capOS Documentation

capOS is a research operating system where every kernel service and cross-process service is a typed Cap'n Proto capability invoked through a shared-memory ring.

This generated PDF bundles the current manual, runnable demos, architecture notes, security and verification material, and reference indexes. Links to bundled documentation resolve to sections inside this PDF where available.

Contents

capOS Documentation

capOS is a research operating system where every kernel service and every cross-process service is a **typed Cap'n Proto capability** invoked through a **shared-memory ring**. There is no ambient authority, no global path namespace, and the only remaining syscalls are `cap_enter` and `exit`. The current implementation boots on x86_64 QEMU, loads a Cap'n Proto boot manifest, starts a single manifest-selected init process, and exercises ring-native IPC, capability transfer, Timer, and init-driven spawning through QEMU smoke binaries.

Use this book as the current system manual. The sidebar is organized so the current implementation and operating workflow come first, while backlog, proposal, and research material stays reachable as archive/reference material. What capOS Is has the short version of what makes the design unusual.

The generated Documentation Bundle page collects the current manual and archive indexes into one mdBook-rendered page; a PDF copy is published alongside it.

Start Here

- What capOS Is describes the implemented system model and the main authority boundaries.

- Current Status lists what works today, what is partial, and what remains future work.
- Build, Boot, and Test gives the commands used to build the ISO, boot QEMU, and run host-side validation.
- Repository Map maps the main subsystems to source files.
- Topics Index cross-references every doc by subject so readers can find current, proposal, and research material on the same theme together.
- First Chat Demo shows the smallest runnable resident-service chat proof and its current single-terminal limits.
- Aurelian Frontier (proof slice) shows the current runnable multi-process slice of the Aurelian Frontier game and its QEMU proof.
- Paperclips Terminal Demo shows a clean-room incremental terminal game running as an ordinary shell-launched process.

Site Map

- **System Architecture** is the current system manual: boot, process, capability, runtime, memory, scheduling, IPC, threading, and park behavior.
- **Security and Verification** is the reviewer path: trust boundaries, validation workflow, trusted inputs, panic inventory, and DMA design.
- **Planning** keeps live and historical project direction together: Roadmap, Changelog, Whitepaper Evidence Gaps, and Backlog.
- **Design Archive** starts at the Proposal Index, which classifies active, future, and rejected proposal documents before a reader opens long-form design text.
- **Research and Papers** collects active long-form paper drafts and prior-art reports through Papers and the Research Index.

Operational planning starts from `docs/roadmap.md`, `WORKPLAN.md`, and `REVIEW_FINDINGS.md`. Treat those as live planning and review records; use the book's backlog, changelog, and paper evidence pages for detailed decompositions, historical reports, and publication-evidence tracking.

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 host-local Telnet 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 `CapId`s. 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

This page describes current repository behavior, not the full long-term design. For operational priority and open review items, read `WORKPLAN.md` and `REVIEW_FINDINGS.md`.

Status 2026-04-28 22:02 UTC: normal shell `client @...` grants reject explicit `badge N` selector syntax and preserve delegated client endpoint identity when the selector is omitted. Low-level and hostile-path tests still carry explicit selector fixtures.

Status 2026-04-29 05:59 UTC: the focused chat manifest now routes the kernel singleton `chat_endpoint` through init to the resident chat server, and the focused chat shell no longer receives a manifest-forwarded `chat` service export. Its normal chat authority comes from the broker-issued operator shell bundle, matching the default and remote shell paths while the resident bot keeps its manifest service grant.

Status 2026-04-29 07:35 UTC: Session-Bound Invocation Context core gates are landed. Implemented pieces include the process-session invariant, endpoint caller-session metadata, stale normal endpoint rejection, transfer scopes, field-granular disclosure gating, session expiry for broker-issued shell bundle caps, guest bundle narrowing, chat session-keyed membership, and Aurelian player state keyed by live endpoint caller-session metadata.

Status 2026-04-29 09:44 UTC: later Gate 4 cleanup put terminal output behind live caller-session dispatch, bound shell-serviced stdio bridge waits to opaque live caller-session metadata, removed remaining badge-facing service-common handler APIs from normal chat paths, and widened non-adventure endpoint caller-session opaque references to 128 bits while preserving the `scoped_ref` ABI field as the low half.

Status 2026-04-29 10:20 UTC: non-adventure endpoint caller-session references now use an entropy-backed boot secret and HMAC-SHA256 over a non-reused endpoint service-scope id plus kernel session id. The ABI layout is unchanged, but `scoped_ref` is no longer value-compatible with the old unkeyed hash. References rotate on reboot and endpoint object replacement. Remaining cleanup covers peer-owned adventure slices, compatibility aliases, and future stable service-audit identity across upgrades.

Implemented

Visible Milestone Proofs

- **First Packet**: commit `b56a5c1` at `2026-04-24 15:37 UTC`.
- **First HTTP**: commit `a4f1722` at `2026-04-24 16:47 UTC`.
- **The Unprivileged Stranger**: commit `d4016ab` at `2026-04-22 16:35 UTC`.
- **Native Cap Shell**: commit `f554e88` at `2026-04-23 08:41 UTC`.
- **Boot to Shell**: commit `e5adafb` at `2026-04-23 13:39 UTC`.
- **The Revocable Read**: commit `7f19af2` at `2026-04-23 16:15 UTC`.
- **First Chat**: commit `2cd85a8` at `2026-04-24 00:13 UTC`.
- **Local MUD**: commit `add7f9b` at `2026-04-24 01:40 UTC`.
- **Verified Core**: commit `d43b691` at `2026-04-23 22:09 UTC`.
- **Ring as Black Box**: commit `da5f5e9` at `2026-04-24 03:13 UTC`.
- **First AP Scheduler**: commit `d88bca7` at `2026-04-25 11:31 UTC`.
- **Telnet Shell Demo**: commit `2834bfc` at `2026-04-25 20:25 UTC`. **Demo status:** plaintext, loopback-only research demo proving the `TerminalSession`/`SessionManager`/`AuthorityBroker`/`RestrictedShellLauncher` boundary over a real TCP socket; not a shippable Telnet service. Production remote shell tracks the SSH Shell Gateway in `proposals/ssh-shell-proposal.md`.
- **Default Run Telnet Wiring**: commit `4304b0e` at `2026-04-26 23:12 EEST`; `make run` starts the host-local Telnet gateway on `127.0.0.1:2323` (loopback-bound by design — see Telnet Shell Demo status above). Follow-up commit `7a155f4` at `2026-04-27 00:02 EEST` moves Telnet IAC filtering into the kernel socket terminal (best-effort silent swallow for the BSD/netkit clients we test against; no WONT/DONT replies) so a normal Telnet client lands at the shell prompt without a userspace pre-handoff recv, and refactors the gateway to loop accept/handoff/launch_shell/wait per connection so repeated host Telnet connections succeed.

- **Service Object Routing/Lifecycle**: commit `a4655f0` at `2026-04-28 14:10 UTC`; `make run-service-object-routing` proves trusted service-object minting, receiver-cookie dispatch, payload-spoof rejection, copy/move IPC transfer, nested spawn delegation, generation-checked receiver cookies, close/revoke rejection, and stale-cookie rejection after record reuse. This is now historical low-level coverage: the active direction is Session-Bound Invocation Context, where normal workload processes have one immutable session context and endpoint subject disclosure is private by default.
- **Session Context Invariant**: commit `3edee90` at `2026-04-28 16:26 UTC` adds `make run-session-context`, proving every spawned process has one immutable session context, raw child spawns inherit the caller context, a copied `UserSession` cap cannot relabel invocation context, and a broker-issued launcher can select a validated child context while mismatched profile requests fail closed. Commit `3469c27` at `2026-04-28 16:54 UTC` extends the proof so expired guest-session bundle refreshes fail closed at the broker. Commit `687511a` at `2026-04-28 17:43 UTC` adds privacy-preserving endpoint caller-session metadata and stale normal endpoint rejection: endpoint servers receive only a service-scoped opaque caller-session reference, epoch, and live/stale flags by default, spoofed `user`/`session`/`role` payload labels do not affect the delivered invocation context, and calls after the process session expires fail before transfer preparation or enqueue. Commit `f0cb74b` at `2026-04-28 18:38 UTC` adds session-aware cap transfer scopes: same-session-only caps cannot cross into another session, explicitly shareable caps may cross and then invoke under the receiver session, and service-regrant-only caps require a trusted fixed-session broker/launcher path. Commit `0f92d77` at `2026-04-28 19:33 UTC` adds explicit endpoint subject disclosure gating: request without scope and scope without request expose no subject fields, request plus matching scope exposes only allowed fields, and broader requests are narrowed. Commit `dc7ece4` at `2026-04-28 20:06 UTC` migrates the chat demo to session-keyed membership: chat member state is keyed by the endpoint caller-session reference, the focused chat manifest no longer assigns static chat badges, and `make run-chat` proves operator-session chat clients plus rejected delegated endpoint relabeling. A follow-up review fix keeps join handles as request data and uses service-assigned visible member labels.
- `2026-04-28 20:48 UTC` narrows guest shell bundles: guest sessions require an explicit manifest guest seed, guest bundles receive no default chat/adventure service endpoint caps, and guest launcher policy comes from resource-profile `launcherProfile` rather than the full manifest binary list.

Boot and Kernel Baseline

- Limine boots the x86_64 kernel in QEMU.
- The kernel initializes dual UART output, GDT, IDT, LAPIC, syscall MSRs, memory management, page tables, heap allocation, and the global capability registry. The legacy PIC/PIT path remains as a fallback when LAPIC timer setup or PIT-based calibration is unavailable.
- User page-table map, unmap, and protect operations are routed through a TLB shootdown helper keyed by address-space CPU residency. Remote targets get pending full-TLB flush generations plus vector-49 IPIs, and the sender waits for observed target completion after ring dispatch releases address-space, cap-table, and scratch locks. Deferred queue slots are reserved before page-table mutation, and drains flush the current CPU before waiting. Delayed maskable

interrupt delivery is covered by syscall-entry and flush-before-user-return hooks. Scheduler CR3 handoff marks the current CPU resident, including AP cpu=1 during the first AP scheduler-owner proof, so remote shootdown targets become active when an address space has run on more than one CPU.
- AP cpu=1 can own scheduler/user execution under `-smp 2`: APs register their `PerCpu` records, program LAPIC timers from the BSP calibration, update AP TSS.RSP0 during context switches, and enter the scheduler from the AP idle loop when AP timer setup succeeds. This proof keeps one scheduler owner; when AP cpu=1 is online with a programmed timer, the BSP stays in kernel idle so the process-wide capability ring is not executed concurrently.
- The kernel creates its own page tables with per-section permissions and keeps the higher-half direct map for physical memory access.
- SMEP/SMAP are enabled when the QEMU CPU advertises support.

Code: `kernel/src/main.rs`, `kernel/src/arch/x86_64/`, `kernel/src/mem/`.

Validation: `cargo build --features qemu`, `make run-smoke`.

Process and Userspace Runtime

- Processes have isolated address spaces, one or more internal Thread records with per-thread kernel stacks and saved CPU context, CapSet bootstrap pages, capability rings, and local capability tables.
- ELF loading supports static no_std userspace binaries and TLS setup.
- `capos-rt` owns the userspace entry path, allocator initialization, ring-client access, typed clients, result-cap parsing, and owned-handle release.
- `capos-rt` is the only source owner for the userspace `_start`, panic, global allocator, raw syscall, and `capos_rt_main` handoff surfaces; a source check guards this split.
- `targets/x86_64-unknown-capos.json` defines the capOS userspace target for booted `init`, `demos`, `shell`, and `capos-rt` runtime builds; the kernel default remains `x86_64-unknown-none`.
- The 7.1.0 in-process threading contract defines the split between process-owned address-space/capability state and thread-owned execution state, plus thread/kernel-stack quotas and generation-checked waiter identity. 7.2.0 moved saved context, kernel stack, FS base, and block state into `Thread` records; 7.2.1 schedules and wakes generation-checked `ThreadRef` values; 7.2.2 adds process-local `ThreadSpawner` and `ThreadHandle` caps plus `ThreadControl.exitThread` for create, join, detach, self-join rejection, exit-code observation, and last-thread process exit; and 7.2.3 adds private ParkSpace wait/wake with timeout, wake, and reserved waiter completion semantics. SharedParkSpace park-words remain future work.

Code: `kernel/src/spawn.rs`, `kernel/src/process.rs`, `capos-rt/src/`, `init/src/main.rs`, `demos/`, `shell/`, `targets/x86_64-unknown-capos.json`, `tools/check-userspace-runtime-surface.sh`.

Design: In-Process Threading, Park Authority.

Validation: `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`, `make run-smoke`, `make run-spawn`.

Capability Ring and IPC

- The shared ring ABI supports CALL, RECV, RETURN, RELEASE, NOP, and compact ParkSpace PARK/UNPARK transport operations.
- `cap_enter` processes submissions and can block until completions arrive or a timeout expires.
- Endpoints route ring-native IPC between processes.
- Direct IPC handoff lets a blocked receiver run before unrelated round-robin work after a matching CALL arrives.
- Transport errors and application exceptions are surfaced through CQEs and typed runtime client errors.
- Ordinary capability implementation errors, revoked ordinary/endpoint use, live endpoint target errors after endpoint identification, and endpoint RETURN application failures use serialized `CapException` payloads when a caller result buffer can safely receive one. No-payload application failures report `CAP_ERR_APPLICATION_EXCEPTION_TRUNCATED`; malformed transport metadata and unsafe result-buffer paths remain transport errors.
- Endpoint RETURN can propagate a serialized `CapException` from a userspace endpoint server to the original cross-process caller.
- `debug_tap` builds export metadata-only `ringtap:` records for observed SQEs and posted CQEs on the QEMU/debug UART. The format is fixed, bounded, and deliberately records `payload_len = 0` until a separate payload-capture authority lands.
- `tools/ringtap-viewer/` parses `ringtap:` logs into SQE/CQE summaries and can decode authorized Cap'n Proto payloads for `CapException`, `TerminalSession.readLine` params, and `ProcessHandle.wait` results when future tap output includes `payload_schema` and `payload_hex` fields.
- `make run-ringtap-failing-call` boots the default shell smoke with `debug_tap`, drives the known `typed-call` method-99 launcher failure, runs the viewer over the captured kernel log, and leaves offline inspection logs in `target/ringtap-failing-call-*.log`.

Code: `capos-config/src/ring.rs`, `kernel/src/cap/ring.rs`, `kernel/src/cap/endpoint.rs`, `kernel/src/debug_tap.rs`, `capos-rt/src/ring.rs`, `capos-rt/src/client.rs`, `tools/ringtap-failing-call-smoke.sh`, `tools/ringtap-viewer/`.

Validation: `cargo test-ring-loom`, `make run-smoke`, `make run-spawn`, `make run-smoke CARGO_FLAGS='--features debug_tap'`, `cd tools/ringtap-viewer && cargo test`, `make run-ringtap-failing-call`.

Capabilities

Implemented kernel capabilities include:

- Console for debug UART output.
- TerminalSession for the separate session UART with line input/output, bounded `readLine`, visible/hidden echo, structured cancellation, and a single move-only foreground holder.
- BootPackage for read-only, chunked boot manifest reads from init.
- FrameAllocator for typed `MemoryObject` frame ownership grants.
- MemoryObject for owned physical frame ranges, caller-local map/unmap/protect, and final backing release after cap/mapping teardown.

- Endpoint for IPC rendezvous.
- VirtualMemory for anonymous user page map, unmap, and protect operations.
- Timer for monotonic tick/time reads and bounded sleep completions through the capability ring.
- ThreadControl for runtime-owned FS-base get/set and terminal `exitThread` on the current thread.
- ThreadSpawner and ThreadHandle for process-local in-process thread creation, one-shot join, exit-code observation, detach-on-release, and retained-status cleanup.
- ParkSpace for process-local private park wait/wake on 32-bit userspace words, with per-thread blocking and reserved waiter CQE credits.
- ProcessSpawner and ProcessHandle for init-driven child process creation and wait semantics.

MemoryObject holders and anonymous VirtualMemory mappings charge the same per-process `ResourceLedger::frame_grant_pages` quota. Mapping a held MemoryObject records borrowed address-space pages and reserves mapping quota until unmap so backing frames cannot stay pinned after the cap charge is released.

Code: `kernel/src/cap/console.rs`, `kernel/src/cap/terminal_session.rs`, `kernel/src/cap/boot_package.rs`, `kernel/src/cap/frame_alloc.rs`, `kernel/src/cap/endpoint.rs`, `kernel/src/cap/virtual_memory.rs`, `kernel/src/cap/timer.rs`, `kernel/src/cap/thread_control.rs`, `kernel/src/cap/thread_handle.rs`, `kernel/src/cap/process_spawner.rs`.

Validation: `make run-smoke`, `make run-memoryobject-shared`, `make run-spawn`, `make run-shell`, `make run-terminal`, `cargo test-lib`.

Capability Transfer and Release

- IPC CALL and RETURN support sideband transfer descriptors.
- Copy and move transfer are implemented.
- Move transfer reserves the sender slot until destination insertion and commit.
- Transfer result caps carry interface ids to userspace.
- `CAP_OP_RELEASE` removes local capability-table slots. Runtime owned-handle drop queues one local release, and `Runtime::flush_releases()` forces queued releases when code cannot wait for the next ring-client acquisition/drop.

Code: `kernel/src/cap/transfer.rs`, `kernel/src/cap/ring.rs`, `capos-lib/src/cap_table.rs`, `capos-rt/src/ring.rs`.

Validation: `cargo test-lib`, `make run-smoke`.

Manifest Tooling and Smokes

- `tools/mkmanifest` turns `system.cue` into a Cap'n Proto boot manifest.
- The build uses repo-pinned Cap'n Proto and CUE tool paths through the Makefile; direct `mkmanifest` invocation also rejects missing, unpinned, or version-mismatched CUE compilers. `mkmanifest cue-to-capnp` extends the same pinned-tool policy to general CUE-authored data

messages: it exports CUE as JSON, validates `CAPOS_CAPNP`, and delegates arbitrary specified schema-rooted struct serialization to `capnp convert json:binary`.

- Default scripted QEMU smoke still uses the focused shell-led `system-smoke.cue` path: anonymous session on boot, `login` command failing on a wrong password without echo, succeeding on the correct one, broker upgrade to the operator bundle, child terminal isolation, stale-handle release, single-`capos-shell` init boot, and clean halt. The default operator-facing `system.cue` path is init-owned and is exercised by `make run`.
- `system.cue` is now the default init-owned manifest. The kernel starts only the first `init` service, and init starts `capos-shell`, `telnet-gateway`, and the default demo services from the manifest service graph. The shell receives terminal/creds/sessions/audit/broker caps and mints its own anonymous session.
- `system-shell.cue` is the focused anonymous-shell proof (no verifier), which exercises the shell in its anonymous bundle and asserts that the anonymous launcher rejects spawns because its allowlist is empty.
- `system-chat.cue` is the focused First Chat prototype proof. It starts a resident `Chat` endpoint service on the kernel singleton `chat_endpoint`, a resident bot participant, and the shell; `make run-chat` drives `run "chat-client"` with explicit `StdIO` plus the broker-issued `chat` endpoint grant, sends one line, and checks that the bot reply is printed by the foreground client.
- `system-adventure.cue` is the focused adventure prototype proof. It keeps adventure out of shell builtins and drives `run "adventure-client"` through explicit `StdIO`, `adventure`, and `chat` endpoint grants. See the Aurelian Frontier (proof slice) page for the current mission, commands, and transcript coverage.
- `system-paperclips.cue` is the focused clean-room Paperclips-style terminal demo proof. It keeps game state in the `paperclips` process, grants only `StdIO` plus `Timer`, and `make run-paperclips` drives one-at-a-time manual production, bulk-manual rejection, real-time automation, simulation ticks, generated Cap'n Proto content loading, project listing, and clean process exit through the native shell.
- `demos/service-common/` holds the shared caller-session endpoint loop and chat actor bootstrap/polling helpers used by the chat/adventure resident services, chat bot, and adventure NPC processes. New shared endpoint loop code uses `EndpointUserData`; the old badge-named user-data alias remains only for compatibility while peer branches migrate. Shared event queues remain deferred until another service has queue needs matching chat history/inbox behavior.
- `system-spawn.cue` remains the focused ProcessSpawner smoke for endpoint, IPC, VirtualMemory, Timer, ThreadControl, FrameAllocator cleanup, and hostile spawn inputs. `make run-spawn` asserts that the kernel boot-launches only the standalone `init`, that init validates BootPackage metadata, and that the init-owned manifest executor spawns and waits for every focused child service, including the `timer-smoke` monotonic now/sleep proof, `timer-flood` per-process Timer sleep quota proof, `runtime-fs-base` runtime-owned FS-base proof, `single-thread-runtime` VirtualMemory plus Timer runtime checkpoint, and `thread-lifecycle` in-process thread/park proof.

Code: `tools/mkmanifest/`, `system.cue`, `system-chat.cue`, `system-adventure.cue`, `system-paperclips.cue`, `system-spawn.cue`, `demos/`, `capos-rt/`.

Validation: `cargo test-mkmanifest`, `make generated-code-check`, `make run-smoke`, `make run-chat`, `make run-adventure`, `make run-paperclips`, `make run-spawn`.

Partially Implemented

Login Boot and Init-Owned Spawn

Default `make run` now uses the init-owned default manifest. The kernel validates the kernel-owned boot boundary, boot-launches standalone `init`, and leaves the service graph plus login/session/broker flow in userspace. Init starts the foreground `capos-shell` service, resident demo services, and the host-local `telnet-gateway` service; `make run` forwards host `127.0.0.1:2323` to guest port 23. The foreground shell mints its own anonymous `UserSession` on boot; `login` and `setup` commands drive `CredentialStore`/`SessionManager`/`AuthorityBroker` to upgrade the session in place. Password login is on the ordinary foreground shell path. The Telnet gateway is also on the default boot path. The kernel-side socket terminal best-effort filters Telnet IAC negotiation in its line discipline, the gateway loops accept/handoff/launch_shell/wait per connection, and a normal Telnet client lands at the shell prompt and can disconnect-and- reconnect without rebooting.

The focused init-owned spawn path remains under `make run-spawn`. There the kernel boot-launches `init` with Console, BootPackage, and ProcessSpawner. Parent endpoint facets used for later service-sourced imports are returned by `ProcessSpawner` during child spawn, not granted at boot. `init` performs metadata-only manifest validation, resolves kernel and service cap sources, spawns children through ProcessSpawner, records exports, waits for children, and reports failures through Console output. The QEMU target now asserts the single-init boot markers, the three-cap init bundle, BootPackage validation, child exit records, manifest child waits, spawn-loop completion, and clean halt.

Measurement startup now follows the same boundary. `make run-measure` uses a focused `system-measure.cue` manifest where the kernel boots standalone `init` with Console, BootPackage, and ProcessSpawner, and init spawns `ring-nop` with Console, FrameAllocator, the measurement-only NullCap, and the measurement-only ParkBench cap through ProcessSpawner grants. It also spawns `thread-lifecycle` with ThreadControl, ThreadSpawner, ParkSpace, and a measurement marker cap. The demos print compact versus generic park-shaped failed-wait/empty-wake cycle averages plus real ParkSpace blocked/resume cycle averages before the measure-feature kernel prints segmented dispatch counts, total cycles, and averages for SQE processing, validation, cap lookup, capnp decode, method body dispatch, CQE posting, and waiter wake/check. Kernel bootstrap now loads only `initConfig.init` and validates only the kernel-owned manifest boundary; mkmanifest and init own `initConfig.services` graph validation for focused BootPackage executor manifests.

Hardware and Networking

The QEMU virtio-net path has legacy PCI config-space enumeration, a `make run-net` boot target, modern virtio PCI transport discovery for the common, notify, ISR, and device-specific MMIO regions, feature negotiation, and RX/TX split-virtqueue initialization, a TX descriptor completion proof, minimal Ethernet ARP resolution, and ICMP echo validation against the QEMU user-mode

gateway. The same kernel-internal path also wraps the virtio driver in a smoltcp Ethernet device, configures static IPv4 as `10.0.2.15/24`, routes through `10.0.2.2`, and performs a TCP HTTP GET against the host harness on port 8000. QEMU currently exposes a transitional `1af4:1000` virtio-net function with modern vendor capabilities; capOS accepts that shape only through the modern capability layout. The kernel negotiates `VIRTIO_F_VERSION_1` plus MAC when safe and `VIRTIO_NET_F_MRG_RXBUF` for QEMU's merged-buffer virtio-net header, maps the virtio MMIO regions after kernel paging is live, allocates kernel-owned DMA pages for RX/TX descriptor, available, used rings, RX packet buffers, and one-shot TX buffers, submits a descriptor proof frame, sends an ARP request from `10.0.2.15` to `10.0.2.2`, and observes the ARP reply in `make run-net`. The same smoke sends an IPv4 ICMP echo request to `10.0.2.2`, validates the echo reply identifier, sequence, payload, IPv4/ICMP checksums, and addresses, and prints an `icmp echo ok` proof line. The smoltcp smoke uses the same virtqueue path to send the TCP SYN and HTTP request, copies completed RX frames into stack-owned buffers, and prints `smoltcp tcp http ok` after an HTTP 200 response from the host server. The smoltcp interface and socket set now persist as a runtime polled from `schedule()` through `virtio::poll_scheduler()` on scheduler ticks, with timestamps derived from real monotonic `TICK_COUNT`/`TICK_NS` instead of the earlier bounded synthetic 10 ms-per-poll clock. The Phase B TCP capability surface now exposes `NetworkManager`, `TcpListener`, and `TcpSocket` kernel `CapObject`s wrapping that retained runtime, using result-cap indices for listener/socket returns, bounded accept/recv waiters, partial send, receive clamping, and explicit close/drop cleanup. The `SocketTerminalSession` byte handler in `kernel/src/cap/network.rs` also drives the Telnet IAC state machine and writes the initial `IAC WILL ECHO IAC WILL SUPPRESS-GO-AHEAD` burst at handoff so the host telnet client switches to character mode.

This is a deliberate transitional shape: smoltcp, the per-port listeners, the accepted-socket capability state, the line discipline, and the Telnet IAC filter are all in-kernel today. Phase C (networking-proposal.md Part 3) moves the TCP/IP stack into a userspace network stack process and reduces the kernel surface to `DMAPool`/`DeviceMmio`/`Interrupt` device capabilities; that is blocked on the userspace-driver authority gate.

Code: `kernel/src/pci.rs`, `kernel/src/virtio.rs`, `kernel/src/cap/network.rs`, `kernel/src/cap/ring.rs`, `kernel/src/sched.rs`, `kernel/src/mem/paging.rs`, `kernel/src/arch/x86_64/pci_config.rs`, `tools/qemu-net-smoke.sh`, `tools/qemu-net-harness.sh`.

Validation: `make run-net`, `make qemu-net-harness`.

Security and Verification Track

The repo has Miri, proptest, fuzz, Loom, Kani, generated-code, dependency policy, trusted-build-input, panic-surface, and DMA-isolation work. CI now runs a bounded Kani gate for `capos-lib` bitmap, cap-table stale-handle, transfer preflight, transfer rollback split between source-visible rollback and destination-ledger restoration, and frame-grant accounting invariants. The heavier prepare-copy to provisional-destination seam proof passed in the high-memory `make kani-lib-full` Cloud Build gate, but coverage is not complete for every trust boundary.

References: Trusted Build Inputs, Panic Surface Inventory, DMA Isolation, and Security and Verification Proposal.

Future Work

Future architecture includes service restart policy, capability-scoped system monitoring, notification objects, promise pipelining, service-facing SharedBuffer APIs on top of the MemoryObject substrate, scheduling-context donation, session quotas, SMP, storage and naming, userspace networking, cloud boot support, user identity, policy enforcement, multi-front-end terminal hosts, richer native command surfaces, and broader language/runtime support.

Design references:

- Service Architecture
- Storage and Naming
- Networking
- SMP
- Userspace Binaries
- Shell
- Boot to Shell
- System Monitoring
- User Identity and Policy

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
- 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 tools are installed under the clone-shared `.capos-tools` directory next to the git common directory.

Build the ISO

```
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/`.

Boot QEMU

```
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 `127.0.0.1:2323 -> guest :23` forwarding, and writes the separate kernel/debug UART log to `target/qemu-console.log`.

> **Telnet demo status.** The Telnet path on `127.0.0.1:2323` is a plaintext, loopback-only research demo. It exists to exercise the `TerminalSession`/`SessionManager`/`AuthorityBroker`/`RestrictedShellLauncher` boundary over a real TCP socket; it is not a shippable Telnet service, has no transport encryption, and must not be exposed beyond loopback. The production remote-shell successor is the SSH Shell Gateway tracked in `docs/proposals/ssh-shell-proposal.md`. Remote or non-loopback shell exposure remains blocked until pre-auth and post-auth shell authority are isolated into separate processes or the shared authority model is proven constrained, and until the SSH gateway's transport, key, account, storage, and audit gates pass.

After boot, `telnet 127.0.0.1 2323` reaches the forwarded Telnet gateway and lands at the shell prompt; the kernel-side socket terminal silently swallows Telnet IAC option-negotiation bytes (best-effort filtering for the BSD/netkit clients we test against; no explicit WONT/DONT replies), so the gateway never performs a raw recv before the `TerminalSession` handoff. The gateway loops on accept, so disconnect-and-reconnect works without rebooting.

`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

```
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

```
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 failing on a wrong password, succeeding on the correct one, 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.

Focused Service Smokes

```
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, launches the clean-room `paperclips` client with only `StdIO` and `Timer`, rejects bulk manual production, drives one-at-a-time manual production, real-time automation, simulation ticks, project listing, generated Cap'n Proto content loading, 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

```
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

```
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

```
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

```
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. 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.

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 covers the slice-2 extension surface. The design rationale lives in `docs/proposals/system-configuration-proposal.md`.

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.

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.

Worked examples

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).

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
    pendingIpcSubmissionLimit: 8
    ringScratchLimitBytes:      16384
    logQuotaBytesPerWindow:     32768
    networkProfile:             "none"
    cpuBudgetUsPerWindow:       10000
    cpuWindowUs:                100000
    timerWaiterLimit:           2
    launcherProfile:            "bootstrap-guest"
}]
```

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

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

`make run` runs `cue export` with `--inject user=$(USER)`, so the operator session shows the host login as the displayName. Other Make targets leave the tag unset, keeping the canonical `operator` value that smoke harnesses assert against. The audit-correlatable `principalId` is fixed to the canonical 32-byte value regardless of host user, so audit history is stable across `$USER` changes.

`mkmanifest` reads `CAPOS_CUE_TAGS` from the environment and forwards each `key=value` pair as `--inject key=value`. The Makefile sets it target-scoped to `make run` only:

```
run: CAPOS_CUE_TAGS = user=$(USER)
```

Set additional tags via `CAPOS_CUE_TAGS=user=alice,region=eu-west make run` or by passing `--tag key=value` to `mkmanifest` directly. `system.cue` only consumes `user` today; 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 `docs/proposals/user-identity-and-policy-proposal.md`.
- The slice-2 overlay is not a replacement for cloud-instance configuration; cloud-metadata-driven manifest deltas are designed in `docs/proposals/cloud-metadata-proposal.md`.
- The overlay does not auto-ingest `~/.ssh/*.pub`; conversion is manual by design (security review on which keys count).
- Focused-proof manifests are migrating onto the defaults package in slice 3. `system-spawn.cue`, `system-shell.cue`, `system-terminal.cue`, `system-telnet.cue`, `system-login.cue`, `system-login-setup.cue`, `system-local-users.cue`, `system-credential.cue`, and `system-ssh-*.cue`, `system-smoke.cue`, `system-session-context.cue`, `system-restricted-shell-launcher.cue`, `system-chat.cue`, `system-revocable-read.cue`, `system-memoryobject-shared.cue`, `system-ipc-zerocopy.cue`, and `system-service-object-routing.cue`, `system-network-client.cue`, and `system-tcp-listen-authority.cue` are already packaged; the remaining variants still use their legacy single-file shape until their focused checks move with them.

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.
- `WORKPLAN.md` records the current selected milestone and implementation ordering.
- `REVIEW_FINDINGS.md` records open review findings and verification history.
- `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` pins the Rust toolchain.
- `.cargo/config.toml` sets the default bare-metal target and useful cargo aliases.

Schema and Shared ABIs

- `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.

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.
- `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 the current QEMU virtio-net PCI enumeration smoke path.

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

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.
- `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/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.

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/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/park_space.rs` implements the process-local ParkSpace marker capability used by compact park (`CAP_OP_PARK`/`CAP_OP_UNPARK`) opcodes.
- `kernel/src/cap/process_spawner.rs` implements ProcessSpawner and ProcessHandle.
- `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.
- `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.
- `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`.

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-roundtrip, ipc-server, ipc-client
- 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

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, Telnet 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.
- `tools/check-generated-capnp.sh` verifies checked-in generated schema output.
- `tools/qemu-net-harness.sh` runs the current QEMU net harness.
- `fuzz/` contains fuzz targets for manifest Cap'n Proto decoding, mkmanifest JSON conversion/validation, and ELF parsing.

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.md` and `docs/research/` summarize prior art.
- `docs/proposals/mdbook-docs-site-proposal.md` defines the documentation site structure and status vocabulary used by these Start Here pages.

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` Telnet path 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. A Telnet-launched 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. Status also prints a concise regional frontier summary for the generated settlement, outpost, and route metadata, plus a construction foundation summary for generated blueprint, artifact, enchantment slot, and gate metadata; construction jobs, material reservation, and full artifact crafting gameplay are not implemented. Status now also prints disabled-by-default optional fake-agent NPC budget metadata: budget count, aggregate session token budget, tool-call budget, and audit visibility. That is deterministic metadata for future optional chatter and hints, 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, and scripted proof-path metadata 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
transfer scout-marker to player-1
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;
- market coverage proves a Maro route quote, a successful route exchange, a field-engineer repair response, and an Iunia clean-custody trade refusal that names the `temple-seal` gate and price; 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`, 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, construction foundation, 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 game `Universal Paperclips`. It runs as an ordinary capOS userspace process launched from the native shell. The shell grants only `StdIO` and `Timer`; all game state lives in the child process and disappears when that process exits.

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 a short transcript, and asserts clean child and shell exit:

```
run "paperclips" with { stdio: client @stdio, timer: @timer }
status
buy marketing
make

make 100
buy autoclipper
status
run 3
projects
exit
```

For default manual use, boot the ordinary playground:

```
make run
```

After login:

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

Useful commands inside the demo:

```
status
projects
make
sell <n>
price <cents>
buy wire <n>
buy autoclipper
buy marketing
buy processor
buy memory
project <id>
run <ticks>
help
exit
```

`make` creates exactly one paperclip. Automation advances once per second while the process is running; `run <ticks>` is only a fast-forward command. Blank input repeats the last non-empty command. Later-stage purchase commands such as `buy drone`, `buy factory`, and `buy probe` appear in `help` only after the corresponding automation path is unlocked.

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>`. 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, step size, and deterministic generator parameters. It also has an `autoSellEnabled` rule for experiments that should sell during ticks, 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
  -> serialized Cap'n Proto CueValue bytes in src/generated.rs
  -> paperclips-content deserializes the bytes at startup
```

The CUE file owns the game balance: initial state, purchase costs, tick rates, explicit/automatic selling policy, demand rules, trust milestones, project costs, project labels/descriptions, 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 manual single-clip production, sales, wire purchases, autoclippers, processors, and memory;
- repeatable marketing investment raises dynamic demand, while early projects improve autoclippers, unlock creativity, and unlock Yomi generation;
- HypnoDrones moves the game into autonomous matter conversion;
- drone/factory/swarm projects scale harvesting, production, and compute;
- Von Neumann probes move the game into cosmic replication;
- Universal Paperclips 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;
- the child receives only explicit `StdIO` and `Timer` grants;
- 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 timer capability drives real-time automation without ambient clock access;
- the child maintains local game state without a kernel service or 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 a capability and process-boundary demo, but not a distributed game-state service. The shell and Paperclips process communicate over the granted `StdIO` endpoint, and Paperclips independently calls its granted `Timer` cap. The game rules themselves still run inside the Paperclips process; no separate economy or project service participates yet.

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 and project locking; the focused QEMU proof covers launch and the first production loop. Autonomous, cosmic, and completion-stage transcript coverage remains a future test expansion.

Future rule/content expansion is tracked in `docs/backlog/paperclips.md`. 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.

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.

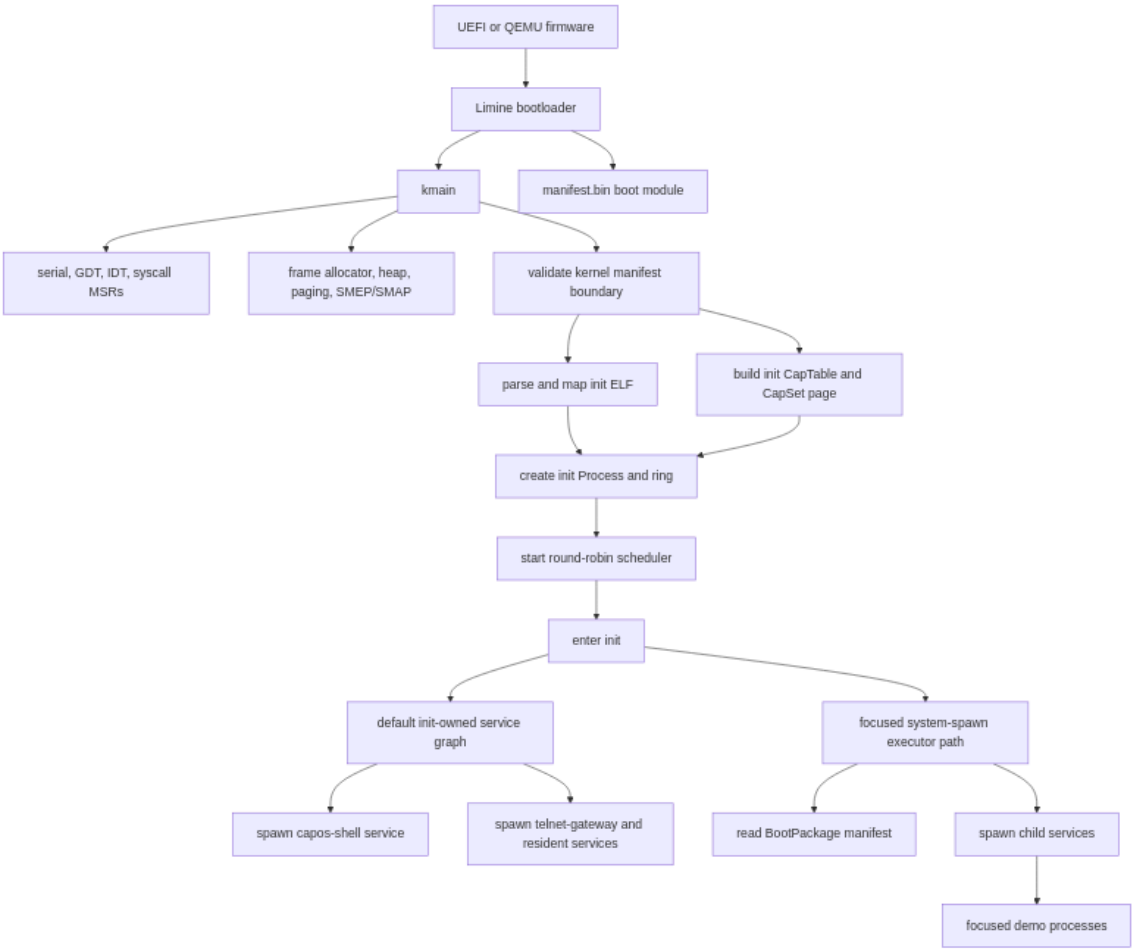

Figure 1: diagram

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, Telnet 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.

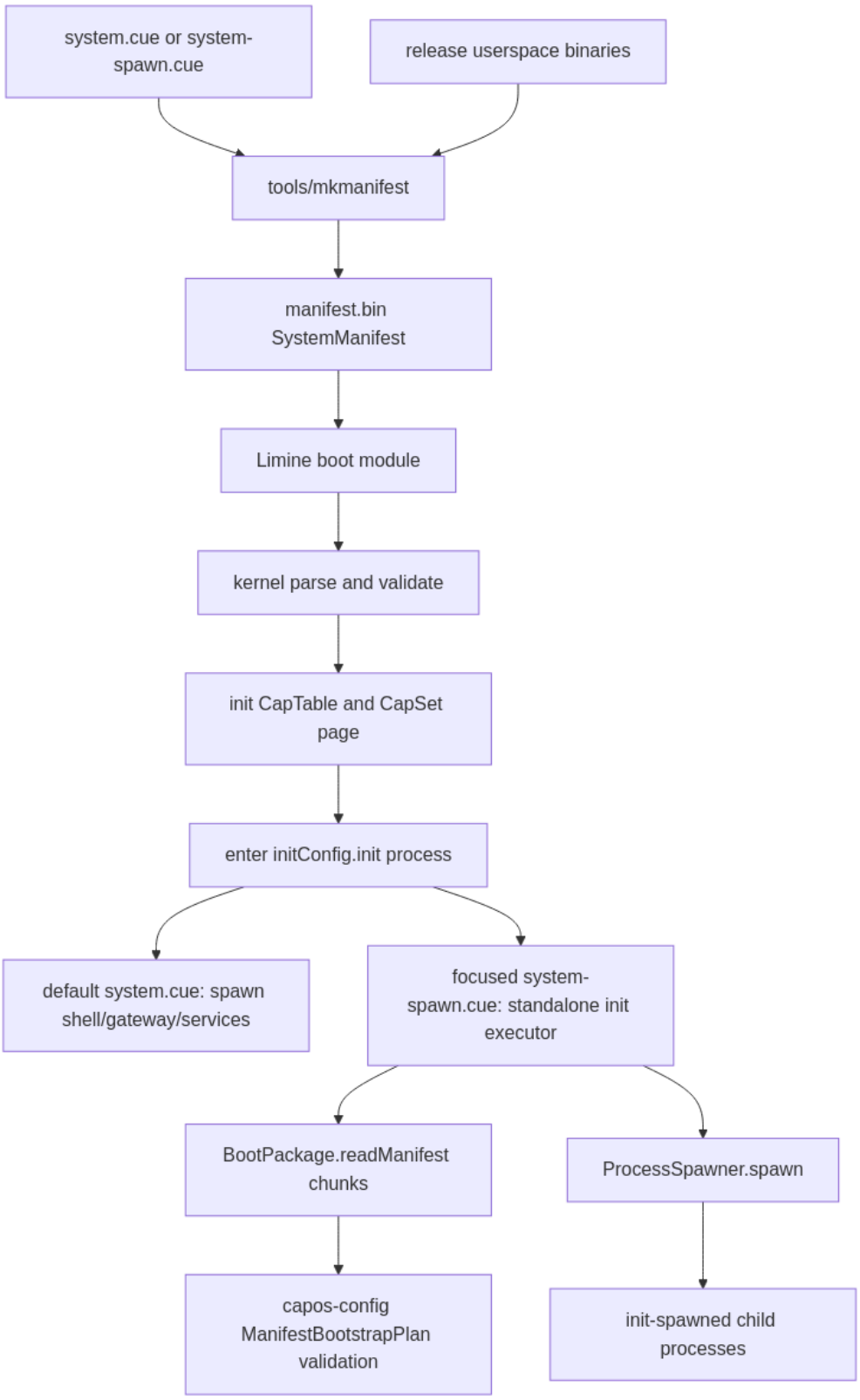

Figure 2: diagram

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, Telnet 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, and VirtualMemory grants mint fresh child-local caps without receiver selectors. 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 `0x60_0000` 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, and ThreadSpawner grants are fresh child-local caps and cannot be badged. 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.

In-Process Threading Contract

This page freezes the 7.1.0 design contract for kernel-managed threads inside one process. The 7.1.1 park authority contract is frozen separately in Park Authority. These pages are the handoff from the single-thread runtime checkpoint to the 7.2 implementation work. The 7.2.3 checkpoint implements the basic single-CPU lifecycle plus private ParkSpace wait/wake.

Scope

The first threading milestone stays single-CPU. It changes the scheduler's unit of execution from process to thread while keeping the process as the authority, address-space, and resource-accounting boundary. SMP, per-CPU run queues, TLB shootdown, SQPOLL, and scheduler-policy services remain later milestones.

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 ring-blocking constraint needed before a sharded or per-thread ring design exists;
- 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:** Capability ring page and ring scratch
  - **Thread-owned state:** 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:** Future scheduling-context binding

The implementation migrates incrementally. The 7.2.0 slice makes each process contain a single initial `Thread`, with saved context, kernel stack, FS base, and blocking state stored on that thread. The 7.2.1 slice changes scheduler-owned queues, current execution, direct IPC handoff, and wake records to generation-checked `ThreadRef` values while still allowing exactly one thread

per process. Later slices widen creation and lifecycle. The single-thread intermediate state must preserve existing QEMU behavior.

Scheduler Contract

`Scheduler` will store 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.

The 7.2.1 checkpoint applies this identity to `Scheduler.current`, run queues, direct IPC targets, Timer sleep waiters, process/terminal waiters, and endpoint caller/receiver wake records while preserving one initial thread per process.

The run queue, `current`, direct IPC target, and blocked waiter scans become 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.

The idle process can remain the existing special user-mode idle process until the kernel-mode/per-CPU idle work lands. It should still be treated as a kernel-owned execution context that cannot block, exit, or hold ordinary caps.

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 stays in the capability-ring transport rather than adding a syscall.

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

- `RDI = arg`;
- `RSI = tid`;
- `RDX = pid`;
- `RCX = RING_VADDR`;
- `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.

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.

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`;
- the single process-ring `cap_enter` waiter is stored as `Option<ThreadRef>`;
- 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 remains process exit. It terminates the whole process, releases the shared capability table, cancels process-owned endpoint state, removes all timer/park/ring waiters for every thread in the process, and completes the parent-facing `ProcessHandle`.

Thread exit is separate and does not add a syscall. The initial implementation adds `ThreadControl.exitThread(code)` as a terminal capability-ring operation on the current thread. 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 `exitThread`, the process exits with that thread's code.

`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 drop follows the existing process-exit rule and happens after another kernel stack is active.

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 threading implementation keeps one capability ring per process. The runtime's single-owner ring-client invariant remains part of the contract: well-formed userspace serializes ring submission and completion matching through `capos-rt`.

The kernel must not admit multiple blocked `cap_enter` waiters on the same process ring in 7.2. If a second thread in the same process asks to block in `cap_enter` while another thread is already the process ring waiter, the kernel returns the current available completion count without blocking that second thread. This preserves the existing syscall return shape and forces the runtime to retry or wait through a runtime-level mutex/park rather than letting two threads race to consume the same CQEs. A thread blocked in `Park` is separate from the process ring's `CapEnter` waiter; it must not consume the one blocked ring-waiter slot.

This constraint avoids freezing a premature per-thread or sharded completion queue ABI. A later runtime/ring milestone can add per-thread rings, completion steering, or a process-level dispatcher thread if measurements show that the single ring waiter is too restrictive.

The full-SMP target is now recorded in Ring v2 For Full SMP: each thread gets its own complete SQ/CQ endpoint, and `cap_enter` waits on the current thread's CQ rather than a shared process CQ. The current process-ring rule remains a compatibility constraint for 7.2 and for any runtime reactor bridge built before Ring v2.

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. ☒ 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 Plan

The first implementation smoke should create two threads in one process, prove they share the address space and CapSet, prove each has an independent FS base, join one thread from another, then let the last thread exit the process. The existing `make run-spawn` path should keep 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.

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, and wake-many;
- 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:

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

`docs/research/` was listed before selecting the milestone. 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 the first object to implement. It will be 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, but VirtualMemory unmap/revoke draining for stale private keys is not implemented yet. Until that lands, the implemented private path is suitable for process lifetime park words and Go runtime bring-up, not for memory regions that are unmapped and reused while waiters may still exist.

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.

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

Validation Plan

The first implementation smoke should create multiple threads in one process, park one or more threads on a userspace park word, wake them through the same ParkSpace, prove timeout and value-mismatch paths, and show that process exit drains pending waits. The runtime smoke should use the same capability through `capos-rt` so future Go work has a direct handoff.

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 compatibility alias while checked-in adventure code migrates. 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:

```
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 Telnet gateway, and resident demo services through ProcessSpawner. 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.

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.

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

Planned Enhancements (from research)

Tracked in `docs/roadmap.md` 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

- **Blocking wait exists, but waits are still process-level.** `cap_enter(min_complete, timeout_ns)` processes pending SQEs and can block the current process until enough CQEs exist or a finite timeout expires. It is not yet a general park/thread wait primitive. Pre-thread measurement now favors a compact capability-authorized park shape over generic Cap'n Proto methods, but blocked/resume timing still depends on in-process threading.
- **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 storage-and-naming-proposal.md "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-by-kernel-transfer path: `CAP_ERR_TRANSFER_NOT_SUPPORTED` (until sideband transfer is enabled),
    - 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 storage-and-naming-proposal.md "Shared Memory for Bulk Data" for the broader interface design.

Future Directions

- **Capability transfer.** Cross-process capability calls already go through the kernel via Endpoint objects with RECV/RETURN SQE opcodes on the existing per-process capability ring (no new syscalls). The remaining transfer work will carry capability references with sideband descriptors and install result caps in the receiver's local table. See storage-and-naming-proposal.md for how this enables `Directory.open()` returning File caps, `Namespace.sub()` returning scoped Namespace caps, etc.
- **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.

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.

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.

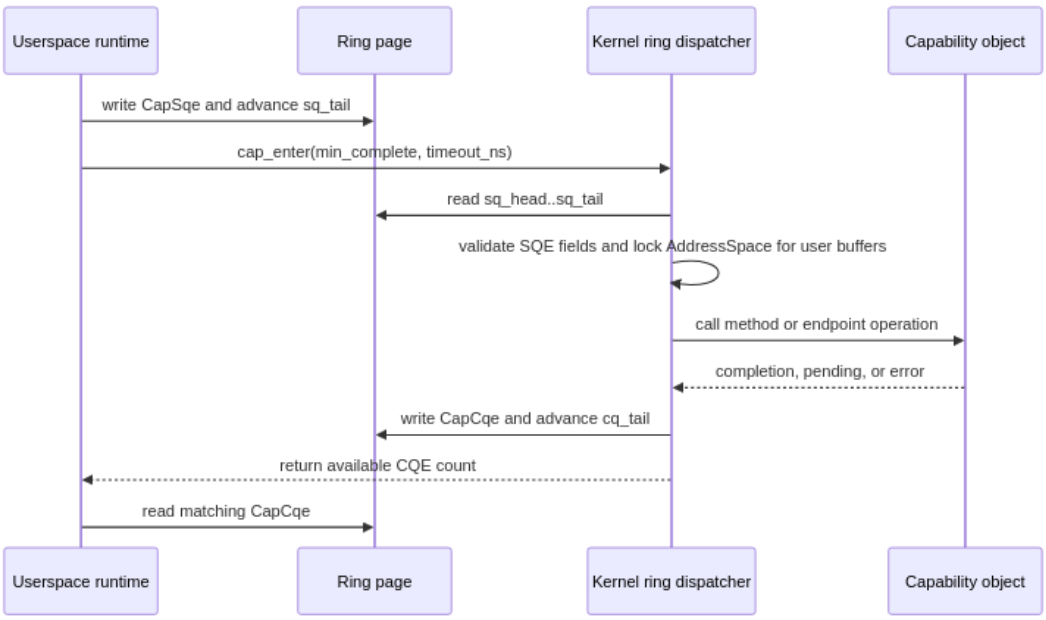

Figure 3: diagram

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_NOP` measures the fixed ring path. `CAP_OP_FINISH` is ABI-reserved and currently returns `CAP_ERR_UNSUPPORTED_OPCODE`.

> **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.

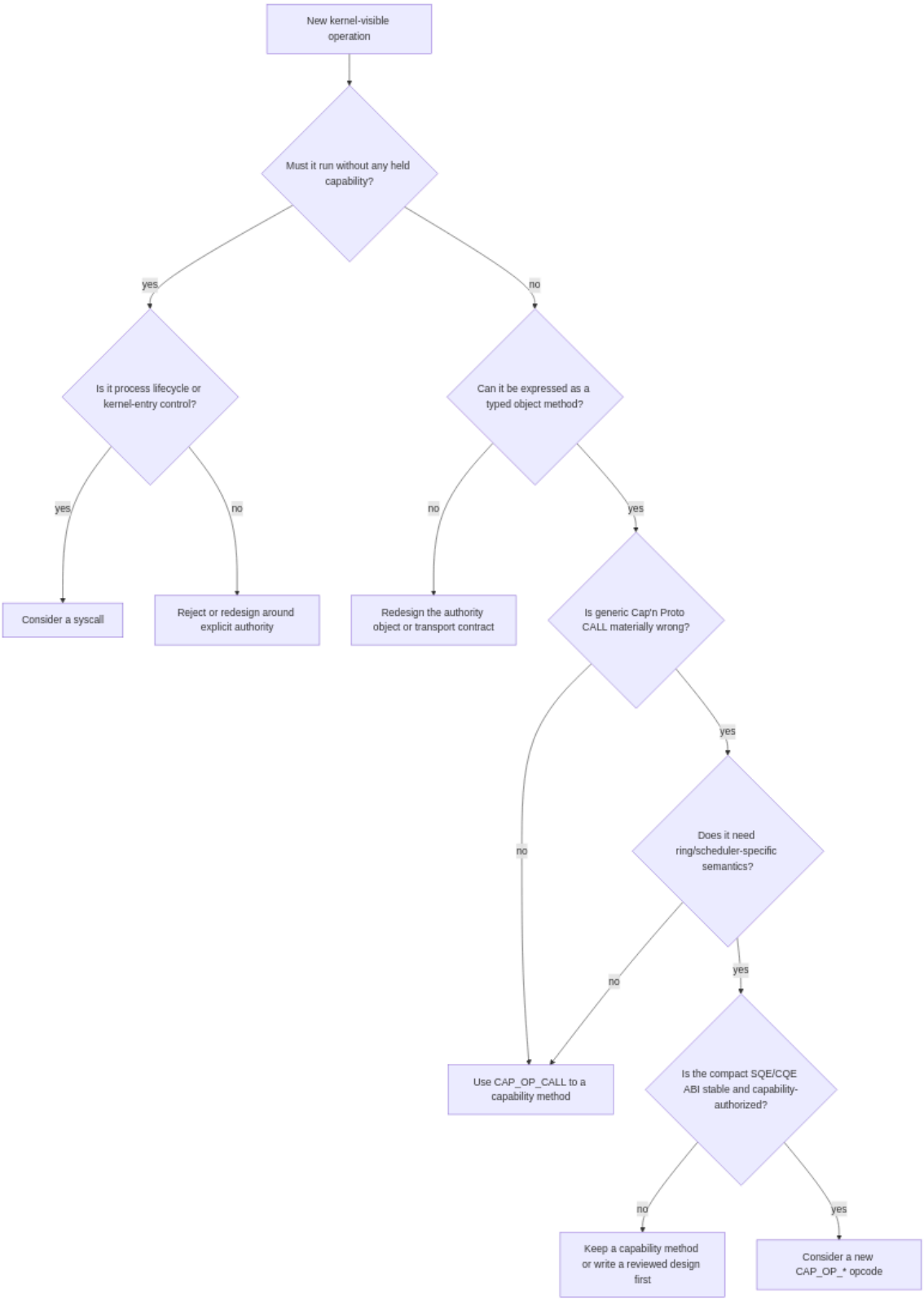

Figure 4: diagram

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 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.

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.

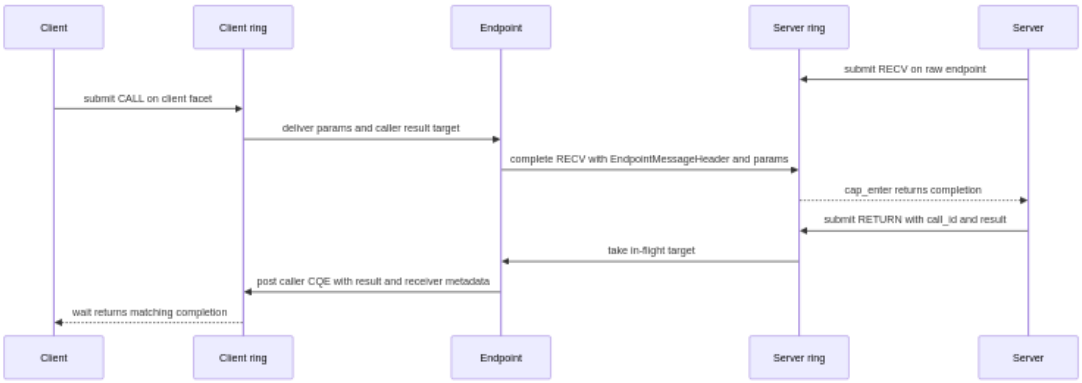

Figure 5: diagram

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 target model. The replacement direction is session-bound invocation context: every normal workload 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 `docs/proposals/session-bound-invocation-context-proposal.md`.

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 should move to 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.

S.9 Design: Authority Graph and Resource Accounting for Transfer

This document defines the concrete S.9 design gate for:

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

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, S.8, and S.12 follow-up work, not requirements for declaring the S.9 design artifact complete. Current capability-semantics follow-up items live in `docs/backlog/stage-6-capability-semantics.md`.

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` with hard limits. Enforcement is fail-closed at reservation time (before side effects).

```
ResourceLedger {
  cap_slots_used / cap_slots_max
  endpoint_queue_used / endpoint_queue_max
  outstanding_calls_used / outstanding_calls_max
  scratch_bytes_used / scratch_bytes_max
  frame_grant_pages_used / frame_grant_pages_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
- `endpoint_queue_max`: 128 messages
- `outstanding_calls_max`: 64
- `scratch_bytes_max`: 256 KiB
- `frame_grant_pages_max`: 4096 pages (16 MiB at 4 KiB pages)
- `log_bytes_per_window`: 64 KiB/sec with 256 KiB burst
- `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:

- S.8 hostile-input tests for quota and invalid-transfer failures.
- 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.
- 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, and `ThreadControl.exitThread` becomes the runtime's current-thread termination path while process `exit` remains whole-process termination.

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.

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.
- 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 processes in a `BTreeMap<Pid, Process>` and runnable execution contexts in a `VecDeque<ThreadRef>`. Each process currently owns one or more `Thread` records; each thread owns its saved CPU context, kernel stack, FS base, and block state. 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 timer tick, the kernel wakes timed-out or satisfied `cap_enter` and park waiters, processes the current thread's process ring in timer mode, saves the current thread context, rotates ready threads, 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 AP cpu=1 is online and its LAPIC timer starts, it is the sole scheduler owner for this proof slice. The BSP yields scheduler-owned work and stays in a kernel `hlt` idle loop with its LAPIC timer active; AP cpu=1 programs its LAPIC timer from the BSP calibration and starts the scheduler from the AP idle loop. Additional APs stay in kernel idle. This proves AP-owned user contexts without running the current process-wide capability ring concurrently on two CPUs.

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 deadline tick. 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.

One capability ring remains process-owned, so the first thread scheduler keeps at most one blocked `cap_enter` waiter per process ring. Run queues, `current`, direct IPC handoff, Timer sleep waiters, process/terminal waiters, endpoint caller/receiver records, and deferred cancellation CQEs store generation-checked `ThreadRef` values. 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 so a parked thread is not the process ring's blocked `cap_enter` waiter. Park timeout and wake completions use reserved CQE credits, then mark 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.

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.

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.

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; later users include 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.

The idle task is currently a user-mode process with one code page and one stack page. It exists because the timer return path assumes interrupts entered from CPL3. A future kernel-mode idle loop requires distinct IRQ entry/restore handling for CPL0 and CPL3 frames.

Future tickless-idle work should not disable the periodic tick directly from this state. The staged design is to split monotonic clocksource reads from clockevent interrupt programming, convert `Timer.sleep`, `cap_enter`, and park timeouts to absolute deadlines, add LAPIC one-shot programming, replace the user-mode idle process with a kernel/per-CPU idle context, then suppress the periodic tick only while no runnable work exists. Generic full-nohz and SQPOLL nohz are 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, and defers final drop until the scheduler is running on another kernel stack.

Invariants

- The idle process 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; until per-CPU run queues land, the scheduler lock remains the authority for current-thread state.
- 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 drop of an exiting process must not occur on its own kernel stack.
- The first thread scheduler must allow at most one blocked `cap_enter` waiter per process ring until a per-thread or sharded ring ABI is designed.
- 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` - process table, thread run queue, blocking, wakeups, Timer sleep waiters, exit, direct IPC target.
- `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, idle process.

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

- Replace the user-mode idle process with a kernel/per-CPU idle context after interrupt restore paths support CPL0 timer entries.
- Split clocksource from clockevent, convert timeout waiters to absolute monotonic deadlines, add a LAPIC one-shot backend, and then implement tickless idle only for no-runnable-work CPU idle. Keep runnable contention on periodic preemption until per-CPU scheduling, accounting, Ring v2, and housekeeping dependencies are explicit.
- 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: per-CPU run queues, reschedule IPIs, and concurrent scheduler-owned work on more than one CPU.
- Add priority or policy scheduling only after the current authority and IPC semantics remain stable.
- Add service restart policy outside the static boot graph.

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

- **Boundary:** Ring 0 to Ring 3
  - **Trust rule:** The kernel trusts no userspace register, pointer, SQE, CapSet, or result buffer field.
  - **Current enforcement:** `kernel/src/arch/x86_64/syscall.rs`, `kernel/src/mem/paging.rs`, `kernel/src/mem/validate.rs`, and `kernel/src/cap/ring.rs` validate syscall arguments, lock process `AddressSpace` state across user-buffer validation/use, check opcodes, and verify capability table lookups before privileged use.
  - **Validation and review source:** `../panic-surface-inventory.md`, `REVIEW.md` (at the repository root)
- **Boundary:** Capability table to kernel object
  - **Trust rule:** A process acts only through a live table-local `CapId` with matching generation and interface.

  - **Current enforcement:** `capos-lib/src/cap_table.rs` owns generation-tagged slots; kernel capability dispatch goes through `CapObject::call`.
  - **Validation and review source:** `cargo test-lib`, QEMU ring and IPC smokes recorded in `REVIEW_FINDINGS.md` (at the repository root)
- **Boundary:** Capability ring shared memory
  - **Trust rule:** Userspace owns SQ writes, but the kernel owns validation, dispatch, completion, and failure semantics.
  - **Current enforcement:** 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 and review source:** `cargo test-ring-loom`, QEMU ring corruption, reserved opcode, fairness, IPC, and transfer smokes
- **Boundary:** 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 its service-visible identity, and shared-service handlers should derive caller identity from live caller-session metadata rather than caller-selected selectors.
  - **Current enforcement:** `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 shell parsing rejects arbitrary badge selection. Chat membership and shared demo endpoint helpers expose caller-session metadata instead of badge identity for normal handlers. Shell-serviced stdio bridges bind active waits to opaque live caller-session metadata, reject mismatched callers, and release transferred caps before rejection returns.
  - **Validation and review source:** `../authority-accounting-transfer-design.md`, open transfer findings in `REVIEW_FINDINGS.md` (at the repository root)
- **Boundary:** Manifest and boot package
  - **Trust rule:** Boot manifest bytes and embedded binaries are untrusted inputs until parsed and validated. Only holders of the read-only BootPackage cap can request chunked manifest bytes; ordinary services receive no default boot-package authority.
  - **Current enforcement:** `tools/mkmanifest` validates the full embedded `initConfig` graph before serialization. Kernel bootstrap validates only `schemaVersion`, `binaries`, `initConfig.init`, and `kernelParams` before loading the single init process, while init BootPackage validation resolves `initConfig.services` graph references before spawning children. `kernel/src/cap/boot_package.rs`, ELF parsing in `capos-lib/src/elf.rs`, and load paths still enforce manifest-read bounds, ELF bounds, load ranges, CapSet layout, and interface IDs at their respective boundaries.
  - **Validation and review source:** `cargo test-config`, `cargo test-mkmanifest`, `cargo test-lib`, manifest and ELF fuzz targets, `make run`
- **Boundary:** 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.

- **Current enforcement:** `ProcessSpawner` validates ELF load, grants, frame exhaustion, parent cap-slot exhaustion, and child-local endpoint creation with parent-only client result facets. Delegated `ClientEndpoint` grants preserve the source identity; explicit different badges and legacy badge-zero relabel encodings fail closed unless the source is an endpoint owner or trusted parent endpoint result cap. Omitted shell badge syntax preserves the source identity. Manifest schema-version guardrails reject unknown manifest vintages before graph validation.
- **Validation and review source:** Spawn QEMU smoke evidence and `REVIEW_FINDINGS.md` (at the repository root)

- **Boundary:** Console authentication and setup
  - **Trust rule:** Console input, password verifiers, setup tokens, and passkey challenge state are hostile or sensitive until a login/session component validates them. Setup tokens are credentials. Challenge state must be bounded, single-use, and expiring.
  - **Current enforcement:** Partial implementation. `Console` remains output-only; the first interactive boundary is a session-scoped `TerminalSession` with bounded `readLine`, `write`, and `writeLine`, one move-only foreground holder, per-call visible/hidden echo, and structured cancellation, with focused QEMU proof in `make run-terminal`. Cancelled reads and owner teardown scrub queued UART input before the next prompt so stale secret bytes do not bleed across sessions; terminal output also requires a live caller session. The bootstrap `CredentialStore` verifies one manifest-supplied Argon2id console password and also supports one bounded volatile RAM-overlay password created by first-boot setup. Authentication is driven by `capos-shell` through `login` and `setup` commands; the default manifest starts that shell as an init-owned service, while focused shell-led smokes still boot it directly as init. There is no separate `ConsoleLogin` service. Focused `make run-login` / `make run-login-setup` proofs cover password login and volatile first-boot setup followed by the operator-bundle upgrade. `CredentialStore` and `SessionManager` hold the only current `EntropySource` authority for salts and session IDs. RAM-only rotate/disable state plus bounded single-use setup-token/challenge state remain future work.
  - **Validation and review source:** `../proposals/boot-to-shell-proposal.md`, boot-to-shell gates in `../../WORKPLAN.md`, `make run-terminal`, `make run-login`, `make run-login-setup`
- **Boundary:** Session authority and audit
  - **Trust rule:** Authenticated sessions must receive only broker-issued narrow caps. Audit output, service logs, terminal output, and failed-auth diagnostics must not disclose secrets or verifier material.
  - **Current enforcement:** Partial implementation. `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, rotating on reboot or endpoint replacement; `AuthorityBroker` validates the session/profile match before minting a bundle; `RestrictedLauncher` returns a shell-scoped path (empty allowlist for anonymous and normal guest profiles, full interactive shell set for operator, and explicitly named one-binary guest proof profiles where configured) instead of `BootPackage` or broad `ProcessSpawner` authority. Focused QEMU proofs show `capos-shell` starting with an anonymous session, then

upgrading to an operator bundle through `login` or `setup` while preserving ungranted-child terminal isolation, and show that manifests without a guest seed cannot mint guest sessions. `CredentialStore.verify` still returns only generic success/denied/unavailable outcomes and the shell avoids password echo. Audit records, stable service-audit identity across endpoint replacement, opaque record IDs, broader policy evaluation, and web-terminal origin/RP-ID validation remain future work.

  - **Validation and review source:** `../proposals/user-identity-and-policy-proposal.md`, `../proposals/boot-to-shell-proposal.md`, `make run-login`, `make run-login-setup`, future auth/session hostile-input tests

- **Boundary:** 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.
  - **Current enforcement:** Partial implementation with schema stubs, one development-only host-key proof, a manifest-seeded `AuthorizedKeyStore` proof, a public-key session bridge proof, an unsupported-feature policy proof, a manifest-declared restricted shell launcher proof, and a bounded terminal-host wiring proof over host-local plain TCP. The SSH shell proposal keeps SSH transport authority in `SshGateway`: host-key signing uses sign-only `SshHostKey`, authorized SSH public keys map through `AuthorizedKeyStore` to principals without granting shell authority directly, scoped listen authority uses `TcpListenAuthority`, unsupported forwarding/subsystem/agent features fail closed, and the spawned shell receives only an `SshTerminalFactory`-produced `TerminalSession` plus the normal scoped session bundle through `RestrictedShellLauncher`. The current `ssh_development_host_key` kernel source requires an explicitly labeled non-production manifest fixture, exposes public host-key metadata, and signs bounded `ssh-ed25519` exchange hashes with the manifest seed for QEMU proof only; wrong algorithms, malformed seeds, mismatched public keys, and oversized exchange hashes fail closed, and private seed bytes are not logged or returned. The current `AuthorizedKeyStore` path accepts only enabled configured public `ssh-ed25519` keys mapped to seed accounts and allowed profiles, returns generic denials otherwise, and `SessionManager.sshPublicKey` can mint a `publicKey` `UserSession` from a matching configured key plus bounded fixture auth bytes/signature; this is not yet a full SSH transport transcript or channel-binding proof. The current `capos-config::ssh_policy` surface allows public-key auth, one session channel, PTY, window-change, and one shell request while denying disabled password auth, exec, subsystem/SFTP, direct and reverse TCP forwarding, agent forwarding, X11 forwarding, environment import, second session-channel opens, and multiple shell channels with stable protocol response classes and audit reason codes; it does not carry command text, environment values, key material, or terminal content. Password auth has no allow switch until the gateway wires a real verifier/backoff path. The current `restricted_shell_launcher` kernel source launches only `capos-shell`, validates the requested profile against the supplied `UserSession`, injects supplied terminal/session caps and child-local stdio, accepts only named capability-sourced shell startup pass-through grants, rejects raw ProcessSpawner/network/TCP/key/SSH authority, and returns only the child `ProcessHandle`; the bounded terminal-host proof wires it to a host-local accepted socket

through `TcpSocket.intoTerminalSession` without exposing raw transport/key authority to the shell. The native shell uses the supplied session when present and otherwise preserves the local-console anonymous startup path.
    - **Validation and review source:** `../proposals/ssh-shell-proposal.md`, `../backlog/runtime-network-shell.md`, `make run-ssh-host-key`, `make run-ssh-authorized-key`, `make run-ssh-public-key-auth`, `make run-ssh-feature-policy`, `make run-restricted-shell-launcher`, `make run-ssh-gateway-terminal-host`
- **Boundary:** Identity metadata and account records
    - **Trust rule:** Users, principals, accounts, sessions, roles, and profile names are policy metadata. They must not be treated as kernel subjects, ambient authority, or substitutes for held capabilities.
    - **Current enforcement:** Planned account records and policy/resource profiles are broker inputs only. A session may receive capabilities after authentication and policy evaluation; a principal or account record does not run or call the kernel. Durable local account-store behavior, profile persistence, rollback checks, and quota enforcement remain future work.
    - **Validation and review source:** `../backlog/local-users-management.md`, `../proposals/user-identity-and-policy-proposal.md`
- **Boundary:** Host tools and filesystem
    - **Trust rule:** Manifest/config input must not escape intended source directories or invoke unconstrained host commands.
    - **Current enforcement:** `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 and review source:** `../trusted-build-inputs.md`, `make generated-code-check`, `make dependency-policy-check`
- **Boundary:** Generated code and schema
    - **Trust rule:** Schema, generated bindings, and no_std patches are trusted build inputs.
    - **Current enforcement:** `schema/capos.capnp`, build scripts, `tools/generated/capos_capnp.rs`, and `tools/check-generated-capnp.sh` make generated-code drift review-visible.
    - **Validation and review source:** `../trusted-build-inputs.md`, `make generated-code-check`
- **Boundary:** Device DMA and MMIO
    - **Trust rule:** Current userspace receives no raw DMA buffer, device physical address, virtqueue pointer, or BAR mapping.
    - **Current enforcement:** The QEMU virtio-net path is allowed only through kernel-owned bounce buffers until typed `DMAPool`, `DeviceMmio`, and `Interrupt` capabilities exist.
    - **Validation and review source:** `../dma-isolation-design.md`, `make run-net`
- **Boundary:** Panic and emergency paths
    - **Trust rule:** Hostile input should produce controlled errors, not panic, allocate unexpectedly, or expose stale state.
    - **Current enforcement:** Ring dispatch is mostly controlled-error; remaining panic surfaces are classified by reachability and tracked as hardening work.

- **Validation and review source:** `../panic-surface-inventory.md`, `REVIEW.md` (at the repository root)

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 advance boot-to-shell login implementation past design stubs without hostile-input tests for bounded console 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:** 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 content CUE-to-Rust freshness, locked generator dependencies, and checked-in baseline drift.
- **Scope:** Dependency policy
  - **Command:** `make dependency-policy-check`
  - **What it checks:** cargo-deny and cargo-audit policy across root and standalone lockfiles.
- **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:** 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 shell, resident demo services, and the host-local Telnet gateway with the terminal UART on stdio and 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:** 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, ARP/ICMP, and host-backed smoltcp TCP proof.

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 `REVIEW_FINDINGS.md` (at the repository root) 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 in `REVIEW_FINDINGS.md` (at the repository root) 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.
- **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.md` 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.md`.
- **Claim type:** Panic-surface hardening
  - **Required evidence:** Updated `../panic-surface-inventory.md` 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 shell launch, `ConsoleLogin` terminal release after handoff, 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`, and arbitrary ELF parser input. Run fuzzers when a change alters those parsers, schema shape, or 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`.

Loom coverage belongs in shared ring logic. Extend `cargo test-ring-loom` when SQ/CQ ownership, ordering, corruption recovery, or wake semantics change.

Documentation Sources

- `REVIEW.md` (at the repository root): rules for security, unsafe code, capability invariants, resource accounting, and emergency paths.
- `REVIEW_FINDINGS.md` (at the repository root): open remediation backlog and latest verification records.
- `../trusted-build-inputs.md`: trusted compiler, generated-code, dependency, bootloader, manifest, and host-tool inputs.
- `../panic-surface-inventory.md`: classified panic-like surfaces and commands used to generate the inventory.
- `../authority-accounting-transfer-design.md`: 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 S.10.0 inventory, records the S.10.2 generated-code drift check, and now also records the S.10.3 dependency policy plus the shared no_std generated-code patch helper.

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:** Floating `nightly` channel with target triples and the `rust-src` component required by custom-target `-Zbuild-std` userspace builds. The Kani job remains pinned to the Kani-compatible nightly bundle installed by `cargo kani setup`.

  - **Drift-review status:** No repo-visible date or hash audit for normal Rust builds. The current local resolver reported `rustc 1.97.0-nightly (66da6cae1 2026-04-20)`.
- **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
  - **Current source:** `init/Cargo.lock`, `demos/Cargo.lock`, `tools/adventure-content-gen/Cargo.lock`, `tools/mkmanifest/Cargo.lock`, `tools/ringtap-viewer/Cargo.lock`, `capos-rt/Cargo.lock`, `shell/Cargo.lock`, `fuzz/Cargo.lock`
  - **Pinning status:** Each standalone workspace has its own lockfile.
  - **Drift-review status:** `make dependency-policy-check` runs the shared deny/audit baseline against every standalone manifest and lockfile. Cross-workspace version drift remains review-visible and intentional where lockfiles differ.
- **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 a shared `.capos-tools/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:** 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` expects 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 a minimal git repository for tools that derive the shared `.capos-tools` path from git metadata, 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. 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:** 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 clone-shared pinned CUE path 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:** 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`
  - **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.
  - **Drift-review status:** Build rerun boundaries are explicit; generated link args are not independently audited.
- **Input:** CUE manifest compiler
  - **Current source:** `Makefile:35-50`, `tools/mkmanifest/src/main.rs`, `tools/mkmanifest/src/lib.rs`, `.github/workflows/ci.yml`
  - **Pinning status:** `make cue-ensure` installs `cuelang.org/go/cmd/cue` pinned to `v0.16.0` into `$(CAPOS_TOOLS_ROOT)/cue/0.16.0/bin/cue`. `CAPOS_TOOLS_ROOT` defaults to `$HOME/.capos-tools` (per-user shared cache); operators may override it explicitly.
  - **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)`, `tools/mkmanifest/src/lib.rs` `cue_export_args` / `cue_tags_from_env`, `target/.cue-tags.<manifest>`
  - **Pinning status:** `make run` sets `CAPOS_CUE_TAGS=user=$(USER)`. `mkmanifest` reads `CAPOS_CUE_TAGS` (and `--tag key=value` CLI repeats) and forwards each entry to `cue export --inject`. The `target/.cue-tags.<manifest-bin>` sentinel records the active tag value via a `FORCE`-prereq rule that touches the file only when content differs, so a tag change invalidates the cached `manifest.bin`.
  - **Drift-review status:** The injected `user` value reaches the manifest via `system.cue`'s `_user: string \| *"operator" @tag(user)` and surfaces as the host-operator seed account `displayName`. Smoke targets leave the tag unset, keeping `principal=operator` invariants stable.
- **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 the clone-shared `.capos-tools` path. `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`
  - **Pinning status:** GitHub release asset for Typst `v0.14.2` is pinned by version and SHA-256 under the clone-shared `.capos-tools/typst/0.14.2` path, 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 as Pandoc's PDF engine for `target/docs-site/capos-docs.pdf`. Generated PDFs are not checked in; source `main.typ`, `references.bib`, and documentation inputs are checked in.

- **Input:** Mermaid CLI and Pandoc documentation PDF converter
  - **Current source:** `Makefile PANDOC_VERSION`, `.node-version`, `package.json`, `package-lock.json`, `tools/docs-bundle.js`, `tools/mermaid-puppeteer-config.json`, `tools/normalize-pandoc-typst-media.js`
  - **Pinning status:** GitHub release asset for Pandoc `3.9.0.2` (`pandoc-3.9.0.2-linux-amd64.tar.gz`) is pinned by version and SHA-256 under `.capos-tools/pandoc/3.9.0.2`. `pandoc-ensure` verifies the tarball checksum and the binary's reported version before PDF generation. Node is pinned to `22.16.0` for Cloudflare Pages and local Node dependency installs. `package-lock.json` pins `@mermaid-js/mermaid-cli` and its Puppeteer dependency tree; `tools/mermaid-puppeteer-config.json` disables the browser sandbox for local and gVisor build containers. `tools/docs-bundle.js` generates `docs/capos-docs.md` from checked-in docs. Mermaid CLI rasterizes diagrams to PNG before Pandoc emits Typst, and the path normalizer constrains image references to `target/docs-bundle`.
  - **Drift-review status:** `make docs-pdf` converts generated `docs/capos-docs.md` to `target/docs-bundle/capos-docs.pdf` with pinned Typst as the PDF compiler; `make docs` copies that generated PDF to `target/docs-site/capos-docs.pdf` for Cloudflare Pages publication. Generated Markdown and PDF files are ignored build artifacts, not tracked source.
- **Input:** QEMU and firmware
  - **Current source:** `Makefile:67-83`
  - **Pinning status:** Host-installed `qemu-system-x86_64`; OVMF path is hard-coded for UEFI.
  - **Drift-review status:** No repo-visible version or firmware checksum. Current local host reported QEMU `10.2.2`.
- **Input:** ISO and host filesystem tools
  - **Current source:** `Makefile:51-65`
  - **Pinning status:** Host-installed `xorriso`, `sha256sum`, `git`, `make`, shell utilities.
  - **Drift-review status:** No version capture except ad hoc local inspection.
- **Input:** Boot manifest and embedded binaries
  - **Current source:** `system.cue:1-144`, `tools/mkmanifest/src/lib.rs:82-115`, `Makefile:28-29`, `Makefile:51-65`
  - **Pinning status:** Source manifest is checked in; embedded ELF payloads are build artifacts.
  - **Drift-review status:** Manifest validation checks references and path containment, but final `manifest.bin` is generated and not checksum-recorded.
- **Input:** Build downloads
  - **Current source:** `Makefile`, Cargo lockfiles, `rust-toolchain.toml`
  - **Pinning status:** Limine and documentation tool tarballs are explicitly fetched; Cargo, Go, and rustup downloads are implicit when caches/toolchains are absent.
  - **Drift-review status:** Limine artifacts and documentation tool tarballs are verified. Cargo, Go, and rustup downloads rely on upstream tooling and lockfiles, with no repo policy.

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

- 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 `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 S.10.3 policy

- Continue Rust toolchain pinning work (date/hash pin, reproducible host compiler inputs) as a separate build-reproducibility task.
- 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.
- Decide whether final ISO/payload hashes become policy-grade inputs in production-hardening stages.

Production hardening must treat the following as unresolved supply-chain gates, not as cosmetic reproducibility work:

```
rust nightly date/hash pin
qemu-system-x86_64 version capture or pinned runner image
xorriso version capture or pinned runner image
OVMF firmware path/version/hash recording for UEFI boots
final ISO checksum recording
manifest.bin checksum recording
embedded ELF payload checksum recording
```

Until those gates land, generated ISO/manifest/payload artifacts 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 `<git-common-dir>/../.capos-tools/limine/<commit>`), shared across linked worktrees with the rest of the `.capos-tools` toolchain.

`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"`
- `targets = ["x86_64-unknown-none", "aarch64-unknown-none"]`
- `components = ["rust-src"]`

This is a floating channel pin, not a reproducible toolchain pin. A future `rustup` resolution can move the compiler even when the repository is unchanged. The current local host resolved to:

- `cargo 1.97.0-nightly (7ecf0285e 2026-04-18)`
- `rustc 1.97.0-nightly (66da6cae1 2026-04-20)`
- host target `x86_64-unknown-linux-gnu`

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. CI host-baseline and optional QEMU jobs install floating `nightly` with `rust-src` for that build surface; the Kani job stays on its verifier-pinned toolchain because it does not run the custom-target userspace build aliases.

Remaining reproducibility gap: pin the nightly by date or exact toolchain hash. Until then, compiler drift can change codegen, linking, lints, and generated bindings without a repository diff.

Cargo Dependencies

The root workspace members are `capos-abi`, `capos-config`, `capos-lib`, `kernel`, and the host-only `tools/capnp-build` build-support crate. `Cargo.toml` keeps default members limited to `capos-config`, `capos-lib`, and `kernel` so ordinary root bare-metal builds do not build the host helper as a target package. `init/`, `demos/`, `tools/mkmanifest/`, `tools/ringtap-viewer/`, `capos-rt/`, `shell/`, 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:7`
    - **Root lock resolution:** `0.6.3` in `Cargo.lock`
- **Dependency:** `spin`
    - **Manifest references:** `kernel/Cargo.toml:8`
    - **Root lock resolution:** `0.9.8` in `Cargo.lock`
- **Dependency:** `x86_64`
    - **Manifest references:** `kernel/Cargo.toml:9`
    - **Root lock resolution:** `0.15.4` in `Cargo.lock`
- **Dependency:** `linked_list_allocator`
    - **Manifest references:** `kernel/Cargo.toml:10`
    - **Root lock resolution:** `0.10.6` in `Cargo.lock`
- **Dependency:** `loom`
    - **Manifest references:** `capos-config/Cargo.toml:17`
    - **Root lock resolution:** `0.7.2` in `Cargo.lock`
- **Dependency:** `proptest`
    - **Manifest references:** `capos-lib/Cargo.toml:9-10`
    - **Root lock resolution:** `1.11.0` in `Cargo.lock`

Standalone lockfile drift observed during this inventory:

- **Lockfile:** `init/Cargo.lock`
    - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.5`
- **Lockfile:** `demos/Cargo.lock`
    - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`
- **Lockfile:** `tools/mkmanifest/Cargo.lock`
    - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `serde_json 1.0.149`
- **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:** `shell/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `linked_list_allocator 0.10.6`; no Argon2 because it uses baseline `capos-config`
- **Lockfile:** `fuzz/Cargo.lock`
  - **Notable direct/runtime resolution:** `capnp 0.25.4`, `capnpc 0.25.3`, `libfuzzer-sys 0.4.12`

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.

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.
- `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.

S.10.2 generated-code drift check:

- `make generated-code-check` runs both `tools/check-generated-capnp.sh` and `tools/check-generated-adventure-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 `2cdcfc383dd4d07c62758c0823dcf242bcb04a4d0f0bec843e20de79c2e7523b`.

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 clone-shared `.capos-tools/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.

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.

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.

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 clone-shared pinned CUE compiler from git state, 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 `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 (shared `.capos-tools/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:** Rust toolchain, targets, and `rust-src`
  - **Producer:** `rustup` from `rust-toolchain.toml` when absent
  - **Pinning/drift status:** Floating nightly channel; `rust-src` is declared for custom-target `-Zbuild-std` userspace builds.
- **Artifact:** `target/` kernel and host artifacts
  - **Producer:** Cargo
  - **Pinning/drift status:** Generated, not checked in.
- **Artifact:** `init/target/`, `demos/target/`, and `capos-rt/target/` ELFs
  - **Producer:** Cargo standalone builds
  - **Pinning/drift status:** Generated, embedded into `manifest.bin` where referenced; no final payload checksums recorded in source.
- **Artifact:** `target/x86_64-unknown-capos/`, `init/target/x86_64-unknown-capos/`, `demos/target/x86_64-unknown-capos/`, `shell/target/x86_64-unknown-capos/`, and `capos-rt/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 and the `capos-rt` smoke binary.

- **Artifact:** `manifest.bin`
  - **Producer:** `tools/mkmanifest`
  - **Pinning/drift status:** Generated from `system.cue` plus ELF payloads; not checked in.
- **Artifact:** `iso_root/` and `capos.iso`
  - **Producer:** Makefile, xorriso, Limine installer
  - **Pinning/drift status:** Generated and gitignored; Limine inputs verified, full ISO checksum not source-recorded.

Remaining gaps for S.10.2/S.10.3:

- Decide whether CI should record or compare hashes for `manifest.bin`, embedded ELF payloads, or the final ISO for reproducible-build tracking.
- Pin or record `xorriso`, `qemu-system-x86_64`, OVMF firmware, and other host tools used by build and boot verification with the same strictness as `capnp` and `cue`.
- Decide whether CI should record the pinned `cue export` JSON or final `manifest.bin` bytes if manifest reproducibility becomes release-critical.

Host Tools

Current local host versions observed during this inventory:

- **Tool:** `capnp`
  - **Observed version:** `1.2.0`
  - **Build role:** Repo-local schema compiler built by `make capnp-ensure` from a SHA-256-pinned official source tarball into the shared `.capos-tools` cache for this clone.
- **Tool:** `cue`
  - **Observed version:** `v0.16.0`
  - **Build role:** Repo-local manifest compiler installed by `make cue-ensure` into the shared `.capos-tools` cache for this clone.
- **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 environment observations, not repository pins. `make run-uefi` also trusts `/usr/share/edk2/x64/OVMF.4m.fd` (`Makefile:82-83`) without a checksum.

Remaining gap for S.10.3: decide the minimum supported host tool versions and whether they are enforced by CI, a container/devshell, or explicit preflight checks.

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 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 one IPC queue invariant.

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_recvs.pop_front().unwrap()`
  - **Reachability:** Cross-process CALL delivered to pending RECV
  - **Classification:** untrusted-input reachable, guarded
  - **Notes:** Guarded by `!inner.pending_recvs.is_empty()` under the same lock. It is still on an IPC path driven by service SQEs, so S.8.4 should convert this to an explicit error/rollback path if panic-free IPC is required.
- **Location:** `kernel/src/cap/endpoint.rs` `endpoint_restore_recv_front`
  - **Surface:** unchecked `push_front` growth

- **Reachability:** IPC rollback path
- **Classification:** untrusted-input reachable, non-panic today
- **Notes:** `VecDeque::push_front` can allocate/panic if spare capacity assumptions are broken. Current pending recv queue is pre-reserved and bounded on normal insert; rollback paths should keep the bound explicit when hardened.

Scheduler And Process Lifecycle

- **Location:** `kernel/src/sched.rs` `init_idle`
  - **Surface:** `Process::new_idle().expect("failed to create idle process")`
  - **Reachability:** Boot scheduler init
  - **Classification:** boot-fatal
  - **Notes:** Idle creation OOM/mapping failure panics before services run.
- **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

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 programs physical addresses
network stack receives copied packet bytes
userspace receives no DMA buffer capability
userspace receives no physical address
userspace receives no virtqueue pointer
userspace receives no BAR mapping
```

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.

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.

All three object families carry generation or epoch identity. A userspace handle is valid only while its recorded generation still matches the owning kernel record:

```
struct DmaHandle {
    pool_id: u32,
    slot: u32,
    generation: u32,
}

struct MmioHandle {
    device_id: u32,
    bar: u8,
    generation: u32,
}

struct InterruptHandle {
    source_id: u32,
    generation: u32,
}
```

Generation mismatch is a hard closed result. Revocation, device reset, reassignment, and pool reuse must all advance the relevant generation before any new owner can receive a handle. A generation value must never wrap back into valid authority: implementations either use a non-wrapping epoch width for the object lifetime, or retire the slot/source permanently when the generation space is exhausted.

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.

`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.

- **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.

`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.
- **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.
- **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 S.11.2. The gate is only open when all rows below are implemented and demonstrated.

- **Gate item:** S.11.2.0 DMA-objected 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:** 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.
- **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:** 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.
- **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.

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.

S.11.2 Decision Record

S.11.2 is not complete until the kernel has a dedicated device manager object model that can produce, transfer, and revoke `DMAPool`, `DeviceMmio`, and `Interrupt` in a single ownership transaction for a driver process.

Current status: transition remains **blocked** pending implementation of the conditions above.

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.

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: 2026-04-29 11:52 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 still go into `REVIEW_FINDINGS.md`; 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 selected 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, and terminal/stdio bridge liveness guards. Remaining cleanup is focused on peer-owned Adventure/shared-service migration, retiring remaining normal user-facing receiver-selector syntax, and final full-gate verification before treating the Tier-1 paper prerequisite as closed.

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 remains the visible tail of the selected migration.
- **Owner.** `docs/proposals/session-bound-invocation-context-proposal.md`, `docs/backlog/stage-6-capability-semantics.md`.
- **Gap.** Finish Adventure/shared-service migration and 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/oom-and-swap-proposal.md`, `docs/proposals/user-identity-and-policy-proposal.md`, `docs/proposals/system-monitoring-proposal.md`.
- **Gap.** CPU accounting/scheduling contexts, log volume accounting, per-service fairness, donation semantics, and unified resource bundles for guest/anonymous/external/service principals are not implemented.

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

- **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/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 waiters on reusable unmapped addresses remain restricted by documentation until VM-unmap stale-key cleanup and tests land.

R8 – Networking lives inside the kernel TCB

- **State.** virtio-net, ARP/ICMP, the smoltcp runtime, and the TCP `CapObject`s currently run in the kernel address space behind capability objects. The Telnet and SSH terminal-host proofs are built on this path.
- **Owner.** `docs/proposals/networking-proposal.md`, `docs/dma-isolation-design.md`, `docs/backlog/runtime-network-shell.md`.
- **Gap.** Userspace NIC driver and userspace TCP stack are Phase C / future work; until then the network stack temporarily expands the kernel TCB against the long-term service-decomposition direction.

R9 – DMA threat model assumes cooperative virtio

- **State.** `docs/dma-isolation-design.md` records that the first virtio-net smoke uses kernel-owned bounce buffers, does not expose userspace DMA buffers or physical addresses, and explicitly assumes a non-hostile QEMU virtio device. Without an IOMMU a malicious bus master can DMA arbitrary RAM.
- **Owner.** `docs/dma-isolation-design.md`, `docs/proposals/networking-proposal.md`, `docs/proposals/cloud-deployment-proposal.md`.
- **Gap.** `DMAPool` / `DeviceMmio` / `Interrupt` userspace-driver gating, and IOMMU-backed isolation for production hardware, are not implemented.

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/backlog/runtime-network-shell.md`, `WORKPLAN.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.

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.

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, and the Kani toolchain bundle are pinned. The Rust nightly toolchain is a floating channel without a date or hash pin; `xorriso`, `qemu-system-x86_64`, and OVMF firmware are observed-not-pinned.
- **Owner.** `docs/trusted-build-inputs.md`.
- **Gap.** Reproducible-build pinning for the Rust nightly (date/hash), host ISO/firmware tools, and final ISO/payload checksums is unfinished; the document already lists these as open S.10.x gaps.

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.
- **Owner.** `docs/proposals/user-identity-and-policy-proposal.md`, `docs/backlog/local-users-management.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.

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.** Schema interface IDs / method IDs / struct layout are reviewed for stability via `make generated-code-check` and `tools/check-generated-capnp.sh`. There is no published ABI-compatibility guarantee window yet.
- **Tracker.** `docs/proposals/error-handling-proposal.md`, `docs/trusted-build-inputs.md`, `schema/capos.capnp`.
- **Status.** Open. A formal ABI-evolution policy (interface-id stability, reserved-field semantics, deprecation window) needs to be authored before external consumers depend on the schema.

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; `WORKPLAN.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. Revocation trees, leases, and supervisor-owned-cap patterns are proposal-shaped.
- **Tracker.** `docs/proposals/service-architecture-proposal.md`, `docs/proposals/session-bound-invocation-context-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.

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.** Round-robin is the current scheduler. Donation, priority inheritance, fixed budgets, and explicit scheduling capabilities are proposed but not implemented.
- **Tracker.** `docs/proposals/smp-proposal.md`, `docs/proposals/resource-accounting-proposal.md`, `docs/architecture/scheduling.md`.
- **Status.** Open. Pick a CPU accounting model before Multi-Process SMP Concurrency or In-Process Threading Scalability milestones land.

Q10 – IOMMU requirement for userspace networking
- **Current answer.** `docs/dma-isolation-design.md` keeps the kernel-owned bounce-buffer model for QEMU virtio and explicitly defers an IOMMU requirement to the userspace driver gate.
- **Tracker.** `docs/dma-isolation-design.md`, `docs/proposals/networking-proposal.md`, `docs/proposals/cloud-deployment-proposal.md`.
- **Status.** Open. The production answer (IOMMU-required vs continued bounce buffers vs both) depends on platform support and will be decided at the userspace driver gate.

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 for the gates listed in `REVIEW_FINDINGS.md`.

Q15 – Language runtimes integration model

- **Current answer.** `capos-rt` is the canonical no_std Rust runtime; Go, Lua, WASI, and POSIX adapters are proposal-shaped and currently framed as separate runtimes consuming generated capnp clients, not as a shared C/WASI ABI layer.
- **Tracker.** `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 clients.

Roadmap

Long-term direction for capOS. Keep this file outcome-oriented. Detailed task decomposition belongs in `docs/backlog/`; current execution order belongs in `WORKPLAN.md`; completed milestone/review reports belong in `docs/changelog.md`.

Current Direction

Current selected milestone: **Session-Bound Invocation Context**.

The visible goal is in cleanup: the core gates are landed, and remaining work is to finish the peer-owned adventure shared-service migration and final full-gate verification. Implemented pieces include the process-session invariant, endpoint caller-session metadata, stale normal endpoint rejection, transfer scopes, field-granular disclosure gating, session expiry for broker-issued shell bundle caps, guest bundle narrowing, chat session-keyed membership, terminal and stdio bridge live-session guards, keyed service-scoped caller references, and the session-context proof that distinct endpoint service scopes derive distinct opaque caller-session reference tuples. The milestone still replaces caller-selected service-visible identity without continuing the service-object identity migration.

The prior core service-object routing/lifecycle subproof landed in commit `a4655f0` at `2026-04-28 14:10 UTC`: it proves trusted service-object minting, generation-checked receiver cookies, copy/move IPC transfer, nested spawn delegation, close/revoke rejection, and stale-cookie rejection after record reuse. That proof remains historical low-level coverage. The active milestone does not continue subject/proof root opening or shared-service service-object migration.

This milestone intentionally precedes more remote shell work. The SSH Shell Gateway remains a planned Stage 7 shell/networking milestone, but safe network-backed shell delegation depends on the same one-session-per-process and privacy-preserving endpoint session model. The SSH version-exchange checkpoint lives on `workplan/ssh-version-exchange-recovery` and still requires QEMU harness review before merge.

Details: - `WORKPLAN.md` - `docs/backlog/session-bound-invocation-context.md` - `docs/backlog/service-object-identity-migration.md` (superseded) - `docs/backlog/stage-6-capability-semantics.md` - `docs/proposals/session-bound-invocation-context-proposal.md` - `docs/proposals/service-object-capabilities-proposal.md` (superseded) - `docs/proposals/user-identity-and-policy-proposal.md` - `docs/proposals/oidc-and-oauth2-proposal.md` - `docs/backlog/local-users-management.md`

Whitepaper Track

A future capOS whitepaper / technical report consumes – not duplicates – work from the other tracks. The plan, outline, and live evidence-gap log remain in `docs/paper/` (`plan.md`, `outline.md`,

evidence-gaps.md). The paper itself is a Typst project at papers/schema-as-abi/ and is built via make paper.

The paper's Tier-1 evidence requirements pull these existing items into explicit paper-supporting roles. They are not new tracks; they are the selection lens this track applies:

- Stage 6 session-bound invocation context migration (closes the "interface IS the permission" claim).
- A measurement harness over make run-measure producing reproducible ring throughput, cap_enter latency, IPC handoff, and schema-dispatch numbers (closes the ring-as-sufficient-boundary claim).
- A paper-scoped persistence proof-of-concept narrower than the storage proposal (closes the wire-format-enables-persistence claim).
- A paper-scoped network-transparency proof-of-concept narrower than the general networking proposal (closes the wire-format-enables-network-transparency claim).
- At least one of {promise pipelining, notification objects} (closes capnp-rpc-shaped composition beyond CALL/RECV).

Tier-2 strengtheners: ring-protocol Kani proof, full concurrent SMP scheduling, end-to-end SSH Shell Gateway, one non-toy demo beyond Adventure or First Chat.

Out of scope for the first paper (acknowledge in Future Work only): aarch64, GPU, live upgrade, formal MAC/MIC, Go/WASI, cloud metadata, production volume encryption.

When workplan slices close a paper-evidence gap they should reference docs/paper/evidence-gaps.md and update it in the same task, including the matching #todo block in papers/schema-as-abi/main.typ. A structural pre-evidence draft already exists at papers/schema-as-abi/main.typ; the abstract, the Evaluation section, the Conclusion, and any contribution claim that depends on missing Tier-1 evidence stay deferred until that evidence lands. New paper content that does not depend on missing artifacts may be drafted at any time and lives next to the existing #todo blocks.

Completed Foundation

- **Stage 0: Foundations**: bitmap physical frame allocator, heap for alloc, IDT exception handling, and initial Cap'n Proto schema scaffolding.
- **Stage 1: Virtual Memory**: kernel and per-process address spaces, page table abstraction, HHDM preservation, and user-half cleanup.
- **Stage 2: User-Space Transition**: GDT/TSS/syscall setup and Ring 3 round-trip path.
- **Stage 3: Process Abstraction**: ELF loading, process ownership of address spaces and cap tables, process exit cleanup, and the current exit / cap_enter syscall surface.
- **Stage 4: Capability Syscalls / Ring Transport**: Console capability, shared-memory submission/completion rings, cap_enter, CQE transport errors, and alloc-free dispatch paths.
- **Stage 5: Scheduling Core**: PIT/PIC timer preemption, round-robin scheduler, context switching, generation-tagged caps, and VirtualMemory cap.
- **Kernel Networking Smoke**: in-kernel QEMU virtio-net + smoltcp proof for ARP, ICMP, and TCP HTTP.

- **Boot To Shell / Native Shell**: shell-led boot flow, split debug/terminal UARTs, local setup/login, anonymous/operator sessions, and shell REPL.
- **Verified Core**: bounded local/GitHub Kani gate plus high-memory proof gate for selected cap-table, frame-bitmap, transfer rollback, and resource accounting invariants.
- **Shared-Service Demo Base**: chat, adventure, NPC-as-process, and shared service harness prototypes.

Historical completion reports live in `docs/changelog.md`.

Stage 6: IPC And Capability Transfer

Outcome: cross-process capability calls, capability transfer, revocation, and process spawning are capability-shaped and usable by init-owned service graphs. Caller-selected service-visible identity is being replaced by session-bound invocation context: each normal process has one immutable session context, endpoint calls expose privacy-preserving caller-session metadata, and broker-granted service roots/facets carry service access.

Implemented: - `cap_enter` blocking wait - Endpoint kernel object - RECV/RETURN ring opcodes - cross-process IPC - direct-switch IPC handoff - legacy endpoint receiver metadata as transitional IPC machinery - copy/move capability transfer - `CAP_OP_RELEASE` - runtime handle release integration - epoch revocation and Revocable Read proof - MemoryObject substrate – the kernel-level mapping mechanism that backs zero-copy IPC. Demonstrated end-to-end by `make run-memoryobject-shared` (single-shot transfer) and `make run-ipc-zerocopy` (multi-message shared point-to-point buffer with metadata-only endpoint CALLs). The typed `SharedBuffer` surface and service APIs that consume it (`File.readBuf`, `BlockDevice.readBlocks`, NIC RX/TX rings) are still pending. - ProcessSpawner / ProcessHandle - init-owned manifest execution and boot package boundary cleanup - immutable per-process `SessionContext` ownership, default child-session inheritance, and trusted broker-selected child sessions, demonstrated by `make run-session-context`

Remaining themes: - typed `SharedBuffer` capability and consuming service APIs (storage, block, network, GPU) on top of the existing `MemoryObject` substrate - notification objects (so zero-copy producers/consumers can signal each other without per-record endpoint CALLs) - promise pipelining - CapabilityManager list/grant interface - remaining session-keyed shared-service migration for adventure and terminal bridges, including service use of bounded disclosure where needed - scheduling context and resource donation - init ELF embedding

Details: - `docs/backlog/session-bound-invocation-context.md` - `docs/backlog/service-object-identity-migration.md` (superseded) - `docs/backlog/stage-6-capability-semantics.md` - `docs/proposals/service-architecture-proposal.md` - `docs/proposals/storage-and-naming-proposal.md` - `docs/proposals/error-handling-proposal.md`

Stage 7: SMP, Runtime, Networking, And Shell

Outcome: capOS moves from single-CPU scheduling and local-only shell access to multi-CPU execution, thread-aware runtime behavior, socket-shaped network capabilities, and agent/web shell entry points.

SMP status: - Phase A complete: BSP per-CPU syscall stack/current-thread state and unified kernel-entry stack hook. - Phase B complete: APs start through Limine MP, switch to capOS kernel paging/stacks, initialize AP-local CPU state, and park. - Phase C selected AP scheduler-owner proof complete: GS/`swapgs`, LAPIC timer/IPI, TLB shootdown, and first AP scheduler-owner proof are complete. Commit `d88bca7` at `2026-04-25 11:31 UTC` proves AP cpu=1 can run scheduler-owned user contexts under `-smp 2` while a scheduler-owner latch keeps the BSP in kernel idle. Full concurrent scheduling remains future work: per-CPU scheduler ownership, reschedule IPIs, and process-ring-safe concurrent scheduler-owned work. - The next visible SMP milestone is **Multi-Process SMP Concurrency**. Its technical prerequisite is full concurrent SMP scheduling: multiple CPUs must own scheduler work at the same time through reviewed per-CPU ownership, runnable handoff, and cross-CPU wakeup paths. The visible proof is that independent worker processes improve wall-clock runtime on a deterministic CPU-bound workload. - A separate later milestone is **In-Process Threading Scalability**. It proves sibling threads in one process can run on different CPUs and scale the same class of workload after per-thread ring/completion routing removes the current process-wide capability-ring bottleneck.

Runtime/network/shell themes: - reconcile in-process threading implementation status and any follow-on work - Telnet Shell Demo as first TCP-backed `TerminalSession` proof. Plaintext, loopback-only research demo; not a shippable Telnet service. - Tickless idle as the near-term timer cleanup: split clocksource from clockevent, convert timeout waiters to absolute deadlines, replace the user-mode idle process with kernel/per-CPU idle, then stop the periodic tick only when no runnable work exists. Generic full-nohz remains deferred; SQPOLL nohz belongs behind Ring v2, per-CPU scheduler ownership, housekeeping, CPU accounting, and CPU-isolation authority. See `docs/proposals/tickless-realtime-scheduling-proposal.md` and `docs/research/nohz-sqpoll-realtime.md`. - SSH Shell Gateway as the production remote CLI successor to Telnet after host-key, authorized-key, audit, and persistence prerequisites exist - decomposed userspace NIC/network-stack milestone after driver authority gates - native shell agent runner - WebShell-Gateway using the same broker-issued shell/agent authority model

Details: - `docs/backlog/smp-phase-c.md` - `docs/backlog/runtime-network-shell.md` - `docs/proposals/smp-proposal.md` - `docs/proposals/tickless-realtime-scheduling-proposal.md` - `docs/proposals/networking-proposal.md` - `docs/proposals/shell-proposal.md` - `docs/proposals/llm-and-agent-proposal.md` - `docs/proposals/boot-to-shell-proposal.md`

Hardware, Boot, And Storage

Outcome: capOS boots beyond the current ISO/QEMU manifest path, discovers real hardware, supports block devices, and exposes local persistent storage through typed capabilities.

Tracks: - bootable GPT/EFI disk image and `make run-disk` - ACPI/MADT/MCFG discovery - reusable interrupt and PCI/PCIe infrastructure - virtio-blk and NVMe block-device paths - boot binary ISO layout that moves ELF payloads out of the manifest blob - RAM-backed `Store`/`Namespace` - read-only local filesystem proof - writable local storage with recovery policy - cloud device tracks for GCP/AWS/Azure NICs

Details: - `docs/backlog/hardware-boot-storage.md` - `docs/proposals/cloud-deployment-proposal.md` - `docs/proposals/storage-and-naming-proposal.md` - `docs/dma-isolation-design.md`

User Identity, Sessions, And Policy

Outcome: shell, service, and future web sessions receive narrow capability bundles based on explicit identity, freshness, policy, and audit context.

Implemented base: - anonymous/operator shell sessions - password setup/login proof - broker-issued shell bundles - redacted auth/session audit records

Remaining themes: - manifest-seeded local accounts, recovery identities, service identities, and initial role/resource profiles - disk-backed local account store over capability-native storage - default per-account, guest, anonymous, external, and service-account resource bundles - explicit external identity bindings for OIDC/passkey/cloud/certificate principals - durable verifier/passkey records - WebAuthn and passkey-only setup path - broader AuditLog completion - ABAC context such as auth freshness, session age, source, and claims - mandatory-policy labels and wrapper caps - guest and anonymous workload demos - POSIX profile adapter metadata - OIDC/OAuth2 integration

Details: - `docs/proposals/user-identity-and-policy-proposal.md` - `docs/backlog/local-users-management.md` - `docs/proposals/oidc-and-oauth2-proposal.md` - `docs/proposals/certificates-and-tls-proposal.md` - `docs/proposals/cryptography-and-key-management-proposal.md` - `docs/security/trust-boundaries.md`

Security And Verification

Outcome: trust boundaries fail closed, proof gates stay practical, and trusted build inputs remain review-visible.

Implemented base: - host tests for pure logic - Loom ring model - Miri/proptest/Kani paths - dependency policy checks - pinned Limine and Cap'n Proto tooling - DMA isolation design gate - panic-surface inventory

Remaining themes: - Stage-6 trust-boundary refresh - untrusted-service hardening and quota/exhaustion smokes - Kani harness bounds refresh when new proof obligations are concrete

Details: - `docs/backlog/security-verification.md` - `REVIEW.md` - `REVIEW_FINDINGS.md` - `docs/proposals/security-and-verification-proposal.md` - `docs/security/verification-workflow.md` - `docs/trusted-build-inputs.md`

Shared-Service Demos

Outcome: multi-process demos prove resident services, shell-spawned clients, session-bound invocation context, shared harnesses, and eventually network-transparent federation.

Implemented: - First Chat MVP - Local MUD/adventure prototype - NPC-as-process fleet - shared service harness extraction

Remaining themes: - session-keyed service state replacing legacy receiver-selected chat/adventure identity - per-principal chat state and audit - Aurelian Frontier game-depth work after the first deterministic mission slice - native command-surface replacement for prototype `StdIO` - federated chat after network transparency

Details: - `docs/backlog/shared-service-demos.md` - `docs/backlog/aurelian-frontier.md` - `docs/demos/adventure.md` - `docs/proposals/aurelian-frontier-proposal.md` - `docs/proposals/interactive-command-surface-proposal.md`

aarch64 Support

Outcome: port the architecture layer after x86_64 hardware abstraction stabilizes.

Shared code expected to carry over: - capability model and schema - ring structs and transport contracts - userspace runtime model - process/capability abstractions above `arch/`

Architecture-specific work: - EL0/EL1 syscall entry/exit - GICv3 interrupts - ARM generic timer - PL011 UART - TTBR0/TTBR1 MMU setup - TPIDR_EL1 per-CPU data - `kernel/linker-aarch64.ld`

Future Tracks

These are not selected unless `WORKPLAN.md` or user direction pulls them into active scope:

- regular Rust runtime support
- C `libcapos`
- Go `GOOS=capos`
- Lua scripting
- POSIX compatibility
- WASI runtime
- C++ experiments
- GPU/CUDA capability integration
- system monitoring
- network transparency
- process persistence/checkpoint-restore
- live upgrade
- cloud metadata
- volume encryption
- formal MAC/MIC modeling
- browser/WASM support
- robotics realtime control

Use proposal files under `docs/proposals/` and research notes under `docs/research/` before promoting any future track into `WORKPLAN.md`. Lua scripting should arrive as an ordinary capability-scoped userspace runner, not as kernel scripting or ambient shell authority.

Observable Milestones

Completed visible milestones: - 2026-04-22 16:35 UTC, commit `d4016ab`: Unprivileged Stranger - 2026-04-23 08:41 UTC, commit `f554e88`: Native Cap Shell - 2026-04-23 13:39 UTC, commit `e5adafb`: Boot to Shell - 2026-04-23 16:15 UTC, commit `7f19af2`: Revocable Read - 2026-04-23 16:34 UTC, commit `8b66c13`: split UART shell session - 2026-04-23 22:09 UTC, commit `d43b691`: Verified Core - 2026-04-24 00:13 UTC, commit `2cd85a8`: First Chat MVP - 2026-04-24 01:40 UTC, commit `add7f9b`: Local MUD/adventure prototype - 2026-04-24 03:13 UTC, commit `da5f5e9`: Ring as Black Box - 2026-04-24 15:37 UTC, commit `b56a5c1`: First Packet - 2026-04-24 16:47 UTC, commit `a4f1722`: First HTTP - 2026-04-25 05:36 UTC, commit `0b79054`: SMP Phase A: per-CPU data on BSP - 2026-04-25 06:59 UTC, commit `d3c30c6`: SMP Phase B: APs running - 2026-04-25 11:31 UTC, commit `d88bca7`: First AP Scheduler - 2026-04-25 20:25 UTC, commit `2834bfc`: Telnet Shell Demo

Visible demo follow-ups: - Adventure/shared-service follow-ups after the Local MUD prototype: `73d83aa`, `da51dc7`, `353c8bc`, `e20cf07`, `948c96e`, and `ca6300c`. These refine discoverability, room context, expedition map, relic custody, explicit resume, and chat-only named actors; detailed reports live in commit history. - 2026-04-26 04:10 UTC, commit `5480304`: Scoped Telnet Gateway Authority. `telnet-gateway` now uses manifest-forwarded scoped listener authority plus `RestrictedShellLauncher`; detailed verification history lives in commit history. - 2026-04-26 23:12 EEST, commit `4304b0e`: Default run Telnet wiring. The default manifest starts `telnet-gateway`, and `make run` attaches host-local `127.0.0.1:2323 -> guest :23` forwarding. - 2026-04-27 00:02 EEST, commit `7a155f4`: Telnet IAC handoff fix and repeat-connect support. Telnet handoff no longer consumes raw socket input before `intoTerminalSession`, repeated host connections succeed, and the harness drives two consecutive sessions. - 2026-04-28 17:46 UTC, commit `d09243d`: Aurelian Phase 9 competency gates. The adventure proof now has host-testable rank/star/circle policy, status output for rank marks and standing, signifer skill gates, first-mission spell gates, and QEMU assertions for rank denial plus debrief reward. - 2026-04-28 18:12 UTC, commit `47dbfc5`: Aurelian Phase 10 market logistics. Adventure now has typed quote/buy/sell/trade/repair calls, bounded market roles, a deterministic Maro route purchase, and QEMU assertions for market quote, successful exchange, and clean-custody trade refusal. - 2026-04-28 19:36 UTC, commit `e204454`: Aurelian Phase 11a calendar foundation. Generated content now carries fixed-smoke season/day/weather and hazard state plus bounded seasonal resources, Adventure status prints that state, and the real scenario process asserts it through `Adventure.status`. - 2026-04-28 20:08 UTC, commit `48c62db`: Aurelian Phase 11b regional foundation. Generated content now carries settlement, outpost, and route metadata with validation and stable ordering; Adventure status prints a regional summary, and the real scenario process asserts it through `Adventure.status`. - 2026-04-28 21:08 UTC, commit `0b7db05`: Aurelian Phase 11c construction foundation. Generated content now carries material, facility, blueprint, artifact, and enchantment-slot metadata with pure Rust validation and deterministic property derivation; Adventure status prints a construction summary, and the real scenario process asserts it through `Adventure.status`. Construction jobs, material reservation, escrow, completion/release, and full artifact crafting gameplay remain future work. - 2026-04-28 21:36 UTC, commit `f53d044`: Aurelian Phase 11d agent NPC budget foundation. Generated content now carries disabled-by-default

optional NPC agent budget metadata with model profiles, per-session/day input/output token limits, tool-call limits, cooldown, fatigue, sleep, refusal, and audit visibility. Pure Rust fake-model tests cover spending, refusals, disabled transcript stability, bounded output, and no authority mutation from model text; Adventure status prints an aggregate budget line asserted through `Adventure.status`. Live LLM integration, hosted-agent execution, durable memory, autonomous NPC actions, and authority mutation from model output remain future work. - 2026-04-28 22:22 UTC, commit `335a9ee`: Aurelian Phase 12 party foundation. Adventure now has typed local party create/invite/accept/leave/delegate calls and `assist`, keyed by service-created local player labels derived from live caller-session keys. The server uses the unit-tested `adventure-content` party transition state for invite, accept, scoped delegation, assist, and leave cleanup; the scenario process asserts the one-client cap surface and party status line. Two-client QEMU proof, transfer escrow, duel/spar/contest authority, and cross-device multiplayer remain future work. - 2026-04-29 06:43 UTC, commit `ac49375`: Aurelian Phase 12 physical-item transfer foundation. Adventure adds typed `transfer` for same-party service-local player labels, with ordinary inventory mutation kept atomic inside the existing service and backed by pure Rust transfer tests. The scenario process asserts one-client refusal paths without faking a second live session. Currency escrow, broad market/trade coordination, and successful two-client QEMU transfer proof remain future work. - Pending branch `paperclips-autoclipper-tick`: Paperclips Terminal Demo follow-up. The default manifest advertises the clean-room `paperclips` terminal game, and `system-paperclips.cue` plus `make run-paperclips` provide the focused QEMU proof for one-at-a-time manual production, real-time automation, generated Cap'n Proto content loading, simulation ticks, project listing, and clean shell exit. The demo is intentionally outside the active Session-Bound Invocation Context milestone because it exercises a standalone `StdIO` plus `Timer` terminal process rather than shared-service caller identity.

Active visible milestone: - Session-Bound Invocation Context: normal workload processes have exactly one immutable live session context, endpoint calls reveal only privacy-preserving caller-session metadata by default, and shared services stop deriving caller identity from caller-selected service-visible metadata. Commit `3edee90` at `2026-04-28 16:26 UTC` lands the first proof for child session inheritance, failed second-session injection, and trusted broker-selected child contexts; commit `3469c27` at `2026-04-28 16:54 UTC` adds broker-side expired-session rejection; commit `687511a` at `2026-04-28 17:43 UTC` adds endpoint caller-session metadata, payload-spoof rejection for invocation context, and stale normal endpoint rejection; commit `f0cb74b` at `2026-04-28 18:38 UTC` adds transfer-scope enforcement for endpoint IPC, endpoint returns, and spawn grants; commit `0f92d77` at `2026-04-28 19:33 UTC` adds explicit endpoint subject disclosure gating by request and scope; commit `dc7ece4` at `2026-04-28 20:06 UTC` migrates chat membership to endpoint caller-session keys. Later Gate 4 slices retired normal shell badge selection, bound terminal and stdio bridge authority to live caller sessions, keyed the 128-bit opaque caller reference with a non-reused endpoint service-scope id, and commit `5e9dc4e` at `2026-04-29 11:05 UTC` proves one child process/session receives distinct opaque caller-session reference tuples across two endpoint service scopes. Remaining selected-milestone work is the peer-owned adventure migration, final full-gate verification, and any documentation alignment needed after that migration lands.

Paused visible milestone: - SSH Shell Gateway: `ssh` reaches the capOS login/native shell flow through an SSH-backed `TerminalSession` in QEMU, using host-local forwarding, public-key authentication, denied unsupported SSH features, and the same child shell capability boundary proven by Telnet. This remains planned Stage 7 work, but network-backed shell delegation should wait for the active session-bound invocation context migration to settle.

Candidate next visible milestones: - Multi-Process SMP Concurrency: implement full concurrent SMP scheduling, then have `make run-smp-process-scale` boot QEMU with multiple CPUs, run a deterministic CPU-bound workload split across independent worker processes, print verified output plus 1/2/4-process timing, and record near-linear 1-to-2 CPU speedup under repeated KVM-backed runs. - In-Process Threading Scalability: after per-thread capability rings and completion routing exist, have `make run-thread-scale` run the same class of workload across sibling threads in one process, verify the result, and record 1/2/4-thread timing without relying on a process-wide ring waiter. - Agent Shell - WebShellGateway - bootable disk image - local disk storage - federated chat

Select the next milestone in `WORKPLAN.md` only after the current selected milestone is achieved and recorded, or when the user explicitly changes the selected milestone.

Changelog

Historical milestone and review reports for capOS.

This file exists to keep mandatory agent context small. `WORKPLAN.md` should contain current execution state, `REVIEW_FINDINGS.md` should contain unresolved findings, and detailed historical reports should move here after they stop affecting the next task.

2026-04-28 / 2026-04-29

Session-Bound Invocation Context Core Gates

- 2026-04-29 08:40 UTC: the in-flight Session-Bound Invocation Context milestone has landed its core gates and remains in cleanup. The implemented pieces now include the process-session invariant, default endpoint caller-session metadata, stale normal endpoint rejection, transfer scopes, field-granular disclosure gating, session expiry for broker-issued shell bundle caps, guest bundle narrowing, chat membership keyed by opaque caller-session references, Aurelian player state keyed by live endpoint caller-session metadata, and terminal output liveness checks. Remaining work stays in `WORKPLAN.md`, `docs/status.md`, and `docs/roadmap.md` until the milestone closes: terminal/stdio bridge completion, broader shared-service cleanup, and final service-scoped reference derivation/rotation rules.

2026-04-25

SMP Phase C: Multi-CPU Scheduling

- 2026-04-25 11:47 UTC: the selected SMP Phase C AP scheduler-owner proof completed after review and post-merge verification. The proof moved syscall/per-CPU access to `KernelGsBase`/

swapgs, added PIT-calibrated xAPIC LAPIC timer/IPI support with PIT/PIC fallback and x2APIC deferred, added resident-mask TLB shootdown, split scheduler current-thread tracking into per-CPU slots, and proved AP cpu=1 can run scheduler-owned user contexts while the BSP stays in kernel idle behind a scheduler-owner latch. Verification passed `cargo fmt --check`, `cargo check -p capos-kernel`, `git diff --check HEAD~1..HEAD`, `make docs`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'`. Broader SMP Phase C work remains: per-CPU run queues, reschedule IPIs, concurrent scheduler-owned work on more than one CPU, and per-thread rings.

- 2026-04-25 11:36 UTC: review fix for the scheduler-ownership gate added a one-way scheduler-owner latch in commit `d88bca7`. AP cpu=1 and the BSP can no longer both enter scheduler-owned user work; if the BSP claims fallback ownership first, late APs remain in kernel idle. `make docs`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'` passed with the usual format/check gates.
- 2026-04-25 11:04 UTC: scheduler-ownership gate implemented on `workplan/smp-phase-c-scheduler-ownership`. Scheduler current state became per-CPU, AP per-CPU records were registered for current-thread and kernel-entry stack updates, AP TSS.RSP0 is updated on context switches, APs program LAPIC timers from BSP calibration, and AP cpu=1 can enter the scheduler from the kernel idle loop. The proof deliberately keeps one scheduler owner because the capability ring is still process-wide.
- 2026-04-25 09:22 UTC: TLB shootdown gate implemented on `workplan/smp-phase-c-tlb-shootdown`. User page-table `map`, `unmap`, and `protect` route through shootdown-aware helpers after local mapper flush. Each `AddressSpace` tracks a conservative resident CPU mask; scheduler CR3 handoff marks the selected address space resident on the current CPU before returning to userspace. Vector 49 handles shootdown requests through per-CPU pending full-TLB flush generations and an allocation-free completion path. Verification passed `cargo fmt --check`, `cargo check -p capos-kernel`, `git diff --check`, `make docs`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'`.
- 2026-04-25 08:21 UTC: x2APIC planning recorded in `docs/research/x2apic-and-virtualization.md`, `docs/proposals/smp-proposal.md`, and `docs/proposals/cloud-deployment-proposal.md`. The Phase C path remains PIT-calibrated xAPIC MMIO LAPIC timer/IPI support; x2APIC is a later backend.
- 2026-04-25 08:20 UTC: LAPIC timer/IPI foundation implemented on `workplan/smp-phase-c-lapic-ipi`. The kernel enables xAPIC MMIO on the BSP, maps the LAPIC page, initializes parked AP LAPIC state, routes scheduler ticks through LAPIC timer vector 48, installs the spurious vector, remaps and masks the legacy PIC when LAPIC ticks are available, and wires vector 49 for IPI users. Verification passed `cargo fmt --check`, `cargo check -p capos-kernel`, `git diff --check`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'`.
- 2026-04-25 07:54 UTC: syscall entry/exit gate implemented on `workplan/smp-phase-c-swapgs`. The BSP initializes `KernelGsBase`, syscall entry executes `swapgs`, and GS-relative `PerCpu` offsets carry saved user RSP and syscall kernel RSP state. Scheduler/exit restore paths use a dedicated `restore_context_after_syscall` trampoline. Verification passed `cargo fmt --check`, `cargo`

`check -p capos-kernel`, `git diff --check`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'`.

- 2026-04-25 07:33 UTC: Phase C grounding gate completed after Ring v2, scheduler-ordering, and LAPIC/TLB grounding review. Remaining implementation gates were GS/`swapgs`, LAPIC timer/IPI, TLB shootdown, scheduler ownership, and AP scheduler-work proof.
- 2026-04-25 07:26 UTC: LAPIC/TLB grounding recorded in `docs/architecture/scheduling.md` and `docs/proposals/smp-proposal.md`. Current page-table mutation methods flushed only the local CPU; Phase C required shootdown-aware wrappers and CPU residency tracking before address spaces could migrate or run on multiple CPUs.
- 2026-04-25 07:15 UTC: scheduler Phase C ordering recorded in `docs/architecture/scheduling.md` and `docs/proposals/smp-proposal.md`: migrate syscall/per-CPU access first, add LAPIC timer/IPI next, enable TLB shootdown before address-space migration, then split scheduler current/run-queue ownership.
- 2026-04-25 07:06 UTC: full-SMP ring/threading direction recorded in `docs/proposals/ring-v2-smp-proposal.md` and grounded by `docs/research/completion-ring-threading.md`. The long-term transport target is per-thread capability rings with completion routing by generation-checked `ThreadRef`.

Telnet Shell Demo Planning

- 2026-04-25 20:25 UTC: `Telnet Shell Demo` completed at reviewed merge `2834bfc`. The merged path adds `telnet-gateway`, `system-telnet.cue`, `make run-telnet`, and `make qemu-telnet-harness`, proving QEMU host-local forwarding from `127.0.0.1:2323` to guest port 23, password login, `caps`, `session`, and clean exit through a socket-backed `TerminalSession`. The child shell transcript proves it receives no raw `NetworkManager`, `ProcessSpawner`, TCP, or unknown capability interfaces. A scoped gateway authority follow-up remains open for replacing the trusted demo gateway's factory network/spawn authority with scoped listener and shell-launch grants.
- 2026-04-25 11:47 UTC: selected `Telnet Shell Demo` as the next visible milestone. The grounding starts from networking, shell, boot-to-shell, service-architecture, trust-boundary, status, roadmap, and review-finding docs plus the existing `ix-on-capos-hosting`, `pingora`, `genode`, and `sel4` research notes. The milestone is networking Phase B: keep the TCP stack in-kernel behind capability objects, expose a socket-backed `TerminalSession`, and leave userspace NIC decomposition, TLS/OIDC, and production remote access to later milestones.

SMP Phase B: APs Running

- 2026-04-25 06:59 UTC: `SMP Phase B: APs running` completed after review. AP startup uses Limine `MpRequest`/`MpInfo::bootstrap`, stable AP records, AP-owned kernel/IST stacks, AP-local GDT/TSS state, capOS kernel PML4 handoff, AP-owned kernel RSP handoff, shared IDT, `KernelGsBase`, syscall MSRs, SMEP/SMAP state, and a parked interrupt-disabled `hlt` loop. Verification passed `cargo fmt --check`, `cargo check -p capos-kernel`, `git diff --check`, `make run-smoke QEMU_KVM_FLAGS='-smp 2'`, and `make run-spawn QEMU_KVM_FLAGS='-smp 2'`.

- 2026-04-25 06:49 UTC: AP entry switched to the capOS kernel PML4 and AP-owned kernel stack before reporting online. The non-returning handoff disables interrupts, clears DF, writes CR3, aligns RSP, and jumps into the Rust AP continuation before loading AP-local CPU state.
- 2026-04-25 06:37 UTC: AP startup first reached a parked kernel idle loop. The BSP starts prepared APs through Limine `MpInfo::bootstrap`, passes a stable AP record pointer through `extra_argument`, waits for bounded online count, and leaves userspace scheduling on the BSP.
- 2026-04-25 06:10 UTC: `kernel/src/arch/x86_64/smp.rs` began owning the Limine `MpRequest`, enumerating non-BSP CPUs, allocating AP-local `PerCpu` objects and stacks, and recording dense capOS CPU ids separately from Limine processor and LAPIC ids.
- 2026-04-25 05:55 UTC: the roadmap and `docs/proposals/smp-proposal.md` were updated to use the local Limine `MpRequest` API and to separate dense capOS CPU ids from Limine ACPI `processor_id`.
- 2026-04-25 05:48 UTC: AP-startup grounding recorded in `docs/proposals/smp-proposal.md`, `docs/architecture/scheduling.md`, `docs/architecture/threading.md`, `docs/research/llvm-target.md`, `docs/research/out-of-kernel-scheduling.md`, and `docs/research/sel4.md`.

SMP Phase A And User-Buffer Protection

- 2026-04-25 05:36 UTC: `SMP Phase A: Per-CPU data on BSP` completed. The BSP has a concrete `PerCpu` object for syscall stack state and current-thread mirroring. Runtime kernel-entry stack updates flow through one per-CPU hook that updates TSS.RSP0 and syscall-entry kernel RSP together. Verification passed `cargo fmt --check`, `cargo check -p capos-kernel`, `git diff --check`, `make run-smoke`, and `make run-spawn`.
- 2026-04-25 04:00 UTC: `workplan/user-buffer-validation-protection` closed the private process-buffer `validate_user_buffer` TOCTOU finding before Stage 7. `AddressSpace` now owns validation plus HHDM-backed user copy/read helpers under the process address-space mutex. Verification passed `make run-spawn`, `make run-measure`, `make check`, `make docs`, `cargo check -p capos-kernel`, `cargo check -p capos-kernel --features measure`, `cargo test-config`, `cargo test-ring-loom`, `cargo test-lib`, `cargo fmt --check`, and `git diff --check`.
- 2026-04-25 03:36 UTC: final review of `workplan/futexspace-private-wait-wake` found and then closed a park ownership bug where a sibling thread could drain a park SQE and park the wrong `ThreadRef`. The fix requires `CAP_SQE_THREAD_OWNED` plus the owning thread id for `CAP_OP_PARK`.

2026-04-24

- 2026-04-24 22:41 UTC: in-process threading design freeze completed. The thread/process ownership contract and park authority contract were frozen; review findings raised during the park pass were fixed before merge.
- 2026-04-24 20:53 UTC: runtime prerequisites for threading and Go completed. Follow-up fixes reject writable-executable user mappings in anonymous `VirtualMemory` and `MemoryObject` paths and add QEMU smoke coverage.

- 2026-04-24 16:45 UTC: kernel networking smoke completed. The QEMU virtio-net path proves modern transport discovery, virtqueue setup, descriptor completion, ARP, ICMP echo, smoltcp handoff, static IPv4, and a host-backed TCP HTTP GET proof.
- 2026-04-24 13:11 UTC: custom userspace target completed. Userspace artifacts build through `targets/x86_64-unknown-capos.json`, and checked-in CUE manifests reference `target/x86_64-unknown-capos/release` artifacts while the kernel remains on `x86_64-unknown-none`.
- 2026-04-24 11:25 UTC: boot-manifest parser finding resolved by `KernelBootstrapManifest`, which decodes only kernel-owned fields from the Cap'n Proto reader and avoids materializing the init-owned service graph.
- 2026-04-24 10:53 UTC: boot package boundary cleanup completed. Kernel bootstrap validates only the kernel-owned manifest boundary and starts one `initConfig.init` process; mkmanifest and init own service-graph validation.
- 2026-04-24 03:06 UTC: `Ring as Black Box` completed. QEMU debug-tap builds export bounded metadata-only ring records, and `tools/ringtap-viewer/` renders correlated SQE/CQE evidence offline.
- 2026-04-24 02:16 UTC: shared service harness extraction completed for the non-speculative duplicated demo-service pieces.
- 2026-04-24 00:48 UTC: status docs refreshed after boot-to-shell and revocation work.
- 2026-04-24 00:34 UTC: dependency policy gate restored by explicitly allowing `BSD-3-Clause` for the current Argon2 dependency closure with rationale in `docs/trusted-build-inputs.md`.
- 2026-04-24 00:12 UTC: current chat schema/server/client/bot path accepted as the First Chat MVP slice; follow-on hardening remains tracked separately.

2026-04-23

- 2026-04-23 22:05 UTC: `Verified Core` completed. `make kani-lib` runs the bounded local/GitHub proof gate, and `make kani-lib-full` adds the high-memory transfer proof gate.
- 2026-04-23 21:30 EEST: shell-led boot flow landed. `capos-shell` is init on shell-led manifests, anonymous shell starts on boot, and `login`/`setup` mint authenticated operator bundles.
- 2026-04-23 16:34 UTC: split UART shell session completed. `make run` presents login/native shell on terminal UART while kernel/debug output goes to `target/qemu-console.log`.
- 2026-04-23 16:34 UTC: the revocable read milestone completed. `make run-revocable-read` proves a parent can revoke a child-local `BootPackage` grant through `CapabilityManager`.
- 2026-04-23 21:30 EEST: boot-to-shell milestone completed. Default `make run` reaches setup/login, volatile credential creation, password-authenticated session minting, broker-issued shell bundles, redacted auth/session audit records, and an interactive native shell REPL over serial.

2026-04-20 To 2026-04-22

- 2026-04-22 23:50 UTC: AP-independent review remediation closed issues around endpoint owner cleanup, ProcessSpawner badge attenuation, ProcessSpawner heap-OOM paths, queued release semantics, pinned CUE toolchain enforcement, stale authority docs, spawn hardening stability, NMI IST coverage, MemoryObject replacement for raw frame grants, generated capnp ownership, and manifest validation modularization.

- 2026-04-21 22:21 EEST: VirtualMemory quota review finding resolved with per-address-space ownership tracking, holder quota, bounded auto-placement probes, owned-range checks, and QEMU coverage.
- 2026-04-21 18:46 EEST: smoke demos moved into a nested `demos/` userspace workspace, and `system.cue` packages each demo as a distinct release-built binary/service.
- 2026-04-21 16:56 UTC: cross-process `CAP_OP_CALL`/`RECV`/`RETURN` flow completed.
- 2026-04-21 12:28 UTC: allocation-free synchronous ring dispatch landed.
- 2026-04-21 01:15 UTC: SMEP/SMAP, `cap_enter` blocking waits with timeout, Endpoint, and RECV/RETURN routing completed.
- 2026-04-20 23:05 EEST: Phase 0 and Phase 1 cleanup completed, including dead-code cleanup, ELF validation hardening, deterministic error paths, corrupted ring recovery policy, `capos-lib` split, and host tests.

2026-04-05

- 2026-04-05 16:39 EEST: Stage 4 and Stage 5 direction took shape. Capability invocation moved from direct calls toward the shared-memory ring and `cap_enter`, preemptive scheduling was documented after the PIT/context-switch scheduler landed, and stale `cap_call` proposal text was replaced with the ring-based model.
- 2026-04-05 17:08 EEST: the planning surface matured. The roadmap moved out of `README.md`, review findings split into their own log, and the project added or revised userspace-binaries, SMP, Go runtime, cloud deployment, storage/naming, service architecture, GPU, error-handling, and persistence planning.
- 2026-04-05 10:35 EEST: design grounding expanded through prior-art research on seL4, Zircon, Plan 9/Inferno, EROS/CapROS/Coyotos, Genode, and LLVM target customization. That research fed the interface-as-permission decision: restrict authority by granting narrower typed capabilities or wrapper objects, not by attaching generic READ/WRITE-style rights to ordinary caps.
- 2026-04-05 02:17 EEST: manifest/config and init-side planning advanced with a no-std manifest config loader, hardening tests, and early init-side manifest parsing demos.

2026-04-04

- 2026-04-04 21:02 EEST: capOS bootstrapped as a Limine-loaded Rust kernel with serial output, then gained the first Cap'n Proto capability invocation path and a staged implementation roadmap.
- 2026-04-04 23:12 EEST: Stage 1 through Stage 3 landed in rapid succession: virtual memory with kernel remapping and isolated process address spaces; Ring 3 user-space transition through GDT/TSS/syscall setup; and process abstraction with ELF loading, per-process address spaces/cap tables, static init, and QEMU auto-exit proof.
- 2026-04-04 23:57 EEST: the first major design proposals appeared, including userspace TCP/IP networking and capability-based service architecture. Networking was split into its own proposal after review, setting up later virtio/smoltcp and service-decomposition work.

Whitepaper Plan

This document plans a future capOS whitepaper / technical report. Its job is **not** to draft the paper, but to keep the paper's evidence requirements visible so that ongoing capOS development can be sequenced to produce them.

Treat this file the way `WORKPLAN.md` is treated: keep it current with user direction, prune as items close, and cross-link to `docs/roadmap.md`, `WORKPLAN.md`, and the relevant proposals rather than duplicating them.

Companion files: - `outline.md` – section-by-section sketch of the future paper. - `evidence-gaps.md` – live tracking of which paper claims still lack artifact-level evidence and which workplan slice will close each gap.

Format and Release Metadata

- Format: whitepaper / technical report. No conference page limit.
- Release metadata is intentionally out of scope for this planning page.
- Deadline: none. Quality drives sequencing; the paper does not freeze before Tier-1 evidence is in place.

Working Title

Schema-as-ABI: Typed Capabilities and Ring-Transport Dispatch in capOS.

Thesis

A research OS where Cap'n Proto schemas serve simultaneously as

1. the kernel ABI,
2. the only access-control mechanism (the interface *is* the permission),
3. the IPC wire format, and
4. the persistence and network-transparency substrate,

replacing the parallel rights bitmasks, untyped IPC primitives, and C-header syscall conventions that prior capability OSes still carry.

Position relative to prior art: - **seL4** – formally verified, but rights are kernel-managed bitmasks and IPC payloads are untyped message registers. - **Zircon** – typed kernel objects, but the typing lives in C handle conventions and a fixed syscall surface, not a schema language. - **EROS / CapROS** – persistent capabilities, but no schema-as-wire layer on top of the capability primitives. - **Genode** – component-typed RPC, but RPC is built on top of kernel IPC primitives that are not themselves the wire format. - **Plan 9** – "everything is a file" universal interface, but file I/O is a single weak type compared to schema-typed methods.

The unique claim under evaluation is that one schema layer subsumes the rights, ABI, IPC, persistence, and network-transparency mechanisms that those systems each provide separately.

Core Claims and Required Evidence

The paper stands on five claims. Each must be backed by an artifact, not by description alone.

- **Claim:** C1. Schema-typed methods can replace parallel rights bitmasks.
  - **Required artifact:** Working capability dispatch where authority narrowing happens via wrapper caps, not flag bits. Session-bound invocation context replaces caller-selected endpoint identity without generic rights flags.
  - **Status:** Partial. Delegated-relabel containment, transitional representation substrate, the historical service-object routing proof, and the Gate 1 process-session invariant proof are done; endpoint caller-session metadata and shared-service migration remain open.
- **Claim:** C2. A shared-memory ring with one progress syscall is a sufficient kernel boundary.
  - **Required artifact:** Implemented `cap_enter` + SQ/CQ + opcodes covering CALL/RECV/RETURN/RELEASE/NOP. Loom model. Real latency/throughput numbers vs. a syscall-per-call baseline.
  - **Status:** Implementation done. **Numbers missing.**
- **Claim:** C3. The wire format enables persistence.
  - **Required artifact:** A capability whose state is serialized via its schema and restored across kernel reboot. Even a RAM-backed `Store` proof-of-concept is sufficient.
  - **Status: Missing.**
- **Claim:** C4. The wire format enables network transparency.
  - **Required artifact:** One capability invocation crossing TCP between two capOS instances (or between QEMU and a host stub) using the same schema.
  - **Status: Missing.**
- **Claim:** C5. The kernel can be developed and verified at this size with practical tooling.
  - **Required artifact:** Kani/Loom/Miri/panic-inventory artifacts plus the workplan/review discipline.
  - **Status:** Strong; needs distillation.

The paper should not promise more than the artifacts deliver. Limitations go in their own section, written before claims tighten.

Tier-1 Development Pre-requisites (load-bearing)

Without these, the corresponding claim is bluff.

1. **Session-Bound Invocation Context.** Closes the C1 gap. Active in `WORKPLAN.md`; detailed execution lives in `docs/backlog/session-bound-invocation-context.md`.
2. **Measurement harness producing real numbers.** Closes the C2 gap. Extends `make run-measure` to record:
   - `cap_enter` round-trip latency (warm and cold ring, single CPU)
   - SQE dispatch throughput in ops/sec, single-process baseline
   - Endpoint IPC: ordinary scheduler path vs. direct-switch handoff
   - Schema dispatch overhead vs. the existing `NullCap` baseline
   - Memory cost per capability slot Output should be reproducible from a single `make` target and saved under a tracked path so the paper can cite specific commits.
3. **Persistence proof-of-concept.** Closes the C3 gap. Minimum viable shape: a RAM-backed `Store` that serializes one non-trivial capability via its capnp schema, dumps it, restores it after

kernel reboot in QEMU, and proves a method call still works. This is explicitly narrower than the full storage proposal; treat it as paper evidence, not the production storage track.

4. **Network-transparency proof-of-concept.** Closes the C4 gap. Minimum viable shape: a `RemoteCap` shim that forwards one capnp method call over an established TCP capability between two capOS QEMU guests (or between QEMU and a host capnp stub), using the existing schema. Naturally rides on the SSH/Telnet networking work already in flight.
5. **At least one of {promise pipelining, notification objects}.** Promotes the schema-as-ABI claim from synchronous CALL/RECV to capnp-rpc-shaped composition. Listed as remaining Stage 6 theme; promote to selected work after session-bound invocation context.

Tier-2 Pre-requisites (strengthening)

These do not block the thesis but materially raise paper quality.

6. **Ring protocol Kani proof.** Currently Loom-only. Promotes "verified core" claim from "we run tools" to "we have proofs."
7. **Full concurrent SMP scheduling** (SMP Phase C beyond first AP scheduler-owner proof). Removes a single-CPU asterisk from the evaluation.
8. **SSH Shell Gateway end-to-end.** Paused behind the selected Session-Bound Invocation Context milestone. When resumed, it gives the system-composition section one concrete external-facing artifact rather than only smoke binaries.
9. **One non-toy demo beyond Adventure / First Chat.** Either a storage-backed service or the federated-chat sketch, depending on which composes better with the persistence and network PoCs.

Tier-3 (out of scope for first paper)

Acknowledge in Future Work; do not block on:

- aarch64 port
- GPU / cloud
- live upgrade
- formal MAC/MIC modeling
- Go runtime / WASI / POSIX
- production volume encryption

Recommended Sequencing

Paper-driven priority order, mapped onto the existing roadmap rather than parallel to it:

1. Finish current selected milestone (**Session-Bound Invocation Context**). Closes C1 and anchors §5.5.
2. **Measurement harness.** Closes C2. Parallelizable with (1); touches the `measure` feature only.
3. **Ring Kani proof.** Parallelizable; security/verification track owns it.
4. **Persistence PoC.** Closes C3. Scope explicitly to paper evidence, not the full storage proposal.
5. **Network-transparency PoC.** Closes C4. Rides on existing network work.
6. **Promise pipelining or notifications.** Pick whichever composes better with the persistence and network PoCs.

7. **SMP concurrent scheduling.** Lifts evaluation quality.

When any of these is selected as a workplan slice, the slice description should reference this file so the paper-evidence motivation stays visible alongside the system motivation.

Honest Framing Notes

- The collaboration methodology section should describe the actual workflow: dedicated worktrees per task, mandatory verification gates, the `WORKPLAN.md` / `REVIEW.md` / `REVIEW_FINDINGS.md` discipline, and the task-branch merge-back-to-main rule. This is itself a contribution: it shows what process structure makes draft-to-review flow tractable on a kernel-shaped artifact.
- Verification claims must state exactly what Kani / Loom / Miri / panic inventory cover and what they do not. No formal-methods overstatement.
- Capability-OS novelty against seL4 / EROS / Genode must be specific about what the schema-as-wire layer adds beyond what those systems already provide. Generic capability-OS claims are not novel.
- Persistence and network-transparency claims must lean only on the actual PoCs once they exist, not on the shape of the future storage and networking proposals.

Process

- This file lives on its own task branches under `paper/...`.
- Edits to this file follow the same dedicated-worktree rule as everything else in the repo.
- The paper sources live at `papers/schema-as-abi/` (Typst). `docs/paper/` remains the proposal/reference home: `plan.md`, `outline.md`, and `evidence-gaps.md` describe what the paper needs and why, while the paper itself (`main.typ`, `references.bib`) lives next to other buildable artifacts under `papers/`. Build via `make paper` from the repository root; output goes to `target/papers/schema-as-abi/main.pdf`.
- This plan is read alongside `docs/roadmap.md` and `WORKPLAN.md` whenever workplan priorities are revisited, so paper-driven motivation does not silently drift out of sequencing.

Whitepaper Outline

Section-by-section sketch of the capOS whitepaper. Each section lists its supporting docs and its evidence dependency on `evidence-gaps.md`. The actual draft now lives at `papers/schema-as-abi/main.typ`; this file is the structural reference and stays in `docs/paper/` alongside the plan and evidence-gap tracker. When a section's evidence lands, update the draft in the same task that updates this outline and `evidence-gaps.md`.

§1. Introduction

- Problem: kernel ABIs are typically C-header syscall conventions plus a separate rights/permission machinery; capability OSes typically still keep rights, IPC, and serialization as three layers.
- Thesis (one paragraph): schema-as-ABI collapses these layers.
- Contributions list. Map each contribution to a paper section and to a Tier-1 artifact in `evidence-gaps.md`.

Backed by: `docs/overview.md`, `docs/capability-model.md`.

§2. Background and Related Work

- seL4: untyped IPC, kernel-managed rights, formal verification. Source: `docs/research/sel4.md`.
- Zircon: typed kernel objects, fixed syscalls. Source: `docs/research/` (Zircon notes).
- EROS / CapROS: persistent capabilities without schema-as-wire.
- Genode: component-typed RPC over kernel IPC primitives. Source: `docs/research/genode.md`.
- Plan 9: universal-interface as file I/O.
- capnp-rpc: prior art for schema-driven RPC; capOS pulls this layer into the kernel boundary.
- LLVM/target customization: relevant for the userspace runtime story.

Backed by: `docs/research.md` and `docs/research/*`.

§3. Capability Model

- Typed schema = permission. No parallel rights bitmask.
- Narrowing via wrapper capabilities, not flag reduction.
- Generation tags, copy/move semantics, release ABI.
- Service objects vs. badged endpoints (the badge migration).

Backed by: `docs/capability-model.md`, `docs/proposals/service-object-capabilities-proposal.md`, `docs/architecture/capability-ring.md`, `docs/authority-accounting-transfer-design.md`.

Evidence dependency: claim C1 in `evidence-gaps.md`.

§4. Ring Transport

- SQ/CQ shared-memory ring as the only kernel boundary.
- `cap_enter` semantics: flush pending SQEs, wait for completions or timeout.
- Opcode set: CALL, RECV, RETURN, RELEASE, NOP. Reserved Finish.
- Direct-switch IPC handoff.
- Transport errors and CQE error model.
- Loom protocol model.
- Backpressure, fairness, scheduler-tick polling rules.

Backed by: `docs/architecture/capability-ring.md`, `docs/architecture/ipc-endpoints.md`, code in `kernel/src/cap/ring.rs`, `capos-config/tests/ring_loom.rs`, `capos-rt/src/ring.rs`.

Evidence dependency: C2.

§5. Capability Lifecycle

- Copy, move, release.
- Revocation: epoch model, Revocable Read proof.
- Resource accounting: per-process ledger, exactly-once transfer rollback.
- Service objects replacing badges.

Backed by: `docs/authority-accounting-transfer-design.md`, `docs/proposals/service-object-capabilities-proposal.md`, `docs/backlog/stage-6-capability-semantics.md`.

Evidence dependency: C1.

§6. System Composition

- Manifest-driven boot, init service graph.
- Authority broker, sessions, anonymous/operator/SSH-public-key paths.
- Demo services: First Chat, Adventure, SSH Shell Gateway as the end-to-end auth/identity boundary.

Backed by: `docs/architecture/manifest-startup.md`, `docs/proposals/service-architecture-proposal.md`, `docs/proposals/user-identity-and-policy-proposal.md`, `docs/proposals/ssh-shell-proposal.md`, `docs/security/trust-boundaries.md`.

Evidence dependency: C2 plus completion of selected SSH milestone.

§7. Verification and Engineering Process

- Kani / Loom / Miri / panic-surface inventory: scope, bounds, gaps.
- Dependency policy, pinned trusted inputs.
- Workplan / review discipline as part of the methodology.
- Collaboration methodology: dedicated worktrees per task, mandatory gates, REVIEW_FINDINGS as durable open-issue tracking, merge-back-to-main rule.

Backed by: `docs/proposals/security-and-verification-proposal.md`, `docs/security/verification-workflow.md`, `docs/panic-surface-inventory.md`, `docs/trusted-build-inputs.md`, `REVIEW.md`, `REVIEW_FINDINGS.md`, `WORKPLAN.md`.

Evidence dependency: C5.

§8. Evaluation

- `cap_enter` round-trip latency.
- Ring throughput.
- Endpoint IPC: scheduler path vs. direct-switch.
- Schema dispatch vs. NullCap baseline.
- Memory cost per capability slot.
- Persistence proof-of-concept: round-trip integrity.
- Network-transparency proof-of-concept: cross-instance call latency.

Evidence dependency: C2, C3, C4. **All three artifacts are currently missing or partial; do not begin drafting this section until they exist.**

§9. Limitations

Honest framing, written before §1 is tightened. Likely items:

- Single architecture (x86_64). aarch64 deferred.
- Partial SMP at time of writing; reflect actual state.

- Persistence and network-transparency demonstrated as proofs-of-concept, not production subsystems.
- Verification scoped to listed invariants; not whole-kernel.
- No production hardware path; QEMU-only artifact.

§10. Future Work

Reference `docs/roadmap.md` Future Tracks rather than restating them. Topics: aarch64, GPU, live upgrade, persistence-at-scale, formal MAC/MIC, Go/WASI, cloud metadata, federated multi-instance demos.

§11. Collaboration Methodology Note

Short standalone section.

- Workflow: per-task worktrees, mandatory verification gates, REVIEW.md conventions, REVIEW_FINDINGS.md as durable open-issue tracking, merge-back-to-main discipline.
- What this enabled: surfacing review and merge hygiene against changing publication assumptions while keeping evidence-driven sequencing intact.
- Reproducibility kit: pinned toolchains, ISO build, QEMU smoke targets.

§12. Conclusion

To be written last, after §8 numbers exist.

Appendices (candidate)

- A. Schema excerpt for one representative capability with method-by-method ABI commentary.
- B. Reproducibility kit: exact commands, pinned commits, expected outputs.
- C. Build/boot flow diagram.
- D. Process model diagram.

Whitepaper Evidence Gaps

Live tracking of which paper claims still lack artifact-level evidence and which workplan slice will close each gap. Update this file in the same task that closes a gap. When a gap closes, also update the corresponding `#todo` block in the paper draft at `papers/schema-as-abi/main.typ`.

Claim C1: Schema-typed methods replace parallel rights

Status: partial.

- ☒ Capability dispatch via `CapObject` + capnp schema (kernel + capos-rt).
- ☒ Authority narrowing via wrapper capabilities (no rights bitmask).
- ☒ Stage 6 Gate 0: delegated endpoint relabeling containment.
- ☒ Transitional service-object representation and selector preservation substrate for endpoint-backed service facets.
- ☒ Historical service-object routing/lifecycle proof: synthetic receiver-cookie spoofing, lifecycle, and transfer coverage in commit `a4655f0`.

- ☒ Session-Bound Invocation Context Gate 1: one immutable session context per process, with child inheritance and hostile second-session injection checks, demonstrated by `make run-session-context`.
- □ Session-Bound Invocation Context Gate 2: privacy-preserving endpoint caller-session metadata, explicit subject disclosure, and payload spoofing/disclosure hostile checks.
- □ Session-Bound Invocation Context Gate 3: chat / adventure / stdio migrated off caller-selected endpoint identity to session-keyed service authority.
- □ Trust-boundary doc refresh recording the post-migration state.

Closes via: `docs/backlog/session-bound-invocation-context.md`. Already next on the mainline path per `WORKPLAN.md`.

Claim C2: Ring + one progress syscall is a sufficient boundary

Status: implementation done, numbers missing.

- ☒ `cap_enter` semantics, SQ/CQ ring, CALL/RECV/RETURN/RELEASE/NOP.
- ☒ Direct-switch IPC handoff.
- ☒ Loom protocol model.
- ☒ `make run-measure` benchmark-only feature scaffolding.
- □ Reproducible measurement run that records:
  - `cap_enter` warm/cold round-trip latency
  - SQE dispatch throughput
  - Endpoint IPC: scheduler vs. direct-switch handoff latency
  - Schema dispatch overhead vs. NullCap baseline
  - Memory cost per capability slot
- □ Tracked path for measurement output so the paper can cite a commit.
- □ Optional: comparison numbers from published seL4/Zircon literature where the comparison is fair.

Closes via: a dedicated measurement-harness slice. Not yet in `WORKPLAN.md`; promote when scheduling allows.

Claim C3: Wire format enables persistence

Status: missing.

- □ Minimal RAM-backed `Store` capability.
- □ One non-trivial capability whose state is serialized via its capnp schema, dumped, and restored across kernel reboot in QEMU.
- □ Method call on the restored capability returns the expected result.
- □ QEMU smoke target wired to this proof.
- □ Documentation explicitly scoping this to "paper evidence" rather than the full storage proposal in `docs/proposals/storage-and-naming-proposal.md`.

Closes via: a paper-scoped persistence-PoC slice, narrower than the storage proposal. Not in `WORKPLAN.md` yet.

Claim C4: Wire format enables network transparency

Status: missing.

- ☐ `RemoteCap` shim forwarding one capnp method call over an established TCP capability.
- ☐ Working setup: two capOS QEMU guests, or one capOS guest plus a host capnp stub.
- ☐ QEMU smoke target proving end-to-end success and clean teardown.
- ☐ Documentation explicitly scoping this to "paper evidence" rather than a general network-transparency subsystem.

Closes via: a paper-scoped network-transparency-PoC slice. Naturally rides on the SSH/Telnet networking work in `docs/backlog/runtime-network-shell.md`.

Claim C5: Practical kernel verification at this size

Status: strong, needs distillation.

- ☒ Bounded Kani gates: cap-table, frame-bitmap, transfer rollback, resource accounting.
- ☒ Loom ring model.
- ☒ Miri host-test path.
- ☒ Panic-surface inventory.
- ☒ Dependency policy (`cargo-deny`, `cargo-audit`).
- ☒ Trusted build inputs inventory.
- ☐ Ring protocol Kani proof (currently Loom-only) – Tier-2 strengthener.
- ☐ Concise summary table for the paper: invariant, tool, scope, bounds.

Closes via: `docs/backlog/security-verification.md` plus a paper-section distillation pass.

Cross-cutting strengtheners

Not blocking the thesis, but materially raise quality:

- ☐ **SSH Shell Gateway** end-to-end. Paused behind the selected Session-Bound Invocation Context milestone, but still the concrete external-facing artifact for §6 when remote shell work resumes.
- ☐ **Full concurrent SMP scheduling.** Removes a single-CPU asterisk.
- ☐ **One non-toy demo** beyond Adventure / First Chat: storage-backed or federated, depending on which composes with C3 / C4.

How to update this file

Update in the same task that lands the artifact. When ticking a checkbox:

1. Reference the closing commit or branch.
2. If the closing artifact reframes the claim, update `plan.md` and `outline.md` in the same task.
3. If a Tier-1 gap closes, revisit `WORKPLAN.md` to verify paper-driven sequencing still matches the next mainline slice.

Backlog

Detailed task decompositions for work that is not useful in mandatory agent context.

Start from `WORKPLAN.md` to choose the active task. Then read only the backlog file linked for that task area.

- SMP Phase C
- Session-Bound Invocation Context
- Service Object Identity Migration
- Stage 6 Capability Semantics
- Runtime, Networking, And Shell
- Go VirtualMemory Contract
- Hardware, Boot, And Storage
- Security And Verification
- Local Users, Storage, And Policy
- Shared-Service Demos
- Paperclips Terminal Demo
- Aurelian Frontier
- Run Targets, Init Mandate, And Default-Run Integration

Proposal Index

This page classifies proposal documents by current role so readers do not confuse implemented behavior, active design direction, future architecture, and rejected alternatives.

The sidebar nests long proposal documents under this index so the public site opens as a current-system manual instead of an archive dump. Use this table as the first status checkpoint before opening a long proposal.

Active or Near-Term

- **Proposal:** Service Architecture
  - **Status:** Partially implemented
  - **Purpose:** Defines authority-at-spawn, service composition, exported capabilities, and the init-owned service graph direction.
- **Proposal:** Session-Bound Invocation Context
  - **Status:** Partially implemented
  - **Purpose:** Replaces caller-selected endpoint identity and the superseded service-object migration with one immutable session context per process, privacy-preserving endpoint caller-session metadata, and broker-granted service roots/facets. Core gates landed: process-session invariant, endpoint caller-session metadata, stale normal endpoint rejection, transfer scopes, disclosure gating, chat session-keyed membership, and Aurelian player state keyed by live caller session. Broader shared-service cleanup remains.
- **Proposal:** Storage and Naming
  - **Status:** Accepted design

  - **Purpose:** Defines capability-native storage, namespaces, boot-package structure, and future persistence instead of a global filesystem.
- **Proposal:** Error Handling
  - **Status:** Implemented
  - **Purpose:** Defines the implemented two-level transport/application error model and the current CQE transport error namespace.
- **Proposal:** Security and Verification
  - **Status:** Partially implemented
  - **Purpose:** Defines the security review vocabulary, trust-boundary checklist, and practical verification tracks used by capOS.
- **Proposal:** mdBook Documentation Site
  - **Status:** Partially implemented
  - **Purpose:** Defines the documentation site structure, status vocabulary, and curation rules for architecture, proposal, security, and research pages.
- **Proposal:** capOS Repository Harness Engineering
  - **Status:** Future design
  - **Purpose:** Applies OpenAI-style harness engineering to the capOS repository through agent-facing maps, run-target inventories, proposal metadata, decision records, compiled knowledge, and workflow evals.
- **Proposal:** SMP
  - **Status:** Accepted design
  - **Purpose:** Defines the selected per-CPU Phase A direction plus later AP startup, multi-core scheduler, and TLB shootdown work.
- **Proposal:** Ring v2 For Full SMP
  - **Status:** Future design
  - **Purpose:** Defines per-thread capability rings, completion routing, and SQPOLL ownership as the target transport model for full SMP.
- **Proposal:** Tickless and Realtime Scheduling
  - **Status:** Future design
  - **Purpose:** Defines staged tickless idle, SQPOLL nohz CPU isolation, request deadline metadata, scheduling-context CPU-time authority, donation, and admitted realtime islands.
- **Proposal:** System Configuration and Operator Extensibility
  - **Status:** Partially implemented
  - **Purpose:** Slices 1 and 2 have landed: the proposal, `WORKPLAN.md` pointer, doc index entry, per-user `~/.capos-tools` cache, `cue/defaults/` package, `@tag(user)` host-name injection, `system.local.cue` overlay hook, `mkmanifest --package` and tag pass-through, generated-check tooling updates, and the operator configuration how-to. Slice 3 is in progress: spawn, shell, terminal, telnet, login/account, credential, SSH, smoke, session-context, restricted-launcher, chat, IPC, memory-object, service-object, and focused networking manifests are packaged; the remaining focused-proof `system-*.cue` manifests still need migration. Slice 4 is implemented through `mkmanifest cue-to-capnp` for schema-aware CUE-authored data conversion.

Future Architecture

- **Proposal:** Networking
  - **Status:** Partially implemented
  - **Purpose:** Plans the in-kernel QEMU virtio-net smoke and the later userspace NIC, network stack, and socket capability architecture.
- **Proposal:** libcapos-service
  - **Status:** Future design
  - **Purpose:** Defines a userspace service framework above `capos-rt` for lifecycle, endpoint serve loops, readiness, shutdown/drain, request/session context, metrics, and resource budgeting hooks, starting with terminal/networking lifecycle rather than HTTP.
- **Proposal:** Resource Accounting and Quotas
  - **Status:** Partially implemented / Future architecture
  - **Purpose:** Generalizes existing per-process `ResourceLedger` mechanisms to cross-service resource profiles, ledgers of record, quota donation, and fail-closed reservation semantics.
- **Proposal:** OOM Handling and Swap
  - **Status:** Future design
  - **Purpose:** Defines memory-pressure policy, explicit OOM outcomes, budgeted anonymous memory, and optional encrypted swap without an ambient OOM killer.
- **Proposal:** Cryptography and Key Management
  - **Status:** Future design
  - **Purpose:** Defines symmetric/private key, key-source, signing, encryption, and vault capabilities used by TLS, signing, instance identity, and audit.
- **Proposal:** Volume Encryption
  - **Status:** Future design
  - **Purpose:** Defines encryption-at-rest for system and user volumes, including passphrase, recovery, cloud KMS, and measured-boot-backed key sources.
- **Proposal:** Userspace Binaries
  - **Status:** Partially implemented
  - **Purpose:** Describes native userspace binaries, `capos-rt`, language support, POSIX compatibility, and runtime authority handling.
- **Proposal:** Go Runtime
  - **Status:** Future design
  - **Purpose:** Plans a custom `GOOS=capos` path, runtime services, memory growth, TLS, scheduling, and network integration for Go.
- **Proposal:** Lua Scripting
  - **Status:** Future design
  - **Purpose:** Defines Lua as an ordinary capability-scoped userspace runner with curated libraries, exact grants, and no ambient shell or POSIX authority.
- **Proposal:** Shell
  - **Status:** Partially implemented
  - **Purpose:** Describes native, agent-oriented, and POSIX shell models over explicit capabilities instead of ambient paths.

- **Proposal:** SSH Shell Gateway
  - **Status:** Partially implemented
  - **Purpose:** Defines production remote CLI shell access through SSH while preserving the same TerminalSession and broker-issued shell-bundle boundary proven by the Telnet shell demo; focused QEMU proofs now cover the non-production `SshHostKey`, manifest-seeded `AuthorizedKeyStore`, public-key session bridge, unsupported-feature policy table, scoped listener, restricted shell launcher, and a bounded plain-TCP terminal-host wiring slice. Full OpenSSH transport remains future work.
- **Proposal:** Telnet over TLS Shell
  - **Status:** Future design
  - **Purpose:** Defines a peer production remote-shell path to the SSH gateway: TLS 1.3 over the existing Telnet `TerminalSession` handoff, with mTLS client certificates as the recommended user-auth path and `CredentialStore` passwords as fallback. Reuses the project's PKI/ACME/cert-rotation track instead of inventing a parallel SSH-only key-management story. Smaller protocol surface than SSH; different operational profile, not a substitute.
- **Proposal:** Language Models and Agent Runtime
  - **Status:** Future design
  - **Purpose:** Defines language-model and embedder capabilities, local and remote backends, capOS-side agent runners, and browser-agent UI orchestration through gateway-enforced tool execution.
- **Proposal:** capOS-Hosted Agent Swarms
  - **Status:** Future design
  - **Purpose:** Defines OpenClaw-like hosted personal agents, swarms, harness controls, task workspaces, agent memory/wiki services, MCP/A2A-style adapters, and the research agenda for capability-scoped background agents.
- **Proposal:** Realtime Voice Agent Shell
  - **Status:** Future design
  - **Purpose:** Extends the agent-shell path for native realtime audio models, direct browser provider media, and browser-agent UI sessions while preserving broker-mediated tool execution and web-shell session boundaries.
- **Proposal:** Interactive Command Surfaces
  - **Status:** Future design
  - **Purpose:** Defines structured command sessions for native interactive applications so familiar text commands compile to typed invocations instead of application-owned `StdIO` parsers.
- **Proposal:** Userspace Authority Broker
  - **Status:** Future design
  - **Purpose:** Proposes moving shell bundle policy out of the kernel and making shutdown an init-owned lifecycle control capability granted only after login.
- **Proposal:** Aurelian Frontier
  - **Status:** Partially implemented
  - **Purpose:** Capability-native persistent-world RPG on a Roman-inspired magical frontier. See the runnable proof slice for current commands and coverage.
- **Proposal:** Contributor Quest Mechanics

  - **Status:** Future design
  - **Purpose:** Defines a post-adventure follow-up where maintainer-witnessed open-source contributions can mint cosmetic badges, states, decorations, and bounded game perks without granting repository or OS authority.
- **Proposal:** Public Release and Maintainer Boundaries
  - **Status:** Future design
  - **Purpose:** Defines the release posture, security-audit disclaimer, issue/PR intake limits, maintainer-load boundaries, and the adventure-repository-split and git-history-rewrite hygiene gates required before making the repository public. Defers the long-term sibling-repository rule to the Repository Composition proposal.
- **Proposal:** Repository Composition
  - **Status:** Future design
  - **Purpose:** Defines the scope rule for the capOS core repository, the list of tracks (adventure, whitepaper, public site, userspace netstack, remote-access services, protocol stacks, language runtimes, GPU, agent shell, cloud images, volume crypto) that should ship as siblings, the when-to-split criteria, the cross-repository mechanics, and the intended `cap-os-dev` GitHub organization placement.
- **Proposal:** Boot to Shell
  - **Status:** Partially implemented
  - **Purpose:** Defines text-only console and web-terminal login/setup, password verifier and passkey authentication, and the authenticated native shell launch path after manifest execution, terminal input, native shell, session, broker, audit, and credential-storage prerequisites are credible.
- **Proposal:** System Info Capability
  - **Status:** Phase 1 + Phase 2 implemented
  - **Purpose:** Unifies the system-wide informational capability (MOTD today; hostname, help topics, manpages later), moves banner printing into the shell, and has `AuthorityBroker.shellBundle` mint `SystemInfo` plus profile-scoped `chat`/`adventure` service endpoint caps for operator shells. Guest and anonymous shells receive no service endpoints by default.
- **Proposal:** System Monitoring
  - **Status:** Future design
  - **Purpose:** Defines capability-scoped logs, metrics, health, traces, crash records, and audit/status views.
- **Proposal:** System Performance Benchmarks
  - **Status:** Future design
  - **Purpose:** Defines correctness-gated primitive, workload, and user-story benchmarks for comparing capOS with other operating systems without distorting capability semantics.
- **Proposal:** User Identity and Policy
  - **Status:** Partially implemented
  - **Purpose:** Defines users, sessions, guest profiles, and policy layers for RBAC, ABAC, and MAC over capability grants. Current implementation has anonymous/operator/guest `UserSession` metadata, bootstrap credential/session flows, broker-issued shell bundles, and seed-account

configuration; durable accounts, external bindings, session revocation, quotas, and broader ABAC/MAC remain future work.

- **Proposal:** Delegated Subject Context
  - **Status:** Future design
  - **Purpose:** Defines bounded act-on-behalf-of subject context as separate from capability transfer and from the selected session-bound invocation context milestone.
- **Proposal:** Cloud Metadata
  - **Status:** Future design
  - **Purpose:** Describes cloud instance bootstrap through metadata/config-drive capabilities and manifest deltas.
- **Proposal:** Cloud Deployment
  - **Status:** Future design
  - **Purpose:** Plans hardware abstraction, cloud VM support, storage/network boot dependencies, and later aarch64 deployment work.
- **Proposal:** Live Upgrade
  - **Status:** Future design
  - **Purpose:** Defines service replacement without dropping capabilities or in-flight calls through retargeting and quiesce/resume protocols.
- **Proposal:** GPU Capability
  - **Status:** Future design
  - **Purpose:** Sketches capability-oriented GPU, CUDA, memory, and driver isolation models.
- **Proposal:** capOS As A Robot Brain
  - **Status:** Future design
  - **Purpose:** Defines capability-oriented robotics service graphs, actuator gateways, safety monitors, realtime control islands, and ROS 2/micro-ROS/MAVLink/OPC UA bridges.
- **Proposal:** Formal MAC/MIC
  - **Status:** Future design
  - **Purpose:** Defines a formal mandatory-access and mandatory-integrity model plus future proof obligations.
- **Proposal:** Browser/WASM
  - **Status:** Future design
  - **Purpose:** Explores running capOS concepts in a browser using WebAssembly and worker-per-process isolation.
- **Proposal:** Certificates and TLS
  - **Status:** Future design
  - **Purpose:** Defines X.509, trust stores, Certificate Transparency, OCSP, pinning, ACME, and TLS configuration as capability-native interfaces on top of the key-management primitives.
- **Proposal:** OIDC and OAuth2
  - **Status:** Future design
  - **Purpose:** Defines federated login, OAuth2 clients, typed token capabilities, JWKS, DPoP, token-exchange workload identity federation, and the broker integration for scopes/claims as ABAC input.

Rejected or Superseded

- **Proposal:** Endpoint Badges as Service Identity
  - **Status:** Rejected
  - **Purpose:** Post-mortem for the seL4-style endpoint badge identity model that was superseded by Service Object Capabilities, then by Session-Bound Invocation Context.
- **Proposal:** Service Object Capabilities
  - **Status:** Superseded
  - **Purpose:** Historical service-minted object capability model; the landed synthetic routing/lifecycle proof remains low-level coverage, but the active direction is Session-Bound Invocation Context.
- **Proposal:** Cap'n Proto SQE Envelope
  - **Status:** Rejected
  - **Purpose:** Records why ring SQEs stay fixed-layout transport records instead of becoming Cap'n Proto messages themselves.
- **Proposal:** Sleep(INF) Process Termination
  - **Status:** Rejected
  - **Purpose:** Records why infinite sleep should not replace explicit process termination, while preserving typed status and future `sys_exit` removal as separate lifecycle work.

Maintenance

When a proposal becomes implemented, rejected, or stale, update this index in the same change that changes the proposal or corresponding implementation. Long proposal files may describe target behavior; this index is the first status checkpoint before a reader opens those documents.

Papers

Long-form research write-ups produced from the capOS codebase. Each paper is typeset with Typst from sources under `papers/<slug>/` in the repository and published as a PDF alongside this site.

Schema-as-ABI: Typed Capabilities and Ring-Transport Dispatch in capOS

Download PDF

A pre-evidence draft describing the schema-as-ABI thesis: Cap'n Proto schemas acting as kernel ABI, access-control mechanism, IPC wire format, and (planned) persistence and network-transparency substrate, layered over a shared-memory SQ/CQ ring with a two-syscall surface (`exit` and `cap_enter`).

The draft separates closed contributions (capability ring transport, exactly-once accounting rollback, capability lifecycle, the verification stack) from evidence-gated claims that depend on outstanding artifacts (C1 service-object migration, C2 measurement run, C3 persistence proof-of-concept, C4 network-transparency proof-of-concept). Sections that depend on missing artifacts

are flagged with `TODO` admonitions naming the gap and the entry in `docs/paper/evidence-gaps.md` that closes them.

Source: `papers/schema-as-abi/main.typ` in the repository. Build locally with `make paper`; the same target runs in `make cloudflare-pages-build` and publishes the PDF at the link above.

Research: Capability-Based and Microkernel Operating Systems

Survey of existing systems to inform capOS design decisions across IPC, scheduling, capability model, persistence, VFS, and language support.

The sidebar nests the long reports under this index. Read the consequences below first; open individual reports only when their design context is relevant.

Design consequences for capOS

- Keep the flat generation-tagged capability table; seL4-style CNode hierarchy is not needed until delegation patterns demand it.
- Treat the typed Cap'n Proto interface as the permission boundary; avoid a parallel rights-bit system that would drift from schema semantics.
- Continue the ring transport plus direct-handoff IPC path, with shared memory reserved for bulk data once `SharedBuffer`/`MemoryObject` exists.
- Treat seL4-style endpoint badges as historical receiver metadata, not as the active service identity model; use move/copy transfer descriptors, object-epoch revocation, and session-bound invocation context to make authority delegation explicit and reviewable.
- Keep persistence explicit through Store/Namespace capabilities; do not adopt EROS-style transparent global checkpointing as a kernel baseline.
- Push POSIX compatibility and VFS behavior into libraries and services rather than adding a kernel global filesystem namespace.
- Add resource donation, scheduling-context donation, notification objects, and runtime/thread primitives only when the corresponding service or runtime path needs them.
- Use Pingora-style lifecycle frameworks only above the capability transport: userspace service libraries can provide phase hooks, per-request context, readiness, graceful shutdown, retry policy, and observability, while kernel interfaces remain narrow typed capabilities with explicit authority.

Individual deep-dive reports:

- seL4 – formal verification, CNode/CSpace, IPC fastpath, MCS scheduling
- Fuchsia/Zircon – handles with rights, channels, VMARs/VMOs, ports, FIDL vs Cap'n Proto
- Plan 9 / Inferno – per-process namespaces, 9P protocol, file-based vs capability-based interfaces
- EROS / CapROS / Coyotos – persistent capabilities, single-level store, checkpoint/restart
- Genode – session routing, VFS plugins, POSIX compat, resource trading, Sculpt OS
- LLVM target customization – target triples, TLS models, Go runtime requirements

- Cap'n Proto error handling – protocol, schema, and Rust crate error behavior used by the capOS error model
- OS error handling – error patterns in capability systems and microkernels used by the capOS error model
- IX-on-capOS hosting – clean integration of IX package/build model via MicroPython control plane, native template rendering, Store/Namespace, and build services
- Out-of-kernel scheduling – whether scheduler policy can move to user space, and which dispatch/enforcement mechanisms must stay in kernel
- Completion rings and threaded runtimes – completion ownership, `io_uring`, futex, and IOCP precedents for capOS's full-SMP ring/threading ABI
- x2APIC and APIC virtualization – x2APIC backend direction, QEMU/KVM validation constraints, and why the current xAPIC MMIO LAPIC branch should remain the Phase C foundation
- NO_HZ, SQPOLL, and realtime scheduling – Linux NO_HZ, clocksource/clockevent, CPU isolation/housekeeping, io_uring SQPOLL, SCHED_DEADLINE, PREEMPT_RT, and seL4 MCS grounding for capOS tickless idle, SQPOLL nohz, scheduling contexts, and realtime islands
- Pingora – phase-oriented service framework design, operational lifecycle, pooling/retry lessons, and why capOS should borrow the userspace library shape without importing Pingora's HTTP or process model. The concrete capOS follow-up is libcapos-service, starting with terminal/networking lifecycle rather than HTTP.
- Multimedia pipeline latency – PipeWire and JACK lessons for a capOS media graph optimized for the minimal possible guaranteed-stable stack latency, explicit latency ranges, admitted realtime islands, and xrun/deadline telemetry
- Realtime multimodal agent APIs – OpenAI Realtime, Google AI Gemini Live API, and Vertex AI Live API implications for capOS voice agent-shell, realtime media sessions, tool-call gating, and provider adapters
- Hosted agent harnesses – OpenClaw-like harness controls, hosted agent swarms, LLM-maintained wiki memory, schema-guided reasoning, MCP/A2A-style adapters, and implications for capability-scoped capOS agent services
- Game mechanics prior art – Stardew Valley, EVE Online, and Evil Islands mechanics translated into capability-shaped Aurelian Frontier calendar, market, construction, and combat tasks
- Robotics realtime control – ROS 2, micro-ROS, ros2_control, seL4 MCS, PREEMPT_RT, Xenomai, Orocos, Nav2, PX4, ArduPilot, Autoware, and OPC UA lessons for using capOS as a robot brain with explicit actuator authority and admitted realtime islands

---

Cross-Cutting Analysis

1. Capability Table Design

All surveyed systems store capabilities as process-local references to kernel objects. The key design variable is how capabilities are organized.

- **System:** seL4
  - **Structure:** Tree of CNodes (power-of-2 arrays with guard bits)

  - **Lookup:** O(depth)
  - **Delegation:** Subtree (grant CNode cap)
  - **Revocation:** CDT (derivation tree), transitive
- **System:** Zircon
  - **Structure:** Flat per-process handle table
  - **Lookup:** O(1)
  - **Delegation:** Transfer through channels (move)
  - **Revocation:** Close handle; refcount; no propagation
- **System:** EROS
  - **Structure:** 32-slot nodes forming trees
  - **Lookup:** O(depth)
  - **Delegation:** Node key passing
  - **Revocation:** Forwarder keys (O(1) rescind)
- **System:** Genode
  - **Structure:** Kernel-enforced capability references
  - **Lookup:** O(1)
  - **Delegation:** Parent-mediated session routing
  - **Revocation:** Session close
- **System:** capOS
  - **Structure:** Flat table with generation-tagged `CapId`, hold-edge metadata, and `Arc<dyn CapObject>` backing
  - **Lookup:** O(1)
  - **Delegation:** Manifest exports plus copy/move transfer descriptors through Endpoint IPC
  - **Revocation:** Local release/process exit plus object-epoch revocation for child-local grants

Recommendation for capOS: Keep the flat table. It is simpler than seL4's CNode tree and sufficient for capOS's use cases. Augment each entry with:

1. **Badge** (from seL4) – u64 value delivered to the server on invocation, allowing a server to distinguish callers without separate capability objects.
2. **Generation counter** (from Zircon) – upper bits of CapId detect stale references after a slot is reused. (Implemented.)
3. **Epoch** (from EROS) – per-object revocation epoch. Incrementing the epoch invalidates all outstanding references. O(1) revoke, O(1) check.

Not adopted: per-entry rights bitmask. Zircon and seL4 use rights bitmasks (READ/WRITE/EXECUTE) because their handle/syscall interfaces are untyped. capOS uses Cap'n Proto typed interfaces where the schema defines what methods exist. Method-level access control is the interface itself – to restrict what a caller can do, grant a narrower capability (a wrapper `CapObject` that exposes fewer methods). A parallel rights system would create an impedance mismatch: generic flags (READ/WRITE) mapped arbitrarily onto typed methods. Meta-rights for the capability reference itself (TRANSFER/DUPLICATE) may be added when Stage 6 IPC needs them. See capability-model.md for the full rationale.

2. IPC Design

IPC is the most performance-critical kernel mechanism. Every capability invocation across processes goes through it.

- **System:** seL4
  - **Model:** Synchronous endpoint, direct context switch
  - **Latency (round-trip):** ~240 cycles (ARM), ~400 cycles (x86)
  - **Bulk data:** Shared memory (explicit)
  - **Async:** Notification objects (bitmask signal/wait)
- **System:** Zircon
  - **Model:** Channels (async message queue, 64KiB + 64 handles)
  - **Latency (round-trip):** ~3000-5000 cycles
  - **Bulk data:** VMOs (shared memory)
  - **Async:** Ports (signal-based notification)
- **System:** EROS
  - **Model:** Synchronous domain call
  - **Latency (round-trip):** ~2x L4
  - **Bulk data:** Through address space nodes
  - **Async:** None (synchronous only)
- **System:** Plan 9
  - **Model:** 9P over pipes (kernel-mediated)
  - **Latency (round-trip):** ~5000+ cycles
  - **Bulk data:** Large reads/writes (iounit)
  - **Async:** None (blocking per-fid)
- **System:** Genode
  - **Model:** RPC objects with session routing
  - **Latency (round-trip):** Varies by kernel (uses seL4/NOVA/Linux underneath)
  - **Bulk data:** Shared-memory dataspaces
  - **Async:** Signal capabilities

Recommendation for capOS: Continue the **dual-path** IPC design:

Fast synchronous path (seL4-inspired, for RPC): - When process A calls a capability in process B and B is blocked waiting, perform a **direct context switch** (A -> kernel -> B, no unrelated scheduler pick). The current single-CPU direct handoff is implemented. - Future fastpath work can transfer small messages (<64 bytes) through registers during the switch instead of copying through ring buffers.

Async submission/completion rings (io_uring-inspired, for batching): - SQ/CQ in shared memory for batched capability invocations. This is the current transport for CALL/RECV/RETURN/RELEASE/NOP. - Support SQE chaining for Cap'n Proto promise pipelining. - Signal/notification delivery through CQ entries (from Zircon ports). - User-queued CQ entries for userspace event loop integration.

Bulk data (Zircon/Genode-inspired): - `SharedBuffer` capability for zero-copy data transfer between processes. - Capnp messages for control plane; shared memory for data plane. - Critical for file I/O, networking, and GPU rendering.

3. Memory Management Capabilities

Zircon's VMO/VMAR model is the most mature capability-based memory design. The Go runtime proposal shows why these primitives are essential.

VirtualMemory capability (baseline implemented; still central for Go and advanced allocators):

```
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) -> ();
}
```

MemoryObject capability (needed for IPC bulk data, shared libraries). Zircon calls this concept a VMO (Virtual Memory Object); capOS uses the name **SharedBuffer** – see `docs/proposals/storage-and-naming-proposal.md` for the canonical interface definition.

```
interface MemoryObject {
    read @0 (offset :UInt64, count :UInt64) -> (data :Data);
    write @1 (offset :UInt64, data :Data) -> ();
    getSize @2 () -> (size :UInt64);
    createChild @3 (offset :UInt64, size :UInt64, options :UInt32)
        -> (child :MemoryObject);
}
```

4. Scheduling

- **System:** seL4 (MCS)
  - **Model:** Scheduling Contexts (budget/period/priority) + Reply Objects
  - **Priority inversion solution:** SC donation through IPC (caller's SC transfers to callee)
  - **Temporal isolation:** Yes (budget enforcement per SC)
- **System:** Zircon
  - **Model:** Fair scheduler with profiles (deadline, capacity, period)
  - **Priority inversion solution:** Kernel-managed priority inheritance
  - **Temporal isolation:** Profiles provide some isolation
- **System:** Genode
  - **Model:** Delegated to underlying kernel (seL4/NOVA/Linux)
  - **Priority inversion solution:** Depends on kernel
  - **Temporal isolation:** Depends on kernel
- **System:** Out-of-kernel policy
  - **Model:** Kernel dispatch/enforcement + user-space policy service
  - **Priority inversion solution:** Scheduling-context donation through IPC
  - **Temporal isolation:** Kernel-enforced budgets, user-chosen policy
- **System:** User-space runtimes
  - **Model:** M:N work stealing, fibers, async tasks over kernel threads

  - **Priority inversion solution:** Requires futexes, runtime cooperation, and OS-visible blocking events
  - **Temporal isolation:** Usually runtime-local only

Recommendation for capOS: Start with round-robin (already done). When implementing priority scheduling:

1. Add **scheduling context donation** for synchronous IPC: when process A calls process B, B inherits A's priority and budget. Prevents inversion through the capability graph.
2. Support **passive servers** (from seL4 MCS): servers without their own scheduling context that only run when called, using the caller's budget. Natural fit for capOS's service architecture.
3. Add **temporal isolation** (budget/period per scheduling context) for the cloud deployment scenario.

For moving scheduler policy out of the kernel, see Out-of-kernel scheduling. The key finding is a split between kernel dispatch/enforcement and user-space policy: dispatch, budget enforcement, and emergency fallback remain privileged, while admission control, budgets, priorities, CPU masks, and SQPOLL/core grants can be represented as policy managed by a scheduler service. Thread creation, thread handles, scheduling contexts, and park authority should be capability-based from the start; the remaining research task is measurement: compare generic capnp/ring calls against compact capability-authorized park-shaped operations before deciding the park hot-path encoding.

5. Persistence

- **System:** EROS/CapROS
  - **Model:** Transparent global checkpoint (single-level store)
  - **Consistency:** Strong (global snapshot)
  - **Application effort:** None (automatic)
- **System:** Plan 9
  - **Model:** User-mode file servers with explicit writes
  - **Consistency:** Per-file server
  - **Application effort:** Full (explicit save/load)
- **System:** Genode
  - **Model:** Application-level (services manage own persistence)
  - **Consistency:** Per-component
  - **Application effort:** Full
- **System:** capOS (planned)
  - **Model:** Content-addressed Store + Namespace caps
  - **Consistency:** Per-service
  - **Application effort:** Full (explicit capnp serialize)

Recommendation for capOS: Three phases, as informed by EROS:

1. **Explicit persistence** (current plan) – services serialize state to the Store capability as capnp messages. Simple, gives services control.

2. **Opt-in Checkpoint capability** – kernel captures process state (registers, memory, cap table) as capnp messages stored in the Store. Enables process migration and crash recovery for services that opt in.
3. **Coordinated checkpointing** – a coordinator service orchestrates consistent snapshots across multiple services.

Persistent capability references (from EROS + Cap'n Proto):

```
struct PersistentCapRef {
    interfaceId @0 :UInt64;
    objectId @1 :UInt64;
    permissions @2 :UInt32;
    epoch @3 :UInt64;
}
```

Do NOT implement EROS-style transparent global persistence. The kernel complexity is enormous, debuggability is poor, and Cap'n Proto's zero-copy serialization already provides near-equivalent benefits for explicit persistence.

6. Namespace and VFS

Plan 9's per-process namespace is the closest analog to capOS's per-process capability table. The key insight: Plan 9's `bind`/`mount` with union semantics provides composability that capOS's current Namespace design lacks.

Recommendation: Extend `Namespace` with union composition:

```
enum UnionMode { replace @0; before @1; after @2; }

interface Namespace {
    resolve @0 (name :Text) -> (hash :Data);
    bind @1 (name :Text, hash :Data) -> ();
    list @2 () -> (names :List(Text));
    sub @3 (prefix :Text) -> (ns :Namespace);
    union @4 (other :Namespace, mode :UnionMode) -> (merged :Namespace);
}
```

VFS as a library (from Genode): `libcapos-posix` should be an in-process library that translates POSIX calls to capability invocations. Each POSIX process receives a declarative mount table (capnp struct) mapping paths to capabilities. No VFS server needed.

FileServer capability (from Plan 9): For resources that are naturally file-like (config trees, debug introspection, `/proc`-style interfaces), provide a `FileServer` interface. Not universal (as in Plan 9) but available where the file metaphor fits.

7. Resource Accounting

Genode's session quota model addresses a gap in capOS: without resource accounting, a malicious client can exhaust a server's memory by creating many sessions.

Recommendation: Session-creating capability methods should accept a resource donation parameter:

```
interface NetworkManager {
    createTcpSocket @0 (bufferPages :UInt32) -> (socket :TcpSocket);
}
```

The client donates buffer memory as part of the session creation. The server allocates from donated resources, not its own.

8. Language Support Roadmap
From the LLVM research, the recommended order:

- **Step:** 1
  - **What:** Custom target JSON (`x86_64-unknown-capos`)
  - **Blocks:** Done for booted userspace crates
- **Step:** 2
  - **What:** VirtualMemory capability
  - **Blocks:** Done for baseline map/unmap/protect; Go allocator glue remains
- **Step:** 3
  - **What:** TLS support (PT_TLS parsing, FS base save/restore)
  - **Blocks:** Done for static ELF processes and current-process ThreadControl; per-thread TLS remains
- **Step:** 4
  - **What:** Park authority capability + measured ABI
  - **Blocks:** Go threads, pthreads
- **Step:** 5
  - **What:** Timer capability (monotonic clock)
  - **Blocks:** Done for monotonic now/sleep; wall-clock and event timers remain future work
- **Step:** 6
  - **What:** Go Phase 1: minimal `GOOS=capos` (single-threaded)
  - **Blocks:** Runtime capability checkpoint done; Go fork remains
- **Step:** 7
  - **What:** Kernel threading
  - **Blocks:** Go GOMAXPROCS>1
- **Step:** 8
  - **What:** C toolchain + `libcapos`
  - **Blocks:** C programs, musl
- **Step:** 9
  - **What:** Go Phase 2: multi-threaded + concurrent GC
  - **Blocks:** Go network services
- **Step:** 10
  - **What:** Go Phase 3: network poller
  - **Blocks:** `net/http` on capOS

Key decisions: - Keep `x86_64-unknown-none` for kernel, `x86_64-unknown-capos` for userspace. - Use `local-exec` TLS model (static linking, no dynamic linker). - Implement park as capability-authorized from the start. Because it operates on memory addresses and must be fast, measure

generic capnp/ring calls against a compact capability-authorized operation before fixing the ABI.
- Go can start with cooperative-only preemption (no signals).

Recommendations by Roadmap Stage

Stage 5: Scheduling

- **Source:** Zircon
  - **Recommendation:** Generation counter in CapId (stale reference detection)
  - **Priority:** Done
- **Source:** seL4
  - **Recommendation:** Add notification objects (lightweight bitmask signal/wait)
  - **Priority:** Medium
- **Source:** LLVM
  - **Recommendation:** Custom target JSON for userspace (`x86_64-unknown-capos`)
  - **Priority:** Done
- **Source:** LLVM
  - **Recommendation:** Per-thread TLS state for Go/threading
  - **Priority:** Medium

Stage 6: IPC and Capability Transfer

- **Source:** seL4
  - **Recommendation:** **Direct-switch IPC** for synchronous cross-process calls
  - **Priority:** Done baseline
- **Source:** seL4
  - **Recommendation:** Badge field on capability entries for server-visible caller identity
  - **Priority:** Historical / rejected as service identity; see Rejected: Endpoint Badges as Service Identity
- **Source:** Zircon
  - **Recommendation:** Move semantics for capability transfer through IPC
  - **Priority:** Done
- **Source:** Zircon
  - **Recommendation:** MemoryObject capability (shared memory for bulk data)
  - **Priority:** Done baseline
- **Source:** EROS
  - **Recommendation:** Epoch-based revocation (O(1) revoke, O(1) check)
  - **Priority:** High
- **Source:** Zircon
  - **Recommendation:** Sideband capability-transfer descriptors and result-cap records
  - **Priority:** Done baseline
- **Source:** Genode
  - **Recommendation:** SharedBuffer capability for data-plane transfers
  - **Priority:** High
- **Source:** Plan 9

  - **Recommendation:** Promise pipelining (SQE chaining in async rings)
  - **Priority:** Medium
- **Source:** Genode
  - **Recommendation:** Session quotas / resource donation on session creation
  - **Priority:** Medium
- **Source:** seL4
  - **Recommendation:** Scheduling context donation through IPC
  - **Priority:** Medium
- **Source:** Plan 9
  - **Recommendation:** Namespace union composition (before/after/replace)
  - **Priority:** Low

Post-Stage 6 / Future

- **Source:** seL4
  - **Recommendation:** MCS scheduling (passive servers, temporal isolation)
  - **Priority:** When needed
- **Source:** EROS
  - **Recommendation:** Opt-in Checkpoint capability for process persistence
  - **Priority:** When needed
- **Source:** Genode
  - **Recommendation:** Dynamic manifest reconfiguration at runtime
  - **Priority:** When needed
- **Source:** Plan 9
  - **Recommendation:** exportfs-pattern capability proxy for network transparency
  - **Priority:** When needed
- **Source:** EROS
  - **Recommendation:** PersistentCapRef struct in capnp for storing capability graphs
  - **Priority:** When needed
- **Source:** seL4
  - **Recommendation:** Rust-native formal verification (track Verus/Prusti)
  - **Priority:** Long-term

Design Decisions Validated

Several capOS design choices are validated by this research:

1. **Cap'n Proto as the universal wire format.** Superior to FIDL (random access, zero-copy, promise pipelining, persistence-ready). The right choice. See zircon.md Section 5.
2. **Flat capability table.** Simpler than seL4's CNode tree, sufficient for capOS. Only add complexity (CNode-like hierarchy) if delegation patterns demand it. See sel4.md Section 4.
3. **No ambient authority.** Every surveyed capability OS confirms this is essential. EROS proved confinement. seL4 proved integrity. capOS has this by design.

4. **Explicit persistence over transparent.** EROS's single-level store is elegant but the kernel complexity is enormous. Cap'n Proto zero-copy gives most of the benefits. See eros-capros-coyotos.md Section 6.
5. **io_uring-inspired async rings.** Better than Zircon's port model for capOS (operation-based > notification-based). See zircon.md Section 4.
6. **VFS as library, not kernel feature.** Genode's approach, matched by capOS's planned `libcapos-posix`. See genode.md Section 3.
7. **No fork().** Genode has operated without fork() for 15+ years, proving it unnecessary. See genode.md Section 4.

Design Gaps Identified

1. **Bulk data path is only a substrate.** Copying capnp messages through the kernel works for control but not for file/network/GPU data. MemoryObject now provides the mapped-frame substrate; service-facing SharedBuffer APIs remain future Stage 6+ work.
2. **Resource accounting is partially unified.** The authority-accounting design exists, and VirtualMemory plus FrameAllocator/MemoryObject frame grants now charge the process `ResourceLedger::frame_grant_pages` counter. Future shared-buffer, DMA, log-volume, and CPU-budget resources still need the same treatment.
3. **No notification primitive.** seL4 notifications (lightweight bitmask signal/wait) are needed for interrupt delivery and event notification without full capnp message overhead.
4. **No per-thread TLS object yet.** Static ELF TLS, context-switch FS-base state, and current-process ThreadControl exist, but future user threads still need independently settable FS bases per thread.

References

See individual deep-dive reports for full reference lists. Key primary sources:

- Klein et al., "seL4: Formal Verification of an OS Kernel," SOSP 2009
- Lyons et al., "Scheduling-context capabilities," EuroSys 2018
- Shapiro et al., "EROS: A Fast Capability System," SOSP 1999
- Shapiro & Weber, "Verifying the EROS Confinement Mechanism," IEEE S&P 2000
- Pike et al., "The Use of Name Spaces in Plan 9," OSR 1993
- Feske, "Genode Foundations" (genode.org/documentation)
- Fuchsia Zircon kernel documentation (fuchsia.dev)

Topics Index

This page is generated from document front matter fields during `mdbook` builds: - `status` - `description` - `topics`

Quick Orientation

- Backlog — *Detailed task decompositions.*
- Build, Boot, and Test — *Build, ISO, QEMU, host-test commands.*
- capOS Repository Harness Engineering — *Repository-local harness engineering for making capOS legible, checkable, and safer for long-running coding agents.*
- Changelog — *Historical milestone reports.*
- Current Status — *What works, what is partial.*
- Design Risks and Open Questions — *Consolidated index of long-horizon design risks.*
- Documentation Bundle — *Single-page generated documentation bundle.*
- Introduction — *Top-level book entry.*
- Proposal Index — *Proposal status table.*
- Repository Map — *Source-tree subsystem index.*
- Research Index — *Design consequences pulled from the survey.*
- Roadmap — *Long-term architectural plan.*
- What capOS Is — *One-page system model.*

Capabilities, IPC, and Authority

- Authority Accounting — *Authority accounting rules for capability transfer and resource charges.*
- Cap'n Proto Error Handling — *Prior-art on capnp-rpc error semantics.*
- Capability Model — *Core capability object model, cap tables, schema interface IDs, grants, receiver metadata, and transfer.*
- Capability Ring — *Shared-memory capability ring ABI, dispatch paths, and completion semantics.*
- Delegated Subject Context — *Future delegated-subject and act-on-behalf-of capability model.*
- Error Handling — *Transport and application error model for capability calls and CQE results.*
- IPC and Endpoints — *Endpoint IPC, capability transfer, direct handoff, and shared-memory data paths.*
- OS Error Handling — *Cross-OS error-model comparison.*
- Rejected: Cap'n Proto SQE Envelope — *Rationale for keeping ring SQEs fixed-layout instead of Cap'n Proto envelopes.*
- Rejected: Endpoint Badges as Service Identity — *Post-mortem of the rejected seL4-style endpoint badge service identity model.*
- Resource Accounting and Quotas — *Resource profiles, quota ledgers, donation, reservation, and fail-closed accounting semantics.*
- Service Architecture — *Capability-based service composition, authority-at-spawn, exports, and service graph policy.*
- Service Object Identity Migration — *Superseded large-chunk migration plan for service object identity, retained as historical context after the active direction changed to session-bound invocation context.*
- Session-Bound Invocation Context — *Implementation plan for one-session-per-process invocation context and session-keyed shared services.*
- Session-Bound Invocation Context — *Session-bound invocation context and privacy-aware disclosure model replacing service-object identity migration.*
- Stage 6 Capability Semantics — *Stage 6 capability work.*

- Superseded: Service Object Capabilities — *Superseded service-minted object capability model that was replaced by session-bound invocation context.*
- System Info Capability — *SystemInfo capability for MOTD, host metadata, help topics, and shell bundle integration.*
- Userspace Authority Broker — *Userspace shell-bundle broker and lifecycle-control authority model.*

Boot, Manifests, and Init

- Boot Flow — *Kernel boot, manifest handoff, init launch, and QEMU boot-proof flow.*
- Boot to Shell — *Login, setup, session, credential, and broker path from boot into the native shell.*
- Cloud Metadata — *Cloud metadata and config-drive bootstrap through scoped configuration capabilities.*
- Configuration — *How operators extend the default capOS boot manifest with a gitignored `system.local.cue` overlay and convert CUE-authored data to specified Cap'n Proto schemas.*
- Hardware, Boot, and Storage — *Hardware bring-up backlog.*
- Manifest and Service Startup — *Manifest encoding, service graph validation, bootstrap grants, and init-side spawning.*
- Run Targets, Init Mandate, and Default-Run Integration — *Run-target governance.*
- System Configuration and Operator Extensibility — *Layered CUE configuration model for operator boot-manifest overlays, host-user injection, and per-user toolchain caches.*

Process Model, Threading, and Scheduling

- Completion Rings And Threaded Runtimes — *Io_uring-style transports under threaded runtimes.*
- In-Process Threading — *In-process thread lifecycle, scheduler references, ThreadControl, and ParkSpace integration.*
- NO_HZ, SQPOLL, and Realtime Scheduling — *Linux NO_HZ, io_uring SQPOLL, CPU isolation, PREEMPT_RT, SCHED_DEADLINE, and seL4 MCS grounding for capOS timer and realtime design.*
- Out-of-Kernel Scheduling — *Userspace scheduling prior art.*
- Park Authority — *ParkSpace wait/wake authority, ABI, and shared park-word constraints.*
- Process Model — *Process isolation, ELF loading, bootstrap ABI, lifecycle, and spawn authority.*
- Rejected: Sleep(INF) Process Termination — *Rationale for explicit process termination instead of infinite-sleep lifecycle semantics.*
- Ring v2 For Full SMP — *Per-thread ring, completion routing, SQPOLL ownership, and full-SMP transport model.*
- Scheduling — *Preemption, run queues, blocking waits, timer wakeups, and SMP scheduler proof points.*
- SMP — *Per-CPU state, AP startup, scheduler ownership, TLB shootdown, and multi-core roadmap.*
- SMP Phase C — *SMP backlog.*
- Tickless and Realtime Scheduling — *Tickless idle, SQPOLL nohz CPU isolation, request deadlines, scheduling contexts, and realtime islands.*
- x2APIC And APIC Virtualization — *Interrupt routing on modern x86.*

Memory and Resource Accounting

- DMA Isolation — *DMA isolation model for device memory, IOMMU policy, and capability-scoped hardware access.*
- Go VirtualMemory Contract — *VirtualMemory cap contract for Go.*
- Memory Management — *Physical frames, address spaces, user buffers, MemoryObject, and VirtualMemory contracts.*
- OOM Handling and Swap — *Memory-pressure, OOM, anonymous-memory budgeting, and optional encrypted swap policy.*
- Resource Accounting and Quotas — *Resource profiles, quota ledgers, donation, reservation, and fail-closed accounting semantics.*

Userspace Runtime, Languages, and Binaries

- Browser/WASM — *Browser-hosted capOS experiment using WebAssembly and worker-per-process isolation.*
- Go Runtime — *Go runtime plan for GOOS=capos, memory growth, TLS, scheduling, and networking.*
- libcapos-service — *Userspace service framework for lifecycle, endpoint loops, readiness, shutdown, metrics, context, and resource hooks.*
- LLVM Target — *Requirements for a custom LLVM target triple.*
- Lua Scripting — *Capability-scoped Lua runner with curated libraries and explicit grants.*
- Runtime, Networking, and Shell — *Runtime/network/shell backlog.*
- Userspace Binaries — *Native userspace binary model, capos-rt authority handling, and language/POSIX support.*
- Userspace Runtime — *capos-rt entry ABI, heap, CapSet lookup, ring client, and typed userspace capability clients.*

Shells and Interactive Surfaces

- Boot to Shell — *Login, setup, session, credential, and broker path from boot into the native shell.*
- capOS-Hosted Agent Swarms — *capOS-hosted OpenClaw-like personal agents, agent swarms, harness controls, memory, retrieval, and research agenda.*
- Interactive Command Surfaces — *Structured command-session model for native interactive applications over typed invocations.*
- Language Models and Agent Runtime — *Language-model, embedder, agent-runner, and browser-agent capability interfaces.*
- Realtime Voice Agent Shell — *Realtime audio agent shell model across browser media, provider sessions, and brokered tools.*
- Shell — *Native, agent-oriented, and POSIX shell models over explicit capability grants.*
- SSH Shell Gateway — *SSH terminal gateway design preserving TerminalSession and broker-issued shell boundaries.*
- System Info Capability — *SystemInfo capability for MOTD, host metadata, help topics, and shell bundle integration.*
- Telnet over TLS Shell — *TLS-protected Telnet TerminalSession gateway with client certificates and credential fallback.*

Networking

- libcapos-service — *Userspace service framework for lifecycle, endpoint loops, readiness, shutdown, metrics, context, and resource hooks.*
- Networking — *Network capability architecture from virtio-net smoke to TCP sockets and terminal handoff.*
- Pingora — *Proxy/server framework as a userspace runtime case study.*
- SSH Shell Gateway — *SSH terminal gateway design preserving TerminalSession and broker-issued shell boundaries.*
- Telnet over TLS Shell — *TLS-protected Telnet TerminalSession gateway with client certificates and credential fallback.*

Storage, Persistence, and Naming

- Hardware, Boot, and Storage — *Hardware bring-up backlog.*
- IX-on-capOS Hosting — *IX as a package corpus, content-addressed build/store model, and a capability-native build-service surface for capOS.*
- Storage and Naming — *Capability-native storage, namespaces, boot packages, volumes, and persistence model.*
- Volume Encryption — *Encryption-at-rest model for system and user volumes with recovery and KMS options.*

Identity, Policy, and User Accounts

- Configuration — *How operators extend the default capOS boot manifest with a gitignored `system.local.cue` overlay and convert CUE-authored data to specified Cap'n Proto schemas.*
- Delegated Subject Context — *Future delegated-subject and act-on-behalf-of capability model.*
- Formal MAC/MIC — *Formal mandatory access and integrity model for future policy and proof work.*
- Local Users, Storage, and Policy — *Identity/local-user backlog.*
- OIDC and OAuth2 — *Federated login, OAuth2 clients, token capabilities, JWKS, DPoP, and broker integration.*
- Rejected: Endpoint Badges as Service Identity — *Post-mortem of the rejected seL4-style endpoint badge service identity model.*
- Service Object Identity Migration — *Superseded large-chunk migration plan for service object identity, retained as historical context after the active direction changed to session-bound invocation context.*
- Session-Bound Invocation Context — *Implementation plan for one-session-per-process invocation context and session-keyed shared services.*
- Session-Bound Invocation Context — *Session-bound invocation context and privacy-aware disclosure model replacing service-object identity migration.*
- System Configuration and Operator Extensibility — *Layered CUE configuration model for operator boot-manifest overlays, host-user injection, and per-user toolchain caches.*
- User Identity and Policy — *User, session, profile, RBAC/ABAC/MAC, and policy-layer model for capability grants.*

Cryptography, Certificates, and Trust

- Certificates and TLS — *Capability-native X.509, trust store, ACME, pinning, and TLS configuration model.*
- Cryptography and Key Management — *Capability model for keys, signing, encryption, vaults, entropy, and cryptographic policy.*
- OIDC and OAuth2 — *Federated login, OAuth2 clients, token capabilities, JWKS, DPoP, and broker integration.*
- Telnet over TLS Shell — *TLS-protected Telnet TerminalSession gateway with client certificates and credential fallback.*
- Volume Encryption — *Encryption-at-rest model for system and user volumes with recovery and KMS options.*

Security and Verification

- DMA Isolation — *DMA isolation model for device memory, IOMMU policy, and capability-scoped hardware access.*
- Formal MAC/MIC — *Formal mandatory access and integrity model for future policy and proof work.*
- Panic Surface Inventory — *Panic/unwrap/expect inventory.*
- Public Release and Maintainer Boundaries — *Public release posture, maintainer boundaries, issue intake, and repository hygiene gates.*
- Repository Composition — *Repository scope, sibling project split criteria, and cross-repository organization plan.*
- Security and Verification — *Security/verification backlog.*
- Security and Verification — *Security review vocabulary, trust-boundary checklist, and verification tracks for capOS.*
- Trust Boundaries — *The reviewer's authority-boundary inventory.*
- Trusted Build Inputs — *Trusted toolchain inventory.*
- Verification Workflow — *The verification gates used by capOS.*

Services, Operations, and Monitoring

- Cloud Deployment — *Cloud VM deployment plan covering hardware abstraction, storage, networking, and aarch64.*
- Cloud Metadata — *Cloud metadata and config-drive bootstrap through scoped configuration capabilities.*
- Configuration — *How operators extend the default capOS boot manifest with a gitignored `system.local.cue` overlay and convert CUE-authored data to specified Cap'n Proto schemas.*
- libcapos-service — *Userspace service framework for lifecycle, endpoint loops, readiness, shutdown, metrics, context, and resource hooks.*
- Live Upgrade — *Service replacement, capability retargeting, quiesce/resume, and in-flight call handling.*
- Rejected: Endpoint Badges as Service Identity — *Post-mortem of the rejected seL4-style endpoint badge service identity model.*

- Service Architecture — *Capability-based service composition, authority-at-spawn, exports, and service graph policy.*
- Session-Bound Invocation Context — *Session-bound invocation context and privacy-aware disclosure model replacing service-object identity migration.*
- Superseded: Service Object Capabilities — *Superseded service-minted object capability model that was replaced by session-bound invocation context.*
- System Configuration and Operator Extensibility — *Layered CUE configuration model for operator boot-manifest overlays, host-user injection, and per-user toolchain caches.*
- System Monitoring — *Capability-scoped logs, metrics, health checks, traces, crash records, and status views.*
- System Performance Benchmarks — *Correctness-gated benchmark model for primitives, workloads, and user stories.*

AI, Agents, GPU, and Robotics

- capOS As A Robot Brain — *Robotics service graph, actuator gateway, safety monitor, realtime island, and ROS bridge model.*
- capOS Repository Harness Engineering — *Repository-local harness engineering for making capOS legible, checkable, and safer for long-running coding agents.*
- capOS-Hosted Agent Swarms — *capOS-hosted OpenClaw-like personal agents, agent swarms, harness controls, memory, retrieval, and research agenda.*
- GPU Capability — *Capability-oriented GPU access, driver isolation, memory sharing, and CUDA-style compute model.*
- Hosted Agent Harnesses — *OpenClaw-like harnesses, swarms, memory/wiki systems, and agent orchestration research for capOS-hosted agents.*
- Language Models and Agent Runtime — *Language-model, embedder, agent-runner, and browser-agent capability interfaces.*
- Multimedia Pipeline Latency — *Research note.*
- NO_HZ, SQPOLL, and Realtime Scheduling — *Linux NO_HZ, io_uring SQPOLL, CPU isolation, PREEMPT_RT, SCHED_DEADLINE, and seL4 MCS grounding for capOS timer and realtime design.*
- Realtime Multimodal Agent APIs — *Research note.*
- Realtime Voice Agent Shell — *Realtime audio agent shell model across browser media, provider sessions, and brokered tools.*
- Robotics Realtime Control — *Research note.*
- Small LLM Survey — *Model candidates for the on-ISO local LLM.*
- Tickless and Realtime Scheduling — *Tickless idle, SQPOLL nohz CPU isolation, request deadlines, scheduling contexts, and realtime islands.*

Demos, Onboarding, and Contributor Surfaces

- Aurelian Frontier — *Aurelian Frontier game-depth backlog.*
- Aurelian Frontier — *Capability-native Aurelian Frontier game design, mission model, content pipeline, and QEMU proof slice.*
- Aurelian Frontier (proof slice) — *Multi-process Aurelian Frontier smoke proof.*

- Contributor Quest Mechanics — *Contributor reward mechanics layered on Aurelian Frontier without granting repository authority.*
- First Chat Demo — *Smallest resident-service proof.*
- Game Mechanics Prior Art — *Grounded mechanics research for Aurelian Frontier seasonal play, markets, construction, and tactical combat.*
- Paperclips Terminal Demo — *Clean-room incremental terminal demo.*
- Paperclips Terminal Demo — *Paperclips terminal demo backlog and content migration notes.*
- Shared-Service Demos — *Demo backlog.*

Build, Tooling, and Documentation Site

- Build, Boot, and Test — *Build, ISO, QEMU, host-test commands.*
- capOS Repository Harness Engineering — *Repository-local harness engineering for making capOS legible, checkable, and safer for long-running coding agents.*
- Documentation Bundle — *Single-page generated documentation bundle.*
- mdBook Documentation Site — *Documentation-site structure, metadata, status vocabulary, and curation workflow.*
- Repository Composition — *Repository scope, sibling project split criteria, and cross-repository organization plan.*
- Repository Map — *Source-tree subsystem index.*
- Trusted Build Inputs — *Trusted toolchain inventory.*

Research and Papers

- Papers — *Long-form research write-ups.*
- Whitepaper Evidence Gaps — *Tracks unresolved whitepaper evidence needs and the milestones that close them.*
- Whitepaper Outline — *Section outline and evidence dependency map for the schema-as-ABI capOS whitepaper.*
- Whitepaper Plan — *Planning baseline for the future schema-as-ABI capOS whitepaper.*

Prior Art and Comparative OS Research

- EROS, CapROS, Coyotos — *Persistent capability-system lineage.*
- Game Mechanics Prior Art — *Grounded mechanics research for Aurelian Frontier seasonal play, markets, construction, and tactical combat.*
- Genode — *Componentized OS framework.*
- Plan 9 and Inferno — *Namespace-oriented systems.*
- Research Index — *Design consequences pulled from the survey.*
- seL4 — *Microkernel and capability reference.*
- Zircon — *Handle-based OS reference.*

Stage Backlogs and Long-Form Planning

- Aurelian Frontier — *Aurelian Frontier game-depth backlog.*
- Go VirtualMemory Contract — *VirtualMemory cap contract for Go.*
- Hardware, Boot, and Storage — *Hardware bring-up backlog.*
- Local Users, Storage, and Policy — *Identity/local-user backlog.*

- Paperclips Terminal Demo — *Paperclips terminal demo backlog and content migration notes.*
- Proposal Group Archive — *Archived proposal cluster.*
- Run Targets, Init Mandate, and Default-Run Integration — *Run-target governance.*
- Runtime, Networking, and Shell — *Runtime/network/shell backlog.*
- Security and Verification — *Security/verification backlog.*
- Service Object Identity Migration — *Superseded large-chunk migration plan for service object identity, retained as historical context after the active direction changed to session-bound invocation context.*
- Session-Bound Invocation Context — *Implementation plan for one-session-per-process invocation context and session-keyed shared services.*
- Shared-Service Demos — *Demo backlog.*
- SMP Phase C — *SMP backlog.*
- Stage 6 Capability Semantics — *Stage 6 capability work.*
- Whitepaper Evidence Gaps — *Tracks unresolved whitepaper evidence needs and the milestones that close them.*
- Whitepaper Outline — *Section outline and evidence dependency map for the schema-as-ABI capOS whitepaper.*
- Whitepaper Plan — *Planning baseline for the future schema-as-ABI capOS whitepaper.*