TargetNotLive` (`sched.rs:1371,1377,1385`) if the thread record is absent post-cleanup. `exitCode` (`sched.rs:1418-1420`) is a non-consuming idempotent read. The `join_or_register` consumed-state check is the stale guard; the `joined` flag is the epoch. Proved by `run-thread-lifecycle` (`demos/thread-lifecycle/src/main.rs:293-298`).

Per-cap epoch tests are applicable only to Category A caps.

**Category B — Idempotent-stale-target**

The cap returns silent success (or a latched result) on a stale target. No `ensure_*_live` guard is present by design.

- `ProcessHandle` (`kernel/src/cap/process_spawner.rs`): `terminate` on an already-exited process returns `Complete(0)`; `wait` re-reads the latched exit code. Writing fail-closed tests for Category B caps would test the opposite of intended behavior.

**Category C — Soft-EOF / zero-write**

The cap uses v0 `ExceptionType` policy: closing one side causes the other to drain and receive EOF; writes return zero bytes rather than an error.

- `Pipe` (`kernel/src/cap/pipe.rs`): `close` causes read to drain + EOF, write returns zero bytes (schema lines 2429-2433). No epoch test needed.

**Category D — No revoke verb (kernel singletons)**

These caps expose no `revoke` or `close` method in the schema. The backing object lives for the process lifetime.

`CredentialStore`, `AuthorizedKeyStore`, `SshHostKey`, `EntropySource`, `SystemInfo`, `AuditLog`, `HardwareAuditLog`, `SessionManager`, `AuthorityBroker`, `RestrictedLauncher`, `BootPackage`. Nothing to test for stale-handle behavior.

**Category E — DDF caps with release/scrub semantics**

These caps use internal handle epoch validation. The full stale-handle behavior for each requires targeted per-cap investigation when a behavior gap is identified.

`DmaBuffer`, `DeviceMmio`, `Interrupt`.

**Open residuals**

- **UserSession expiry path (Category A)**: the `expiresAtMs`/`anonymousMs`- driven expiry path is not yet covered by a focused smoke. `run-ssh-public-key-session` covers the explicit `logout()` close-side path. Note that `run-session-context` is flaking on TCG-only hosts — a stability fix is needed before that smoke can be strengthened.

### Access Control: Interfaces, Not Rights Bitmasks

capOS deliberately does **not** use a rights bitmask (READ/WRITE/EXECUTE) on capability entries, despite this being standard in Zircon and seL4. The reason is that Cap'n Proto typed interfaces already serve as the access control mechanism, and a parallel rights system creates an impedance mismatch.

**Why rights bitmasks exist in other systems:** Zircon and seL4 use rights because their syscall interfaces are untyped – a handle is an opaque reference to a kernel object, and the kernel needs something to decide which fixed syscalls are allowed. capOS has typed interfaces where the `.capnp` schema defines exactly what methods exist.

**capOS's approach: the interface IS the permission.** To restrict what a caller can do, grant a narrower capability:

- `Fetch` (full HTTP) ⟶ `HttpEndpoint` (scoped to one origin)
- `Store` (read-write) ⟶ `Store` wrapper that rejects write methods
- `Namespace` (full) ⟶ `Namespace` scoped to a prefix

The "restricted" capability is a different `CapObject` implementation that wraps the original. The kernel doesn't know or care – it dispatches to whatever `CapObject` is in the slot. Attenuation is userspace/schema logic, not a kernel mechanism.

**Session transfer scope:** capability holds now carry reference-level transfer scope. `same_session` caps cannot move into another process session through raw IPC, endpoint return, or spawn grants. `cross_session_shareable` caps may cross and then invoke under the receiver process session. `service_regrant_only` caps require a trusted fixed-session broker/launcher path. These meta-rights are about the reference, not the referenced object, and do not overlap with interface-level method access control.

**Non-writable filesystem caps are forwardable to a same-session child; writable caps are not.** `Directory`/`File` caps are minted `Copy`/`same_session` at the read-only and RAM mint sites, so a holder can forward an opened directory or file to a `ProcessSpawner.spawn` child within the same session – the kernel handoff that backs POSIX fd inheritance across fork/execve. The security argument is the same for all of them: the child gains no authority the parent does not already hold, `same_session` keeps the cap from escaping the session, and the spawn-grant epoch wrapper keeps a forwarded child cap from outliving a revoked parent. Two flavours exist:

- **Read-only views** – the read-only filesystem (`readonly_fs`) and the packaged-image source (`installable_image`), plus their `read_only_fs_root`/ `installable_image_source` bootstrap roots. Their interfaces fail closed on every mutation, so forwarding shares a pure read view. Here *the interface is the permission* makes the share unambiguously benign.
- **The holder's own RAM scratch namespace** – the `directory::transfer_result_cap` results and the `kernel:directory`/`kernel:file` bootstrap sources (via `boot_cap_hold`). This `Directory`/`File` interface includes mutation methods, so the forwarded cap is shared *read/write* with the child, not a read view. It is still safe to forward because it is the parent's own scratch tree shared within one session, not a privilege the parent lacked.

The disk-backed *writable* filesystem (`writable_fs`) is a distinct `CapObject` type minted `NonTransferable`: a writable cap carries the filesystem-wide single-writer claim, so forwarding it would let two processes hold that claim. The `ProcessSpawner` Raw/Move grant modes reject a `NonTransferable` source, so the single-writer policy is preserved by the mint-time mode rather than a separate check. Proven by `make run-spawn-grant-directory`.

**`TerminalSession` is forwardable to a same-session child, parent-retained.** The bootstrap `TerminalSession` cap is minted `Copy`/`same_session` (matching `Console`) in `boot_cap_hold`, so a holder can forward its terminal-backed stdout/stderr to a `ProcessSpawner.spawn` child without losing its own terminal. `TerminalSessionCap` is a stateless unit struct: `write`/`writeLine` dispatch onto the shared kernel terminal and `readLine` resolves the caller's session context per call (`requires_live_caller_session` stays true), so there is no per-session ownership state to strip on a forward. The child gains no terminal authority the parent did not already hold, and `same_session` keeps the cap from escaping the session. This is the non-destructive capability-model realization of POSIX "all children share the controlling tty"; the prior `Move`/ `service_regrant_only` mint was a policy default, not a state-ownership requirement, and a destructive `Move` would have stripped a shell of its terminal on its first child spawn under full fd inheritance. Two writers reaching the same terminal serialize at the shared kernel UART; sub-line interleaving between a parent and a child writing concurrently is an accepted research-surface limitation, not an authority leak. Proven by `make run-posix-terminal-forward`.

See research survey for the cross-system analysis that led to this decision (§1 Capability Table Design).

### Planned Enhancements (from research)

Tracked in Roadmap Stages 5-6:

- **Legacy badge / receiver selector** – the current storage field is a `u64` per capability hold edge, delivered to endpoint servers on invocation. Existing code still calls it a badge because it began as seL4-style client identity metadata. The active model keeps that field out of service identity: new service capability should use one immutable process session, broker-granted service roots/ facets, privacy-preserving endpoint caller-session metadata, and explicit subject disclosure plus a matching disclosure scope when a service needs more than an opaque service-scoped session reference.
- **Epoch** (from EROS) – per-object revocation epoch. Incrementing the epoch invalidates all outstanding references. O(1) revoke, O(1) check.

**Current Limitations**

- **Process-ring blocking remains process-level; private ParkSpace waits are per-thread.** `cap_enter(min_complete, timeout_ns)` processes pending SQEs and can block one admitted thread per process until enough CQEs exist or a finite timeout expires. That ring wait is still process-owned and does not make the capability ring itself a per-thread completion queue. Separately, the implemented private ParkSpace path provides process-local per-thread wait/wake on userspace words through compact `CAP_OP_PARK`/`CAP_OP_UNPARK` operations. SharedParkSpace park-words and runtime safe park clients remain future work.
- **No persistence.** Capabilities exist only at runtime.
- **Capability transfer is implemented for Endpoint CALL/RECV/RETURN.** Transfer descriptors on the capability ring let callers and receivers copy or move transferable local caps through IPC messages. Delivery also enforces the cap hold's session transfer scope; an unsupported cross-session transfer fails with `CAP_ERR_TRANSFER_NOT_SUPPORTED` and is reported to the caller instead of being requeued to the endpoint. See Proposal: Storage, Naming, and Persistence "IPC and Capability Transfer" for the full design.
- **Transfer ABI (3.6.0 draft).** Sideband transfer descriptors are defined in `capos-config/src/ring.rs` as `CapTransferDescriptor`:
  - `cap_id` is the sender-side local capability-table handle.
  - `transfer_mode` is either `CAP_TRANSFER_MODE_COPY` or `CAP_TRANSFER_MODE_MOVE`.
  - `xfer_cap_count` in `CapSqe` is the descriptor count.
  - For CALL/RETURN, descriptors are packed at `addr + len` after the payload bytes and must be aligned to `CAP_TRANSFER_DESCRIPTOR_ALIGNMENT`.
  - Result-cap insertion semantics are defined by `CapCqe`: `result` reports normal payload bytes, while `cap_count` reports how many `CapTransferResult { cap_id, interface_id }` records were appended immediately after those payload bytes in `result_addr` when `CAP_CQE_TRANSFER_RESULT_CAPS` is set. User space must bound-check `result + cap_count * CAP_TRANSFER_RESULT_SIZE` against its requested `result_len`.
  - Future promise pipelining must target that sideband result-cap namespace: `pipeline_dep` names a process-local promised answer, and `pipeline_field` is a zero-based `CapTransferResult` record index in that answer's completion. It is not a Cap'n Proto schema field number; the kernel must not traverse opaque result payload bytes to find a capability.
  - Transfer-bearing SQEs are fail-closed:
    - unsupported transfer scope or object class: `CAP_ERR_TRANSFER_NOT_SUPPORTED`,
    - malformed descriptor metadata (invalid mode, reserved bits, non-zero `_reserved0`, misalignment, overflow): `CAP_ERR_INVALID_TRANSFER_DESCRIPTOR`,
    - all other reserved-field misuse remains `CAP_ERR_INVALID_REQUEST`.
- **Revocation propagates through object epochs.** `CapabilityManager.revoke` invalidates child-local grant copies for the revoked object, and the ring maps revoked ordinary and endpoint use to typed `Disconnected` exceptions where a result buffer exists. Broader supervision/restart policy remains future work.
- **MemoryObject is the mapped bulk-data substrate.** `FrameAllocator` returns owned `MemoryObject` result caps instead of raw physical addresses. The object exposes metadata plus

caller-local map/unmap/protect operations for page-aligned ranges. File I/O, networking, GPU data planes, and zero-copy IPC still need service-level SharedBuffer operations built on this substrate. See Proposal: Storage, Naming, and Persistence “Shared Memory for Bulk Data” for the broader interface design.

**Future Directions**

- **Broader capability-bearing services.** Endpoint CALL/RECV/RETURN already carry copy/move sideband transfer descriptors and install result caps in the receiver’s local table. Remaining work is to use that transport in higher service layers: capability-bearing naming and persistence services, Directory/File and Namespace-style object models, promise pipelining over result-cap indexes, and policy for durable references. See Proposal: Storage, Naming, and Persistence.
- **Persistence.** Persistent object references should be restored through a capability-bearing naming or persistence service that can authorize the request and mint a fresh live object. Do not serialize local cap-table handles, endpoint generations, receiver selectors, or server cookies as durable authority.
- **Network transparency.** Remote capability transport should use connection-local export/import tables and explicit disconnect semantics. A remote Console capability can expose the same typed interface as a local one, but the portable authority is the live object reference, not a global URL or serialized local routing selector.

## ABI Evolution Policy

This policy governs externally visible capOS ABIs:

- Cap'n Proto schema in `schema/capos.capnp`.
- Generated schema bindings checked by `make generated-code-check`.
- Ring and bootstrap ABI constants and layouts in `capos-config/src/ring.rs`, `capos-config/src/capset.rs`, and `capos-abi/src/lib.rs`.
- Debug/log formats only when a document explicitly declares them stable.

The current project is still a research tree, not a released platform with a public compatibility promise. Even so, schema and ring changes must follow this policy before external clients, host tools, or out-of-tree runtimes depend on them.

### Design Grounding

This policy is grounded in current capOS docs and the checked-in prior-art notes that apply to schema and transport evolution:

- `docs/architecture/capability-ring.md` for the implemented process-wide ring, fixed 64-byte `CapSqe`, fixed 32-byte `CapCqe`, opcode boundary, and current completion semantics.
- `docs/proposals/ring-v2-smp-proposal.md` for the undecided future per-thread-ring version-negotiation shape.
- `docs/proposals/error-handling-proposal.md` for the transport/application error split and unsupported-operation behavior.
- `docs/trusted-build-inputs.md` for generated-code drift checks and pinned Cap'n Proto tooling.
- `docs/design-risks-register.md` for the prior open ABI compatibility and Ring v2 compatibility questions.
- `docs/research/capnp-error-handling.md` for Cap'n Proto exception and schema error-model precedent. OS scheduling, filesystem, networking, and hardware prior-art research does not directly change this schema/ring ABI policy.

### Compatibility Classes

Every ABI change must name one class in its task, review, or commit message.

- **Class:** Compatible addition
  - **Meaning:** Existing clients keep working without recompilation or behavior change.
  - **Required handling:** Add tests or generated-code drift evidence. Update docs when semantics matter.
- **Class:** Compatible tightening
  - **Meaning:** Existing malformed or previously unspecified inputs fail earlier or more specifically.
  - **Required handling:** Document the rejected shape and expected error. Add hostile coverage when reachable from userspace.
- **Class:** Soft deprecation

  - **Meaning:** Old shape still works, but new callers should stop using it.
  - **Required handling:** Mark the field/method/opcode as deprecated in docs and keep a replacement path live through the deprecation window.
- **Class:** Breaking change
  - **Meaning:** Existing valid clients can fail, observe different semantics, or require regenerated code.
  - **Required handling:** Requires a proposal or backlog plan, migration notes, compatibility proof or explicit break decision, and task/risk updates when relevant.
- **Class:** Internal-only
  - **Meaning:** Not visible outside one crate or generated artifact and not serialized, mapped, or invoked across a boundary.
  - **Required handling:** Normal code review; do not label serialized or mapped data as internal-only.

### Cap'n Proto Schema Rules

Schema interface IDs, method ordinals, struct field ordinals, enum discriminants, union tags, and named constants are stable once checked in.

Allowed compatible changes:

- Add a new field with a new ordinal and a default value that old readers can safely ignore.
- Add a new method with a new ordinal when old clients do not need it.
- Add a new result union arm only when old clients already treat unknown or unsupported domain outcomes as a controlled failure.
- Add a new interface or struct with a fresh ID/name.
- Add documentation that narrows previously undocumented behavior without changing wire compatibility.

Disallowed without a breaking-change plan:

- Reuse a removed field, method, enum, or union ordinal.
- Change the meaning, type, units, authority, or lifetime of an existing field.
- Rename a schema item when generated code or logs expose the old name as a public integration surface.
- Make an optional/defaulted field mandatory for existing callers without a versioned fallback.
- Replace a schema result union with a transport error or vice versa without an error-layer migration note.

Removed schema space stays reserved. If a field or method is retired, leave a comment at the old ordinal explaining why it is reserved and where the replacement lives.

### Ring ABI Rules

The ring ABI is a fixed-layout shared-memory contract. `CapSqe`, `CapCqe`, ring header fields, opcodes, flags, transfer descriptor layout, CQE result codes, and fixed virtual addresses are kernel/userspace ABI.

Rules for the current process-wide ring:

- Do not change the size, alignment, byte order, or meaning of an existing ring struct field without a breaking-change plan.
- Preserve objective layout checks for current ABI structs. At minimum, `capos-config/src/ring.rs` must keep compile-time checks for `CapSqe`, `CapCqe`, `CapTransferDescriptor`, endpoint caller-session metadata, endpoint message headers, and ring capture records. Any new negotiated ring layout must add equivalent checked constants for SQE size, CQE size, transfer descriptor size, ring header offsets, SQE/CQE array offsets, and feature/version fields.
- Do not change `SQE_ARRAY_OFFSET`, `CQE_ARRAY_OFFSET`, `SQ_ENTRIES`, `CQ_ENTRIES`, `RING_VADDR`, or fixed SQE/CQE sizes by arithmetic side effect. A change to any of those values is a layout change and must name its compatibility class.
- Reserved SQE fields must be rejected unless the opcode explicitly defines them. New meanings for reserved fields require hostile tests that old kernels fail closed.
- New opcodes must start as reserved or unsupported. A reserved opcode should return `CAP_ERR_UNSUPPORTED_OPCODE`; malformed non-reserved opcodes should return `CAP_ERR_INVALID_REQUEST`.
- New flags must specify whether old kernels reject them, ignore them, or treat them as malformed. Silent ignore is allowed only for flags that cannot carry authority or resource effects.
- New negative CQE result codes must be appended as new constants. Existing negative result codes cannot be renumbered or repurposed.
- Capability transfer descriptors must continue to reject unknown reserved bits until a documented transfer mode consumes them.

Ring v2 or per-thread-ring work must declare whether it is:

- a negotiated compatible extension to the current ring page;
- a new ring layout selected by boot/runtime version negotiation; or
- an intentional ABI break.

That decision belongs in the Ring v2 proposal/backlog before implementation.

### Version Negotiation

When an ABI cannot be evolved by compatible addition, introduce an explicit version gate instead of inferring compatibility from struct size or accidental behavior.

Acceptable gates include:

- manifest or boot-package `schemaVersion` fields;
- a future runtime boot-info field that names ring layout and feature bits;
- interface methods that return a structured unsupported-version result;
- manifest/tooling checks that reject unsupported data versions before boot.

Unsupported versions must fail closed with a stable, documented error. A client must not need to parse debug text to distinguish “unsupported version” from “malformed input”.

### Deprecation Window

Before external consumers exist, a deprecation may be removed after the replacement path, docs, and smokes land in `main`.

After external consumers are declared for an ABI, deprecated schema or ring surfaces must remain for at least one full selected milestone after the replacement is documented and tested. Removing them earlier is a breaking change and must be called out as such.

Deprecation notes must name:

- the old field, method, opcode, flag, or constant;
- the replacement;
- the last proof target that still exercises the old shape;
- the planned removal condition.

### Review Gates

Schema or ring ABI changes must include the relevant checks:

- `make generated-code-check` for `schema/capos.capnp` changes.
- `cargo test-config` for manifest/schema validation changes.
- `cargo test-ring-loom` for ring queue protocol changes.
- Compile-time layout assertions and host tests for ring struct size, alignment, offsets, entry counts, and fixed virtual addresses when a ring layout changes.
- `cargo test-lib` for CapTable/capability transfer semantics.
- A focused QEMU smoke when a userspace-visible behavior changes.
- `make docs` for policy or manual changes.

Reviewers should reject ABI changes that lack a compatibility class, migration notes for breaking behavior, or an unsupported-version/error story for new version gates.

### Current Open ABI Decisions

- Ring v2 backward compatibility remains undecided. Until it is decided, do not claim per-thread rings are compatible with the current process-wide ring.
- Production release reproducibility remains separate from ABI compatibility. Final ISO, manifest, and embedded ELF checksums are tracked in `docs/trusted-build-inputs.md` and relevant task records.

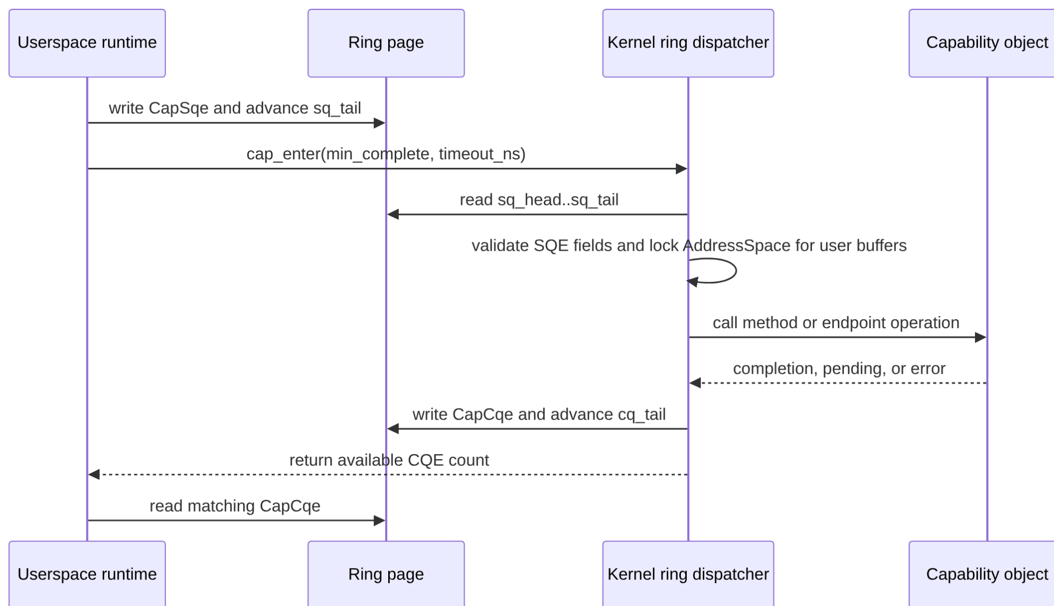
## Capability Ring

The capability ring is the userspace-to-kernel transport for capability invocation. It avoids one syscall per operation while preserving a typed Cap'n Proto method boundary and explicit completion reporting.

The current error model is documented in Error Handling. Ring CQE status values report transport failures; typed capability exceptions and ordinary schema result unions sit above that transport layer.

### Current Behavior

Each non-idle process gets one 4 KiB ring page mapped at `RING_VADDR`. The page contains a volatile header, a 16-entry submission queue, and a 32-entry completion queue. Userspace writes `CapSqe` records, advances `sq_tail`, and uses `cap_enter(min_complete, timeout_ns)` to make ordinary calls progress.




Timer polling also processes each current process's ring before preemption, but only non-CALL operations and CALL targets that explicitly allow interrupt dispatch may run there. Ordinary CALLs wait for `cap_enter`.

> **Why ordinary CALL waits for cap_enter:** Submitting a `CALL` SQE is only a shared-memory write. The kernel still needs a safe execution point to drain the ring and run capability code. Timer polling runs in interrupt context, so it must not execute arbitrary capability methods that may allocate, block on locks, mutate page tables, spawn processes, parse Cap'n Proto messages, or perform IPC side effects. `cap_enter` is the normal process-context drain point: it processes pending SQEs, posts CQEs, and then either returns the available completion count or blocks until enough completions arrive. The design keeps SQE publication syscall-free and batchable, keeps the syscall ABI limited to `exit` and `cap_enter`, and avoids turning the timer interrupt into a general capability executor. A future SQPOLL-style path can remove the

> explicit syscall from the hot path only by running dispatch in a worker context, not from arbitrary timer interrupt execution.

### Design

`CapSqe` is a fixed 64-byte ABI record. `CAP_OP_CALL` names a local cap-table slot and method ID plus parameter/result buffers. `CAP_OP_RECV` and `CAP_OP_RETURN` implement endpoint IPC. `CAP_OP_RETURN` normally returns successful result bytes to the original caller; with `CAP_SQE_RETURN_APPLICATION_EXCEPTION`, its payload is a serialized `CapException` and the original caller completes with `CAP_ERR_APPLICATION_EXCEPTION` or the truncated application-exception code. `CAP_OP_RELEASE` removes a local cap-table slot through the transport. `CAP_OP_CANCEL` (opcode 6) cancels a pending endpoint receive posted by the same process on the same endpoint cap; `pipeline_dep` carries the receive SQE's `user_data`. `CAP_OP_NOP` measures the fixed ring path. `CAP_OP_PARK_BENCH` (opcode 7) is a measurement-only compact opcode dispatched only by kernels built with the `measure` feature; normal kernels reject it as malformed. `CAP_OP_FINISH` is ABI-reserved and currently returns `CAP_ERR_UNSUPPORTED_OPCODE`.

`CAP_OP_RELEASE` is deliberately scoped to local transport cleanup. It removes one holder's cap-table slot after the SQE is processed, or as part of process exit cleanup; it does not revoke peer-held caps, cancel delegated authority, or stand in for an application `close` method. Services that need security-visible invalidation must use an explicit control path such as `CapabilityManager.revoke`, session expiry, object epochs, or a service-specific close/revoke protocol. Reviewers should treat claims based only on handle drop, RAII, GC finalizers, or queued release flushing as local-cleanup claims, not revocation claims.

> **Opcode boundary:** Ring opcodes are kernel ABI, not a loophole around the syscall surface. `cap_enter` and `exit` remain the CPU trap entrypoints, but every accepted authority-bearing or resource-mutating `CAP_OP_*` still adds distinct kernel semantics that must pass the capability method / ring opcode / syscall decision graph. No-authority diagnostics such as `CAP_OP_NOP` are still kernel ABI and must stay side-effect-free and review-visible, but they are not resource authority paths. `CAP_OP_PARK` and `CAP_OP_UNPARK` are justified because blocking wait mutates scheduler state, must be thread-owned on the process ring, reserves completion credit for later wake/timeout delivery, and needs compact capability-authorized hot-path framing. They are not a precedent for moving ordinary object methods into the opcode table for convenience.

`CAP_OP_CALL` may set `CAP_SQE_THREAD_OWNED` with `call_id` equal to the owning thread id. If another thread drains the shared process ring first, the kernel leaves that SQE at the head instead of consuming it and returns a distinct owner-head `cap_enter` result instead of blocking the non-owner behind it. This is limited to context-sensitive self-thread operations such as `ThreadControl.exitThread`; ordinary runtime submissions leave `call_id = 0`.

`CAP_OP_PARK` and `CAP_OP_UNPARK` are compact capability-authorized operations for process-local ParkSpace. Wait SQEs must set `CAP_SQE_THREAD_OWNED` with `call_id` equal to the owning thread id; a non-owner `cap_enter` leaves the SQE at the head just like a thread-owned CALL. They reject promise-pipeline fields and run only from syscall-context ring dispatch, not timer polling. A blocking wait consumes the SQE but posts no caller CQE immediately; instead it reserves one waiter CQE credit, parks the current thread, and later completes with a non-negative park status.

Ordinary CQE posting treats reserved park credits as unavailable so wake and timeout delivery cannot lose waiter completions.

The kernel copies user params into preallocated per-process scratch, dispatches capability methods, copies serialized results into caller-provided result buffers, and posts `CapCqe`. Current-process user copies and transfer-descriptor loads hold the caller's `AddressSpace` mutex across permission validation and the actual HHDM-backed copy/read. A successful method returns non-negative bytes written. Transport failures are negative `CAP_ERR_*` codes. Application exceptions are serialized `CapException` payloads with `CAP_ERR_APPLICATION_EXCEPTION`. Ordinary capability implementation errors and live endpoint CALL/RETURN target errors use this application-exception path once a valid target cap or accepted endpoint relationship has been identified; malformed ring metadata, bad user buffers, lookup failures, and endpoint rollback/transfer failures stay in the transport namespace.

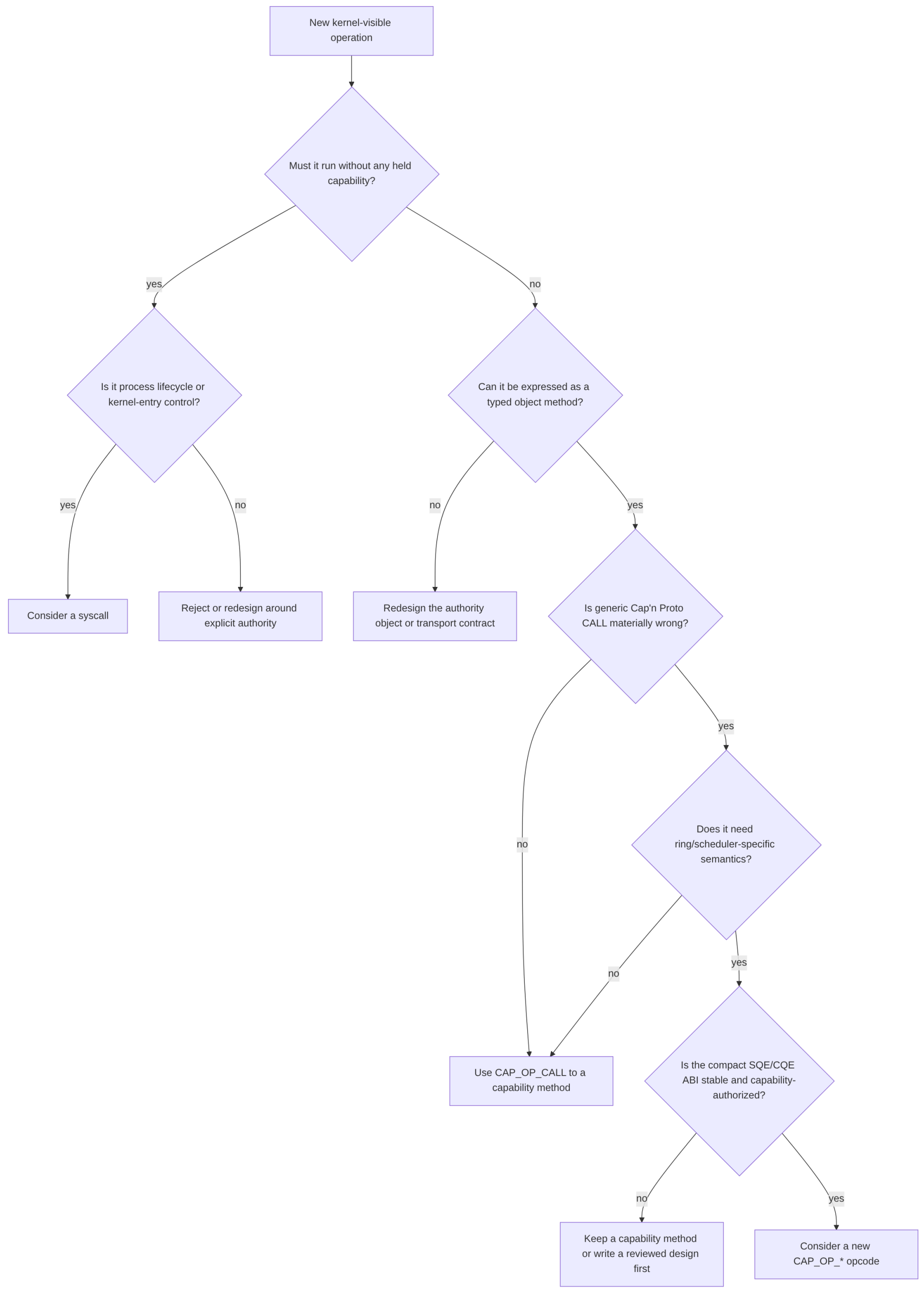
Transfer-bearing CALL and RETURN SQEs pack `CapTransferDescriptor` records after the params/result payload. Successful result-cap transfers append `CapTransferResult` records after normal result bytes.

Promise-pipelined CALLs remain rejected by current kernels. When that flag is enabled, `pipeline_dep` names a process-local promised-answer identifier, and `pipeline_field` selects a zero-based `CapTransferResult` record from that answer's completion. It is not a Cap'n Proto schema field number or payload path. The kernel resolves dependencies only through the sideband result-cap records it already owns; normal result bytes stay opaque to the transport.

Future behavior should use the reserved SQE fields for system transport features, not ad hoc per-interface extensions.

### Choosing A Capability Method, Ring Opcode, Or Syscall

New kernel functionality should default to a normal typed capability method. The small syscall surface is only the trap surface; the ring opcode table is also a reviewed kernel ABI and must stay narrow. The decision tree below is a full-page reference in the PDF because the branches are easier to read at diagram scale than as compressed prose.



New kernel-visible operation
Must it run without any held capability?
yes
no
Is it process lifecycle or kernel-entry control?
Can it be expressed as a typed object method?
yes
no
no
yes
Consider a syscall
Reject or redesign around explicit authority
Redesign the authority object or transport contract
Is generic Cap'n Proto CALL materially wrong?
yes
no
Does it need ring/scheduler-specific semantics?
no
yes
Use CAP_OP_CALL to a capability method
Is the compact SQE/CQE ABI stable and capability-authorized?
no
yes
Keep a capability method or write a reviewed design first
Consider a new CAP_OP_* opcode

Use a normal capability method when the operation is control plane, policy driven, service-specific, infrequent, or naturally represented by Cap'n Proto params/results. Process spawning, credential checks, storage naming, shell or network policy, virtual-memory control-plane calls, and most device-specific commands belong here unless measurement and design review prove otherwise.

Consider a compact ring opcode only when all of these are true:

- The operation is a hot path or scheduler path where generic Cap'n Proto framing is materially wrong.
- The operation has a small, stable field layout that fits the existing SQE/CQE model without per-interface ad hoc extensions.
- It needs ring-specific behavior such as thread ownership, reserved completion credit, CQ ordering/backpressure, asynchronous completion delivery, or interaction with the process ring head.
- It remains authorized by a held capability in `cap_id`, not by ambient process identity or guessed kernel object names.
- It cannot be handled as a normal capability method plus a future generated fast client without losing an essential scheduler or transport invariant.

Consider a new syscall only when the operation is about entering or leaving the kernel execution context itself and cannot sensibly be authorized by a capability already available to the process. That bar is intentionally higher than the opcode bar. Ordinary resource operations should not become syscalls just because they are common.

#### Full-SMP Direction

The current process-wide ring is not the target ABI for full SMP. Once sibling threads in one process can run on different CPUs, a shared process CQ would force userspace to serialize completion consumption or the kernel to invent specific-wait state on top of circular-buffer slots.

The selected future direction is per-thread ring ownership, documented in Ring v2 For Full SMP. In that model, `cap_enter(min_complete, timeout_ns)` keeps its current aggregate wait shape, but the aggregate is the current thread's CQ. Completion paths post by generation-checked `ThreadRef`, while result-cap transfers and authority still belong to the process cap table.

The first Ring v2 implementation should use kernel-chosen child-thread ring mappings. The initial fixed `RING_VADDR` mapping becomes a compatibility special case backed by the same `RingEndpoint` lifetime and waiter rules as child-thread rings. Runtime-supplied ring address ranges are deferred until `VirtualMemory` can reserve a ring arena without racing ordinary mappings.

The initial Phase C multi-CPU scheduler proof may continue to use the current process-wide ring as long as userspace serializes ring consumption. Ring v2 is the target for full SMP with sibling threads from one process running and waiting independently on different CPUs.

A runtime reactor can bridge the current process-wide ring for multithreaded runtimes before Ring v2: one runtime-owned drainer consumes the process CQ, matches completions by

`user_data`, and wakes waiting threads through ParkSpace. That bridge is not the full-SMP kernel ABI.

### Invariants

- SQ and CQ sizes are powers of two and fixed by the ABI.
- Unknown opcodes fail closed; `FINISH` is reserved, not silently accepted.
- Reserved fields must be zero for currently implemented opcodes, except `CAP_SQE_THREAD_OWNED` CALL and PARK SQEs may carry the owning thread id in `call_id`.
- Park PARK/UNPARK SQEs must keep unsupported fields zero and must not be dispatched from timer context.
- `cap_enter` rejects `min_complete > CQ_ENTRIES`.
- User-buffer validation and copy/read must hold the owning process `AddressSpace` mutex for CALL params/results, RECV result buffers, RETURN payloads, transfer descriptors, and deferred same-process completions.
- Timer dispatch must not run capabilities that allocate, block on locks, or mutate page tables unless the cap explicitly opts in.
- Per-dispatch SQE processing is bounded by `SQ_ENTRIES`.
- Transfer descriptors must be aligned, valid, and bounded by `MAX_TRANSFER_DESCRIPTORS`.
- Promise-pipelined dependency resolution must use sideband `CapTransferResult` ordinals, never general Cap'n Proto result traversal in the kernel.

### Code Map

- `capos-config/src/ring.rs` - shared ring ABI, opcodes, errors, SQE/CQE structs, endpoint message headers, transfer records.
- `kernel/src/cap/ring.rs` - kernel dispatcher, SQE validation, CQE posting, cap calls, endpoint CALL/RECV/RETURN, release, transfer framing.
- `kernel/src/arch/x86_64/syscall.rs` - `cap_enter` syscall.
- `kernel/src/sched.rs` - timer polling, cap-enter blocking, direct IPC wake.
- `kernel/src/process.rs` - ring page allocation and mapping.
- `capos-rt/src/ring.rs` - runtime ring client, pending calls, transfer packing, result-cap parsing.
- `capos-rt/src/entry.rs` - single-owner runtime ring client token and release queue flushing.
- `capos-config/tests/ring_loom.rs` - bounded producer/consumer model.

### Validation

- `cargo test-ring-loom` validates SQ/CQ producer-consumer behavior, capacity, FIFO, CQ overflow/drop behavior, and corrupted SQ recovery.
- `make run` exercises Console CALLs, reserved opcode rejection, ring corruption recovery, NOP, fairness, transfers, and endpoint IPC.

- `make run-measure` exercises measurement-only counters, dispatch segment cycle summaries, the NullCap baseline, the ParkBench compact-versus-generic comparison, and the real ParkSpace blocked/resume timing path.
- `cargo test-config` covers shared ring layout and helper invariants.
- `make capos-rt-check` checks userspace runtime ring code under the bare-metal target.

**Open Work**

- Implement `CAP_OP_FINISH` as part of the system Cap'n Proto transport.
- Implement promise pipelining using the reserved `pipeline_dep` answer ID and `pipeline_field` result-cap ordinal mapping.
- Define LINK, DRAIN, and MULTISHOT semantics before accepting those flags.
- Add runtime-level ParkSpace wrappers and completion demultiplexing on top of the compact opcodes.
- Add the runtime reactor bridge for multithreaded use of the current process ring, then replace it as the kernel fast path with per-thread Ring v2 completion ownership.
- Add SQPOLL after SMP gives the kernel a spare execution context.

## Error Handling

capOS uses three error layers for capability invocation. Keeping the layers separate prevents malformed transport state from looking like a service-domain decision, and prevents ordinary business outcomes from becoming generic kernel exceptions.

### Current Model

- **Layer:** Transport status
  - **Carrier:** Negative `CapCqe.result` codes
  - **Use:** Ring, opcode, lookup, buffer, transfer, and dispatch failures where no safe typed payload boundary exists.
- **Layer:** Capability exception
  - **Carrier:** Serialized `CapException` plus `CAP_ERR_APPLICATION_EXCEPTION` or `CAP_ERR_APPLICATION_EXCEPTION_TRUNCATED`
  - **Use:** Capability-level infrastructure failures after a target capability or accepted endpoint relationship exists.
- **Layer:** Schema result union
  - **Carrier:** Interface-specific result payload
  - **Use:** Expected service or domain outcomes such as not-found, denied-by-policy, conflict, invalid domain input, or accepted/rejected business results.

Transport failures are intentionally small and mechanical. Examples include a bad SQE layout, an invalid params or result buffer, an unsupported opcode, a malformed transfer descriptor, or a capability lookup that fails before a live target object is identified.

Capability exceptions are for infrastructure failures at a valid capability boundary: target gone, target overloaded, method unimplemented, argument value rejected by the documented capability contract, or a target-side invariant failure. The exception message is diagnostic and must not carry kernel pointers, secret bytes, or unrelated process-private state.

Schema result unions are the normal application surface. A filesystem `notFound`, service-level `permissionDenied`, ordinary conflict, or accepted conditional rejection belongs in the interface result, not in `CapException`.

### Current Transport Namespace

The ring transport uses signed 32-bit completion results. Non-negative values are opcode-specific successes. Negative values are defined in `capos-config/src/ring.rs`:

- **Code:** `-1`
  - **Name:** `CAP_ERR_INVALID_REQUEST`
  - **Meaning:** Malformed request metadata or a non-reserved opcode value.
- **Code:** `-2`
  - **Name:** `CAP_ERR_INVALID_PARAMS_BUFFER`
  - **Meaning:** Params buffer is unmapped, out of range, or unreadable.
- **Code:** `-3`

- **Name:** `CAP_ERR_INVALID_RESULT_BUFFER`
  - **Meaning:** Result buffer is unmapped, out of range, or unwritable.
- **Code:** `-4`
  - **Name:** `CAP_ERR_INVOKE_FAILED`
  - **Meaning:** Lookup or dispatch failed before a successful typed result was produced.
- **Code:** `-5`
  - **Name:** `CAP_ERR_UNSUPPORTED_OPCODE`
  - **Meaning:** Opcode is reserved but not dispatched by this kernel.
- **Code:** `-6`
  - **Name:** `CAP_ERR_TRANSFER_NOT_SUPPORTED`
  - **Meaning:** Transfer mode or descriptor layout is recognized but unsupported.
- **Code:** `-7`
  - **Name:** `CAP_ERR_INVALID_TRANSFER_DESCRIPTOR`
  - **Meaning:** Transfer descriptor layout is malformed or carries reserved bits.
- **Code:** `-8`
  - **Name:** `CAP_ERR_TRANSFER_ABORTED`
  - **Meaning:** Transfer transaction failed without committing partial capability state.
- **Code:** `-9`
  - **Name:** `CAP_ERR_APPLICATION_EXCEPTION`
  - **Meaning:** A structured `CapException` was written to the result buffer.
- **Code:** `-10`
  - **Name:** `CAP_ERR_APPLICATION_EXCEPTION_TRUNCATED`
  - **Meaning:** An exception occurred, but no complete detail fit in the result buffer.

### Capability Exceptions

`schema/capos.capnp` defines `ExceptionType` and `CapException`. The current exception kinds are `Failed`, `Overloaded`, `Disconnected`, `Unimplemented`, and the capOS-specific `InvalidArgument`.

The kernel serializes ordinary capability implementation errors through `kernel/src/cap/ring.rs`. `capos-rt/src/client.rs` decodes application-exception CQEs into `ClientError::Application(ApplicationException)`. The runtime treats `Disconnected` as a broken local handle.

A path should produce `CapException` only when all of these are true:

- a live target capability was identified, or an endpoint operation is acting on an already accepted call, receive, or return relationship;
- the failure is attributable to capability semantics rather than malformed ring metadata;
- the affected caller supplied a result buffer large enough to receive the serialized exception, otherwise the result is the truncated exception code.

### Endpoint RETURN

Endpoint RETURN is asymmetric because the result belongs to the original caller, not the returning receiver. A server can set `CAP_SQE_RETURN_APPLICATION_EXCEPTION` on `CAP_OP_RETURN` to return a serialized `CapException` to the caller. The server's own RETURN completion reports only whether the return transport succeeded.

Revoked endpoint RETURN also reports `Disconnected` to the original caller when that caller supplied a result buffer. Receiver-side lookup and CQ-space failures that cannot be tied to the caller's result buffer remain transport failures.

### Code Map

- `capos-config/src/ring.rs` - transport error constants, SQE/CQE layout, and endpoint transport flags.
- `schema/capos.capnp` - `ExceptionType`, `CapException`, and per-interface result unions.
- `kernel/src/cap/ring.rs` - exception serialization, ring dispatch, endpoint RETURN exception handling, and `InvalidArgument` sentinel mapping.
- `kernel/src/cap/endpoint.rs` - endpoint queue, in-flight call, and revoked endpoint state.
- `capos-rt/src/client.rs` - runtime decoding into `ClientError`.
- `docs/architecture/capability-ring.md` - ring ABI and opcode dispatch rules.
- `docs/architecture/ipc-endpoints.md` - endpoint CALL/RECV/RETURN transport.

### Validation

- `make run-spawn` covers cross-process endpoint RETURN propagation for `Failed`, `Overloaded`, and `Unimplemented`, plus reserved opcode and no-result-buffer exception paths.
- `make run-smoke` covers same-process endpoint use and revoked-cap behavior.
- `cargo test-lib` covers cap-table stale-slot and transfer rollback behavior that the transport error paths depend on.
- `cargo test-ring-loom` covers ring queue behavior that completion delivery depends on.

### Open Work

- Promise pipelining and future multishot/link/drain ring behavior must carry the same three-layer error split.
- Long-lived services should prefer stable result-union variants over generic text errors for ordinary domain outcomes.
- Future external clients need compatibility rules for exception taxonomy evolution once the ABI is treated as cross-version or separately released.

### Design Grounding

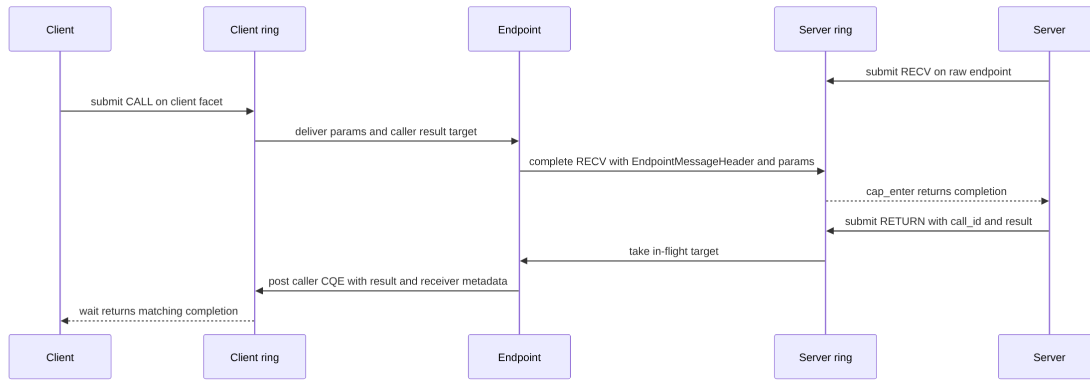
The archival decision record is Proposal: Error Handling for Capability Invocations. Relevant research notes are Cap'n Proto Error Handling: Research Notes and OS Error Handling in Capability Systems: Research Notes.

## IPC and Endpoints

Endpoints let one process serve capability calls to another process without adding a separate IPC syscall surface. The same ring transport carries ordinary kernel capability calls and cross-process endpoint calls.

### Current Behavior

An `Endpoint` is a kernel capability object with queues for pending client calls, pending server receives, and in-flight calls awaiting RETURN. A service that owns the raw endpoint can receive and return. Importers receive a `ClientEndpoint` facet that can CALL but cannot RECV or RETURN.




If a CALL arrives before a RECV, the endpoint queues bounded params. If a RECV arrives before a CALL, the endpoint queues the receive request. Delivered calls move into the in-flight queue until the server returns or cleanup cancels them.

### Design

Endpoint IPC is capability-oriented. The manifest can export a raw endpoint from one service; importers get a narrowed client facet. This keeps server-only authority out of clients without introducing rights bitmasks.

CALL and RETURN may carry sideband transfer descriptors. Copy transfers insert a new cap into the receiver while preserving the sender. Move transfers reserve the sender slot, insert the destination, then remove the source on commit. RETURN-side transfers append result-cap records after the normal result payload. Cross-session delivery is additionally checked against the cap hold transfer scope: same-session caps fail closed, cross-session-shareable caps may cross, and service-regrant-only caps need a trusted fixed-session regrant path. CALL SQEs may also request field-granular session disclosure. The kernel intersects that request with the invoked cap's disclosure scope before delivering any subject fields, so a request without scope or scope without a request exposes only the default opaque caller-session metadata.

Legacy receiver metadata is stored on cap-table hold edges and delivered to servers with endpoint invocation metadata, so one endpoint can distinguish transitional callers without one object per caller. Some ABI structs still name this field `badge`; that name is compatibility state, not the normal shared-service authority model. Session-bound invocation context is the replacement model for normal workload paths: every normal process has one immutable session context,

endpoint calls expose privacy-preserving caller-session metadata by default, and shared services derive user-facing state from broker-granted capabilities plus service-scoped session references. See Session Context.

**Delegated Client Relabeling Containment**

The Gate 0 containment rule is narrow: a process that holds an imported `ClientEndpoint` may delegate that same client identity, but it may not mint a sibling identity by setting another legacy badge during spawn. Endpoint owners and explicit trusted mint paths remain transitional mechanisms for low-level tests. Normal shared services use broker-granted roots/facets plus session-bound invocation context instead of service-object badges.

Normal `capos-shell` help and smoke expectations must therefore omit arbitrary `badge N` launch examples. Omitted shell badge syntax preserves the source identity instead of selecting badge zero. Legacy badge syntax may remain reachable only as a debug or hostile-test input, and QEMU coverage for the Telnet blocker must prove both explicit `client @name badge N` and low-level legacy badge-zero relabel encodings from a nonzero delegated client facet fail closed.

Shell-serviced stdio bridges now bind the active child wait to the first opaque live caller-session reference seen on the bridge endpoint. A later call from a different live caller session is answered with an empty result and the child is terminated; transferred caps are released before either normal transfer rejection or caller-session rejection returns. Normal `StdIO.close` is treated as a clean child close rather than a security rejection.

Future IPC should add notification objects for lightweight signaling and promise pipelining for Cap'n Proto-style dependent calls.

### Invariants

- Only raw endpoint holders may RECV or RETURN.
- Imported endpoint caps are `ClientEndpoint` facets and must reject RECV and RETURN from userspace.
- Delegating an imported client facet must preserve its server-visible object identity. Only endpoint owners or explicit trusted mint paths may create sibling client identities, and normal services should not treat that identity as user/session authority.
- Endpoint queues are bounded by call count, receive count, in-flight count, per-call params, and total queued params.
- Each in-flight call has a kernel-assigned non-zero `call_id`.
- CALL delivery copies params into kernel-owned queued storage before the caller can resume.
- Move transfer commit must not leave both source and destination live.
- Transfer rollback must preserve source authority if destination insertion or result delivery fails.
- Process exit must cancel queued state involving that pid and wake affected peers when possible.

### Code Map

- `kernel/src/cap/endpoint.rs` - endpoint queues, client facet, call IDs, cancellation by pid.

- `kernel/src/cap/ring.rs` - endpoint CALL/RECV/RETURN dispatch, result copying, deferred cancellation CQEs.
- `kernel/src/cap/transfer.rs` - transfer descriptor loading and transaction preparation.
- `capos-lib/src/cap_table.rs` - cap-table transfer primitives and rollback.
- `kernel/src/cap/mod.rs` - manifest export resolution and client-facet construction.
- `capos-config/src/ring.rs` - `EndpointMessageHeader`, transfer descriptors, transfer result records, endpoint opcodes.
- `demos/capos-demo-support/src/lib.rs` - endpoint, IPC, transfer, and hostile IPC smoke routines.
- `demos/endpoint-roundtrip`, `demos/ipc-server`, `demos/ipc-client` - QEMU smoke binaries.
- `demos/ipc-zerocopy-producer`, `demos/ipc-zerocopy-consumer` - QEMU smoke for the multi-message shared-buffer zero-copy IPC pattern.

**Validation**

- `make run-smoke` validates same-process endpoint RECV/RETURN, cross-process IPC, endpoint exit cleanup, legacy badged calls, transfer success/failure paths, and clean halt.
- `make run-spawn` validates init-spawned endpoint-roundtrip, server, and client processes.
- `make run-memoryobject-shared` validates a one-shot shared-buffer handoff over an endpoint cap transfer.
- `make run-ipc-zerocopy` validates the multi-message zero-copy IPC pattern at the substrate level: the producer transfers one `MemoryObject` to the consumer and then exchanges four record payloads through the shared mapping while endpoint CALLs carry only sequence numbers and checksums. The demo drives raw SQE/CQE construction through `capos-demo-support` rather than a typed runtime client and uses an ad-hoc seq+checksum framing because the typed `SharedBuffer` ABI, ring-shaped producer/consumer metadata, and notification primitives are still pending; production services (`File.readBuf`, `BlockDevice.readBlocks`, NIC RX/TX rings) will reuse the same `MemoryObject` substrate through that future surface, not the demo's framing.
- `cargo test-lib` covers cap-table transfer preflight, provisional insertion, commit, rollback, stale generation, and slot exhaustion cases.
- `cargo test-ring-loom` covers ring queue behavior that endpoint IPC depends on for completion delivery.

**Open Work**

- Add notification objects for signal-style events.
- Add Cap'n Proto promise pipelining after endpoint routing can resolve dependent answers.
- Add a typed `SharedBuffer` capability surface (ring-shaped producer/consumer metadata, completion signaling, lifetime/quota rules) on top of the raw `MemoryObject` substrate exercised by `make run-ipc-zerocopy`.
- Add epoch-based revocation if broad authority invalidation becomes necessary.

## Authority Graph and Resource Accounting for Transfer

This document defines the authority graph and resource-accounting contract originally tracked as Security Verification Track S.9 in `docs/proposals/security-and-verification-proposal.md`. It covers:

- capability transfer (`xfer_cap_count`, copy/move, rollback)
- ProcessSpawner prerequisites (spawn quotas and result-cap insertion)

Security Verification Track S.9 is complete when this design contract is concrete enough to guide implementation. The invariants and acceptance criteria below are implementation gates for capability transfer, ProcessSpawner, Security Verification Track S.8, and Security Verification Track S.12 follow-up work, not requirements for declaring the Security Verification Track S.9 design artifact complete. Current capability-semantics follow-up items live in `docs/backlog/stage-6-capability-semantics.md`.

### Current Implementation and Target Contract

The current implementation defines `ResourceLedger` fields in `capos-lib/src/cap_table.rs` for capability slots, outstanding calls, scratch bytes, frame-grant pages, and virtual-reservation pages. Cap-slot and frame/virtual page reservations are wired into current reservation paths. Outstanding-call and scratch-byte counters are present ledger fields but are not yet fully wired into reservation/preflight paths. Endpoint queue quota, diagnostic log-rate accounting, and CPU token-bucket accounting below are target contract fields for future implementation work, not current `ResourceLedger` members.

### 1. Authority Graph Model

Authority is modeled as a directed multigraph:

- Nodes:
  - `Process(Pid)`
  - `Object(ObjectId)` (kernel object identity, independent of per-process `CapId`)
- Edges:
  - `Hold(Pid -> ObjectId)` with metadata:
    - `cap_id` (table-local handle)
    - `interface_id`
    - `badge`
    - `transfer_mode` (`copy`, `move`, `non_transferable`)
    - `origin` (`kernel`, `spawn_grant`, `ipc_transfer`, `result_cap`)

Security invariant A1: all authority is represented by `Hold` edges; no operation can create object authority outside this graph.

Security invariant A2: each process mutates only its own `CapTable` edges except through explicit transfer/spawn transactions validated by the kernel.

Security invariant A3: for every live `Hold` edge there is exactly one `cap_id` slot in one process table referencing the object generation.

## 2. Per-Process Resource Ledger and Quotas

Each process owns a kernel-maintained `ResourceLedger`. For wired reservation paths, enforcement is fail-closed at reservation time (before side effects). The target contract completes enforcement for present-but-unwired fields and extends the ledger with endpoint queue, diagnostic log, and CPU budget counters.

```
ResourceLedger {
  // Current ledger fields.
  cap_slots_used / cap_slots_max
  outstanding_calls_used / outstanding_calls_max
  scratch_bytes_used / scratch_bytes_max
  frame_grant_pages_used / frame_grant_pages_max
  virtual_reservation_pages_used / virtual_reservation_pages_max

  // Target/future fields.
  endpoint_queue_used / endpoint_queue_max
  log_bytes_window_used / log_bytes_per_window (token bucket)
  cpu_time_us_window_used / cpu_budget_us_per_window (token bucket)
}
```

Initial quota profile for Stage 6/5.2 bring-up (tunable by kernel config):

- `cap_slots_max`: 256
- `outstanding_calls_max`: 64
- `scratch_bytes_max`: 256 KiB
- `frame_grant_pages_max`: 4096 pages (16 MiB at 4 KiB pages)
- `virtual_reservation_pages_max`: kernel-configured virtual reservation budget
- Future target fields: `endpoint_queue_max` 128 messages, `log_bytes_per_window` 64 KiB/sec with 256 KiB burst, and `cpu_budget_us_per_window` 10,000 us per 100,000 us window.

Security invariant Q1: no counter may exceed its max.

Security invariant Q2: every resource reservation has a matched release on all success, error, timeout, process-exit, and rollback paths.

Security invariant Q3: quota checks for transfer/spawn happen before mutating sender or receiver capability state.

## 3. Diagnostic Rate Limiting and Aggregation

Repeated invalid ring/cap submissions are aggregated per process and error key.

- Key: `(pid, error_code, opcode, cap_id_bucket)`
- Buckets:
  - `cap_id_bucket = exact cap id` for stale/invalid cap failures
  - `cap_id_bucket = 0` for structural ring errors
- Per-key token bucket: allow first `N=4` emissions/sec, then suppress.
- Suppressed counts are flushed once per second as one summary line:

- `pid=X invalid submissions suppressed=Y last_err=...`

Security invariant D1: invalid submission floods cannot consume unbounded serial bandwidth or scheduler time in log formatting.

Security invariant D2: suppression never hides first-observation diagnostics for a new `(pid,error,opcode,cap bucket)` key.

**4. Transfer and Rollback Semantics**

Transfers (`xfer_cap_count > 0`) use a kernel transfer transaction (`TransferTxn`) scoped to a single SQE dispatch. The current ring ABI does not provide kernel-owned SQE sequence numbers or a durable transaction table, so userspace replay of a copy-transfer SQE is repeatable: each replay is treated as a new copy grant. Move-transfer replay fails closed after the source slot is removed or reserved by the first successful dispatch.

Future exactly-once replay suppression requires transaction identity scoped to `(sender_pid, call_id, sqe_seq)` and a monotonic transfer epoch. Until that exists, exactly-once claims apply only within one dispatch attempt, not across malicious rewrites of shared SQ ring indexes.

Sensitive interfaces must choose their transfer mode deliberately:

- **Transfer mode:** `copy`
  - **Semantics:** Repeatable grant; sender keeps authority and replaying the same copy-transfer SQE can mint another receiver hold.
  - **Suitable for:** Stateless or explicitly shareable caps where duplicate receivers are acceptable and audited.
  - **Required negative tests:** Replay mints only allowed duplicate holds; quota exhaustion fails closed; copy across forbidden session/transfer scope is rejected.
- **Transfer mode:** `move`
  - **Semantics:** Single authority handoff; sender loses the source hold after successful destination insertion. Replay fails closed after source reservation/removal.
  - **Suitable for:** Linear resources, accepted sockets, terminal sessions, one-shot result caps, and authority that should have one active owner.
  - **Required negative tests:** Replay after success fails; rollback restores sender on partial failure; receiver cannot observe authority before commit.
- **Transfer mode:** `non_transferable`
  - **Semantics:** No IPC/spawn transfer.
  - **Suitable for:** Process-local control caps, raw spawn/network/device authority, private keys, and caps whose authority depends on caller-local state.
  - **Required negative tests:** IPC/spawn transfer attempts fail closed and leave sender/receiver tables unchanged.

Copy-transfer replay is therefore acceptable only for caps whose interface contract says repeated receivers are safe. Sensitive caps must be move-only or non-transferable until the interface has an explicit replay threat model and hostile tests.

Phases:

1. `Prepare`:
   - validate SQE transport fields and `xfer_cap_count`
   - validate sender ownership/generation/transferability for each exported cap
   - reserve receiver quota (`cap_slots`, `outstanding_calls`, scratch if needed)
   - pin sender entries in txn state (no sender table mutation yet)
2. `Commit`:
   - insert destination edges exactly once
   - for `copy`: increment object refcount/export ref
   - for `move`: remove sender slot only after destination insertion succeeds
   - publish completion/result
3. `Finalize`:
   - release transient reservations
   - mark txn terminal (`committed` or `aborted`)

On any error before `Commit`, rollback is full:

- receiver inserts are not visible
- sender slots/refcounts unchanged
- reservations released
- CQE returns transfer failure (`CAP_ERR_TRANSFER_ABORTED` / subtype)

On error during `Commit`, kernel executes compensating rollback to preserve exactly-once visibility: either all inserts are visible with matching sender state transition, or none are visible.

Security invariant T1: each transfer descriptor is applied at most once within a single SQE dispatch attempt.

Security invariant T2: move transfer is atomic from observer perspective; no state exists where both sender and receiver lose authority due to partial apply.

Security invariant T3: copy-transfer SQE replay is explicitly repeatable until kernel-owned transaction identity exists. Move-transfer replay fails closed after source removal or source reservation.

Security invariant T4: `CAP_OP_RELEASE` removes one local hold edge only from the caller table and decrements remote export refs exactly once.

### 5. Integration with 3.6 Capability Transfer

3.6 implementation must consume this design directly:

- `CALL` and `RETURN` validate all currently-reserved transfer fields fail-closed when unsupported.
- `xfer_cap_count` path is wired through `TransferTxn` (no ad hoc direct inserts).
- Badge propagation is explicit in transfer descriptors and copied into destination edge metadata.
- `CAP_OP_RELEASE` uses the same authority ledger and refcount bookkeeping.

3.6 acceptance criteria:

1. Copy transfer produces one new receiver edge and retains sender edge.
2. Move transfer produces one new receiver edge and deletes sender edge atomically.
3. Any transfer failure leaves sender and receiver `CapTables` unchanged.
4. Copy replay is an explicit repeatable-grant policy until a kernel-owned transaction identity is added; move replay fails closed after source removal or reservation.
5. `CAP_OP_RELEASE` on stale/non-owned cap fails closed without mutating other process tables.

### 6. Integration with 5.2 ProcessSpawner Prerequisites

5.2 must use the same accounting and transfer machinery:

- `spawn()` preflights child quotas (`cap_slots`, `outstanding_calls`, `scratch`, `frame_grant_pages`, endpoint queue baseline) before mapping child memory or scheduling.
- Parent-provided `CapGrant` entries are inserted via the same transfer transaction semantics (copy for initial grants in 5.2.2).
- Returned `ProcessHandle` is inserted through the standard result-cap insertion path and accounted as a normal cap slot.
- Child setup rollback must unwind:
  - address space mappings
  - ring page
  - CapSet page
  - kernel stack
  - allocated frames
  - provisional capability edges/reservations

5.2 acceptance criteria:

1. Spawn failure at any step leaves no child-visible process and no leaked ledger usage.
2. Successful spawn accounts all child bootstrap resources within quotas.
3. Parent and child cap-table accounting remains balanced under repeated spawn/exit cycles.
4. `ProcessHandle.wait` and exit cleanup release outstanding-call/scratch/frame usage deterministically.

### 7. Implementation Notes for Verification Tracks

This design unblocks:

- Security Verification Track S.8 hostile-input tests for quota and invalid-transfer failures.
- Security Verification Track S.12 Kani bounds refresh for ledger and transfer invariants.
- Target 12 in `docs/proposals/security-and-verification-proposal.md` with explicit allocator hooks and fail-closed exhaustion behavior.

## Userspace Runtime

The userspace runtime owns the repeated mechanics that every service needs: bootstrap validation, heap initialization, typed capability lookup, ring submission, completion matching, application exception decoding, and handle lifetime.

### Related

- Go VirtualMemory Contract defines the caller-buffer reserve, commit, and decommit methods allocator paths need.
- Programming Languages summarizes current native Rust support and planned language-runtime tracks.
- Memory Management documents the implemented kernel `VirtualMemory` and `MemoryObject` behavior.
- Go Runtime is the owning language runtime proposal; LLVM Target records the Go runtime OS hooks that drive this work.

### Current Behavior

Runtime-owned `_start` receives `(ring_addr, pid, capset_addr)`, initializes a fixed heap, validates the ring address, reads the read-only CapSet page, installs an emergency Console panic path when available, calls `capos_rt_main(runtime)`, and exits with the returned code.

The `Runtime` lends out at most one `RuntimeRingClient` at a time. The client wraps the raw ring page, keeps request buffers alive until completions are matched, handles out-of-order completions, packs copy-transfer descriptors, and parses result-cap records. Owned runtime handles queue `CAP_OP_RELEASE` when the last local reference is dropped; the release queue flushes when a ring client is borrowed or dropped, or when code calls `Runtime::flush_releases()` explicitly. Promise placeholders are currently bookkeeping only; their future SQE coordinates map `AnswerId.raw()` to `pipeline_dep` and a result-cap record index to `pipeline_field`.

### Design

The runtime separates non-owning bootstrap references from owned local handles. CapSet entries produce typed `Capability<T>` values only when the interface ID matches the requested type, and the same manifest-order CapSet entries remain available for diagnostic and shell surfaces that need to list or inspect what a process was actually granted. Result-cap adoption performs the same interface check before producing `OwnedCapability<T>`.

Typed clients are thin wrappers over the ring client. They encode Cap'n Proto params, submit CALL SQEs, wait for a matching CQE, decode transport errors, and decode kernel-produced `CapException` payloads into client errors. Endpoint servers can use `submit_endpoint_return_exception()` to return a serialized `CapException` to the original caller over the same endpoint RETURN path. The handwritten `TimerClient` exposes monotonic `now` reads and sleep calls over the same completion-matching path. The handwritten `VirtualMemoryClient` exposes map, reserve, commit, decommit, unmap, and protect calls for runtime heap/arena allocation over anonymous user pages. It has both the ordinary allocation-backed async methods and synchronous caller-buffer methods for allocator growth paths that

cannot allocate while asking the kernel for more memory. This matches the reserve/commit/decommit surface specified in Go VirtualMemory Contract. The handwritten `ThreadControlClient` exposes current-process FS-base reads and updates for runtimes that need to swap a language-managed TLS base after process startup.

The 7.1.0 threading contract keeps one process ring and the runtime's single-owner ring-client invariant for the first in-process threading implementation. Future multi-threaded runtimes must serialize blocking ring entry through `capos-rt` until a runtime reactor or Ring v2 lands. The reactor bridge uses one runtime-owned CQ drainer plus ParkSpace-backed wait records; the full-SMP kernel target is per-thread rings, where `cap_enter` waits on the current thread's CQ. After 7.2, the existing `ThreadControlClient` methods apply to the current thread's FS base rather than to a process-wide saved FS base. `ThreadControl.exitThread` and the raw `exit(code)` syscall both terminate the current thread; the process exits when its last live thread exits.

The 7.2.3 park slice adds a process-local ParkSpace marker type and compact `CAP_OP_PARK` / `CAP_OP_UNPARK` operations. `capos-rt` should expose those operations as runtime synchronization primitives in a later slice; the current `thread-lifecycle` proof uses raw SQEs so the runtime does not prematurely claim the park `user_data` namespace. Blocking park wait is not an ordinary `RuntimeRingClient` call: the wait SQE must be thread-owned for the current thread, and the runtime must reserve park `user_data` values, write the wait SQE under its ring-submission lock, release that lock before `cap_enter`, and demultiplex park CQEs into runtime-owned wait slots so a sibling thread can still submit the wake. The temporary single-thread park fallback remains only as the pre-thread runtime checkpoint proof.

Future generated clients should preserve this split: transport lifetime and completion matching belong in the runtime, while interface-specific encoding belongs in generated or handwritten client wrappers.

### Invariants

- `ring_addr` must equal `RING_VADDR`; runtime bootstrap rejects any other address.
- The CapSet header magic/version must validate before lookup.
- CapSet handles are non-owning unless explicitly adopted.
- Only one runtime ring client may be live at a time for a process.
- Until Ring v2, multithreaded generic client waits must flow through a runtime reactor/demux path rather than letting multiple threads consume the process CQ directly.
- Park wait must not hold the live runtime ring client while the kernel parks the current thread.
- Request params and result buffers must outlive their matching CQE.
- A result cap can be consumed only once and only with the expected interface ID.
- Promise placeholders must map to sideband result-cap record indexes, not schema field paths.
- Dropping the final owned handle queues exactly one local `CAP_OP_RELEASE`; `Runtime::flush_releases()` forces queued releases and reports rejected kernel release results.
- Release flushing treats stale or already-removed caps as non-fatal cleanup.

### Code Map

- capos-rt/src/entry.rs - _start, Runtime, bootstrap validation, single-owner ring token, release queue flushing.
- capos-rt/src/alloc.rs - fixed userspace heap initialization.
- capos-rt/src/capset.rs - typed CapSet lookup and manifest-order iteration wrappers.
- capos-rt/src/ring.rs - ring client, pending calls, completion matching, copy-transfer packing, result-cap parsing.
- capos-rt/src/client.rs - Console, TerminalSession, BootPackage, ProcessSpawner, ProcessHandle, VirtualMemory, Timer, ThreadControl, ThreadSpawner, and ThreadHandle clients, and exception decoding.
- capos-rt/src/lib.rs - typed capability marker types and owned handle reference counting.
- capos-rt/src/panic.rs - emergency Console output path.
- capos-rt/src/syscall.rs - raw syscall instructions and public syscall wrappers, including the hostile smoke probe for the removed ambient write syscall.
- targets/x86_64-unknown-capos.json - userspace target specification.
- tools/check-userspace-runtime-surface.sh - source check that keeps runtime primitives owned by capos-rt.
- init/src/main.rs, capos-rt/src/bin/smoke.rs, and shell/src/main.rs - current runtime users.

### Validation

- make capos-rt-check builds the runtime smoke binary against targets/x86_64-unknown-capos.json, matching the booted userspace target.
- make init-capos-build, make demos-capos-build, make shell-capos-build, and make capos-rt-capos-build expose focused custom-target build wrappers for the current userspace crates and runtime smoke binary.
- tools/check-userspace-runtime-surface.sh verifies init, demos, and shell do not define _start, panic handlers, global allocators, raw syscall instructions, or entry-point macros outside capos-rt.
- make run-smoke validates runtime entry, typed Console calls, exception decoding, owned handle release, result-cap parsing through IPC, and clean process exit.
- make run-spawn validates ProcessSpawnerClient, ProcessHandleClient, VirtualMemoryClient, TimerClient, ThreadControlClient, ThreadSpawnerClient, ThreadHandleClient, result-cap adoption, and release behavior under init spawning. The single-thread-runtime child proves the first runtime-shaped checkpoint over caller-buffer VirtualMemory calls and Timer; the thread-lifecycle child proves in-process create, self-join rejection, join, detach, last-thread exitThread, and private ParkSpace wait/wake correctness.
- make run-shell validates CapSet iteration, capability inspection, typed application-error decoding, guest session metadata, exact-grant spawning, ProcessHandle waits, and stale-handle release behavior in the focused shell-launch proof manifest.
- make run-terminal validates TerminalSessionClient writes, bounded line reads, hidden-echo input handling, and structured cancellation in the focused terminal proof manifest.

- `cd capos-rt && cargo test --lib --target x86_64-unknown-linux-gnu` covers host-testable runtime invariants when run explicitly.

**Open Work**

- Add generated client bindings after the schema surface stabilizes.
- Implement promise/answer transport semantics beyond current placeholders.
- Add typed ParkSpace clients with runtime-owned `user_data` demultiplexing.
- Define release behavior for queued handles when a process exits before the release queue flushes.

## Memory Management

Memory management gives the kernel controlled ownership of physical frames, separates user processes, enforces page permissions, and exposes memory authority only through explicit capabilities.

### Related

- Go VirtualMemory Contract records the reserve/commit/decommit contract that extended the `VirtualMemory` implementation for Go-style arena allocation.
- Userspace Runtime describes the typed `VirtualMemoryClient` surface used by allocator and runtime code.
- OOM Handling and Swap and Resource Accounting and Quotas define future memory-pressure and quota policy.
- Memory Authority Model defines the future cross-cutting contract for memory authority classes, residency, mapping consistency, pins, DMA boundaries, swap eligibility, and proof obligations.

### Current Behavior

The frame allocator builds a bitmap from the Limine memory map, marks all non-usable frames as used, reserves frame zero, and reserves its own bitmap frames. The heap is initialized separately for kernel allocation.

Paging initialization builds a new kernel PML4, remaps kernel sections with section-specific permissions, copies upper-half mappings with NX applied and user access stripped, switches CR3, then enables page-global support. SMEP/SMAP are enabled after those mappings are active.

Each user `AddressSpace` owns its lower-half page tables and clones the kernel's upper-half mappings. Dropping an address space walks the user half and frees mapped frames, committed anonymous frames retained behind `VM_PROT_NONE`, and page-table frames. `VirtualMemory` lets a process reserve anonymous address ranges, commit and decommit physical backing, unmap reservations, and protect committed pages. Anonymous reservations charge the process virtual reservation ledger. Committed anonymous pages charge `ResourceLedger::frame_grant_pages`.

`FrameAllocator` allocation methods return a `MemoryObject` result capability, not a physical address. The normal result payload carries the result-cap index, and the CQE transfer-result record carries the local cap id plus `MemoryObject` interface id. `MemoryObject.info` exposes page count and size; `MemoryObject.map` maps page-aligned object ranges into the caller address space, `MemoryObject.unmap` removes those borrowed mappings, and `MemoryObject.protect` updates their page-table flags. Held `MemoryObject` caps charge the holder's `frame_grant_pages` ledger, and final `CAP_OP_RELEASE` or process exit frees the owned frames once no borrowed address-space mapping still holds the backing alive.

### Design

The kernel keeps physical allocation host-testable by placing bitmap logic in `capos-lib` and wrapping it with kernel HHDM access in `kernel/src/mem/frame.rs`. Page-table manipulation stays in the kernel because it is architecture-specific.

ELF loading and VirtualMemory both use page-table flags to preserve W^X: non-executable data gets NX, writable mappings are explicit, and userspace pages must be USER_ACCESSIBLE. The CapSet and ring bootstrap pages occupy reserved virtual pages; VirtualMemory rejects ranges that overlap either one.

User-buffer validation for process-owned buffers uses the process AddressSpace mutex. The kernel checks that user pointers stay below the user address limit, verifies page-table permissions for the requested read/write access, and copies through the HHDM mapping while holding the same address-space lock. This keeps validation and use tied to one stable page-table view. The legacy current-CR3 validator remains only for callers that already provide an equivalent page-table stability guarantee.

Committed VirtualMemory pages and held MemoryObject caps use the same per-process frame-grant ledger, with quota checks before frame allocation or mapping side effects. Anonymous reservation consumes a separate virtual page quota, so guard ranges and Go-style sysReserve arenas do not spend physical commit budget. Held MemoryObject caps charge for the backing they keep reachable, and each live borrowed MemoryObject mapping reserves frame-grant pages until it is unmapped. This prevents a process from mapping an object, releasing the cap to drop the cap-slot charge, and keeping the backing pinned without quota. The address space records borrowed pages separately from sparse anonymous reservations so teardown and unmap can distinguish anonymous pages from object-backed pages. Future file/network/DMA resources should reuse that authority ledger instead of adding one-off counters per cap.

### Invariants

- Frame addresses are 4 KiB aligned.
- The frame bitmap's own frames are never returned as free frames.
- Upper-half kernel mappings are not user-accessible.
- Kernel text is RX, rodata is read-only NX, and data/bss are RW NX.
- User address spaces own only lower-half page-table frames.
- Process frame-grant usage covers committed anonymous VM pages, held MemoryObject caps, and live borrowed MemoryObject mappings.
- Process virtual-reservation usage covers reserved anonymous VM pages whether or not they are committed.
- Committed VM_PROT_NONE pages retain their frames and data while exposing no present user PTE; reserved uncommitted pages consume no frame-grant quota.
- Object-backed user mappings are tracked as borrowed pages and hold the MemoryObject backing alive until unmapped or address-space teardown.
- MemoryObject unmap/protect only succeeds for borrowed pages backed by the same object.
- VirtualMemory caps are bound to one address space and are not valid cross-process service exports.
- CapSet is read-only/no-execute; ring is writable/no-execute.
- VirtualMemory cannot reserve, map, commit, decommit, unmap, or protect the ring or CapSet pages.

- `VirtualMemory` commit/decommit/protect/unmap only succeeds for ranges covered by anonymous reservations owned by the cap's address space.
- Capability-ring CALL/RECV/RETURN buffers, transfer descriptors, process and thread wait completions, and private ParkSpace word reads must validate and copy/read while holding the target process `AddressSpace` lock.

**Code Map**

- `capos-lib/src/frame_bitmap.rs` - host-testable physical frame bitmap core.
- `capos-lib/src/cap_table.rs` - capability holds and per-process `ResourceLedger` frame-grant accounting.
- `capos-lib/src/frame_ledger.rs` - bounded frame-grant helper retained for host tests.
- `kernel/src/mem/frame.rs` - Limine memory-map integration and global frame allocator wrapper.
- `kernel/src/mem/heap.rs` - kernel heap setup.
- `kernel/src/mem/paging.rs` - kernel remap, `AddressSpace`, page mapping, VM-cap page tracking, user copy helpers.
- `kernel/src/mem/validate.rs` - user-address bounds and legacy current-CR3 validation helper.
- `kernel/src/cap/frame_alloc.rs` - FrameAllocator capability and cleanup.
- `demos/memoryobject-shared-parent/` and `demos/memoryobject-shared-child/` - QEMU shared MemoryObject smoke.
- `tools/qemu-memoryobject-shared-smoke.sh` - transcript checks for the shared MemoryObject smoke.
- `kernel/src/cap/virtual_memory.rs` - VirtualMemory capability.
- `kernel/src/spawn.rs` - ELF, stack, and TLS user mappings.
- `kernel/src/arch/x86_64/smap.rs` - SMEP/SMAP setup and legacy direct user access guard.

**Validation**

- `cargo test-lib` covers frame bitmap, frame ledger, ELF parser, and cap-table pure logic.
- `cargo miri-lib` runs host-testable `capos-lib` tests under Miri when installed.
- `make kani-lib` proves the bounded mandatory frame-bitmap, stale-handle, cap-slot/frame-grant accounting, and transfer preflight fail-closed invariants when Kani is installed.
- `make run-smoke` validates ELF mapping, process teardown, TLS, and clean shell-led halt.
- `make run-spawn` validates MemoryObject-backed FrameAllocator cleanup, VirtualMemory reserve/commit/decommit/`VM_PROT_NONE`/quota/release smoke, and runtime spawn checks.
- `make run-memoryobject-shared` validates a parent allocating and mapping a MemoryObject, transferring it to a child, observing a child write through the same backing pages, unmapping both sides, and halting cleanly.
- `make run-ipc-zerocopy` validates the multi-message shared point-to-point buffer pattern at the substrate level: a producer transfers one `MemoryObject` to the consumer and then exchanges four record payloads through the shared mapping while endpoint CALLs carry only sequence numbers and checksums. This is a substrate proof, not the production data-plane

shape: typed `SharedBuffer` with explicit producer/consumer ring metadata, notification primitives, and consuming service APIs (`File.readBuf`, `BlockDevice.readBlocks`, NIC RX/TX rings) are tracked under Open Work.
- `make run-spawn` validates ELF load failure rollback and frame exhaustion handling through `ProcessSpawner`.

**Open Work**

- Extend frame-grant accounting only if future DMA pinning or service-owned shared-buffer pools need authority beyond held MemoryObject caps and live borrowed mappings.
- Define page-pinning or mapping-identity rules for future shared WaitSet, DMA, and service-owned shared-buffer paths that must keep physical backing stable beyond a single locked copy/read. The owning planning track is Memory Authority Model.
- Add file, block, network, and DMA service APIs that use MemoryObject-backed SharedBuffer caps for zero-copy data paths.
- Add DMA isolation and device memory capability boundaries before userspace drivers.
- Add huge-page handling only with explicit ownership and teardown rules.

## Scheduling

Scheduling decides which thread runs, preserves CPU state across preemption and blocking, and integrates capability-ring progress with process-owned execution resources.

### Current Behavior

The scheduler stores shared process/thread metadata in `Scheduler::processes: BTreeMap<Pid, Process>`. Dispatch-owned runnable state lives in `SchedulerDispatch`: a per-CPU `run_queues: [VecDeque<ThreadRef>; SCHEDULER_CPUS]` array ordered ascending by `Thread.virtual_finish_ns`, per-CPU `current` and `handoff_current` slots, idle-thread slots, the direct-IPC target preference, run-queue reservation accounting, and deferred drop/stack release slots. Each live thread has at most one queued owner across all per-CPU queues combined, and every per-CPU queue reserves capacity up to the live runnable-capable thread count before a new thread is published as runnable, so later timer, unblock, requeue, and steal-requeue paths do not allocate. The shared live-reservation count is released when processes or threads exit or when pre-publication reservation is rolled back. Reserving each queue to the full live-thread count is required because the bounded steal path may migrate every live thread into a single sibling queue between two scheduler passes.

Phase D accepted its Task 6 diagnostic closeout at commit `77caafc0` (`2026-05-10 19:39 UTC`, `docs(scheduler): record phase d thread-scale gate`) and closed in docs commit `1a08ec23` (`2026-05-10 21:47 UTC`, `docs(scheduler): close phase d`). The accepted state is the WFQ scheduler described here: per-thread weights and latency classes are mutated only through `SchedulingPolicyCap`, each per-CPU runnable queue is ordered by freshly derived `virtual_finish_ns`, migration preserves `virtual_runtime_ns`, and bounded stealing selects the most-overdue runnable sibling candidate. The controlled Task 6 benchmark pair on `capos-bench` recorded capOS 1-to-4 work/total speedups `3.088x` / `2.700x` versus the previous single-global-queue baseline `1.566x` / `1.538x`; the matching Linux pthread baseline on the same host and physical-core logical CPUs `0,1,2,3` recorded `3.974x` / `3.850x`. The host harness enforced the configured 1-to-2 work/total gates; the 1-to-4 row was manually accepted from recorded diagnostics. Phase E `SchedulingContext` is the next scheduler authority phase; EEVDF is a follow-on ordering-policy evaluation rather than a Phase D blocker.

Phase D Task 3 (2026-05-07) restored the per-CPU runnable queues that the 2026-05-02 collapse retired and gave them the WFQ ordering Task 2's `virtual_finish_ns` was prepared for. Newly created processes and threads publish onto the creating scheduler CPU's per-CPU queue; the bounded steal path balances the queues when other CPUs run out of local work. The publish-time placement is intentionally simple in this slice — "place locally, let steal balance" — and a more sophisticated caller-aware spread or least-loaded scan is a milestone-gate follow-up, not a Task 3 acceptance requirement. Wake policy carries `WakePolicy::QueueCpu(u32)` for endpoint, timer, park, process-wait, thread-join, and process-spawn completions so the wake target matches the queue placement, and `DirectTarget` keeps its original direct-IPC handoff role. The transitional `CAPOS_SCHED_DISABLE_WFQ=1` / `WakePolicy::QueueAny` fallback has been removed before Phase E `SchedulingContext` schema work.

`wake_idle_scheduler_cpus_locked` first probes the placement target when the policy is `QueueCpu`, then walks eligible idle scheduler CPUs and wakes the first that accepts a fresh

reschedule IPI, skipping CPUs that already have a pending IPI so a burst of ready work cross-wakes more than one neighbor instead of stranding the rest behind one already-targeted CPU.

**Ring SQ Consumer Ownership**

Each ring endpoint has kernel-owned SQ-consumer metadata outside the writable userspace ring page. `cap_enter` and the bounded timer-side current-thread ring service both acquire a syscall-mode owner lease before calling `process_ring()`. The lease carries a nonzero generation and owner identity; `process_ring()` verifies that generation before flushing deferred ring work or advancing SQ head, and stale owners return `StaleSqConsumer` without consuming the head SQE. Duplicate owners fail closed as a retryable busy `cap_enter` status.

CQ publication remains independent of SQ ownership. Already accepted completions stay visible through CQ head/tail even after the SQ owner releases, and thread/process teardown releases any live SQ owner before ring unmapping or record drop without clearing accepted CQEs.

**Bounded SQPOLL ring mode**

Phase F adds a bounded SQPOLL mode for the caller thread's ring through `CpuIsolationLease` with `allowedMode = kernelSqpoll` and `namedRing = callerThread`. The transition is explicit: syscall-owned dispatch may request SQPOLL start while it still owns the SQ, then releases its generation-checked owner; the poller finalizes into `SqpollRunning`, may publish `NEED_WAKEUP` and enter `SqpollSleeping`, wakes back to running when a producer publishes a new SQ tail, and stops or rolls back on lease revoke, cap release, teardown, or failed start. Timer-side syscall-mode ring service fails closed while SQPOLL owns the same endpoint, so no second SQ consumer can advance the SQ head.

The Phase F poller runs from the periodic scheduler service path and from a bounded current-thread syscall service entry used for SQPOLL producer wakes and explicit syscall kicks. Both entries borrow the SQPOLL owner lease rather than acquiring syscall SQ ownership. The current default admits two SQEs per selected SQPOLL worker, and a worker is not reselected again in the same periodic service pass or syscall service entry. Poller elapsed time is charged to the admitted scheduler ledger or scheduling-context target. The wake/sleep protocol uses a shared ring flag: the poller publishes `NEED_WAKEUP`, performs a full ordering barrier, and rechecks SQ tail before sleeping; producers publish initialized SQEs, store SQ tail with a barrier, and enter the kernel if `NEED_WAKEUP` is visible. A `cap_enter` producer wake that finds SQPOLL already owns SQ head can run one bounded SQPOLL batch, return visible CQ availability when the requested threshold is satisfied, preserve ordinary blocked-current-thread and thread-owned-head results, and otherwise fail closed as a retryable busy result. Stale owner generations fail before deferred ring work or SQE start. If teardown requests stop after a live owner has already accepted a SQE, the poller still publishes SQ head for that accepted SQE before releasing ownership, preserving accepted CQEs without leaving work replayable by syscall mode. The focused `make run-scheduler-generic-sqpoll-nohz` proof admits this explicit ring-coupled shape into SQPOLL nohz, drives producer wake and bounded service progress without depending on a periodic tick, then rolls back on stale owner/lease revoke. Policy-service automatic nohz, broader userspace-poller/device-queue admission, and production realtime admission remain future work.

**Per-CPU run queue ordering structure**

Each per-CPU `VecDeque<ThreadRef>` is kept ordered ascending by `Thread.virtual_finish_ns`. Enqueue performs an ordered insert via a linear scan from the front; selection scans the queue by index for the first destination-Runnable entry (via `pop_first_runnable_local_locked`), removes Drop entries it walks past, and leaves RetryLater entries undisturbed for the next scheduler pass. Because the queue is ordered ascending, the first Runnable hit is also the lowest-`virtual_finish_ns` candidate the destination CPU can accept (the most overdue against fair share that this CPU is allowed to run). Linear-scan insert is O(n) per enqueue; with `SCHEDULER_CPUS = 4` and bounded thread counts in this slice the constant is small enough to defer a smarter structure (sorted bucket arrays, intrusive trees) until benchmark evidence shows it dominates scheduler-lock hold time. Promoting to a smarter structure is a follow-up under this plan if the Task 6 milestone gate proves the need.

`virtual_finish_ns` is recomputed on every enqueue from the thread's current `virtual_runtime_ns`, `weight`, and `latency_class`; it is never carried as committed state across blocking, and migrations between per-CPU queues recompute it at the destination so the destination's view of fair-share progress applies. The derivation rule per latency class is documented in `capos-abi/src/scheduler.rs` and the "Latency-class semantics for Phase D" section of `docs/proposals/scheduler-evolution-proposal.md`.

**Bounded steal path**

When a CPU's local queue has no immediately runnable entry the scheduler walks sibling per-CPU queues. For each sibling queue the scan walks indices ascending and selects that queue's first entry that the destination CPU considers `Runnable`; because each queue is ordered ascending by `virtual_finish_ns`, the first Runnable hit is also the lowest `virtual_finish_ns` candidate available to the destination on that source queue. The steal then picks the source queue whose first-Runnable candidate has the **lowest** `virtual_finish_ns` overall, with ties broken by lower CPU id. The chosen entry is removed from its current position in the source queue (not necessarily the head: a RetryLater or single-CPU-owner thread may sit at the source's front and stay there), the WFQ tag is recomputed at the destination, and the entry is inserted at the destination's ordered position. The destination queue is reserved to the full live-thread count, so the steal-requeue is allocation-free. The scan walks at most `SCHEDULER_CPUS * max_queue_len` entries, but in practice each sibling scan stops at the first Runnable candidate per queue.

**RetryLater semantics in the local scan**

The local pop scan walks the per-CPU queue by index instead of popping the front and re-pushing RetryLater candidates. Re-pushing a RetryLater entry whose `virtual_finish_ns` has not changed would ordered-insert it back at the same head position, so a naive pop-then-requeue loop would re-pop the same RetryLater head every iteration and starve runnable entries behind it. The index scan removes Drop entries in place, leaves RetryLater entries undisturbed for the next scheduler pass to re-evaluate, and returns the first Runnable candidate it finds. The bounded steal path uses the same index scan on the destination queue after a steal so a stolen RetryLater entry does not get re-popped in the same dispatch pass.

**Phase E preflight fallback cleanup**
The one-bisect-cycle `CAPOS_SCHED_DISABLE_WFQ=1` opt-out has been removed. Enqueues always target the selected per-CPU WFQ queue, and wake-up sites always carry `WakePolicy::QueueCpu(slot)` for queued work. Phase E `SchedulingContext` work therefore starts from the accepted Phase D WFQ behavior rather than from a source-level single-global-queue fallback.

**Phase E Task 1: scheduling-context object shape**
The first `SchedulingContext` slice is info-only: schema, config, runtime, and kernel code expose `SchedulingContext.info()` and a bootstrap grant shape, but no dispatcher enforcement, replenishment, donation/return, depletion notification, realtime island, SQPOLL, or nohz behavior. `SchedulingContextSpec.cpuMask` uses the canonical little-endian bitset defined in `schema/capos.capnp`: CPU `n` maps to bit `n % 8` of byte `n / 8`, with bit `0` as the least-significant bit of that byte. Empty data means no CPUs are selected rather than all CPUs. Producers omit trailing zero bytes, so the all-zero set's canonical form is empty and any non-empty canonical mask ends with a nonzero byte.

**Phase E Task 2: bind, revoke, and generation identity**
The second `SchedulingContext` slice adds the first bounded authority lifecycle. `SchedulingContext.create()` creates a same-interface result cap for a validated spec, `bindCallerThread()` records one caller-thread binding for the current context generation, and `revoke()` advances the generation and clears the matching thread metadata binding. Bootstrap-granted contexts and contexts returned by `create()` use the same non-wrapping context-id allocator; the binding identity remains `(contextId, generation)`, but distinct cap objects no longer share bootstrap ids. Stale caps report `staleGeneration` and cannot create, bind, or revoke scheduler metadata for a new generation; already-revoked contexts report `revoked`. Release cleanup clears only a thread metadata binding that matches the released cap identity.

**Phase E: SchedulingContext budget enforcement**
`make run-scheduling-context` is the focused Phase E QEMU proof. It starts one process with two independently granted bootstrap contexts, verifies their identities cannot alias, adopts a created result cap, drives bind/revoke and stale-generation calls, confirms release cleanup by rebinding after the released cap drops, and now checks the first dispatcher budget behavior. `bindCallerThread()` installs a fixed budget ledger in the caller thread's scheduler metadata. Runtime charge decrements that ledger at the same scheduler-lock-contained points that update per-thread runtime/vruntime. Runnable selection replenishes elapsed periods and treats exhausted bound contexts as `RetryLater` until their next period, leaving the queued owner in place rather than allocating or moving emergency-path state. Stale or revoked contexts still fail closed before mutating scheduler metadata or accounting.

The current enforcement granularity is the existing periodic scheduler tick: a running thread may overshoot its budget by the current tick quantum before the next dispatch charge throttles it. The smoke therefore proves bounded dispatcher behavior, not nohz/SQPOLL activation or hard realtime admission. It prints `dispatch_effect=budgetEnforced`, visible budget charge, replenishment to full budget after a period, and a throttled wall-clock window.

**Phase F: CpuIsolationLease and automatic nohz activation**

`CpuIsolationLease` is a separate authority surface from `SchedulingContext` CPU-time budget enforcement. The scaffold records owner identity, allowed CPU set, allowed isolation mode, live accounting target reference, housekeeping exclusions, maximum revocation latency, and generation identity. It rejects stale generations, duplicate or overlapping active leases, fabricated or stale `SchedulingContext` accounting targets, malformed CPU masks, and lease sets that would leave no online scheduler housekeeping CPU outside the globally admitted active lease CPUs.

The scheduler-side preflight reports a bounded nohz activation/deactivation decision surface: lease identity, target CPU mask, target runnable entity count, active housekeeping CPU availability after subtracting all active lease CPUs, selected housekeeping CPU mask, deferred cleanup, timer/deadline, network polling, IRQ-affinity, accounting-target, monotonic clocksource/ accounting readiness, one-SQ-consumer, revocation latency, rollback, and periodic-fallback labels. The accepted QEMU proof uses `-smp 4` so an active lease can report ready housekeeping CPUs outside the target CPU, selected housekeeping placement, and exactly one runnable caller on that target CPU.

The clockevent/deadline substrate uses a calibrated TSC-backed monotonic clocksource on normal QEMU/x86_64, with the periodic LAPIC tick disciplining the TSC epoch so QEMU guest halt windows cannot stall wall-clock progress. `Timer.sleep`, finite `cap_enter`, and park timeouts store absolute monotonic `deadline_ns` values, and the LAPIC clockevent backend can program a bounded one-shot deadline and restore periodic mode.

**Automatic nohz activation state machine**

When the preflight finds every proof obligation satisfied – a single runnable entity on the target CPU, a ready housekeeping CPU outside the lease, no local deferred-cleanup/timer dependency, a valid accounting target, a live monotonic clocksource, a non-stale one-SQ-consumer when a ring is named, a bounded revocation latency, and the lease's `allowedCpuMask` naming exactly one scheduler-owned CPU – it performs **real per-CPU periodic-tick suppression** for that narrow single-runnable window. The target CPU may be the CPU running the preflight call (local activation) or a different scheduler CPU (remote-CPU activation via a reschedule IPI – see *Remote-CPU activation* below). The single-runnable shape differs by target: a local activation requires the caller itself to be that single entity (`exactly-one-runnable-caller`); a remote activation requires the target CPU's single runnable entity to be some thread pinned there, not the caller (which runs on a different CPU – `exactly-one-runnable-remote-target`).

- **Admission gates.** Two lease shapes can be admitted for tick suppression: a pure `namedRing = none` compute lease, and a ring-coupled `allowedMode = kernelSqpoll` lease whose bound ring is being actively driven by a live SQPOLL consumer.
  - *Compute lease (`namedRing = none`).* Declares no local network/IRQ dependency, so the read-only network-polling and IRQ-affinity admission gates pass.
  - *Ring-coupled SQPOLL lease (`allowedMode = kernelSqpoll`, `namedRing = callerThread`).* The lease's declared kernel-polled work IS the bounded SQPOLL ring poller, which the scheduler keeps progressing through `cap_enter`/producer-wake even while the periodic tick is masked. The preflight admits it only when the bound ring is in SQPOLL running/sleeping mode with a non-stale `Sqpoll` owner; the one-SQ-consumer label is then `blocked-sqpoll-owner` (the worker owns the ring). The preflight ring-state read is a **best-effort hint** – it

never takes the per-ring lock inside the scheduler lock (it uses `try_lock`, and a contended snapshot does not admit activation). The decisive disqualifier is the IPI/timer re-check below.
  - A `namedRing = callerThread` lease that is *not* `kernelSqpoll` (compute-with-ring) keeps the conservative refusal until network polling and IRQ affinity are routed to a housekeeping CPU, as does any device-owning mode. The kernel still services virtio RX/TX and `Interrupt` waiters inline from the periodic scheduler path.
- **Activate.** The preflight masks the periodic LAPIC timer on the current CPU and arms a one-shot deadline at `min(nearest pending timer wakeup, now + max revocation latency)`. The CPU now runs on a bounded one-shot deadline instead of the periodic tick. The eligible lease generation is registered so revoke/cleanup paths can stale it.
- **Re-check.** On every timer interrupt and on every reschedule IPI the handler re-checks the activation window before the scheduler picks the next thread. The reschedule-IPI handler also drains any pending remote-CPU activation request parked for this CPU (the IPI vector is shared with the remote-activation path – see *Remote-CPU activation* below), and the periodic timer handler drains it too as a backstop. An unchanged eligible window re-arms the bounded one-shot deadline; a reschedule IPI (the prompt signal that another CPU woke runnable work onto this CPU) drives an immediate rollback. The re-check runs in interrupt context and uses `try_lock` to avoid deadlocking against a held scheduler lock. **Armed-timer invariant:** the masked-periodic one-shot does not auto-rearm, so a timer-interrupt re-check NEVER returns leaving a tickless CPU without an armed timer – on scheduler-lock contention it arms a bounded minimum-delta fallback one-shot (or restores the periodic tick) before returning. A lock-free per-CPU `nohz-active` bitmask lets the contention path distinguish a tickless CPU (the consumed timer was the nohz one-shot and must be replaced) from a normal CPU (the periodic tick auto-rearms). A reschedule IPI does not consume the one-shot, so its contention skip is safe – the still-armed one-shot bounds the next re-check.
- **Rollback.** Any disqualifying change rolls the CPU back to the periodic LAPIC tick *first*, before any further ordinary work: a stale lease generation (explicit revoke, process exit, service replacement, session logout), a second runnable entity or stealable sibling work on the target CPU, a local deferred-cleanup dependency, a direct-IPC target becoming runnable, a target-CPU mismatch, or a one-shot backend that can no longer arm a deadline. For a ring-coupled SQPOLL activation the re-check also carries a `sqpoll-ring-mode-changed-or-owner-staled` disqualifier (the bound ring leaving SQPOLL running/sleeping mode or its owner staling); that re-check runs under the scheduler lock and uses `try_lock` on the per-ring lock, so a contended ring is treated as disqualifying (fail-closed – restore the periodic tick rather than keep a CPU tickless on an unverifiable ring). That SQPOLL ring-mode branch is **defense-in-depth, currently subsumed by lease-generation staling**: every reachable SQPOLL-stop path today (`stop_sqpoll_for_lease` / `stop_sqpoll_if_owned`) is a revoke/cleanup-path caller that also stales the lease, and `stale-lease-generation` is checked first – so the lease-generation stale is the load-bearing SQPOLL rollback trigger in practice. The SQPOLL ring-mode branch becomes independently load-bearing, and would then need its own proof, only if a future change introduces a SQPOLL-stop path that keeps the lease live. Runtime accounting stays boundary/counter driven and monotonic, so suppressing the tick never strands `SchedulingContext` budget charging.

**Remote-CPU activation**

Masking the periodic LAPIC tick and arming the one-shot deadline are per-CPU operations – only the target CPU can program its own LAPIC timer. When the preflight runs on CPU A but the lease's single-CPU `allowedCpuMask` targets a different CPU B, the kernel does **not** refuse: it parks a bounded remote-activation request in CPU B's per-CPU slot and sends a reschedule-style IPI to CPU B. CPU B drains the request from its IPI handler (and from its periodic timer handler as a backstop), re-runs the full disqualification check **locally** under its own scheduler-lock acquisition, and only then arms its own one-shot deadline. A remote activation is never trusted blind – the preflight's eligibility snapshot was taken on a different CPU and may be stale by the time the IPI is drained, so the target CPU re-checks before committing. The relevant invariants:

- **Bounded request slot, no nesting.** The pending-request store is a fixed `[Option<_>; SCHEDULER_CPUS]` array – one single-entry slot per CPU, so it can never grow unbounded. If a slot already holds an undrained request, a new preflight fails closed (`rejected`) rather than queuing behind it. The IPI-context drain never nests the scheduler lock: it takes only the small per-CPU slot mutex, then calls the activation in `try_lock` mode.
- **Contention retry.** If the IPI-context drain finds the scheduler lock contended, it leaves the request parked and returns; the target CPU's next periodic timer tick (still live – the tick has not been suppressed) retries the drain. Progress is bounded by the periodic tick the same way the existing local re-check contention path is.
- **Fail-closed IPI ordering.** A remote rollback (`rollback_nohz_for_lease`) stales the lease generation *before* clearing the activation record. The drain re-checks the generation before arming, so a rollback that races the drain fails closed (the request is dropped, the periodic tick stays live). If the drain already committed before the rollback cleared the record, the target CPU's next `nohz_recheck` sees the `nohz-active` bit set with no record and restores its periodic tick. Either ordering converges on the periodic tick.
- **Compute-only.** Remote-CPU activation is limited to `namedRing = none` compute leases in this slice. A ring-coupled SQPOLL lease whose target differs from its ring owner's CPU is not an admitted shape; it fails closed.

Generic full-nohz admission for ordinary budgeted compute threads is available only through an explicit `SchedulingContext`-targeted compute lease and the same fail-closed placement gates described above. The SQPOLL nohz state machine now admits explicitly leased caller-thread rings when the SQPOLL worker is live, single-consumer, and bounded by producer wake/deadline rollback. Broader userspace-poller/device-queue admission, automatic CPU-isolation issuance, and production realtime island admission remain future work; `auto_nohz` stays disabled. Timeout-based auto-revoke landed 2026-05-30 15:22 UTC: a `CpuIsolationLease` created with `leaseLifetimeNs > 0` records an absolute expiry deadline, auto-revokes through the existing generation-advancing cleanup on first observation past it (`reason=lease-expired`), and the nohz activation record carries the lifetime deadline so a tickless CPU rolls back at the next timer/IPI recheck (`lease-lifetime-expired` disqualifier), bounded by `maxRevocationLatencyNs`. A `leaseLifetimeNs` of `0` preserves the prior revoke/cleanup-only lifecycle. The current SQPOLL-driven activation is the bounded case: tick suppression for a ring-coupled `kernelSqpoll` lease on the CPU running the preflight, rolled back through lease-generation staling on revoke/cleanup, with the SQPOLL ring-state re-check as defense-in-depth for any future SQPOLL-stop path that does not stale the lease.

Lease revocation and cleanup are generation-aware. Explicit revoke, process exit, service replacement through process termination, and session logout stale the matching generation so old caps cannot keep isolation eligibility alive, and rolling the matching lease's active nohz window back to the periodic tick is part of the same cleanup path. `make run-scheduler-cpu-isolation-lease` is the broad QEMU proof for grant, info, revoke, cleanup, real nohz activation and fail-closed rollback, bounded SQPOLL start/sleep/stop, rollback labels, generic full-nohz, and SQPOLL nohz. `make run-scheduler-generic-sqpoll-nohz` is the focused SQPOLL proof for eligible ring admission, producer wake, SQPOLL service, rollback, and stale owner rejection.

**Phase E: endpoint donation and return**

Synchronous endpoint delivery now carries a bounded internal donation token when a caller thread with a bound active `SchedulingContext` delivers a CALL to a receiver thread that has no scheduling context of its own. Donation is strictly passive-server shaped: receivers that already have a scheduling context keep their own authority, unbound callers donate nothing, and callers that receive a donation token are blocked from returning to userspace until the in-flight endpoint call returns or is canceled.

At delivery, the scheduler charges pre-donation caller runtime before moving the context ledger to the receiver. While the receiver handles the endpoint message, normal dispatcher runtime charging decrements the donated context. When endpoint RETURN commits the caller completion, the scheduler first charges receiver runtime since dispatch, then returns the remaining budget and next-replenishment state to the caller's thread metadata and rebinds the `SchedulingContext` record to the caller. Return preflight failures leave the in-flight donation in place, while application-exception RETURN, invalid-result RETURN errors, delivery failure, return cancellation, endpoint teardown, process/thread exit, and stale-caller cleanup return or clear the donation before waking the caller and without allocating new emergency-path storage. Nested donation of an already donated context is rejected; supporting stacked donation is deferred until it has an explicit return-token stack design.

`make run-scheduling-context` proves the behavior with a same-process endpoint round trip. The caller binds a fresh context, burns CPU immediately before CALL, the passive server burns CPU while servicing the endpoint CALL and again immediately before RETURN, and after RETURN the caller observes the reduced budget restored. The same smoke covers application-exception RETURN, oversized-result RETURN under donation, and deterministic rejection of A-to-B-to-C nested donation. It also submits a delivered donated CALL and then uses `cap_enter(0, 0)` while the server delays RETURN, proving the donor cannot continue outside the donated ledger. A fast-return variant covers the race where the receiver returns before the caller commits to the donation-blocked scheduler state. The smoke prints `endpoint_donation=ok`, `endpoint_return=ok`, `endpoint_exception_return=ok`, `endpoint_invalid_return=ok`, `endpoint_nested_rejected=ok`, `endpoint_donor_block=ok`, `endpoint_donor_fast=ok`, `endpoint_donation_server`, `endpoint_donation_after`, `endpoint_exception_return_after`, `endpoint_invalid_return_after`, `endpoint_nested_after`, `endpoint_donor_block_elapsed_ns`, `endpoint_donor_block_after`, `endpoint_donor_fast_elapsed_ns`, and `endpoint_donor_fast_after`.

**Phase E: SchedulingContext notifications**

Every `SchedulingContext` now owns fixed notification storage allocated at context creation or bootstrap. The storage has two coalescing slots: `budgetDepleted` and `deadlineOrTimeout`. Each slot records context id/generation, a saturating sequence, a saturating coalesced-event count, the last holder thread, remaining budget, the next replenishment/deadline timestamp, and whether the holder was using an endpoint-donated context. Runtime charge records depletion when remaining budget transitions to zero and records deadline/timeout expiry against the same context generation. Failed bind attempts do not arm a new budget/deadline window.

`SchedulingContext.drainNotifications()` returns typed observer results: `ok` drains the matching fixed cells, `revoked` reports the current revoked generation, and `staleGeneration` reports an old observer generation without draining the current record. Explicit `revoke()` records an `explicitRevoke` lifecycle event. These notifications explain already-enforced scheduler state; they do not donate budget, reorder runnable entities, bypass throttling, publish result caps, append unbounded queues, allocate on scheduler hard paths, or imply auto-nohz/SQPOLL/tickless behavior. A pre-armed observer waiter/wakeup path remains a future extension.

`make run-scheduling-context` proves the notification slice by repeatedly draining a depleted context after coalescing, observing deadline expiry, recording explicit revoke and stale-observer labels, and confirming that endpoint-donated runtime records notification state on the donated context. The smoke prints `notification_coalescing=ok`, `deadline_notification=ok`, `revoke_notification=explicitRevoke`, `stale_notification=staleGeneration`, and `endpoint_donated_notification=ok`.

**Phase E: session logout lifecycle hook**

`UserSession.logout()` now notifies the scheduler after the session liveness cell transitions from live to logged out. That covers explicit `UserSession.logout()` calls, including the remote DTO gateway logout command and connection-teardown path because those paths already call the same kernel `UserSession.logout()` method. The hook scans scheduler-owned process/thread metadata for live processes whose immutable `SessionContext` shares the logged out liveness cell, removes each non-donated matching thread binding from the scheduler ledger, and asks the bound `SchedulingContext` record to advance its generation and mark itself revoked. Old ordinary `SchedulingContext` grants therefore report stale generation through `info()` with zero visible remaining budget and `InfoOnlyNoDispatchChange`. The focused session-context smoke also proves stale `bindCallerThread()` does not rebind, stale `create()` does not publish a result cap, stale `revoke()` does not mutate the current metadata generation, and stale notification draining reports a stale observer result.

The hook intentionally does not use session code as a second scheduling-context ledger: session lifecycle code only flips liveness and notifies the scheduler, and the scheduler owns the scan and binding removal. The scan takes one binding at a time under the scheduler lock, drops that lock, then calls the `SchedulingContextExitCleanup` record hook so it does not invert the existing `SchedulingContext` record-lock to scheduler-lock order used by `bindCallerThread()`.

In-flight endpoint donation uses a conservative counted/skipped logout policy. If the logged-out session owns a receiver thread that currently holds a donated context, the logout hook records that the donated binding was skipped rather than returning donor budget while the endpoint call remains in flight. The focused session-context smoke proves the donor remains blocked in

`cap_enter(0, 0)` until the receiver returns, the hook reports `donation_inflight_skipped=1`, and endpoint RETURN removes the receiver binding while restoring only the reduced remaining budget to the donor. This does not add a new logout-triggered cancellation semantic. Local owner-shell exit now calls the held `UserSession.logout()` before clean shell process exit, so the same scheduler hook observes shell logout with `stale_marked=0 donation_inflight_skipped=0` in the shell smoke. The ordinary bound-context stale proof remains the focused session-context smoke, because the normal shell does not hold a bound `SchedulingContext`. Process and thread exit cleanup already have their own stale-context coverage and are unchanged.

Realtime islands, SQPOLL, auto-nohz, and CPU placement enforcement remain future Phase F/G work.

### Phase D Task 4: migration fairness invariants

Phase D Task 4 (2026-05-08) made three migration-fairness invariants explicit:

- **`virtual_runtime_ns` travels with the thread.** It lives on `Thread.cpu_accounting`, not on a per-CPU slot, so a migration from CPU A to CPU B preserves the thread's accumulated weighted-fair share. The accounting field was promoted out of `cfg(measure)` in Task 2 and continues to advance through `charge_runtime` regardless of which CPU charges the quantum.
- **`virtual_finish_ns` is derived per enqueue, never committed.** Every enqueue site – the initial publish in `enqueue_ready_thread_on_slot_locked`, the post-block requeue in `enqueue_unblocked_thread_on_slot_locked`, and the steal-insert in `steal_from_sibling_queues_locked` – routes through `refresh_virtual_finish_ns_locked`, which reads `thread.weight`, `thread.latency_class`, and `thread.cpu_accounting.virtual_runtime_ns` fresh and recomputes the WFQ ordering tag. The field is never carried as committed state across blocking and is never carried with the thread on migration; the destination CPU's view of weight, latency class, and quantum decides the new tag.
- **Steal recomputes at the destination.** The pop-from-source step in `steal_from_sibling_queues_locked` is followed by `refresh_virtual_finish_ns_locked` against the destination slot before the ordered insert, so a `SchedulingPolicyCap.setWeight` that landed between source enqueue and steal takes effect at the steal itself.

### Migrations counter shape

`ThreadCpuAccounting.migrations` is `cfg(feature = "measure")`-gated and remains a benchmark-only operator-observability counter; it is not load-bearing for ordering and is not exposed through `SchedulingPolicyCap.snapshot`. Phase D Task 4 moved the increment from the dispatch-time `scheduled_measure` path to two enqueue-time arms in `kernel/src/sched.rs`:

- **Placement-time spread** (`record_placement_spread_migration_locked`) fires from `push_reserved_run_queue_locked` when the enqueue target slot differs from the thread's previously dispatched CPU (`ThreadCpuAccounting.last_cpu`). A thread that has never been dispatched (`last_cpu == None`) does not register a migration on first publish; otherwise placement spread is counted exactly once per enqueue.

- **Steal** (record_steal_migration_locked) fires from steal_from_sibling_queues_locked after the source-queue removal and before the destination-queue insert. The steal scan skips the destination slot, so the counter increments unconditionally each time the steal arm is reached.

scheduled_measure still maintains last_cpu so the placement-spread check has the previous CPU available; only the migrations++ moved. The pre-collapse counter shape is preserved in steady state – a thread that runs on a different CPU than its previous run still records exactly one migration – but the increment is now attributed to the enqueue decision (placement spread or steal) rather than the dispatch that follows it.

The aggregate process-wide thread_placement counter family in kernel/src/measure.rs (migrations, migration_to_cpu0..3, consumed by tools/qemu-thread-scale-harness.sh) is a separate measurement device. It is incremented from account_thread_selected_locked at dispatch time and continues to observe "thread ran on a different CPU than its previously dispatched CPU" rather than the per-thread Task 4 enqueue-time shape, so the thread-scale harness regex does not need to change. The per-thread ThreadCpuAccounting.migrations field and the aggregate thread_placement counter intentionally measure different events at different points in the scheduling pipeline; both stay behind cfg(feature = "measure").

**Phase H: per-thread saturation status surface**

The Phase H AutoNoHz placement heuristic (a future policy-service feature) needs to read per-thread saturation observation in the normal dispatch build, not only under cfg(feature = "measure"). The non-measure per-thread saturation status surface (2026-05-30) promoted the inputs it consumes into ordinary ThreadCpuAccounting state and exports them through SchedulingPolicyCap.snapshot @2:

- **voluntary_blocks** and **preemptions** moved out of cfg(feature = "measure"). They are charged at the same sites as before – voluntary_blocks when a thread blocks itself (cap_enter wait, park, endpoint scheduling-context donation) and preemptions when the timer requeues a still-runnable running thread – so the measure build's counts are unchanged; only the cfg gate was removed. A low voluntary_blocks count distinguishes a CPU-saturating thread from an IPC/IO-bound one.
- **runnable_accumulated_ns** is a new always-built cumulative counter of runnable-but-not-running time. It is charged at the scheduler-lock-held enqueue/select boundary: push_reserved_run_queue_locked stamps a monotonic runnable_since_ns when a thread is published to a per-CPU run queue without being selected (idempotent across re-publish, so the whole runnable span is counted once), and account_thread_scheduled accumulates the monotonic delta and clears the stamp when the thread is next selected. The stamp/accumulate pair nets to zero for a thread selected at the same monotonic instant it becomes runnable. The clock is monotonic_ns() only (no wall-clock, no rewind), matching charge_runtime's discipline, and the stamp respects the runnable-ownership rules above (a thread holds a live stamp only between enqueue and selection).

migrations stays measure-gated; it is a placement diagnostic, not a saturation input. The surface exports raw cumulative counters only – windowing, smoothing, and the saturation decision are policy-service choices, never kernel state (see docs/proposals/tickless-realtime-scheduling-proposal.md). Proof: make run-thread-fairness reads the extended snapshot on

the weighted workers and asserts the CPU-bound hog reports high `runtime_ns` with `voluntary_blocks` at or near zero while at least one preempted lower-weight worker reports nonzero `preemptions` and `runnable_accumulated_ns`.

**Weight-change-while-enqueued contract**
`SchedulingPolicyCap.setWeight` writes the validated weight directly to `Thread.weight` through `Process::set_thread_weight` and does not clear `Thread.virtual_finish_ns`. A weight change observed while the thread is blocked, running, or already queued takes effect on the **next dequeue and re-enqueue** because every enqueue site refreshes `virtual_finish_ns` from current `weight`/`latency_class`/ `virtual_runtime_ns`. The kernel proves the contract two ways:

- **By construction.** `Process::refresh_thread_virtual_finish_ns` reads each input field fresh on every call; there is no cached derivation between enqueues. The function bears a doc-comment asserting the contract.
- **By `debug_assert!`.** Inside the same function, a debug assertion verifies that the recomputed `virtual_finish_ns` is at or beyond the current `virtual_runtime_ns` – a future deadline, never a past one. The assertion catches any future regression where the formula could underflow or where a stale cache could drift below the current vruntime.

The focused QEMU smoke that drives `setWeight` and verifies the post-block dispatch picks up the new weight landed under Phase D Task 5: `make run-thread-fairness-weight-change` (manifest `system-thread-fairness-weight-change.cue`, demo `demos/thread-fairness/`). Two competing child threads run a fixed wallclock window: a baseline worker stays at `DEFAULT_WEIGHT`, while a heavy worker self-calls `SchedulingPolicyCap.setWeight(weight=128)` and then blocks on `Timer.sleep` so it leaves the run queue before the contention window opens. Each worker snapshots its scheduler state at wake and at window end via `SchedulingPolicyCap.snapshot`, and the parent verifies three independent properties: (1) the heavy snapshot reads `weight == 128` and the baseline snapshot reads `weight == DEFAULT_WEIGHT`; (2) the observed `runtime_ns` ratio matches the weight ratio inside a configured tolerance; (3) the heavy worker's `virtual_runtime_ns` advances at roughly half the rate of its `runtime_ns` (vruntime/runtime  = 0.5 for weight=128,  = 1.0 for DEFAULT_WEIGHT). A scheduler that re-enqueued or dispatched the heavy worker using a stale `virtual_finish_ns` derived from `DEFAULT_WEIGHT` would not show the weight-proportional CPU share, and a scheduler that held a stale weight inside `charge_runtime` would yield heavy vruntime/runtime  = 1.0 instead of  = 0.5; the smoke trips on either regression. The capability is bound to `CapCallContext::caller_thread` (Phase D Task 2 decision), so same-thread self-mutation is the only authorized shape for this proof; cross-thread weight authority remains a Phase H privileged scheduler-policy service concern.

The thread-scale benchmark was repaired before accepting the milestone. The old 1 MiB/ spinning-parent shape was not a valid four-core reference because the matching Linux pthread baseline also failed at four workers. The accepted benchmark shape uses a blocking parent join, 262,144 blocks (16 MiB), and `work_rounds=64`. The formal accepted-evidence pair is the `capos-bench` 2026-05-02 21:38 UTC 5-run pair pinned to physical-core logical CPUs `0,1,2,3` against `main` commit `374f8556`: capOS work `1.883x` and total `1.787x` clear the configured `1.6x` gates, while the matching Linux pthread baseline records `1.988x`/`1.987x`. Its 1-to-4 row became the diagnostic that justified Phase D's fair-share enqueue policy: capOS `1.566x`/`1.538x` versus Linux

3.963x/3.858x, a clear bottleneck in the then-current single-global-queue scheduler. Phase D's WFQ evidence on 2026-05-10 manually accepted the recorded 1-to-4 diagnostic with capOS 3.088x/2.700x and matching Linux 3.974x/3.850x on the same host/CPU pin set. The harness still enforced only the configured 1-to-2 work/total speedup gates. Historical pre-collapse 1-to-2 (1.828x/1.687x) and the post-collapse 3-run diagnostic on capos-bench 2026-05-02 10:42 UTC (1.890x/1.792x, 1.504x/1.436x) remain in docs/benchmarks.md for reference. Four-worker capOS scaling was a follow-up rather than a completed claim under the pre-collapse model: the unsuppressed diagnostic recorded 1-to-4 work/total speedups 3.029x/2.386x, while suppressing scheduler switch logs recorded 3.272x/2.303x; remaining guest-measure evidence pointed at global Scheduler lock contention plus exit/join/block/schedule overhead, and normal scheduler-owned execution is still capped at temporary CPU slots 0-3. Each process currently owns one or more Thread records; each thread owns its saved CPU context, kernel stack, FS base, block state, and – since Phase D Task 2 – the WFQ ordering inputs weight: u16, latency_class: LatencyClass, and virtual_finish_ns: u64. The Phase D constants in capos-abi/src/scheduler.rs set the defaults weight = DEFAULT_WEIGHT and latency_class = LatencyClass::Normal, so unmodified workloads observe no behavior change versus the pre-Phase-D scheduler. virtual_finish_ns is recomputed on every enqueue (Task 2 ships the derivation; Task 3 will consume it for ordered insertion) and is not meaningful while the thread is blocked.

Phase D Task 2 split the per-thread CPU accounting record so the WFQ-load- bearing fields are available in the normal qemu build: runtime_ns, virtual_runtime_ns, and last_started_ns are unconditional; context_switches, preemptions, voluntary_blocks, migrations, last_cpu, and the *_runtime_stable_observed and blocked/exited bookkeeping stay behind the measure feature because they are pure operator-observability counters that do not participate in dispatch ordering and need a separate operator snapshot path. runtime_ns advances 1:1 with elapsed CPU time, while virtual_runtime_ns advances by elapsed_ns * REFERENCE_WEIGHT / weight so per-thread weight changes the cumulative WFQ share rather than only the enqueue tag. The runtime-charge path is invoked when a current thread stops running through timer preemption, blocking cap_enter or park, thread/process exit, or direct switch/handoff paths that select another current thread; the wrapping helpers in kernel/src/sched.rs route through Process::charge_thread_runtime / Process::account_thread_scheduled unconditionally now.

The SchedulingPolicyCap cap surface mutates these per-thread fields through the **caller-thread fallback** binding selected in Phase D Task 2: every method (setWeight, setLatencyClass, snapshot) routes to CapCallContext::caller_thread, so a holder can only mutate or observe its own running thread. Cross-thread or cross-process authority is reserved for the Phase H privileged scheduler policy service. The SchedulingPolicyCap.snapshot reply intentionally exposes only the four fields promoted out of the measure feature gate; context_switches/preemptions/voluntary_blocks/migrations are benchmark-only and a future operator-observability slice may add them through a separate cap. The BSP scheduler tick normally arrives through the local APIC timer on vector 48 with LAPIC EOI after calibrating the LAPIC initial count against PIT channel 2; if LAPIC setup or calibration is unavailable, the kernel falls back to the legacy PIT/PIC IRQ0 path on vector 32. On each user-mode timer tick (kernel-mode ticks bypass the scheduler entirely through kernel_timer_interrupt_handler, as described under

Design), the kernel wakes timed-out or satisfied `cap_enter` and park waiters, processes the current thread's ring endpoint in timer mode, saves the current thread context, picks the next ready thread from the single global run queue (the earlier per-CPU local-first / steal scan was retired with the queue collapse), switches CR3 when needed, updates the current CPU's kernel-entry stack through the per-CPU hook, restores FS base, mirrors the next `ThreadRef` into the current `PerCpu`, and returns to the next user context.

When APs are online and their LAPIC timers start, scheduler CPU slots `0-3` can temporarily own scheduler/user execution. The earlier AP-owner proof kept the BSP in kernel idle; the current same-process scaling slice allows sibling threads with distinct ring endpoints to run on different scheduler CPUs while processes that hold broad launch/authority caps or live endpoint objects remain pinned to the legacy single-owner CPU. Additional APs beyond CPU `3` stay in kernel idle until a later scheduler-owner policy replaces the temporary CPU mask. The runnable queues are a per-CPU array of `VecDeque<ThreadRef>` shared by the scheduler-owned CPUs under the global scheduler lock and ordered ascending by `virtual_finish_ns`; process/thread metadata remains shared under that lock. A bounded steal path migrates the most overdue sibling candidate (each sibling queue's first entry that the destination CPU considers Runnable) when a CPU's local queue has no runnable entry.

Syscall entry initializes kernel GS with `swapgs`, saves the user RSP through the GS-relative `PerCpu.user_rsp` slot, and switches to the GS-relative `PerCpu.kernel_rsp` slot. Normal syscall returns swap back before `sysretq`. Blocking `cap_enter`, process `exit`, and `ThreadControl.exitThread` paths that leave through scheduler `iretq` restore use `restore_context_after_syscall` so GS ownership is returned to userspace before the next user context resumes.

`Timer.sleep` records a bounded scheduler waiter keyed by caller `ThreadRef`, user data, and an absolute monotonic `deadline_ns`. Due sleeps validate the thread generation, post an empty completion directly to the caller's CQ, and then flow through the same blocked `cap_enter` wake scan as other completions. Each process has a separate sleep waiter quota, so one Timer holder cannot fill the global sleep queue by itself.

`ThreadControl.setFsBase` validates runtime-provided FS bases as user-canonical addresses, updates the caller thread's saved FS base, and writes the CPU FS base immediately when the caller is the running thread. There is no process-global FS base; context switch treats FS base as per-thread state.

The initial thread still uses the compatibility ring at `RING_VADDR`, while each spawned child thread receives a kernel-chosen ring mapping in the process ring arena. Run queues, per-CPU `current`, direct IPC handoff, Timer sleep waiters, process/terminal waiters, endpoint caller/receiver records, and deferred cancellation CQEs store generation-checked `ThreadRef` values and route completions to the target thread's ring endpoint. Process-owned thread and kernel-stack ledger limits are enforced by `ThreadSpawner.create` before additional thread records become runnable. The frozen contract is in In-Process Threading. Park wait uses a separate `Blocked(Park { ... })` reason and park timeout/wake completions use reserved CQE credits before marking generation-checked waiter threads runnable. The authority and ABI contract is in Park Authority.

`cap_enter(min_complete, timeout_ns)` processes pending SQEs immediately. If the requested completion count is not available and the timeout permits blocking, the current thread enters `Blocked(CapEnter { ... })` and the syscall entry path switches to another runnable thread.

The LAPIC user-timer path enters `sched::schedule()` unconditionally on every tick. An earlier slice carried a bounded user-mode continuation fast path with a per-CPU one-skip budget and a release/acquire slow-path-required summary; that path has been retired (see `docs/backlog/scheduler-evolution.md` "Cleanup: Retire Benchmark-Driven Scaffolding Before Phase D"). The fast path saved at most one scheduler entry every other tick on an uncontended single-CPU-effective scheduler while paying for shadow-state publication on every slow-path exit, so the simpler always-schedule shape is preferred until a future Phase D or Phase F slice ships an evidence pair where the fast path measurably reduces scheduler-lock hold time on a contended SMP run.

When endpoint delivery satisfies a blocked server RECV, the scheduler can set a direct IPC target. The next scheduling decision runs that server before ordinary round-robin work when it is ready and its `ThreadRef` generation still matches the captured direct target. When the direct slot is unavailable, endpoint completions fall back to the queued path with `WakePolicy::QueueCpu(slot)` targeting the current CPU's per-CPU queue, so the wake scan probes the placed CPU first.

### Design

The implementation keeps ring dispatch outside the global scheduler lock. Timer dispatch extracts ring/cap/scratch handles, releases the scheduler lock, processes bounded SQEs, then reacquires the scheduler lock to choose the next thread. This prevents Cap'n Proto decode, serial output, and capability method bodies from running under the global scheduler lock.

There is no longer a slow-path-required summary or a per-CPU skip budget for the user-mode timer path. Every user-mode LAPIC timer tick enters `sched::schedule()`, which services run-queue entries, direct IPC targets, deferred process termination/drop and thread-stack cleanup, Timer sleep waiters, and blocked threads with timer-backed `cap_enter` or Park timeouts under the scheduler lock. Those timeout paths compare absolute monotonic deadlines, but periodic ticks still decide when the checks run. Ring SQEs and ordinary cap waiters run on the same per-tick cadence. Kernel-mode timer ticks (e.g., on AP cores parked in the kernel idle loop) still go through `kernel_timer_interrupt_handler`, which sends EOI without entering the scheduler. The shared `advance_bsp_tick` helper still increments the compatibility `TICK_COUNT` only on CPU 0; normal runtime accounting and timeout comparisons use `monotonic_ns()` instead. Future per-CPU fair-share slices may reintroduce a continuation path under explicit Phase D or Phase F authority; until then the always-schedule shape keeps the scheduler's authority over thread metadata and runnable ownership single-source.

The runnable queues keep a single-owner contract behind the global scheduler lock. A live generation-checked `ThreadRef` may have at most one runnable dispatch owner across per-CPU `current/handoff_current` slots, the per-CPU run queues, and the single `direct_ipc_target` preference slot. Blocked waiters, sleep waiters, park waiters, endpoint state, process waiters, and join waiters are not runnable owners; they may make a thread ready only after liveness and generation checks succeed.

Migration between per-CPU queues is represented as a scheduler-lock- contained transfer, not as a second published owner. The source owner is removed or popped first and the `ThreadRef` is then inserted in the destination queue at the position determined by a freshly recomputed `virtual_finish_ns`, or selected as the next running thread. `virtual_runtime_ns` travels with the thread; `virtual_finish_ns` is recomputed at every enqueue and never carried as committed state, so weight or class mutations applied while the thread was blocked take effect on the next dequeue and re-enqueue. Retry paths requeue the candidate after dropping duplicate queued copies. Direct IPC keeps its preference slot only while the target remains live and runnable; if the direct target cannot run immediately, it falls back through the normal queued-owner path on the current CPU's per-CPU queue.

Idle-to-runnable wake targeting reuses the same ownership boundary. A thread that becomes ready through endpoint completion, timer sleep, park wake, process wait, or thread join is pushed to the placement target's per-CPU run queue, and `wake_idle_scheduler_cpus_locked` first probes the placement target when the policy is `QueueCpu`, then walks eligible idle scheduler CPUs to wake the first that accepts a fresh reschedule IPI; CPUs that already have a pending IPI (or that fail LAPIC delivery) are skipped without breaking the scan, so a burst of ready work cross-wakes more than one neighbor instead of stranding the rest behind one already-targeted CPU. Direct IPC uses the same path. Measurement builds expose aggregate and per-phase counters for wake scans, eligible idle CPUs, targeted CPUs, IPIs sent, already-pending IPI skips, not-ready target skips, missing LAPIC targets, and send failures.

Each per-CPU run queue is reserved up to the live runnable-capable thread count before publication; the shared live reservation count is released on process/thread exit or pre-publication rollback. Reserving each queue to the full live-thread count is required because the bounded steal path may migrate every live thread into a single sibling queue between two scheduler passes. Timer preemption, unblock, direct- IPC fallback, requeue, and steal-requeue paths therefore must not allocate while the thread is already live.

Process and thread exit cleanup proves the removal side of that ownership contract at the cleanup site. After removing queued owners and clearing a matching direct IPC target, the scheduler lock remains held while the kernel scans every per-CPU runnable queue and the direct target slot; any stale exiting process or thread reference is a kernel assertion failure. The focused spawn smoke asserts the corresponding serial proof markers on exercised process and thread exit paths.

The Phase C migration order is constrained by hardware state, not only by scheduler data structures. The first gate moved syscall entry/exit off BSP-symbol-relative `PerCpu` fields and onto `KernelGsBase`/`swapgs` on user syscall paths, including blocking `cap_enter`, `exit`, and `ThreadControl.exitThread` paths that leave through `iretq` rather than the normal `sysretq` epilogue. The second gate added xAPIC initialization, a PIT-calibrated BSP LAPIC timer tick, LAPIC EOI routing, AP LAPIC initialization, a LAPIC spurious-vector handler, and an IPI vector plus bounded vector-49-only fixed IPI send primitive. The third gate added address-space resident CPU masks, per-CPU pending full-TLB flush generations, completion waits, and a vector-49 TLB shootdown handler for user page-table `map`, `unmap`, and `protect`. The fourth gate split current-thread tracking into per-CPU slots, registers AP `PerCpu` records for current-thread and syscall

stack mirrors, updates AP TSS.RSP0 on context switches, and hands the single scheduler-owner role to AP cpu=1 when it is online with a programmed LAPIC timer.

The LAPIC slice replaces the BSP-oriented PIT/PIC scheduler tick on supported QEMU and hardware paths. `kernel/src/arch/x86_64/idt.rs` keeps vector 32 for the PIT/PIC fallback, reserves vector 48 for LAPIC timer delivery plus vector 49 for cross-CPU requests, and installs vector 255 for LAPIC spurious interrupts. `pic.rs` can remap and mask all legacy IRQs once LAPIC ticks are active, and `context.rs` sends LAPIC EOI or PIC EOI according to the active timer source. The IPI vector now handles TLB shootdown requests and bounded reschedule requests for AP idle-to-runnable handoff.

The TLB slice wraps user page-table mutations that can affect an address space resident on another CPU. `AddressSpace::map`, `AddressSpace::unmap`, and `AddressSpace::protect` still perform the local `x86_64` mapper flush, then call the architecture shootdown helper with the address space's resident CPU mask. The helper records pending full-TLB flush generations for online resident CPUs other than the caller, sends vector-49 IPIs, and returns a completion token. Capability handlers drop the address-space guard and enqueue completion work; `cap_enter` and timer polling drain that queue after ring dispatch releases the cap-table and scratch locks. This keeps a remote syscall that is contending on the same process locks from blocking maskable IPI delivery forever. Capability handlers reserve fixed-size deferred queue slots before page-table mutation, so full queues fail closed as capability overload errors instead of surfacing after rollback, unmap, or protect has already changed state. Drains flush the current CPU before waiting so a CPU that is itself in the target mask cannot wait on its own pending generation. Target CPUs drain the generation in the IPI handler, at syscall entry, or before returning to userspace from syscall, timer, and scheduler restore paths. Generation counters avoid losing overlapping shootdowns while a target CPU is already draining a prior request. This relies on kernel user-buffer access continuing through address-space-locked HHDM copy/read helpers rather than raw user virtual addresses while a delayed flush generation exists. Callers include `VirtualMemoryCap` dispatch through `parse_map`, `parse_unmap`, and `parse_protect`, plus `MemoryObjectCap::{map,unmap,protect}` in `kernel/src/cap/frame_alloc.rs`. Scheduler CR3 handoff now marks the selected address space resident on the current CPU, including AP cpu=1 during the AP scheduler-owner proof.

### Idle paths

There are two distinct idle paths, and both run genuine **CPL0 (kernel-mode)** idle. There is no user-mode idle process: when no real work is runnable a CPU runs the kernel idle code at CPL0 on the kernel PML4. The two paths differ only in how the CPU got there.

The **cooperative CPL0 kernel-mode idle path** is the boot/AP path. `start` (BSP), `start_ap` (APs), and the `start_current_cpu` loop call `next_start_context`; when that returns no real runnable work they fall into `idle_current_cpu_once`, which `hlt`s at CPL0 on the per-CPU kernel stack with interrupts enabled (no `CpuContext`, no `restore_context` — the same way `start_current_cpu` itself runs). A kernel-CPL timer tick or reschedule IPI taken during that `hlt` runs the kernel-mode handler (`kernel_timer_interrupt_handler` / `handle_reschedule_ipi`, both of which call `nohz_recheck`), so the nohz one-shot deadline is preserved and re-armed across the `hlt`; control then returns to the loop, which re-checks for work. `idle_current_cpu_once` increments the `KERNEL_IDLE_HLT_ENTRIES` counter and emits a bounded

cpu-isolation: kernel-idle hlt cpu=… idle_path=cooperative-cpl0 … nohz_active=… timer_source=… log line so this path is observable from the kernel log; the run-scheduler-cpu-isolation-lease smoke asserts it is reached. Once any dispatch path restore_contexts into a real thread, the start_current_cpu frame is abandoned.

The **steady-state CPL0 idle-thread path** is reached from the four interrupt/syscall-return dispatch call sites — schedule() (timer), capos_block_current_syscall, exit_current, and exit_current_thread. When choose_next_locked falls through to this CPU's idle thread, each site builds the dispatch tuple from the per-CPU CPL0 idle-thread context. The dispatch call sites hand a CpuContext to assembly that restore_contexts (or, for the timer path, return a context pointer plus a CR3 the timer handler loads), so they need a schedulable context when no real work is runnable; the CPL0 idle context is that context.

**CPL0 idle-thread context infrastructure.** arch::smp::init_idle_kernel_stacks allocates one dedicated CPL0 idle kernel stack per scheduler CPU slot from fresh contiguous frame ranges, so they do not overlap the boot kernel stacks, the per-thread kernel stacks, or the IST slots. CpuContext::new_cpl0_idle builds a kernel-shaped context (kernel-code/kernel-data selectors, rip = kernel_idle_entry, rsp into the idle kernel stack). sched::sched_init, called from kmain, constructs and stores one CpuContext per CPU slot in CPL0_IDLE_CONTEXTS and then calls register_idle_process_locked to seed the **slot-0 synthetic idle Process record** before the scheduler runs (this keeps the BSP idle process's low PID and the init-process PID ordering stable); the remaining per-CPU slots are registered lazily by current_cpu_idle_thread_locked the first time their CPU reaches idle. sched_init panics on OOM, as does the lazy path: the CPL0 idle contexts and the synthetic idle records are scheduler idle infrastructure and there is no fallback idle path, so a failure to build them is unrecoverable. The idle kernel stack is sized as a **full per-thread kernel stack** (PROCESS_THREAD_KERNEL_STACK_PAGES), not an IST slot, because kernel_idle_entry runs the deep service_periodic_work() call chain on it (see periodic-service parity below).

**Synthetic idle process records.** The idle thread is never a runnable user-mode process. The synthetic idle Process (Process::new_idle) maps no user code, no user stack, and no cap ring, and carries an empty cap table. It exists only so the idle ThreadRef resolves through sched.processes and the scheduler's ThreadRef-centric bookkeeping — set_thread_state, account_thread_selected_locked, current-thread tracking, and the is_idle_thread guard predicate used pervasively across the scheduler — keeps working unchanged. Its address_space is a bare page-table root with nothing user-mapped; it is required by the Process struct but is **never loaded as CR3**. Every idle dispatch site routes the CPU onto the kernel PML4 via the CPL0 idle context, so the synthetic idle AddressSpace is never made resident and never participates in resident_cpu_mask or TLB-shootdown idle-residency handling.

**Dispatch-tuple rewire.** After choose_next_locked returns, when the chosen thread is idle_threads[current_cpu_slot()], each dispatch site builds the dispatch tuple from the CPL0 context pointer, the dedicated idle kernel stack top, the kernel PML4 CR3, and the current FS base (no FS-base change). sched_init builds one CPL0 idle context per scheduler CPU slot or panics, so cpl0_idle_context(slot) is infallible at every dispatch site. The schedule() timer path does **not** route through a dedicated CR3-loading restore helper: the existing timer_interrupt_handler already loads the tuple's CR3 with write_cr3 before the privilege-

agnostic five-element `iretq`. The three syscall-path sites (`capos_block_current_syscall`, `exit_current`, `exit_current_thread`) keep their `restore_context_after_syscall` restore tail: they are entered via `syscall_entry` (which already executed `swapgs`), so the exit `swapgs` is required to leave the CPL0 idle thread running with the *user* GS base — the same GS-base state the timer path's CPL0 idle thread runs with. Each site emits a distinct marker: `sched: dispatch idle cpu=N idle_path=cpl0-dispatch-timer` (timer), `…cpl0-dispatch-block` (blocking syscall), and `…cpl0-dispatch-exit` (both `exit_current` and `exit_current_thread`). `debug_assert!s` guard the CPL0 dispatch tuple: context `cs`/`ss` are the kernel selectors and their RPL bits are `0`.

**CPL0 idle periodic-service parity.** `schedule()`'s timer Phase 2 runs periodic service work on every tick — deferred process drops, pending terminations, `wake_cap_waiters`, `service_sqpoll_workers()`, `drain_pending_endpoint_cancellations()`, `terminal_session::poll_input()`, `virtio::poll_scheduler()`, and the network / pipe / interrupt `poll_waiters()` calls. A CPL0 idle thread's timer ticks are kernel-mode and go through `kernel_timer_interrupt_handler`, which never enters `schedule()` — so without explicit parity handling that servicing would be stranded whenever a CPU is parked on the CPL0 idle thread. That work is factored into a single `service_periodic_work()` function with one lock discipline: the scheduler lock is taken only for the bounded deferred-drop / thread-stack-release / `wake_cap_waiters` / pending-termination extraction, then **dropped** before `drop_pending_process` / `finish_terminated_process` and the lock-free poll block. `schedule()` calls it after ring dispatch; `kernel_idle_entry` is its own cooperative loop that, each iteration, runs `service_periodic_work()`, then `next_start_context(false)` to re-dispatch a real runnable thread the moment one appears (`allow_idle = false` so it never re-selects the idle thread), then `idle_current_cpu_once()` to `hlt`. The re-dispatch is required: without it a kernel-mode timer tick taken during the idle `hlt` returns through `kernel_timer_interrupt_handler`, which does not re-enter `schedule()`, so the CPU would be stranded. `service_periodic_work()` and `next_start_context()` run with **interrupts disabled** in that loop — the CPL0 idle context is built `IF=1` so the periodic tick can preempt the `hlt`, so the loop must `cli` before the deep service call; otherwise a CPL0 timer tick taken *during* `service_periodic_work()` nests a `kernel_timer_interrupt_handler` frame onto the idle kernel stack (same-privilege interrupts do not switch stacks). `idle_current_cpu_once` re-enables interrupts only across its `enable_and_hlt` and disables them again before returning. There is no double-service: a CPU running a real thread gets the service block via `schedule()`, a CPU on the CPL0 idle thread gets it via the `kernel_idle_entry` loop, and a given tick on a given CPU is CPL3 (`schedule()`) xor CPL0-idle (the loop). nohz cadence stays honest because the loop iterates at the timer/IPI cadence — when the periodic tick is suppressed the re-armed one-shot still wakes the `hlt`, so `service_periodic_work()` still runs.

#### `iretq` CPL0 restoration invariant and CPL0 idle-thread prerequisites

This subsection records the load-bearing x86-64 architectural invariant that any future CPL0 idle-thread context migration must satisfy, along with the prerequisites the implementation will need to meet.

**Authoritative reference:** Intel 64 and IA-32 Architectures Software Developer's Manual (SDM), Volume 2A, `IRET`/`IRETQ` instruction reference, "Operation" pseudocode (the `IF OperandSize = 64` / 64-bit-mode path), and Volume 3A, Section 6.14.3 "Returning from an Exception or Interrupt

Procedure." The description below applies to `IRETQ` in 64-bit long mode; the legacy 32-bit `IRET` paths behave differently and are called out explicitly where it matters.

**`iretq` frame layout and the 64-bit unconditional five-element pop.** `iretq` in 64-bit long mode **unconditionally** pops five 64-bit (8-byte) values from the top of the current kernel stack, in order: `RIP`, `CS`, `RFLAGS`, `RSP`, `SS`. This is true **regardless of whether the privilege level changes** — both a CPL0→CPL3 return and a CPL0→CPL0 return consume the same five-element frame and load `RSP`:`SS` from it. AMD deliberately removed the legacy conditional stack switch for long mode: the "skip `SS`:`ESP` on a same-privilege return" behavior exists **only** in the legacy 32-bit `IRET` operand-size paths, never in `IRETQ`.

- **CPL0 → CPL3 (privilege change, ring exit):** The target `CS` has RPL=3, which differs from the current CPL=0. The CPU installs `RIP`, `CS`, and `RFLAGS` from the frame, then loads `RSP` and `SS` from the same frame and transfers to the user-space instruction at `RIP` on the user stack.
- **CPL0 → CPL0 (same-privilege, no ring change):** The target `CS` has RPL=0, matching the current CPL=0. `iretq` **still pops all five elements**: it installs `RIP`, `CS`, and `RFLAGS`, and **also loads `RSP` and `SS`** from the frame, exactly as in the CPL3 case. There is no same-privilege short-circuit in 64-bit mode. The practical consequence for a CPL0 restore is the opposite of the legacy intuition: the frame's `rsp` and `ss` fields are load-bearing and **must** carry a valid kernel stack pointer and a valid RPL=0 stack selector, because the CPU will load them.

**Current code.** `restore_context` (`kernel/src/arch/x86_64/context.rs` lines 311–328) sets `RSP` to the supplied `CpuContext` pointer, pops all fifteen caller-saved and callee-saved GPRs (lines 315–327), and executes `iretq` (line 328). The `CpuContext` struct (`context.rs` lines 133–155) places `rip`, `cs`, `rflags`, `rsp`, and `ss` at the high end of the struct (lines 150–154), matching the hardware interrupt-frame layout that the CPU pushes when it enters the timer interrupt handler. The comment at line 149 ("Pushed by CPU on interrupt from Ring 3") reflects how every `CpuContext` is populated today, but the five-element `iretq` frame itself is not CPL3-specific — `iretq` consumes the same five elements for any target CPL.

**User-thread contexts.** Every *user-thread* `CpuContext` is built by `Thread::new_user` (`kernel/src/process.rs`), which sets `cs = sel.user_code.0 as u64` (RPL=3, value `0x23`) and `ss = sel.user_data.0 as u64` (RPL=3, value `0x1B`). Every `iretq` issued by `restore_context` or `restore_context_after_syscall` into a user thread is therefore a CPL0→CPL3 privilege change into a fully user-shaped context.

**CPL0 idle contexts coexist with user contexts.** The blocker for a CPL0 target is *not* `iretq` frame arithmetic: `iretq` pops the same five elements for a CPL0 target as for a CPL3 target, so a frame carrying kernel selectors and a valid kernel `rsp` `iretq`s correctly. The real requirements are in the surrounding dispatch plumbing, all of which the CPL0 idle path satisfies:

- **CR3.** The dispatch call sites set `CR3` to the kernel PML4 for the CPL0 idle path, not to any user `AddressSpace` page table. The synthetic idle `Process`'s `AddressSpace` is never loaded as CR3.
- **swapgs / GS-base.** A CPL0 idle context was never entered through the `syscall` path. The `schedule()` timer path reaches it through the timer handler's own CR3 load and the privilege-agnostic `iretq` tail (no `swapgs` in that path at all). The three syscall-path sites (`capos_block_current_syscall`, `exit_current`, `exit_current_thread`) keep their `restore_context_after_syscall` tail: those sites *were* entered via `syscall_entry` (which

already swapgsed), so the exit swapgs is required to undo it — leaving the CPL0 idle thread running with the *user* GS base, the same state the timer path produces.

- **Kernel-code and kernel-data selectors.** A CPL0 CpuContext uses cs = sel.kernel_code.0 as u64 (RPL=0, value 0x08) and ss = sel.kernel_data.0 as u64 (RPL=0, value 0x10). Because iretq loads ss unconditionally in 64-bit mode, ss must be a valid RPL=0 stack selector; the GDT data-selector privilege checks require an RPL=0 ss to be paired with an RPL=0 cs, so the whole context (cs, ss, rsp, CR3, GS base) is kernel-shaped together.
- **Idle kernel stack.** Each CPL0 idle thread has its own dedicated kernel stack (arch::smp::init_idle_kernel_stacks) that does not overlap any IST slot, any per-thread kernel stack, or the BSP/AP boot stacks. Because iretq loads rsp from the frame, the context's rsp points into this dedicated stack. It is sized as a full per-thread kernel stack because kernel_idle_entry runs the deep service_periodic_work() call chain on it.
- **No user AddressSpace residency.** The synthetic idle Process's AddressSpace is never made resident and never participates in resident_cpu_mask, so TLB shootdown never stalls waiting for an idle CPU.
- **No blocking, no exit.** The idle thread never calls cap_enter, parks, blocks on any waiter, or exits. The Invariants section entry "The idle thread must never block in cap_enter or exit" carries forward unchanged.

CpuContext::new_cpl0_idle builds the kernel-shaped context, sched::kernel_idle_entry is the entry point, and sched::sched_init wires the per-CPU CPL0 idle contexts and seeds the slot-0 synthetic idle process record (the remaining slots' records are registered lazily by current_cpu_idle_thread_locked). All four dispatch call sites — schedule(), capos_block_current_syscall, exit_current, exit_current_thread — route idle dispatch onto the CPL0 idle context: the timer path returns the CPL0 context pointer plus the kernel PML4 CR3 in its dispatch tuple and relies on the existing timer_interrupt_handler CR3-load; the three syscall-path sites keep their restore_context_after_syscall tail so the syscall-entry swapgs is undone. The CPL0 contexts are kernel-shaped across cs, ss, rsp, and CR3 together.

### Measurement Policy

Design grounding for this policy: this document's scheduler invariants, docs/backlog/scheduler-evolution.md, docs/proposals/scheduler-evolution-proposal.md, docs/research/future-scheduler-architecture.md, docs/research/out-of-kernel-scheduling.md, docs/research/nohz-sqpoll-realtime.md, and docs/research/completion-ring-threading.md. In particular, docs/research/future-scheduler-architecture.md keeps the always-on versus benchmark-only scheduler telemetry split as an open scheduler question, and the current answer is intentionally conservative.

The current kernel/src/measure.rs counters are benchmark instrumentation, not normal operator observability. They stay behind the measure feature and CAPOS_THREAD_SCALE_GUEST_MEASURE=1 because they add atomics, cycle-counter reads, phase bookkeeping, and in some cases sampled user RIP values to hot scheduler, timer, TLB, ring, and serial paths. Normal QEMU and dispatch builds must not depend on those counters being present.

The per-thread runtime-accounting ledger is split. The WFQ load-bearing core fields, `runtime_ns`, `virtual_runtime_ns`, and `last_started_ns`, are unconditional normal-build state on `ThreadCpuAccounting`: WFQ ordering, `SchedulingPolicyCap.snapshot`, and `SchedulingContext` budget charging depend on them outside `cfg(feature = "measure")`. The diagnostic fields (`context_switches`, `preemptions`, `voluntary_blocks`, `migrations`, `last_cpu`, blocked/exited stability probes, placement buckets, and per-phase attribution counters) stay behind the `measure` feature. Permanent operator observability is still separate work: it should expose low-rate, non-symbolic snapshots derived from the unconditional ledger plus event counters such as runnable queue depths or high-water marks, reschedule IPI sent/failed/pending counts, TLB shootdown request/failure counts, and scheduler policy admission or denial counts. Those counters must not allocate, log, read raw user PCs, or perform cycle-timing in timer, unblock, direct-IPC fallback, requeue, or steal-requeue paths.

Benchmark-only attribution stays in `measure`: per-phase thread-scale checkpoints, guest cycle timings for ring/capnp/method/scheduler segments, scheduler-lock wait and hold cycles, scheduler-lock site attribution, serial byte attribution, timer-mode breakdown, CR3/TLB event totals, thread-placement selection/migration buckets, raw user-PC samples, logging-suppression A/B evidence, and workload/cacheline diagnostics. The publish-placement publish/caller-aware buckets were retired with the per-CPU run-queue collapse. Phase D shipped the fair-share enqueue policy but did not reintroduce those placement counters. A future branch may promote a specific event count only by adding the normal-build storage/API and proving the same emergency-path constraints; it should not simply remove the current `cfg(feature = "measure")` boundaries from the benchmark module.

The publish-placement publish/caller-aware buckets are still retired; Phase D Task 3 brought back per-CPU placement semantics but does not re-emit the publish counters. Re-instate them through a separate operator-observability slice that proves the same emergency-path constraints, not by removing the existing `cfg(feature = "measure")` boundary on the historical buckets.

Tickless idle is enabled only for true idle. A scheduler-owned CPU may mask the periodic LAPIC tick when it is running the CPL0 idle context, has no runnable non-idle work, has no active `CpuIsolationLease` nohz record, has no local deferred cleanup, has no cap-enter polling dependency, and the one-shot clockevent plus non-tick-derived monotonic clocksource are available. The replacement one-shot is bounded by the nearest `Timer`/`ParkSpace` deadline or a 100 ms idle housekeeping floor, and the scheduler restores periodic mode before non-idle dispatch, reschedule-IPI wake, or rollback. Cap-enter polling waiters, including the current terminal shell path, and ready threads paused in a `SchedulingContext` retry window keep the periodic tick until those dependencies move behind explicit deadlines or housekeeping placement.

Generic full-nohz for ordinary budgeted compute threads carries the clockevent/deadline substrate into the CPU-isolation state machine and suppresses ticks only after network polling, IRQ affinity, accounting, deadline, lifetime, and rollback obligations pass. SQPOLL nohz applies the same substrate to explicitly leased caller-thread rings once the SQPOLL worker is live and the single-consumer, owner-lease, wake, and rollback gates pass. Automatic policy issuance and broader SQPOLL userspace-poller/device-queue admission remain separate later CPU-isolation features; see Tickless and Realtime Scheduling and NO_HZ, SQPOLL, and Realtime Scheduling.

Exit switches to the kernel PML4 before tearing down the exiting address space, releases capability authority, completes process waiters, defers final process teardown until the scheduler is running on another kernel stack, and then releases remaining thread kernel stacks through the scheduler-owned `OffStackToken` path before the `Process` value is dropped.

**Invariants**

- The idle thread must never block in `cap_enter` or exit.
- Ring dispatch must not hold the scheduler lock.
- Timer dispatch copies current-process user buffers through that process's locked `AddressSpace`; it must not rely on a raw current-CR3 validate/use window.
- Blocked `cap_enter` waiters wake when enough CQEs are available or their finite timeout expires.
- Timer sleep waiters must be bounded per process, tied to the caller `ThreadRef` generation, and removed when the caller process exits.
- Runtime-controlled FS bases must stay in user canonical space.
- Direct IPC handoff is a scheduling preference, not a bypass of process liveness, generation, or state checks.
- The scheduler must update TSS.RSP0 and the per-CPU syscall kernel RSP through `percpu::set_kernel_entry_stack` on each switch.
- Each `PerCpu.current_thread` mirrors that CPU's scheduler current slot; the scheduler lock remains the authority for current-thread and queue ownership even though dispatch/runnable state is now separate from shared process and thread metadata.
- Each live `ThreadRef` may appear in the per-CPU runnable queues at most once across all queues, and every per-CPU queue's capacity must be reserved up to the live runnable-capable thread count before a new process or thread becomes runnable.
- A live generation-checked `ThreadRef` must have at most one runnable dispatch owner across per-CPU `current`/`handoff_current` slots, the per-CPU runnable queues, and the direct IPC target.
- Queue migration (including the bounded steal path) must be a scheduler-lock-contained remove-before-publish transfer; no path may publish the same `ThreadRef` twice into any queue or leave a stale direct target after exit. Migration must recompute `virtual_finish_ns` at the destination and never carry the source's WFQ tag as committed state.
- Each per-CPU run queue must remain ordered ascending by `virtual_finish_ns` after every enqueue, requeue, or steal-requeue. Local selection scans the queue by index for the first destination-Runnable entry; RetryLater entries are left in place for the next scheduler pass. The bounded steal path scans each sibling queue's indices ascending for that queue's first Runnable-for- destination entry — because each queue is ordered ascending, the first Runnable hit per queue is the lowest `virtual_finish_ns` candidate the destination can accept on that source — then picks the source queue whose first-Runnable candidate has the lowest `virtual_finish_ns` globally, with ties broken by lower CPU id. The chosen entry is removed from its actual position on the source queue (not necessarily the head).

- Process and thread exit cleanup must assert, before releasing the scheduler lock, that the exiting process or thread has no remaining entry in any per-CPU runnable queue and no remaining direct IPC target slot.
- Timer, unblock, direct-IPC fallback, requeue, and steal-requeue paths must use reserved run-queue capacity and avoid allocation.
- Runtime accounting must use the normal monotonic clocksource, not benchmark-only cycle counters, and must charge only running intervals.
- FS base is saved and restored across context switches for TLS.
- Thread records remain generation-checked `ThreadRef` identities; exited records are retained only while a live handle, pending join, or unjoined status can still observe them.
- The final teardown of an exiting process must not release thread kernel stacks until another kernel stack is active, and the implicit `Thread::Drop` path must not free kernel-stack frames.
- A scheduler CPU must never run the same generation-checked `ThreadRef` twice at once; same-process siblings may run on different scheduler CPUs only when their completions route through distinct per-thread ring endpoints.
- Park waiters must be keyed by generation-checked `ThreadRef` values, reserve one waiter CQE credit, and must not allocate in wait, wake, timeout, or process-exit cleanup paths.

### Code Map

- `kernel/src/sched.rs` - shared process table plus `SchedulerDispatch` ownership of the per-CPU runnable queues (ordered ascending by `virtual_finish_ns`), per-CPU current/handoff slots, idle-thread slots, direct IPC target, run-queue reservation accounting, pending drops, and pending stack releases; also blocking, wakeups, Timer sleep waiters, the bounded steal path, and exit.
- `kernel/src/arch/x86_64/context.rs` - CPU context layout, timer entry/restore, tick counter.
- `kernel/src/arch/x86_64/idt.rs` - timer and IPI interrupt handler wiring.
- `kernel/src/arch/x86_64/lapic.rs` - xAPIC MMIO setup, PIT-calibrated LAPIC timer, LAPIC EOI, spurious-vector handling, and fixed-IPI send primitive.
- `kernel/src/arch/x86_64/tlb.rs` - serialized vector-49 TLB shootdown request, pending flush generations, completion token, and interrupt/user-return drain path.
- `kernel/src/arch/x86_64/pic.rs` and `kernel/src/arch/x86_64/pit.rs` - legacy PIC remap and PIT fallback setup.
- `kernel/src/arch/x86_64/gdt.rs` - BSP/AP TSS and kernel stack storage.
- `kernel/src/arch/x86_64/syscall.rs` - blocking syscall transition for `cap_enter`.
- `kernel/src/arch/x86_64/percpu.rs` - per-CPU syscall stack registry, TSS.RSP0 update hook, and current thread storage.
- `kernel/src/arch/x86_64/tls.rs` - FS base save/restore.
- `kernel/src/process.rs` - process state, kernel stacks, the synthetic idle process record, and per-thread CPU accounting storage/accessors.

### Validation

- `make run-smoke` validates timer preemption, ring fairness, direct IPC handoff, blocked `cap_enter` wakeups, process exit, and clean halt.
- `make run-spawn` validates process wait blocking and child exit completion through `ProcessHandle.wait`, Timer monotonic now/sleep completion through `timer-smoke`, per-process sleep quota isolation through `timer-flood`, and thread/park lifecycle behavior through `thread-lifecycle`.
- `make run-measure` validates the post-thread park blocked/resume timing path and process exit while a park waiter is parked.
- `cargo build --features qemu` verifies QEMU-only scheduler and halt paths.
- QEMU smoke output for IPC includes direct handoff diagnostics when the server is woken from a blocked RECV.

### Open Work

- Prove SQPOLL/poller progress that does not depend on periodic scheduler ticks before automatic nohz activation. Then implement tickless idle only for no-runnable-work CPU idle. Keep runnable contention on periodic preemption until the activation proof closes the remaining network polling, IRQ affinity, and housekeeping dependencies.
- Keep SMP behind per-CPU scheduler state and review of any path that needs page pinning beyond the `AddressSpace`-locked copy/read contract.
- Implement the remaining SMP Phase C slices: split shared scheduler metadata, replace the temporary scheduler-owner mask, and collect accepted benchmark evidence.
- Add priority or policy scheduling only after the current authority and IPC semantics remain stable.
- Add service restart policy outside the static boot graph.

# Part IV: Security and Verification

Review boundaries, verification commands, trusted inputs, panic surfaces, DMA constraints, and known design risks.

## Security Verification Track Registry

The `S.x` labels used across this manual are registry identifiers for the Security Verification Track. They are not product stages. When a section mentions one of these labels, read it as shorthand for the track name below.

- `S.1` — CI bootstrap. Status: Landed.
- `S.2` — Miri and proptest on capos-lib. Status: Landed.
- `S.3` — Manifest and mkmanifest fuzzing. Status: Landed.
- `S.4` — Ring Loom harness. Status: Landed.
- `S.5` — Kani on capos-lib. Status: Initial bounded gate landed.
- `S.6` — Security review docs stay aligned. Status: Ongoing.
- `S.7` — Stage-6-aware security refresh. Status: Planned/ongoing.
- `S.8` — Untrusted-service hardening gate. Status: Planned.
- `S.9` — Authority graph and resource accounting. Status: Design landed.
- `S.10` — Supply-chain and generated-code trusted computing base. Status: Partially landed.
- `S.11` — Device and DMA isolation gate. Status: Design accepted; implementation gates open.
- `S.12` — Kani harness bounds refresh. Status: Planned.
- `S.13` — ELF parser arbitrary-input coverage. Status: Landed.
- `S.14` — Telnet IAC filter fuzz coverage. Status: Landed.
- `S.15` — Telnet differential round-trip and line-discipline extraction. Status: Landed.
- `S.16` — Ring SQE wire-validation extraction and fuzz target. Status: Landed.
- `S.17` — Sanitizers on host tests. Status: Planned.

### Subtracks Used In This Manual

- `S.10.0` under `S.10` — Trusted build input inventory.
- `S.10.2` under `S.10` — Generated-code drift check.
- `S.10.3` under `S.10` — Dependency policy and no_std review gate.
- `S.11.1` under `S.11` — DMA capability invariants.
- `S.11.2` under `S.11` — Userspace-driver ownership-transition gate.

The S.11.2.0 through S.11.2.9 labels in the DMA chapter are local checklist rows for the userspace-driver transition gate. They are acceptance criteria under S.11.2, not separate project tracks.

## Trust Boundaries

This page gives reviewers one place to find the hostile-input boundaries, trusted inputs, and current isolation assumptions that matter for capOS security review.

### Current Boundaries

#### Ring 0 to Ring 3

- **Trust rule:** the kernel trusts no userspace register, pointer, SQE, CapSet, or result-buffer field.
- **Implemented:** syscall arguments, user-buffer ranges, page permissions, opcodes, and capability-table lookups are validated before privileged use in `kernel/src/arch/x86_64/syscall.rs`, `kernel/src/mem/paging.rs`, `kernel/src/mem/validate.rs`, and `kernel/src/cap/ring.rs`.
- **Validation:** Panic-Surface Inventory and `REVIEW.md` at the repository root.

#### Capability Table to Kernel Object

- **Trust rule:** a process acts only through a live table-local `CapId` with matching generation and interface.
- **Implemented:** `capos-lib/src/cap_table.rs` owns generation-tagged slots; kernel capability dispatch goes through `CapObject::call`.
- **Validation:** `cargo test-lib` plus QEMU ring and IPC smokes recorded in done task records.

#### Capability Ring Shared Memory

- **Trust rule:** userspace owns SQ writes, but the kernel owns validation, dispatch, completion, and failure semantics.
- **Implemented:** SQ/CQ headers and entries live in `capos-config/src/ring.rs`; kernel dispatch bounds indexes, opcodes, transfer descriptors, and CQ posting, and copies CALL/RECV/RETURN buffers while holding the owning process `AddressSpace` lock.
- **Validation:** `cargo test-ring-loom` plus QEMU ring corruption, reserved opcode, fairness, IPC, and transfer smokes.

#### Endpoint IPC and Transfer

- **Trust rule:** IPC cannot create or destroy authority except through explicit copy, move, release, or spawn transactions. Delegating an imported client facet must preserve service-visible identity, and shared-service handlers should derive caller identity from live caller-session metadata.
- **Implemented:** `kernel/src/cap/endpoint.rs`, `kernel/src/cap/transfer.rs`, and `capos-lib/src/cap_table.rs` implement queued calls, `RECV`/`RETURN`, copy/move transfer, legacy receiver metadata propagation, and rollback. Legacy badge metadata is a debug/test surface only. Normal chat, stdio bridge, and shared demo handlers use live caller-session metadata instead of caller-selected badge identity.
- **Validation:** Authority Graph and Resource Accounting for Transfer and any open transfer review-finding task records.

#### Manifest and Boot Package

- **Trust rule:** boot manifest bytes and embedded binaries are untrusted until parsed and validated. Only BootPackage holders can request chunked manifest bytes; ordinary services receive no default boot-package authority.
- **Implemented:** `tools/mkmanifest` validates the embedded `initConfig` graph before serialization. Kernel bootstrap validates the kernel-owned manifest boundary before loading one init process; init BootPackage validation resolves service graph references before spawning children. `kernel/src/cap/boot_package.rs`, `capos-lib/src/elf.rs`, and load paths still enforce manifest-read, ELF, load-range, CapSet, and interface bounds.
- **Validation:** `cargo test-config`, `cargo test-mkmanifest`, `cargo test-lib`, manifest and ELF fuzz targets, and `make run`.

#### Process Spawn Inputs

- **Trust rule:** parent-supplied spawn params, ELF bytes, grants, legacy badges, and result-cap insertion must fail closed. Endpoint kernel grants must not share owner caps with the parent. Delegated client facets must not be relabeled into another service identity.
- **Implemented:** `ProcessSpawner` validates ELF load, grants, frame exhaustion, parent cap-slot exhaustion, child-local endpoint creation, and parent-only client result facets. Delegated `ClientEndpoint` grants preserve source identity; explicit relabel encodings fail closed except for owner or trusted parent endpoint-result caps.
- **Validation:** spawn QEMU smoke evidence and review-finding task records.

#### Console Authentication and Setup

- **Trust rule:** console input, account selectors, password verifiers, setup tokens, and passkey challenge state are hostile or sensitive until a login/session component validates them.
- **Implemented:** `Console` remains output-only. The first interactive boundary is session-scoped `TerminalSession` with bounded `readLine`, visible/hidden echo, structured cancellation, one move-only foreground holder, caller-session-checked output, and stale-input scrubbing on cancel or owner teardown. `CredentialStore` verifies one manifest-supplied Argon2id operator credential and one bounded volatile RAM-overlay password created by first-boot setup. `capos-shell` drives `login` and `setup`; there is no separate `ConsoleLogin` service.
- **Validation:** Proposal: Boot to Shell, boot-to-shell gates in `../../docs/tasks/README.md`, `make run-terminal`, `make run-login`, and `make run-login-setup`.
- **Open/future:** durable multi-account credential storage, multiple verifier records, rotate/disable state, broader anti-enumeration audit policy, and bounded single-use setup-token/challenge state.

#### Session Authority and Audit

- **Trust rule:** authenticated sessions receive only broker-issued narrow caps. Audit output, service logs, terminal output, and failed-auth diagnostics must not disclose secrets or verifier material.
- **Implemented:** `SessionManager` mints entropy-backed `UserSession` metadata for operator, explicitly seeded guest, and anonymous profiles. Endpoint caller-session references are HMAC-SHA256 values scoped by an entropy-backed boot key and non-reused endpoint service-scope

id. `AuthorityBroker` validates session/profile matches before minting bundles. `RestrictedLauncher` returns shell-scoped launch paths instead of BootPackage or broad ProcessSpawner authority.

- **Validation:** Proposal: User Identity, Sessions, and Policy, Proposal: Boot to Shell, `make run-login`, `make run-login-setup`, and future auth/session hostile input tests.
- **Open/future:** audit records, stable service-audit identity across endpoint replacement, opaque record IDs, mutable session liveness cells, `UserSession.logout`, owner-shell/gateway close propagation, narrow renewal paths, broader policy evaluation, and web-terminal origin/RP-ID validation.

**SSH Remote Shell Ingress**

- **Trust rule:** SSH network input, keys, usernames, channel requests, PTY state, environment requests, and disconnects are hostile until the gateway validates protocol state, authenticates the user, and receives a broker-issued shell bundle.
- **Implemented:** current proofs cover schema stubs, one development-only host-key fixture, manifest-seeded `AuthorizedKeyStore`, public-key session minting, unsupported-feature policy, restricted shell launch, and bounded terminal-host wiring over host-local plain TCP. The proposal keeps SSH transport authority in `SshGateway`; the spawned shell receives only an `SshTerminalFactory`-produced `TerminalSession` plus the normal scoped session bundle through `RestrictedShellLauncher`. Fixture host-key signing, authorized-key mapping, public-key session minting, SSH policy rejection, and restricted launcher inputs fail closed for malformed or unsupported cases.
- **Validation:** Proposal: SSH Shell Gateway, Runtime, Networking, And Shell Backlog, `make run-ssh-host-key`, `make run-ssh-authorized-key`, `make run-ssh-public-key-session`, `make run-ssh-public-key-auth`, `make run-ssh-feature-policy`, and `make run-restricted-shell-launcher` (the plain-TCP `run-ssh-gateway-terminal-host` terminal-host proof is retired with the kernel socket owner).
- **Open/future:** full SSH transport transcript and channel binding, password-auth verifier/backoff wiring, production host-key storage, broader account storage, and production remote-shell hardening.

**Identity Metadata and Account Records**

- **Trust rule:** users, principals, accounts, sessions, roles, and profile names are policy metadata, not kernel subjects, ambient authority, or substitutes for held capabilities.
- **Implemented:** sessions receive capabilities only after authentication and broker policy evaluation; a principal or account record does not run or call the kernel.
- **Validation:** Local Users, Storage, And Policy Backlog and Proposal: User Identity, Sessions, and Policy.
- **Open/future:** durable local account-store behavior, profile persistence, rollback checks, and quota enforcement.

**Host Tools and Filesystem**

- **Trust rule:** manifest/config input must not escape intended source directories or invoke unconstrained host commands.

- **Implemented:** `tools/mkmanifest` validates references and path containment, rejects unpinned CUE compilers, and Makefile targets route CUE and Cap'n Proto through pinned tool paths.
- **Validation:** Trusted Build Inputs, `make generated-code-check`, and `make dependency-policy-check`.

**Generated Code and Schema**

- **Trust rule:** schema, generated bindings, and no_std patches are trusted build inputs.
- **Implemented:** `schema/capos.capnp`, build scripts, `tools/generated/capos_capnp.rs`, and `tools/check-generated-capnp.sh` make generated-code drift review-visible.
- **Validation:** Trusted Build Inputs and `make generated-code-check`.

**Device DMA and MMIO**

- **Trust rule:** current userspace receives no raw DMA buffer, device physical address, virtqueue pointer, BAR mapping, or device interrupt handle.
- **Implemented:** the QEMU virtio-net path is allowed only through kernel-owned bounce buffers and kernel-owned MSI-X source records routed through the kernel device interrupt dispatch table, bounded device MSI vector pool, and kernel-owned route lifecycle checks.
- **Validation:** DMA Isolation Design and `make run-net`.
- **Open/future:** typed `DMAPool`, `DeviceMmio`, and `Interrupt` capabilities for userspace-driver transition.

**Panic and Emergency Paths**

- **Trust rule:** hostile input should produce controlled errors, not panic, allocate unexpectedly, or expose stale state.
- **Implemented:** ring dispatch is mostly controlled-error; remaining panic surfaces are classified by reachability and tracked as hardening work.
- **Validation:** Panic-Surface Inventory and `REVIEW.md`.

### Security Invariants

- All authority is represented by capability-table hold edges; no syscall or host tool path should bypass the capability graph.
- Identity metadata is not authority. Principals identify audit and policy subjects, accounts store durable policy inputs, profiles select bundle and quota templates, and sessions receive caps only through explicit broker minting.
- The interface is the permission: method authority is expressed by the typed Cap'n Proto interface or by a narrower wrapper capability, not by ambient process identity.
- Kernel operations at hostile boundaries validate structure, bounds, ownership, generation, interface ID, and resource availability before mutating privileged state.
- Failed transfer, spawn, manifest, and DMA setup paths must leave ledgers, cap tables, frame ownership, and in-flight call state unchanged or explicitly rolled back.
- Trusted build inputs must be pinned or drift-review-visible before their output becomes part of the boot image or generated source baseline.

- Authentication/session code must treat credential records, setup tokens, passkey challenges, session IDs, and audit logs as security boundaries, not ordinary console text.

**Open Work**

- Unify fragmented resource ledgers into the authority-accounting model so reviewers can audit quotas without following parallel counters.
- Harden open panic-surface entries that become more exposed as spawn, lifecycle, SMP, or userspace drivers expand hostile input reachability.
- Keep DMA in kernel-owned bounce-buffer mode until the `DMAPool`, `DeviceMmio`, and `Interrupt` transition gates have code and QEMU proof.
- Do not expand production authentication or remote-shell surfaces without hostile-input tests for bounded terminal input, credential failure paths, challenge expiry/replay, audit redaction, and narrow shell cap bundles.

## Verification Workflow

This page maps capOS claims to the commands, QEMU smokes, fuzz targets, proof tools, and review documents that currently support them.

### Local Command Set

Use the repo aliases and Makefile targets instead of bare host commands. The workspace default Cargo target is `x86_64-unknown-none`, so host tests rely on aliases that set the host target explicitly.

- **Scope:** Formatting
  - **Command:** `make fmt-check`
  - **What it checks:** Rust formatting across kernel, shared crates, standalone userspace crates, and demos.
- **Scope:** Config and manifest logic
  - **Command:** `cargo test-config`
  - **What it checks:** Cap'n Proto manifest encode/decode, CUE value handling, CapSet layout, and config validation.
- **Scope:** Ring concurrency model
  - **Command:** `cargo test-ring-loom`
  - **What it checks:** Bounded SQ/CQ producer-consumer invariants and corrupted-SQ recovery behavior.
- **Scope:** Deferred-completion concurrency model
  - **Command:** `make model-dma-deferred-completion-loom`
  - **What it checks:** Bounded Loom over the kernel `DeferredCompletionQueue` reservation budget and the multi-CPU TLB shootdown generation re-read (`kernel/src/arch/x86_64/tlb.rs`); budget never exceeded, no completion dropped/double-popped, no retire ahead of a covering flush.
- **Scope:** DMA authority lifecycle model
  - **Command:** `make model-dma-tla`
  - **What it checks:** Pinned TLC bounded check of `models/dma/dma_authority.tla`: allocate->map->publish->complete->revoke->scrub->reuse ordering plus generation-keyed stale completion, record-before-PTE-install split, drive-pin/quarantine, and queue-enable epoch-fence interleavings. Fails closed on any invariant violation, deadlock, or analyzer error.
- **Scope:** DMA assurance aggregate
  - **Command:** `make dma-assurance-model-check`
  - **What it checks:** Local aggregate over the DMA Alloy, TLA+, Loom, and Kani gates. Requires installed `cargo-kani`; GitHub CI splits the same evidence across the DMA Assurance Models and Kani Proofs jobs.
- **Scope:** Shared library logic
  - **Command:** `cargo test-lib`

- **What it checks:** ELF parser, frame bitmap, frame ledger, capability table, and property-test coverage.

- **Scope:** Manifest tool
  - **Command:** `cargo test-mkmanifest`
  - **What it checks:** Host-side manifest conversion and validation behavior.
- **Scope:** Userspace runtime
  - **Command:** `tools/check-userspace-runtime-surface.sh`; `make capos-rt-check`; `make init-capos-build demos-capos-build shell-capos-build`
  - **What it checks:** Runtime primitive ownership, custom-target boot build path, entry ABI, typed clients, ring helpers, and no_std constraints.
- **Scope:** Kernel build
  - **Command:** `cargo build --features qemu`
  - **What it checks:** Kernel build with the QEMU exit feature enabled.
- **Scope:** Generated code
  - **Command:** `make generated-code-check`
  - **What it checks:** Cap'n Proto compiler path/version, schema binding output equality, no_std patch anchors, Adventure and Paperclips content freshness, locked generator dependencies, and checked-in generated-output drift.
- **Scope:** Dependency policy
  - **Command:** `make dependency-policy-check`
  - **What it checks:** cargo-deny and cargo-audit policy across root and standalone Cargo lockfiles, plus npm lockfile validation and audit checks for the docs toolchain.
- **Scope:** Mandatory Kani gate
  - **Command:** `make kani-lib`
  - **What it checks:** Bounded `capos-lib` harness set for frame allocation, stale-handle rejection, frame-grant and cap-slot fail-closed accounting, and transfer-origin fail-closed behavior.
- **Scope:** DMA-authority core Kani gate
  - **Command:** `make kani-dma-authority`
  - **What it checks:** Bounded Kani over the extracted pure DMA-authority core (`capos_lib::dma_authority`): a recycled slot's generation strictly increases and never aliases a live handle, a stale-generation completion is rejected without mutating completion/free/reuse state, and a buffer cannot be re-exposed until its in-flight completion is observed. Faithful model of the `kernel/src/device_dma.rs` authority arithmetic; the kernel call-through is a tracked follow-up.
- **Scope:** Full image build
  - **Command:** `make`
  - **What it checks:** Kernel, userspace demos, runtime smoke binaries, manifest, Limine artifacts, and ISO packaging.
- **Scope:** Default interactive boot
  - **Command:** `make run`

- **What it checks:** Operator-facing default init-owned boot path from layered `system.cue`: standalone init starts the foreground shell, resident demo services, and the remote-session CapSet gateway, forwards only the remote CapSet endpoint on loopback, and keeps console/debug output logged separately.

- **Scope:** Default QEMU smoke
  - **Command:** `make run-smoke`
  - **What it checks:** Scripted focused shell-led boot from `system-smoke.cue`: kernel boot-launches `capos-shell` as `init`, grants the shell bootstrap cap bundle, then proves anonymous-session bootstrap, `login` failed-auth redaction, successful password auth, broker upgrade to operator bundle, terminal isolation, and clean halt.
- **Scope:** Focused spawn QEMU smoke
  - **Command:** `make run-spawn`
  - **What it checks:** Narrower init-owned ProcessSpawner graph: kernel boot-launches only standalone `init` with Console, BootPackage, and ProcessSpawner; init validates BootPackage metadata, spawns endpoint/IPC/VirtualMemory/Timer/FrameAllocator children, waits for them, exercises hostile spawn checks, and halts cleanly.
- **Scope:** Shell, terminal, and local-auth smokes
  - **Command:** `make run-shell`; `make run-terminal`; `make run-credential`; `make run-login`; `make run-login-setup`
  - **What it checks:** Anonymous shell behavior, TerminalSession line input and cancellation, CredentialStore verifier behavior, username-aware password login, broker-issued operator bundle upgrade, volatile first-boot setup credential creation, terminal isolation, and stale-handle release.
- **Scope:** Focused service smokes
  - **Command:** `make run-chat`; `make run-adventure`; `make run-paperclips`; `make run-revocable-read`; `make run-memoryobject-shared`; `make run-ringtap-failing-call`
  - **What it checks:** Resident-service demos, the clean-room Paperclips terminal demo, revocation behavior, MemoryObject sharing, and debug-tap viewer behavior.
- **Scope:** Networking smoke
  - **Command:** `make run-net`; `make qemu-net-harness`
  - **What it checks:** QEMU virtio-net attachment, kernel PCI/device-discovery path, descriptor-accounting guard evidence, ARP, and ICMP. TCP/UDP socket proof lives under the Phase C userspace network-stack gates.
- **Scope:** SSH gateway proof smokes
  - **Command:** `make run-ssh-host-key`; `make run-ssh-authorized-key`; `make run-ssh-public-key-session`; `make run-ssh-public-key-auth`; `make run-ssh-feature-policy`; `make run-restricted-shell-launcher`
  - **What it checks:** Development host-key fixture validation, authorized-key mapping, public-key session minting, public-key authentication failure privacy, unsupported SSH feature policy, and restricted shell launch authority. The bounded host-local socket-to-TerminalSession wiring proof is retired with the kernel socket owner.

Do not claim full verification unless the relevant command actually ran in the current change. For doc-only changes, use an appropriately narrower check such as `mdbook build`.

**Review Workflow**

1. Identify the changed trust boundary or state that the change is docs-only.
2. Read `REVIEW.md` (at the repository root) for the applicable security, unsafe, memory, performance, capability, and emergency-path checklist.
3. Read the relevant review-finding task records under `docs/tasks/` before judging correctness so known open findings are not treated as solved behavior.
4. For system-design work, list the concrete design and research files read; reviewers should reject vague grounding such as "docs" or "research".
5. Run the smallest command set that exercises the changed behavior, then add QEMU proof for user-visible kernel or runtime behavior.
6. Record unresolved non-critical findings as task records under `docs/tasks/` or `docs/tasks/on-hold/` with concrete remediation context before treating the task as reviewed.

**Evidence by Claim**

- **Claim type:** Parser or manifest validation
  - **Required evidence:** Host tests for valid and malformed input; fuzz target when arbitrary bytes can reach the parser.
- **Claim type:** Kernel/user pointer safety
  - **Required evidence:** QEMU hostile-pointer smoke plus code review of address, length, permissions, and validation-to-use windows.
- **Claim type:** Ring or IPC transport behavior
  - **Required evidence:** Host model/property tests where possible, plus QEMU process output proving success and failure paths.
- **Claim type:** Userspace runtime primitive ownership
  - **Required evidence:** `tools/check-userspace-runtime-surface.sh` plus review of `capos-rt/src/entry.rs`, `alloc.rs`, `panic.rs`, and `syscall.rs`.
- **Claim type:** Capability transfer or release
  - **Required evidence:** Rollback tests for copy/move/release failure, cap-slot exhaustion, stale caps, and process-exit cleanup. A release-only proof shows local cleanup only; any claim that peers, children, sessions, or delegated holders lose authority needs a separate explicit revoke, session-expiry, object-epoch, or service-specific invalidation proof.
- **Claim type:** Resource accounting
  - **Required evidence:** Tests that prove quota rejection, matched release on success and failure, and process-exit cleanup.
- **Claim type:** Generated code, schema, or generated content changes
  - **Required evidence:** `make generated-code-check` and a checked-in baseline diff generated by the pinned compiler or pinned CUE/generator path.
- **Claim type:** Dependency or toolchain changes

- **Required evidence:** Dependency-class review plus `make dependency-policy-check`; update Trusted Build Inputs when trust assumptions change.

- **Claim type:** Device or DMA work
  - **Required evidence:** `make run-net` or a targeted QEMU smoke; no userspace-driver transition without the gates in DMA Isolation Design.
- **Claim type:** Panic-surface hardening
  - **Required evidence:** Updated Panic-Surface Inventory when reachability or classification changes.
- **Claim type:** Authentication and session work
  - **Required evidence:** Host tests for `TerminalSession` line-input bounds, secret-mode echo suppression, cancellation behavior, exclusive terminal handoff, non-inheritance without an explicit grant, verifier encoding, entropy-unavailable fail-closed behavior, bootstrap-plus-RAM-overlay credential handling, volatile credential/disable-state disclosure, bounded single-use setup-token/challenge first-consume/expiry/replay semantics, generic failure/backoff policy, and audit redaction with opaque record IDs plus pre-auth serial-safe failure events; QEMU proof for setup/login, failed auth, successful `capos-shell` launch through `TerminalSession`/`CredentialStore`/`SessionManager`/`AuthorityBroker`, lack of terminal access for an ungranted child, absence of broad `BootPackage`/raw `ProcessSpawner` caps in the shell, and fail-closed behavior when the secure-randomness path is unavailable.

### Fuzzing and Proof Tracks

The current fuzz corpus lives under `fuzz/` and covers manifest Cap'n Proto input, exported JSON conversion for `mkmanifest`, arbitrary ELF parser input, Telnet IAC filtering, terminal line discipline, and ring SQE wire validation. Run fuzzers when a change alters those parsers, schema shape, terminal/network byte-stream handling, SQE validation, or related validation rules.

Kani coverage is intentionally narrow and lives in `capos-lib`, where pure logic can be bounded without hardware state. Add or refresh Kani harnesses for ledger, cap-table, bitmap, and parser invariants when those invariants become part of a security claim. The required local/CI gate is `make kani-lib`. The extracted DMA-authority core (`capos_lib::dma_authority`) has its own bounded gate, `make kani-dma-authority`, which proves ownership-generation bump on recycle, stale-handle rejection without mutation, and no-re-expose before completion — a faithful model of the `kernel/src/device_dma.rs` arithmetic whose kernel call-through is a tracked follow-up.

Loom coverage belongs in shared ring logic. Extend `cargo test-ring-loom` when SQ/CQ ownership, ordering, corruption recovery, or wake semantics change.

DMA assurance model files live under `models/dma/` and are bounded checked evidence for device and cloud-backend claims. The Alloy relational authority graph (`models/dma/dma_authority.als`) is now an **analyzer-checked** gate: `make model-dma-alloy` runs the pinned Alloy Analyzer 6.2.0 headless at scope `for 4` and fails on any counterexample (free-page reachability, same-domain IOVA uniqueness, and the ownership-generation stale-handle gate), with the checked verdict table recorded in `models/dma/README.md`. The focused Loom gate for the `DeferredCompletionQueue` reservation budget and the multi-CPU generation re-read is also checked (`make model-dma-deferred-completion-loom`, pinned loom 0.7.2). The TLA+ lifecycle

model (`models/dma/dma_authority.tla`) is now a **model-checked** gate as well: `make model-dma-tla` runs the pinned TLC 2.19 (tla2tools 1.7.4) over the bounded configuration (2 devices / 2 domains / 2 pages / 2 iovas, generations 0..1) and fails closed on any invariant violation, deadlock, or analyzer error, with the checked result recorded in `models/dma/README.md`. It covers the lifecycle ordering plus the landed generation-keyed stale completion, record-before-PTE-install split, drive-pin/ quarantine, and queue-enable epoch-fence interleavings. The extracted pure DMA-authority core is checked by Kani as well (`make kani-dma-authority`, pinned `kani-verifier` 0.67.0): ownership-generation bump on recycle, stale-handle rejection without mutation, and no-re-expose before completion over `capos_lib::dma_authority`.

The DMA checked gates are wired into CI. The GitHub `dma-assurance-models` job runs `make model-dma-alloy`, `make model-dma-tla`, and `make model-dma-deferred-completion-loom`; the `kani-proofs` job runs `make kani-dma-authority` after the mandatory `make kani-lib` gate. The local `make dma-assurance-model-check` aggregate runs all four when `cargo-kani` is installed. Do not claim Verus evidence – or any Alloy/TLC/Loom/Kani DMA-authority result beyond what these targets actually check – unless the exact command, checker version, configuration, model bounds, and output are recorded in the task closeout.

For DMA work, map claims through DMA Assurance Model: TLA+ for lifecycle ordering and races, Alloy for authority topology, Kani for pure Rust validators/accounting, and Loom for atomic or queue interleavings such as `DeferredCompletionQueue`. The model supplements the required QEMU or cloud evidence; it does not replace hardware-facing smokes.

### Documentation Sources

- `REVIEW.md` (at the repository root): rules for security, unsafe code, capability invariants, resource accounting, and emergency paths.
- `docs/tasks/`: open remediation backlog, review-finding task records, and latest verification task records.
- Trusted Build Inputs: trusted compiler, generated-code, dependency, bootloader, manifest, and host-tool inputs.
- Panic-Surface Inventory: classified panic-like surfaces and commands used to generate the inventory.
- Authority Graph and Resource Accounting for Transfer: authority graph, quota, transfer, rollback, and ProcessSpawner accounting invariants.

## Trusted Build Inputs

This inventory covers the build inputs currently trusted by the capOS boot image, generated bindings, host tooling, and verification paths. It started as the Security Verification Track S.10.0 inventory, records the Security Verification Track S.10.2 generated-code drift check, and now also records the Security Verification Track S.10.3 dependency policy plus the shared no_std generated-code patch helper. The consolidated long-horizon supply-chain risk view – floating Rust nightly, repo-pinned `qemu-system-x86_64` / `xorriso` digests (CI now apt-installs `qemu-system-x86=1:8.2.2+ds-0ubuntu1.16`, `xorriso=1:1.5.6-1.1ubuntu3`, and `ovmf=2024.02-2ubuntu0.8` so package identity is captured; the OVMF firmware blob is now repo-pinned by SHA-256 (`OVMF_CODE_SHA256`, landed at commit `f1c8c8fb`, merged at `ca5a1fea`) and the `ovmf-verify` Makefile gate fails the build on drift, but download-and-verify of the `qemu-system-x86_64` / `xorriso` tool blobs remains a future step), PR-blocking CI environment provenance comparison, and the remaining immutable-runner-image / repo-managed tool-digest gap – is tracked as R13 in `docs/design-risks-register.md`; the gap text below stays consistent with that entry.

### Summary

- **Input:** Limine bootloader binaries
  - **Current source:** `Makefile:5-10`, `Makefile:34-49`
  - **Pinning status:** Git commit and selected binary SHA-256 values are pinned.
  - **Drift-review status:** `make limine-verify` fails if the checked-out commit or copied bootloader artifacts drift.
- **Input:** Rust toolchain
  - **Current source:** `rust-toolchain.toml:1-4`, `.github/workflows/ci.yml`
  - **Pinning status:** Date-pinned `nightly-2026-04-20` channel with target triples and the `rust-src` component required by custom-target `-Zbuild-std` userspace builds. The CI `host-baseline`, `dma-assurance-models`, and `qemu-smoke` jobs explicitly request the same dated nightly. The Kani job remains pinned separately to `nightly-2025-11-21` paired with the Kani-compatible bundle installed by `cargo kani setup`.
  - **Drift-review status:** The dated channel resolves to `rustc 1.97.0-nightly (e22c616e4 2026-04-19)` (the 2026-04-20 manifest carries the previous day's rustc commit). Bumps are review-visible as `rust-toolchain.toml` and workflow diffs; the advance procedure is recorded in the Rust Toolchain section below.
- **Input:** Workspace cargo dependencies
  - **Current source:** `Cargo.toml`, crate `Cargo.toml` files, `Cargo.lock`
  - **Pinning status:** Lockfile pins exact crate versions and checksums for the root workspace. Manifest requirements remain semver ranges.
  - **Drift-review status:** `make dependency-policy-check` runs `cargo deny check` plus `cargo audit` against the root workspace and lockfile in CI.
- **Input:** Standalone cargo dependencies (covered by `make dependency-policy-check`)
  - **Current source:** `init/Cargo.lock`, `demos/Cargo.lock`, `demos/wasi-hello-rust/Cargo.lock`, `demos/wasi-cli-args/Cargo.lock`, `demos/wasi-env/Cargo.lock`, `demos/wasi-

fs/Cargo.lock, demos/wasi-random/Cargo.lock, demos/wasi-preview1-refusals/Cargo.lock, demos/wasi-stdio-fd/Cargo.lock, tools/adventure-content-gen/Cargo.lock, tools/paperclips-content-gen/Cargo.lock, tools/mkmanifest/Cargo.lock, tools/remote-session-client/Cargo.lock, tools/ringtap-viewer/Cargo.lock, capos-rt/Cargo.lock, capos-service/Cargo.lock, shell/Cargo.lock, libcapos/Cargo.lock, libcapos-posix/Cargo.lock, capos-wasm/Cargo.lock, fuzz/Cargo.lock
  - **Pinning status:** Each standalone workspace has its own lockfile. The Makefile DEPENDENCY_POLICY_MANIFESTS / DEPENDENCY_POLICY_LOCKFILES lists drive the gate.
  - **Drift-review status:** make dependency-policy-check runs the shared deny/audit baseline against every standalone manifest and lockfile listed above (root workspace Cargo.lock plus the 21 standalone lockfiles in this row). Cross-workspace version drift remains review-visible and intentional where lockfiles differ.
- **Input:** Standalone cargo dependencies (not yet under policy gates)
  - **Current source:** tools/remote-session-client/src-tauri/Cargo.lock, vendor/wasmi-no_std/wasmi-1.0.9/Cargo.lock
  - **Pinning status:** Two checked-in lockfiles fall outside DEPENDENCY_POLICY_LOCKFILES. tools/remote-session-client/src-tauri/Cargo.lock is the Tauri scaffold lockfile; make remote-session-tauri only exposes deterministic policy and check modes and reviewed dev mode – distributable package and desktop automation modes are blocked. vendor/wasmi-no_std/wasmi-1.0.9/Cargo.lock is part of the vendored upstream snapshot covered separately by the wasmi =1.0.9 path-dependency pin in capos-wasm/Cargo.toml.
  - **Drift-review status:** Both lockfiles are review-visible through ordinary diffs but are not run through cargo deny check / cargo audit today. Promoting either into DEPENDENCY_POLICY_LOCKFILES is gated on the matching authority decision (Tauri scaffold scope decision; wasmi refresh procedure in vendor/wasmi-no_std/VENDORED_FROM.md).
- **Input:** Cap'n Proto compiler
  - **Current source:** Makefile:12-80, tools/capnp-build/src/lib.rs, capos-config/build.rs, tools/check-generated-capnp.sh, tools/mkmanifest/src/lib.rs, tools/mkmanifest/src/main.rs
  - **Pinning status:** Official capnproto-c++-1.2.0.tar.gz source tarball URL, version, and SHA-256 are pinned in Makefile; make capnp-ensure builds $(CAPOS_TOOLS_ROOT)/capnp/1.2.0/bin/capnp under the per-user tool cache so linked worktrees reuse it. The build rule patches the distributed CLI version placeholder to the pinned version before compiling.
  - **Drift-review status:** The shared build helper defaults to the pinned path and rejects CAPOS_CAPNP when it points elsewhere. Make targets export the pinned path and CI persists it through $GITHUB_ENV. make generated-code-check verifies both the exact compiler path and Cap'n Proto version 1.2.0 before regenerating bindings through Cargo. mkmanifest cue-to-capnp also rejects missing or non-canonical CAPOS_CAPNP, checks Cap'n Proto version 1.2.0, and delegates schema-aware JSON-to-binary conversion to that pinned compiler.
- **Input:** Cap'n Proto Rust runtime/codegen crates

- **Current source:** `capos-config/Cargo.toml`, `kernel/Cargo.toml`, `tools/capnp-build/Cargo.toml`, `Cargo.lock`
- **Pinning status:** Cargo manifests use exact `capnp = "=0.25.4"` and `capnpc = "=0.25.3"` requirements where declared; lockfiles pin exact crate versions and checksums.
- **Drift-review status:** Security Verification Track S.10.3 now requires dependency-class and no_std review before these changes are accepted.

- **Input:** Kani verifier toolchain
  - **Current source:** `.github/workflows/ci.yml`, `Makefile`, `tools/run-kani-proofs.sh`, `tools/cloudbuild-kani.yaml`, `.gcloudignore`
  - **Pinning status:** GitHub CI pins `kani-verifier 0.67.0`; `cargo kani setup` installs the matching Kani bundle plus `nightly-2025-11-21-x86_64-unknown-linux-gnu` into the user-local Kani/rustup paths. Local `make kani-lib` and `make kani-dma-authority` expect a compatible `cargo-kani` install. The high-memory `make kani-lib-full` path uses Google Cloud Build image digest `rust@sha256:adab7941580c74513aa3347f2d2a1f975498280743d29ec62978ba12e3540d3a` on `E2_HIGHCPU_32`, installs rustup from `https://sh.rustup.rs`, sources `/usr/local/cargo/env`, initializes minimal git metadata for build tooling that expects a repository, then pins `nightly-2025-11-21` plus `cargo-kani 0.67.0`.
  - **Drift-review status:** The CI `kani-proofs` job installs `kani-verifier 0.67.0`, runs `cargo kani setup`, and executes the bounded `make kani-lib` harness list plus the DMA-authority `make kani-dma-authority` harness group. The Cloud Build config installs the same Kani version and runs `make kani-lib-full`; it depends on explicit source staging and logs in maintainer-private GCS buckets configured in `tools/cloudbuild-kani.yaml`, `.gcloudignore` secret exclusions, and account/project IAM for Cloud Build submission and the selected runtime service account. Version, image, worker, bucket, IAM, rustup bootstrap, synthetic git metadata, or setup-path changes are review-visible in the workflow, Cloud Build config, runner script, and this inventory.
- **Input:** Alloy Analyzer (DMA assurance model checker)
  - **Current source:** `Makefile` `ALLOY_VERSION`/`ALLOY_TARBALL_URL`/`ALLOY_TARBALL_SHA256`, `tools/run-dma-alloy-model.sh`, `models/dma/dma_authority.als`
  - **Pinning status:** Self-contained linux/amd64 Alloy Analyzer `6.2.0` app image (bundled Temurin JRE + native SAT solvers) pinned by SHA-256; `make alloy-ensure` downloads and verifies it into `$(CAPOS_TOOLS_ROOT)/alloy/6.2.0/` (the jar is not vendored). This slice owns the Alloy pin shared with the scheduler lease model track.
  - **Drift-review status:** `make model-dma-alloy` verifies the tarball SHA-256, checks the launcher reports version `6.2.0`, and runs the relational authority-graph checks/witnesses headless at scope `for 4`, failing on any counterexample or analyzer error. GitHub CI runs it in the `dma-assurance-models` job.
- **Input:** TLC model checker (DMA assurance lifecycle model)
  - **Current source:** `Makefile` `TLA_TOOLS_VERSION`/`TLA_TOOLS_JAR_URL`/`TLA_TOOLS_JAR_SHA256`/`TLA_JRE_URL`/`TLA_JRE_SHA256`, `tools/run-dma-tla-model.sh`, `models/dma/dma_authority.tla`

- **Pinning status:** `tla2tools.jar 1.7.4` (TLC 2.19) pinned by SHA-256 plus a SHA-256-pinned Temurin JRE `17.0.19+10` (the bare jar needs a JVM, unlike the self-contained Alloy app image); `make tla-ensure` downloads and verifies both into `$(CAPOS_TOOLS_ROOT)/tla/` (neither is vendored). This slice owns the TLC pin shared by the scheduler/IRQ TLA+ model tracks.
- **Drift-review status:** `make model-dma-tla` re-verifies the jar SHA-256 and the pinned JRE version, then runs TLC over the bounded `.cfg` (2 devices / 2 domains / 2 pages / 2 iovas, generations 0..1), failing closed on any invariant violation, deadlock, or analyzer error (exit code *and* the "No error" marker are both asserted). GitHub CI runs it in the `dma-assurance-models` job.

- **Input:** Generated capnp bindings
  - **Current source:** `capos-config/src/lib.rs:10-12`, `tools/generated/capos_capnp.rs`, `tools/check-generated-capnp.sh`
  - **Pinning status:** Generated into Cargo `OUT_DIR`; the expected patched output is checked in under `tools/generated/`.
  - **Drift-review status:** `make generated-code-check` regenerates the canonical `capos-config` output and fails if that output differs from the checked-in baseline or if kernel-generated output reappears.
- **Input:** no_std patching of generated bindings
  - **Current source:** `tools/capnp-build/src/lib.rs`, `capos-config/build.rs`, `tools/check-generated-capnp.sh`
  - **Pinning status:** One shared build-support crate asserts the patch anchor and injects the no_std imports after generation. `capos-config/build.rs` calls that helper as the single schema binding owner.
  - **Drift-review status:** `make generated-code-check` verifies the patched output contains the expected no_std imports and matches the checked-in baseline.
- **Input:** Generated adventure content
  - **Current source:** `demos/adventure-content/content/prototype.cue`, `tools/adventure-content-gen/`, `demos/adventure-content/src/generated.rs`, `tools/check-generated-adventure-content.sh`
  - **Pinning status:** Prototype mission content is authored in checked-in CUE and generated by a standalone locked Cargo host tool into a checked-in no_std Rust content blob. The checker requires the pinned CUE path under `$(CAPOS_TOOLS_ROOT)` and `cue version v0.16.0`.
  - **Drift-review status:** `make generated-code-check` runs `generated-adventure-content-check`, which exports the CUE source as JSON, runs `tools/adventure-content-gen` with `cargo run --locked`, formats the generated output, and fails on drift from the checked-in baseline.
- **Input:** Generated Paperclips content
  - **Current source:** `demos/paperclips-content/content/paperclips.cue`, `schema/paperclips-content.capnp`, `tools/paperclips-content-gen/`, `demos/paperclips-content/src/generated.rs`, `tools/check-generated-paperclips-content.sh`

- **Pinning status:** Paperclips game content is authored in checked-in CUE, schema-validated through the typed `PaperclipsContent` Cap'n Proto root, and generated by a standalone locked Cargo host tool into checked-in typed Cap'n Proto bytes embedded by a no_std Rust wrapper. The checker requires the pinned CUE path under `$(CAPOS_TOOLS_ROOT)`, `cue version v0.16.0`, and the pinned Cap'n Proto compiler path/version used for schema-aware conversion.
- **Drift-review status:** `make generated-code-check` runs `generated-paperclips-content-check`, which exports the CUE source as JSON, converts it through `mkmanifest cue-to-capnp` against `schema/paperclips-content.capnp`, runs `tools/paperclips-content-gen` with `cargo run --locked`, formats the generated output, and fails on drift from the checked-in generated content.

- **Input:** Userspace custom target
  - **Current source:** `targets/x86_64-unknown-capos.json`, `.cargo/config.toml`, `Makefile`, `system*.cue`
  - **Pinning status:** Source-controlled target specification plus Cargo aliases, Makefile build wrappers, and manifest paths for booted `init`, `demos`, `shell`, and `capos-rt` runtime builds. The target JSON uses Rust nightly custom-target support and builds `core,alloc` from `rust-src`.
  - **Drift-review status:** `make init-capos-build demos-capos-build shell-capos-build capos-rt-capos-build` verifies the userspace crates against `target_os = "capos"`; QEMU smokes embed `target/x86_64-unknown-capos/release` userspace artifacts.
- **Input:** Userspace runtime surface check
  - **Current source:** `tools/check-userspace-runtime-surface.sh`
  - **Pinning status:** Source-controlled script that treats `capos-rt` as the only owner of `_start`, panic, allocator, raw syscall, and entry-point macro definitions.
  - **Drift-review status:** Run directly when runtime or userspace entry code changes; it is not a QEMU transcript assertion and does not live inline in `Makefile`.
- **Input:** Linker script build scripts
  - **Current source:** `kernel/build.rs`, `init/build.rs`, `demos/*/build.rs`, `capos-rt/build.rs`, `capos-wasm/build.rs`
  - **Pinning status:** Source-controlled scripts and linker scripts. `capos-rt/build.rs` emits the runtime linker script for both the legacy `target_os = "none"` userspace build path and the booted custom `target_os = "capos"` path. `capos-wasm/build.rs` mirrors the same pattern for the wasm-host bin (Phase W.2 onward) and uses `cargo:rustc-link-arg-bins` so the linker script applies only to the bin and not the lib.
  - **Drift-review status:** Build rerun boundaries are explicit; generated link args are not independently audited.
- **Input:** CUE manifest compiler
  - **Current source:** `Makefile` `CUE_TARBALL_URL`/`CUE_TARBALL_SHA256`, `tools/mkmanifest/src/main.rs`, `tools/mkmanifest/src/lib.rs`, `.github/workflows/ci.yml`
  - **Pinning status:** `make cue-ensure` downloads the official `cue_v0.16.0_linux_amd64.tar.gz` release binary, verifies its SHA-256, extracts `cue` into `$(CAPOS_TOOLS_ROOT)/cue/0.16.0/`

bin/cue, and checks the reported version – the same download-and-verify pattern used for Typst and uv. CAPOS_TOOLS_ROOT defaults to $HOME/.capos-tools (per-user shared cache); operators may override it explicitly. This replaces the prior go install cuelang.org/go/cmd/cue, which compiled from source under a floating Go toolchain rather than verifying a pinned binary by hash.

- **Drift-review status:** Make exports CAPOS_CUE and CAPOS_TOOLS_ROOT to tools/mkmanifest, and CI records that exact path through $GITHUB_ENV before both the host-baseline cargo test-mkmanifest gate and QEMU smoke. mkmanifest::expected_cue_path derives the same per-user path, rejects missing or non-canonical CAPOS_CUE, and checks cue version v0.16.0 before export. The same path and version checks now gate both boot-manifest compilation and mkmanifest cue-to-capnp data-message conversion.

- **Input:** Default boot manifest defaults package
  - **Current source:** cue/defaults/defaults.cue, cue.mod/module.cue, system.cue, tools/mkmanifest/src/lib.rs
  - **Pinning status:** cue/defaults/defaults.cue declares package defaults and exports #DefaultSystem, the shared scaffold for the default boot manifest. cue.mod/module.cue pins module: "capos.local" with language v0.16.0. system.cue imports the defaults via capos.local/cue/defaults, declares package capos, and mkmanifest --package capos system.cue manifest.bin exports the unified package.
  - **Drift-review status:** make invokes mkmanifest with --package capos only when MANIFEST_SOURCE is system.cue; focused-proof system-*.cue manifests stay in single-file mode. The defaults package is a manifest-rule prerequisite, so edits trigger rebuilds.
- **Input:** Operator overlay surface
  - **Current source:** system.local.cue.example, system.local.cue (gitignored), .gitignore
  - **Pinning status:** The repo-root overlay file is system.local.cue (package capos); system.local.cue.example is the committed worked-example template. CUE's package mode unifies it with system.cue automatically.
  - **Drift-review status:** Operators copy the example, edit, and rebuild — system.local.cue is a wildcard-resolved manifest-rule prerequisite. The overlay is gitignored explicitly to avoid accidental commits of host-specific keys or principals.
- **Input:** Host-user manifest tag
  - **Current source:** Makefile, system.cue _user @tag(user) / _displayName @tag(displayName), tools/mkmanifest/src/lib.rs cue_export_args / cue_tags_from_env_values, target/.cue-tags.<manifest>
  - **Pinning status:** make run sets CAPOS_CUE_USER=$(USER). mkmanifest reads that structured account variable, derives displayName from the same account's first GECOS/comment field in /etc/passwd when CAPOS_CUE_DISPLAY_NAME is unset, and falls back to the account name when the passwd comment is unavailable. It also reads generic CAPOS_CUE_TAGS (and --tag key=value CLI repeats) and forwards each entry to cue export --inject; structured CAPOS_CUE_USER / CAPOS_CUE_DISPLAY_NAME override duplicate generic keys. The target/.cue-tags.<manifest-bin> sentinel records the active tag state via a FORCE-prereq rule that touches the file only when content differs, so a tag change invalidates the cached

manifest.bin; the recipe reads exported environment values at shell runtime rather than splicing tag text into shell syntax.
  - **Drift-review status:** The injected user value reaches the manifest via system.cue's _user: string \| *"operator" @tag(user) and surfaces as the default local operator seed account name; displayName reaches the seed account display name. Untagged system.cue keeps the operator account-name/display-name defaults, while focused demo and smoke manifests pin their own demo fixtures.
- **Input:** mdBook documentation tools
  - **Current source:** Makefile, book.toml
  - **Pinning status:** GitHub release assets for mdBook v0.5.0 and mdbook-mermaid v0.17.0 are pinned by version and SHA-256 under $(CAPOS_TOOLS_ROOT), which defaults to $HOME/.capos-tools. mdbook-mermaid supplies the pinned mermaid.min.js browser bundle used by both mdBook HTML rendering and docs-PDF Mermaid rasterization.
  - **Drift-review status:** make docs and make cloudflare-pages-build verify the tarball checksums and executable versions, refresh the Mermaid assets, and build target/docs-site.
- **Input:** Typst typesetter (paper and docs PDF builds)
  - **Current source:** Makefile TYPST_VERSION, papers/schema-as-abi/main.typ, docs/manual.typ
  - **Pinning status:** GitHub release asset for Typst v0.14.2 is pinned by version and SHA-256 under $(CAPOS_TOOLS_ROOT)/typst/0.14.2, mirroring the mdBook pinning pattern. typst-ensure verifies the tarball checksum and the binary's reported version before paper and docs-PDF targets invoke it. Bundled New Computer Modern font keeps builds reproducible across hosts.
  - **Drift-review status:** make paper rebuilds target/papers/schema-as-abi/main.pdf using the pinned Typst binary; make cloudflare-pages-build additionally publishes the PDF as target/docs-site/papers/schema-as-abi.pdf. make docs also uses Typst to compile the generated system manual PDF from docs/manual.typ plus per-page converted Markdown body content. Generated PDFs are not checked in; source main.typ, references.bib, docs/manual.typ, and documentation inputs are checked in.
- **Input:** Documentation PDF converter
  - **Current source:** Makefile UV_VERSION / MD2TYPST_VERSION, .node-version, package.json, package-lock.json, tools/md2typst-constraints.txt, tools/docs-bundle.js, tools/build-typst-manual.js, tools/mermaid-puppeteer-config.json, docs/manual.typ, docs/manual-overrides/*.typ
  - **Pinning status:** GitHub release asset for uv 0.11.8 (uv-x86_64-unknown-linux-gnu.tar.gz) is pinned by version and SHA-256 under $(CAPOS_TOOLS_ROOT)/uv/0.11.8. uv-ensure verifies the tarball checksum and the binary's reported version before PDF generation. uv tool run --constraints tools/md2typst-constraints.txt --from md2typst==0.3.3 md2typst pins the Markdown-to-Typst converter and its Python dependency set. Node version 22.16.0 is declared by .node-version and package.json; the current Makefile invokes node and npm from PATH, so host Node selection remains an operator/CI environment responsibility. package-lock.json pins @mermaid-js/mermaid-cli

and its Puppeteer dependency tree; `make mermaid-cli-ensure` runs `npm ci --ignore-scripts` with `PUPPETEER_SKIP_DOWNLOAD=1`, so Puppeteer's install script cannot fetch a browser during dependency installation. Mermaid rasterization uses the explicit `MERMAID_BROWSER_BIN` Chromium/Chrome executable, passes it to Puppeteer as `PUPPETEER_EXECUTABLE_PATH`, and renders PDF diagrams at `MERMAID_PDF_SCALE=3` by default; `tools/mermaid-puppeteer-config.json` disables the browser sandbox for local and gVisor build containers. `tools/docs-bundle.js` reads the explicit manual page list from `docs/manual.typ`, generates `target/docs-bundle/manual.md` plus one Markdown file per manual page, and `docs/manual.typ` owns the PDF title page, contents, page order, page styling, and override placeholders.

  - **Drift-review status:** `make docs-pdf` converts each generated manual Markdown page to Typst with md2typst, normalizes anchors and links with `tools/build-typst-manual.js`, uses any matching checked-in `docs/manual-overrides/<page-id>.typ` instead of the generated page, rasterizes Mermaid diagrams through the explicit browser executable, and compiles `target/docs-bundle/manual.pdf` with pinned Typst. `make docs` copies that generated PDF to `target/docs-site/manual.pdf` for Cloudflare Pages publication. Generated Markdown, Typst body pages, and PDF files are ignored build artifacts, not tracked source.
- **Input:** QEMU and firmware
  - **Current source:** `Makefile:85-96`, `tools/build-provenance.sh`, `.github/workflows/ci.yml` `qemu-smoke`
  - **Pinning status:** The `qemu-smoke` CI job installs `qemu-system-x86=1:8.2.2+ds-0ubuntu1.16` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04) and `ovmf=2024.02-2ubuntu0.8` (amd64, `noble-updates`/main, Ubuntu 24.04). OVMF delivers `/usr/share/ovmf/OVMF.fd` – the first entry in the Makefile's `OVMF_CODE_CANDIDATES` list, so the wildcard discovery resolves to that path on the pinned runner. The Makefile now also pins the selected OVMF firmware blob by SHA-256 (`OVMF_CODE_SHA256`) and gates the ISO and cloud-disk rules on `ovmf-verify`, which fails on hash drift and emits a NOTICE skip when no OVMF candidate is installed. Local boot verification still uses the host-installed `qemu-system-x86_64`.
  - **Drift-review status:** `make build-provenance` records the current QEMU version, selected executable path, package identity when discoverable, OVMF selected path or explicit absence, OVMF package identity when discoverable, and OVMF firmware hash when the configured firmware path exists. QEMU and OVMF are identified on the CI runner by package name, exact version, architecture, normalized apt source pocket, and selected path; the QEMU binary identity is captured via `dpkg-query`/`apt-cache policy` by `make build-provenance` per run and the OVMF firmware-blob SHA-256 is captured the same way. `make ovmf-verify` fails the build when the on-host OVMF firmware blob does not match the pinned `OVMF_CODE_SHA256`.
- **Input:** ISO and host filesystem tools
  - **Current source:** `Makefile:317-341`, `tools/build-provenance.sh`, `.github/workflows/ci.yml` `qemu-smoke`
  - **Pinning status:** The `qemu-smoke` CI job installs `xorriso=1:1.5.6-1.1ubuntu3` (amd64, `noble`/main, Ubuntu 24.04), `make=4.3-4.1build2` (amd64, `noble`/main, Ubuntu 24.04), and

git=1:2.43.0-1ubuntu7.3 (amd64, noble-updates/main or noble-security/main, Ubuntu 24.04). Local builds still use host-installed xorriso, sha256sum, git, make, and shell utilities.

  - **Drift-review status:** make build-provenance records selected executable paths and package identities when discoverable for xorriso, sha256sum, make, git, and related local build tools, plus final ISO hashes. xorriso, make, and git are identified on the CI runner by package name, exact version, architecture, normalized apt source pocket, and selected path; the per-run identity is captured via dpkg-query/apt-cache policy by make build-provenance. The remaining host tools, including sha256sum, shell, build-essential, and curl, remain host-provided or package-observed rather than repo-digest-pinned.
- **Input:** Boot manifest and embedded binaries
  - **Current source:** system.cue:1-144, tools/mkmanifest/src/lib.rs:339-379, tools/mkmanifest/src/main.rs, tools/build-provenance.sh, tools/compare-build-provenance.py, Makefile:168-169, Makefile:332-341
  - **Pinning status:** Source manifest is checked in; embedded ELF payloads are build artifacts or inline manifest bytes.
  - **Drift-review status:** Manifest validation checks references and path containment. make build-provenance now writes a local provenance record with runner OS/kernel/architecture identity, Rust toolchain details, selected host-tool paths and package identities when discoverable, hashes for the selected manifest, ISO, kernel, OVMF firmware when present, and every embedded binary reported by mkmanifest --print-binaries, including file-backed and inline payloads. make build-provenance-compare compares two retained records for material drift while ignoring generated timestamp and allowed local target/ or .capos-tools/ path-root movement.
- **Input:** Vendored upstream snapshots
  - **Current source:** vendor/wasmi-no_std/, vendor/dns-c-wahern/, vendor/fatfs-no_std/, vendor/rustls-webpki/, vendor/webpki-roots/, vendor/embedded-tls/, each with a VENDORED_FROM.md
  - **Pinning status:** Each vendored tree is a static, pinned snapshot recorded by version/tag, commit SHA when available, commit date when available, vendoring date, and license. vendor/wasmi-no_std/wasmi-1.0.9/ pins wasmi v1.0.9 (commit 61ba65e6563d8b2f5b699b018349d3330b28b9f3, Apache-2.0 OR MIT) consumed by capos-wasm/; vendor/dns-c-wahern/src/ pins William Ahern's dns.c rel-20160808 (commit 4ec718a77633c5a02fb77883387d1e7604750251, MIT). vendor/rustls-webpki/rustls-webpki-0.103.13/ pins rustls-webpki 0.103.13 (artifact SHA-256 61c429a8…f756e, commit 2879b2ce…728e86, ISC) and vendor/webpki-roots/webpki-roots-1.0.7/ pins webpki-roots 1.0.7 (artifact SHA-256 52f5ee44…2eb9d, commit be948464…221688, CDLA-Permissive-2.0); both are the certificates/TLS Phase-1 verifier deps consumed by the capos-tls/ Phase-1 verifier crate. vendor/embedded-tls/embedded-tls-0.19.0/ pins the embedded-tls 0.19.0 crates.io package (embedded VCS commit 865e1fd983c583228e3bbeb9f4996f1abc454ca3, Apache-2.0) consumed only by the local TLS client handshake smoke. No source patches (one integration-only empty-[workspace] marker per crate); each path dep carries an exact version = "=X.Y.Z" pin so cargo-deny's wildcards gate stays happy.

- **Drift-review status:** The wasmi snapshot is exercised by `make capos-wasm-build` and the WASI smokes plus `make dependency-policy-check` (cargo-deny + cargo-audit against `capos-wasm/Cargo.lock`). The `rustls-webpki` / `webpki-roots` snapshots are exercised by `capos-tls/` under `cargo build` / `cargo build --features qemu` (bare-metal `x86_64-unknown-none`) and by `make dependency-policy-check` against the root `Cargo.lock`. The `embedded-tls` snapshot is exercised by `make run-cloud-tls-client-handshake`, the focused capOS demo build, and `make dependency-policy-check` against `demos/Cargo.lock`. The dns.c snapshot is not yet on the v0 build path; `demos/posix-dns-resolver/` compiles only `main.c` with a commented-out `dns.h` include. Refreshes follow the procedure recorded in each `VENDORED_FROM.md`. No `vendor/dash/` source-build is present.

- **Input:** Build downloads
  - **Current source:** `Makefile`, Cargo lockfiles, `rust-toolchain.toml`
  - **Pinning status:** Limine, CUE, and documentation tool tarballs are explicitly fetched and SHA-256-verified; Cargo and rustup downloads are implicit when caches/toolchains are absent. The build no longer uses a Go toolchain: CUE is now a hash-verified release binary rather than a `go install` compile, so the `actions/setup-go` CI step and the floating `go-version` pin were removed.
  - **Drift-review status:** Limine artifacts, the CUE release binary, and documentation tool tarballs are verified by SHA-256. Cargo downloads rely on upstream tooling and lockfiles, with no separate repo policy beyond the lockfile checksums. Rustup downloads are now gated by the dated `nightly-2026-04-20` channel pin (see Rust Toolchain section); only the dist tarballs themselves are not yet mirrored.
- **Input:** GitHub Actions identities and runner OS
  - **Current source:** `.github/workflows/ci.yml`
  - **Pinning status:** Every third-party Action is pinned by 40-character commit SHA with a trailing `# v<X.Y.Z>` comment marker. The runner OS is pinned to `ubuntu-24.04` rather than the floating `ubuntu-latest` label.
  - **Drift-review status:** Pin bumps are review-visible as workflow diffs and the trailing version comment makes the intended release auditable. See the GitHub Actions Runner and Workflow Pinning section below for the current pin table and the bump procedure.

### Security Verification Track S.10.3 Dependency Policy

Dependency changes are accepted only if they satisfy this policy and are recorded in the owning task checklist.

### Dependency classes

Use these classes when reviewing a dependency change:

- **Kernel-critical no_std:** crates used directly by `kernel`, `capos-lib`, `capos-config`, and `capos-abi`.
- **Userspace-runtime no_std:** crates used by `init`, `demos`, and `capos-rt`.
- **Host/build:** crates used by `tools/*`, `build.rs` helpers, and generated output pipelines.
- **Test/fuzz/dev:** crates gated by `dev-dependencies` or `target-specific` for fuzz/proptests/smoke support.

**Required pre-merge criteria**

For any added dependency (or bump in any class):

1. **Manifest and features are explicit.** Dependency entries must include explicit feature choices; avoid `default-features = true` unless justified.
2. **No_std compatibility is proven for no_std classes.** Kernel-critical and userspace-runtime dependencies must compile in a `#![no_std]` mode with `alloc` where expected. `cargo build -p <crate> --target x86_64-unknown-none` must succeed for every kernel/no_std crate affected.
3. **Security policy checks run and pass.** CI-equivalent checks for the touched workspace are required through `make dependency-policy-check`, which runs `cargo deny check` on every Cargo manifest and `cargo audit` on every lockfile.
4. **Dependency class change is justified in review.** PR text must include target class, ownership rationale, transitive graph impact, and why the crate is not a transitive replacement for an already-allowed dependency.
5. **Lockfile behavior is explicit.** Update only intended lockfiles and record intentional cross-workspace drift in this document if workspace purpose differs.

**No_std add/edit checklist {#trusted-build-inputs-no_std-addedit-checklist}**

- Reject crates that require `std`, OS I/O, or unsupported platform APIs in the dependency path intended for kernel classes.
- Reject dependencies that re-export broad platform facades or large unsafe surface unless there is a replacement with smaller scope and better audit visibility.
- Record a license and supply-chain review result (via policy checks) before merge.
- Confirm no `unsafe` contract escapes are added without a review surface note in the relevant module.

**Standing requirements**

- Add Security Verification Track S.10.3 checks to the target branch plan item for any kernel/no_std crate dependency change and document the exact pass command set.
- Keep lockfile deltas review-visible in normal PR flow; lockfile pinning is the minimum bar, not the gate.
- Keep transitive drift in sync with the trust class: class-wide divergence across lockfiles requires explicit justification.

**Remaining gaps after Security Verification Track S.10.3 policy**

- Mirror the resolved dated nightly dist tarballs (and their SHA-256 checksums) into the per-user tool cache as a further hardening step, so bumping the pin does not depend on rustup retaining its historical manifests. The dated pin closes the floating-channel gap; tarball mirroring would close the historical-availability gap.
- Decide whether the local `make kani-lib` workflow should grow a repo-managed installer/bootstrap helper or continue to rely on separately provisioned user-local `cargo-kani` plus the Kani bundle/toolchain setup path.

- CI now publishes `target/build-provenance.txt` as a named artifact on every `qemu-smoke` run (see `actions/upload-artifact` step in `.github/workflows/ci.yml`) and, on `pull_request` events, downloads the most recent successful main-branch artifact via `actions/download-artifact` and runs `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment` against it as a blocking PR gate. Missing base provenance is a CI failure, not a silent skip; artifact retention is therefore part of the gate.

**Build Provenance Retention And Comparison Policy**

Status 2026-06-07 06:35 UTC: this policy applies to local and CI proof artifacts produced by `make build-provenance`. The `qemu-smoke` CI job now publishes the candidate record as a named artifact on every run and, on `pull_request` events, runs `make build-provenance-compare` against the most recent successful main-branch artifact with `BUILD_PROVENANCE_COMPARE_POLICY=ci-environment`. That CI policy is PR-blocking for runner, tool, Rust, OVMF package, and OVMF hash drift while allowing expected base-vs-head source commit, ISO/kernel/manifest hash, and embedded-payload hash differences. This remains a reproducibility evidence policy, not a claim that production images are third-party reproducible before the unresolved pinning gates below are closed.

Package-pin bumps for `qemu-system-x86`, `xorriso`, `make`, `git`, or `ovmf` are the only planned baseline-refresh case where the PR comparison can fail on purpose: the candidate provenance records the new reviewed package identity, while the base-branch artifact still records the old one. That failure is not a green PR exception and is not a workflow bypass. The bump can land only through a reviewed local-main integration or maintainer push path after the branch's `qemu-smoke` build, `make run-smoke`, and `make build-provenance` steps pass with the new pins, and the compare diff contains only the reviewed package-identity changes introduced by the same branch. After the bump lands, the next successful main-branch `qemu-smoke` push artifact becomes the refreshed base provenance; unrelated PRs must wait for that artifact before their blocking environment comparisons can pass.

For every externally cited QEMU proof, release candidate, paper artifact, or public performance/security claim, retain the following as one immutable evidence bundle:

- `target/build-provenance.txt` from the exact checked commit, manifest, and recorded worktree state;
- the kernel, manifest, ISO, OVMF firmware if used, and embedded-binary hashes recorded in that provenance file;
- the exact command set and QEMU transcript or host-test log used as evidence;
- the source commit hash, clean-tree assertion or retained `git diff` plus untracked-file inventory, and any non-default Make variables such as `MANIFEST_SOURCE`, `CAPOS_CUE_TAGS`, `QEMU_NET`, `MERMAID_BROWSER_BIN`, or `CAPOS_TOOLS_ROOT`;
- the `system.local.cue` overlay state for default-manifest builds: record explicit absence, or retain the file content plus SHA-256 and size. Because the overlay is gitignored and unified into `system.cue`, a commit hash alone is not enough to reconstruct a default-manifest build that used it;

- the runner identity: either a pinned CI/container image digest, or the host package identities for Rust, QEMU, xorriso, make, git, OVMF firmware package, and the operating-system image when the runner is not pinned.

Retention requirements:

- Keep evidence bundles for any tagged release, published paper result, public benchmark, or public security claim for at least the lifetime of that claim.
- Keep pre-merge task evidence until the reviewed branch has merged and the next full relevant verification has superseded it.
- Keep failed evidence when it explains a known regression, review finding, or release blocker; otherwise failed local scratch logs may be discarded.
- Do not rely on `target/` as the retention store. `target/` artifacts are local build output; retained evidence must be copied to the release, CI, or paper artifact store that owns the claim.

Comparison requirements:

- Run local comparisons with `make build-provenance-compare BASE_PROVENANCE=... CANDIDATE_PROVENANCE=...` or `tools/compare-build-provenance.py BASE CANDIDATE`. The command exits zero only when records differ by generated timestamp and allowed local path roots such as worktree `target/` or `.capos-tools/`, while all hashes, versions, package identities, and runner identities match.
- Run PR base-vs-head environment comparisons with `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment BASE_PROVENANCE=... CANDIDATE_PROVENANCE=...`. This policy compares the default manifest source, host target, runner identity, GitHub-hosted image identity when present, Rust toolchain, selected executable identities, tool versions, OVMF selection, and OVMF hash, but ignores expected source commit, kernel/manifest/ISO hash, and embedded-binary hash changes between the base branch and PR head.
- For package-pin bump branches, treat a `ci-environment` comparison failure as acceptable review evidence only when every reported difference is an intended package-identity change for `qemu-system-x86`, `xorriso`, `make`, `git`, or `ovmf` from the same branch. Any runner image, Rust toolchain, OVMF firmware hash, tool-version, or unrelated package drift remains blocking.
- Compare two provenance records by commit, clean or retained-diff state, `system.local.cue` absence/hash/content policy, manifest source, manifest binary hash, kernel hash, ISO hash, embedded-binary table, OVMF hash or explicit absence, host-tool versions, package identities, and operating-system image identity.
- A byte-identical ISO requires all recorded hashes to match. Equal source commits with different Rust, QEMU, xorriso, OVMF, or host package identities are compatible proof reruns, not reproducible-production evidence.
- If a comparison differs only in paths under `.capos-tools` or worktree-local `target/` directories while all hashes and versions match, treat the result as the same proof environment.
- If a comparison differs in worktree state, overlay state, package identity, operating-system image identity, host-tool version, OVMF hash, Rust compiler commit/date, embedded-binary

hash, or ISO hash, record the difference in the owning review or release note before citing the result.

Minimum runner identity for production-hardening branches:

- Rust must be a date-pinned nightly or stronger hash-pinned toolchain, not the floating `nightly` channel.
- QEMU, xorriso, make, and git must come from a pinned runner image digest or a documented package set with package name, version, architecture, repository, and distribution release.
- OVMF firmware must be either repo-pinned by digest or identified by package name, version, architecture, repository, distribution release, selected path, and SHA-256.
- Any runner image used for production reproducibility claims must be cited by immutable digest. Mutable tags are acceptable only for local proof evidence.

Production hardening must treat the following as unresolved supply-chain gates, not as cosmetic reproducibility work:

```
immutable runner image digest or repo-managed tool digests for qemu/
xorriso/make/git
```

The Rust nightly date pin (currently `nightly-2026-04-20`) closes the floating-channel gate; tarball mirroring is tracked as a further hardening step in the Remaining gaps section above. The `qemu-smoke` job now installs `qemu-system-x86=1:8.2.2+ds-0ubuntu1.16` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04), `xorriso=1:1.5.6-1.1ubuntu3` (amd64, `noble`/main, Ubuntu 24.04), `make=4.3-4.1build2` (amd64, `noble`/main, Ubuntu 24.04), `git=1:2.43.0-1ubuntu7.3` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04), and `ovmf=2024.02-2ubuntu0.8` (amd64, `noble-updates`/main, Ubuntu 24.04) so the QEMU, ISO writer, make, git, and OVMF firmware identities are all captured for UEFI smoke builds: package name, exact version, architecture, normalized apt source pocket, and the per-run identity captured via `dpkg-query`/`apt-cache policy` by `make build-provenance`. OVMF additionally records the selected path (`/usr/share/ovmf/OVMF.fd`) and per-run SHA-256, and the Makefile now pins the selected firmware blob by SHA-256 through the `ovmf-verify` gate wired into the ISO and cloud-disk rules. Repo-pinned digests (download-and-verify rather than apt-installed) for `qemu-system-x86`, `xorriso`, `make`, and `git`, or an immutable runner image digest that contains them, remain future hardening tracked in `docs/design-risks-register.md` (R13). `xorriso` has no version in `noble-updates`; the pin uses the only available `noble`/main version, which is what every Ubuntu 24.04 host resolves to.

Until those gates land, generated ISO/manifest/payload artifacts plus `target/build-provenance.txt` are suitable for local and CI proof evidence, but not for claims that a third party can reproduce an identical production boot image from source alone.

### Bootloader and ISO Inputs

The Makefile now pins Limine at commit `aad3edd370955449717a334f0289dee10e2c5f01` and verifies these copied artifacts:

| Artifact | Checksum reference |
| --- | --- |
| $(LIMINE_DIR)/limine-bios.sys | LIMINE_BIOS_SYS_SHA256 in Makefile |
| $(LIMINE_DIR)/limine-bios-cd.bin | LIMINE_BIOS_CD_SHA256 in Makefile |
| $(LIMINE_DIR)/limine-uefi-cd.bin | LIMINE_UEFI_CD_SHA256 in Makefile |
| $(LIMINE_DIR)/BOOTX64.EFI | LIMINE_BOOTX64_EFI_SHA256 in Makefile |

$(LIMINE_DIR) resolves to $(CAPOS_TOOLS_ROOT)/limine/<LIMINE_COMMIT> (default $HOME/.capos-tools/limine/<commit> unless CAPOS_TOOLS_ROOT is overridden), shared with the rest of the per-user pinned tool cache.

make limine-ensure clones https://github.com/limine-bootloader/limine.git only when $(LIMINE_DIR)/.git is absent, fetches the pinned commit if needed, checks it out detached, and runs make inside the Limine tree (the limine-ensure recipe). make limine-verify then checks the repository HEAD and artifact checksums (the limine-verify recipe). The ISO copies the kernel, generated manifest.bin, Limine config, and verified Limine artifacts into iso_root/, runs xorriso, then runs limine bios-install (the $(ISO) recipe).

Remaining reproducibility gap: Limine source is pinned, but the Limine build host compiler and environment are not pinned or recorded.

### Rust Toolchain

rust-toolchain.toml specifies:

- channel = "nightly-2026-04-20"
- targets = ["x86_64-unknown-none", "aarch64-unknown-none", "wasm32-wasip1"]
- components = ["rust-src"]

The wasm32-wasip1 target is needed for the WASI Preview 1 demo payloads (demos/wasi-hello-rust/, demos/wasi-cli-args/, demos/wasi-random/) built by make wasi-hello-rust-build, make wasi-cli-args-build, and make wasi-random-build; the wasm-host binary itself is built for the booted x86_64-unknown-capos userspace target instead.

The pinned dated channel resolves to:

- rustc 1.97.0-nightly (e22c616e4 2026-04-19)
- host target x86_64-unknown-linux-gnu

The 2026-04-20 manifest packages the rustc commit cut on 2026-04-19; that is the upstream dist naming convention, not a drift. Rustup will continue to install the same dist tarball for nightly-2026-04-20 as long as upstream retains it.

The Makefile derives HOST_TARGET from rustc -vV (Makefile:12) and uses that for tools/mkmanifest (Makefile:28-29). Cargo aliases in .cargo/config.toml:4-48 hard-code x86_64-unknown-linux-gnu for host tests. The custom userspace target aliases in .cargo/config.toml use targets/x86_64-unknown-capos.json plus -Zjson-target-spec and -Zbuild-std=core,alloc, so rust-src is a required toolchain component. The CI host-baseline and qemu-smoke jobs install the same nightly-2026-04-20 toolchain so CI matches the local rust-

toolchain.toml resolution. The kani-proofs job stays on nightly-2025-11-21 because Kani requires its own paired nightly bundle installed by cargo kani setup; advancing the Kani pin is tracked separately through that bundle's compatibility matrix.

**Rust Nightly Date Pin Policy**

The pin is one of the supply-chain-trust controls listed in this proposal alongside the Limine commit, OVMF firmware SHA-256, capnp tarball SHA-256, CUE binary, mdBook/mdbook-mermaid release assets, Typst binary, uv binary, and pinned cargo-deny/cargo-audit/cargo-kani releases. All of these must be pinned at the same trust level – date- or hash-anchored, never a floating channel or moving tag. This subsection states the policy for the Rust nightly entry; the next subsection states the mechanical advance procedure.

**Where the pin lives.** Exactly one source: rust-toolchain.toml as a date-anchored nightly channel of the form nightly-YYYY-MM-DD. The CI workflow's host-baseline and qemu-smoke toolchain: values must mirror that same dated channel. No other file may declare a nightly date; no float, no nightly shorthand, no commit-hash override.

**Promotion criteria.** A bump is accepted only when the candidate nightly satisfies all of the following against the worktree where the pin lands:

- make builds the full workspace clean (kernel + standalone userspace + ISO) with no new warnings under cargo build --features qemu.
- make fmt-check passes across the workspace and all standalone crates.
- make workflow-check passes (CLAUDE.md token budget, mandatory-context budgets, slice trailers).
- make check passes (the aggregate build/test gate that includes generated-code-check and the host-test aliases).
- make run-smoke passes on the developer host when QEMU smoke is feasible there; if QEMU is unavailable locally, the bump branch's CI qemu-smoke run is the authoritative gate.
- Any new rustc warning, lint, or unrelated build failure introduced by the new nightly is treated as a real gate failure. Do not relax capOS code or silence the lint to land the bump.

**Rollback.** If a promotion exposes a regression in a downstream crate that capOS depends on (limine, x86_64, spin, smoltcp, wasmi, capnp/capnpc, or any cargo-deny/cargo-audit pinned tool), revert the pin to the prior dated channel on main, file a tracking note in docs/tasks/ with the failing date, the failing crate, and the upstream issue if one exists, and resume normal cadence only after the downstream regression is resolved or worked around.

**Cadence.** Bump the pin at least once per quarter even without a specific feature trigger so production-provenance evidence does not lag upstream. Bump out of cadence when (a) a security advisory affects the current pinned nightly's rustc/cargo/std (consult rust-lang/rust, rustsec/advisory-db, and the cargo-audit output for the pinned dist), or (b) a compiler feature, fix, or lint that capOS depends on lands upstream. Unbounded float is not permitted: the dated channel must always resolve to a concrete YYYY-MM-DD.

**Approvals.** Maintainer-driven, single reviewed slice per bump. No automated promotion bot. The pin bump is its own contract change and must not be bundled with unrelated behavior changes; the reviewed diff must show only rust-toolchain.toml, the CI workflow, this

proposal's summary table and resolved-`rustc` line, and any minimal lint/code adjustments forced by the new nightly with an inline justification.

**Trust-input dimension.** The pin closes the floating-channel supply- chain gate listed in the Build Provenance Retention And Comparison Policy ("Minimum runner identity for production-hardening branches: Rust must be a date-pinned nightly or stronger hash-pinned toolchain, not the floating `nightly` channel"). Mirroring the resolved dist tarballs into the per-user tool cache (the same shape as Limine, capnp, CUE, mdBook, and Typst pins) remains a future hardening step tracked in the Remaining gaps section.

**Advance procedure (bumping the dated nightly)**

When to bump:

- A compiler feature, fix, or lint that capOS depends on lands in upstream nightly after `2026-04-20`. Example triggers: a Cargo or rustc fix that unblocks a build path; a `core`/`alloc` change that affects `-Zbuild-std`; a `rustfmt` change required for the project formatting baseline.
- Toolchain drift hygiene: schedule a bump at least once per release window even without a specific feature trigger, so production-provenance evidence does not lag too far behind upstream.

How to bump:

1. Choose a candidate nightly date and verify all required targets and the `rust-src` component are simultaneously available for that date:

   ```
   rustup toolchain add nightly-<YYYY-MM-DD> \
       --target x86_64-unknown-none \
       --target aarch64-unknown-none \
       --target wasm32-wasip1 \
       --component rust-src
   ```

   If any target or component is missing, try adjacent dates (rustup's nightly dist manifests sometimes drop a target for a single day) until one is found that provides the full set.
2. Update both files in the same commit:
   - `rust-toolchain.toml` `channel` value.
   - `.github/workflows/ci.yml` – both the `host-baseline` and `qemu-smoke` `toolchain:` values. Leave `kani-proofs` on its own pin.
3. Run the full local gate set against the candidate before pushing: `make fmt-check`, `cargo build --features qemu`, `make check`, `make workflow-check`, `make run-smoke`. Treat any new warning or unrelated build failure as a real gate failure – do not patch around compiler drift by relaxing capOS code.
4. Update the `Rust toolchain` row in this file's summary table, the resolved `rustc` line above, and the `last_reviewed` front-matter timestamp. Cite the new dated channel.
5. Land the pin bump as its own reviewed slice, not bundled with unrelated behavior changes. The pin is itself the provenance contract.

Remaining reproducibility gap: rustup retains nightly dist manifests for a finite window. A future hardening slice may mirror the resolved dist tarballs plus their SHA-256 checksums into the per-user tool cache the same way Limine and capnp are pinned today, so a bump-without-mirror does not become a silent loss of historical reproducibility.

### CI Runner Package Pins

The `qemu-smoke` CI job installs `qemu-system-x86`, `xorriso`, `make`, `git`, and `ovmf` via apt on an `ubuntu-24.04` runner. Those packages provide the QEMU emulator that executes every QEMU smoke, the ISO writer that builds the bootable image consumed by smokes, the build and repository tools used after checkout, and the UEFI firmware blob selected by `make run-uefi` and the cloud-disk path. A floating apt install (no `=<version>` specifier) would let upstream Ubuntu silently roll any of them on the next CI run, so this section names the version pins, the file that owns them, and the procedure for advancing them.

### CI Package Pin Policy

The pin is one of the supply-chain-trust controls listed in this proposal alongside the Limine commit, OVMF firmware SHA-256, Rust nightly date pin, capnp tarball SHA-256, CUE binary, mdBook/mdbook-mermaid release assets, Typst binary, uv binary, and pinned `cargo-deny`/`cargo-audit`/`cargo-kani` releases. All of these must be pinned at the same trust level – date- or hash-anchored, never a floating channel or moving tag. This subsection states the policy for the QEMU, xorriso, make, git, and OVMF package entries; the next subsection states the mechanical advance procedure.

**Where the pin lives.** Exactly one source: the `Install boot smoke dependencies` step of the `qemu-smoke` job in `.github/workflows/ci.yml`. Each package must be invoked as `<name>=<exact-version>` (no `*` wildcard and no major-only floor) so the apt resolver fails closed rather than silently rolling forward. The currently pinned versions are `qemu-system-x86=1:8.2.2+ds-0ubuntu1.16` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04), `xorriso=1:1.5.6-1.1ubuntu3` (amd64, `noble`/main, Ubuntu 24.04), `make=4.3-4.1build2` (amd64, `noble`/main, Ubuntu 24.04), `git=1:2.43.0-1ubuntu7.3` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04), and `ovmf=2024.02-2ubuntu0.8` (amd64, `noble-updates`/main, Ubuntu 24.04). The summary table, the `QEMU and firmware` and `ISO and host filesystem tools` rows, the `Host Tools` section, and the Build Provenance Retention And Comparison Policy mirror these strings; the policy text is the single source of truth and the other locations track it.

**Promotion criteria.** A bump is accepted only when all of the following hold against the bump branch:

- The Ubuntu base image rolls (`noble`/`noble-updates`/`noble-security` publishes a newer version of the package) or a security advisory affects the currently pinned version. Cosmetic version bumps without an upstream trigger are not accepted; the pin moves forward when there is a reason to move it.
- `apt-cache madison <package>` on a current Ubuntu 24.04 host lists the candidate version, and the candidate is available from `noble-updates`/main (or `noble`/main when no `noble-updates` entry exists, as is the case for `xorriso` today). Versions sourced from third-party PPAs or `*-proposed` pockets are not accepted.

- The bump branch's `qemu-smoke` execution reaches and passes the new-pin build evidence steps: `make build`, `make run-smoke`, `make build-provenance`, and candidate provenance artifact upload. The pull-request `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment` step is expected to fail only on reviewed package-identity fields that match the new pinned strings rather than the previous ones; every other comparison difference remains blocking.
- No new QEMU, xorriso, make, git, or OVMF behavior is silently relied on: if the bump unlocks a smoke that previously failed, that smoke must be enabled and reviewed in the same bump branch rather than treated as incidental.
- The `Trusted Build Inputs` summary table, `QEMU and firmware` row, `ISO and host filesystem tools` row, `Host Tools` section, and Build Provenance Retention And Comparison Policy text are updated to cite the new versions and the new resolved repository (`noble/noble-updates`) in the same commit.

**Rollback.** If a promotion exposes a regression in the QEMU smoke path, the ISO writer, build orchestration, repository operations, or UEFI boot, revert the `.github/workflows/ci.yml` change to the prior pinned version on `main`, file a tracking task under `docs/tasks/` with the failing version, the failing smoke, and the upstream Ubuntu/QEMU/xorriso/OVMF issue if one exists, and resume normal cadence only after the regression is resolved or worked around. Reverting also requires reverting the summary-table and policy text mirrors so the recorded versions stay consistent with the workflow file.

**Cadence.** Bump the pins at least once per quarter even without a specific security trigger, so production-provenance evidence does not lag upstream Ubuntu point releases. Bump out of cadence when (a) a security advisory affects the current pinned version of any of the packages (consult the Ubuntu Security Notices, the QEMU security mailing list, the Git security advisories, GNU make release notes, and the `ovmf/edk2` advisories), or (b) a fix that capOS depends on lands in a newer Ubuntu point release. Unbounded float is not permitted: each package must always resolve to a concrete `<epoch>:<upstream>-<debian>` version string.

**Approvals.** Maintainer-driven, single reviewed slice per bump. No automated promotion bot. The pin bump is its own contract change and must not be bundled with unrelated behavior changes; the reviewed diff must show only `.github/workflows/ci.yml`, this proposal's summary table and resolved-version mirrors, the relevant production-provenance task record when sub-items move, and any minimal smoke adjustments forced by the new package versions with an inline justification.

**Trust-input dimension.** The pin closes the runner/OS/tool identity gate listed in the Build Provenance Retention And Comparison Policy ("Minimum runner identity for production-hardening branches: QEMU, xorriso, make, and git must come from a pinned runner image digest or a documented package set with package name, version, architecture, repository, and distribution release") for the apt-installed package set it owns. A pinned runner image digest (replacing the `ubuntu-24.04` mutable label with an immutable image SHA) or repo-managed tool digests for those packages remain future hardening tracked in `docs/design-risks-register.md` (R13).

**Advance procedure (bumping the apt-pinned versions)**

When to bump:

- An Ubuntu Security Notice affects the currently pinned version of `qemu-system-x86`, `xorriso`, `make`, `git`, or `ovmf`.
- A QEMU, xorriso, make, git, or OVMF point release lands in `noble-updates`/main that capOS needs (typically a virtio, MSI-X, ISO writer, build-tool, repository-tool, or UEFI fix).
- Quarterly hygiene cadence with no specific feature trigger, so the pin does not lag too far behind upstream.

How to bump:

1. On a current Ubuntu 24.04 host (or a `ubuntu:24.04` container that has refreshed `apt-get update`), list available versions of each package:

   ```
   apt-cache madison qemu-system-x86
   apt-cache madison xorriso
   apt-cache madison make
   apt-cache madison git
   apt-cache madison ovmf
   ```

   Pick the highest stable version from `noble-updates`/main. If a package has no `noble-updates` entry (as is the case for `xorriso` today), pick from `noble`/main. Do not select from `*-proposed`, `*-backports`, or third-party PPAs.
2. Update the single source in the `Install boot smoke dependencies` step of the `qemu-smoke` job in `.github/workflows/ci.yml` so each package line reads <name>=<exact-version>.
3. Update the mirrors in this file in the same commit: the summary-table rows for `QEMU and firmware` and `ISO and host filesystem tools`, the `Host Tools` section, the Build Provenance Retention And Comparison Policy text, and the `Remaining gaps for Security Verification Track S.10.2/S.10.3` block under `Manifest, Embedded Binaries, and Downloaded Artifacts`. Refresh the `last_reviewed` front-matter timestamp.
4. If the OVMF package version moves, the OVMF firmware blob SHA-256 may change. Recompute `OVMF_CODE_SHA256` in `Makefile` from the resolved firmware path (`/usr/share/ovmf/OVMF.fd` on Ubuntu 24.04) and verify `make ovmf-verify` passes against the new digest. Land the `OVMF_CODE_SHA256` change in the same commit as the package bump.
5. Push the bump branch and let `qemu-smoke` exercise the new pins through `make`, `make run-smoke`, `make build-provenance`, and candidate provenance artifact upload. The acceptance gate for the bump itself is those steps passing plus a reviewed PR `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment` failure whose diff is limited to the intended package-identity strings replacing the previous ones. Land the bump through the reviewed local-main integration or maintainer push path; after it reaches `main`, the next successful main-branch `qemu-smoke` push artifact is the new base record for unrelated PR comparisons.
6. Land the pin bump as its own reviewed slice, not bundled with unrelated behavior changes. The pin is itself the provenance contract.

Remaining reproducibility gap: the `ubuntu-24.04` runner label is still managed by GitHub Actions, not by an immutable image digest, so the host package set underneath the apt-installed `qemu-system-x86`, `xorriso`, `make`, `git`, and `ovmf` pins can still roll between runs. A future hardening slice may move the `qemu-smoke` job to a self-built runner image referenced by digest, mirror the apt package files into the per-user tool cache the same way Limine and capnp are pinned today, or both, so a bump-without- mirror does not become a silent loss of historical reproducibility.

### Cargo Dependencies

The root workspace members are `capos-abi`, `capos-config`, `capos-lib`, `capos-tls`, `kernel`, and the host-only `tools/capnp-build` build-support crate. `Cargo.toml` keeps default members to `capos-config`, `capos-lib`, `capos-tls`, and `kernel` so ordinary root bare-metal builds do not build the host helper as a target package but do build the `capos-tls` certificates/TLS verifier-dependency probe. The vendored `rustls-webpki` / `webpki-roots` path dependencies declare their own `[workspace]` and are listed in the root `Cargo.toml` `exclude` set (the same isolation as the vendored `fatfs` crate), so they are not workspace members. The vendored `embedded-tls` client-state machine snapshot follows the same workspace isolation and is consumed only by the standalone `demos/` workspace. `init/`, `demos/`, `tools/mkmanifest/`, `tools/ringtap-viewer/`, `capos-rt/`, `shell/`, `libcapos/`, `libcapos-posix/`, `capos-wasm/`, and `fuzz/` are standalone workspaces with their own lockfiles.

Important direct dependencies and current root-lock resolutions:

- **Dependency:** `capos-abi`
  - **Manifest references:** `capos-config/Cargo.toml`, `capos-lib/Cargo.toml`
  - **Root lock resolution:** local path package in `Cargo.lock`
- **Dependency:** `argon2`
  - **Manifest references:** `capos-lib/Cargo.toml`; optional `capos-config/Cargo.toml` `credential-validation` feature used by kernel/init/mkmanifest bootstrap validation
  - **Root lock resolution:** `0.5.3` in `Cargo.lock`
- **Dependency:** `capnp`
  - **Manifest references:** `capos-config/Cargo.toml`, `capos-lib/Cargo.toml`, `kernel/Cargo.toml`
  - **Root lock resolution:** `0.25.4` in `Cargo.lock`
- **Dependency:** `capos-capnp-build`
  - **Manifest references:** `capos-config/Cargo.toml`
  - **Root lock resolution:** local path package in `Cargo.lock`
- **Dependency:** `capnpc`
  - **Manifest references:** `tools/capnp-build/Cargo.toml`
  - **Root lock resolution:** `0.25.3` in `Cargo.lock`
- **Dependency:** `limine` crate
  - **Manifest references:** `kernel/Cargo.toml:8` ("`0.6`" range)
  - **Root lock resolution:** `0.6.3` in `Cargo.lock`

- **Dependency:** `spin`
  - **Manifest references:** `kernel/Cargo.toml:9` ("`0.9`" range)
  - **Root lock resolution:** `0.9.8` in `Cargo.lock`
- **Dependency:** `x86_64`
  - **Manifest references:** `kernel/Cargo.toml:10` ("`0.15`" range)
  - **Root lock resolution:** `0.15.4` in `Cargo.lock`
- **Dependency:** `linked_list_allocator`
  - **Manifest references:** `kernel/Cargo.toml:11` ("`0.10`" range)
  - **Root lock resolution:** `0.10.6` in `Cargo.lock`
- **Dependency:** `smoltcp`
  - **Manifest references:** `kernel/Cargo.toml:16` ("`0.13.0`" caret range)
  - **Root lock resolution:** `0.13.0` in `Cargo.lock`
- **Dependency:** `loom`
  - **Manifest references:** `capos-config/Cargo.toml:27`
  - **Root lock resolution:** `0.7.2` in `Cargo.lock`
- **Dependency:** `proptest`
  - **Manifest references:** `capos-lib/Cargo.toml`
  - **Root lock resolution:** `1.11.0` in `Cargo.lock`
- **Dependency:** `rustls-webpki` (vendored path)
  - **Manifest references:** `capos-tls/Cargo.toml` (=`0.103.13`, `default-features = false`, `alloc`)
  - **Root lock resolution:** local path package (`vendor/rustls-webpki/rustls-webpki-0.103.13`) in `Cargo.lock`
- **Dependency:** `webpki-roots` (vendored path)
  - **Manifest references:** `capos-tls/Cargo.toml` (=`1.0.7`, `default-features = false`)
  - **Root lock resolution:** local path package (`vendor/webpki-roots/webpki-roots-1.0.7`) in `Cargo.lock`
- **Dependency:** `rustls-pki-types`
  - **Manifest references:** transitive of the vendored `rustls-webpki`/`webpki-roots` (`alloc`)
  - **Root lock resolution:** `1.14.1` in `Cargo.lock`
- **Dependency:** `untrusted`
  - **Manifest references:** transitive of the vendored `rustls-webpki`
  - **Root lock resolution:** `0.9.0` in `Cargo.lock`
- **Dependency:** `zeroize`
  - **Manifest references:** transitive of `rustls-pki-types` (`alloc`)
  - **Root lock resolution:** `1.8.2` in `Cargo.lock`

The four kernel-critical crates `limine`, `spin`, `x86_64`, and `smoltcp` are declared with semver-range requirements ("`0.6`", "`0.9`", "`0.15`", and the caret "`0.13.0`"), not the exact =`X.Y.Z` requirements applied to `capnp` (=`0.25.4`) in `kernel/Cargo.toml` and `sha2` (=`0.10.9` in `capos-lib/Cargo.toml`).

This requirement-level asymmetry is currently unintentional drift in manifest style rather than a deliberate policy: the exact crate version that ships is still pinned by the checked-in `Cargo.lock` checksums above and is review-visible through lockfile diffs, so a range requirement does not widen what actually compiles without a lockfile change. Tightening these four manifest requirements to `=X.Y.Z` to match `capnp`/`sha2` is a separate build-risk change (a manifest edit plus lockfile regeneration and re-verification), tracked as a doc-accuracy gap here rather than changed in this inventory pass.

Standalone lockfile drift observed during this inventory:

The TLS client handshake smoke adds a userspace-runtime no_std dependency in the standalone `demos/` workspace: `embedded-tls = "=0.19.0"` as a path dependency under `vendor/embedded-tls/embedded-tls-0.19.0/`, with `default-features = false` and only the `rustpki` feature enabled. `demos/Cargo.lock` pins the resulting RustCrypto TLS 1.3 closure. The capOS custom target forces the software AES and POLYVAL backends in `.cargo/config.toml` so those crypto dependencies do not select x86 accelerated backend code that is outside the custom-target build contract.

- **Lockfile:** `init/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`
- **Lockfile:** `demos/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`
- **Lockfile:** `demos/wasi-hello-rust/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the `wasm32-wasip1` Rust hello payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-cli-args/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the Phase W.3 argv-grant `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-env/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the WASI environment-grant `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-fs/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the WASI filesystem `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-random/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the Phase W.4 `random_get` `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-preview1-refusals/Cargo.lock`
  - **Notable direct/runtime resolution:** Single-package leaf lockfile for the WASI Preview 1 refusal-coverage `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `demos/wasi-stdio-fd/Cargo.lock`

- **Notable direct/runtime resolution:** Single-package leaf lockfile for the WASI stdio-fd `wasm32-wasip1` Rust payload; no third-party direct dependencies.
- **Lockfile:** `tools/mkmanifest/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `serde_json 1.0.149`
- **Lockfile:** `tools/adventure-content-gen/Cargo.lock`
  - **Notable direct/runtime resolution:** Host generator for adventure content; locked dependencies include `serde_json` and the cue-export-to-JSON pipeline; no `capnp` runtime dependency.
- **Lockfile:** `tools/paperclips-content-gen/Cargo.lock`
  - **Notable direct/runtime resolution:** Host generator for Paperclips content; locked dependencies include `serde_json` and `capnp 0.25.4` for schema-aware JSON-to-binary conversion through `mkmanifest cue-to-capnp`.
- **Lockfile:** `tools/remote-session-client/Cargo.lock`
  - **Notable direct/runtime resolution:** Standalone Linux host-side remote-session client; pins `capnp 0.25.4` and `serde 1.0.228`; no transitive `wasmi`, Argon2, or smoltcp dependency. Covered by `make dependency-policy-check`.
- **Lockfile:** `tools/ringtap-viewer/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`; no Argon2 because it uses baseline `capos-config`
- **Lockfile:** `capos-rt/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`
- **Lockfile:** `capos-service/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6` (the same allocator resolution as `capos-rt`/`demos`/`libcapos`; no cross-workspace drift).
- **Lockfile:** `libcapos/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6` plus the local `capos-rt` path dependency.
- **Lockfile:** `libcapos-posix/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`, plus local `capos-rt` and `libcapos` path dependencies.
- **Lockfile:** `shell/Cargo.lock`
  - **Notable direct/runtime resolution:** `blake2 0.10.6`, `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`; no Argon2 because it uses baseline `capos-config`
- **Lockfile:** `capos-wasm/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`, `wasmi 1.0.9` (vendored static-pinned at `vendor/wasmi-no_std/wasmi-1.0.9/`); no Argon2.
- **Lockfile:** `fuzz/Cargo.lock`

  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `libfuzzer-sys 0.4.12`
- **Lockfile:** `tools/remote-session-client/src-tauri/Cargo.lock` (not yet under policy gates)
  - **Notable direct/runtime resolution:** Tauri scaffold lockfile carrying 435 transitive packages pinned through `tauri = "=2.11.1"`; only reachable through `make remote-session-tauri` policy / check / dev modes. Not covered by `make dependency-policy-check` today; promotion is gated on the Tauri authority decision.
- **Lockfile:** `vendor/wasmi-no_std/wasmi-1.0.9/Cargo.lock` (vendored snapshot lockfile)
  - **Notable direct/runtime resolution:** Upstream wasmi workspace lockfile preserved with the static-pinned snapshot; `capos-wasm` consumes wasmi only through its own `=1.0.9` path dependency, which lands in `capos-wasm/Cargo.lock` and is covered there. The vendored lockfile is not separately gated; see `vendor/wasmi-no_std/VENDORED_FROM.md` for the refresh procedure and policy re-check.

Cargo lockfiles pin exact crate versions and crates.io checksums, so ordinary crate upgrades are review-visible through lockfile diffs. They do not, by themselves, define whether a dependency is acceptable for kernel/no_std use, whether multiple lockfiles must converge, or whether advisories/licenses block the build.

Security Verification Track S.10.3 policy gate:

- `deny.toml` defines the shared license, advisory, ban, and source baseline.
- The allowed license set is intentionally limited to permissive licenses used by current locked dependencies. `BSD-3-Clause` is accepted for the Argon2 credential-validation dependency closure (`subtle` through `password-hash`, `digest`, and `blake2`); it is OSI-approved, FSF-free, and carries only the standard non-endorsement clause beyond the already-allowed `BSD-2-Clause`. `0BSD` is accepted for the smoltcp networking dependency closure (`smoltcp` and `managed`); it is OSI-approved and carries no attribution or non-endorsement condition beyond the existing permissive-license baseline.
- `make dependency-policy-check` runs `cargo deny check` on the root workspace, `init`, `demos`, `tools/mkmanifest`, `tools/ringtap-viewer`, `capos-rt`, `shell`, and `fuzz`.
- The same target runs `cargo audit --deny warnings` on every checked-in lockfile, with one explicit audit ignore: `RUSTSEC-2026-0173` (`proc-macro-error2` unmaintained warning). The ignored path is pulled into lockfiles through `smoltcp`'s optional `defmt` logging feature; capOS does not enable `defmt` for `smoltcp`, but `cargo audit` scans lockfiles rather than the target feature set. Remove the ignore when upstream `smoltcp` / `defmt` no longer resolves that crate.
- The same target copies `package.json` and `package-lock.json` into a private temporary directory and runs a dry-run install there:

  ```
  PUPPETEER_SKIP_DOWNLOAD=1 npm ci --ignore-scripts --dry-run
  ```

  That preserves the `npm ci` package/lock synchronization check without modifying the worktree install. It also runs `npm audit --package-lock-only --audit-level=high`. Lifecycle scripts stay disabled for the docs dependency install path; the browser used by Mermaid PDF rendering is an explicit host executable selected by `MERMAID_BROWSER_BIN`.

- capos-config keeps Argon2 behind the credential-validation feature. Bootstrap/config validation remains available in the baseline feature set, while validators that need to parse PHC credential strings enable the feature. Runtime clients and inspection tools that only need ring/schema/CapSet data use the baseline feature set.
- Local packages are marked publish = false so cargo-deny treats them as private, and local path dependencies include version = "0.1.0" so registry wildcard requirements can remain denied.
- CI installs pinned cargo-deny 0.19.4 and cargo-audit 0.22.1 and runs the target.

Remaining dependency-policy gap: decide whether standalone lockfiles may intentionally drift from the root lockfile, especially for capnp and allocator crates used by userspace.

### Cap'n Proto Compiler, Runtime, and Generated Bindings

The trusted Cap'n Proto inputs are:

- schema/capos.capnp, the source schema.
- Repo-local pinned capnp, invoked through the capnpc Rust build dependency via CAPOS_CAPNP.
- capnp runtime crate with default-features = false and alloc.
- capnpc codegen crate.
- Generated capos_capnp.rs written to Cargo OUT_DIR.
- Local no_std patching applied after generation by tools/capnp-build.

capos-config/build.rs delegates schema generation to tools/capnp-build. That shared helper runs capnpc::CompilerCommand over schema/capos.capnp, reads the generated capos_capnp.rs, asserts that the expected #![allow(unused_variables)] anchor is present, and injects:

```
#![allow(unused_imports)]
use ::alloc::boxed::Box;
use ::alloc::string::ToString;
```

The generated code used by builds is included from OUT_DIR in capos-config/src/lib.rs:10-12. The expected patched output is checked in as tools/generated/capos_capnp.rs, so schema, compiler, capnpc crate, and patch-output changes must update that baseline and become review-visible as a source diff.

Security Verification Track S.10.2 generated-code drift check:

- make generated-code-check first builds the checked-in init ELF required by kernel build-script validation, exports its absolute path as CAPOS_INIT_ELF, and runs tools/check-generated-capnp.sh, tools/check-generated-adventure-content.sh, and tools/check-generated-paperclips-content.sh.
- The script invokes the actual Cargo build-script path for capos-config in an isolated target directory, so it checks the generated artifact that crate would include from OUT_DIR.
- During that build, tools/capnp-build also copies the patched binding to a deterministic package-scoped path under the isolated target directory. The checker consumes those explicit paths rather than searching Cargo's hashed build-script output directories.

- The script verifies that the patched file still contains the capnpc anchor plus the local no_std patch imports, compares the output against `tools/generated/capos_capnp.rs`, and fails if a kernel-generated output path appears in the isolated target directory.
- Any intentional schema/codegen/patch change must update the checked-in baseline in the same review, making generated output drift review-visible.
- `make check` runs `fmt-check` plus `generated-code-check` for a single local or CI entry point.
- Current pinned compiler source is `capnproto-c++-1.2.0.tar.gz` from `https://capnproto.org/` with SHA-256 `ed00e44ecbbda5186bc78a41ba64a8dc4a861b5f8d4e822959b0144ae6fd42ef`. The checked-in `tools/generated/capos_capnp.rs` baseline must be regenerated with that compiler when schema or codegen behavior intentionally changes. The current pinned baseline SHA-256 is `5ab84731324fe9cc984d7aba7dd97963a773800cc52c4c1693fcb6bb448329a6`.

Adventure content generation uses:

- `demos/adventure-content/content/prototype.cue` as the checked-in source.
- `tools/adventure-content-gen`, a standalone Cargo host tool with `tools/adventure-content-gen/Cargo.lock`.
- `demos/adventure-content/src/generated.rs` as the checked-in generated no_std Rust baseline consumed by `demos/adventure-content/src/lib.rs`.
- `tools/check-generated-adventure-content.sh`, which derives the same `$(CAPOS_TOOLS_ROOT)/cue/0.16.0/bin/cue` path as the Makefile, rejects a mismatched `CAPOS_CUE`, checks `cue version v0.16.0`, exports explicit JSON, runs the generator with `cargo run --locked`, formats the output with `rustfmt --edition 2024`, and fails if the result differs from `demos/adventure-content/src/generated.rs`.

Any intentional content-source or generator change must update the checked-in generated Rust baseline in the same review. The generator manifest and lockfile are included in `make dependency-policy-check`.

The no_std patch source is single-owned by `tools/capnp-build`; `capos-config/build.rs` emits its crate-specific rerun directives and calls the helper.

### Alloy Analyzer (DMA Assurance Model)

The DMA assurance Alloy model (`models/dma/dma_authority.als`) is checked by a pinned Alloy Analyzer, the same trust level as the Limine, capnp, CUE, Typst, and uv pins.

- **Pinned artifact:** `alloy-6.2.0-linux-amd64.tar.gz` from the official `AlloyTools/org.alloytools.alloy` GitHub release `v6.2.0` (`https://github.com/AlloyTools/org.alloytools.alloy/releases/download/v6.2.0/alloy-6.2.0-linux-amd64.tar.gz`), SHA-256 `5a5494a4bac6e243e471590bb44a91e25a35794a5af1ae1f332be30b9c54a9e7`. This is the self-contained linux/amd64 app image: it bundles a Temurin JRE under `lib/runtime/` and the native SAT solver libraries, so the gate needs no host JVM and pins the analyzer *and* its runtime by one hash. The `org.alloytools.alloy.dist.jar` (bare jar, host-JVM dependent) is deliberately not used.

- **Where the pin lives:** `Makefile` `ALLOY_VERSION` / `ALLOY_PLATFORM` / `ALLOY_TARBALL_URL` / `ALLOY_TARBALL_SHA256`. `make alloy-ensure` downloads the tarball (curl with retry), verifies the SHA-256, extracts the app image into `$(CAPOS_TOOLS_ROOT)/alloy/6.2.0/` (shared per-user cache, default `$HOME/.capos-tools`), and confirms the launcher reports version `6.2.0`. The jar is not vendored into the repository.
- **Drift review:** `make model-dma-alloy` re-verifies the tarball SHA-256 and the reported launcher version on every run before invoking the model. A bump is a `Makefile` `ALLOY_VERSION` + `ALLOY_TARBALL_SHA256` diff plus a refreshed checked-result record in `models/dma/README.md`.
- **Why output is parsed, not exit-code-gated:** the Alloy CLI exec subcommand always exits `0`; a check that finds a counterexample, a `run` that finds no instance, and a syntax/resolution error all return success with the failure visible only in the printed verdict table. `tools/run-dma-alloy-model.sh` parses that table and fails closed on any check that is not `UNSAT`, any `run` that is not `SAT`, or any analyzer error marker.
- **Platform/CI scope:** the pinned app image is linux/amd64 (the dev/CI host architecture). GitHub CI runs `make model-dma-alloy` in the `dma-assurance-models` job on `ubuntu-24.04`. Other architectures would need the matching Alloy app image (or the bare jar plus a host JVM). Ownership of the Alloy pin is shared with the scheduler lease model track (`scheduler-cpu-isolation-lease-authority-model`).

### TLC Model Checker (DMA Assurance Lifecycle Model)

The DMA assurance TLA+ lifecycle model (`models/dma/dma_authority.tla`) is checked by a pinned TLC, the same trust level as the Limine, capnp, CUE, Typst, uv, and Alloy pins. Unlike the self-contained Alloy app image, `tla2tools.jar` is a bare Java jar, so a JVM is pinned alongside it.

- **Pinned artifacts:** `tla2tools.jar` from the official `tlaplus/tlaplus` GitHub release `v1.7.4` (TLC 2.19), `https://github.com/tlaplus/tlaplus/releases/download/v1.7.4/tla2tools.jar`, SHA-256 `936a262061c914694dfd669a543be24573c45d5aa0ff20a8b96b23d01e050e88`; and a Temurin JRE `17.0.19+10` linux/x64 tarball (`OpenJDK17U-jre_x64_linux_hotspot_17.0.19_10.tar.gz` from the `adoptium/temurin17-binaries` release), SHA-256 `adb5a2364baa51de1ef91bb9911f5a61d24b045fe1d6647cb8050272a3a8ee75`. Pinning the JRE as well as the jar fixes both the checker *and* its runtime by hash.
- **Where the pin lives:** `Makefile` `TLA_TOOLS_VERSION` / `TLA_TOOLS_JAR_URL` / `TLA_TOOLS_JAR_SHA256` / `TLA_JRE_URL` / `TLA_JRE_SHA256`. `make tla-ensure` downloads both (curl with retry), verifies their SHA-256, extracts the JRE into `$(CAPOS_TOOLS_ROOT)/tla/jre/` and places the jar at `$(CAPOS_TOOLS_ROOT)/tla/1.7.4/tla2tools.jar` (shared per-user cache, default `$HOME/.capos-tools`), and confirms the launcher reports `17.0.19`. Neither is vendored into the repository.
- **Drift review:** `make model-dma-tla` re-verifies the jar SHA-256 and the JRE launcher version on every run before invoking the model. A bump is a `Makefile` pin diff plus a refreshed checked-result record in `models/dma/README.md`.
- **Why output is parsed *and* exit-code-gated:** TLC returns a non-zero exit code (12) on an invariant violation, a deadlock, or a parse/semantic error, but `tools/run-dma-tla-model.sh` additionally asserts the `Model checking completed. No error has been found.` marker and rejects any violation/error marker, so a future TLC behaviour change cannot turn a violation

into a green gate. The model is checked with deadlock detection enabled; the spec provides an explicit terminating self-loop for the all-pages-parked state, so any other stuck state is a genuine modelling gap.

- **Platform/CI scope:** the pinned JRE tarball is linux/x64 (the dev/CI host architecture). GitHub CI runs `make model-dma-tla` in the `dma-assurance-models` job on `ubuntu-24.04`; other architectures would need the matching Temurin JRE. Ownership of the TLC pin is shared by the scheduler/IRQ TLA+ model tracks (`scheduler-nohz-activation-model`, `irq-msix-waiter-determinism-model`).

### Cargo Build Scripts

Build scripts currently do these trusted operations:

- **Script:** `kernel/build.rs`
  - **Behavior:** Watches `kernel/linker-x86_64.ld` and itself.
- **Script:** `capos-config/build.rs`
  - **Behavior:** Calls `tools/capnp-build` to watch `schema/capos.capnp`, generate bindings, and apply the shared no_std patch. Checked by `make generated-code-check`.
- **Script:** `tools/capnp-build/src/lib.rs`
  - **Behavior:** Host build-support helper for pinned capnp path validation, schema generation, and no_std generated-binding patching. Unit tests cover patch injection and missing-anchor rejection.
- **Script:** `tools/adventure-content-gen/src/main.rs`
  - **Behavior:** Host generator for the prototype adventure CUE source. Checked by `make generated-code-check` through `tools/check-generated-adventure-content.sh`, which uses pinned CUE and locked Cargo dependencies.
- **Script:** `init/build.rs`
  - **Behavior:** Emits a linker script argument for `init/linker.ld`.
- **Script:** `demos/*/build.rs`
  - **Behavior:** Emits a linker script argument for `demos/linker.ld`.
- **Script:** `capos-rt/build.rs`
  - **Behavior:** Emits a linker script argument for `capos-rt/linker.ld` when building current `target_os = "none"` userspace or custom-target `target_os = "capos"` probes.
- **Script:** `capos-wasm/build.rs`
  - **Behavior:** Emits a linker script argument for `capos-wasm/linker.ld` (Phase W.2 onward; uses `cargo:rustc-link-arg-bins` so the script applies only to the wasm-host bin and not the lib).

The linker build scripts derive `CARGO_MANIFEST_DIR` from Cargo and only emit link arguments plus rerun directives. The capnp build scripts read and rewrite generated code under `OUT_DIR`. None of these scripts fetch network resources.

Security Verification Track S.10.2 coverage: `make generated-code-check` exercises the canonical `capos-config` capnp build script through Cargo, validates the patched generated file, fails if

kernel-generated output reappears, and fails if the canonical output no longer matches the checked-in generated baseline.

### Manifest, Embedded Binaries, and Downloaded Artifacts

`system.cue` declares named binaries and services. `Makefile` builds `manifest.bin` by running `tools/mkmanifest` on the host. `mkmanifest` runs:

1. Resolve the pinned CUE compiler from `$(CAPOS_TOOLS_ROOT)`, reject missing or mismatched `CAPOS_CUE`, check `cue version v0.16.0`, then run `cue export system.cue --out json` or package-mode equivalent.
2. JSON-to-`CueValue` conversion and manifest validation (`tools/mkmanifest/src/lib.rs`).
3. Binary embedding from relative paths (`tools/mkmanifest/src/lib.rs`).
4. Binary-reference validation and Cap'n Proto serialization (`tools/mkmanifest/src/main.rs`).

The adjacent `mkmanifest cue-to-capnp` subcommand uses the same pinned CUE export path but does not parse the result as `SystemManifest`. Instead, it resolves and validates `CAPOS_CAPNP`, checks `Cap'n Proto version 1.2.0`, and passes the exported JSON to Cap'n Proto:

```
capnp convert json:binary <schema.capnp> <RootType>
```

It is the supported schema-aware path for CUE-authored data messages rooted at arbitrary specified Cap'n Proto structs; live capabilities and interface objects are outside that data-file contract.

Path handling rejects absolute paths, parent traversal, non-normal components, and canonicalized paths that escape the manifest directory (`tools/mkmanifest/src/lib.rs`). The generated `manifest.bin` is copied into the ISO as `/boot/manifest.bin` and loaded by Limine via `limine.conf:5`.

Downloaded or generated artifacts in the current build:

- **Artifact:** `$(LIMINE_DIR)` checkout (`$(CAPOS_TOOLS_ROOT)/limine/<commit>`)
  - **Producer:** `git clone`/`git fetch` in the `limine-ensure` recipe
  - **Pinning/drift status:** Commit-pinned and artifact-verified.
- **Artifact:** Cargo registry crates
  - **Producer:** `cargo build`, `cargo run`, tests, fuzz
  - **Pinning/drift status:** Lockfile-pinned checksums plus CI-enforced deny/audit checks through `make dependency-policy-check`.
- **Artifact:** Node registry packages
  - **Producer:** `npm ci --ignore-scripts` for docs Mermaid rendering
  - **Pinning/drift status:** `package-lock.json` pins package tarball integrity. Lifecycle scripts are disabled, Puppeteer's browser download path is skipped, and `make dependency-policy-check` enforces the `npm ci` package/lock synchronization invariant plus high-severity npm audit state.
- **Artifact:** Chromium/Chrome for Mermaid PDF rendering

  - **Producer:** Host executable selected by `MERMAID_BROWSER_BIN` or auto-detected from `chromium-browser`, `chromium`, `google-chrome-stable`, or `google-chrome`
  - **Pinning/drift status:** Host-provided browser, not repo-pinned. The docs PDF target fails closed if no executable is available and passes the selected path to Puppeteer as `PUPPETEER_EXECUTABLE_PATH`, rather than allowing Puppeteer's npm install script to download an implicit browser artifact.
- **Artifact:** Rust toolchain, targets, and `rust-src`
  - **Producer:** `rustup` from `rust-toolchain.toml` when absent
  - **Pinning/drift status:** Date-pinned `nightly-2026-04-20` channel; `rust-src` is declared for custom-target `-Zbuild-std` userspace builds. The advance procedure for bumping the pin lives in the Rust Toolchain section above.
- **Artifact:** `target/` kernel and host artifacts
  - **Producer:** Cargo
  - **Pinning/drift status:** Generated, not checked in.
- **Artifact:** `init/target/`, `demos/target/`, `capos-rt/target/`, `capos-wasm/target/` ELFs
  - **Producer:** Cargo standalone builds
  - **Pinning/drift status:** Generated, embedded into `manifest.bin` where referenced; `make build-provenance` records hashes for embedded file-backed and inline payloads.
- **Artifact:** `target/x86_64-unknown-capos/`, `init/target/x86_64-unknown-capos/`, `demos/target/x86_64-unknown-capos/`, `shell/target/x86_64-unknown-capos/`, `capos-rt/target/x86_64-unknown-capos/`, `libcapos/target/x86_64-unknown-capos/`, `libcapos-posix/target/x86_64-unknown-capos/`, and `capos-wasm/target/x86_64-unknown-capos/` userspace artifacts
  - **Producer:** Cargo aliases using `targets/x86_64-unknown-capos.json`
  - **Pinning/drift status:** Generated artifacts for booted userspace manifests, the `capos-rt` smoke binary, the `wasm-host` Phase W.2 binary, and the libcapos / libcapos-posix C-substrate staticlibs.
- **Artifact:** `manifest.bin`
  - **Producer:** `tools/mkmanifest`
  - **Pinning/drift status:** Generated from `system.cue` plus ELF payloads; not checked in. Hash is recorded by `make build-provenance`.
- **Artifact:** `iso_root/` and `capos.iso`
  - **Producer:** Makefile, xorriso, Limine installer
  - **Pinning/drift status:** Generated and gitignored; Limine inputs verified. Final ISO hash is recorded by `make build-provenance`.
- **Artifact:** `target/build-provenance.txt`
  - **Producer:** `tools/build-provenance.sh` via `make build-provenance`
  - **Pinning/drift status:** Generated and gitignored; records runner OS/kernel/architecture identity, GitHub Actions image identity when present, Rust toolchain details, selected executable paths, package identities when discoverable, OVMF selected path/package/absence state, tool versions, git commit, manifest/ISO/kernel/OVMF hashes, and embedded

payload origin plus hashes. CI publishes the artifact as `build-provenance-<sha>` on every `qemu-smoke` run (30-day retention). On `pull_request` events the `qemu-smoke` job locates the most recent successful main-branch `build-provenance-<sha>` artifact, downloads it via `actions/download-artifact`, and runs `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment` against the candidate record as a blocking PR gate.

Remaining gaps for Security Verification Track S.10.2/S.10.3:

- CI now publishes `target/build-provenance.txt` as a named artifact on every `qemu-smoke` run (30-day retention) and, on `pull_request` events, downloads the most recent successful main-branch `build-provenance-<sha>` artifact and runs `make build-provenance-compare` against the candidate record with `BUILD_PROVENANCE_COMPARE_POLICY=ci-environment`. The compare step is PR-blocking for runner/tool/Rust/OVMF environment drift and fails when the base artifact cannot be found.
- `qemu-smoke` apt-pins `qemu-system-x86`, `xorriso`, `make`, `git`, and `ovmf`, and `make build-provenance` records normalized package identity where the runner exposes it. Repo-pinned digests (download-and-verify rather than apt-installed packages) for `qemu-system-x86`, `xorriso`, `make`, and `git`, or an immutable runner image digest containing that package set, remain future production-reproducibility hardening tracked in `docs/design-risks-register.md` (R13).
- Decide whether CI should record the pinned `cue export` JSON or final `manifest.bin` bytes if manifest reproducibility becomes release-critical.

### Vendored Upstream Snapshots

The repository carries static, pinned snapshots of selected upstream sources under `vendor/`. Each snapshot has its own `VENDORED_FROM.md` recording the upstream URL, tag/version, commit SHA, commit date, vendoring date, license, vendoring posture, and refresh procedure. Snapshots are kept byte-identical to their pinned upstream artifact (git commit, or the crates.io published crate as noted below); the only non-upstream changes permitted without a `patches/` unified diff are the documented integration-only empty-`[workspace]` marker and build-inert files restored from the same upstream commit when the publish `include` omitted them (for `rustls-webpki`, `src/test_utils.rs` and `rustfmt.toml`, both recorded in its `VENDORED_FROM.md`). Any future functional patch must be recorded as a unified diff under the snapshot's `patches/` directory plus a `Patches` entry per the procedure in the snapshot's `VENDORED_FROM.md`.

- **Snapshot:** `vendor/wasmi-no_std/wasmi-1.0.9/`
  - **Upstream:** https://github.com/wasmi-labs/wasmi
  - **Tag/Version:** `v1.0.9`
  - **Commit SHA:** `61ba65e6563d8b2f5b699b018349d3330b28b9f3`
  - **License:** Apache-2.0 OR MIT (dual)
  - **Consumer:** `capos-wasm/` (WASI host adapter wasm-host bin and Preview 1 import surface)
- **Snapshot:** `vendor/dns-c-wahern/src/`
  - **Upstream:** https://github.com/wahern/dns
  - **Tag/Version:** `rel-20160808`

  - **Commit SHA:** 4ec718a77633c5a02fb77883387d1e7604750251
  - **License:** MIT
  - **Consumer:** POSIX adapter Phase P1.2 Phase B DNS smoke; not yet on the v0 build path (the smoke compiles only demos/posix-dns-resolver/main.c with a commented-out dns.h include)
- **Snapshot:** vendor/rustls-webpki/rustls-webpki-0.103.13/
  - **Upstream:** https://github.com/rustls/webpki
  - **Tag/Version:** 0.103.13 (crates.io crate)
  - **Commit SHA:** 2879b2ce7a476181ac3050f73fe0835f04728e86
  - **License:** ISC
  - **Consumer:** capos-tls/ Phase-1 verifier (WebPKI X.509 path building + signature verification, no_std + alloc, no crypto provider in the default build)
- **Snapshot:** vendor/webpki-roots/webpki-roots-1.0.7/
  - **Upstream:** https://github.com/rustls/webpki-roots
  - **Tag/Version:** 1.0.7 (crates.io crate)
  - **Commit SHA:** be948464fd5907af6227213a066743a161221688
  - **License:** CDLA-Permissive-2.0
  - **Consumer:** capos-tls/ Phase-1 trust-anchor bootstrap (compiled-in Mozilla NSS root bundle, no_std)
- **Snapshot:** vendor/embedded-tls/embedded-tls-0.19.0/
  - **Upstream:** https://github.com/drogue-iot/embedded-tls
  - **Tag/Version:** 0.19.0 (crates.io crate)
  - **Commit SHA:** 865e1fd983c583228e3bbeb9f4996f1abc454ca3
  - **License:** Apache-2.0
  - **Consumer:** demos/cloud-tls-client-handshake-smoke/ TLS 1.3 client state machine (no_std + alloc, default std/tokio features disabled, rustpki enabled)

The rustls-webpki and webpki-roots snapshots use a published-crate posture distinct from the git-clone snapshots above: each is the crates.io published .crate artifact, SHA-256-verified against the crates.io index (61c429a8…f756e and 52f5ee44…2eb9d respectively), with the upstream commit recorded from the artifact's embedded .cargo_vcs_info.json. For rustls-webpki, two build-inert files the publish include omitted (src/test_utils.rs, a #[cfg(test)] module, and rustfmt.toml) are restored from the same upstream commit so cargo fmt resolves the module tree and formats the snapshot under upstream's own style config; see its VENDORED_FROM.md. capos-tls/ (a root-workspace member and the Phase-1 host verifier crate) depends on both as exact-pinned path dependencies (version = "=0.103.13" / version = "=1.0.7") and forces them to link for x86_64-unknown-none under cargo build and cargo build --features qemu. rustls-webpki is selected with default-features = false, features = ["alloc"], so neither std nor a ring / aws-lc-rs crypto provider is compiled; the active compiled closure is rustls-pki-types (alloc, pulling zeroize) and untrusted (ISC), all pinned in the root Cargo.lock and covered by make dependency-policy-check. The ring optional dependency appears in Cargo.lock as an unselected optional entry; aws-lc-rs is a feature-gated

optional / dev-only dependency and does not resolve into the root lockfile at all. Neither is ever feature-activated, so no crypto provider is compiled and `cargo deny` does not evaluate `ring` (`cargo tree -p capos-tls -e features` activates only `rustls-pki-types`, `zeroize`, `untrusted`, `webpki-roots`, and `rustls-webpki[alloc]`). The `webpki-ring` feature is host-test-only and supplies the signature algorithms for `cargo test-tls`; the default bare-metal build remains provider-free.

The `embedded-tls` snapshot uses the same published-crate posture: the vendored tree is the crates.io `0.19.0` package with `.cargo_vcs_info.json` recording upstream commit `865e1fd983c583228e3bbeb9f4996f1abc454ca3`. The local handshake smoke depends on it with an exact path pin, disables default `std` and `tokio`, and enables only `rustpki` so the TLS 1.3 client path can run under `target_os = "capos"` over a `TcpSocket` cap. The empty `[workspace]` marker is the only local integration change.

`capos-wasm/Cargo.toml` pins the wasmi path dependency to `version = "=1.0.9"` so cargo-deny's wildcards gate continues to pass; the snapshot is exercised by `make capos-wasm-build`, every `make run-wasi-*` smoke, and `make dependency-policy-check` (cargo-deny + cargo-audit on `capos-wasm/Cargo.lock`). Refreshing wasmi to a newer tag requires the rsync pattern, manifest pin bump, lockfile regeneration, and policy re-check recorded in `vendor/wasmi-no_std/VENDORED_FROM.md`.

The dns.c snapshot is intentionally a strict subset (only `src/dns.c`, `src/dns.h`, `LICENSE`, and `README.md`); ancillary upstream files (`cache`, `mem`, `spf`, `zone`, `regress`) are excluded because the v0 build path does not need them. Future POSIX-adapter phases that widen `libcapos-posix` enough to compile `dns.c` whole will start consuming the snapshot in the build instead of carrying it as a documentation-only reference.

`vendor/dash/` is not present at this revision. If a future POSIX-adapter phase imports dash, add a new row above plus a `vendor/dash/VENDORED_FROM.md` recording the same provenance fields.

### Host Tools

Current local host versions observed during this inventory:

- **Tool:** `capnp`
  - **Observed version:** `1.2.0`
  - **Build role:** Repo-selected schema compiler built by `make capnp-ensure` from a SHA-256-pinned official source tarball into `$(CAPOS_TOOLS_ROOT)`.
- **Tool:** `cue`
  - **Observed version:** `v0.16.0`
  - **Build role:** Repo-selected manifest compiler installed by `make cue-ensure` into `$(CAPOS_TOOLS_ROOT)` from the SHA-256-verified official release binary.
- **Tool:** `qemu-system-x86_64`
  - **Observed version:** `10.2.2`
  - **Build role:** Boot verification via `make run` and `make run-uefi`.
- **Tool:** `xorriso`
  - **Observed version:** `1.5.8`

- **Build role:** ISO generation.
- **Tool:** `make`
  - **Observed version:** `4.4.1`
  - **Build role:** Build orchestration.
- **Tool:** `git`
  - **Observed version:** `2.53.0`
  - **Build role:** Limine checkout/fetch and review workflow.

These are local environment observations, not repository pins. On the `qemu-smoke` CI runner, `qemu-system-x86`, `xorriso`, `make`, and `git` are apt-pinned to `qemu-system-x86=1:8.2.2+ds-0ubuntu1.16` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04), `xorriso=1:1.5.6-1.1ubuntu3` (amd64, `noble`/main, Ubuntu 24.04), `make=4.3-4.1build2` (amd64, `noble`/main, Ubuntu 24.04), and `git=1:2.43.0-1ubuntu7.3` (amd64, `noble-updates`/main or `noble-security`/main, Ubuntu 24.04) by the "Install boot smoke dependencies" step; the per-run identity for each is captured via `dpkg-query` and normalized apt source pockets by `tools/build-provenance.sh`. The bump procedure mirrors OVMF: run `apt-cache madison <tool>` on a current Ubuntu 24.04 host, pick the highest stable version from `noble-updates/main` (or `noble/main` when no `noble-updates` entry exists, as is the case for `xorriso` and `make` today), and update the pinned version string in `.github/workflows/ci.yml` plus this row. `make run-uefi` selects an OVMF firmware blob from `OVMF_CODE_CANDIDATES` in `Makefile:96-97`; the Makefile pins the expected blob via `OVMF_CODE_SHA256` and the `ovmf-verify` target enforces the match before ISO and cloud-disk construction. On the `qemu-smoke` CI runner the `ovmf=2024.02-2ubuntu0.8` apt-install resolves the first candidate (`/usr/share/ovmf/OVMF.fd`), `ovmf-verify` succeeds against the pinned digest, and `make build-provenance` records the resulting firmware-blob SHA-256 per run. Build hosts without any OVMF candidate installed see an `ovmf-verify` NOTICE skip rather than a failure, so research workflows that never invoke `make run-uefi` continue to build the ISO unchanged.

Remaining gap for Security Verification Track S.10.3: decide whether full production reproducibility uses an immutable runner image digest, repo-managed download-and-verify tool digests for the apt-pinned build/boot tools, or both. `build-essential`, `curl`, `sha256sum`, the shell, and the checkout-time git used by `actions/checkout` remain runner-provided; the PR-blocking provenance gate records and compares the post-checkout build environment, but it does not turn the mutable `ubuntu-24.04` runner label into an immutable production image.

### GitHub Actions Runner and Workflow Pinning

The CI harness in `.github/workflows/ci.yml` is itself a supply-chain input: its identities determine which third-party code runs against every push and pull request, and the chosen runner image determines the host package set underneath every host-baseline, Kani, and optional QEMU job. Mutable `@v<N>` or `@master` references on third-party Actions would allow upstream owners to swap out the executed code at any time without a repository diff, and `ubuntu-latest` would silently roll the runner OS when GitHub re-points it.

The current policy is to pin every third-party Action to a 40-character commit SHA and to pin the runner OS to a specific release rather than the floating label. Each pinned `uses:` line carries a

trailing `# v<X.Y.Z>` comment so reviewers and bump PRs can read the intended release without following the SHA through the GitHub UI.

- **Identity:** `runs-on:` runner image
  - **Pinned reference:** `ubuntu-24.04`
  - **Notes:** Replaces `ubuntu-latest`; applied to `host-baseline`, `kani-proofs`, `dma-assurance-models`, and `qemu-smoke`. GitHub-hosted `ImageOS` and `ImageVersion` are recorded in `target/build-provenance.txt` when present and are compared by the PR-blocking CI environment policy. Bump only when the next LTS is needed and the full `make check` plus QEMU smokes are reverified against the new image.
- **Identity:** `actions/checkout`
  - **Pinned reference:** `34e114876b0b11c390a56381ad16ebd13914f8d5  # v4.3.1`
  - **Notes:** Resolved from the `actions/checkout` v4 major-version tag.
- **Identity:** `swatinem/rust-cache`
  - **Pinned reference:** `c19371144df3bb44fab255c43d04cbc2ab54d1c4  # v2.9.1`
  - **Notes:** Canonical `Swatinem/rust-cache v2.9.1` release commit. The v2 major-tracking tag carries the same `2.9.1` message but points at a distinct republication commit; always dereference the exact release tag rather than the major tag.
- **Identity:** `dtolnay/rust-toolchain`
  - **Pinned reference:** `3c5f7ea28cd621ae0bf5283f0e981fb97b8a7af9  # master @ 2026-03-27`
  - **Notes:** The upstream action does not publish numbered releases; its documented usage is `@master`. The pin is a snapshot of `master` at the dated commit.
- **Identity:** `actions/upload-artifact`
  - **Pinned reference:** `ea165f8d65b6e75b540449e92b4886f43607fa02  # v4.6.2`
  - **Notes:** Resolved from the `actions/upload-artifact` `v4.6.2` lightweight tag (same SHA as the moving v4 major tag at resolution time). Used in `qemu-smoke` to publish `target/build-provenance.txt`.
- **Identity:** `actions/download-artifact`
  - **Pinned reference:** `d3f86a106a0bac45b974a628896c90dbdf5c8093  # v4.3.0`
  - **Notes:** Resolved from the `actions/download-artifact` `v4.3.0` release tag. Paired with `actions/upload-artifact@v4.6.2` (both v4 series) and used in `qemu-smoke` on `pull_request` events to fetch the most recent successful main-branch `build-provenance-<sha>` artifact for the blocking `make build-provenance-compare BUILD_PROVENANCE_COMPARE_POLICY=ci-environment` step.

Bump procedure for any of the entries above:

1. Resolve the candidate release to its commit SHA via the upstream release tag, e.g. `gh api repos/<owner>/<repo>/git/ref/tags/v<X.Y.Z>` (dereference any annotated tag through `gh api repos/<owner>/<repo>/git/tags/<sha>`), or `gh api repos/<owner>/<repo>/commits/<branch>` for branch-tracked actions like `dtolnay/rust-toolchain`. Always dereference the exact release tag (`vX.Y.Z`) rather than the moving major-version tag (`vX`): major-version tags can be re-cut at a republication commit whose tag message still names the same release (as

observed for `swatinem/rust-cache@v2`), so following `vX` can pin a different commit than the canonical `vX.Y.Z` release.

2. Update both the SHA and the trailing # v<X.Y.Z> comment in `.github/workflows/ci.yml` so the reviewer sees the intended release.
3. Run `make fmt-check` and `make workflow-check` locally for the bump branch. Workflow hygiene plus YAML well-formedness must pass before review. The acceptance gate for the bump itself is a green CI run on the bump branch – `make check` plus the existing QEMU smokes – which exercises the new Action versions end-to-end.
4. Treat any `master`-branch SHA pin (currently `dtolnay/rust-toolchain`) as a manual-bump dependency: the upstream action does not publish release tags, so bumping its SHA is the only way to absorb upstream fixes. Schedule those bumps explicitly rather than relying on a floating reference.

This pinning closes the mutable-tag supply-chain gap for the CI harness itself. It does not by itself satisfy the "pinned runner image digest" line of the Build Provenance Retention And Comparison Policy: `ubuntu-24.04` is still a label managed by GitHub Actions, not an immutable image digest. The current documented equivalent for PR gating is to retain the GitHub-hosted `ImageOS`/`ImageVersion` fields in `target/build-provenance.txt` and compare them against the latest successful main-branch record. A future production-hardening slice may move to a self-built runner image referenced by digest, mirror the build-tool packages, or both.

### Inventory Method

This inventory is based on source inspection, Cargo metadata, lockfile checks, and local host-tool version queries. Local host-tool versions are observations, not repository pins; the tables above distinguish enforced pins from observed environment state.

Useful commands for refreshing the inventory:

- `git status --short --branch`
- `rg -n "S\\.10|trusted|supply|Limine|limine|capnp|capnpc|QEMU|qemu|download|curl|git clone|wget|build\\.rs|rust-toolchain|Cargo\\.lock" ...`
- `rg --files`
- `cargo metadata --locked --format-version 1 --no-deps`
- `rg -n '^name = |^version = |^checksum = ' Cargo.lock init/Cargo.lock demos/Cargo.lock tools/mkmanifest/Cargo.lock tools/ringtap-viewer/Cargo.lock capos-rt/Cargo.lock shell/Cargo.lock libcapos/Cargo.lock libcapos-posix/Cargo.lock capos-wasm/Cargo.lock fuzz/Cargo.lock`
- `command -v rustc cargo capnp cue qemu-system-x86_64 xorriso sha256sum git make`
- `rustc -Vv`, `cargo -V`, `capnp --version`, `cue version`, `qemu-system-x86_64 --version`, `xorriso -version`, `make --version`, `git --version`

## Panic-Surface Inventory

Scope: `panic!`, `assert!`, `debug_assert!`, `.unwrap()`, `.expect()`, `todo!`, and `unreachable!` surfaces relevant to boot manifest loading, ELF loading, SQE handling, params/result buffers, IPC, and future spawn inputs.

Classification terms:

- `trusted-internal`: depends on kernel/shared-code invariants, static ABI layout, or host build/test code; not directly controlled by a service.
- `boot-fatal`: reached during boot/package setup before mutually untrusted services run. Bad platform/package state can halt the system.
- `untrusted-input reachable`: reachable from userspace-controlled SQEs, Cap'n Proto params/result buffers, IPC state, manifest/package data, or future spawn-controlled service/binary data.

### Summary

No current `panic!/assert!/unwrap()/expect()` site found in the kernel ring dispatch path directly consumes raw SQE fields or user params/result-buffer pointers. Those paths mostly return CQE errors through `kernel/src/cap/ring.rs`.

The remaining relevant surfaces are boot-fatal setup assumptions, scheduler internal invariants that would become more exposed once untrusted spawn/lifecycle inputs can create or destroy processes dynamically, and IPC rollback queue capacity assumptions.

Locations use `path::function` anchors rather than line numbers; line numbers drift on every refactor. Grep the path plus the quoted surface text to re-locate a site.

### Manifest And Future Spawn Inputs

- **Location:** `kernel/src/main.rs` `run_init`
  - **Surface:** `MODULES.response().expect("no modules from bootloader")`
  - **Reachability:** Boot package/module table
  - **Classification:** boot-fatal
  - **Notes:** Missing Limine modules abort before manifest validation.
- **Location:** `kernel/src/main.rs` `run_init`
  - **Surface:** `elf_cache.get(service.binary.as_str()).ok_or_else(...)`
  - **Reachability:** Manifest service binary reference
  - **Classification:** untrusted-input reachable, controlled error
  - **Notes:** Not a panic surface. Included because it is the future spawn shape to preserve: unknown or unparsed binaries return an error.
- **Location:** `kernel/src/spawn.rs` `spawn_service`
  - **Surface:** `Process::new(...).map_err(...)`
  - **Reachability:** Manifest-spawned process creation
  - **Classification:** untrusted-input reachable, controlled error

- **Notes:** Current boot path converts allocation/mapping failures into boot errors. Future `ProcessSpawner` should keep this shape instead of adding unwraps.

**ELF Inputs**

- **Location:** `kernel/src/spawn.rs` `load_elf`
  - **Surface:** `debug_assert!(stack_top % 16 == 0, ...)`
  - **Reachability:** ELF load path
  - **Classification:** trusted-internal
  - **Notes:** Constant stack layout invariant, not ELF-controlled.
- **Location:** `kernel/src/spawn.rs` `align_up`
  - **Surface:** `debug_assert!(align.is_power_of_two())`
  - **Reachability:** TLS mapping from parsed ELF
  - **Classification:** trusted-internal
  - **Notes:** `elf::parse` rejects non-power-of-two TLS alignment; `load_tls` also caps the size before calling `align_up`.
- **Location:** `capos-lib/src/elf.rs` parser
  - **Surface:** no runtime panic surfaces outside tests/Kani
  - **Reachability:** Boot manifest ELF bytes; future spawn ELF bytes
  - **Classification:** untrusted-input reachable, controlled error
  - **Notes:** Parser uses checked offsets/ranges and returns `Err(&'static str)`. Test-only assertions/unwraps are excluded from runtime classification.
- **Location:** `kernel/src/spawn.rs` `load_elf`
  - **Surface:** slice `init_data[src_offset..]`
  - **Reachability:** Parsed ELF PT_LOAD file range
  - **Classification:** untrusted-input reachable, guarded
  - **Notes:** Not matched by the panic-token grep, but it is an index panic candidate if parser invariants are bypassed. `elf::parse` checks segment file ranges before `load_elf`.
- **Location:** `kernel/src/spawn.rs` `load_tls`
  - **Surface:** slice `&init_data[init_start..init_end]`
  - **Reachability:** Parsed ELF TLS file range
  - **Classification:** untrusted-input reachable, guarded
  - **Notes:** Not matched by the panic-token grep, but it is an index panic candidate if parser invariants are bypassed. `elf::parse` checks TLS file bounds before `load_tls`.

**SQE And Params/Result Buffers**

- **Location:** `kernel/src/cap/ring.rs` `process_ring` / `dispatch_call` / `dispatch_recv` / `dispatch_return`
  - **Surface:** no matched panic-like surfaces
  - **Reachability:** Userspace SQEs, params, result buffers
  - **Classification:** untrusted-input reachable, controlled error

  - **Notes:** SQ corruption, unsupported fields/opcodes, oversized buffers, invalid user buffers, and CQ pressure return transport errors or defer consumption.
- **Location:** `capos-config/src/ring.rs` `const _: () = assert!(...)` ABI size checks
  - **Surface:** const `assert!` layout checks
  - **Reachability:** Shared ring ABI
  - **Classification:** trusted-internal
  - **Notes:** Compile-time ABI guard; not runtime input reachable.
- **Location:** `capos-config/src/capset.rs` `const _: () = assert!(...)` ABI size checks
  - **Surface:** const `assert!` layout checks
  - **Reachability:** Shared CapSet ABI
  - **Classification:** trusted-internal
  - **Notes:** Compile-time ABI/page-fit guard; not runtime input reachable.
- **Location:** `capos-lib/src/frame_bitmap.rs` (`alloc_frame` and `alloc_contiguous`)
  - **Surface:** `.try_into().unwrap()` on 8-byte bitmap windows
  - **Reachability:** Frame allocation, including work triggered by manifest/process creation and capability methods
  - **Classification:** trusted-internal
  - **Notes:** Guarded by `frame + 64 <= total` or `i + 64 <= to`, assuming the caller-provided bitmap covers `total_frames`. Kernel constructs that bitmap at boot.

### IPC

- **Location:** `kernel/src/cap/endpoint.rs` `Endpoint::endpoint_call`
  - **Surface:** pending receive pop on CALL delivery
  - **Reachability:** Cross-process CALL delivered to pending RECV
  - **Classification:** untrusted-input reachable, controlled error
  - **Notes:** The former guarded `pending_recvs.pop_front().unwrap()` now returns a failed capnp error if the queue is inconsistent. Endpoint pending-RECV exhaustion has QEMU coverage in `endpoint-roundtrip`.
- **Location:** `kernel/src/cap/endpoint.rs` `endpoint_restore_recv_front`
  - **Surface:** rollback `push_front` growth
  - **Reachability:** IPC rollback path
  - **Classification:** untrusted-input reachable, controlled error
  - **Notes:** CALL delivery reserves the popped pending-RECV slot until rollback restores the RECV or receiver completion releases the reservation, so concurrent receives cannot consume rollback capacity. Recovery helpers resolve the original endpoint object through revoked cap epochs and wrapper recovery methods bypass liveness checks, without reopening ordinary CALL/RECV/RETURN authority. If restore still fails after reaching the endpoint, the ring path posts or defers an explicit receiver cancellation instead of silently dropping the popped RECV. `endpoint-roundtrip` includes QEMU coverage for same-process CQ-pressure rollback with both available and saturated pending-RECV capacity, then

consuming the restored undersized RECV through the controlled receiver-error path; `capos-lib` host coverage checks revoked-cap recovery lookup.

**Scheduler And Process Lifecycle**

- **Location:** `kernel/src/sched.rs` `register_idle_process_locked`
  - **Surface:** `Process::new_idle().expect("failed to create idle process")`
  - **Reachability:** Boot scheduler init (`sched_init`, slot 0) and lazy per-CPU registration (`current_cpu_idle_thread_locked`)
  - **Classification:** boot-fatal at slot 0; per-CPU-fatal on first AP idle
  - **Notes:** Synthetic idle `Process` creation OOM panics. There is no fallback idle path after the user-mode idle process removal, so this panic is the deliberate unrecoverable-OOM behavior.
- **Location:** `kernel/src/sched.rs` `sched_init`
  - **Surface:** CPL0 idle kernel stack `.expect`, idle-context registry `try_reserve_exact().expect`, per-CPU `CpuContext` `Box::try_new` `panic!`
  - **Reachability:** Boot scheduler init
  - **Classification:** boot-fatal
  - **Notes:** CPL0 idle-context infrastructure OOM panics before services run. Same rationale as the synthetic idle records: no fallback idle path exists, so the failure is deliberately unrecoverable.
- **Location:** `kernel/src/sched.rs` `block_current_on_cap_enter`
  - **Surface:** `current.expect`, idle `assert!`, process-table `expect`
  - **Reachability:** `cap_enter(min_complete > 0)` path
  - **Classification:** untrusted-input reachable, internal invariant
  - **Notes:** Userspace can request blocking, but these unwraps assert scheduler state, not user values. Future process lifecycle/spawn changes increase this exposure.
- **Location:** `kernel/src/sched.rs` `capos_block_current_syscall`
  - **Surface:** `current.expect`, idle `assert!`, table `expect`, `panic!` if not blocked
  - **Reachability:** Blocking syscall continuation
  - **Classification:** untrusted-input reachable, internal invariant
  - **Notes:** Triggered after `cap_enter` chooses to block. User controls the request, but panic requires kernel state inconsistency.
- **Location:** `kernel/src/sched.rs` `run_queue references missing process` `expect` (context-switch + start paths)
  - **Surface:** run-queue/process-table consistency
  - **Reachability:** Scheduling after queue selection
  - **Classification:** trusted-internal now; future spawn/lifecycle sensitive
  - **Notes:** A stale run-queue PID panics. Dynamic spawn/exit must preserve run-queue/process-table invariants.
- **Location:** `kernel/src/sched.rs` `exit_current`

- **Surface:** `current.expect`, idle `assert!`, `processes.remove(...).unwrap()`, next-process `unwrap()`
- **Reachability:** Ambient `exit` syscall and future process exit
- **Classification:** untrusted-input reachable, internal invariant
- **Notes:** Any service can exit itself. Panic requires scheduler corruption or idle misuse, but future spawn/process APIs should harden this boundary.

- **Location:** `kernel/src/sched.rs` `current_ring_and_caps`
  - **Surface:** `current.expect`, process-table `expect`
  - **Reachability:** `cap_enter` flush path
  - **Classification:** untrusted-input reachable, internal invariant
  - **Notes:** User can call `cap_enter`; panic requires no current process or missing table entry.
- **Location:** `kernel/src/sched.rs` `start`
  - **Surface:** initial run-queue `expect`, process-table `unwrap`, CR3 `expect`
  - **Reachability:** Boot service start
  - **Classification:** boot-fatal
  - **Notes:** Manifest with zero services is rejected earlier, and process creation errors out; panics indicate scheduler/CR3 invariant breakage.
- **Location:** `kernel/src/arch/x86_64/context.rs` timer context restore
  - **Surface:** CR3 `expect("invalid CR3 from scheduler")`
  - **Reachability:** Timer interrupt scheduling
  - **Classification:** trusted-internal; future lifecycle sensitive
  - **Notes:** Scheduler should only return page-aligned CR3s from `AddressSpace`.

#### Boot Platform And Memory Setup

- **Location:** `kernel/src/main.rs` `kmain`
  - **Surface:** `assert!(BASE_REVISION.is_supported())`
  - **Reachability:** Limine boot protocol
  - **Classification:** boot-fatal
  - **Notes:** Platform/bootloader contract check.
- **Location:** `kernel/src/main.rs` `kmain`
  - **Surface:** memory-map and HHDM `expect`
  - **Reachability:** Limine boot protocol
  - **Classification:** boot-fatal
  - **Notes:** Missing bootloader responses halt before untrusted services.
- **Location:** `kernel/src/main.rs` `kmain`
  - **Surface:** `cap::init().expect("failed to initialize kernel capabilities")`
  - **Reachability:** Kernel cap table bootstrap
  - **Classification:** boot-fatal
  - **Notes:** Fails on kernel-internal cap-table exhaustion.

- **Location:** `kernel/src/mem/frame.rs init`
  - **Surface:** frame-bitmap region `expect("no region large enough for frame bitmap")`
  - **Reachability:** Boot memory map
  - **Classification:** boot-fatal
  - **Notes:** Bad or too-small memory map halts.
- **Location:** `kernel/src/mem/frame.rs free_frame`
  - **Surface:** `try_free_frame(...).expect("free_frame failed")`
  - **Reachability:** Kernel-owned frame teardown
  - **Classification:** trusted-internal
  - **Notes:** Capability handlers use `try_free_frame`; this panic surface is for kernel-owned frames and rollback/Drop paths.
- **Location:** `kernel/src/mem/frame.rs` HHDM cache helper
  - **Surface:** `assert!(offset != 0, "frame allocator not initialized")`
  - **Reachability:** HHDM cache use before frame init
  - **Classification:** trusted-internal
  - **Notes:** Initialization-order invariant.
- **Location:** `kernel/src/mem/heap.rs init`
  - **Surface:** `alloc_contiguous(HEAP_FRAMES).expect("out of memory for heap")`
  - **Reachability:** Boot heap init
  - **Classification:** boot-fatal
  - **Notes:** Fails if the frame allocator cannot provide the fixed kernel heap.
- **Location:** `kernel/src/mem/paging.rs alloc_page_table_frame` / `kernel_pml4_frame` / `assert!(addr != 0, "paging not initialized")`
  - **Surface:** page-alignment `.unwrap()` / paging initialized `assert!`
  - **Reachability:** Kernel frame/page-table internals
  - **Classification:** trusted-internal
  - **Notes:** `frame::alloc_frame` returns page-aligned addresses.
- **Location:** `kernel/src/mem/paging.rs init_kernel_page_tables`
  - **Surface:** kernel PML4 `expect("failed to allocate kernel PML4")`, page-lookup and map expects
  - **Reachability:** Kernel page-table setup
  - **Classification:** boot-fatal
  - **Notes:** Assumes kernel image is mapped in bootloader tables and enough frames exist.
- **Location:** `kernel/src/arch/x86_64/syscall.rs init`
  - **Surface:** STAR selector `expect("invalid STAR segment configuration")`
  - **Reachability:** Syscall init
  - **Classification:** boot-fatal
  - **Notes:** GDT selector layout invariant.

- **Location:** `kernel/src/sched.rs` context-switch / `exit_current` / `start`
  - **Surface:** CR3 `expect("invalid CR3")`
  - **Reachability:** Context switch/exit/start
  - **Classification:** trusted-internal; future lifecycle sensitive
  - **Notes:** Scheduler should only carry page-aligned address-space roots.

### Audit Method

Candidate sites come from panic-token searches over runtime source plus manual review of nearby indexing and allocation paths on untrusted-input boundaries. The table excludes test-only assertions unless they enforce runtime ABI or layout contracts. Re-run the searches after code changes and classify new sites by reachability, not by token alone.

Search commands:

```
rg -n "\b(panic!|assert!|assert_eq!|assert_ne!|debug_assert!|
debug_assert_eq!|debug_assert_ne!|unwrap\(|expect\(|todo!|unreachable!)"
kernel capos-lib capos-config init demos tools schema system.cue
Makefile docs -g '*.rs' -g '*.cue' -g '*.md' -g 'Makefile'
rg -n "\b(panic!|assert!|assert_eq!|assert_ne!|debug_assert!|
debug_assert_eq!|debug_assert_ne!|unwrap\(|expect\(|todo!|unreachable!)"
kernel/src capos-lib/src capos-config/src init/src demos/capos-demo-
support/src demos/*/src tools/mkmanifest/src -g '*.rs'
```

## DMA Isolation Design

Security Verification Track S.11 gates PCI, virtio, and later userspace device-driver work on an explicit DMA authority model. The immediate goal is narrow: let the kernel bring up a QEMU virtio-net smoke without creating a user-visible raw physical-memory escape hatch.

### Short-Term Decision

Use **kernel-owned bounce buffers** for the first in-kernel QEMU virtio-net smoke.

The first virtio-net smoke stays on this conservative path:

- kernel-owned DMA pages
- kernel-owned virtqueue descriptor tables
- kernel-owned packet buffers
- kernel-programmed physical addresses
- copied packet bytes delivered to the network stack
- no DMA buffer capability exposed to userspace
- no physical address exposed to userspace
- no virtqueue pointer exposed to userspace
- no BAR mapping exposed to userspace

The kernel allocates DMA-capable pages from its own frame allocator, owns the virtqueue descriptor tables and packet buffers, programs the device with the corresponding physical addresses, and copies packet payloads between those buffers and the networking stack.

This is deliberately conservative:

- It works before ACPI/DMAR or AMD-Vi parsing, IOMMU page-table management, MSI/MSI-X routing, and userspace driver lifecycle supervision exist.
- It keeps all physical-address programming inside the kernel, where the same code that allocates the frames also bounds the descriptors that reference them.
- It does not make the current `FrameAllocator` or `MemoryObject` capability part of the DMA path. `FrameAllocator` no longer exposes raw physical addresses, but DMA still needs device-owned buffer objects with IOVA and reset/revoke semantics rather than repurposed general memory caps.
- It gives the smoke a disposable implementation path. When NIC or block drivers move to userspace, bounce-buffer authority becomes a typed `DMAPool` object instead of an ad hoc physical-address grant.

An IOMMU-backed DMA-domain model remains the target for direct device access from mutually untrusted userspace drivers, but it is not a prerequisite for the first QEMU smoke. Without an IOMMU, a malicious bus-mastering device can still DMA to arbitrary RAM at the hardware level; the short-term smoke assumes QEMU-provided virtio hardware and protects against confused or untrusted userspace, not hostile hardware.

### IOMMU Staging

IOMMU support is a deferred-with-known-dependency prerequisite for production hardware claims and for moving direct DMA-capable NIC or block drivers into userspace. capOS now discovers bounded ACPI IOMMU table summaries for Intel DMAR and AMD-Vi/IVRS and records static DMAR DRHD include-all or single-hop PCI endpoint device-scope coverage for retained DMA-capable PCI diagnostics functions. Bridge and multi-hop scopes are retained for diagnostics but do not prove endpoint attachment until PCI topology traversal exists, and include-all fallback fails closed when retained DRHD units or scopes are capped.

The selected QEMU Intel remapping path now programs VT-d root/context and second-level tables for manager-owned `DMAPool` pages, reports bounded fault state, exports only domain-scoped IOVAs, and proves two claimed DMA-capable functions receive distinct per-device domains and second-level roots. It also asserts the production-path S.11.2 hostile-smoke matrix over the active `DMAPool` / `DMABuffer` ledger. The decomposed integration umbrella for this path closed `2026-05-23 23:35 UTC` (`ddf-iommu-remapping-production-closeout`). This is still QEMU-only evidence for the selected path, not a general production hardware-isolation claim: trusted sharing groups, AMD-Vi programming, and production NIC/block userspace driver authority remain future work, and VM shapes without usable remapping hardware remain on the explicit bounce-buffer fallback.

The discovery parser is intentionally shallow and follows the static-table formats documented by the Intel VT-d architecture specification, the AMD IOMMU specification, and QEMU's q35-only `-device intel-iommu` emulation:

Future real remapping work is grounded by the primary-source IOMMU remapping research note, which records Intel VT-d, AMD-Vi, and QEMU sections relevant to table programming, invalidation, fault/status diagnostics, and QEMU-only smoke tests. That note is source grounding only; it does not make the current diagnostics path a real remapping implementation.

- Intel VT-d architecture specification: https://www.intel.com/content/www/us/en/content-details/671081/intel-virtualization-technology-for-directed-i-o-architecture-specification.html
- AMD IOMMU specification: https://docs.amd.com/v/u/en-US/48882_IOMMU
- QEMU manpage: https://www.qemu.org/docs/master/system/qemu-manpage.html?highlight=numa

The staged implementation order is:

1. Discover firmware IOMMU topology from ACPI static tables and fail closed if the tables are malformed, unsupported, or inconsistent with the PCI root complex being used. This first bounded table-discovery step is implemented for DMAR/IVRS summaries only; domain attachment is still planned.
2. Record each DMA-capable PCI function's attachment to an IOMMU unit, or explicitly keep the function on the prototype bounce-buffer-required policy when no trusted IOMMU domain can be created. This reporting step is implemented for retained PCI diagnostics functions when DMAR DRHD include-all or single-hop PCI endpoint device-scope metadata proves PCI segment/BDF coverage. Bridge and multi-hop scopes are not treated as attachment proof until PCI topology traversal exists, and include-all fallback fails closed when retained DMAR coverage metadata is capped; trusted domain creation is still planned.

3. Define and prove the claimed-device domain policy: one device-manager-owned DMA domain per claimed device or trusted sharing group, with all exported device addresses represented as IOVAs scoped to that domain rather than host physical addresses. The selected QEMU Intel path now implements the per-device form for two claimed DMA-capable functions; trusted sharing groups remain disabled and out of scope.
4. Attach `DMAPool` allocation, descriptor validation, MMIO ownership, interrupt ownership, and revocation state to the same device-manager ledger before any doorbell write can make a descriptor visible to hardware.
5. On revoke, reset, or driver death, stop new submissions, remove or invalidate IOMMU mappings before page reuse, and flush the relevant IOTLB state where the hardware model requires it.

Until those gates exist, direct DMA and userspace driver handoff remain blocked. Devices that cannot be placed in a trusted IOMMU domain must stay on kernel-owned bounce buffers or remain unsupported for production claims. This also affects the hostile-smoke gate: S.11.2 smokes must prove that stale DMA handles, stale completions, reset races, and teardown ordering fail closed for IOMMU-backed IOVA mappings, while the process-exit / exit-under-DMA rows remain covered by the selected backend evidence before a cloud or hardware driver can be treated as isolated from the rest of memory.

**Fallback Policy For No Usable IOMMU Exposure**

Some providers or VM shapes may not expose remapping hardware that capOS can trust. That includes absent, malformed, unsupported, capped, or incomplete DMAR/IVRS metadata; scopes that require PCI topology traversal capOS has not implemented yet; and platforms where remapping hardware is unavailable or cannot be programmed safely. Those shapes use a fail-closed fallback policy:

- Direct device DMA remains blocked. `direct_dma_trusted_domains` stays zero and `remapping_tables` stays `not-programmed`.
- Prototype devices that remain enabled use kernel-owned bounce buffers only. The kernel or device manager owns the pages, descriptor validation, physical-address programming, and packet or block-data copies between device-visible memory and non-device memory. General `FrameAllocator` and `MemoryObject` capabilities are not DMA authorities.
- capOS does not expose direct hardware authority for userspace `DMAPool`, `DMABuffer`, `DeviceMmio`, or `Interrupt` in the fallback shape. Result-only `.info` skeletons and bounded manifest grants may report conservative status. The current `DMAPool` manifest grant may allocate and free eight fixed manager-attached, kernel-owned, single-page bounce-buffer `DMABuffer` result caps, with backing pages scrubbed before frame release and no host physical address or IOVA exposed. That narrow fixed-slot allocation/free authority does not map DMA, program device-visible addresses, publish arbitrary CQ entries, program IOMMU/remapping tables, access arbitrary BAR registers or doorbells, or own hardware interrupt acknowledgement, mask, or unmask. The selected provider-TX proof is the current bounded exception: after the same manager-owned `DMABuffer` authority and bounce-scrub gates, queue `1` may publish the full selected TX queue-depth descriptor/avail window into the existing kernel-owned virtio-net TX ring before the first completion, ring one selected notify doorbell per accepted provider descriptor through the live no-write `notify_mmio` policy, and hand those

bounded completions back through descriptor/generation-matched `DMABuffer.completeDescriptor` plus live `tx_interrupt.wait` completion events. The same selected path can also use `tx_interrupt.mask/unmask` to toggle only the selected TX MSI-X table vector-control bit and matching route state after live issue-id and route validation, and can retire one deferred LAPIC EOI for each delivered selected TX used-ring completion event, with `Interrupt.acknowledge` returning ABI-visible provider CQ/ack ledger fields plus hardware dispatch ack count, delta, token, and mutation flag for that bounded pairing. Full-queue QEMU bursts that coalesce selected TX MSI-X delivery use a bounded `INT $vector` proof hook only while the virtio TX completion path has an active full-window coalescing budget, so the selected IDT handler and deferred-EOI path remain observable without claiming full production IRQ ownership. Successful selected queue `1` `DMABuffer.completeDescriptor`, `tx_interrupt.wait`, and `tx_interrupt.acknowledge` results also carry bounded CQ event identity: sequence, queue, descriptor id, slot, slot generation, software descriptor generation, completion length, provider issue id, source id/generation, and route generation. Pre-event, duplicate ack, masked-route ack, wrong-order completion, teardown-drain, stale issue after release/regrant, reset, and stale-after-release paths keep that identity empty and do not mutate the bounded identity queue. Provider TX release also retires delivered but unacknowledged bounded CQ events for the live issue before clearing that issue: the stale post-release ack path is revoked, and the release proof records seven pending provider completion acks and their deferred EOIs as release-retired. The same selected path also has a bounded teardown-only drain for seven incomplete provider-published TX descriptors while one completed descriptor remains live: release may explicitly drain only the incomplete matching used-ring entries, retire those allocation-backed device-DMA TX queue ledgers, and free only after manager in-flight state is drained, without publishing provider CQ/IRQ events or issuing `DMABuffer.completeDescriptor` results. The paired provider RX bootstrap grant can now validate the live RX issue and selected virtio-net RX route before toggling only the selected RX MSI-X table vector-control bit and route state, and it can complete one selected-route RX `Interrupt.wait` after a delivered RX MSI-X/LAPIC dispatch. The paired `Interrupt.acknowledge` accounts exactly one RX dispatch token and retires one deferred LAPIC EOI for that delivered zero-CQ RX event; pre-event, masked-route, duplicate, and stale-after-release paths fail closed without mutating delivery or acknowledgement state. RX descriptor accounting and RX CQ ownership remain bounded to the synthetic proof path, and full hardware IRQ ownership remains blocked. These exceptions do not transfer full virtio-net ownership, direct DMA, IOMMU authority, arbitrary doorbells, production NIC/storage authority, or cloud readiness.

- capOS does not claim hostile-hardware isolation for those shapes. A malicious or compromised bus-mastering device without a trusted remapping domain can still write arbitrary RAM at the hardware level. The fallback is acceptable only for prototype devices and trusted emulator or provider shapes where that hardware threat is outside the claim; otherwise the device remains unsupported.
- Before any userspace driver path can rely on DMA or IRQ authority, S.11.2 hostile smokes must pass for the selected backend. That includes stale DMA handles, stale completions, descriptor abuse, revoke/reset races, stale IRQs, teardown-under-DMA for IOMMU-backed IOVA mappings, and exit-under-DMA for the fallback bounce-buffer path when the fallback is used.

This fallback policy is separate from current diagnostics-only IOMMU metadata coverage and from future real remapping-domain integration. Diagnostics can report static firmware-table coverage for a PCI function, but unless capOS creates a device-manager-owned remapping domain and programs mappings, the active direct-DMA policy remains blocked. Future real integration must attach `DMAPool`, `DeviceMmio`, `Interrupt`, ledger teardown, mapping removal or invalidation, and required IOTLB flushes to the same ownership transaction before a direct-DMA trusted-domain count can become nonzero.

### DMA Assurance Model Checked Evidence And Cloud Backend Inputs

The DMA assurance model records the claim boundary and checked bounded evidence for DMA authority; the cloud backend contract it feeds is authoritative and lives in the "Cloud DMA Backend" section below: DMA Assurance Model and `models/dma/`. It is a design/evidence scaffold, not a new production hardware gate by itself. The checked gates are `make model-dma-tla`, `make model-dma-alloy`, `make kani-dma-authority`, and `make model-dma-deferred-completion-loom`; `make dma-assurance-model-check` aggregates them locally, while GitHub CI runs the Alloy/TLA+/Loom gates in `dma-assurance-models` and the Kani gate in `kani-proofs`. The operationalization track that reconciled the skeletons against landed DMA code is tracked in Security And Verification Backlog ("DMA Assurance Model Operationalization").

Cloud NIC/storage work must use the model as the checklist for backend selection. Backend selection is a runtime, fail-closed decision the kernel makes on each boot, with an optional operator override declared in the system manifest; it is not a per-VM-shape safety assertion that a person signs off. The authoritative selection rule and the manifest override contract are defined in the "Cloud DMA Backend" section below.

Cloud backend evidence must separate provider-side DMA isolation from guest-controlled remapping authority. SR-IOV, virtual NIC, GPU, accelerator, or local NVMe support can identify a DMA-capable surface, but it is not enough to claim direct-DMA isolation. A direct-remapping backend needs guest-visible IOMMU or equivalent translation authority that capOS can discover and program. The cloud evidence matrix must record provider API or documentation sources, retrieval date, region or zone, instance type, image and kernel, live guest PCI/device probes, IOMMU table/group observations, and maintenance or device revocation behavior as the support-policy record for advertised targets. The runtime probe, not this matrix, makes the binding per-boot selection.

The matrix does not replace runtime selection. capOS must choose the safest backend on each boot from what it can actually observe and validate. Direct remapping is enabled only when guest-programmable remapping authority is present and passes the selected self-tests. A provider-remapped or bounce path is selected only when direct DMA remains blocked and device-visible memory can stay manager-owned. Ambiguous, contradictory, or unvalidated observations select `Unsupported`.

The backend candidates are:

- **Direct remapping domain.** The provider shape must expose guest-programmable remapping hardware; capOS must discover and program a device-manager-owned domain for the target device; descriptor publication must be ordered after mapping; and teardown must remove mappings, observe required invalidation completion, and scrub before page reuse. The selected

path must carry stale-handle, stale-completion, descriptor-abuse, revoke/reset-race, teardown-under-DMA, no-host-physical-exposure, and cross-domain alias evidence.

- **Labeled bounce-buffer fallback.** Direct DMA stays blocked, device-visible memory remains manager-owned bounce pages, host physical addresses and generic `MemoryObject` authority stay hidden from the driver, and stale handle/completion/teardown evidence covers the selected fallback. This path must keep `hostile_hardware_isolation=not-claimed` unless separate per-domain remapping evidence justifies a stronger provider-specific claim.
- **Unsupported.** Devices whose DMA behavior cannot satisfy either candidate stay unbound or disabled. A serial boot result or PCI enumeration line is not enough to claim cloud NIC/storage readiness.

Downstream cloud driver preflights must declare the candidate backend and map their evidence to the assurance model's invariants: no host-physical exposure, mapping before publication, no page reuse before teardown, stale-handle and stale-completion fail-closed behavior, domain-scoped aliasing only, bounded fail-closed holds, and explicit backend evidence. The evidence matrix is a support-policy record of advertised targets; the runtime probe, not the matrix, selects the backend on each boot.

### Cloud DMA Backend

This section is the authoritative contract for how capOS selects a DMA backend for cloud NIC/storage devices. Selection is a runtime, fail-closed decision the kernel makes on each boot from what it can actually probe and validate, with an optional declarative override in the system manifest. There is no human sign-off in the selection path: the runtime probe decides by default, and the manifest override is config that an operator sets for a deployment, not a doc-signing ritual gated on any specific person. Downstream cloud NIC/storage driver slices consume this contract directly as their DMA-backend authority.

The preceding "DMA Assurance Model Checked Evidence And Cloud Backend Inputs" section defines the three backend candidates; this section adds the per-candidate trade-off analysis, the runtime selection rule, the manifest override field, and the downstream-contract scaffolding that a cloud NIC/storage driver declares. The research substrate is the provider evidence inventory Cloud DMA Provider Evidence Inventory, and the invariants and tool mapping are in DMA Assurance Model.

#### Provider-Written Addresses And No-IOMMU Brokered Bounce

Two DMA-address ownership models can be valid, but they do not apply to the same backend.

- **Provider-written, kernel-validated addresses** (the NVMe Model B validator) are valid only when the provider's device-visible address is not a host physical address: a verified direct-remapping/vIOMMU domain-scoped IOVA, or a future synthetic software address namespace that the manager translates before hardware sees it.
- **Brokered address publication** is the no-IOMMU bounce-buffer model. The provider may own protocol state and buffer capabilities, but the kernel or device manager writes device-visible queue-base, PRP/SGL, or virtqueue address fields because those values are host physical or bus addresses on current no-IOMMU hardware.

Correction recorded 2026-05-27: the earlier reconciliation that treated a no-IOMMU bounce window as a provider-visible, non-host-physical device address space is not valid for the current implementation. On the `run-pci-nvme` no-IOMMU shape, `DeviceDmaAllocation` carries host physical pages and the reviewed IOVA export discipline keeps userspace IOVA/host-physical export disabled. Therefore a provider-written NVMe queue base or PRP on that gate would export a host physical address, violating the no-host-physical-exposure invariant. A bounce buffer protects data ownership and copy discipline; it does not create an untrusted-driver-safe IOVA namespace by itself.

The kernel on-notify DMA validator (`kernel/src/cap/nvme_doorbell_validator.rs`, `validate_doorbell_scan`) remains useful evidence for the provider-written model. On a queue-arm/`CC.EN` write and on an SQ tail doorbell it scans the device-visible addresses the provider published (queue bases; PRP1/PRP2 and one level of PRP-list indirection) and fails closed before the doorbell takes effect on any address that is not wholly within a window granted to the doorbell claim's owner at the live generation: out-of-window, host-physical, cross-owner-alias, region-overrun, unaligned, deeper-than-one-level PRP chain, or stale generation (`ScanReject`). Owner identity and live generation come from the grant ledger, never from provider-supplied metadata. A completion whose submission scan was never validated, or was validated under a now-retired generation, does not wake a waiter (`completion_wakes_waiter`), matching the stale-completion gate on the virtio-net path. That mechanism is the right fit for the QEMU direct-remapping lane and any future cloud shape that exposes guest-programmable remapping.

For the current GCP/no-IOMMU target, the storage path must use brokered bounce: userspace supplies typed commands, queue ownership intent, and live `DMABuffer`/buffer-cap handles; the manager materializes the actual device-visible queue-base and PRP/SGL fields and orders publication, teardown, copy, and scrub. That still leaves protocol-specific NVMe logic in userspace, but it does not let userspace author raw device addresses.

The brokered admin-queue enable landed 2026-05-27 (`nvme-no-iommu-brokered-controller-enable`, `device_manager::nvme_brokered_admin_queue_enable`). The provider allocates the admin submission/completion queue pages through its `DMAPool` cap and requests enable through the `CC` selected-write claim (`CC.EN` set); the manager resolves those pages from the live ledger (`proof_buffers` slots 0/1), validates the authored bases through `validate_doorbell_scan` (`ScanKind::QueueArm`), and authors `AQA`/`ASQ`/`ACQ` plus the `CC.EN` write itself. The provider never receives the host physical / device-visible queue-base address; `CSTS.RDY=1` is observed only through brokered reads. This is the brokered model applied to the admin queue-arm; the steady-state SQ-tail doorbell over provider-written PRPs still needs the direct-remapping/synthetic-address lane above. Proof `make run-pci-nvme`; provenance `docs/devices/nvme.md` §6.

### Provider-Side Isolation Versus Guest-Programmable Remapping

The decisive distinction for backend selection is between a *DMA-capable surface* and *guest-programmable remapping authority*:

- SR-IOV (AWS ENA, Azure Accelerated Networking VF), a virtual NIC (gVNIC, virtio-net), a GPU/accelerator, or local NVMe identifies a device that does or could bus-master. This is a DMA-capable surface, not a safety property.

- A **direct-remapping** classification requires a usable Intel VT-d, AMD-Vi, or Arm SMMU unit that the guest can discover, program, and validate, with translation/fault/invalidation behavior matching IOMMU Remapping Grounding. A DMA-capable surface alone never implies this.
- Provider-side isolation facts (host-enforced VPC isolation, Nitro/host data-path bypass, hypervisor-side IOMMU) are **support-policy assumptions** a guest cannot prove from inside, not evidence that capOS can safely program direct DMA.

Runtime probing is authoritative for selecting the safe backend on a particular boot: capOS chooses from the device inventory, the remapping authority it can actually program, driver self-tests, and fail-closed probe results, and unknown or contradictory observations select the labeled bounce-buffer path or `Unsupported`. The cloud VM evidence matrix is the separate support-policy record for advertised targets and provider assumptions a guest cannot fully prove by itself; it does not override the boot-time runtime selection.

**Candidate Trade-Off Analysis**

- **Dimension:** IOMMU coverage requirement
  - **Direct remapping domain:** Requires a guest-programmable VT-d/AMD-Vi/SMMU unit capOS can program and validate per device.
  - **Labeled bounce-buffer fallback:** None: used precisely when no usable guest IOMMU is exposed.
  - **Unsupported:** N/A: device stays unbound.
- **Dimension:** Cloud VM shape coverage (per inventory)
  - **Direct remapping domain:** No probed GCE shape exposes a guest-programmable IOMMU; AWS/Azure shapes not yet probed. So no probed shape currently qualifies.
  - **Labeled bounce-buffer fallback:** Indicated for shapes with a DMA-capable surface but no guest IOMMU (the probed GCE rows); fail-closed default for unproven shapes with a manager-ownable surface.
  - **Unsupported:** Ambiguous, contradictory, or unvalidated observations.
- **Dimension:** Per-operation cost
  - **Direct remapping domain:** Translation only; no data copy. IOTLB/context-cache invalidation on teardown.
  - **Labeled bounce-buffer fallback:** Copy between device-visible bounce pages and non-device memory on every transfer; Confidential VMs force this in hardware regardless.
  - **Unsupported:** None.
- **Dimension:** Hostile-smoke coverage today
  - **Direct remapping domain:** Bounded QEMU Intel path only (`make run-iommu-remapping`, `ddf-iommu-remapping-production-closeout`); no cloud guest-IOMMU evidence.
  - **Labeled bounce-buffer fallback:** S.11.2.7/8/9 rows enforced by the `make run-net` gate (`tools/qemu-net-smoke.sh`), with bounce-buffer virtio-net provider evidence in `ddf-provider-virtio-net-driver-closeout`; bounce-buffer `DMAPool` lifecycle by `make run-dmapool-grant`. The GCP-shape local binding precursor (`cloud-gcp-virtio-net-local-qemu-binding`) asserts, in both `make run-net` and `make run-ddf-provider-consumer`, that the enumerated/bound function matches the documented GCP 1st/2nd-gen virtio-net device

surface (standard virtio-net, vendor `0x1af4`) and that the resolved backend is this labeled bounce-buffer path; it does not claim live GCP enumeration.
  - **Unsupported:** N/A.
- **Dimension:** Hostile-hardware isolation claim
  - **Direct remapping domain:** Claimable only with per-domain remapping evidence and the IOMMU hostile smokes; not yet established for any cloud shape.
  - **Labeled bounce-buffer fallback:** `not-claimed`: a malicious bus-mastering device without a trusted remapping domain can still write arbitrary RAM.
  - **Unsupported:** N/A.

The GCE live-probe rows in the evidence inventory record that every probed GCE shape (`1`st-gen `n1`, `2`nd-gen `e2`, `3`rd-gen Intel `c3`, and AMD-SEV Confidential `n2d`) boots with `intel_iommu=off`, `DMAR: IOMMU disabled`, SWIOTLB software bounce buffering, empty `/sys/kernel/iommu_groups`, and no DMAR/IVRS/IORT table; the Confidential shape forces bounce buffering as a memory-encryption invariant. These rows are support-policy expectations, not a hardcoded selection table: they describe what capOS's runtime probe should expect to find on those shapes today, and they confirm that the fail-closed default lands on the labeled bounce-buffer path there. The boot-time probe, not this matrix, makes the binding selection on each boot, so a shape whose IOMMU exposure changes is handled by the probe re-evaluating rather than by editing this text. AWS and Azure shapes carry no live-probe evidence yet; the probe treats them the same as any other unproven platform and defaults to the bounce-buffer path.

#### Runtime Selection Rule (Fail-Closed Default)

On each boot, capOS probes the platform for guest-programmable remapping authority — IOMMU presence, programmability, and coverage for the DMA-capable functions it intends to bind — and selects the backend fail-closed:

1. Probe the platform. Discover DMA-capable functions, then test whether a usable Intel VT-d, AMD-Vi, or Arm SMMU unit is present, discoverable, and programmable, and whether its translation/fault/invalidation behavior passes the self-tests in IOMMU Remapping Grounding.
2. Select fail-closed. Select the **direct remapping domain** backend for a device only when the probe positively verifies a usable+safe IOMMU for that device. If the probe cannot verify it — IOMMU absent, not programmable, coverage unproven, self-test failed, or observations ambiguous — select the **labeled bounce-buffer fallback** for any DMA-capable surface the manager can keep manager-owned, or **Unsupported** when even that cannot hold. This probe-gated rule is the default: on an unproven platform the probe cannot verify, so direct DMA is not used and the bounce-buffer path is chosen.
3. There is no human in this loop. The machine decides per boot; the only external authority is the optional manifest override below.

This is the boot-time authority. The cloud VM evidence matrix above is the support-policy expectation of what the probe should find, not the decision itself.

#### Manifest Override Field (Operator Authority Lever)

An operator can override the runtime default for a deployment through one declarative, auditable enum field in the system manifest's kernel parameters: the `dmaBackendPolicy` field of the

`SystemConfig` struct in `schema/capos.capnp`. It is config, not a doc-signing ritual, and is not gated on any specific person. The field is absent by default, and the absent default applies the probe-gated runtime selection rule above. The enum values and their interaction with the probe result are:

- **Value:** *(field absent)*
  - **Probe verifies usable+safe IOMMU:** direct remapping domain
  - **Probe cannot verify:** bounce-buffer fallback
  - **Notes:** Default: the probe-gated runtime selection rule. Direct-DMA when the probe verifies a usable+safe IOMMU, bounce-buffer fallback otherwise (fail-closed). Identical to `enable-if-verified`.
- **Value:** `enable-if-verified`
  - **Probe verifies usable+safe IOMMU:** direct remapping domain
  - **Probe cannot verify:** bounce-buffer fallback
  - **Notes:** The explicit, auditable form of the default. Probe-gated direct-DMA with fail-closed bounce-buffer fallback. Redundant with the absent default but kept for explicit configuration.
- **Value:** `enable-unsafe`
  - **Probe verifies usable+safe IOMMU:** direct remapping domain
  - **Probe cannot verify:** direct remapping domain
  - **Notes:** Force direct-DMA even when the probe cannot verify it. The operator takes responsibility for the platform's DMA isolation; the value name carries the warning. Use only on a platform whose isolation is known-good out of band.
- **Value:** `bounce-buffer`
  - **Probe verifies usable+safe IOMMU:** bounce-buffer fallback
  - **Probe cannot verify:** bounce-buffer fallback
  - **Notes:** Pin the labeled bounce-buffer path and disable direct-DMA entirely, even where the probe would verify a usable IOMMU. The most conservative value.

Selection rules that hold for every value:

- The absent default and `enable-if-verified` select direct DMA only when the probe verifies a usable+safe IOMMU, and otherwise fall back to bounce-buffer.
- `enable-unsafe` is the sole value that can pick direct DMA without probe verification. The value name is the acknowledgement; there is no separate per-shape "I-accept-unverified" ceremony.
- `bounce-buffer` never selects direct DMA, even where the probe would verify a usable IOMMU.
- When the selected backend is `Unsupported` for a device (no manager-ownable DMA-capable surface at all), the device stays unbound regardless of the override value. The override governs direct-vs-bounce, not whether an unbindable device is forced online.

This selection mechanism is implemented. The `dmaBackendPolicy` capnp enum encodes an absent field as ordinal `0` (`unspecified`), which decodes identically to `enable-if-verified`; an unrecognized ordinal decodes fail-closed to the bounce-buffer path (never direct DMA) rather than failing the manifest parse or honoring the probe-gated default. The kernel resolves the

backend on each boot from the IOMMU probe verdict and this override and emits a boot proof line of the form dma: backend selection dma_backend=<direct-remapping|bounce-buffer> dma_backend_override=<absent|enable-if-verified|enable-unsafe|bounce-buffer> probe_verified_usable_iommu=<bool>. The bounded QEMU shapes prove the probe-gated default end-to-end: make run-iommu-remapping (verifiable Intel VT-d shape) records dma_backend=direct-remapping dma_backend_override=absent, and make run-dmapool-grant (no usable IOMMU) records dma_backend=bounce-buffer dma_backend_override=absent. The override values and the unknown-ordinal fail-closed decode are covered by cargo test-config over the shared selection rule, capnp round-trip, and CUE decode.

The production cloud (non-qemu) build emits the same backend selection on the cloudboot harness's serial-port-1 path as cloudboot-evidence: dma-backend bounce_buffer (in the harness token namespace), and on the bounce-buffer verdict drives the production-path bounce-buffer DMAPool + DMABuffer grant proof (kernel/src/cap/dmapool_bounce_buffer_grant_proof.rs, called from kernel::run_init): stage a parked manager-attached DMAPool over one DMA-capable PCI function from the inventory through device_manager::stage_bounce_buffer_dmapool_record (kernel/src/device_manager/stub.rs), allocate one bounded bounce-buffer DMABuffer through device_manager::issue_manager_attached_dmabuffer_handle_with_request (which calls device_dma::allocate_manager_attached_dmapool_bounce_buffer_page), assert the cap-info labels (userspace_dmapool=manager-issued-bounce-buffer, allocation=single-bounce-buffer-page, real_dma=not-attempted, direct_dma=blocked, host_physical_user_visible=0, iova_export=disabled-future-only), quiesce-before-release (release_dmapool_record_for_cap_release returns pending-buffer-release while the buffer is live), scrub-before-reuse (the released bounce-buffer frame is zeroed in place between scrub and frame-free), and stale-handle-after-detach, then emit cloudboot-evidence: dma-pool-grant <token> with shape <seg>.<bus>.<dev>.<fn>-pool.<slot>.gen.<gen>-phys.<hex> (every character inside the harness grammar [A-Za-z0-9._-]+). The proof stages a pool, allocates one bounce-buffer page, asserts the invariants, and emits a marker: it does **not** program real DMA, attach a queue, program interrupts, claim a device for sustained ownership beyond the grant, or emit provider-nic-bound / storage-bound. A future direct-remapping verdict skips the proof rather than aliasing direct-DMA onto the bounce-buffer assertion shape.

Implementation note, 2026-05-29 11:50 UTC: the production-cloud bounce-buffer stub now implements the cap-side DMABuffer.map(R+W) / DMABuffer.unmap admission and state-machine entry points (validate_dmabuffer_map_admission, record_dmabuffer_user_mapping, begin_dmabuffer_user_mapping_unmap, restore_dmabuffer_user_mapping, clear_dmabuffer_user_mapping in kernel/src/device_manager/stub.rs) so a userspace holder of a manager-issued DMABuffer cap can map the single bounce-buffer page read/write, write payload bytes through that mapping, unmap it, and observe DMAPool.info.mapped_vmas reflect the live mapping (make run-cloud-dmapool-grant). The same Absent -> Mapped -> Unmapping -> Absent state machine the QEMU side enforces governs the parked slot: a duplicate map fails closed with the DmaPoolLive shape, a clear before the cap-side aspace unmap returns DmaPoolTeardownEvidenceInvalid, an identity-mismatched begin/clear fails closed, and a post-freeBuffer map fails closed with the DmaBufferStaleHandle shape. The manager remains the single owner of the bounce-buffer page's device-visible host-physical address and IOVA: the

mapping does not expose either to userspace, real DMA stays `not-attempted`, direct DMA stays `blocked`, and IOVA export stays `disabled-future-only`. This is a local-QEMU proof of the userspace mapping path only; it does not unlock a live cloud NIC bind, IOMMU programming, or production direct-DMA authority.

Implementation note, 2026-06-03: Phase C slice 2 (`cloud_virtio_net_userspace_ownable_vring_proof`, `make run-cloud-prod-nic-driver-userspace-ownable-vring`) wires this landed bounce-buffer authority to a userspace virtio-net driver's **own vring** without adding a new isolation backend. The driver allocates its descriptor / available / used ring pages through a granted `DMAPool`, and the kernel programs the virtio queue-address registers (`queue_desc` / `queue_driver` / `queue_device`) with the manager-owned bounce host-physical address. The brokered-address- publication model (above) holds: the driver never authors a device address. It writes an opaque per-buffer device-usable handle (exported through `DMABuffer.info.deviceIova` with scope `bounce-handle`, a deterministic non-address encoding of the buffer's manager identity), and the kernel resolves that handle against the live grant ledger (`device_manager::stub::resolve_virtio_vring_device_address`) to the real host-physical address before any MMIO write. The no-host-physical-exposure invariant is preserved end-to-end: `host_physical_user_visible=0`, `iova_export=disabled-future-only`, and reads of the queue-address base registers are refused so the resolved address is never read back into userspace. Out-of-grant, host-physical-looking, and stale-generation handle writes fail closed, mirroring the NVMe doorbell-scan reject classes. `queue_enable` / `DRIVER_OK` stay fail-closed (slice 3); this is a local-QEMU proof of the userspace-ownable-vring path only, not a live cloud NIC bind.

**Downstream-Contract Scaffolding**

A cloud NIC/storage driver declares its chosen backend through the device-manager policy fields that already label the local paths in this design. The contract below scaffolds the values each candidate requires; it is the shape a driver preflight declares, not an authorization to enable any value.

- **Policy field:** `direct_dma`
  - **Direct remapping domain:** `enabled` for the programmed per-device domain
  - **Labeled bounce-buffer fallback:** `blocked`
  - **Unsupported:** `blocked`
- **Policy field:** `trusted_domain`
  - **Direct remapping domain:** manager-owned domain id for the device
  - **Labeled bounce-buffer fallback:** `none`
  - **Unsupported:** `none`
- **Policy field:** `bounce_buffer`
  - **Direct remapping domain:** not required (mapped IOVA path)
  - **Labeled bounce-buffer fallback:** `required`
  - **Unsupported:** N/A (device unbound)
- **Policy field:** `remapping_tables`
  - **Direct remapping domain:** `programmed`

- **Labeled bounce-buffer fallback:** `not-programmed`
- **Unsupported:** `not-programmed`

- **Policy field:** `hostile_hardware_isolation`
  - **Direct remapping domain:** claimed only with per-domain remapping evidence + IOMMU hostile smokes
  - **Labeled bounce-buffer fallback:** `not-claimed`
  - **Unsupported:** N/A
- **Policy field:** `exported_device_addresses`
  - **Direct remapping domain:** `iova-only`, domain-labeled
  - **Labeled bounce-buffer fallback:** none (no host physical or IOVA exposed)
  - **Unsupported:** none

Each candidate must satisfy the following gates, mapped to the assurance-model invariants in DMA Assurance Model:

- **Direct remapping domain** — mapping before publication; no page reuse before teardown, with mapping removal and observed invalidation completion ordered strictly before scrub/reuse; stale-handle and stale-completion fail-closed; domain-scoped aliasing only (per-device context entries where a peer domain cannot resolve another domain's IOVA); no host-physical exposure (export only the domain-scoped IOVA, labeled as meaningless outside the domain); backend evidence explicit. The only landed remapping-domain evidence is the bounded QEMU Intel path in `ddf-iommu-remapping-production-closeout`, exercised by `make run-iommu-remapping`. A cloud shape needs its own guest-programmable remapping evidence before this candidate applies to it.
- **Labeled bounce-buffer fallback** — `direct_dma=blocked`; all device-visible memory is manager-owned bounce pages; no host physical address and no generic `MemoryObject/FrameAllocator` authority is exposed to the driver; stale-handle, stale-completion, and exit-under-DMA teardown fail closed; `hostile_hardware_isolation=not-claimed` stays explicit. The landed evidence is the S.11.2.7/8/9 hostile-smoke rows enforced by the `make run-net` gate (`tools/qemu-net-smoke.sh`), with the bounce-buffer virtio-net provider evidence in `ddf-provider-virtio-net-driver-closeout`, plus the bounce-buffer `DMAPool` grant lifecycle in `make run-dmapool-grant`. As of `ddf-real-dma-s112-hostile-smokes`, the S.11.2.7 stale-IRQ-after-reset and S.11.2.8 stale-DMA-completion-after-reset closure summaries are asserted on the dedicated `make run-dmapool-grant` gate as well, so the DMA-grant gate fails closed on a hostile-row regression without depending on `make run-net`; exit-under-DMA (in-flight drain, scrub, page free) is enforced by `make run-dmapool-grant-exit`. See the "Fallback Policy For No Usable IOMMU Exposure" and "Hostile-Smoke Acceptance Matrix" sections in this document for the full gate text.
- **Unsupported** — the device stays unbound or disabled; no driver-visible DMA, MMIO doorbell, interrupt ownership, or storage/network readiness claim is made. A serial boot line or PCI enumeration line is not readiness evidence.

The `models/dma/` TLA+ and Alloy files, the extracted Kani core, and the focused Loom model are checked bounded evidence for these invariants. They supplement the QEMU/host evidence above; they do not satisfy a candidate hardware or cloud backend gate by the mere presence of

model files. Each checked result records its tool version, command, bounds, and output in `../models/dma/README.md`, and the CI placement is tracked in Security And Verification Backlog.

**First QEMU Intel Remapping Smoke Acceptance Gate**

Decision recorded `2026-05-14 09:07 UTC`. The first slice that programs real Intel VT-d remapping state under QEMU has an explicit, bounded acceptance gate. This unblocks the `ddf-iommu-qemu-intel-remapping-smoke` task, whose `## Acceptance` section carries the full gate; the summary here is the design-level decision.

The first slice programs the minimum Intel VT-d path for exactly one selected DMA-capable function under a pinned QEMU q35 `-device intel-iommu,aw-bits=39` shape: one root entry, one context entry bound to a device-manager-owned domain ID and a 39-bit second-level page-table root, and a single second-level mapping from one device-visible IOVA page to one kernel-owned `DMAPool` page, plus the root-table-address register write and the global-command/global-status translation-enable handshake. Acceptance requires observable proof that the mapped IOVA was translated and that an out-of-domain IOVA faults closed in the fault-status/fault-recording registers.

Invalidation is part of the gate, not a follow-up. On revoke, device reset, driver death, and `DMAPool` page release, the slice must remove the second-level mapping and invalidate the relevant context-cache and IOTLB state through the selected invalidation interface, observe invalidation completion, and order page scrub/reuse strictly after that completion. This sits at the existing "remove IOMMU mappings" and "scrub and free DMA pages" steps of the `DeviceOwnerState` revocation order and must not be reachable before `QueuesQuiesced` or a completed `Resetting` transition.

IOVA export discipline: host physical addresses stay hidden from userspace in every result cap, diagnostic line, and audit record. The selected QEMU Intel IOMMU-backed `DMABuffer.info` path may export only the domain-scoped IOVA for the live mapped buffer generation, explicitly labeled as meaningless outside that domain; fallback bounce-buffer paths keep IOVA export disabled.

Per-device domain granularity: the selected QEMU Intel path programs two distinct per-device context entries and second-level roots for two claimed DMA-capable functions under the same DRHD. Both domains may use the same IOVA, but the peer domain's second-level walk is proven not to resolve that IOVA to the primary page; stale and wrong-owner domain assignment fail closed, and trusted multi-device sharing groups remain disabled.

The kernel-owned bounce-buffer fallback stays the path for VM shapes without usable remapping hardware and must remain explicitly labeled (`fallback_policy=kernel-owned-bounce-buffer-only`, `remapping_tables=not-programmed`, `hostile_hardware_isolation=not-claimed`); it must never be silently reinterpreted as direct-DMA or hostile-hardware isolation. The IOMMU-backed path adds stale-DMA-handle, stale-completion, descriptor-abuse, revoke/reset-race, teardown-under-DMA, cross-domain stale-handle, and fail-closed teardown branch hostile smokes while the existing bounce-buffer exit-under-DMA and stale-DMA evidence (the `device-dma` stale-handle and stale-completion proofs and the S.11.2.7/S.11.2.8 closure summaries) is preserved unchanged.

Explicitly out of scope for this first slice: AMD-Vi table programming; trusted multi-device sharing groups; scalable-mode context entries; interrupt remapping and device-IOTLB options; 48-bit IOVA space / 4-level tables; production NIC or storage driver ownership; userspace `DMAPool` direct-DMA authority; and moving the live virtio-net path off bounce buffers. The acceptance evidence is QEMU-only emulator evidence, not a hardware-isolation claim. The smoke adds or selects a focused `make run-iommu-remapping` gate asserted by `tools/qemu-iommu-remapping-smoke.sh`.

Implementation note, 2026-05-02 04:58 UTC: the ACPI discovery path recognizes DMAR and IVRS in the root table walk, reports absent/valid/malformed/unsupported state, records bounded table length/header facts, DMAR host address width and flags, IVRS IVinfo/flags, and bounded remapping-structure type counts. Malformed DMAR or IVRS structure lengths stop parsing, and unsupported shapes such as parser scan-cap overflow leave `direct_dma=blocked` with `bounce_buffer=required`.

Implementation note, 2026-05-02 05:31 UTC: the attachment-policy slice also retains DMAR DRHD include-all and bounded PCI endpoint device-scope metadata, including segment, single-hop BDF, and remapping-hardware register base, and reports each retained DMA-capable PCI function as IOMMU-attached/covered when that static table metadata covers its segment/BDF. Bridge and multi-hop scopes are diagnostic-only until PCI topology traversal can resolve them, and include-all fallback fails closed when retained DRHD units or scopes are capped. Functions without trusted static coverage are reported as uncovered; covered functions are reported as attached/covered, but both paths keep `dma_policy=prototype-bounce-buffer-only`, `bounce_buffer_required`, and `blocked_direct_dma_devices` because remapping domains are unsupported. The direct-DMA trusted-domain count remains zero, and userspace `DMAPool`, `DeviceMmio`, and `Interrupt` authority remain unavailable.

Implementation note, 2026-05-02 07:27 UTC: the domain-policy staging slice adds a `pci: dma-domain policy proof` line and a diagnostics mirror. The proof reports the future domain owner as the device manager, the domain granularity as per-device or trusted-sharing-group, exported device addresses as IOVA-only, `host_physical_user_visible=0`, `direct_dma_trusted_domains=0`, `claimed_device_domains_ready=0`, `remapping_tables=not-programmed`, `remapping_domains=not-started`, userspace `DMAPool/DeviceMmio/Interrupt` as `not-started`, and prototype devices as kernel-owned bounce-buffer-only. Malformed, unsupported, absent, or retained-capped metadata leaves direct DMA blocked; `proof_result=ok` is only evidence for that conservative blocked-direct-DMA policy.

Implementation note, 2026-05-09 18:47 UTC: the blocked-direct-DMA admission decision now lives in the pure `capos-lib::device_authority` validator next to the DMA/MMIO/IRQ handle validators. Host tests cover the current all-prototype bounce-buffer shape, fail-closed results if any direct trusted domain is claimed before the policy is ready, fail-closed results if the prototype bounce-buffer count does not cover every DMA-capable function, and the absent, malformed, unsupported, and retained-capped metadata labels. The kernel PCI proof line and diagnostics mirrors consume that pure decision while preserving the existing `direct_dma=blocked`, `remapping_tables=not-programmed`, `domain_activation=not-started`, and `policy=blocked-direct-dma` labels. This is IOMMU/remapping groundwork only; it does not program remapping

tables, create trusted domains, expose host physical addresses, or enable production userspace DMA authority.

Implementation note, 2026-05-02 15:29 UTC: the COM1 `devices` diagnostics command now prints the same bounded DMA-domain policy facts without naming an owner identity. The line explains that all current DMA-capable prototype functions remain on `direct_dma=blocked`, `bounce_buffer=required`, `direct_dma_trusted_domains=0`, `claimed_device_domains_ready=0`, `remapping_tables=not-programmed`, `exported_device_addresses=iova-only`, `host_physical_user_visible=0`, and `prototype_devices=kernel-owned-bounce-buffer-only`. This is a diagnostics mirror for the current conservative policy, not evidence that IOMMU remapping domains or userspace DMA authority exist.

Implementation note, 2026-05-02 15:45 UTC: attached device-manager `DMAPool` records now store the current explicit bounce-buffer policy and the QEMU device-manager proofs read it back through the active device record plus the matching `DmaPoolHandle`. The logged policy scope is `device-manager-attached-dmapool-bounce-buffer-policy`, with `direct_dma=blocked`, `bounce_buffer=required`, `trusted_domain=none`, `remapping_tables=not-programmed`, `remapping_domain=not-started`, `userspace_dmapool=not-started`, `host_physical_user_visible=0`, and `policy_bound_to_manager=true`. This binds the conservative policy to current manager state; it still does not program remapping domains, expose userspace `DMAPool`, or perform real DMA mapping teardown.

Implementation note, 2026-05-11 00:00 UTC: attached device-manager `DMAPool` policy records now also carry an explicit manager-owned remapping-domain ledger staging record. The lifecycle and imported-live proofs report `remapping_domain_ledger_scope=device-manager-attached-dmapool-remapping-domain-ledger`, `static_iommu_coverage=acpi-pci-diagnostic-only`, `remapping_domain_owner=device-manager`, `remapping_domain_granularity=per-device-or-trusted-sharing-group`, `remapping_domain_ledger=manager-owned-staging-record`, `remapping_domain_ready=false`, and `iova_export=disabled-future-only`, while preserving `direct_dma=blocked`, `remapping_tables=not-programmed`, and `host_physical_user_visible=0`. This is a software ledger/readiness record only: capOS still does not program Intel VT-d/AMD-Vi/QEMU remapping tables, create a trusted direct-DMA domain, expose host physical addresses or IOVAs, or claim production hostile-hardware DMA isolation.

Implementation note, 2026-05-11 17:07 UTC: the same manager-owned remapping-domain staging record is now an explicit activation gate tied to the active `DMAPool` record and matching handle. The device manager validates that gate before current `DMAPool` policy/accounting and buffer issue paths proceed. The gate reports `domain_ownership=manager-owned-active-dmapool`, but keeps `direct_dma=blocked` because `remapping_table_programming=not-programmed`, `iova_export=disabled-future-only`, `remapping_invalidation_policy=not-installed`, `remapping_iotlb_flush_policy=not-installed`, and `remapping_stale_mapping_cleanup=not-installed`; the selected fallback remains `remapping_fallback_policy=kernel-owned-bounce-buffer-only`. The activation result is `blocked-remapping-prerequisites-missing` with `remapping_activation_gate=fail-closed`, `remapping_activation_blocker=remapping-tables-not-programmed`, and `remapping_activation_side_effect=side-effect-blocked`. This is a software policy gate and proof surface only: capOS still does not program Intel VT-d/AMD-Vi/QEMU remapping tables, create a trusted direct-DMA domain, export IOVAs or host physical

addresses, remove real IOMMU mappings, flush IOTLB state, or prove IOMMU-backed hostile stale-DMA behavior.

Implementation note, 2026-05-12 18:49 UTC: the manager-owned staging record now includes a concrete per-device remapping-domain identity for the active `DMAPool` handle: claimed-device domain identity, staged single-device sharing group, BDF-derived device id, pool slot, pool generation, and owner generation. The activation preflight treats that identity binding as a prerequisite before direct DMA could be considered, and the QEMU lifecycle/imported-live proofs emit a `dmapool remapping domain identity proof` line. The direct-DMA blocker remains unchanged: remapping tables are still `not-programmed`, IOVA export is still `disabled-future-only`, invalidation, IOTLB flush, and stale mapping cleanup are still `not-installed`, direct DMA remains `blocked`, and the fallback remains `kernel-owned-bounce-buffer-only`.

Implementation note, 2026-05-12 21:19 UTC: the same manager-owned remapping-domain ledger now carries a separate mapping-lifecycle preflight record. The record is bound to the active `DMAPool` handle and claimed-device domain identity, and the existing device-manager policy gate validates it before accepting the current bounce-buffer attach/accounting/buffer-issue paths. Its direct-DMA result remains fail-closed with explicit blockers: IOVA space, mapping install, removal before page reuse, invalidation policy, IOTLB flush policy, and stale mapping cleanup are all `not-installed`. This is still an in-repo software preflight only; it does not program remapping tables, expose IOVAs or host physical addresses, enable direct DMA, remove real IOMMU mappings, flush IOTLB state, or prove IOMMU-backed hostile stale-DMA behavior.

Implementation note, 2026-05-12 22:26 UTC: capOS now has an Intel/QEMU remapping table scaffold that can represent a DRHD identity field, PCI segment/BDF/source ID, domain ID, QEMU Intel address-width choice, disabled root/context entries, and a second-level page-table-root placeholder. PCI diagnostics can bind that scaffold to discovered DRHD/segment metadata when ACPI/PCI discovery provides it. The disabled backend registry's only accepted active state is `disabled`. The proof labels distinguish representability from programming: root-table pointer, context-entry programming, invalidation registers, fault registers, protected-memory registers, and invalidation queue remain `not-written`; remapping tables remain `not-programmed`; hardware programming remains `not-attempted`; direct DMA remains `blocked`; IOVA export remains `disabled-future-only`; and host physical addresses remain hidden from userspace. This is still not Intel VT-d, AMD-Vi, or QEMU IOMMU programming.

Implementation note, 2026-05-23 18:06 UTC: the first production `DMAPool` ledger integration for the QEMU Intel remapping path now maps the selected virtio-rng request-buffer IOVA to an active manager-owned `DMABuffer` page. Mapping install is admitted through the matching active `DmaPoolHandle` and `DmaBufferHandle` generations, stale pool and buffer generations fail closed, and wrong-owner mapping attempts are side-effect blocked. On teardown, the target second-level leaf is removed, the context-cache and IOTLB completion polls finish, and only then is the `DMABuffer` released through the production device-manager ledger. The proof keeps the IOVA internal, keeps `host_physical_user_visible=0`, keeps userspace IOVA export disabled for this slice, and leaves the no-remapping fallback policy as `kernel-owned-bounce-buffer-only`.

Implementation note, 2026-05-23 19:18 UTC: QEMU Intel remapping fault reporting now decodes VT-d `FSTS` plus `FRCD[0]` into a bounded kernel record for the faulting IOVA, reason, requester source ID, and DMA read/write type. The unmapped-IOVA, stale-handle, and stale-completion

proofs record the fault, clear it with write-1-to-clear semantics, verify the clear-after-record state, and report source/IOVA match status without exposing host physical addresses. The COM1 `devices` diagnostics path now prints an IOMMU fault summary that reserves `fault_summary=clean` for a successful clear fault-status read and labels unavailable fault-status reads as unavailable/fail-closed. Owner identity, DRHD register bases, and host physical addresses stay hidden. The optional audit route is explicitly not wired for this slice (`volatile_audit=not-routed`).

Implementation note, 2026-05-13 15:29 UTC: active manager-owned `DMAPool` remapping preflight records now consume the same retained DMAR DRHD/requester metadata when PCI coverage is complete. The active disabled Intel/QEMU scaffold records the retained DRHD identity and requester segment/BDF/source ID for the bound pool handle, but absent, malformed, capped, unsupported, or uncovered metadata still leaves the scaffold not-bound and disabled. This is metadata-only binding: capOS still does not program root/context tables, install or remove mappings, invalidate remapping caches, flush IOTLB state, export IOVAs, enable direct DMA, or expose host physical addresses.

Implementation note, 2026-05-12 23:07 UTC: PCI diagnostics now include a separate Intel/QEMU remapping MMIO-status proof for the selected DMA-capable function. When complete retained DMAR DRHD metadata covers that function and the register base is page-aligned, capOS maps only the selected remapping-register page for bounded volatile diagnostic reads of the version, capability, global-status, root-table-address, and fault-status registers. The mapped label describes the diagnostic access pattern, not page-table write protection. When the default diagnostics shape has no DRHD, metadata is capped, or the retained DRHD base is invalid, the same proof reports `mmio_window=not-mapped`, `mmio_read=not-attempted`, unavailable capability/status/fault reads, and a fail-closed reason. The labels preserve `remapping_tables=not-programmed`, `direct_dma=blocked`, `fallback_policy=kernel-owned-bounce-buffer-only`, and `hostile_hardware_isolation=not-claimed`. This is not remapping-domain activation: capOS still does not write VT-d, AMD-Vi, or QEMU remapping registers, install root/context tables or invalidation queues, export IOVAs, or claim hostile-hardware DMA isolation.

Implementation note, 2026-05-13 01:20 UTC: active manager-owned `DMAPool` records now also carry a generic disabled IOVA ledger under the remapping-domain record. The ledger binds a domain-scoped reservation identity, hidden internal range metadata, and reservation generation to the active owner and pool generation. The same device-manager accounting path validates that ledger before current bounce-buffer allocation, descriptor submission, completion accounting, buffer free/page release, and pool release checks proceed. Pure tests and QEMU proof labels reject stale reservation generations and wrong owner generations as `disabled-iova-ledger-stale` with side effects blocked. The active state remains disabled: proof output keeps the internal range hidden and reports `iova_base_user_visible=0`, `host_physical_user_visible=0`, `iova_export=disabled-future-only`, `direct_dma=blocked`, `mapping_install=not-installed`, `mapping_remove_before_page_reuse=not-installed`, `invalidation_policy=not-installed`, `iotlb_flush_policy=not-installed`, and `stale_mapping_cleanup=not-installed`. This still does not program remapping tables, export IOVAs, expose host physical addresses, install or remove real IOMMU mappings, flush IOTLB state, or claim hostile-hardware isolation.

Implementation note, 2026-05-11 18:44 UTC: the selected userspace virtio-net TX provider smoke now grants a runtime-visible `DeviceMmio` notify BAR cap named `notify_mmio`, but keeps the active DMA posture unchanged. The provider still uses manager-owned bounce buffers, `direct_dma=blocked`, and `host_physical_user_visible=false`; the notify cap is a no-write MMIO admission boundary over the selected virtio-net TX notify offset, not a direct DMA or descriptor-ring ownership transfer. The selected submit path validates descriptor authority and scrubs the bounce page before consuming the grant-derived notify policy, and it proves wrong value, wrong offset, stale handle, and stale generation block before any doorbell. This does not program IOMMU/remapping tables, export IOVA or host physical addresses, mutate the real virtio-net descriptor ring, or claim production NIC isolation.

Implementation note, 2026-05-11 19:05 UTC: the `notify_mmio` grant remains runtime-visible but is now explicitly no-direct-MMIO as well as no-write. `DeviceMmio.map` and `DeviceMmio.read32` for that provider notify cap return typed blocked results before any user VMA or register read, `DeviceMmio.info` validates the live mapping generation before accepting the provider-notify record, and `notify_mmio` detach clears the submit-path notify policy so a later selected submit reports stale-handle blocking with no doorbell write. The submit path also invalidates the cached notify policy on owner-generation transitions and stale/missing cap-release detach boundaries before accepted no-write authority can be reported.

Implementation note, 2026-05-11 19:56 UTC: the same provider smoke now grants a runtime-visible `Interrupt` cap named `tx_interrupt` for the selected virtio-net TX MSI-X route snapshot. This extends the grantable authority boundary without changing the DMA posture: the provider still uses manager-owned bounce buffers, direct DMA and IOMMU remapping remain blocked, and the interrupt cap only proves generation-checked admission for info/ack/unmask/wait/mask plus waiter cancellation and stale-after-release blocking. Later bounded follow-ups add selected TX MSI-X vector-control mask/unmask; this first grant slice did not deliver provider IRQs to userspace, acknowledge or mask/unmask hardware, ring a doorbell, or mutate real virtio-net descriptor rings.

Implementation note, 2026-05-11 20:09 UTC: provider TX waiters are now separate no-delivery waiter-table entries rather than generic delivery waiters. A pending `tx_interrupt.wait` remains pending across TX route delivery-count advancement and only completes through the explicit mask/release cancellation path. The staged provider TX grant source also tracks a live-issued cap and refuses another live `tx_interrupt` alias for the same route snapshot.

Implementation note, 2026-05-12 09:13 UTC: the selected userspace virtio-net TX provider path now performs one bounded real descriptor/avail publication on queue `1` while keeping the DMA posture conservative. The descriptor points at the manager-owned bounce buffer already governed by the `DMABuffer` record; the submit path validates live buffer identity, scrubs before publication, requires the live no-write `notify_mmio` policy, asserts that the descriptor page is ledgered to the virtio-net TX queue, and blocks wrong queue, stale notify policy, and a real stale `DMABuffer.submitDescriptor(queue=1)` attempt at the stale capability/liveness boundary before touching the real ring. The proof logs descriptor/avail/used ring physical addresses for kernel evidence, but those addresses are not returned to userspace. Because this slice does not claim real used-ring/CQ completion, the published page remains pinned in the manager in-flight record; userspace `completeDescriptor`, `freeBuffer`, post-publication remap, and cap-release drain do

not retire, remap, or release it. A follow-up rings one selected TX virtqueue notify doorbell after the same descriptor authority, submit-scrub, live `notify_mmio` policy, submit-effect write, and publication gates, while wrong-queue and stale-notify or stale-`DMABuffer` paths remain not-written. Readback-mismatch publication failures do not write the immediate doorbell and are treated as possibly-published ring state that quiesces later TX notification and keeps the manager buffer pinned rather than claiming rollback. Pre-publication bounce-page metadata remains `doorbell_write=not-written`. Any immediate used-ring or IRQ effect from that doorbell is recorded only as an out-of-scope hardware side effect. Later 2026-05-12 follow-ups advanced the selected path to a bounded used-ring completion handoff: `DMABuffer.completeDescriptor` validates the live manager-attached buffer and in-flight descriptor id, observes the real TX used ring for the stored software descriptor generation, consumes that entry through the existing descriptor tracker and DMA ledger, and only then clears the manager in-flight record. As of commit `e248d42b` (`2026-05-23 13:36 UTC`), kernel TX helpers stay quiesced after provider ownership starts while the provider path can publish the full selected TX queue-depth window of eight descriptors before the first completion; the smoke records `live_inflight_after_submits=1/2/3/4/5/6/7/8` (the ninth allocation rejected `dmapool-budget-exceeded`), blocks map/free/reuse while any buffer is in flight, and proves wrong-order descriptor `7` used-ring handling preserves the observed descriptor `0` completion for its matching generation. The provider-facing `tx_interrupt` waiter is a runtime-visible completion-event consumer for the same selected route; delivery validates the expected TX source id, source generation, route generation, owner, driver-unmasked state, and live issue id before completing each bounded event. A 2026-05-13 follow-up adds the bounded incomplete-descriptor teardown drain: when one descriptor has completed and seven remain incomplete, release retires only the incomplete descriptors' allocation-backed TX DMA ledgers and clears only the selected virtqueue descriptor/used-ring tracking needed for those releases, while CQ publication and provider IRQ delivery stay blocked and the pending waiter remains undelivered until the smoke explicitly cancels it through the existing mask/cancel path. Commit `e248d42b` (`2026-05-23 13:36 UTC`) extends that drain to the full selected TX queue-depth window and keeps the completed descriptor's buffer retained until it is explicitly freed. A later 2026-05-13 remediation binds each provider TX in-flight descriptor to the submission-time provider issue/source/route generation. If that old descriptor completes after `tx_interrupt` release/regrant, `DMABuffer.completeDescriptor` fails closed as `dmabuffer-provider-tx-stale-issue` before consuming the used ring, publishing provider CQ/IRQ state, or advancing provider acknowledgements; later cap release may still drain the descriptor as teardown-only. `tx_interrupt.wait` posting is serialized with provider release, mask, and delivery, and stale issue ids fail closed at admission and insertion. A later 2026-05-13 follow-up lets `tx_interrupt.acknowledge` account exactly one already observed selected TX dispatch token paired with one delivered provider CQ event; the smoke proves pre-event, duplicate, teardown-drain, masked-route, reset/regrant, stale-after-release, and stale issue acknowledgements fail closed before delivery-count, route-state, CQ, ack-ledger, or hardware-dispatch-ack mutation. This is still bounded selected-route evidence: provider IRQ ownership, deferred EOI, LAPIC/MSI-X acknowledgement, direct DMA, IOMMU mapping, full virtio-net ownership, production NIC/storage migration, and cloud readiness remain open. Commit `e248d42b` (`2026-05-23 13:36 UTC`) adds release-time retirement for delivered but unacknowledged bounded provider TX CQ events: the release proof now records seven pending provider completion acks retired from the ledger in

one live issue while preserving the separate stale-bound in-flight descriptor proof, with stale post-release ack revoked and no hardware ack claimed. A 2026-05-13 follow-up adds bounded selected TX MSI-X mask/unmask only: live provider `tx_interrupt.mask` and `unmask` toggle the selected TX vector-control bit plus route state, preserve generations and delivery counts, and block stale issues before side effects.

Implementation note, 2026-05-02 19:43 UTC: the bounded zero-live device-manager `DMAPool` lifecycle proof now treats its manager-attached DMA buffer record as teardown-blocking metadata. The pool detach path still checks authoritative live accounting first, then rejects zero-live detach while the proof buffer is attached as `dmapool-buffer-attached`. Before the active free path, the proof validates stale same-slot and wrong-identity `FreeBuffer` operations through `capos-lib::device_authority::validate_dma_buffer_operation`. The wrong identity cases cover wrong owner generation, wrong pool slot, wrong pool generation, and wrong buffer slot; each records `dmabuffer-stale-handle`, the exact validator reason (`stale-owner-generation`, `wrong-pool`, `stale-pool-generation`, or `wrong-slot`), `side-effect-blocked`, and a preserved manager-owned buffer record. The stale same-slot case continues to record `stale-slot-generation` and `buffer_stale_free_preserved=true`, then the proof observes that pool detach still fails as `dmapool-buffer-attached`. The proof clears the gate only after validating a proof-scoped active `FreeBuffer` operation, scrubbing and freeing the kernel-owned proof frame, and detaching the manager-owned buffer record. This remains lifecycle evidence only: no userspace `DMAPool` or DMA-buffer authority is exposed, no physical address or IOVA is exposed, and S.11.2 hostile stale-DMA smokes remain open.

Implementation note, 2026-05-03 02:31 UTC: the same zero-live device-manager `DMAPool` lifecycle proof now validates manager-record `CompleteDescriptor` authority for the attached `DmaBufferHandle`. Active completion validation records `buffer_active_complete_result=ok`; freed-buffer, reused-slot generation, and stale-after-revoke completion attempts fail closed as `dmabuffer-stale-handle` with exact pure-validator reasons and `side-effect-blocked`. This is manager-record validation evidence only: it does not complete a real descriptor, publish a completion queue entry, grant a userspace `DMABuffer`, run real DMA, or clean up or reuse production userspace DMA pages.

Implementation note, 2026-05-14 14:05 UTC (DDF IOMMU remapping Slice A1): the first slice that programs real Intel VT-d remapping state under QEMU has landed. The `## First QEMU Intel Remapping Smoke Acceptance Gate` above defines the full bounded gate; that gate is being delivered as a sequenced A1/A2/B/C split (the slice was correctly scoped as bigger than one reviewable unit). This note records Slice A1.

Pinned QEMU shape: `qemu-system-x86_64` 8.2.2, `-machine q35`, `-device intel-iommu,aw-bits=39` (3-level second-level page tables, 39-bit IOVA space). The kernel `iommu` module (`kernel/src/iommu.rs`, `cfg(qemu)`-only) selects one DMA-capable function that is **not** the live virtio-net bounce-buffer path (virtio-rng under the default smoke shape), allocates a root table, one context table, a 3-level second-level page table, and one mapped DMA page; encodes and writes one root entry, one context entry (binding the requester source id to a domain id, `aw-bits=39`, and the second-level root), and the second-level table-pointer / leaf entries through the HHDM; writes the Root Table Address Register; and runs the global-command/global-status `SRTP`-then-`TE` handshake, polling the status register for each step. The capability-register extended-capability

IRO field (IOTLB register offset) is decoded and reported for Slice B's benefit. MMIO ordering invariants are enforced with SeqCst fences: between the last in-memory table-entry write and the RTAR write, between the RTAR write and GCMD.SRTP, and between the latched root pointer and GCMD.TE.

Slice A1 proves translation with **kernel-side structural evidence only**, which the gate's IOVA-export-discipline clause explicitly permits. Hardware confirms translation-enabled (GSTS.TES + GSTS.RTPS polled set), the written entry words are read-back-verified through the HHDM, the pure capos_lib::device_authority validator accepts the layout, and the unmapped IOVA's 3-level walk structurally terminates at a non-present entry. The proof labels are scrupulously honest: mapped_iova_translated=structural (not hardware-dma), unmapped_iova_fault=structural-not-present (not observed), proof_evidence=kernel-side-structural. A real hardware-DMA translation and a real fault-status fault require driving a device virtqueue through the IOMMU; that is **deferred to follow-on task A2** (a virtio-rng virtqueue driver as the DMA proof vehicle). Invalidation + IOTLB flush with completion polling (Slice B) and the IOMMU-backed stale-handle / stale-completion hostile smokes (Slice C) are also follow-on slices; at A1 their proof lines emit proof_result=deferred-next-slice. A2, B, and C have since landed — see the implementation notes below.

The table pages are recorded in a bounded ledger modeled on the device-manager DMAPool page-accounting discipline (allocate-record, scrub-before-free on the fail-closed path); mapping removal with IOTLB-flush-ordered scrub/free is Slice B. IOVA export stays disabled (iova_export=disabled-this-slice), no host physical address is user-visible (host_physical_user_visible=0), and no hostile-hardware isolation is claimed (hostile_hardware_isolation=not-claimed). The kernel-owned bounce-buffer fallback is unchanged for QEMU shapes without usable intel-iommu hardware and is emitted with the explicit fallback_policy=kernel-owned-bounce-buffer-only / remapping_tables=not-programmed labels. The new make run-iommu-remapping gate is asserted by tools/qemu-iommu-remapping-smoke.sh; make run-net, make run-dmapool-grant, and make run-diagnostics continue to prove the fallback path unchanged.

Implementation note, 2026-05-14 15:19 UTC (DDF IOMMU remapping Slice A2): the device-DMA proof vehicle has landed, upgrading the A1 structural proof to a real hardware-DMA proof and closing the literal hardware-DMA text of gate part

1. After the VT-d tables are programmed and GCMD.TE is set, a minimal virtio-rng virtqueue driver — split into a mapped-DMA phase (crate::virtio::prove_iommu_rng_mapped_dma) and an unmapped-DMA phase (crate::virtio::prove_iommu_rng_unmapped_dma) — drives the device QEMU exposes on the intel-iommu shape. The second-level table now installs four leaf entries inside one shared L1 page: the request buffer plus the three virtqueue ring pages (descriptor table, available ring, used ring). The driver programs the device's QUEUE_DESC / QUEUE_DRIVER / QUEUE_DEVICE registers and the request descriptor's addr field with the **programmed IOVAs**, never the host-physical page addresses. Because VT-d translation is global per requester once GCMD.TE is set, every DMA the device issues — every ring access and the entropy write — must walk the second-level table. A used-ring completion plus a non-zero buffer reading therefore proves a real hardware DMA reached the kernel page through the

programmed IOVA translation: mapped_iova_translated=hardware-dma, proof_evidence=virtio-rng-hardware-dma.

The driver then publishes a second descriptor whose addr is the deliberately-unmapped IOVA and kicks the device. The device's DMA to that IOVA raises a real VT-d translation fault; the kernel reads it back out of the Fault Status Register (FSTS.PPF), the first Fault Recording Register's fault bit (FRCD[0].F at the decoded CAP.FRO offset), and the faulting page address recorded in FRCD[0] — which must equal the unmapped IOVA the device was pointed at — and reports unmapped_iova_fault=observed with the fault_recording_reason code. The fault gate is purely the VT-d register surface: whether QEMU's virtio-rng still pushes the faulting descriptor onto the used ring afterward (it does, with the entropy write dropped) is QEMU device behavior, reported as the unmapped_descriptor_uncompleted diagnostic field but deliberately not a gate condition. The fault registers are cleared (write-1-to-clear) before the device DMA and again after the observed-fault read so no stale fault is mistaken for the proof and no fault is left for a later VT-d consumer. The MMIO discipline reuses A1′s NO_CACHE mapping and the descriptor/available-ring writes are SeqCst-fenced before the notify doorbell. The two phases are deliberately split so the kernel reads the fault registers strictly between them — the unmapped-IOVA descriptor is never in flight while the mapped-DMA result is judged. The virtio-rng device negotiates VIRTIO_F_ACCESS_PLATFORM, which is what makes QEMU route its DMA through the platform IOMMU rather than treating the ring registers as host-physical addresses; the run-iommu-remapping make target therefore creates the virtio-rng device with iommu_platform=on (a target-scoped override of the shared QEMU_SECOND_DEVICE, which no other run target needs because none of them drives virtio-rng DMA). The IOMMU-backed hostile smokes (Slice C) were a follow-on at A2 (proof_result=deferred-next-slice) and have since landed — see the Slice C implementation note below; IOVA export stays disabled and no host-physical address is user-visible.

Implementation note, 2026-05-14 17:19 UTC (DDF IOMMU remapping Slice B): the invalidation + IOTLB flush + invalidation-ordered scrub/free has landed, closing gate part 2. After the A2 hardware-DMA proof, kernel/src/iommu.rs runs a revocation cycle (run_invalidation_revocation_cycle) that models the device-manager DeviceOwnerState revocation FSM at the QueuesQuiesced -> Resetting -> DmaMappingsRemoved -> Dead steps. The cycle removes the four second-level leaf entries the A2 layout installed (request buffer + the three virtqueue ring pages), SeqCst-fences so the in-memory removal is visible to the IOMMU before the flush, then issues two **register-based** invalidations: a context-cache invalidation through the Context Command register (CCMD_REG at offset 0x28, CCMD.ICC set with CCMD.CIRG global granularity, polling CCMD.ICC clear for completion) and a domain-selective IOTLB invalidation through the IOTLB register at the A1-decoded CAP.IRO offset + 8 (IOTLB.IVT set with IOTLB.IIRG domain-selective granularity and the domain id in IOTLB.DID, polling IOTLB.IVT clear and reading IOTLB.IAIG back non-zero to confirm the request was serviced). Both completion polls are bounded by the same VTD_STATUS_POLL_BUDGET the A1 status handshakes use.

The hard ordering invariant — the whole point of the slice — is that the eight VT-d ledger-owned table/ring/used pages and the separate production DMAPool-owned request-buffer page are scrubbed and returned to their ledgers **strictly after** both completion polls return. A SeqCst fence sits between the completion reads and the scrub/free so the ordering is explicit in program

order. A poll that exhausts its bounded budget **fails closed**: `invalidation_completed` is `false`, the pages are deliberately **not** freed (a page reused while hardware may still hold a cached translation through it is a stale-DMA hole), the ledgers keep them accounted rather than leaked-and-forgotten, and the proof line reports `proof_result=fail-closed`. Slice B uses **register-based invalidation only**: no `GCMD.QIE` queued-invalidation bit is set, so the A1 single-bit-`GCMD` discipline (correct only by minimalism — no other persistent `GCMD` bit set) still holds and the `GCMD`-reconstruct boundary is not crossed. The production DMAPool programming-abort path follows the same rule: if VT-d programming fails before root/translation state can expose the DMAPool page to hardware, the prepared DMABuffer/DMAPool records are detached; if the mapping may already be hardware-visible, the partial VT-d ledger is carried through the same leaf-removal and invalidation teardown before any VT-d table/ring page or production DMAPool page can be reused. The `make run-iommu-remapping` smoke and `tools/qemu-iommu-remapping-smoke.sh` now assert the `invalidation proof` line as `proof_result=ok` with `mapping_removed=true context_cache_invalidated=true iotlb_flushed=true iotlb_actual_granularity_nonzero=true invalidation_completed=true page_reuse_ordered_after_invalidation=true table_pages_live_after=0 invalidation_interface=register-based-ccmd+iotlb`, and forbid both a regression to the Slice-A deferred label and any `invalidation_interface=queued` value. The IOMMU-backed stale-handle / stale-completion hostile smokes (Slice C) were the deferred follow-on; they have since landed (see the note below), and as part of that work the single-phase `run_invalidation_revocation_cycle` was refactored into a two-phase `run_target_revocation_phase` + `complete_revocation_teardown` so the hostile re-drive can sit between the phases — the Slice B contract (every page freed strictly after its mappings are invalidated) is unchanged, and the combined `invalidation proof` line still asserts `proof_result=ok` for the complete teardown. IOVA export stays disabled and no host-physical address is user-visible.

Implementation note, 2026-05-14 19:13 UTC (DDF IOMMU remapping Slice C): the IOMMU-backed hostile stale-DMA smokes have landed, closing gate part 5 and the parent IOMMU remapping task. Closing the slice required refactoring the Slice B revocation into **two phases** so the hostile re-drive can run against a *partially* revoked remapping — the original single-phase cycle freed every page (including the virtio-rng descriptor table and available ring) before any hostile re-drive could observe a fault, so the re-driven device read an all-zero descriptor and issued no DMA at all. The `kernel/src/iommu.rs` ledger now classes each page by revocation-phase role: `Target` (request buffer + used ring — what the device's DMA *lands on*), `RingInfra` (descriptor table + available ring — what the device *reads* to issue a DMA), and `Table` (root/context/second-level tables). `run_target_revocation_phase` removes the `Target` second-level leaves, invalidates the context-cache + IOTLB, and frees the `Target` pages — while deliberately keeping the `RingInfra` + `Table` pages mapped and live. `run_hostile_stale_dma_cycle` then re-drives the **same still-live old-generation** virtio-rng device through the new `crate::virtio::prove_iommu_rng_stale_dma` (each re-drive uses a fresh available-ring index past the A2 phases so the device sees a genuinely new descriptor): because the ring-infra pages are still mapped, the device reads a *valid* descriptor whose `addr` is a *revoked* target IOVA, so the DMA faults in the IOMMU (`FSTS.PPF` + `FRCD[0].F`, recorded faulting address is the stale IOVA) instead of reaching memory. A stale mapping-install attempt is refused (`attempt_stale_mapping_install` — the `RevokedRemapping` token is a dead-domain receipt with

no live table handle, not install authority), and the freed `Target` page reads back as the scrubbed zeros. A second re-drive at the revoked used-ring IOVA faults too, publishes no device-written used-ring CQ entry into the freed page, exposes no memory to a would-be new owner, and makes no freed page eligible for reuse. `complete_revocation_teardown` then finishes the Slice B teardown by revoking + freeing the `RingInfra` + `Table` pages; the combined gate-part-2 `invalidation proof` line (`invalidation_phases=target-then-ringinfra`) still asserts `proof_result=ok` for the full two-phase teardown, with the same hard ordering invariant in each phase (pages freed strictly after that phase's invalidation completion polls return; a poll that exhausts its budget fails closed and the phase's pages are not freed). The load-bearing observation is the **revoked translation state** — not device cooperation, not a software ledger drop — blocking the stale DMA, confirmed by the VT-d fault registers plus a freed-page-stays-scrubbed read-back through the HHDM. Existing bounce-buffer stale-DMA evidence (the `device-dma` S.11.2.7/S.11.2.8 proofs) is preserved unchanged; the IOMMU-backed hostile smokes are strictly additive. The `make run-iommu-remapping` smoke and `tools/qemu-iommu-remapping-smoke.sh` now assert both `hostile` proof lines as `proof_result=ok` and forbid regression to the deferred, `not-reached`, or `fail-closed` labels. IOVA export stays disabled (`iova_export=disabled-this-slice`), no host-physical address is user-visible (`host_physical_user_visible=0`), and no hostile-hardware isolation is claimed (`hostile_hardware_isolation=not-claimed`).

Implementation note, 2026-05-23 (domain-scoped IOVA export): the selected QEMU Intel production `DMAPool` path now exposes the mapped request-buffer IOVA through `DMABuffer.info` only while the matching active `DmaBufferHandle` generation is live. The schema fields are `deviceIova`, `deviceIovaScope=domain-scoped-iova`, `deviceIovaMeaning=meaningless-outside-domain`, and `iovaExport=domain-scoped-only`; the production remapping proof asserts that `deviceIova=0x200000` matches the installed second-level mapping. After the buffer is freed and the pool is released, an export attempt on the same handle fails closed with `side-effect-blocked`. The bounce-buffer grant path still reports `deviceIova=0`, `deviceIovaScope=none`, `deviceIovaMeaning=not-exported`, and `iovaExport=disabled-future-only`.

Implementation note, 2026-05-23 21:34 UTC (production-path hostile smokes): the selected QEMU Intel path now emits and asserts `iommu-remapping: production dmapool hostile proof` over the active manager-owned `DMAPool` / `DMABuffer` ledger. The proof ties the raw VT-d stale-handle and stale-completion faults to the production mapped IOVA, synthetic stale pool/buffer generation mismatch candidates, post-teardown stale-handle export failure, and per-device cross-domain boundary. It covers stale IOVA after revoke/reset, descriptor abuse, revoke/reset race ordering, stale completion after reset, teardown-under-DMA ordering, and cross-domain stale-handle attempts; no second-level entry is installed for stale authority, no CQ entry is published, no new-owner memory is exposed, and page reuse stays ordered after invalidation completion. It does not claim a process-exit trigger for the IOMMU path; the existing `make run-net` bounce-buffer evidence remains the exit-under-DMA source. The same smoke asserts the `complete_iommu_dmapool_mapping_teardown` prerequisite-false return and the `hold_iommu_dmapool_mapping_ledger_after_abort` path as fail-closed branch evidence. Existing bounce-buffer S.11.2 evidence from `make run-net` and `make run-dmapool-grant` is preserved unchanged.

Implementation note, 2026-05-26 05:55 UTC (direct-DMA posture transition for the selected QEMU Intel path): the closeout slices above landed the full mechanism — real hardware DMA over a manager-owned `DMAPool` `DMABuffer` page mapped through the per-device IOMMU domain, domain-scoped IOVA export, per-device domains, and the production hostile matrix — but deliberately deferred the headline `direct_dma=enabled` claim behind `iova_export=disabled-this-slice`. The selected QEMU Intel path now emits `iommu-remapping: direct-dma posture real_dma=attempted direct_dma=enabled remapping_tables=programmed trusted_domain=<domain-id> descriptors_reference=domain-scoped-iova mapped_page_source=manager-owned-dmabuffer mapping_installed_before_doorbell=true invalidated_before_page_reuse=true bounce_buffer=not-required exported_device_addresses=iova-only host_physical_user_visible=0 hostile_hardware_isolation=not-claimed proof_result=ok`. Every field is computed from the real proof facts, not asserted as a constant: `remapping_tables=programmed` requires the root/context/second-level entries written plus the SRTP/TES handshakes; `real_dma=attempted` requires the virtio-rng device's mapped DMA to have completed through the programmed IOVA (`hardware_dma_translation_proven`); `direct_dma=enabled` additionally requires the manager-owned `DMAPool` mapping to have been installed before the device doorbell; and `invalidated_before_page_reuse` folds in the two-phase revocation's `page_reuse_ordered_after_invalidation` and `invalidation_completed` results. This is bounded QEMU-emulator evidence, so `hostile_hardware_isolation` stays `not-claimed` (real hostile-hardware isolation needs real hardware, not QEMU). The no-IOMMU `run-net` / `run-dmapool-grant` bounce-buffer fallback is untouched: it keeps `direct_dma=blocked` with no IOVA export, and `make run-iommu-remapping` now forbids this path from regressing to the bounce-buffer fallback proof or to a blocked/not-attempted posture. The contract table in "Downstream-Contract Scaffolding" (direct-remapping domain: `direct_dma=enabled`, `remapping_tables=programmed`, `exported_device_addresses=iova-only`) is now backed by an emitted, asserted posture on the selected path.

### Authority Model

Device authority is split into three independent capabilities:

- `DMAPool`: authority to allocate, expose, and revoke device-visible memory within a kernel-owned physical range or IOMMU domain.
- `DeviceMmio`: authority to map and access one device's register windows.
- `Interrupt`: authority to wait for and acknowledge one interrupt source.

Holding one of these capabilities never implies the others. A driver needs all three for a normal device, but the kernel and init can grant, revoke, and audit them separately.

### Production Handle Epoch Invariants

All three object families use opaque handles whose identity is checked against kernel-owned records before every operation. A raw object id is never enough to authorize DMA, MMIO, interrupt waits, acknowledgements, descriptor submission, or teardown. A handle is accepted only when all of these facts match in the same ownership transaction:

- the object id resolves to a live record of the expected type;
- the handle's device owner generation matches the current device-manager owner record;

- the handle's pool, mapping, slot, source, or route generation matches the current reusable subrecord;
- the record state permits the requested operation.

The exact ABI shape may change when the capability surface is implemented, but production handles must carry the equivalent identity:

```
struct DmaPoolHandle {
    device_id: u32,
    owner_generation: u64,
    pool_id: u32,
    pool_generation: u64,
}

struct DmaBufferHandle {
    device_id: u32,
    owner_generation: u64,
    pool_id: u32,
    pool_generation: u64,
    slot: u32,
    slot_generation: u64,
}

struct DeviceMmioHandle {
    device_id: u32,
    owner_generation: u64,
    bar: u8,
    mapping_id: u32,
    mapping_generation: u64,
}

struct InterruptHandle {
    device_id: u32,
    owner_generation: u64,
    source_id: u32,
    source_generation: u64,
    route_generation: u64,
}
```

Object identity fields have distinct jobs:

- `DMAPool` handles name the claimed device, the device owner generation, and the pool record generation. Buffer handles issued by the pool repeat the device-owner and pool identity and additionally name a buffer slot and slot generation. The pool identity prevents a handle from crossing devices or owners; the slot identity prevents a freed or reused buffer slot from accepting an old descriptor, free, or completion.

- `DeviceMmio` handles name the claimed device, owner generation, BAR or subrange mapping record, and mapping generation. The physical range, cache attributes, and access policy remain in the kernel record and are not user-editable handle fields.
- `Interrupt` handles name the claimed device, owner generation, source record, source generation, and route generation. Waiter records may carry their own waiter generation internally, but they must be invalidated whenever the source or route generation changes.

Owner generations and subrecord generations are intentionally separate. The device owner generation belongs to the device-manager ownership record and invalidates every `DMAPool`, `DeviceMmio`, and `Interrupt` handle for the old owner when ownership is revoked, transferred, reset, or reassigned. Pool, buffer-slot, MMIO-mapping, interrupt-source, and route generations belong to records that may be reused below the device owner. They prevent stale buffer, mapping, route, waiter, and completion handles from matching a newly allocated subrecord even when the device id or pool id is reused.

Every epoch is non-wrapping for authority purposes. Implementations must use an epoch width that cannot wrap during the object's lifetime, or permanently retire the exhausted device, pool, slot, mapping, source, or route record. Epoch exhaustion is a closed allocation or reassignment failure; it must never wrap back to a value that could match an old handle.

Generation mismatch, wrong object type, wrong device owner, freed slot, detached source, revoked mapping, and wrong device-owner state are hard closed results. The failed operation must not program a descriptor, ring a doorbell, perform an MMIO write, unmask or acknowledge an interrupt, wake a waiter, publish a CQE, decrement completion accounting, free a page for reuse, or mutate the device ledger except for bounded failure accounting or audit metadata.

Transfer, revoke, reset, and reassignment are ordered around those epoch checks:

- **Transfer:** The old owner leaves `Active`, the owner generation advances, and old handles become invalid before a new owner receives handles. A transfer may preserve hardware state only after old interrupt notifications, MMIO write authority, and DMA submissions are either quiesced or represented by old-generation ledger entries that the new owner cannot consume.
- **Revoke:** The device manager invalidates user-visible handles first, then follows the revocation order below: MMIO write authority removed, interrupts masked or detached, queues quiesced or reset, mappings removed, and pages scrubbed before release.
- **Reset:** Reset or disable advances the owner generation and any affected source, route, pool, mapping, and buffer-slot generations before new handles can be issued. If old DMA writes cannot be proven stopped, buffer slots stay unavailable until reset completion and mapping invalidation prove reuse is safe.
- **Reassignment:** Interrupt sources, MMIO mappings, and DMA pool records are detached or unmapped, their subrecord generations advance, pending waiters or completions are drained or marked stale, and only then can a new owner receive authority for the reused source, mapping, or slot.

Handle reuse rules:

- stale handles fail closed;
- freed-handle reuse fails closed;

- reallocated slots must not restore authority to old handles;
- old interrupt waiters must not observe or acknowledge a new owner's interrupt source;
- old DMA handles must not reference a newly allocated buffer in the same slot.

Production proof obligations are split between host tests and QEMU smokes. Host tests must cover the pure validator and state-machine cases: stale owner generation, stale pool or mapping generation, stale buffer slot, stale interrupt source or route, wrong owner, wrong device, wrong object type, freed object, wrong state, epoch exhaustion/retirement, and no side effects on failure. QEMU smokes must prove the hardware-facing ordering: stale DMA handles after free/reuse cannot submit descriptors, stale DMA completions after revoke/reset cannot publish CQEs or mutate reused buffers, stale MMIO handles cannot ring doorbells after revoke, stale interrupt waiters or acknowledgements cannot wake or affect a new owner, and process-exit or driver-crash teardown reaches a zero-live ledger before pages are reused. These production handles and proofs remain open; the current QEMU scratch proofs are prerequisite evidence for this contract, not completion of it.

Implementation note, 2026-05-02 13:18 UTC: `capos-lib::device_authority` now implements the bounded pure host-testable validator prerequisite for these handle epoch invariants. The module models the documented handle and record identity fields for `DMAPool`, DMA buffer, `DeviceMmio`, and `Interrupt`, separates device-owner generations from pool, slot, mapping, source, and route generations, returns explicit fail-closed error labels, blocks the relevant side-effect class on validation failure, and refuses epoch wrap or retired epoch reuse. This does not expose production userspace handles, wire kernel device paths, attach budget/OOM policy to real handle creation, or complete the QEMU stale-handle or S.11.2 hostile-smoke gates.

Implementation note, 2026-05-03: the pure host-test operation matrix now enumerates every current validator operation variant: `DMAPool::{AllocateBuffer,IssueBufferHandle}`, `DMABuffer::{SubmitDescriptor,CompleteDescriptor,FreeBuffer}`, `DeviceMmio::{Map,Read,Write,RingDoorbell,Unmap}`, and `Interrupt::{Wait,Acknowledge,Mask,Unmask}`. Each row asserts active acceptance plus stale owner/subrecord, freed, revoked, and retired failures with the exact blocked side-effect class for that operation. This remains ABI-independent host-test evidence only; it does not create production userspace handles or replace the QEMU stale-handle and S.11.2 hostile-smoke gates.

Implementation note, 2026-05-02 13:43 UTC: the current kernel device-manager `DMAPool` lifecycle and imported-live accounting proofs now adapt their BDF, owner generation, pool slot, and pool generation into `capos-lib::device_authority` records. The QEMU proof records active validator success, stale-after-revoke failure as `dmapool-stale-handle`, the validator reason `stale-owner-generation`, and `side-effect-blocked`. This is still a bounded kernel-proof adapter, not production userspace handle exposure, `DeviceMmio`/`Interrupt` handle wiring, production page cleanup, or S.11.2 hostile smoke completion.

Implementation note, 2026-05-02 17:04 UTC: the current kernel device-manager also has a bounded manager-owned `DeviceMmio` record proof adapter. The record carries BAR, mapping id, mapping generation, and owner generation fields, and the QEMU `virtio-rng` device-manager path validates a `RingDoorbell` operation through `capos-lib::device_authority`. After revoke begins, the old handle fails through the pure validator as `stale-owner-generation`, records `devicemmio-stale-handle` plus `side-effect-blocked`, and no doorbell write is attempted. The

lifecycle proof blocks RevokingHandles -> MmioRevoked while the record is attached, then allows the transition after bounded detach. This does not expose production userspace DeviceMmio authority, program real BAR mappings, create mapping objects, or complete hostile stale-MMIO smokes.

Implementation note, 2026-05-02 17:29 UTC: the bounded DeviceMmio adapter now derives the proof mapping from the first decoded PCI memory BAR on the tested PciDevice through the shared BAR-region validator. The attached manager-owned record carries that BDF/BAR/base/length metadata and validates that it is the same BDF, a memory BAR, nonzero length, and the same BAR named by the handle before constructing the pure capos-lib::device_authority record. The QEMU smoke asserts region_source=pci-decoded-memory-bar, region_bound_to_manager=true, bar_present=true, bar_memory=true, bar_base, and bar_length. This is still prerequisite evidence only: it does not create userspace DeviceMmio handles, program real MMIO mappings, enforce cache attributes or write policy, or write a real doorbell.

Implementation note, 2026-05-02 17:54 UTC: the same bounded DeviceMmio adapter now records fail-closed malformed-region evidence before the positive attach. Wrong-BDF metadata, wrong BAR/handle mismatch, and zero-length region metadata all report devicemmio-region-invalid, with region_invalid_mapping=not-created and region_negative_side_effect=side-effect-blocked; the proof still records real_mmio_mapping=not-programmed and real_doorbell=not-written. This is bounded manager-proof evidence only. It does not create userspace DeviceMmio handles, map real BAR pages, enforce cache attributes or write policy, complete hostile stale-MMIO smokes, or perform a real doorbell write.

Implementation note, 2026-05-02 20:14 UTC: the bounded DeviceMmio adapter now stores future mapping policy metadata on the attached manager-owned record and reads it back through the active record plus matching DeviceMmioHandle. The proof line records policy_scope=manager-attached-devicemmio-cache-write-policy, cache_policy=device-uncacheable, page_table_protection=capability-scoped-device-nx, write_policy=claimed-registers-and-doorbells-only, executable=blocked, userspace_devicemmio=not-started, host_physical_user_visible=0, policy_bound_to_manager=true, and policy_result=ok. A tampered cache/write policy record fails closed with policy_tamper_result=fail-closed, policy_tamper_mapping=not-created, and policy_tamper_side_effect=side-effect-blocked. This is still metadata proof only: no PAT/MTRR or page-table programming is performed, no userspace DeviceMmio handle is created, no real BAR mapping object exists, and no doorbell is written.

Implementation note, 2026-05-02 20:45 UTC: while the same bounded manager-owned DeviceMmio record is still active, the proof now validates hostile RingDoorbell handles through a proof-scoped adapter that uses the already-attached record rather than manager lookup short-circuits. Wrong owner generation, wrong mapping generation, wrong mapping id, wrong BAR, and wrong BDF/device fail closed with exact pure-validator reasons stale-owner-generation, stale-mapping-generation, wrong-mapping, wrong-bar, and wrong-device. Each records side-effect-blocked; the proof also records that the attached manager record is preserved, no fake mapping is created, and no doorbell is written. This remains bounded proof evidence only:

production userspace handles, real MMIO mappings, real cache/write-policy enforcement, and hostile stale-MMIO smokes remain open.

Implementation note, 2026-05-03 00:36 UTC: the schema and kernel now include a result-only `DeviceMmio.info` skeleton that can wrap a manager-issued `DeviceMmioHandle`. The object validates the live device-manager record through `validate_devicemmio_record()` before returning status labels such as `userspaceDeviceMmio=manager-issued-skeleton`, `managerRecord=validated-active`, `realMmioMapping=not-programmed`, `realDoorbell=not-written`, `hostPhysicalUserVisible=false`, `directMmio=blocked`, `registerRead=blocked`, `registerWrite=blocked`, and `bootstrapGrant=blocked`. The QEMU device-manager lifecycle proof constructs that cap object while the attached record is active, records `devicemmio_cap_info_result=ok`, exercises the serialized `CapObject::call(0, &[])` path and decodes the returned `DeviceMmio.info` Cap'n Proto result as `devicemmio_cap_serialized_call_result=ok`, then verifies the same cap fails closed after revoke begins as `devicemmio_cap_stale_after_revoke_result=devicemmio-stale-handle`; the same stale object also fails the serialized method-0 path as `devicemmio_cap_serialized_stale_after_revoke_result=invoke-failed`. A later manifest-grant smoke explicitly releases the granted `DeviceMmio` cap through `CAP_OP_RELEASE` and proves a subsequent typed `DeviceMmio.info` call fails closed from userspace. A focused grant-cycle smoke now repeats that grant, release, and stale-info proof twice in sequence and asserts the second manager-grant-source acquire receives a fresh mapping generation after the first release; the same smoke also decodes both acquire/release cycles through the typed volatile `HardwareAuditLog.snapshot` surface. The focused hardware-audit interrupt-waiter smoke also decodes recent boot-time `DmaBuffer`, `DmaPool`, and `Interrupt` `driver-crash` / `reset-disable` lifecycle records from the current volatile 16-record snapshot window. The same smoke now uses the `startSequence` cursor to decode older retained `DeviceMmio` lifecycle rows that the default latest 16-record tail has skipped. A `2026-05-10 15:33 UTC` manifest-grant follow-up turns `DeviceMmio.map` from admission-only into a read-only userspace VMA over the boot-preseeded BAR page already used by brokered `read32`. The typed smoke validates the active `DeviceMmioOperation::Map` authority check, rejects writable, executable, unknown, zero-size, unaligned, out-of-BAR, and overflow requests with typed no-side-effect results, reads the same QEMU BAR value through the returned userspace address and `DeviceMmio.read32`, rejects a duplicate active map, explicitly calls `DeviceMmio.unmap`, proves a second unmap is a typed no-op, remaps, and proves stale `unmap` fails closed after cap release. Release/drop/driver-crash/reset-disable cleanup revokes any borrowed user VMA before detaching the manager record. This is read-only BAR VMA evidence only: it does not add writable MMIO, volatile register writes, doorbells, host physical/IOVA exposure, post-userspace kernel MMIO mappings, IOMMU programming, durable/signed audit persistence, concurrent sharing semantics, or a production provider-driver consumer. A `2026-05-10 20:06 UTC` follow-up promotes `DeviceMmio.write32` to one bounded brokered volatile dword write through that same boot-preseeded kernel MMIO cache after active manager-attached handle, owner/state, policy/region, pure Write authority, dword alignment, decoded-BAR range validation, and a single provider-scoped claim derived from PCI MSI-X metadata, including BDF, BAR, BAR base, offset, and value. The focused proof writes the claimed virtio-rng MSI-X entry-0 vector-control mask dword, reads it back through both brokered `read32` and the read-only userspace VMA, then proves an unclaimed message-address dword write leaves the original value unchanged. Invalid range and unclaimed calls

remain typed no-write results, while stale or released handles fail closed before any write and do not return a `write32` result payload. This does not add writable userspace BAR mappings, arbitrary register writes, doorbells, host physical/IOVA exposure, post-userspace arbitrary remaps, IOMMU programming, or a production provider-driver consumer.

A `2026-05-26 07:32 UTC` follow-up (`ddf-userspace-writable-devicemmio-interrupt`) proves the cross-authority non-implication required by the gate ("holding one authority must not imply either of the others"). Each grant smoke's granted `CapSet` is its sole authority source: the `DeviceMmio` smoke holds only `console` + `device_mmio` and the `Interrupt` smoke only `console` + `interrupt`. The smokes assert that the other DDF grants (`interrupt`/`device_mmio` and `dmapool`) are absent from the `CapSet`, and that the held cap cannot be reinterpreted as another interface because the kernel-delivered interface id is fixed at grant time (`negative-authority ... result=ok` lines, with the kernel "spawned … 2 caps" structural counterpart). This is a non-implication proof over the already-landed authorities; it adds no new kernel MMIO/IRQ/DMA surface. Real userspace `wait`/`acknowledge` over the live route with deferred LAPIC EOI is proven separately by the provider `tx_interrupt`/`rx_interrupt` consumer (`make run-ddf-provider-consumer`).

#### `DMAPool` Invariants

`DMAPool` is the only future userspace-facing authority that may cause a device-visible DMA address to exist.

- **Authority:** A holder may allocate buffers only from the pool object it was granted. It may not request arbitrary physical frames, import caller virtual memory by address, or derive another pool.
- **Handle identity:** A pool operation checks the claimed device id, owner generation, pool id, and pool generation before changing pool state. Buffer operations additionally check the buffer slot and slot generation before descriptor validation, completion accounting, free, scrub, or reuse.
- **Physical range:** Every exported device address must resolve to pages owned by the pool. The kernel records the allowed host-physical page set and validates every descriptor mapping against that set before a device can use it. If an IOMMU domain backs the pool, the exported address is an IOVA, not raw host physical memory.
- **Ownership:** Each DMA buffer has one pool owner, one device-domain owner, and explicit CPU mappings. Sharing a buffer with another process requires a later typed memory-object transfer; copying packet data is the default until that object exists.
- **No raw grants:** Userspace never receives an unrestricted host-physical address. A driver may receive an opaque DMA handle or an IOVA meaningful only to its `DMAPool`/device domain. It cannot turn that value into access to unrelated RAM.
- **Residency:** DMA pages are committed before exposure to the device, resident for the entire device-visible lifetime, unswappable, and scrubbed before reuse by another owner.
- **Bounds:** Buffer length, alignment, segment count, and queue depth are bounded by the pool. Descriptor chains that point outside an allocated buffer, wrap arithmetic, exceed device limits, or reference freed buffers fail closed before doorbell writes.

- **Revocation:** Revoking the pool first quiesces the device path using it, prevents new descriptors, waits for or cancels in-flight descriptors, then removes IOMMU mappings or invalidates bounce-buffer handles before freeing pages.
- **Reset:** If in-flight DMA cannot be proven stopped, revocation escalates to device reset through the owning device object before pages are reused.
- **Residual state:** Pages returned from a pool are zeroed or otherwise scrubbed before reuse by a different owner. Receive buffers are treated as device-written untrusted input until validated by the driver or stack.

Device-visible memory authority is not ordinary `MemoryObject` authority. `FrameAllocator` and `MemoryObject` must not become raw physical-address escape hatches. A future shared-buffer transfer may share CPU-visible packet bytes after validation, but it does not by itself grant IOVA creation, descriptor programming, or device write authority.

For the in-kernel QEMU smoke, the kernel is the only `DMAPool` holder. The same invariants apply internally even though no userspace capability object is exposed yet.

Implementation note, 2026-04-24: the initial virtio-net transport uses kernel-owned frame-allocator pages for RX/TX split-virtqueue descriptor, available, and used rings plus the one-shot TX descriptor proof buffer, ARP TX buffer, ICMP TX buffer, smoltcp adapter/TCP TX buffers, and posted RX packet buffers. The smoltcp adapter copies completed RX frame bytes out of those device-written pages before handing them to the stack. Those pages are programmed into the device only by kernel code after modern PCI transport discovery and feature negotiation; no userspace process receives a DMA buffer, physical address, or BAR mapping.

Implementation note, 2026-05-02: the current QEMU virtio-net DMA path routes those kernel-owned pages through a bounded `device_dma` pool ledger. The net smoke proves live pool bytes, page counts, page-rounded MMIO mapping bytes, config/RX/TX interrupt holds, RX/TX ring depths, and RX/TX submission/completion and in-flight descriptor accounting. This is the first kernel-owned `DMAPool` accounting proof; it does not expose userspace DMA, MMIO, or interrupt handles and does not complete the production S.11.2 hostile-smoke gate.

Implementation note, 2026-05-02 06:59 UTC: the kernel-owned `device_dma` ledger now has an explicit bounded budget/OOM policy for the current virtio-net proof path: 32 DMA pages, 131072 DMA bytes, queue depth 8, submission depth 8, four page-rounded MMIO mapping holds, 16384 MMIO mapped bytes, and three interrupt holds. `make run-net` emits a scratch-ledger `device-dma: budget oom proof ... proof_result=ok` line proving page and byte allocation over budget, overlarge queue depth, duplicate and over-budget MMIO holds, MMIO byte over budget, duplicate and over-budget interrupt holds, and descriptor submission beyond queue depth all fail closed without mutating the live virtio-net ledger; the proof also revalidates the live ledger. This is a bounded prerequisite for the production `DMAPool` contract. It still does not expose userspace `DMAPool`, `DeviceMmio`, or `Interrupt` handles, wire real lifecycle hooks, program IOMMU remapping domains, or close the S.11.2 hostile smoke matrix.

Implementation note, 2026-05-02 16:31 UTC: attached device-manager `DMAPool` records now also carry that budget profile. The lifecycle and imported live-accounting proofs read page, byte, queue-depth, submission-depth, MMIO mapping, MMIO byte, and interrupt-hold budgets through the active device-manager record plus the matching `DmaPoolHandle`. Queue and

submission depth remain per-queue limits; the manager proof records queue_count=2 plus derived aggregate in-flight/submission budgets and checks imported virtio-net accounting against those aggregate totals. This keeps the budget policy tied to manager state but still does not create userspace DMAPool handles or enforce budgets at userspace handle creation, transfer, or revoke.

Implementation note, 2026-05-02 21:13 UTC: the zero-live device-manager DMAPool lifecycle proof now validates a proof-scoped tampered AttachedDmaPoolBudgetPolicyRecord through the manager budget-policy helper while the attached pool record is active. The tampered record uses the wrong policy scope/source/label plus stricter page, byte, queue, in-flight, MMIO, interrupt, and submission budgets, and it fails closed before it can be treated as a usable policy. The QEMU proof records budget_policy_tamper_result=fail-closed, budget_policy_tamper_allocation=not-created, budget_policy_tamper_ledger=not-mutated, budget_policy_tamper_teardown=not-advanced, and budget_policy_tamper_side_effect=side-effect-blocked. This is still bounded metadata proof only: no userspace DMAPool handle is exposed, no production userspace DMA page is allocated, freed, or reused, and no real DMA teardown is claimed.

Implementation note, 2026-05-02 22:16 UTC: the manager-owned DMAPool budget-accounting proof now fails closed on accounting over the attached budget instead of only logging passive booleans. The positive zero-live and imported-live budget_*_within_policy=true labels call a helper that revalidates the active attached record, matching DmaPoolHandle, owner, active state, and attached budget policy before accepting the record accounting. While the zero-live record remains active, synthetic attached-accounting candidates exceed buffer count, page count, byte count, and the current aliased in-flight/submission total; each candidate fails closed before it can be treated as usable manager state. The QEMU proof records exact overrun reasons, no fake allocation, no ledger mutation, no teardown advancement, and side-effect blocking. A proof-scoped over-budget attach candidate now fails before pool generation allocation and records preserved generation state. This remains bounded manager-record evidence only; later grant slices add the first single-page bounce-buffer allocation/free authority, but multi-buffer allocation, DMA mapping, descriptor execution, IOMMU programming, production driver consumption, and S.11.2 hostile smokes remain open.

Implementation note, 2026-05-02 23:59 UTC, updated 2026-05-03: the schema and kernel now include a result-only DMAPool.info skeleton that can wrap a manager-issued DmaPoolHandle. The object validates the live device-manager record through validate_dmapool_record() before returning status labels such as userspaceDmaPool=manager-issued-bounce-buffer, realDma=not-attempted, hostPhysicalUserVisible=false, and directDma=blocked. The QEMU zero-live device-manager lifecycle proof constructs that cap object while the attached record is active, records dmapool_cap_info_result=ok, exercises the serialized CapObject::call(0, &[]) path and decodes the returned DMAPool.info Cap'n Proto result as dmapool_cap_serialized_call_result=ok, then verifies the same cap fails closed after revoke begins as dmapool_cap_stale_after_revoke_result=dmapool-stale-handle; the same stale object also fails the serialized method-0 path as dmapool_cap_serialized_stale_after_revoke_result=invoke-failed. The same proof now exercises DMAPool.allocateBuffer through call_with_table() on a real DmaPoolCap in a CapTable: it decodes bufferIndex=0, verifies CAP_CQE_TRANSFER_RESULT_CAPS, cap_count=1, the

transfer-result record's DMABUFFER_INTERFACE_ID, the non-transferable same-session result-cap hold, and DMABuffer.info through the returned result cap. Duplicate allocation and stale-after-revoke allocation both fail closed without adding result caps. The duplicate-active valid-size path now reports a structured schema result with result=dmapool-already-attached, reason=active-buffer-attached, sideEffect=side-effect-blocked, and bufferPresent=false; the duplicate path also preserves the manager generation counter. Invalid-size requests use the same no-result-cap response shape with result=dmapool-allocation-request-invalid, the exact request reason, sideEffect=side-effect-blocked, and bufferPresent=false. As of the 2026-05-08 DMAPool grant-source follow-up, the same bounded path is also available through KernelCapSource::DmaPool: the grant source attaches a fresh zero-live manager-owned pool record, stages matching zero-live release evidence, and lets the child mint one DMABuffer result cap. A 2026-05-09 release-order follow-up has the smoke explicitly release the parent DMAPool before the result DMABuffer; the parent detach remains pending until the DMABuffer frees the page and completes the staged zero-live pool detach. A later 2026-05-09 follow-up adds a typed DMABuffer.freeBuffer method for that bounded result cap: the method reuses the same FreeBuffer authority validation and page scrub/ledger/frame-free cleanup path as cap release, emits a free-buffer audit event, invalidates later DMABuffer.info, and makes the later cap release a no-op detach. A second bounded follow-up keeps the parent DMAPool live after that first explicit free, reallocates the same slot with a fresh slotGeneration, and then repeats the parent-first release proof on the second buffer. The focused read-side HardwareAuditLog.snapshot smoke also decodes both slot generations, both typed free-buffer records, the parent DmaPool release, and both no-op release-after-free records through the volatile audit cap. The run-net DMABuffer driver-crash and reset-disable proofs also cover the same pending-parent completion path so successful buffer cleanup cannot orphan the staged parent release. A later admission follow-up adds typed DMABuffer.submitDescriptor to the same manifest-granted bounce-buffer path: the method validates the active manager-attached buffer epoch through the existing DmaBufferOperation::SubmitDescriptor authority validator, echoes the queue/descriptor/length and generation identity, and proves the same call fails closed after freeBuffer revokes the old cap. A later symmetric follow-up adds typed DMABuffer.completeDescriptor to the same bounded path. The 2026-05-10 request-shaping follow-up routes both typed descriptor calls through a shared pure bounded descriptor validator: valid bounce-buffer requests return ok request labels plus queue_count=4, descriptor_count=8, and buffer_bytes=4096, while out-of-range queues/descriptors, zero submit lengths, submit lengths beyond the bounce buffer, and completion lengths beyond the bounce buffer fail closed as dmabuffer-descriptor-request-invalid with side-effect-blocked before any descriptor side effect. A later manager-accounting follow-up records only the bounded manager counter: submit returns manager-inflight-recorded and raises DMAPool.info live_inflight to 1, completion returns manager-inflight-completed and restores it to 0, and a valid completion with no outstanding submission returns dmabuffer-no-inflight-submission. Too-small descriptor result buffers are rejected before accounting mutation, and cap-table release drains bounded in-flight accounting before detaching the bounce buffer. The 2026-05-10 06:37 UTC follow-up makes this allocateBuffer/freeBuffer page lifecycle the first production-labeled single-page bounce-buffer allocation/free authority. The typed surfaces report userspaceDmaPool=manager-issued-bounce-buffer, allocation=single-bounce-buffer-page, recordPool=userspace-bounce-buffer-live, zero-live-dmapool-bounce-buffer, and freeBuffer=bounce-buffer-page; the

underlying device_dma ledger uses a manager-attached bounce-buffer helper and scrubs before frame free. A 2026-05-10 11:44 UTC follow-up extends that bounded manifest-granted path to two fixed manager-owned slots; a 2026-05-10 12:49 UTC follow-up extends the same path to three fixed manager-owned slots: slot 0, slot 1, and slot 2 can be live together, DMAPool.info reports three live buffers/pages while all are attached, a fourth allocation fails closed as dmapool-already-attached, and slot 0 can be freed and reused with a fresh generation while slots 1 and 2 remain live. There is still no allocation beyond those three fixed slots, real device-visible DMA mapping, host physical address or IOVA exposure, BAR mapping, production descriptor-ring mutation, CQ publication, IOMMU programming, or production driver consumer. Stale allocation attempts preserve the live backing, and page allocation failure occurs before buffer-generation allocation so it does not burn a generation. The 2026-05-10 13:45 UTC follow-up (3bbeb3d4) adds explicit typed DMABuffer.unmap for the mapped bounce-buffer userspace VMA. The method validates the live DMABuffer handle before reporting success or no-op, removes only the borrowed VMA owned by that mapping for the calling process, and publishes the mapping as absent only after the borrowed-range ownership check, page-table unmap, TLB wait, and waiter cleanup succeed. While teardown is in progress, concurrent map/free/release paths fail closed against an in-progress mapping state. A second unmap returns dmabuffer-mapping-absent / no-user-mapping with no side effect. This is userspace VMA cleanup only: it does not free or scrub the bounce page, detach the buffer record, change DMAPool.info live buffer/page/in-flight counts, program or remove real DMA or IOMMU mappings, expose host physical/IOVA addresses, mutate descriptor rings, publish CQ entries, or add a production driver consumer. The 2026-05-10 14:12 UTC follow-up moves bounded descriptor accounting from a single pool-global descriptor identity to per-slot state on each live manager-owned DMABuffer record. DMAPool.info live_inflight remains the aggregate sum across live slots. A valid submit on slot 0 and a valid submit on slot 1 can coexist; duplicate submit on either same slot still fails closed; mismatched completion preserves that slot's descriptor without touching other slots; matching completion of slot 0 decrements the aggregate while slot 1 remains in flight; explicit freeBuffer of an in-flight slot fails closed; and cap-release/process-exit cleanup drains only the releasing slot's descriptor before detach. This is still bounded manager accounting and does not mutate descriptor rings, publish CQ entries, expose host physical or IOVA addresses, attempt direct DMA, program IOMMU state, or add a production driver consumer. The 2026-05-10 18:11 UTC follow-up makes the single manager-owned bounce-buffer page exclusive between userspace borrowed-VMA ownership and manager in-flight descriptor ownership for each DMABuffer cap. A valid submit while the same cap still has a live mapping fails closed as dmabuffer-mapping-live / user-mapping-live before manager in-flight accounting changes; explicit unmap restores submit. A valid map while the slot has an in-flight descriptor fails closed as dmabuffer-inflight-submission / in-flight-submission, returns addr=0, and does not publish a borrowed VMA; matching completion restores map for that slot. The lock order remains cap mapping state before manager validation, with no address-space lock held across manager state mutation. The 2026-05-10 19:29 UTC follow-up adds bounded completion data on that same manager-owned bounce-buffer page. The successful matching DMABuffer.completeDescriptor path keeps the existing manager-inflight-completed result labels, validates the active owner, pool/slot generation, queue/descriptor identity, and submitted length, then writes a deterministic byte pattern into only the accepted completionLength bytes before clearing the in-flight record. Because submit is blocked while a cap-owned borrowed VMA is live and map is blocked while

the slot is in flight, the write happens while no live user VMA exists for that slot; a later successful map lets userspace read the pattern. Invalid requests, stale caps, no-inflight completions, descriptor mismatches, length-exceeded completions, mapped-live cases, and after-free calls do not write. This is userspace-visible bounce-buffer completion data, not device DMA completion: there is still no descriptor-ring mutation, CQ publication, direct DMA, host physical/IOVA exposure, IOMMU programming, or production driver consumer. Implementation note, 2026-05-10 04:40 UTC: duplicate-active valid-size and invalid-size `DMAPool.allocateBuffer` requests now use schema result data for domain rejection instead of an application-exception label. The no-result-cap response reports either `result=dmapool-already-attached` / `reason=active-buffer-attached` or `result=dmapool-allocation-request-invalid` with the exact request reason, plus `sideEffect=side-effect-blocked` and `bufferPresent=false`, before any allocation side effect. The 2026-05-10 live-accounting follow-up also carries that bounded frame into the attached `DMAPool` record exposed by typed `DMAPool.info`: the manager record starts as `zero-live-dmapool-bounce-buffer`, changes to `userspace-bounce-buffer-live` with one, two, or three live 4096-byte pages while manager-attached `DMABuffer` result caps exist, and returns to zero-live after typed `DMABuffer.freeBuffer` or cap release scrubs/releases every live bounce page. The same lifecycle proof validates that active manager-record accounting against the attached budget policy before treating allocation as usable state. This still does not expose a device-visible DMA address, IOVA, host physical address, production descriptor side effect, DMA mapping, or production driver consumer.

Implementation note, 2026-05-11 03:00 UTC: the manifest-granted manager-owned fixed bounce-buffer `DMAPool` path now has its own device-manager budget policy instead of importing the live virtio-net `device_dma` policy. The policy covers three live buffers/pages, 12288 bytes, four queues, eight descriptors per queue, one in-flight descriptor per live slot, zero MMIO mappings/bytes, and zero interrupt holds. `DMAPool.allocateBuffer` validates the next-live manager accounting against that policy before slot selection, frame allocation, generation allocation, result-cap minting, or manager ledger mutation. With all three fixed slots live, a fourth valid-size allocation returns no result cap and reports `result=dmapool-budget-exceeded`, `reason=over-buffer-budget`, `sideEffect=side-effect-blocked`, and `bufferPresent=false`. Imported live virtio-net proof records continue to use the `device_dma:virtio-net` budget policy. This remains the bounded manager-owned bounce-buffer path: direct DMA is blocked, host physical addresses and IOVAs stay hidden, descriptor rings and completion queues are not mutated, and IOMMU/remapping plus production driver consumption remain out of scope.

Implementation note, 2026-05-11 06:10 UTC: the same fixed-slot manager-owned `DMAPool` family now revalidates current budget accounting before publishing an allocated `DMABuffer` result cap, before acquire-audit publication, before parent `DMAPool` release intent or detach, before grant-rollback/drop/teardown detach, before pending-parent release completion, before `DMABuffer` page release, and before descriptor-completion state advancement. The focused grant smoke labels the full-pool budget rejection as no leaked result cap, no generation burn, no ledger mutation, and no stale authority publication. `DeviceMmio` and `Interrupt` budget propagation is not changed by this slice.

Implementation note, 2026-05-03 02:05 UTC: the schema and kernel now include a result-only `DMABuffer.info` skeleton that can wrap the manager-attached `DmaBufferHandle` record already issued inside the zero-live `DMAPool` lifecycle proof. The object validates the live manager-owned

buffer record through validate_dmabuffer_record() and the existing pure DMA-buffer validator before returning status labels such as userspaceDmaBuffer=manager-issued-bounce-buffer, managerRecord=validated-active, bufferRecord=manager-attached-buffer, realDma=not-attempted, hostPhysicalUserVisible=false, directDma=blocked, descriptorSubmit=manager-inflight-accounting, descriptorComplete=manager-inflight-accounting, freeBuffer=bounce-buffer-page, and bootstrapGrant=blocked. The QEMU zero-live lifecycle proof constructs that cap object while the buffer record is active, records dmabuffer_cap_info_result=ok, exercises the serialized CapObject::call(0, &[]) path and decodes the returned DmaBuffer.info Cap'n Proto result as dmabuffer_cap_serialized_call_result=ok, then verifies the same cap fails closed after revoke begins as dmabuffer_cap_stale_after_revoke_result=dmabuffer-stale-handle; the same stale object also fails the serialized method-0 path as dmabuffer_cap_serialized_stale_after_revoke_result=invoke-failed. Later bounded grant slices expose this result cap through the DMAPool manifest-grant path, add typed DMABuffer.freeBuffer, and add bounded userspace bounce-buffer DMABuffer.map / DMABuffer.unmap plus manager-accounted request-shaped DMABuffer.submitDescriptor / DMABuffer.completeDescriptor; the path still exposes no real DMA, descriptor-ring mutation, CQ publication, host physical address or IOVA export, production page cleanup/reuse, or production userspace DMABuffer completion.

Implementation note, 2026-05-11 11:22 UTC: the provider-consumer smoke now uses that same bounded bounce-buffer path to prove a descriptor-ring-equivalent provider side effect. After DMABuffer.submitDescriptor validates live owner/pool/slot authority, descriptor queue/id/length bounds, no live user mapping, and no duplicate in-flight descriptor, the manager scrubs the page and writes a provider-visible shadow descriptor entry with magic, queue, descriptor id, submitted length, and flags before writing the existing submit marker and committing the in-flight record. DMABuffer.completeDescriptor still writes completion bytes only inside the validated completion length, and the smoke proves the shadow entry and marker remain visible outside that completion window. Provider-effect submits shorter than the 24-byte shadow-descriptor-plus-marker footprint are rejected as a typed no-side-effect boundary, even though the shared descriptor request shape is otherwise valid. This is a bounded bounce-buffer side effect only; no hardware descriptor ring, CQ publication, MMIO doorbell, direct DMA, IOVA, host physical address, or remapping-domain claim is added.

Implementation note, 2026-05-11 12:01 UTC: the same submit path now replaces the shadow-descriptor payload with a selected provider-owned queue entry plus marker. The entry records queue magic, queue id, tail, descriptor id, submitted length, and flags after descriptor authority validation and the submit scrub; make run-ddf-provider-consumer maps the buffer after completion and proves the queue entry and marker remain visible outside the completion window. Provider-effect submits shorter than the current 72-byte provider queue-entry-plus-marker footprint are rejected before in-flight accounting or provider-visible mutation. This remains bounded bounce-buffer evidence only: no hardware descriptor ring, CQ publication, MMIO doorbell, direct DMA, IOVA, host physical address, or remapping-domain claim is added.

Implementation note, 2026-05-12 20:30 UTC: the accepted DMABuffer.submitDescriptor path now constructs the candidate in-flight descriptor in manager-local state and validates the resulting DMAPool budget/accounting before scrubbing or writing provider-visible bounce-buffer

bytes. The provider-consumer smoke snapshots the selected provider queue-entry and marker bytes, drives a short provider-effect submit rejection, and proves both those bytes and `live_inflight=0` are preserved. The selected virtio-net TX publication gate remains separately bounded after provider-entry write: quiesced publication still fails with no extra pin and no doorbell, not with rollback of the already-written shadow entry.

Implementation note, 2026-05-11 14:39 UTC, branch commit `f04a14f4`: the selected provider-owned queue entry now carries a staged claimed virtio-net notify-offset admission record instead of only a "requires claim" gate. The selected queue `1` path records accepted notify-offset admission plus blocked wrong-queue and wrong-offset admissions after descriptor authority validation and submit scrub; a separate queue `0` submit proves non-selected queues remain neutral and blocked from selected-backend doorbell metadata. This is not a real doorbell path: no virtio-net notify BAR handle is granted, no notify register is written, no real virtio-net descriptor ring is mutated, and production userspace NIC readiness is not claimed.

Current QEMU evidence: the same `make run-net` path now exports a bounded live-pool snapshot from the kernel-owned virtio-net `device_dma` ledger and feeds it into a device-manager `DMAPool` record proof. The live record carries buffer/page count, live bytes, current in-flight submissions, committed/resident/unswappable flags, and scrub-before-release policy. The device-manager proof rejects both `DmaMappingsRemoved` and teardown detach while that authoritative ledger snapshot remains live. The proof now calls the `device_dma` teardown-evidence API and records the expected `authoritative-ledger-live` block with matching imported live accounting, then reports completion as deferred because no authoritative zero-live/scrubbed evidence is available for the live virtio-net ledger. It does not zero the imported record to simulate teardown, does not claim `DmaMappingsRemoved`, terminal `Dead`, or release for the live virtio-net pages, and does not scrub or free live virtio-net DMA pages. This is still a prerequisite record-accounting proof: the current pages remain kernel-owned, only bounded info-skeleton hardware cap grants are exposed to userspace, production page release hooks for live virtio-net DMA are not wired, IOMMU remapping is not programmed, and S.11.2 hostile smokes remain open.

The same smoke emits a separate `device_dma` scratch proof for the positive zero-live teardown-evidence path: `teardown_evidence()` fails closed before quiesce and scrub markers, rejects one-marker states, and reports `authoritative-ledger-zero-live` only after both markers are set. The scratch proof revalidates the live virtio-net ledger but does not mutate, zero, scrub, free, or claim teardown completion for real virtio-net DMA pages.

Implementation note, 2026-05-03 03:21 UTC: the zero-live device-manager `DMAPool` lifecycle proof now consumes that scratch zero-live evidence before final pool detach, and binds that evidence to the attached record source. The manager-owned zero-live record is labeled `device-manager` / `zero-live-dmapool-bounce-buffer`; imported live records keep the source labels from the authoritative `device_dma` snapshot. After the proof-scoped buffer record is actively freed and detached, a zero-live pool detach with mismatched `virtio-net` / `kernel` scratch evidence fails closed as `dmapool-zero-live-evidence-invalid`, and detach without authoritative evidence fails closed as `dmapool-zero-live-evidence-absent`. Only scratch `authoritative-ledger-zero-live` evidence carrying the same record source plus both quiesce and scrub markers allows the manager-owned record to detach and the revocation path to advance to `DmaMappingsRemoved`, `Dead`, and release. This remains scratch/no-real-DMA evidence: it does not tear down live virtio-

net pages, program or remove IOMMU mappings, expose userspace `DMAPool` mapping/descriptor authority, or claim production page cleanup beyond the bounded manager-attached bounce page.

The current scratch proof set also covers stale DMA page handles without touching real virtio-net pages: reusing the same synthetic physical page bumps the DMA page generation, the old handle fails closed as `stale-dma-handle`, wrong-queue and wrong-label frees preserve the active page record, and duplicate free remains rejected. Production userspace `DMAPool` stale-handle smokes, descriptor-abuse coverage, revoke/reset races, and real quiesce/scrub/release remain open.

Implementation note, 2026-05-02 23:23 UTC: the kernel-owned virtio-net `device_dma` page release path now validates the `DeviceDmaAllocation` against the live ledger before any scrub or frame-allocator call, scrubs the frame through the HHDM mapping, removes the ledger entry only after scrub succeeds, and then returns the frame to the allocator. The frame scrub helper checks frame alignment, HHDM/allocator initialization, range, and allocated state before zeroing the page. `make run-net` emits a bounded `device-dma: release scrub proof` line using a proof-only kernel-owned page: stale generation, wrong queue, and wrong label release attempts fail before scrub, frame free, or ledger mutation; the active release path records `scrub_before_frame_free=true` and `ledger_removed_after_scrub=true`. This is still no-userspace-handle/no-real-teardown evidence for the current kernel-owned virtio-net path only; production `DMAPool` handles, real device-manager teardown hooks, IOMMU/bounce-buffer mapping removal, hostile stale-DMA smokes, and full page lifecycle cleanup remain open.

Implementation note, 2026-05-02 12:28 UTC: the same scratch proof family now covers stale DMA completion ordering without touching real virtio-net pages. A synthetic reused DMA page slot bumps generation, stale completion validation checks the page generation before queue-completion accounting, and the old completion fails closed as `stale-dma-handle` before completion counters can underflow or the reused page/submission state can mutate. This is still prerequisite evidence only: production userspace `DMAPool` hostile smokes, reset/revoke races with outstanding descriptors, CQ notification publication, real quiesce/scrub/release, and IOMMU or bounce-buffer teardown remain open.

Implementation note, 2026-05-02 14:03 UTC: that scratch stale-DMA-completion proof now adapts the synthetic DMA buffer slot into `capos-lib::device_authority` before completion accounting can mutate. The QEMU line records current-handle validation as `ok`, stale same-slot reuse as `stale-slot-generation`, and `side-effect-blocked`, then preserves the existing `stale-dma-handle` completion outcome. This remains a scratch/no-real-DMA validator adapter, not production userspace `DMAPool` authority or S.11.2 hostile-smoke completion.

Implementation note, 2026-05-02 15:15 UTC: the same scratch proof family now adds a paired stale-completion publication check. A synthetic reset bumps the device owner generation so an old completion fails as `stale-owner-generation` with `side-effect-blocked` before any CQ publication. The same-slot reuse path then fails the old completion as `stale-dma-handle`, preserves the new submission accounting, and records both `cq_publication_blocked=true` and `new_owner_exposure_blocked=true`. This is still scratch/no-real-DMA evidence; production userspace `DMAPool` completion notification, real hardware stale-completion injection, reset/revoke races, and IOMMU or bounce-buffer teardown remain open.

Implementation note, 2026-05-02 12:46 UTC: the device-manager interrupt handoff proof now includes a bounded stale IRQ after-detach check. After an attached interrupt route is detached, the proof delivers the old LAPIC vector through the dispatch path and requires `stale_irq_delivery_after_detach` to be `unregistered` with `stale_irq_wake_blocked=true`; the old route handle also continues to fail as `interrupt-stale-route`. This is prerequisite evidence for route teardown ordering only. It does not cover a pending hardware IRQ across reset, userspace `Interrupt` waiter wakeup semantics, or reassignment reuse by a new owner.

Implementation note, 2026-05-02 14:24 UTC: the same device-manager interrupt handoff proof now adapts the attached source into `capos-lib::device_authority` before exercising the active wait path. After revoke begins, the old handle fails the pure validator as `stale-owner-generation` with `side-effect-blocked`, then the proof preserves the existing `interrupt-stale-route`, detached-vector `unregistered`, and `stale_irq_wake_blocked=true` checks. This remains a proof adapter, not production userspace `Interrupt` handles, real waiters, reset/reassignment reuse proof, or S.11.2 hostile-smoke completion.

Implementation note, 2026-05-02 14:51 UTC: the interrupt handoff smoke now adds two bounded stale IRQ ordering points. After revoke begins and while the route is still attached, delivery to the old vector remains `masked`, matches the attached route generation, reports wake blocking, and leaves the old route delivery count unchanged. During the reset phase, a synthetic route-registry same-vector reuse proof re-registers and claims the route with bumped source and route generations, then shows delivery to that vector is still `masked`, matches the new route generation, and leaves the reused route delivery count unchanged. This is route-manager prerequisite evidence only: it is not a true pending hardware MSI/reset hostile smoke, does not involve userspace `Interrupt` waiters, and does not prove DMA buffer reuse race closure.

Implementation note, 2026-05-03 18:48 UTC: the interrupt handoff smoke now snapshots a bounded pending IRQ token from the old vector, source id, source generation, and route generation before revocation. Checking that token after revoke blocks as `stale-pending-irq-masked` with reason `route-masked`; after detach it blocks as `stale-pending-irq-unregistered`; and after reset/reuse it blocks as `stale-pending-irq-generation` with reason `source-route-generation-mismatch`. Each check records `side-effect-blocked`, wake blocking, and unchanged delivery counts, and the reset/reuse check records that the new route did not receive a delivery count. The same proof rejects a malformed pending token with a zero generation as `stale-pending-irq-invalid-state` before any delivery-count mutation. This was bounded token-generation evidence only and did not inject a real pending MSI across reset; the real-`int $vector` injection added at `2026-05-05 18:17 UTC` (see below) closes the S.11.2.7 stale IRQ hostile-smoke gate row by exercising the production CPU exception entry path across the same revoke, detach, and reset/reuse boundaries.

Implementation note, 2026-05-09 18:12 UTC: the pending IRQ token decision path now has a pure `capos-lib::device_authority` validator. Host tests cover current-route acceptance and the same fail-closed label space used by the kernel/QEMU proof: stale source generation, stale route generation, both generations changed after reuse, source mismatch, route masked, route unregistered, invalid route state, invalid owner, malformed zero-generation or unassigned source identity, and unsupported vector. The kernel still snapshots the live dispatch slot and delivery count, but delegates the pending-token identity/state decision to that shared helper before

returning stale-pending-irq-* labels. This is validator/adapter coverage only; it does not expose production userspace Interrupt waiters or wait/ack/mask/unmask authority.

Implementation note, 2026-05-05 18:17 UTC: the interrupt handoff smoke now fires a real INT $vector instruction at the device MSI vector at three points across the revoke/reset/reuse boundary, exercising the production IDT entry, extern "x86-interrupt" stub, record_lapic_delivery dispatch slot read, and LAPIC EOI write rather than the helper-call path the prior proofs used. The new proof scope strings drop "no-real-msi" and read stale-vector-after-detach-real-int-vector-injected, manager-attached-claimed-masked-after-revoke-real-int-vector-injected, route-registry-vector-reuse-during-reset-real-int-vector-injected, and bounded-pending-irq-token-generation-real-int-vector-cross-reset-injection. Each injection point requires the slot's delivery count to remain zero before and after the real INT, and the post-INT outcome to match masked (after revoke, route still attached but masked), unregistered (after detach, slot cleared), and masked (after reset/reuse, slot now belongs to a freshly registered+claimed route with bumped source and route generations). The proof emits a closure summary s11_2_7_proof_scope=s11-2-7-stale-irq-after-reset-real-int-vector-cross-reset-injection-no-userspace-waiter, s11_2_7_real_irq_injected_across_reset=ok, s11_2_7_old_waiter_cannot_wake_new_owner=true, and s11_2_7_stale_ack_blocked=true. This closes the S.11.2.7 row of the hostile-smoke acceptance matrix at make run-net (which invokes tools/qemu-net-smoke.sh). It does not yet create a userspace Interrupt waiter object; the in-flight delivery is observed via the kernel-owned dispatch slot atomic state machine that the production path consumes. S.11.2.9 hostile-smoke gate-wiring closed 2026-05-05 20:49 UTC (see the implementation note below).

Implementation note, 2026-05-05 19:37 UTC: the device-manager hostile-smoke suite now closes the S.11.2.8 stale-DMA-completion-after-reset row. prove_qemu_stale_dma_completion_handoff claims a fresh probe-then-driver record on the virtio-net PCI BDF (separate from the live virtio-net driver state) and walks it through the same revoke, detach, and reset/reuse boundaries S.11.2.7 uses. At each boundary the proof allocates a real virtio-net DMA page through the production device_dma::allocate_virtio_net_page helper, frees the page through device_dma::free_virtio_net_page, reallocates so the live ledger's page generation advances, and synthesizes a stale DeviceDmaAllocation keyed to the live phys with a decremented generation. The synthesized stale handle is then fed to the production device_dma::record_virtio_net_completion_for_allocation path – the same function the live virtio-net Virtqueue::record_used_completion_for_allocation invokes after descriptor tracking validates a hardware used-ring entry. The production validator rejects each stale injection as stale-dma-handle before any queue accounting decrement, completion side effect, CQ publication, or new-owner memory exposure. The bounded run-net proof records real_completion_inject_after_revoke_result=stale-dma-handle, real_completion_inject_after_detach_result=stale-dma-handle, real_completion_inject_after_reset_reuse_result=stale-dma-handle, all three with side-effect-blocked, queue_account_preserved=true, live_page_preserved=true, cq_publication_blocked=true, new_owner_exposure_blocked=true, freed_buffer_unchanged=true, and generation_bumped=true, plus a closure summary s11_2_8_proof_scope=s11-2-8-stale-dma-completion-after-reset-real-free-realloc-

cross-revoke-detach-reset-reuse-no-userspace-dmapool,
s11_2_8_real_completion_injected_across_reset=ok,
s11_2_8_old_completion_cannot_publish_to_new_owner=true,
s11_2_8_freed_buffer_reuse_blocked=true, and
s11_2_8_accounting_underflow_blocked=true. The new shape is enforced in tools/qemu-net-smoke.sh and runs from make run-net. This is the production paired stale-DMA-completion proof showing old completions cannot publish stale CQ notifications or expose new-owner memory after real revoke, detach, and reset/reuse boundaries with real free + realloc page generation advances on the live kernel-owned ledger; S.11.2.9 hostile-smoke gate-wiring closed 2026-05-05 20:49 UTC (see the implementation note below). Userspace DMAPool handles and real device-manager page quiesce/scrub/release hooks remain open as separate follow-ups.

Implementation note, 2026-05-05 20:49 UTC: the S.11.2.9 hostile-smoke coverage row of the acceptance matrix is closed by aggregating every matrix-row proof line into the make run-net -> tools/qemu-net-smoke.sh gate. Every proof line referenced by the matrix has at least one assertion in the harness today; the assertion shape varies by row. The two driver-crash lines wired by that gate slice, the existing S.11.2.8 device-manager: dma completion handoff proof closure-summary line, and the S.11.2.7 device-manager: interrupt handoff proof closure-summary line (whose trailing anchor was added by this slice for harness-strictness consistency with S.11.2.8) all use anchored extended-regex assertions (field-by-field match plus proof_result=ok[[:cntrl:]]?$ trailing anchor); other matrix-row rows reuse the harness's pre-existing mix of unanchored extended-regex and fixed-string grep -Fq assertions on the emitted proof lines. The two previously unasserted lines wired by this slice are device-manager: devicemmio driver crash hook proof source=devicemmio-driver-crash-hook ... trigger_path=trigger-driver-crash-for-devicemmio and device-manager: interrupt driver crash hook proof source=interrupt-driver-crash-hook ... trigger_path=trigger-driver-crash-for-interrupt. Both proofs were already emitted by the kernel on every boot (via prove_qemu_devicemmio_driver_crash_hook and prove_qemu_interrupt_driver_crash_hook in kernel/src/device_manager/proofs.rs) and exercise the explicit driver-crash teardown trigger path with a stale rerun noop, validate-live revoked cap state, and cap_release_after_crash as noop. The chosen gate strategy keeps S.11.2.9 inside make run-net rather than splitting into a separate make run-hostile-smokes target, because all six matrix rows depend on the same virtio-net device bring-up state (probe-then-driver records on the virtio-net BDF, real IDT vector injection, real DMA page free + reallocate). A separate target would duplicate the bring-up cost without adding coverage. Tightening the remaining unanchored assertions to the same anchored shape is a follow-up harness-hardening task and is not part of S.11.2.9 closure because each affected proof line is still uniquely identified by its emitted prefix and the asserted field set. Production userspace DMAPool/DeviceMmio/Interrupt handles, real device-manager page quiesce/scrub/release hooks, hardware-backed provider-driver Interrupt wait/ack dispatch beyond the current bounded route-dispatch waiter proof, durable/signed production audit consumption beyond the first volatile HardwareAuditLog.snapshot cap, and IOMMU domain programming all remain open as separate follow-ups tracked in docs/backlog/hardware-boot-storage.md and the docs/tasks/README.md userspace-driver-transition bullet.

Implementation note, 2026-05-08 09:44 UTC: the same make run-net gate now also asserts cap-specific DMA driver-crash proofs. DmaBufferCap routes the explicit trigger through the bounded FreeBuffer cleanup path and proves page scrub/ledger/frame-free labels before stale rerun and post-trigger cap release both return noop; DmaPoolCap routes the explicit trigger through the zero-live evidence-gated detach path and proves authoritative zero-live, quiesced, and scrubbed evidence labels before stale rerun and post-trigger cap release return noop.

Implementation note, 2026-05-03 01:05 UTC: the schema and kernel now include a result-only Interrupt.info skeleton that can wrap a manager-issued device handle plus the attached DeviceInterruptRoute. The object validates the live manager record, owner, claimed route, and attached route record through validate_interrupt_record() before returning status labels such as userspaceInterrupt=manager-issued-skeleton, managerRecord=validated-active, routeRecord=manager-attached-route, realInterruptDelivery=not-delivered, wait=admission-check-only, acknowledge=admission-check-only, mask=route-state-control, unmask=route-state-control, and bootstrapGrant=blocked. The interrupt handoff QEMU proof constructs that cap object while the route record is active, records interrupt_cap_info_result=ok, exercises the serialized CapObject::call(0, &[]) path and decodes the returned Interrupt.info Cap'n Proto result as interrupt_cap_serialized_call_result=ok, then verifies the same cap fails closed after revoke begins as interrupt_cap_stale_after_revoke_result=interrupt-stale-handle; the same stale object also fails the serialized method-0 path as interrupt_cap_serialized_stale_after_revoke_result=invoke-failed. A later manifest-grant smoke explicitly releases the granted Interrupt cap through CAP_OP_RELEASE and proves a subsequent typed Interrupt.info call fails closed from userspace. A focused grant-cycle smoke now repeats that grant, release, and stale-info proof twice in sequence and asserts the second manager-grant-source acquire preserves the source generation and receives a fresh route generation after the first release; the same smoke also decodes both acquire/release cycles through the typed volatile HardwareAuditLog.snapshot surface. The focused hardware-audit interrupt-waiter smoke also decodes recent boot-time DmaBuffer, DmaPool, and Interrupt driver-crash / reset-disable lifecycle records from the current volatile 16-record snapshot window. The same smoke now uses the startSequence cursor to decode older retained DeviceMmio lifecycle rows that the default latest 16-record tail has skipped. A 2026-05-09 19:18 UTC follow-up adds a bounded Interrupt.wait admission method to that skeleton. The method validates the same manager-attached route, snapshots a pending-token candidate, delegates to the shared capos-lib::device_authority pending-IRQ validator, and returns typed labels through capos-rt; the focused grant smoke asserts the current masked-route result stale-pending-irq-masked, reason route-masked, side-effect-blocked, matching token/current source and route generations, unchanged delivery counts, no waiter wake, and fail-closed behavior after cap release. This is bounded manager-issued skeleton evidence only: there is no blocking wait, real hardware acknowledgement, real hardware mask/unmask side effect, interrupt delivery authority, real waiter object, production Interrupt completion, durable/signed audit persistence, or concurrent sharing claim. A 2026-05-09 23:21 UTC follow-up adds bounded Interrupt.acknowledge admission to the same skeleton. It validates the manager-attached route through the existing Acknowledge authority path, returns admission-check-only, interrupt-ack-not-attempted, and side-effect-blocked, and proves delivery counts remain unchanged with no waiter wake or hardware acknowledgement. A 2026-05-09 23:52 UTC follow-up adds

bounded Interrupt.mask and Interrupt.unmask admission to the same skeleton. They validate the manager-attached route through the existing Mask and Unmask authority paths, return admission-check-only, interrupt-mask-not-attempted / interrupt-unmask-not-attempted, and side-effect-blocked, and prove route state and delivery counts remain unchanged with no hardware mask/unmask, waiter wake, or IRQ delivery. A 2026-05-10 04:01 UTC follow-up promotes those methods to bounded route-state control over the manager-attached dispatch slot. Interrupt.unmask now changes claimed-masked to driver-unmasked, Interrupt.mask changes it back to claimed-masked, and both preserve delivery counts while still avoiding hardware MSI/MSI-X table programming, waiter wakeup, hardware acknowledgement, or real IRQ delivery. A 2026-05-10 22:54 UTC follow-up wires real waiter completion to the existing route-dispatch delivery counter from scheduler/poll context. The poller observes matching delivered routes by vector, source generation, and route generation without taking the waiter-table lock in the IRQ dispatch path, then revalidates the manager-attached route under the route-post exclusion before posting the deferred cap completion. The focused grant smoke proves the first unmasked manifest-granted wait completes as interrupt-delivered / waiter-completed-irq with real_interrupt_delivery=delivered and an advanced delivery count, while a second unmasked wait still remains pending until Interrupt.mask completes it through the existing interrupt-waiter-cancelled / waiter-completed-no-irq path. Stale, masked, released, reset, or reused routes do not wake as delivered IRQs. This remains a bounded route-dispatch proof; it does not program hardware MSI/MSI-X tables, acknowledge hardware, add provider-driver interrupt consumption, or claim hostile hardware isolation.

Implementation note, 2026-05-03 13:49 UTC: the result-only DMAPool.info, DMABuffer.info, DeviceMmio.info, and Interrupt.info surfaces now return numeric identity fields alongside the conservative status labels. The fields mirror the documented handle identity shape: deviceId, BDF bus/device/function, owner generation, and the relevant pool id/generation, buffer slot/generation, BAR/mapping id/generation, or interrupt source/generation/route generation. The QEMU proof logs and net smoke assert the active serialized method-0 decode for those fields, while stale method-0 calls still fail closed as invoke-failed. This remains a result-only manager-issued skeleton surface; it does not add production DMA allocation, free, map, submit, or completion authority, production MMIO mapping or doorbell authority, production interrupt wait/ack/mask/unmask authority, real DMA page cleanup/reuse, hostile hardware isolation, or S.11.2 completion.

Implementation note, 2026-05-03 16:37 UTC: the bounded interrupt route identity skeleton now carries separate source and route generations end to end. DeviceInterruptRoute, LegacyIoApicInterruptRoute, route records, diagnostics summaries, dispatch-slot metadata, and the device-manager attached interrupt bridge store both generations. Registration allocates both fields, PCI MSI-X route reassignment preserves source generation while bumping only the route generation, release/re-register allocates both generations fresh, and Interrupt.info returns the independent values. The QEMU net smoke asserts the split in PCI and legacy route logs, metadata proof logs, handoff proof logs, serialized Interrupt.info, and cap-release proof logs. This closes only the bounded identity proof gap; it does not expose production userspace Interrupt authority, create real waiters, or complete the S.11.2 hostile IRQ smoke matrix.

### `DeviceMmio` Invariants

`DeviceMmio` is register authority, not memory authority.

- **Authority:** A holder may map only BARs or subranges recorded in the claimed device object. It may not map PCI config space globally, another function's BAR, RAM, ROM, or synthetic kernel pages.
- **Handle identity:** Each call checks the claimed device id, owner generation, BAR or subrange mapping record, and mapping generation before mapping, unmapping, reading, or writing registers.
- **Physical range:** Each mapping is bounded to the BAR's decoded physical range, page-rounded by the kernel, and tagged as device memory with cache attributes appropriate for MMIO. Partial BAR grants must preserve page-level isolation; otherwise the grant must cover the whole page-aligned register window and be treated as that much authority.
- **Ownership:** At most one mutable driver owner controls a device function's MMIO at a time. Management capabilities may inspect topology, but register writes require the claimed `DeviceMmio` object.
- **No DMA implication:** Mapping registers does not grant any DMA buffer, frame allocation, interrupt, or config-space authority. Doorbell writes are accepted only as effects of register access; descriptor validity is enforced by `DMAPool` before queues are made visible to the device.
- **Revocation:** Revocation unmaps the driver's register pages, marks the device object unavailable for new calls, and invalidates outstanding MMIO handles. Stale mappings or calls fail closed.
- **Reset:** Revoking the final mutable `DeviceMmio` owner resets or disables the device unless a higher-level device manager explicitly transfers ownership without exposing it to an untrusted holder.

### `Interrupt` Invariants

`Interrupt` is event authority for one routed source.

- **Authority:** A holder may wait for, mask/unmask where supported, and acknowledge only its assigned vector, line, or MSI/MSI-X table entry. It may not reprogram arbitrary interrupt controllers or claim another source.
- **Handle identity:** Each wait, mask, unmask, and acknowledge checks the claimed device id, owner generation, source id, source generation, route generation, and any live waiter generation before affecting delivery state.
- **Ownership:** Each interrupt source has one delivery owner at a time. Shared legacy lines must be represented as a kernel-demultiplexed object with explicit device membership, not as ambient access to the whole line.
- **Range:** The capability records the hardware source, vector, trigger mode, polarity, and target CPU/routing state. User-visible operations are checked against that record.
- **Revocation:** Revocation masks or detaches the source, drains pending notifications for the old holder, invalidates waiters, and prevents stale acknowledgements from affecting a new owner.
- **Reset:** If the source cannot be detached cleanly, the owning device is reset or disabled before the interrupt is reassigned.

- **No MMIO or DMA implication:** Interrupt delivery does not grant register access, DMA buffers, or packet memory.

### Revocation Ordering

Device revocation must follow a fixed order:

1. Stop new submissions by invalidating the driver's user-visible handles.
2. Revoke MMIO write authority by write-blocking or unmapping BAR pages, or by disabling the device before any DMA teardown starts.
3. Mask or detach interrupts.
4. Quiesce virtqueues or device command queues.
5. Reset or disable the device if in-flight DMA cannot be accounted for.
6. Remove IOMMU mappings or invalidate bounce-buffer handles.
7. Scrub and free DMA pages.

This order prevents a stale driver from racing revocation with doorbell writes, interrupt acknowledgement, or descriptor reuse. Logical handle invalidation is not sufficient while a BAR remains mapped; register-write authority must be removed or the device must be disabled before descriptor or DMA-buffer ownership is reclaimed.

Implementation should represent the order as an explicit device-owner state machine rather than as ad hoc booleans:

```
enum DeviceOwnerState {
    Active,
    RevokingHandles,
    MmioRevoked,
    InterruptsDetached,
    QueuesQuiesced,
    Resetting,
    DmaMappingsRemoved,
    Dead,
}
```

No path may free or reassign DMA pages until the state has reached `QueuesQuiesced` with all in-flight descriptors accounted for, or `Resetting` has completed and the device can no longer write old buffers. `Dead` means all user-visible handles are invalid, interrupts are detached or masked, DMA mappings are removed, and pages have been scrubbed or transferred to a trusted owner.

Hard invariants:

- DMA pages cannot be freed before `QueuesQuiesced` or a completed `Resetting` transition proves old DMA writes are stopped.
- MMIO write authority must be revoked before DMA ownership teardown.
- Interrupt reassignment cannot happen before old pending notifications are drained or generation-invalidated.
- Device reset is mandatory if in-flight DMA cannot be proven stopped.

### Future Userspace-Driver Transition Criteria

Moving NIC or block drivers out of the kernel is gated by Security Verification Track S.11.2. The gate is only open when all rows below are implemented and demonstrated. The S.11.2.N labels are local checklist row IDs for this gate.

The completed Device Driver Foundation selected milestone used this track as the prerequisite for the `DMAPool`, accounting, and hostile-smoke sub-gate. Future DDF follow-ups still use these rows as the userspace-driver transition gate: generic MSI/MSI-X dispatch and second-device reuse may land first, but userspace `DeviceMmio` and `Interrupt` exposure stays blocked until these rows pass.

### Production `DMAPool` Ledger Prerequisite

Before userspace NIC or block drivers receive `DeviceMmio`, `Interrupt`, or `DMAPool` handles, the device manager must own one ledger of record for each claimed device. That ledger is the authoritative source for every device-visible hold, not a diagnostic mirror of separate subsystems.

The ledger records:

- DMA pool bytes reserved and live;
- DMA buffer count, slot generation, and owner generation;
- mapped userspace DMA VMAs, quiesce state, scrub state, and release eligibility for each attached DMA pool;
- descriptor and ring depth limits, including live in-flight submissions and completions;
- page-rounded MMIO mappings and their owning `DeviceMmio` generations;
- interrupt holds, waiter generations, and routed-source generations;
- budget and OOM policy for allocation, queue growth, mapping, and interrupt attachment;
- teardown state in the device-owner state machine.

Every operation that creates, consumes, or releases device-visible authority must update this ledger as part of the same ownership transaction that changes device-manager state. That includes DMA buffer allocation/free, descriptor submission, completion accounting, BAR mapping/unmapping, interrupt attach/detach, reset, revoke, process exit, and capability release.

Implementation note, 2026-05-03 13:18 UTC: the QEMU `virtio-rng` metadata path now runs a bounded teardown-trigger proof for `cap-release`, `process-exit`, `driver-crash`, `reset-disable`, `interrupt-waiter`, `future-devicemmio`, and `future-dmapool`. Each trigger row sequentially claims and transfers the same PCI function, begins revocation, walks the existing device-owner state machine to `Dead`, releases only after `Dead`, and proves generation bumps, stale handle rejection, direct state-skip rejection, pre-`Dead` release rejection, and per-trigger coverage without duplicates. The `cap-release` row attaches a bounded manager-owned `DeviceMmio` record to the active driver handle, removes a `DeviceMmioCap` from a cap table, runs the `CapOpRelease` hook, and records cap-table removal plus detached/stale manager validation before normal revocation. The `process-exit` row attaches the same bounded `DeviceMmio` record shape to a real proof `Process`, runs `Process::release_caps_for_exit()`, and records cap-table removal plus detached/stale manager validation before normal revocation. The `driver-crash`, `reset-disable`, and `interrupt-waiter` rows register and claim bounded PCI MSI-X lifecycle-probe routes, attach them to the device manager, prove `InterruptsDetached` is blocked as `interrupts-attached`,

detach and release the routes while still in `MmioRevoked`, and then advance normally. The `future-devicemmio` row attaches a bounded manager-owned `DeviceMmio` record from the first decoded PCI memory BAR, proves `MmioRevoked` is blocked as `devicemmio-attached`, detaches while still in `RevokingHandles`, and then advances normally. The `future-dmapool` row attaches a bounded zero-live `DMAPool` record, proves `DmaMappingsRemoved` is blocked as `dmapool-attached`, detaches while still in `Resetting`, and then advances normally. The generic teardown-trigger summary reports no label-only rows and seven object-backed rows, while the route-aware interrupt handoff smoke also labels the claimed MSI-X route as bounded interrupt-waiter blocker evidence: `interrupt_waiter_object=interrupt-route-record`, `interrupt_waiter_block_state=InterruptsDetached`, `interrupt_waiter_block_result=interrupts-attached`, `interrupt_waiter_detach_result=ok`, and `interrupt_waiter_route_generation_preserved=true`. This bounded route-record evidence is contract proof for the shared ownership transaction only: it does not expose production userspace authority handles, real MMIO, real DMA, a userspace waiter, or production crash/reset observers. Separate first `DeviceMmioCap`, `InterruptCap`, `DmaPoolCap`, and `DmaBufferCap` release-hook proofs now exercise both the production ring `CAP_OP_RELEASE` dispatch path and a real `Process::release_caps_for_exit()` path for those cap objects, validating cap-table removal plus exact manager-owned `DeviceMmio` detach, manager-attached interrupt-route release, bounded zero-live `DMAPool` detach, or proof-owned DMA-buffer record cleanup. The generic route-record trigger rows and remaining DMA production work do not yet implement production observers, production interrupt-waiter objects, userspace `DeviceMmio`, production userspace `DMAPool`/`DMABuffer` authority, full device authority, or true pending hardware MSI/reset-hostile route teardown.

Implementation note, 2026-05-08 10:08 UTC: the first cap-specific reset/disable trigger entry points now exist for `DeviceMmioCap` and `InterruptCap`. `trigger_reset_disable_for_devicemmio` and `trigger_reset_disable_for_interrupt` route through the same idempotent stale-safe detach helpers as cap release and driver-crash cleanup, emit one `cap-audit: ... event=reset-disable detach=ok` line on the first successful trigger, and keep stale reruns silent. This is still bounded trigger plumbing: the reset/disable observer, non-proof DMA cleanup integration, userspace MMIO/interrupt operations, and IOMMU-backed remapping work remain future requirements.

Implementation note, 2026-05-08 10:39 UTC: the DMA caps now have the matching cap-specific reset/disable trigger plumbing. `DmaPoolCap::on_reset_disable` uses the same authoritative zero-live/quiesced/scrubbed evidence-gated detach as cap release and driver-crash cleanup. `DmaBufferCap::on_reset_disable` reuses the bounded `FreeBuffer` authority validation and page-scrub/frame-free cleanup path, then leaves the parent pool attached until staged zero-live cleanup. `make run-net` asserts the `dmabuffer-reset-disable-hook` and `dmapool-reset-disable-hook` proof lines, stale rerun `noop`, revoked cap validation, post-trigger release `noop`, and exact-one `cap-audit: cap={dmabuffer,dmapool} event=reset-disable` lines. This is still proof-owned no-real-DMA cleanup; production userspace `DMAPool`/`DMABuffer` authority and non-proof page lifecycle integration remain future work.

Budget or OOM failure is closed before the driver can observe a new handle, program a descriptor, map MMIO, attach an interrupt, or ring a doorbell. A failed submission must leave no

live descriptor hold behind, or must leave an explicit in-flight record that teardown can drain or reset. A completed teardown must reconcile the ledger to zero live DMA buffers, zero live MMIO mappings, zero interrupt holds, and no in-flight descriptor submissions for the released device generation.

Implementation note, 2026-05-02 06:59 UTC, updated 2026-05-11 06:10 UTC: the current kernel-owned virtio-net ledger now proves the closed budget/OOM cases above with a scratch ledger and the live ledger validation described earlier. Imported live device-manager `DMAPool` records still preserve the `device_dma:virtio-net` source policy and prove imported live accounting stays within its aggregate in-flight budget while preserving that policy's per-queue queue/submission depth limits. The manifest-granted manager-owned bounce-buffer `DMAPool` path now attaches its own device-manager budget policy to userspace `DMAPool.allocateBuffer` handle creation and the current fixed-slot `DMAPool`/`DMABuffer` transfer, release, pending-release, drop, rollback, teardown-detach, page-release, and descriptor-completion cleanup paths. The full eight-slot pool fails as `dmapool-budget-exceeded` / `over-buffer-budget` before allocation, cap minting, or ledger mutation, and the selected release paths revalidate current or next accounting before advancing manager-owned state. Production userspace `DMAPool` records must still attach budget checks to broader provider-driver transfer/revoke/reset transactions, IOMMU or direct-DMA mapping state, and non-fixed-slot allocation before this row can be treated as the complete userspace-driver transition gate.

Implementation note, 2026-05-02 08:33 UTC: the QEMU `virtio-rng` metadata path now runs a bounded `DMAPool` record lifecycle proof on the device-manager teardown state. The first slice keeps the record zero-live: it records a pool slot, pool generation, and owner generation, rejects stale and owner-mismatched attach attempts, rejects duplicate attachment, and proves that `begin_revocation` invalidates the user-visible pool handle by bumping the device owner generation. The ordered teardown path now fails closed with `dmapool-attached` if it tries to enter `DmaMappingsRemoved` while the pool record remains attached. The current continuation also proves that the revoke handle cannot detach the zero-live pool without scratch authoritative zero-live, quiesced, and scrubbed evidence bound to that record's source; a mismatched scratch source is rejected before detach. With matching proof-scoped evidence, the record detaches after queues are quiesced/reset and before `DmaMappingsRemoved`. Later bounded manifest grants expose conservative `DMAPool`, `DeviceMmio`, and `Interrupt` surfaces; the current `DMAPool` grant can mint only eight fixed manager-attached proof `DMABuffer` result caps. The remaining gap is production userspace authority, allocation beyond those eight fixed slots, real device-visible page allocation through the device manager, non-proof DMA page lifecycle integration, IOMMU remapping, and the S.11.2 hostile smoke matrix.

Current QEMU evidence: the QEMU virtio-net path now adds the corresponding imported live-accounting prerequisite proof. A device-manager `DMAPool` record is attached with accounting derived from the live `device_dma` ledger: live buffer/page count, live bytes, current in-flight submissions, committed/resident/unswappable residency flags, and scrub-before-release policy. `DmaMappingsRemoved` fails closed with `dmapool-attached` while the record remains attached, direct teardown detach fails closed with `dmapool-live` while the authoritative ledger remains live, and the live proof consumes the `device_dma` teardown-evidence API, observes `authoritative-ledger-live` with matching imported live accounting, and explicitly defers completion with no real DMA teardown attempted. The same proof path now validates the

imported `DMAPool` record through `capos-lib::device_authority` for the active handle and stale-after-revoke failure labels. This does not create production userspace handles, real page-release hooks, IOMMU mapping invalidation, scrubbed release, terminal `Dead`, or hostile-smoke coverage for the live virtio-net record. The companion scratch-ledger proof covers the positive zero-live teardown-evidence result without claiming that the live virtio-net record has been torn down. The manager-owned zero-live lifecycle proof consumes matching-source device-manager teardown evidence for the positive detach/`DmaMappingsRemoved` path and separately proves mismatched-source and missing-evidence detach attempts fail closed. The manifest-granted bounded `DMAPool` path now keeps mapped userspace VMA count, in-flight descriptor holds, residency, quiesce/scrub state, and release eligibility in that manager record. Borrowed or device-visible pages remain committed, resident, unswappable, generation-bound, and unavailable for reuse until the manager record is zero-live, unmapped, quiesced, and scrubbed. Descriptor submission is refused while a buffer is borrowed to userspace, and release consumes manager-owned teardown evidence instead of proof-only `device_dma` zero-live evidence. The corresponding `DMAPool.info` ABI reports mapped VMA count, quiesce state, scrub state, and release eligibility for QEMU proof assertions. This is still bounded bounce-buffer lifecycle authority only: direct DMA, host physical or IOVA exposure, IOMMU/remapping, production provider-driver consumption, durable audit, and broader transfer/revoke policy remain future work.

- **Gate item:** S.11.2.0 DMA-owned buffers
  - **Required state:** `DMAPool` owns every driver-visible DMA mapping.
  - **Must-have proof:** A driver receives opaque buffer handles or IOVA-only values; no path hands out raw host physical addresses.
- **Gate item:** S.11.2.1 Bound checks
  - **Required state:** Allocation, descriptor chain length, alignment, segment length, and ring depth are bounded and constant-time validated before ring submission.
  - **Must-have proof:** Ring submissions fail closed on overflow, wrap, stale-handle, and freed-handle reuse attempts.
- **Gate item:** S.11.2.2 Explicit remap/ownership
  - **Required state:** `DeviceMmio` can only grant claimed BAR pages; cache attributes and write policy are enforced.
  - **Must-have proof:** Driver cannot access unclaimed BARs, ROM, RAM pages, config-space globals, or stale mappings after revoke.
- **Gate item:** S.11.2.3 Interrupt correctness
  - **Required state:** `Interrupt` owns exactly one logical source at a time and drains/waits only for that source.
  - **Must-have proof:** Reassigning an owner invalidates old waiters and masks or detaches the source first.
- **Gate item:** S.11.2.4 Quiesce + reset contract
  - **Required state:** Device manager can force reset/disable on failed revoke or teardown.
  - **Must-have proof:** No in-flight descriptor may continue touching freed buffers after driver removal.

- **Gate item:** S.11.2.5 Process lifecycle
  - **Required state:** Capability release, process exit, and process-spawn cleanup paths cannot leak DMA pages/MMIO/intr ownership.
  - **Must-have proof:** Crash-path teardown removes holds and invalidates user-visible handles before page free.
- **Gate item:** S.11.2.6 Isolation and accounting
  - **Required state:** Security Verification Track S.9 quota and authority ledgers include DMA, MMIO, and interrupt hold edges.
  - **Must-have proof:** A malicious or buggy driver cannot consume more than its allocated authority budget.
- **Gate item:** S.11.2.7 Stale IRQ ordering
  - **Required state:** Stale interrupt delivery after revoke cannot wake, acknowledge, or signal a new owner.
  - **Must-have proof:** Interrupt generation mismatch is ignored, or the source is masked/detached/reset before reassignment. Hostile smoke revokes a driver while an interrupt is pending, reassigns the source, and proves the old waiter cannot wake against the new owner. **Closed `2026-05-05 18:17 UTC`** by `make run-net`'s `device-manager: interrupt handoff proof` line: real `INT $vector` injection across revoke, detach, and reset/reuse exercises the production IDT entry/handler/EOI path, asserts `s11_2_7_real_irq_injected_across_reset=ok`, `s11_2_7_old_waiter_cannot_wake_new_owner=true`, and `s11_2_7_stale_ack_blocked=true`, and is enforced by `tools/qemu-net-smoke.sh`. Userspace `Interrupt` waiter objects remain a future requirement for a full production-driver path.
- **Gate item:** S.11.2.8 Stale DMA completion ordering
  - **Required state:** Stale DMA completion after revoke cannot cause freed buffer reuse, stale CQ notification, or new-owner memory exposure.
  - **Must-have proof: Closed `2026-05-05 19:37 UTC`** by `make run-net`'s `device-manager: dma completion handoff proof` line: real virtio-net DMA page free + reallocate cycle bumps the live page generation, then the production `device_dma::record_virtio_net_completion_for_allocation` path (the same function the live `Virtqueue::record_used_completion_for_allocation` invokes) is fed a stale `DeviceDmaAllocation` keyed to the live phys with a decremented generation, at three boundaries (after revoke, after detach, after reset/reuse). All three reject as `stale-dma-handle` with `side-effect-blocked`, queue accounting unchanged, live new-owner page preserved, no CQ publication, no new-owner exposure, and the freed-buffer slot remaining unchanged. The closure summary asserts `s11_2_8_real_completion_injected_across_reset=ok`, `s11_2_8_old_completion_cannot_publish_to_new_owner=true`, `s11_2_8_freed_buffer_reuse_blocked=true`, and `s11_2_8_accounting_underflow_blocked=true`, and is enforced by `tools/qemu-net-smoke.sh`. Prior acceptance text: in-flight DMA is accounted for, or device reset/disable completes before buffer reuse; hostile smoke covers revoke/reset with outstanding descriptors and proves no old completion can publish new-owner memory. S.11.2.9 hostile-smoke gate-wiring also closed `2026-05-05 20:49 UTC` (see the row below). Userspace

`DMAPool` handles and real device-manager page quiesce/scrub/release hooks remain open as separate follow-ups.

- **Gate item:** S.11.2.9 Hostile-smoke coverage
  - **Required state:** QEMU/CI smokes cover stale handles, descriptor abuse, revoke races, stale IRQ after reset, stale DMA completion after reset, and exit-under-dma.
  - **Must-have proof:** Smoke output has explicit closed-case proof lines for each above failure mode. **Closed `2026-05-05 20:49 UTC`** by aggregating the existing per-row proof lines into the `make run-net` -> `tools/qemu-net-smoke.sh` gate. Every matrix-row proof line has at least one assertion in the harness; the original two driver-crash assertions, the existing S.11.2.8 `device-manager: dma completion handoff proof` closure-summary assertion, and the S.11.2.7 `device-manager: interrupt handoff proof` closure-summary assertion (whose trailing anchor was added by this slice for harness-strictness consistency with S.11.2.8) all use the anchored extended-regex shape (field-by-field match plus `proof_result=ok[[:cntrl:]]?$` trailing anchor), and the other matrix-row rows reuse the harness's pre-existing mix of unanchored extended-regex and fixed-string `grep -Fq` assertions. A `2026-05-08 09:44 UTC` follow-up adds anchored assertions for the cap-specific `dmabuffer-driver-crash-hook` and `dmapool-driver-crash-hook` proof lines; a `2026-05-08 10:08 UTC` follow-up adds anchored assertions and exact-one audit counts for the first cap-specific `devicemmio-reset-disable-hook` and `interrupt-reset-disable-hook` proof lines; a `2026-05-08 10:39 UTC` follow-up does the same for `dmabuffer-reset-disable-hook` and `dmapool-reset-disable-hook`; a `2026-05-08 13:42 UTC` follow-up (`aeef8b41`) adds the cap-specific `device-manager: interrupt waiter hook proof source=interrupt-waiter-hook ... trigger_path=trigger-interrupt-waiter-for-interrupt` assertion plus an exact-one `cap-audit: cap=interrupt event=interrupt-waiter` count. Per-row coverage: stale DMA handle (`device-dma: stale dma handle proof`, `device-dma: live stale dma completion accounting proof`); descriptor abuse (`virtio-net: software descriptor generation model proof`, `virtio-net: invalid used descriptor id software-token proof`, `virtio-net: descriptor generation guard proof ok`, `virtio-net: invalid used descriptor id live software-token proof ok`, plus `device-dma: budget oom proof`); revoke/reset race (`device-manager: ownership proof`, the seven `device-manager: teardown trigger proof trigger=...` variants plus the final aggregate, `device-manager: dma completion handoff proof` for S.11.2.8, `device-manager: interrupt handoff proof` for S.11.2.7, the `device-manager: devicemmio driver crash hook proof source=devicemmio-driver-crash-hook ... trigger_path=trigger-driver-crash-for-devicemmio`, `device-manager: interrupt driver crash hook proof source=interrupt-driver-crash-hook ... trigger_path=trigger-driver-crash-for-interrupt`, `device-manager: dmabuffer driver crash hook proof source=dmabuffer-driver-crash-hook ... trigger_path=trigger-driver-crash-for-dmabuffer`, `device-manager: dmapool driver crash hook proof source=dmapool-driver-crash-hook ... trigger_path=trigger-driver-crash-for-dmapool`, `device-manager: devicemmio reset disable hook proof source=devicemmio-reset-disable-hook ... trigger_path=trigger-reset-disable-for-devicemmio`, `device-manager: interrupt reset disable hook proof source=interrupt-reset-disable-hook ... trigger_path=trigger-reset-disable-for-interrupt`, `device-manager: dmabuffer reset disable hook proof source=dmabuffer-reset-disable-hook ...`

trigger_path=trigger-reset-disable-for-dmabuffer, device-manager: dmapool reset disable hook proof source=dmapool-reset-disable-hook ... trigger_path=trigger-reset-disable-for-dmapool, and device-manager: interrupt waiter hook proof source=interrupt-waiter-hook ... trigger_path=trigger-interrupt-waiter-for-interrupt lines, all requiring first-trigger ok, stale rerun noop, cap validate_live=revoked, post-trigger release noop, and proof_result=ok with cap-specific cleanup/evidence labels); stale IRQ after reset (S.11.2.7 closure summary, see row above); stale DMA completion after reset (S.11.2.8 closure summary, see row above); exit-under-DMA (device-manager: teardown trigger proof trigger=process-exit owner=virtio-rng, the teardown-trigger aggregate triggers=cap-release,process-exit,driver-crash,reset-disable,interrupt-waiter,future-devicemmio,future-dmapool line, the four cap-release-hook proofs each containing process_exit_path=process-release-caps-for-exit, plus hardware-cap-release: ... reason=process-exit count assertions). A 2026-05-23 21:34 UTC follow-up adds the IOMMU production DMAPool hostile proof over the active mapped ledger, covering stale IOVA after revoke/reset, descriptor abuse, revoke/reset race ordering, stale completion after reset, teardown-under-DMA ordering, cross-domain stale-handle attempts, and the fail-closed teardown branch proof; process-exit/exit-under-DMA remains the existing run-net bounce-buffer evidence. Production userspace DeviceMmio/Interrupt handles, broader non-proof device-manager page quiesce/scrub/release hooks outside the selected IOMMU smoke, hardware-backed provider-driver Interrupt wait/ack dispatch beyond the bounded route-dispatch waiter proof, and durable/signed production audit consumption beyond the first volatile HardwareAuditLog.snapshot cap remain open as separate follow-ups.

For each row, the transition requires an owner, implementation notes, and a CI-backed verification path. Until all rows pass, Phase 4.2 NIC/block drivers remain in-kernel for functionality, and only kernel-mapped bounce-buffer mode is allowed for prototype DMA.

**Hostile-Smoke Acceptance Matrix**

These smokes are the acceptance requirements for the userspace driver transition. The S.11.2.7, S.11.2.8, and S.11.2.9 rows are now backed by current make run-net QEMU evidence enforced by tools/qemu-net-smoke.sh (see the per-row "Closed" notes for closure timestamps and the proof-line shapes). The other matrix rows remain acceptance requirements for future implementation work; their proof lines are emitted by the kernel today and asserted by the same harness, but the production userspace handles, real device-manager page quiesce/scrub/release hooks, real userspace Interrupt waiter objects, IOMMU domain programming, and durable/signed production audit consumption beyond the volatile HardwareAuditLog.snapshot cap that complete each row's full closure remain open as separate follow-ups.

- **Hostile case:** Stale DMA handle
  - **Required setup:** Allocate a DMA buffer, revoke or free it, advance the slot or pool generation, then attempt descriptor submission or buffer reuse through the old handle.
  - **Closed-case proof expectation:** The operation fails closed on generation mismatch; no descriptor is made visible to the device, no DMA byte or buffer hold is restored, and any reused slot remains owned only by the new generation.
- **Hostile case:** Descriptor abuse

- **Required setup:** Submit chains with out-of-pool addresses, stale or freed buffer slots, arithmetic wrap, misalignment, overlong segments, excessive chain length, or ring-depth overflow.
- **Closed-case proof expectation:** Validation rejects the chain before any doorbell write; the ledger shows no leaked descriptor hold, no in-flight increment without an owning buffer, and no access outside the pool range.

- **Hostile case:** Revoke/reset race
  - **Required setup:** Race revoke, reset, or process teardown against a driver that is submitting descriptors or ringing the device doorbell.
  - **Closed-case proof expectation:** Revocation first invalidates handles and MMIO write authority; later submissions fail closed, existing in-flight records are either completed under the old generation or reset/disabled before page reuse, and teardown cannot skip to `DmaMappingsRemoved` while the ledger has live submissions.
- **Hostile case:** Stale IRQ after reset
  - **Required setup:** Leave an interrupt pending or a waiter blocked, reset or reassign the device/source, then deliver or acknowledge using the old generation.
  - **Closed-case proof expectation:** The old waiter cannot wake against the new owner, stale acknowledgements do not affect the reassigned source, and the source is masked, detached, or generation-invalidated before reassignment. **Closed `2026-05-05 18:17 UTC`**: `make run-net` injects a real `INT $vector` through the IDT/handler/EOI path at three points across revoke, detach, and reset/reuse and records `s11_2_7_real_irq_injected_across_reset=ok`, `s11_2_7_old_waiter_cannot_wake_new_owner=true`, `s11_2_7_stale_ack_blocked=true`, plus matching `real_irq_inject_after_revoke_result=masked`, `real_irq_inject_after_detach_result=unregistered`, `real_irq_inject_after_reset_reuse_result=masked` on the kernel proof line.
- **Hostile case:** Stale DMA completion after reset
  - **Required setup:** Reset with outstanding descriptors, reuse or prepare to reuse pool slots, then inject or observe a completion from the old device generation.
  - **Closed-case proof expectation:** The stale completion cannot publish a CQE to a new owner, cannot expose new-owner memory, cannot underflow accounting, and cannot make a freed buffer eligible for reuse unless reset/disable has proven old DMA stopped. **Closed `2026-05-05 19:37 UTC`**: `make run-net` walks a fresh device-manager record on the virtio-net BDF through the `Active>RevokingHandles>MmioRevoked>InterruptsDetached>QueuesQuiesced>Resetting>DmaMappingsRemove` revocation path, exercises a real virtio-net DMA page free + reallocate cycle at three boundaries (after revoke, after detach, after reset/reuse), and feeds a synthesized stale `DeviceDmaAllocation` (live phys, decremented generation) to the production `device_dma::record_virtio_net_completion_for_allocation` path. Each boundary records `real_completion_inject_after_*_result=stale-dma-handle`, `_side_effect=side-effect-blocked`, `_queue_account_preserved=true`, `_live_page_preserved=true`, `_cq_publication_blocked=true`, `_new_owner_exposure_blocked=true`, `_freed_buffer_unchanged=true`, and `_generation_bumped=true`, plus a closure summary

`s11_2_8_real_completion_injected_across_reset=ok`, `s11_2_8_old_completion_cannot_publish_to_new_owner=true`, `s11_2_8_freed_buffer_reuse_blocked=true`, `s11_2_8_accounting_underflow_blocked=true`.

- **Hostile case:** Exit-under-DMA
  - **Required setup:** Terminate or crash a driver process while it holds DMA buffers, MMIO mappings, interrupt waiters, and in-flight descriptors.
  - **Closed-case proof expectation:** Process exit enters the device-manager teardown path, invalidates all user-visible handles, revokes MMIO, detaches interrupts, quiesces or resets queues, scrubs DMA pages before release, and reports a terminal ledger with no live holds for the old owner generation.

#### Security Verification Track S.11.2 Decision Record

Security Verification Track S.11.2 is backend-scoped. The current brokered-bounce userspace-provider path has enough reviewed evidence to close the retained DDF production-authority finding, but that closeout is not a general direct-DMA, hostile-hardware, or device-autonomous interrupt claim.

Current status: the brokered-bounce transition path is represented by done task evidence for `DMAPool`, `DeviceMmio`, and `Interrupt` lifecycle ownership, provider virtio-net/NVMe chains, and hardware-audit consumption of abort-held DMA mappings. The broader S.11.2 matrix remains the canonical gate for future direct-remapping/vIOMMU, trusted-sharing-group, hostile-hardware-isolation, or provider-written-address work. This document fixes the production handle epoch invariants, `DMAPool` ledger of record, and hostile-smoke acceptance criteria used by the completed Device Driver Foundation documentation gate. The current QEMU virtio-net path has a kernel-owned DMA pool ledger for page, descriptor, MMIO mapping, and interrupt-hold accounting proof coverage plus static IOMMU attachment-policy reporting for retained DMA-capable PCI functions and the bounded teardown trigger contract proof, bounded kernel-owned budget/OOM proof, manager-bound `DMAPool` budget-profile proof plus bounded budget-policy tamper and accounting-over-budget fail-closed proofs, bounded manager-owned `DeviceMmio` proof adapter bound to decoded PCI memory-BAR metadata plus future cache/write-policy metadata, bounded zero-live device-manager `DMAPool` record lifecycle proof, and imported live-accounting block/defer proof plus zero-live teardown-evidence scratch proof, stale DMA handle scratch proof, stale DMA completion scratch proof, paired scratch CQ-publication/new-owner-exposure proof, live software descriptor-generation guard proof, bounded invalid used-descriptor-id proof, and bounded stale IRQ after-detach, counter-backed after-revoke, counter-backed route-registry reset-reuse, and pending IRQ token checks described above. The bounded pure `capos-lib::device_authority` validator and host tests cover the documented identity, state, side-effect-blocking, non-wrapping epoch cases, and every current operation variant's exact blocked side-effect label for stale owner/subrecord, freed, revoked, and retired failures. The zero-live device-manager `DMAPool` lifecycle proof now validates a proof-scoped tampered budget-policy record through the manager policy helper and records fail-closed, no fake allocation, no ledger mutation, no teardown advancement, and side-effect blocking while preserving the positive `budget_policy_result=ok` path. The positive zero-live and imported-live budget-accounting labels now go through the manager-owned active-record helper, and synthetic over-

budget attached-accounting candidates fail closed with exact reasons while preserving the active manager record and blocking allocation, ledger, teardown, and side effects; an over-budget attach candidate fails before pool generation allocation. It also records a bounded manager-attached DMA buffer handle under the attached pool, validates active `SubmitDescriptor` and manager-record `CompleteDescriptor` through the pure DMA-buffer validator, and records stale-after-revoke, freed-buffer, and reused-slot rejection with exact reasons and `side-effect-blocked`; it now also blocks pool teardown as `dmapool-buffer-attached`, rejects a stale same-slot proof-scoped `FreeBuffer` as `dmabuffer-stale-handle` with `stale-slot-generation` and `side-effect-blocked`, rejects wrong-owner-generation, wrong-pool, wrong-pool generation, and wrong-buffer-slot `FreeBuffer` attempts with exact pure validator reasons and `side-effect-blocked`, preserves that manager-owned buffer record after each failed free, and clears the record only after a proof-scoped active `FreeBuffer` validation, proof-page scrub/free, and manager-owned buffer-record detach. The completion proof does not publish a CQ entry, complete a real descriptor, grant userspace authority, or clean up or reuse production userspace DMA pages. The live virtio-net queue-completion path now gates completion accounting on the completed descriptor's `DeviceDmaAllocation` rather than the queue id alone: callers must validate the used descriptor id, recover the matching `DmaPage`, and pass its physical address, queue, label, and generation to the kernel-owned ledger before in-flight accounting is decremented. The paired `run-net` proof records that a stale generation for a live kernel-owned page fails as `stale-dma-handle`, leaves queue accounting and the live page unchanged, and blocks CQ publication plus new-owner exposure. This closes a live accounting prerequisite only; it does not inject a real post-reset device completion or expose userspace DMA authority. The live virtio-net used-ring path also carries bounded software descriptor generations: submissions reject invalid or already-active descriptor ids before accounting, completions must consume the active software token exactly once, and the `run-net` proof records side-effect blocking for active reuse, double completion, and an old software token after descriptor-id reuse. That guard does not make a stale hardware used-ring id distinguishable after deliberate id reuse because virtio used entries carry no device generation. The same gate now also covers invalid used-descriptor ids without corrupting the hardware ring: an out-of-range id fails as `descriptor-id-out-of-range` before completion observation, completion accounting, `used_seen_idx`, CQ publication, or new-owner exposure can change. This is still a software-token and constructed-token prerequisite, not a real malformed-device or post-reset completion injection. The same zero-live proof now also constructs the result-only `DMAPool.info` cap skeleton from the manager-issued `DmaPoolHandle`, validates the active manager record before returning conservative status labels plus numeric device/BDF/owner/pool identity fields, proves the serialized cap call path decodes to those labels and identity fields with host physical exposure off and direct DMA blocked, and proves the cap's info path fails closed as `dmapool-stale-handle` after revoke begins. It also exercises `DMAPool.allocateBuffer` through `call_with_table()` on a real cap-table entry, returns zero-indexed `DMABuffer` result caps for eight fixed manager-owned bounce-buffer slots, validates those result caps' `DMABuffer.info`, and proves a ninth allocation fails through the manager-owned budget policy as `dmapool-budget-exceeded` / `over-buffer-budget` before publishing another result cap or corrupting live slot state; full-pool allocation also preserves manager generation counters. Stale-after-revoke allocations still fail closed without publishing another result cap. The same zero-live proof constructs the result-only `DMABuffer.info` cap skeleton from the manager-attached `DmaBufferHandle`, validates the active manager-owned buffer record through the pure

DMA-buffer validator before returning conservative no-authority labels plus numeric device/BDF, owner/pool/slot identity fields, proves the serialized cap call path decodes to those labels and identity fields with host physical exposure off and direct DMA blocked, and proves the cap's info path fails closed as `dmabuffer-stale-handle` after revoke begins; the same stale cap's serialized method-0 path fails as `invoke-failed`. The first `DmaBufferCap` release hook now reuses the bounded `FreeBuffer` validation shape to clear only the manager-attached `proof_buffer` record during cap-table removal, production ring `CAP_OP_RELEASE`, and real `Process::release_caps_for_exit()` paths. It proves stale same-slot release is `side-effect-blocked`, proves the parent `DMAPool` remains attached after buffer release, proves the bounded manifest grant can allocate the slot again after explicit `freeBuffer` with a fresh slot generation, and still requires staged zero-live evidence before the parent pool can detach. The selected provider-TX path now adds a bounded exception to the default manager-accounting descriptor contract: queue `1` submits may publish the selected eight-entry TX queue depth, descriptors `0..7`, into the existing kernel-owned virtio-net TX ring before the first completion, ring one selected notify doorbell per accepted provider descriptor, and then complete each descriptor only after `DMABuffer.completeDescriptor` observes the matching used-ring entry for the stored software descriptor generation. Those handoffs clear the matching manager in-flight records, record bounded provider CQ completion and acknowledgement counts, and can deliver ordered bounded completion events to live `tx_interrupt.wait` calls for the same selected route. The selected provider-TX path also proves a teardown-only drain when one descriptor has completed and seven provider-published descriptors remain incomplete: direct `DMABuffer.freeBuffer` remains blocked while in flight, release explicitly drains only the incomplete matching used-ring entries and retires those allocation-backed TX DMA queue ledgers without `DMABuffer.completeDescriptor` results, no provider CQ/IRQ event is published for the quiesced descriptors, release retires seven delivered-but-unacked completion events, and later slot reuse requires a fresh generation plus normal completion. Wrong-queue, stale-buffer, stale-notify, inflight-publication, wrong-descriptor, duplicate-completion, and stale-`tx_interrupt` issue paths remain side-effect-blocked before their guarded effects. This does not grant direct DMA, arbitrary doorbells, arbitrary CQ ownership outside the selected TX route, full virtio-net ownership, production NIC/storage migration, IOMMU programming, hardware IRQ ownership, hardware acknowledgement, or broad interrupt ownership beyond the bounded selected TX MSI-X mask/unmask proof. The bounded `DeviceMmio` proof also records the manager-attached policy metadata listed above, fails closed on a tampered cache/write-policy record before creating any mapping, and validates active hostile handle identities for wrong owner generation, wrong mapping generation, wrong mapping id, wrong BAR, and wrong BDF/device with exact pure-validator reasons while preserving the attached record and blocking mapping/doorbell side effects. Its serialized cap call path also decodes to the direct `DeviceMmio.info` no-authority labels plus numeric device/BDF, owner, BAR, mapping id, and mapping generation identity fields with host physical exposure off and direct MMIO blocked, and its stale serialized method-0 path fails as `invoke-failed`. The `DMAPool.info` skeleton has the same kernel-side serialized stale failure evidence. The interrupt handoff proof now also constructs a result-only `Interrupt.info` cap skeleton from the manager-issued device handle and attached route record, records active info success, proves the serialized cap call path decodes to the direct no-authority labels plus numeric device/BDF, owner, source, source generation, and route generation identity fields, proves those source and route generations are distinct in the bounded route record, and proves stale-after-

revoke info fails closed as `interrupt-stale-handle` plus stale serialized method-0 failure as `invoke-failed` before any acknowledgement, mask, unmask, blocking wait, or delivery authority exists. The manifest-granted skeleton now also exposes an admission-only `Interrupt.wait` method that returns the pending-token validator's fail-closed labels without waking a waiter or changing delivery counts, and an admission-only `Interrupt.acknowledge` method that validates the active route while blocking hardware acknowledgement and preserving delivery counts. It also exposes route-state-control `Interrupt.mask` and `Interrupt.unmask` methods that validate the active route before changing the manager-attached dispatch slot between `claimed-masked` and `driver-unmasked`, while preserving delivery counts. A bounded `Interrupt.wait` call observed after unmask installs a fixed-table userspace waiter object for the current manager-granted route; the existing route-dispatch delivery counter can now complete that waiter as `interrupt-delivered` / `waiter-completed-irq` with `real_interrupt_delivery=delivered` and an advanced delivery count. The same focused smoke then submits a second unmasked wait, observes it remains pending, calls `Interrupt.mask`, and finishes that wait as `interrupt-waiter-cancelled` / `route-masked` / `waiter-completed-no-irq` with `wake_blocked=false`, preserved source/route generations, and unchanged delivery counts. The selected provider TX `tx_interrupt` cap can now observe the bounded used-ring completion event described above and account the already observed selected TX dispatch token paired with that delivered provider CQ event, but hardware MSI/MSI-X programming beyond the selected vector-control proof, full hardware IRQ ownership, deferred EOI, LAPIC/MSI-X acknowledgement, and broader production interrupt dispatch remain blocked. Provider TX MSI-X mask/unmask is limited to the selected-route vector-control proof described earlier. Provider RX MSI-X mask/unmask remains bounded to the selected RX route as well; release while masked restores that selected vector-control bit and route state before clearing the live issue gate. RX unmask admits the route transition before exposing the MSI-X vector-control bit, and the focused QEMU proof shows a failed route unmask leaves the selected vector masked with the route ledger preserved. Cleanup failure still leaves the issue uncleared so future RX cap issuance stays blocked on uncertain route state. RX wait/ack is now bounded to one selected-route zero-CQ dispatch token; RX descriptors and CQ ownership remain blocked. This is manager-record skeleton/no-production-DMA, no-real-MMIO-mapping, and bounded route-dispatch interrupt-waiter prerequisite evidence only. Production `DMAPool`, `DeviceMmio`, and `Interrupt` capability handles, production userspace `DMAPool` buffer handles, real `DeviceMmio` BAR mapping objects, real cache attributes/write policy enforcement, production kernel device-path wiring beyond the current proof adapters, real device-manager page quiesce/scrub/release hooks and real page cleanup/reuse beyond the bounded kernel-owned proof pages, production handle-attached budget/OOM enforcement beyond the current manager-owned `DMAPool.allocateBuffer` budget slice, IOMMU remapping domains, production handle-attached host tests, QEMU stale-handle smokes, broader userspace exposure, production NIC/storage migration, cloud readiness, and S.11.2 hostile smokes remain open.

Do not weaken the short-term virtio-net bounce-buffer path until `DMAPool`, `DeviceMmio`, `Interrupt`, device-manager ownership transactions, lifecycle teardown, accounting, and hostile smokes all exist.

# Appendix: Risk Register

Known design risks and follow-up areas that frame future review work.

## Design Risks and Open Questions Register

Consolidated index of known design risks and open architectural questions for capOS. Every entry routes to the file that owns the long-form design or the remediation backlog for that risk; this register itself is a pointer document, not a place to put new design.

Use this document to answer "is this risk already tracked, and where?" without re-deriving the state from the proposal tree on each review.

Last refresh: 2026-06-07 08:02 UTC.

### How To Use

- Each design-risk row records the **current observable state** (what the code and docs say today), the **owning tracker** (the proposal/backlog/design file to update when the state changes), and the **remaining gap** (what is still open).
- Each open-question row records a **current answer** if one exists in the tree, plus a **pointer** to the canonical tracker. Questions that are genuinely unanswered are marked **Open**; those should not be closed by guessing here – update the relevant proposal, then update this register.
- When a risk is closed by code or by an explicit design decision, move the short closure summary into `docs/changelog.md` and remove the row.
- New review findings go into task records under `docs/tasks/`; this register is about long-horizon design risks, not concrete unresolved review issues.

### Design Risks

#### R1 – Process-wide ring vs multi-threaded userspace and full SMP

- **State.** The capability ring is one per process. `capos-rt` enforces a single-owner `RuntimeRingClient`. After in-process threading, at most one process-ring waiter is allowed. The first SMP Phase C AP scheduler-owner proof deliberately keeps process-wide ring execution on a single CPU at a time behind a scheduler-owner latch.
- **Owner.** `docs/proposals/ring-v2-smp-proposal.md`, `docs/research/completion-ring-threading.md`, `docs/backlog/smp-phase-c.md`, `docs/architecture/threading.md`.
- **Gap.** Per-thread capability rings, per-thread completion routing, and the Multi-Process / In-Process Threading Scalability milestones in `docs/roadmap.md` remain future work. Userspace threading scales only as far as the single ring waiter allows.

#### R2 – "Interface IS the permission" pushes safety into wrapper TCB

- **State.** capOS deliberately has no parallel rights bitmask: attenuation is done by handing out a narrower `CapObject` wrapper, not a flag-reduced copy of the same cap. Wrapper correctness is therefore part of the trust base.
- **Owner.** `docs/capability-model.md`, `docs/proposals/session-bound-invocation-context-proposal.md`, `docs/security/trust-boundaries.md`, `docs/backlog/stage-6-capability-semantics.md`.
- **Gap.** The completed Session-Bound Invocation Context migration has the one-session-per-process proof, privacy-preserving endpoint caller-session metadata, explicit subject-disclosure coverage, chat session-keyed state, Adventure service grants, terminal/stdio bridge liveness

guards, and final Gate 4 verification. The first Tier-1 paper claim, covering session-bound invocation context evidence for implementation review, is closed. Remaining non-gating cleanup is stable service-audit identity across service replacement and legacy internal receiver-selector naming.

**R3 – Legacy endpoint metadata as transitional service identity**

- **State.** Legacy endpoint receiver metadata is contained as internal transport/debug state for normal paths. Chat uses session-keyed membership, terminal/stdio bridges enforce live caller-session guards, and delegated relabeling containment plus the historical service-object routing/lifecycle proof have landed. Adventure/shared-service cleanup is landed for normal workload paths.
- **Owner.** `docs/proposals/session-bound-invocation-context-proposal.md`, `docs/backlog/stage-6-capability-semantics.md`.
- **Gap.** Finish final legacy cleanup. Receiver metadata must remain internal transport state or hostile-test fixture, not subject identity or disclosure.

**R4 – Resource accounting is fragmented**

- **State.** Per-process memory, cap-table, and thread quotas exist; `ResourceProfile`, session quotas, scheduling-context donation, and cross-service donation/fairness are still proposal-shaped.
- **Owner.** `docs/proposals/resource-accounting-proposal.md`, `docs/proposals/memory-authority-model-proposal.md`, `docs/proposals/oom-and-swap-proposal.md`, `docs/proposals/user-identity-and-policy-proposal.md`, `docs/proposals/system-monitoring-proposal.md`, `docs/proposals/scheduler-evolution-proposal.md`, `docs/backlog/scheduler-evolution.md`.
- **Gap.** Phase D WFQ has landed; Phase E `SchedulingContext` bind/revoke, budget, donation/return, and depletion notification are closed at the scheduler-cap layer, but cross-service donation semantics, per-service fairness beyond thread weights, log volume accounting, memory authority/residency proof obligations, unified resource bundles for guest/anonymous/external/service principals, and the scratch-bytes / outstanding-calls / endpoint-queue / in-flight-call quota fields tracked in review-finding task records remain open.

**R5 – Copy-transfer SQE replay is repeatable by design**

- **State.** `docs/authority-accounting-transfer-design.md` documents that userspace replay of a copy-transfer SQE is repeatable per dispatch attempt, with move-transfer replay failing closed once the source slot is removed/reserved. Exactly-once replay suppression is explicitly future work (security invariant T3).
- **Owner.** `docs/authority-accounting-transfer-design.md`, `docs/proposals/security-and-verification-proposal.md`.
- **Gap.** The `(sender_pid, call_id, sqe_seq)` plus monotonic transfer-epoch identity needed for exactly-once replay across dispatch attempts is not implemented. Each transferable interface must continue to acknowledge this in its threat model.

#### R6 – `CAP_OP_RELEASE` is deferred / queued, not synchronous {#design-risks-register-r6-cap_op_release-is-deferred-queued-not-synchronous}

- **State.** Owned-handle drop in `capos-rt` queues one local `CAP_OP_RELEASE` on the ring; process exit performs fallback cleanup. Release does not run before the next ring flush (`cap_enter` or process exit).
- **Owner.** `docs/authority-accounting-transfer-design.md`, `docs/proposals/error-handling-proposal.md`, `docs/capability-model.md`.
- **Gap.** Resource-pressure or revocation-sensitive flows must not assume a Drop call has already taken effect at the kernel layer. Time-critical revocation should use `CapabilityManager.revoke` or epoch revocation rather than relying on Drop.

#### R7 – Shared memory / zero-copy / shared park are incomplete

- **State.** `MemoryObject` substrate exists; `SharedBuffer` provenance, file/network/DMA zero-copy paths, and shared park/`SharedParkSpace` are blocked on mapping provenance / object pinning work.
- **Owner.** `docs/proposals/storage-and-naming-proposal.md`, `docs/proposals/memory-authority-model-proposal.md`, `docs/proposals/networking-proposal.md`, `docs/architecture/park.md`, `docs/backlog/runtime-network-shell.md`.
- **Gap.** Workloads that need true zero-copy IPC, storage, or network pipelines pay a copy/serialization cost until provenance/pinning lands. ParkSpace private cleanup now covers anonymous `VirtualMemory.unmap`, `VirtualMemory.decommit`, and explicit `MemoryObject.unmap` for borrowed mappings; shared park keys and address-space generation cleanup remain open.

#### R8 – Networking lives inside the kernel TCB

- **State.** Largely resolved: the Phase C userspace NIC driver and smoltcp network-stack process own the production socket path, the kernel no longer depends on `smoltcp`, and the kernel socket `CapObjects` are qemu-only fixtures that fail closed without a kernel socket owner. The Telnet and SSH terminal-host proofs that sat on the kernel path are retired.
- **Owner.** `docs/proposals/networking-proposal.md`, `docs/dma-isolation-design.md`, `docs/backlog/runtime-network-shell.md`.
- **Gap.** The remaining qemu-only kernel virtio-net fixture and socket `CapObject` surface is fixture code, not production authority. The kernel-side `SocketTerminalSession` transitional shim is retired (2026-06-10): `TcpSocket.intoTerminalSession` fails closed, and a network-backed `TerminalSession` must be re-built as a userspace terminal-session service over the userspace TCP stack if byte-stream terminal transport is needed again.

#### R9 – DMA isolation is backend-scoped, not a hostile-hardware blanket

- **State.** `docs/dma-isolation-design.md` now records runtime fail-closed DMA backend selection. The current no-IOMMU cloud/DDF path uses manager-owned, brokered bounce buffers for userspace provider authority and hides host physical addresses and IOVAs from the driver. The selected QEMU Intel VT-d path has bounded per-device remapping evidence, but that remains emulator evidence rather than a general hardware-isolation claim. Without trusted remapping, hostile bus-mastering hardware remains out of scope.

- **Owner.** `docs/dma-isolation-design.md`, `docs/proposals/networking-proposal.md`, `docs/proposals/cloud-deployment-proposal.md`, `docs/backlog/hardware-boot-storage.md`.
- **Gap.** The retained DDF production-authority finding is closed in `docs/tasks/done/2026-06-07/ddf-production-authority-closeout.md`. Remaining work is explicit task or proposal scope: direct-remapping/vIOMMU production hardware support, broader provider/device variants, and device-autonomous MSI-X delivery rather than the current polled or kernel-injected waiter proofs.

### R10 – Boot package model embeds all binaries

- **State.** `tools/mkmanifest` embeds every declared binary as a `NamedBlob` inside `manifest.bin`. The kernel loads only `init`; everything else is fetched by `init` from the in-memory `BootPackage`.
- **Owner.** `docs/backlog/hardware-boot-storage.md`, `docs/proposals/storage-and-naming-proposal.md`, `docs/trusted-build-inputs.md`.
- **Gap.** Boot binary ISO layout (separate ELF payloads), package/storage update model, and persistent storage-backed delivery are not yet designed as code; the current scheme is an explicit prototype compromise.

### R11 – Pre-auth and post-auth share a shell process

- **State.** The shell-led boot flow folds `console-login` into `capos-shell` and uses an anonymous-first session that escalates via `login/setup`. The pre-auth and post-auth code paths run in one userspace process and address space.
- **Owner.** `docs/proposals/boot-to-shell-proposal.md`, `docs/proposals/shell-proposal.md`, `docs/security/trust-boundaries.md`, `docs/proposals/user-identity-and-policy-proposal.md`.
- **Gap.** Separation depends on shell/auth implementation quality, not on a process boundary. The future direction (separate login service with minimal authority, restricted launchers, WebShell/SshGateway) is proposal-shaped. Remote and non-loopback shells must remain blocked until pre-auth and post-auth authority are process-isolated or a shared-process proof is accepted.

### R16 – Remote shell ingress is demo/prototype only

- **State.** Telnet is a plaintext loopback-only QEMU demo. SSH has SSH-shaped prerequisites, fixture authentication proofs, dev key material, policy classification, and restricted-shell launcher coverage, but no production encrypted SSH transport, durable key/account storage, full OpenSSH-compatible userauth/channel handling, channel binding, or complete audit/storage gates.
- **Owner.** `docs/proposals/ssh-shell-proposal.md`, `docs/proposals/telnet-tls-shell-proposal.md`, `docs/backlog/runtime-network-shell.md`, `docs/tasks/README.md`, `docs/build-run-test.md`.
- **Gap.** Production/non-loopback shell exposure is blocked on SSH transport, key, account, audit, storage, session-bound delegation, and pre-auth/post-auth isolation gates.

### R17 – Remote-session UI bridge and Tauri wrapper are research-only

- **State.** The Linux remote-session-ui bridge and the repo-local Tauri wrapper run as trusted local backends that hold the upstream capOS session and project view models / call results to

the browser/webview. A policy preflight now proves the wrapper remains check/dev only; distributable packaging and desktop automation modes are intentionally blocked.

- **Owner.** `docs/proposals/remote-session-ui-security-proposal.md`, `docs/proposals/remote-session-capset-client-proposal.md`, `docs/backlog/remote-session-capset-client.md`.
- **Gap.** Distributable packaging, desktop automation, and a reviewed production posture for the remote-session UI surface remain unreviewed in the relevant remote-session proposal/backlog task records. Non-loopback remote-session UI exposure must stay blocked until that posture is accepted.

### R12 – Verification coverage is partial, not full proof

- **State.** Bounded Kani gate (`make kani-lib`/`make kani-lib-full`), Loom ring model, Miri lib tests, proptest, fuzz harnesses, panic-surface inventory, and CI dependency policy exist. Coverage is not whole-system and not seL4-style functional refinement.
- **Owner.** `docs/proposals/security-and-verification-proposal.md`, `docs/security/verification-workflow.md`, `docs/panic-surface-inventory.md`, `docs/backlog/security-verification.md`.
- **Gap.** Public/external claims must distinguish “bounded model checked” from “fully verified”. Promote new properties into Kani/Loom only when the invariant is concrete and bounded. IPC/scheduler panic-surface hardening also remains open around guarded unwraps, rollback restoration, stale queues, blocking waits, process/thread exit, endpoint cancellation, TLB shootdown send failures, and scheduler hot-path expects. Kernel upper-half page-table mutation after AP startup is closed for the current MMIO/firmware helper path by `docs/tasks/done/2026-06-07/kernel-upper-half-pml4-propagation-hardening.md`; future helper windows or allocator-growth paths that need a new kernel-half PML4 slot still require boot preseed or synchronized live-root propagation.

### R13 – Trusted build inputs are partly pinned

- **State.** Limine (commit + artifact SHA-256), `capnp` 1.2.0 source tarball, CUE 0.16.0, mdBook/mdbook-mermaid, Typst 0.14.2, Cargo lockfiles, the Rust nightly date policy, the Kani toolchain bundle, OVMF firmware hash, and the CI apt package versions for `qemu-system-x86`, `xorriso`, `make`, `git`, and `ovmf` are pinned or policy-pinned. `make build-provenance` records local runner identity, GitHub-hosted image identity when present, selected host-tool paths, package identities and normalized apt source pockets when discoverable, and OVMF path/package/hash or absence. CI pull requests run a blocking environment provenance comparison against the latest successful main-branch `qemu-smoke` provenance artifact.
- **Owner.** `docs/trusted-build-inputs.md`, `docs/proposals/cloud-deployment-proposal.md`.
- **Gap.** The PR-blocking environment comparison and qemu-smoke package pins close the previous `make`/`git` identity and advisory-compare gap for CI proof branches, but `ubuntu-24.04` is still a GitHub-managed mutable runner label, not an immutable production image digest. Full production reproducibility still needs a self-built runner image referenced by digest, repo-managed download-and-verify tool digests for the apt-pinned build/boot tools, or both.

**R14 – User identity / policy is proposal-shaped**

- **State.** Anonymous/operator sessions, password setup/login, broker-issued shell bundles, and redacted audit records exist. Durable accounts, ABAC/MAC context, OIDC/passkeys, disk-backed account stores, and resource bundles are proposal-shaped. Stale-session calls and retained shell-bundle caps fail closed for current proof paths, but session liveness is still represented by immutable metadata plus expiry timestamps rather than a mutable session-manager cell with logout, revocation, recovery-only, and renewal state.
- **Owner.** `docs/proposals/user-identity-and-policy-proposal.md`, `docs/backlog/local-users-management.md`, `docs/backlog/session-bound-invocation-context.md`, `docs/proposals/oidc-and-oauth2-proposal.md`, `docs/proposals/certificates-and-tls-proposal.md`, `docs/proposals/cryptography-and-key-management-proposal.md`.
- **Gap.** Until durable identity / persistence / passkey paths land, capOS is not a complete multi-user OS. Demo claims must scope to the proven anonymous + operator + manifest-seeded local accounts model. Before treating fixed short session expiry as production interactive UX, capOS needs explicit `logout`, owner-shell/gateway close propagation, and renewal paths that mint fresh grant leases without reviving stale ordinary grants.

**R15 – App exception serialization depends on result-buffer capacity**

- **State.** Application-level exceptions are serialized into the caller's result buffer; if the target cannot be identified, invocation fails earlier with transport errors. Truncation/transport failures are documented.
- **Owner.** `docs/proposals/error-handling-proposal.md`, `docs/capability-model.md`.
- **Gap.** Service UX/debuggability can degrade for malformed or small-buffer clients. No remediation is required in code today, but each service contract should document its expected result-buffer capacity.

**Open Design Questions**

The following questions came up in external review. Each row gives the **current best answer** observed in the tree, the **canonical tracker** to update, and an explicit **status**.

**Q1 – Cap'n Proto ABI compatibility policy**

- **Current answer.** `docs/abi-evolution-policy.md` defines compatibility classes, stable schema ordinals, reserved-field rules, ring layout rules, version negotiation, deprecation windows, and review gates. Generated-code drift is still checked through `make generated-code-check` and `tools/check-generated-capnp.sh`.
- **Tracker.** `docs/abi-evolution-policy.md`, `docs/trusted-build-inputs.md`, `schema/capos.capnp`, `capos-config/src/ring.rs`.
- **Status.** Answered for the current research tree. Ring v2 compatibility remains a separate open question below.

**Q2 – Ring v2 backward compatibility**

- **Current answer.** `docs/proposals/ring-v2-smp-proposal.md` treats per-thread ring ownership as the full-SMP target and frames it as an evolution that may need ABI changes; `docs/tasks/README.md` calls runtime ring reactor work the compatibility bridge.

- **Tracker.** `docs/proposals/ring-v2-smp-proposal.md`, `docs/backlog/smp-phase-c.md`.
- **Status.** Open. Whether Ring v2 is backward-compatible with the process-wide ring or an explicit ABI break has not been decided.

### Q3 – Which capabilities are copy-transferable vs move-only vs non-transferable

- **Current answer.** `docs/authority-accounting-transfer-design.md` defines copy/move/none transfer modes and the accounting/rollback rules. Per-interface transfer mode is encoded on the schema-defined `CapObject`.
- **Tracker.** `docs/authority-accounting-transfer-design.md`, `schema/capos.capnp`.
- **Status.** Partial. The mode is enforced per object, but the user-visible matrix (which named caps are copy/move/none) is not consolidated in one document.

### Q4 – Copy-transfer replay: feature or compromise

- **Current answer.** Repeatable copy-transfer replay is documented as the current accepted semantics. Exactly-once replay suppression is future work. See R5.
- **Tracker.** `docs/authority-accounting-transfer-design.md`.
- **Status.** Decided as "current semantics, future tightening optional".

### Q5 – When legacy endpoint identity is replaced and what migrates

- **Current answer.** `docs/backlog/session-bound-invocation-context.md` decomposes the selected migration: one immutable session context per process, privacy-preserving endpoint caller-session metadata, chat/adventure/stdio session-keyed migration, and legacy endpoint-identity cleanup. The old service-object identity plan is superseded.
- **Tracker.** `docs/proposals/session-bound-invocation-context-proposal.md`, `docs/backlog/session-bound-invocation-context.md`, `docs/backlog/stage-6-capability-semantics.md`.
- **Status.** Selected milestone. See R3.

### Q6 – Minimum production TCB target

- **Current answer.** `docs/proposals/security-and-verification-proposal.md` now enumerates the current demo/proof TCB and the target production TCB. Current proofs still trust kernel networking, init/supervisors, broker/session services, harnesses, and QEMU virtio. The target production TCB removes ordinary apps and shell children but still includes minimal init/supervisor, credential/session/broker/key/audit services, production device managers, and ABI/schema/build-signature inputs.
- **Tracker.** `docs/security/trust-boundaries.md`, `docs/proposals/userspace-authority-broker-proposal.md`, `docs/proposals/boot-to-shell-proposal.md`.
- **Status.** Partially answered. The TCB statement exists; reducing the actual implementation to that target and proving the non-loopback shell gates remains open.

### Q7 – Revocation strategy

- **Current answer.** Generation/epoch revocation exists for endpoint-backed caps; `CapabilityManager.revoke` cleans up endpoint-backed service objects by object behavior. Session-bound dispatch now fails closed for stale proof paths, but the target lifecycle splits revocation into session liveness cells, grant leases, and object/facet epochs. Revocation trees,

leases, supervisor-owned-cap patterns, and session renewal/close propagation are proposal-shaped.

- **Tracker.** `docs/proposals/service-architecture-proposal.md`, `docs/proposals/session-bound-invocation-context-proposal.md`, `docs/proposals/user-identity-and-policy-proposal.md`, `docs/capability-model.md`.
- **Status.** Open. The chosen revocation primitive set (epochs vs trees vs leases vs explicit-revoke methods per object) needs an explicit decision, and interactive session lifecycle needs a concrete liveness-cell plus renewal protocol.

### Q8 – Boundary between kernel and service-level resource accounting

- **Current answer.** Memory frame grants and cap-table slots are kernel accounting; storage/network buffer accounting is proposed at the service layer. The boundary is not yet implementation-driven.
- **Tracker.** `docs/proposals/resource-accounting-proposal.md`, `docs/proposals/storage-and-naming-proposal.md`, `docs/proposals/networking-proposal.md`.
- **Status.** Open.

### Q9 – CPU accounting and scheduling contexts

- **Current answer.** Per-CPU WFQ run queues, per-thread weighted vruntime, `SchedulingPolicyCap` weight/latency-class authority, and Phase E `SchedulingContext` bind/revoke, budget, donation/return, and depletion notification are implemented per `docs/changelog.md` (Phase D closed 2026-05-10) and `docs/proposals/scheduler-evolution-proposal.md`. Cross-service donation policy, priority inheritance broader than scheduling contexts, explicit scheduling-cap fairness across principals, and full nohz activation remain proposal-shaped.
- **Tracker.** `docs/proposals/smp-proposal.md`, `docs/proposals/scheduler-evolution-proposal.md`, `docs/backlog/scheduler-evolution.md`, `docs/proposals/resource-accounting-proposal.md`, `docs/architecture/scheduling.md`.
- **Status.** Partial. The base CPU accounting and scheduling-context model is implemented through Phase E; the surrounding policy (cross-service donation, full nohz activation, isolation leases, fairness across principals) is the remaining decision.

### Q10 – IOMMU requirement for userspace networking

- **Current answer.** `docs/dma-isolation-design.md` selects a runtime fail-closed backend: direct remapping only when capOS can discover and program trusted translation authority, otherwise a labeled brokered bounce-buffer fallback or unsupported. The current GCP/no-IOMMU userspace-driver evidence uses the brokered bounce path.
- **Tracker.** `docs/dma-isolation-design.md`, `docs/proposals/networking-proposal.md`, `docs/proposals/cloud-deployment-proposal.md`.
- **Status.** Answered for the current no-IOMMU cloud path. Future direct-remapping, vIOMMU, or hostile-hardware isolation claims require their own evidence and remain outside the brokered-bounce production authority closeout.

**Q11 – Capability persistence model**

- **Current answer.** All capabilities are runtime-only today; sealed/stored caps and namespace-mediated reconstitution are storage-proposal scope.
- **Tracker.** `docs/proposals/storage-and-naming-proposal.md`, `docs/proposals/volume-encryption-proposal.md`, `docs/paper/plan.md` (paper-scoped persistence Tier-1 prerequisite).
- **Status.** Open.

**Q12 – Least-privilege shell command invocation**

- **Current answer.** `capos-shell` runs commands using broker-issued bundles; the broker, not the shell, is the policy decision point. `RestrictedShellLauncher` keeps remote shell launches off raw spawn authority.
- **Tracker.** `docs/proposals/shell-proposal.md`, `docs/proposals/userspace-authority-broker-proposal.md`, `docs/proposals/boot-to-shell-proposal.md`.
- **Status.** Direction agreed, complete migration to broker-only authority for every shell-driven invocation is open.

**Q13 – Formal properties to prove**

- **Current answer.** Existing bounded proofs cover cap-table non-forgery, frame-bitmap invariants, transfer rollback, and ring producer-consumer invariants. seL4-style full functional refinement is explicitly out of scope.
- **Tracker.** `docs/proposals/security-and-verification-proposal.md`, `docs/security/verification-workflow.md`, `docs/proposals/formal-mac-mic-proposal.md`.
- **Status.** Partially answered. A definitive list of "what we will keep proving" vs "what we will keep testing" should be added when the next Kani/Loom obligation set is concrete.

**Q14 – Threat model coverage**

- **Current answer.** `docs/proposals/security-and-verification-proposal.md` now contains a threat actor matrix for local physical attackers, malicious DMA devices, malicious boot manifests, compromised init/supervisors, compromised narrow services, hostile network peers, and malicious build dependencies.
- **Tracker.** `docs/security/trust-boundaries.md`, `docs/proposals/security-and-verification-proposal.md`, `docs/dma-isolation-design.md`, `docs/trusted-build-inputs.md`.
- **Status.** Answered at design level. Remaining work is implementation/proof through the relevant task records.

**Q15 – Language runtimes integration model**

- **Current answer.** `capos-rt` is the canonical no_std Rust runtime. Go, Python, Lua, JavaScript/TypeScript, WASI, C/C++, and POSIX-shaped software are future tracks. The current documentation separates native runtime adapters, capability-native bindings, POSIX compatibility adapters, and WASI host adapters instead of treating "compatibility layer" as one shared ABI.

- **Tracker.** `docs/programming-languages.md`, `docs/proposals/userspace-binaries-proposal.md`, `docs/proposals/go-runtime-proposal.md`, `docs/proposals/lua-scripting-proposal.md`.
- **Status.** Open. A common ABI layer vs per-runtime generated clients has not been decided; the current default is per-runtime or adapter-specific clients backed by explicit capabilities.