Section 7: OS Design (Minimal but Real)

Contents

1 Version Comparison

Ver.	Primary Goal	Boot/ABI Contract	Control Transfers	Scheduling Model	Privilege / Protection
OSv1	Kernel runs deterministically	Boot v1: stack init + jump to `kernel_main`	None (no vectors, no syscalls)	None	**No**
OSv2	Deterministic diagnostics (fail loudly)	Boot v2: stages + panic signature, then `kernel_main`	None (no vectors, no syscalls)	None	**No**
OSv3	OS-shaped control transfers	Boot v3: vectors @0x0000..0x00FF, reset @0x0100	`SYSCALL` (vec 0x00), Timer IRQ (vec 0x01), `IRET` return	None (mechanisms only)	**No**(no user mode)
OSv4	Structured kernel initialization + tasks	Boot v4 ABI: `R0=bootinfo_ptr`, BootInfo validation	Syscalls/timer may remain; scheduling remains cooperative	Cooperative round-robin tasks	**No**(single privilege level)
OSv5	Real boundary: user + protection + dual image	Boot v5: kernel owns reset/vectors; kernel→user transition	Syscalls as boundary; protection faults; optional timer preemption	Coop & optional preemptive (timer)	**Yes**(K/I flags, faults)

2 OSv1: Kernel Runs (Minimal but Real)

2.1 Purpose

OSv1 is the most minimal OS version that still qualifies as an "OS" for this project: it boots deterministically, establishes a known execution environment (stack + entry), executes `kernel_main`, and terminates deterministically (HALT or a deliberate infinite loop).

OSv1 is intentionally *kernel-only* and *physical-memory-only*. It is designed to be the earliest stable foundation for later versions (OSv2–OSv5), without pre-committing to user mode, syscalls, interrupts, devices, or scheduling.

2.2 Scope and Non-Goals

2.2.1 In scope

- Boot into the kernel using the **Boot v1** contract (flat binary boot).
- Set a valid stack and transfer control to `kernel_main`.
- Deterministic termination behavior (HALT or defined loop).
- Establish base invariants that all later OS versions must preserve.

2.2.2 Out of scope (explicit non-goals)

- No vectors table, no interrupts, no timer behavior, no `IRET`.
- No syscalls (`SYSCALL`) and no user code.
- No dynamic allocation, no heap, no paging/MMU, no protection regions.
- No processes; no multitasking; no scheduling.
- No device drivers and no MMIO usage.

2.3 Target Platform Compatibility

OSv1 must be buildable and runnable under the earliest platform milestone:

- **CPU:** v2 (baseline execution model without vectors/syscall/timer requirements).
- **Toolchain:** v2 (flat binary output, no dual-image).
- **Boot:** v1 (stack initialization + jump to kernel entry).

Forward-compatibility constraints. Even though OSv1 does not use vectors or syscalls, it must not prevent later upgrades to OSv3+ (vectors at low memory, reset entry at the defined reset region, etc.). Therefore, OSv1 must:

- Avoid hard-coding assumptions that conflict with later fixed regions (e.g., reserving all low memory permanently).
- Keep the kernel stack within the "safe" stack area used across the boot plans.

2.4 Binary and Memory Model (OSv1)

2.4.1 Binary form

OSv1 is delivered as a single **flat binary image**:

```
kernel.bin
```

No separate user image exists in OSv1.

2.4.2 Addressing model

- All addresses are physical.
- No relocation is assumed at runtime beyond what the boot stub enforces.

2.4.3 Minimal memory layout assumptions

OSv1 adopts the project's reserved I/O window and stack recommendations to ensure compatibility with the later boot/toolchain versions:

- **MMIO region reserved:** `0xFE00..0xFFFF` is treated as reserved for devices. OSv1 must not read/write this range.
- **Kernel stack:** stack grows downward and must remain below `0xFE00`. Recommended initial stack top is `0xFDFF`.

Rationale. Even though OSv1 does not implement devices, adopting the reserved MMIO window and stack placement early prevents later incompatibilities when OSv3+ introduces interrupts/vectors and OSv5 introduces protection and user space.

2.5 Boot Contract (Boot v1)

OSv1 relies on the **Boot v1** behavior:

- The boot stub establishes the initial stack.
- Control transfers to the kernel entry point.

2.5.1 Required initial register state

At the instant `kernel_main` begins execution, OSv1 requires:

- `SP = 0xFDFF` (recommended) or another value in the safe stack region strictly below `0xFE00`.
- `FP` initialized to the same value as `SP` (or a value consistent with the calling convention used by the toolchain).
- All other registers are *unspecified* in OSv1 and must not be assumed.

2.5.2 Kernel entry symbol

- The kernel must expose an entry symbol named `kernel_main`.
- In OSv1, `kernel_main` takes no arguments and returns no value.

Signature (conceptual).

```
kernel_main(): void
```

2.6 Kernel Behavior Specification

2.6.1 Functional requirements

OSv1 kernel behavior is minimal and deterministic:

1. **Start:** execution begins at `kernel_main` with a valid stack.
2. **Do:** the kernel may perform internal computations that do not require I/O.
3. **End:** the kernel must terminate deterministically by one of:
   - Executing the architectural `HALT` instruction, or
   - Entering a deliberate, documented infinite loop (e.g., `while(true){}`).

2.6.2 Safety requirements

- The kernel must not access the reserved MMIO window `0xFE00..0xFFFF`.
- The kernel must not corrupt its own stack or execute with an invalid `SP`.
- The kernel must not rely on interrupts, vectors, syscalls, or protection features.

2.6.3 Determinism requirement

Given the same initial memory image and start state, OSv1 must behave identically on every run. Since there is no timer, no interrupts, and no external I/O in OSv1, determinism should be trivially achievable.

2.7 Interfaces Reserved for Later Versions

OSv1 deliberately reserves names and conceptual interfaces that will be introduced later, without implementing them now. This prevents rework while keeping OSv1 minimal.

2.7.1 Reserved exception/interrupt concepts (OSv3+)

OSv1 does not define or install a vectors table. The following are reserved for later:

- Syscall vector (`0x00`) and timer vector (`0x01`) for OSv3+.
- Protection fault handling for OSv5.

2.7.2 Reserved system call surface (OSv3+)

OSv1 provides *no* syscalls. However, names and intent are reserved:

- `sys_putc`, `sys_halt/exit`, `sys_yield` (introduced later).

2.8 Build and Linking Constraints

2.8.1 Toolchain constraints (v2)

- Output is a single flat binary `kernel.bin`.
- No dual-image or user build is involved.

2.8.2 No ABI dependencies (OSv1)

OSv1 intentionally avoids depending on later ABI contracts (BootInfo pointer in `R0`, etc.). Those are introduced at OSv4+.

2.9 Plan of Execution (OSv1)

This plan is written in the same "execution checklist" style used across earlier project sections.

2.9.1 Milestone OSv1.M1 — Boot-to-kernel transfer

- Implement the Boot v1 stub:
  1. Set `SP := 0xFDFF`.
  2. Set `FP := SP`.
  3. Jump to `kernel_main`.
- Verify that execution reaches `kernel_main` deterministically.

2.9.2 Milestone OSv1.M2 — Deterministic termination

- In `kernel_main`, implement one deterministic end-state:
  - Preferred: execute `HALT`.
  - Acceptable: infinite loop with no side effects.

2.9.3 Milestone OSv1.M3 — Invariant checks (non-fatal in v1)

- Optionally include lightweight internal assertions (compile-time or runtime) that:
  - `SP < 0xFE00`
  - no writes occur to `0xFE00..0xFFFF`
- In OSv1 these checks do *not* require a standardized panic signature (that becomes mandatory in OSv2).

2.10 Definition of Done (OSv1)

OSv1 is complete when all of the following are true:

1. The image boots under the Boot v1 flow and reaches `kernel_main`.
2. The stack is initialized in the safe region (recommended `SP=0xFDFF`).
3. The kernel terminates deterministically (HALT or defined loop).
4. No MMIO region accesses occur (`0xFE00..0xFFFF` untouched).

2.11 Upgrade Notes

Path to OSv2. OSv2 adds standardized failure reporting (panic signature) and mandatory sanity checks, while remaining kernel-only and still using a single binary. No OSv1 behavior is removed; OSv2 only adds deterministic diagnostics and stricter boot-time validation.

3 OSv2: Kernel Fails Loudly (Deterministic Diagnostics)

3.1 Purpose

OSv2 keeps the OSv1 model (single kernel image, flat memory, no interrupts/syscalls), but adds a **frozen and deterministic early-boot diagnostics contract**.

If early boot fails, it must fail *loudly* and *repeatably* using a standardized **panic signature** (register dump) and standardized **panic codes**. This makes debugging and later-stage integration reliable without introducing complexity prematurely.

3.2 Scope and Non-Goals

3.2.1 In scope

- Everything in OSv1: a single kernel image that boots, sets a valid stack, and transfers control to `kernel_main`.
- Frozen early-boot **stage IDs** for progress reporting.
- Mandatory **sanity checks** before transferring control to `kernel_main`.
- Frozen **panic contract** (register signature + panic codes) and deterministic HALT on failure.

3.2.2 Out of scope (explicit non-goals)

- No vectors table, no interrupts, no timer, no `IRET` usage.
- No syscalls (`SYSCALL`) and no user code.
- No devices and no MMIO usage.
- No allocator/heap, no scheduling, no tasks.
- No privilege separation or protection mechanisms (introduced later).

3.3 Target Platform Compatibility

OSv2 must be runnable under the same early platform milestone as OSv1:

- **CPU baseline:** v2 (baseline stack/branch model).
- **Toolchain baseline:** v2 (single flat `.bin` loaded at `0x0000`).
- **Boot baseline:** v2 early-boot contract (adds deterministic diagnostics to v1).

3.4 Memory Model and Invariants

OSv2 adopts the project-wide reserved region and stack conventions to preserve forward compatibility (especially with later boot/toolchain versions).

3.4.1 Reserved MMIO window

- **Reserved:** `0xFE00..0xFFFF` is reserved for devices/future MMIO.
- OSv2 must not read or write this region.

3.4.2 Kernel stack placement

- Stack grows downward.
- Recommended stack range: `0xF800..0xFDFF`.
- Recommended initial stack top: `STACK_TOP = 0xFDFF`.

3.5 Boot v2 Diagnostics Contract (Normative)

OSv2 relies on a **Boot v2** style stub whose responsibility is:

1. Establish an initial stack.
2. Track progress using frozen **stage IDs**.
3. Perform mandatory sanity checks.
4. On failure: emit a frozen **panic signature** and HALT deterministically.
5. On success: jump to `kernel_main`.

3.5.1 Boot stage identifiers (frozen IDs)

The stub must track and report progress using the following frozen IDs:

Stage ID	Meaning
`0x01`	Entered `_start`
`0x02`	Stack initialized
`0x03`	Sanity checks running
`0x04`	Transfer to `kernel_main` imminent

3.5.2 Panic contract (register signature, normative)

On unrecoverable early-boot failure, the stub must set the following register signature and then HALT:

Register	Meaning
`R0`	Panic code
`R1`	Stage ID (from frozen IDs)
`R2`	Location identifier (PC snapshot if available, else label constant)
`R3`	SP snapshot
`R4`	FP snapshot

Normative rule. Once the panic signature is emitted, the stub must not attempt recovery. It must transition into a deterministic end-state (HALT is preferred).

3.5.3 Panic codes (normative)

The following panic codes are frozen for OSv2 early-boot failures:

Code	Meaning
`0x10`	SP out of allowed range
`0x11`	SP overlaps reserved future region (`0xFE00..0xFFFF`)
`0x12`	SP overlaps kernel code region (unsafe)
`0x13`	Alignment violation (PC/target not 8-byte aligned)
`0x14`	Kernel entry invalid / out of bounds

3.5.4 Sanity checks (mandatory)

Before jumping to `kernel_main`, the OSv2 stub must verify:

1. **Stack bounds:** `SP` is within `[STACK_BOTTOM..STACK_TOP]` (recommended `0xF800..0xFDFF`).
2. **MMIO exclusion:** `SP < 0xFE00`.
3. **Kernel entry validity:** `kernel_main` is valid and **8-byte aligned**.

If any check fails, the stub must invoke the panic contract with the appropriate panic code.

3.5.5 Control flow (normative)

The following control flow is normative for OSv2 boot:

```
_start:
stage := 0x01
SP := STACK_TOP
FP := STACK_TOP
stage := 0x02
stage := 0x03
run sanity checks
if fail -> PANIC(code, stage, location)
stage := 0x04
JMP_ABS kernel_main
```

3.6 Kernel Entry Contract (OSv2)

3.6.1 Entry symbol

- The kernel must expose a symbol named `kernel_main`.

3.6.2 Signature (OSv2)

OSv2 keeps OSv1's minimal entry signature to avoid depending on later ABI contracts:

```
kernel_main():  void
```

Notes.

- No BootInfo pointer is passed in OSv2 (that begins at OSv4).
- No syscall ABI is required (that begins at OSv3+).

3.7 Kernel Behavior Specification

3.7.1 Functional requirements

1. Execution begins at `kernel_main` after successful sanity checks.
2. The kernel must not depend on interrupts, syscalls, vectors, or MMIO.
3. The kernel must terminate deterministically (HALT preferred, otherwise a defined loop).

3.7.2 Safety requirements

- The kernel must not access `0xFE00..0xFFFF`.
- The kernel must not assume any register values beyond having a valid stack.
- The kernel must not corrupt the stack region reserved for kernel execution.

3.7.3 Determinism requirement

Given identical memory image and reset state, OSv2 must produce identical behavior. This includes identical panic signatures for identical failure conditions.

3.8 Interfaces Reserved for Later Versions

OSv2 remains kernel-only, but reserves the same conceptual interfaces planned in later OS versions:

- Vectors table and reset-at-`0x0100` image model (OSv3+).
- Syscall entry/return mechanism (OSv3+).
- BootInfo + ABI-frozen kernel entry pointer passing (OSv4+).
- Dual-image and user mode/protection boundary (OSv5).

3.9 Plan of Execution (OSv2)

3.9.1 Milestone OSv2.M1 — Stage ID instrumentation

- Add the frozen stage IDs to the boot stub: `0x01`, `0x02`, `0x03`, `0x04`.
- Ensure stage transitions occur exactly as per the normative control flow.

3.9.2 Milestone OSv2.M2 — Mandatory sanity checks

- Implement the three mandatory checks: stack bounds, MMIO exclusion, kernel entry validity + 8-byte alignment.
- Map each failure to a frozen panic code (`0x10..0x14`).

3.9.3 Milestone OSv2.M3 — Panic signature + deterministic HALT

- On failure, set `R0..R4` exactly per the panic contract and HALT.
- Ensure `R1` always contains the current stage ID.
- Provide a stable location identifier in `R2`:
  - Preferred: PC snapshot if available.
  - Otherwise: a constant label ID for the failure site.

3.9.4 Milestone OSv2.M4 — Positive + negative test cases

- Positive: normal boot reaches `kernel_main` exactly like OSv1.
- Negative: at least one deliberate misconfiguration triggers the correct panic signature.

3.10 Definition of Done (OSv2)

OSv2 is complete when:

1. Normal boot reaches `kernel_main` exactly like OSv1.
2. Mandatory sanity checks run before the jump to `kernel_main`.
3. Any sanity-check failure produces a correct panic signature in `R0..R4` and halts deterministically.
4. At least one deliberate misconfiguration test reliably triggers the expected panic code and stage ID.

3.11 Upgrade Notes

Path to OSv3. OSv3 transitions into an OS-shaped boot and execution model with:

- A vectors table at low memory,
- A fixed reset entry,
- Syscall and timer entry/return proof-of-life via `IRET`.

OSv2's diagnostics (stages + panic signature) remain valuable in OSv3+ for continuity.

4 OSv3: OS-shaped Control Transfers (Vectors + Syscalls + Timer)

4.1 Purpose

OSv3 is the first version where the system behaves like an operating system *mechanically*: it installs a vectors table, supports a syscall entry path, supports a timer interrupt entry path, and proves both paths return correctly using `IRET`.

OSv3 remains intentionally minimal: there is still no user/kernel privilege separation, no processes, and no memory protection. However, OSv3 introduces the control-transfer mechanisms required by later versions.

4.2 Scope and Non-Goals

4.2.1 In scope

- Boot using the **Boot v3** image model: vectors region at `0x0000..0x00FF` and reset entry at `0x0100`.
- Provide a vectors table with at least:
  - **Vector 0x00:** syscall entry (`SYSCALL`).
  - **Vector 0x01:** timer interrupt entry.
- Provide correct return from handlers using `IRET`.

- Provide a minimal syscall surface compatible with Compiler Phase 2 proof-of-life: `putc` and `halt/exit`.
- Specify the expected **timer interrupt** behavior for CPU v3/v4 and the resulting constraints on early kernel initialization.

4.2.2 Out of scope (explicit non-goals)

- No user mode, no privilege separation, no protection faults (OSv5).
- No BootInfo/Boot ABI pointer passing (OSv4).
- No task scheduler or multitasking (introduced later; OSv4+).
- No filesystem, no drivers beyond minimal console output.

4.3 Target Platform Compatibility

OSv3 targets the following project milestones:

- **CPU:** v3 (vectors, syscall entry, timer interrupt, `IRET`).
- **Toolchain:** v3 (image layout that includes a vectors table and a fixed reset entry).
- **Boot:** v3 (vectors @ `0x0000..0x00FF`, reset @ `0x0100`).
- **Compiler:** Phase 2 compatibility for syscall proof-of-life programs.

4.4 Memory and Image Layout (Boot v3 Model)

OSv3 uses the fixed boot image model (normative):

Address Range	Meaning
`0x0000..0x00FF`	Vectors table region
`0x0100..`	Reset entry and kernel code/data (`_start` at `0x0100`)
`0xFE00..0xFFFF`	Reserved MMIO window (devices)

4.4.1 Reserved MMIO window

Unlike OSv1/OSv2, OSv3 *may* use MMIO for minimal console output. However, all MMIO access must remain within the reserved MMIO window `0xFE00..0xFFFF`, and must be isolated to explicit driver-like routines.

4.4.2 Kernel stack placement

- Stack grows downward.
- The kernel stack must remain below `0xFE00`.
- Recommended initial stack top: `0xFDFF`.

4.5 Interrupt Model (CPU v3/v4) and Early-Init Constraints

CPU v3 introduces a deterministic timer interrupt (vector `0x01`) that may occur periodically according to the CPU's instruction counter and `TIMER_PERIOD` setting.

No architectural interrupt-mask bit in v3/v4. In the project CPU model, an explicit interrupt mask flag is introduced in CPU v5 (`I` in `FLAGS`). CPU v3/v4 do not provide a normative, architecturally-defined "enable/disable interrupts" bit.

Normative OSv3 rule. OSv3 must be correct assuming that a timer interrupt *can* be delivered at any time after reset, subject to the CPU's deterministic timer schedule. In particular:

- The vectors table must be present in the boot image (by construction).
- Reset code must establish a valid stack early (recommended `SP=FP=0xFDFF`) so that interrupt/trap entry has valid stack space.
- All interrupt/trap handlers used in OSv3 must return via `IRET`.

Note on v5. Once CPU v5 is introduced, the kernel may use `FLAGS.I` to mask timer interrupts around critical sections. OSv3 (v3/v4) does not assume such masking support.

4.6 Vectors Table (Normative Requirements)

4.6.1 Vectors region

- The vectors table occupies `0x0000..0x00FF` (256 bytes).
- The exact encoding of a vector entry is the ISA-defined encoding used by CPU v3.
- Each vector must resolve to the absolute handler address for that vector.

4.6.2 Required vectors

OSv3 must populate at least the following vectors:

Vector	Name	Handler
`0x00`	Syscall entry	`isr_syscall`
`0x01`	Timer IRQ	`isr_timer`

All other vectors may point to a common `isr_default` handler that halts.

4.7 Syscall ABI (OSv3 Minimal, Normative)

OSv3 defines a minimal syscall ABI compatible with compiler proof-of-life programs.

4.7.1 Calling convention

- The caller executes `SYSCALL`.
- Syscall number is passed in `R0`.
- Arguments are passed in `R1, R2, R3` as needed.
- Return value (if any) is placed in `R0`.

4.7.2 Required syscall numbers

The following syscall numbers are reserved starting at OSv3 and remain stable across later OS versions. Later versions may *refine* the meaning in the presence of user tasks, but the intent of each number must remain compatible.

Syscall #	Name	Semantics
1	`putc`	Output low 8 bits of `R1` to console
2	`halt`	Terminate the current execution context. In OSv3 (single-kernel context) this halts th

Compatibility note (normative). Software that targets the proof-of-life syscall surface must treat syscall #2 as *termination*. An implementation that additionally supports multiple tasks may interpret termination as *exit current task*; if no other task is runnable, the system must halt deterministically.

Console output semantics (normative). `putc` must be implemented using the project's MMIO UART/console transmit mechanism in the `0xFE00..0xFFFF` window (device address and protocol are defined elsewhere in the project). If the device is unavailable in a given emulator configuration, `putc` may become a no-op, but must still preserve correct syscall return behavior.

4.7.3 Register preservation rule

- The syscall handler may clobber caller-saved registers per the project calling convention.
- At minimum, `IRET` must return execution to the correct instruction stream with a valid stack.

4.8 Timer Interrupt (OSv3 Minimal, Normative)

OSv3 must include a timer interrupt handler at vector `0x01`. The timer handler exists primarily to prove correct interrupt entry and `IRET` return.

4.8.1 Minimal required behavior

- `isr_timer` must be callable by the hardware timer interrupt.
- `isr_timer` must return via `IRET`.
- The handler may optionally increment a `tick` counter in kernel memory.

4.8.2 Timer interrupts during early initialization

Because OSv3 does not assume an architectural interrupt mask bit, the safe approach is: establish the stack immediately, ensure vectors/handlers are valid, and keep early initialization bounded and handler-safe.

4.9 Reset Entry and Kernel Initialization

4.9.1 Reset entry point

- The reset entry point is `_start` located at `0x0100` (normative).

4.9.2 Initialization steps (normative order)

1. **Establish stack:** set `SP` and `FP` (recommended `0xFDFF`).
2. **Vectors installed:** vectors table must already be present in the image. (If the toolchain emits it, this is satisfied by construction.)
3. **Optional:** install a default handler for unused vectors.
4. **Jump to kernel main:** transfer control to `kernel_main`.

4.10 Kernel Entry Contract (OSv3)

OSv3 keeps the OSv1/OSv2 minimal kernel entry signature to avoid BootInfo coupling:

```
kernel_main(): void
```

4.11 Kernel Behavior Specification

4.11.1 Functional requirements

1. Provide the vectors table with required vectors (0x00, 0x01).
2. Implement `isr_syscall` supporting syscall #1 and #2.
3. Implement `isr_timer` that returns via `IRET`.
4. Provide deterministic termination via syscall #2 or direct HALT.

4.11.2 Safety requirements

- No writes outside kernel-owned memory except explicit MMIO output.
- The reset path must establish vectors + a valid stack before the first possible timer interrupt delivery (per the CPU's deterministic timer schedule).
- All handlers must be `IRET`-safe (no stack corruption, correct return).

4.12 Plan of Execution (OSv3)

4.12.1 Milestone OSv3.M1 — Image layout + vectors

- Emit a vectors table occupying `0x0000..0x00FF`.
- Ensure `_start` is located at `0x0100`.
- Populate vector `0x00` and `0x01`; route all others to `isr_default`.

4.12.2 Milestone OSv3.M2 — Syscall proof-of-life

- Implement `isr_syscall`:
  - `R0=1`: `putc` using MMIO console TX.
  - `R0=2`: `halt` (HALT deterministically).
- Ensure syscall returns to the caller correctly via `IRET`.

4.12.3 Milestone OSv3.M3 — Timer IRQ proof-of-life

- Implement `isr_timer`:
  - Optional: increment `tick`.
  - Mandatory: return via `IRET`.
- Demonstrate that timer IRQ delivery can occur and that execution resumes correctly after `IRET` (no dependence on an interrupt-enable bit in v3/v4).

4.12.4 Milestone OSv3.M4 — Demonstration program

Provide a minimal program (kernel-resident in OSv3) that:

- Calls syscall #1 to output at least one character.
- Calls syscall #2 to halt.

This aligns with Compiler Phase 2 proof-of-life goals (syscall builtin usage).

4.13 Definition of Done (OSv3)

OSv3 is complete when:

1. The image includes a valid vectors table at `0x0000..0x00FF` and reset entry at `0x0100`.
2. Syscall vector `0x00` correctly dispatches syscall #1 and #2.
3. Timer vector `0x01` reaches `isr_timer` and returns via `IRET`.
4. Syscall and timer paths both return control safely (no stack corruption; correct resumption).

4.14 Upgrade Notes

Path to OSv4. OSv4 formalizes kernel initialization by consuming **BootInfo** via the Boot ABI (`R0 = bootinfo_ptr`) and introduces cooperative tasks/scheduling. OSv3 already provides the necessary control-transfer primitives (syscalls and timer IRQ return discipline), so OSv4 can focus on structure rather than mechanism.

5 OSv4: Structured Kernel (BootInfo + Cooperative Tasks)

5.1 Purpose

OSv4 transforms the kernel from "proof-of-life mechanisms" into a *structured system*: it consumes a formal boot descriptor (**BootInfo**), follows a frozen **Boot ABI**, and implements a minimal but real task model with a cooperative scheduler.

OSv4 is still *single-privilege-level* (no true user mode separation yet), and still *physical-memory-only.* The goal is to build durable structure and contracts without jumping early into protection and dual-image complexity.

5.2 Scope and Non-Goals

5.2.1 In scope

- Boot using the **Boot v4** contract: **BootInfo** pointer passed in `R0`.
- Validate and consume BootInfo fields (magic/version/size + key addresses).
- Preserve OSv3 control-transfer mechanisms: vectors, syscall entry, and timer handler *may* remain present.
- Implement **cooperative tasks**: task creation, explicit yield, and round-robin scheduling.
- Introduce a minimal kernel allocator suitable for early kernel work (bump allocator).

5.2.2 Out of scope (explicit non-goals)

- No user mode, no privilege separation, no protection traps (OSv5).
- No preemptive scheduling requirement (timer may exist, but scheduling is cooperative).
- No processes; tasks share a single address space.
- No virtual memory, no filesystem.

5.3 Target Platform Compatibility

OSv4 targets:

- **CPU:** v4 (note: *do not assume* an explicit interrupt enable/disable bit in v4).
- **Toolchain:** v4 (alignment discipline and boot ABI assumptions).
- **Boot:** v4 (BootInfo + Boot ABI).
- **Compiler:** Phase 3 integration (kernel entry signature and ABI freeze).

5.4 Boot v4 Contract (Normative)

5.4.1 BootInfo pointer passing (Boot ABI)

On kernel entry, the boot environment passes the BootInfo pointer as:

```
R0 = bootinfo_ptr
```

This is normative in OSv4.

5.4.2 BootInfo location and non-overlap

- The recommended fixed location for BootInfo is `0x0200`.
- BootInfo must not overlap the vectors region or the kernel image.
- The kernel must treat BootInfo memory as **read-only**.

5.4.3 Alignment discipline (toolchain v4)

OSv4 code must respect the toolchain v4 alignment discipline:

- 16-bit/32-bit operations must be aligned appropriately.
- Structures must be laid out with alignment in mind to avoid undefined behavior.

5.5 Kernel Entry Contract (OSv4)

OSv4 freezes the kernel entry signature to match the boot ABI and compiler Phase 3 integration.

Signature (conceptual).

```
kernel_main(bootinfo: *const BootInfo): void
```

Register contract (normative).

- `R0` contains the BootInfo pointer.
- Other registers are unspecified and must not be assumed.
- A valid stack exists (recommended `SP=0xFDFF`).

5.6 BootInfo Structure (OSv4 Kernel View)

The exact BootInfo binary layout is defined by the Boot v4 section. OSv4 requires at minimum:

- **Magic** value to validate BootInfo presence.
- **Version** and **Size** fields for compatibility.
- Memory layout metadata required for safe kernel initialization, including:
  - Kernel image base/end (or size).
  - Total RAM size or highest usable address.
  - Optional: device availability hints (console present).

5.6.1 Validation (mandatory)

On entry to `kernel_main`, OSv4 must:

1. Validate BootInfo pointer alignment.
2. Validate BootInfo magic.
3. Validate BootInfo version is supported.
4. Validate BootInfo size is at least the minimum required by the OSv4 kernel.

If validation fails, OSv4 must transition to a deterministic failure end-state (see §5.7).

5.7 Failure Policy (OSv4)

OSv4 inherits the OSv2 idea of deterministic diagnostics. However, OSv4 may implement its own structured panic routine using the console syscall path if present.

5.7.1 Normative requirements

- Any fatal failure must end deterministically (HALT preferred).
- If console output exists, OSv4 should print a short error code/tag before halting.
- If console output is unavailable, OSv4 must still halt deterministically.

5.8 Memory Model and Kernel Regions

5.8.1 Reserved MMIO window

- `0xFE00..0xFFFF` remains reserved for MMIO.
- OSv4 may access it only via explicit driver routines (e.g., console).

5.8.2 Kernel stack

- Stack grows downward.
- Recommended kernel stack is within `0xF800..0xFDFF` with top at `0xFDFF`.

5.8.3 Kernel heap (OSv4 minimal allocator)

OSv4 introduces a minimal allocator to support kernel data structures (task control blocks, run queue):

- A **bump allocator** is sufficient.
- The heap region must be chosen so it does not overlap: vectors (`0x0000..0x00FF`), reset code (`0x0100..`), BootInfo (`0x0200..`), stack (`0xF800..0xFDFF`), or MMIO (`0xFE00..`).
- Heap start may be derived from BootInfo (preferred) or from the kernel image end symbol.

5.9 Control Transfers (Syscall + Timer) in OSv4

5.9.1 Compatibility with OSv3 mechanisms

OSv4 may retain the OSv3 vectors + syscall mechanism. However, OSv4 must not *depend* on an explicit interrupt enable/disable bit in CPU v4. Therefore:

- If the timer handler exists, it must be safe under the default interrupt behavior.
- Scheduling remains cooperative; the timer does not preempt tasks in OSv4.

5.9.2 Yield mechanism

OSv4 introduces a cooperative `yield` boundary that switches tasks. Yield may be implemented as either:

- A direct kernel call: `task_yield()`, or
- A syscall number reserved for yield (if syscalls remain present).

The semantic requirement is identical:

> **Calling yield transfers control to the scheduler, which selects the next runnable task.**

5.10 Tasks (OSv4 Minimal but Real)

5.10.1 Task model

OSv4 tasks are lightweight execution contexts within the kernel address space:

- No separate address spaces.
- Each task has its own stack.
- Each task has a saved register context sufficient to resume execution.

5.10.2 Task control block (TCB)

OSv4 defines a minimal TCB containing:

- Saved `SP` and `FP`.
- Saved general registers required by the calling convention.
- Saved program counter / return address (representation is ISA-defined).
- Task state: `RUNNABLE`, `RUNNING`, `BLOCKED` (optional), `DEAD`.

5.10.3 Task stacks

- Each task stack is a fixed-size region allocated from physical memory.
- Stack regions must not overlap the kernel stack, BootInfo, heap, or MMIO.

5.10.4 Scheduler (cooperative round-robin)

OSv4 implements a minimal cooperative scheduler:

- Round-robin over runnable tasks.
- The current task runs until it calls `yield` (or exits).
- No priorities required.

5.10.5 Task creation

OSv4 must provide a way to create tasks:

- `task_create(entry_fn, stack_top)` creates a runnable task.
- The task begins at `entry_fn` with a clean initial register state.

5.10.6 Task exit

A task may terminate by returning from its entry function or calling a kernel exit function. The scheduler must not select dead tasks.

5.11 Plan of Execution (OSv4)

5.11.1 Milestone OSv4.M1 — BootInfo consumption

- Update kernel entry signature to accept `bootinfo_ptr` in `R0`.
- Validate BootInfo: magic, version, size, alignment.
- Derive safe memory region boundaries from BootInfo (preferred).

5.11.2 Milestone OSv4.M2 — Minimal allocator

- Implement a bump allocator:
  - Heap base set to end of kernel image (or BootInfo-provided heap base).
  - Heap limit derived from RAM size and reserved top regions.
- Allocate and initialize kernel structures for tasks/run queue from the heap.

5.11.3 Milestone OSv4.M3 — Cooperative tasks

- Define TCB layout with alignment discipline.
- Implement `task_create`, `task_yield`, and context switch.
- Implement a round-robin runnable queue.

5.11.4 Milestone OSv4.M4 — Demonstration workload

- Create at least two tasks that:
  - Perform a visible action (e.g., console output if available),
  - Call `yield` repeatedly.
- Show deterministic alternation in the round-robin scheduler.

5.12 Definition of Done (OSv4)

OSv4 is complete when:

1. The kernel entry consumes BootInfo via `R0` and validates it.
2. The kernel uses a safe, non-overlapping memory layout for heap and stacks.
3. At least two cooperative tasks run and yield under a round-robin scheduler.
4. Fatal BootInfo validation failures halt deterministically (optionally printing an error tag).

5.13 Upgrade Notes

Path to OSv5. OSv5 introduces a true boundary:

- Dual-image build (`kernel.bin` + `user.bin`),
- User mode and protection rules enforced by the toolchain and CPU,
- Syscalls as the only legal service boundary for user code.

OSv4 prepares for OSv5 by establishing durable kernel structure (BootInfo-driven init, task model), so OSv5 can focus on enforcing privilege separation rather than inventing kernel foundations.

6 OSv5: Boundary Exists (User Mode + Protection + Dual Image)

6.1 Purpose

OSv5 is the first version that provides a **real privilege boundary**: **kernel** and **user** code are separate build artifacts, user code executes in user mode, memory and privileged operations are restricted, and **syscalls are the only supported service boundary**.

OSv5 keeps the OS minimal by limiting user space to a single address space and a small syscall API, but it is "real" because violations trap predictably and the toolchain enforces the model.

6.2 Scope and Non-Goals

6.2.1 In scope

- Dual-image build and runtime model: `kernel.bin` + `user.bin`.
- Kernel owns vectors and reset entry, sets up the system, then transitions to user mode.
- User code can only request services via **SYSCALL**.
- CPU-enforced protection: illegal user accesses to privileged regions cause a protection fault trap.
- A minimal kernel scheduler may run multiple user tasks.
- Timer interrupt may be used for optional preemption (now meaningful with masking).

6.2.2 Out of scope (explicit non-goals)

- No full process model with separate virtual address spaces.
- No filesystem.
- No networking stack.
- No demand paging or advanced memory management.

6.3 Target Platform Compatibility

OSv5 targets the final project milestone stack:

- **CPU:** v5 (privilege flag `K`, interrupt mask semantics via `I`, protection faults).
- **Toolchain:** v5 (dual-image layout and compile-time enforcement).
- **Boot:** v5 (kernel-installed vectors, kernel-to-user transition).
- **Compiler:** Phase 4 (dual-target compilation and user-side restrictions).

6.4 CPU v5 Privilege and Interrupt Model (Normative)

6.4.1 Privilege flag `K`

CPU v5 defines a privilege flag:

- `K=1`: kernel mode.
- `K=0`: user mode.

Normative rule. User-mode code (`K=0`) must not be able to:

- Access privileged memory regions (vectors `0x0000..0x00FF` and MMIO/reserved `0xFE00..0xFFFF`).
- Perform privileged operations (implementation-defined privileged instructions).
- Install vectors or change privileged control registers.

6.4.2 Interrupt mask flag `I`

CPU v5 defines an interrupt mask flag controlling delivery of timer interrupts:

- `I=1`: timer interrupts enabled.
- `I=0`: timer interrupts masked.

Kernel rule. The kernel may temporarily mask interrupts (`I=0`) while manipulating critical structures (e.g., run queue) and must restore `I` to its prior state afterward.

6.5 Memory Regions and Protection (Normative)

OSv5 relies on the CPU v5 protection model, which provides only a **minimal enforced boundary**: certain regions and operations are privileged and will trap in user mode. Beyond those privileged regions, RAM remains a single address space.

6.5.1 CPU-enforced privileged regions

The following address ranges are **kernel-only** under CPU v5 and must be owned by `kernel.bin`:

- **Vector table:** `0x0000..0x00FF`.
- **Reserved / MMIO window:** `0xFE00..0xFFFF`.

All other RAM, `0x0100..0xFDFF`, is accessible in user mode (`K=0`). Therefore, OSv5 must **not rely on secrecy or hardware isolation** for any kernel data placed in that range.

6.5.2 Software ownership layout (toolchain-enforced)

Within `0x0100..0xFDFF`, OSv5 defines a **software ownership layout** to keep the kernel and user images structurally separate:

- `kernel.bin` is linked/loaded into a kernel-owned subrange of `0x0100..0xFDFF`.
- `user.bin` is linked/loaded into a disjoint user-owned subrange of `0x0100..0xFDFF`.
- Kernel runtime allocations (e.g., kernel stack(s), heap, scheduler state) may reside in the kernel-owned subrange; the toolchain must ensure the user image does not overlap these regions.

This layout provides a predictable execution model and prevents accidental overlap at build time, but it is **not a confidentiality boundary**. The kernel must validate any pointers and sizes received from user mode and must copy data through checked buffers when crossing the syscall boundary.

6.5.3 Protection fault behavior

Protection faults occur only for privileged regions/operations (e.g., vector table access, MMIO/reserved window access, or privileged instructions):

- CPU raises a **protection fault** trap.
- Control transfers to the kernel's protection handler vector.
- The kernel must handle the fault deterministically (see §6.12).

6.6 Dual-Image Model (Toolchain v5, Normative)

OSv5 uses the toolchain v5 dual-image model:

- **`kernel.bin`**:
  - Contains vectors table and reset entry.
  - Contains kernel code/data and kernel runtime structures.
  - Executes in kernel mode (`K=1`).
- **`user.bin`**:
  - Contains user program(s) and minimal user runtime support.
  - Executes in user mode (`K=0`).
  - Must not contain privileged instructions or MMIO accesses (enforced by toolchain).

6.6.1 Toolchain enforcement rules

The toolchain must enforce at build time (conceptual requirements):

- User image code/data must lie within `0x0100..0xFDFF` and must not overlap `kernel.bin` placement or kernel-reserved runtime regions.
- User code must not directly reference MMIO/reserved addresses (lint violation).
- User code must not emit privileged instructions (lint violation).

6.7 Boot v5 Runtime Flow (Normative)

OSv5 boot flow is:

1. Reset enters kernel reset entry (kernel owns vectors/reset).
2. Kernel initializes:
   - stack(s), allocator, scheduler structures,
   - vectors and handlers,
   - console (optional) and timer (optional).
3. Kernel loads/locates `user.bin` program entry and prepares a user stack.
4. Kernel transitions to user mode (`K := 0`) and jumps to user entry.

Normative rule. Once in user mode, user code interacts with the kernel *only* via `SYSCALL` (or via traps caused by faults).

6.8 Vectors and Handlers (OSv5)

OSv5 retains the OSv3 vectors semantics and extends them with protection handling.

6.8.1 Required vectors

OSv5 must implement at least:

Vector	Name	Handler
`0x00`	Syscall entry	`isr_syscall`
`0x01`	Timer IRQ	`isr_timer` (optional preemption)
`0x03`	Protection fault	`isr_prot`

All other vectors may route to a default handler that halts or logs deterministically.

6.9 Syscall ABI (OSv5, Normative)

OSv5 continues the OSv3 syscall ABI:

- Syscall number in `R0`.
- Arguments in `R1..R3`.
- Return value in `R0`.
- Entry via `SYSCALL`; return via `IRET`.

6.9.1 Required syscall surface

OSv5 freezes at least the following syscalls:

Syscall #	Name	Semantics
`1`	`putc`	Output low 8 bits of `R1` to console
`2`	`exit`	Terminate current user task (or entire system if single task)
`3`	`yield`	Yield CPU to scheduler (cooperative)

Notes.

- Syscall #1 matches the proof-of-life output mechanism used earlier.
- Syscall #2 is the stable *termination* syscall reserved in OSv3. In OSv5 it is exposed to user mode as `exit` (terminate current user task); if no other task is runnable, the system must halt deterministically.
- Syscall #3 supports cooperative scheduling in user space.

6.10 User Runtime Contract (OSv5)

6.10.1 User entry point

- User image must expose a `user_entry` (or `_start`) symbol.
- Kernel transfers control to user entry in user mode (`K=0`).

6.10.2 User restrictions

In OSv5, user code must:

- Avoid direct MMIO access.
- Avoid privileged instructions.
- Use syscalls for I/O, exit, and yield.

6.11 Scheduler Model (OSv5 Minimal)

OSv5 may schedule one or more user tasks.

6.11.1 Minimal requirement

- Support running at least one user program to completion.
- Support syscall-based `yield` if multiple tasks are configured.

6.11.2 Optional preemption

If enabled, timer interrupts may drive preemption:

- Timer handler increments `tick`.
- Periodically, handler triggers a context switch.
- Kernel may mask interrupts (`I=0`) while switching.

6.12 Protection Fault Policy (Normative)

When `isr_prot` is invoked due to a protection fault from user mode, the kernel must:

1. Record or print a short diagnostic (optional if console exists).
2. Terminate the offending user task deterministically.
3. Continue scheduling other runnable user tasks if any; otherwise HALT.

Normative safety rule. The kernel must not panic the entire system for a single user fault unless no recovery policy is implemented (single-task configuration). In the minimal case, halting is acceptable.

6.13 Plan of Execution (OSv5)

6.13.1 Milestone OSv5.M1 — Dual-image build + kernel owns boot

- Configure build to emit `kernel.bin` and `user.bin`.
- Ensure kernel image contains vectors and reset entry.
- Ensure user image passes toolchain range checks and MMIO/privileged lints.

6.13.2 Milestone OSv5.M2 — Kernel-to-user transition

- Kernel locates user entry and allocates a user stack.
- Kernel sets `K := 0` and jumps to user entry.
- Verify user executes and returns only via syscalls.

6.13.3 Milestone OSv5.M3 — Syscalls as boundary

- Implement syscall dispatcher with `putc`, `exit`, `yield`.
- Provide user-side stubs that invoke `SYSCALL`.
- Demonstrate a user program that prints and exits via syscalls only.

6.13.4 Milestone OSv5.M4 — Protection fault handling

- Implement protection fault handler at vector `0x03`.
- Demonstrate that an illegal user access triggers the handler deterministically.
- Apply fault policy: terminate task and continue (or HALT if single task).

6.13.5 Milestone OSv5.M5 — Optional timer preemption

- Enable timer interrupts (`I=1`) in kernel after initialization.
- Use timer ticks to drive optional preemptive switching.
- Mask interrupts in critical sections as needed (`I=0`).

6.14 Definition of Done (OSv5)

OSv5 is complete when all of the following are true:

1. Dual images build successfully: `kernel.bin` and `user.bin`.
2. Kernel boots, installs vectors, and transitions to user mode correctly.
3. User program performs console output via syscall #1 and terminates via syscall #2.
4. Illegal user access triggers protection fault handler and is handled deterministically.
5. If multiple tasks are enabled, `yield` switches tasks correctly (cooperative).

6.15 Compatibility and Design Notes

Compatibility with OSv3/OSv4. OSv5 preserves:

- The syscall ABI conventions introduced in OSv3.
- The structured kernel foundation (BootInfo-driven init, allocator, task model) introduced in OSv4.

Minimal but real. OSv5 remains minimal by not implementing processes, virtual memory, or filesystems, but it is real because user code is constrained by enforceable rules and violations trap predictably.