UmkaOS: A Rust Kernel for 50-Year Uptime

Download as PDF

What UmkaOS Is

UmkaOS is a production kernel written entirely in Rust, designed as a drop-in Linux replacement. The name echoes UNICOS/mk, Cray Research's distributed microkernel for the T3E — a system that was doing multikernel thinking decades before it became fashionable. The goal is that an existing Ubuntu or RHEL installation can swap its kernel package for `umka-kernel` and boot normally — unmodified glibc, systemd, Docker, Kubernetes, and the full userspace stack run without recompilation. The external ABI — syscall numbers, ioctl values, procfs/sysfs layouts, signal semantics, ELF loading, ptrace, netlink — matches Linux exactly. The internal implementation is redesigned for crash isolation, live evolution, and hardware heterogeneity.

Why Rust: No undefined behavior. The ownership model eliminates data races at compile time. Unsafe blocks are audited and carry mandatory `// SAFETY:` comments. Every public and private item is documented. Clippy enforces cognitive complexity limits, function size limits, and exhaustive enum matching. The result is a kernel where entire classes of bugs — use-after-free, double-free, buffer overflow, data races — are structurally impossible outside of explicitly marked unsafe boundaries.

Rust also enables compile-time lock ordering: locks carry a const generic level parameter, and acquiring a lower-level lock while holding a higher-level one is a type error, not a runtime lockdep warning. This eliminates an entire category of deadlocks that Linux can only detect at runtime (and only when the right code paths are exercised).

The target is ~99.99% Linux userspace compatibility across 8 architectures (x86-64, AArch64, ARMv7, RISC-V 64, PPC32, PPC64LE, s390x, LoongArch64) with less than 5% overhead on macro benchmarks versus Linux.

What was deliberately dropped: UmkaOS does not support Linux `.ko` kernel modules (KABI replaces them), `/dev/mem` or `/dev/kmem` (incompatible with memory domain isolation), kexec (live evolution replaces it), or ia32 compatibility mode on 64-bit kernels (no 32-bit syscall table — 32-bit binaries run on dedicated 32-bit kernel builds). These are intentional design decisions, not gaps.

This post covers what UmkaOS does differently from Linux, why, and what the practical consequences are.

---

1. Driver Crash Isolation Without Microkernel Overhead

Linux runs all drivers in Ring 0 with full kernel memory access. A null pointer dereference in a NIC driver takes down the entire system. Microkernels (L4, seL4, QNX) solve this by running drivers in userspace, but the IPC overhead makes high-throughput I/O impractical at 100+ Gbps.

UmkaOS uses a three-tier protection model:

- **Tier 0**: Core kernel code (scheduler, memory allocator, capability system). Runs in Ring 0 with full access. A bug here is a kernel panic — same as Linux.
- **Tier 1**: Performance-sensitive drivers (NIC, NVMe, filesystem). Run in Ring 0 but inside hardware-isolated memory domains. On x86-64, this uses Memory Protection Keys (MPK) — a single instruction (~23 cycles) switches the active domain. The driver cannot read or write core kernel memory

or other drivers' memory. If it crashes, the core kernel revokes its DMA access, resets the device, and hot-reloads the driver — typically within 50-150ms. Applications see a brief latency spike, not a reboot.

- **Tier 2**: Untrusted or less performance-sensitive drivers. Run in Ring 3 (userspace) with full process isolation and IOMMU. Crash recovery in ~10ms. Same model as a microkernel, used when isolation is more important than raw speed.

Each of the 8 supported architectures has its own fast-isolation mechanism: MPK on x86-64, Permission Overlay Extension on newer ARM, Domain Access Control on ARMv7, segment registers on PPC32, Radix PID on PPC64LE. On architectures without a fast mechanism (RISC-V, s390x, LoongArch64), Tier 1 drivers run as Tier 0 — a documented tradeoff, not a gap.

Tier assignment is an operational decision, not a compile-time one. Every KABI driver ships with support for all three tiers in the same binary. The administrator chooses the tier at deployment time — or changes it at runtime without recompiling or reloading the driver. A latency-sensitive trading deployment might run its NIC driver in Tier 0 (maximum performance, no isolation overhead). A multi-tenant cloud host might run the same driver in Tier 1 (crash containment at <1% overhead) or even Tier 2 (full userspace isolation for untrusted third-party drivers). The choice is per-driver, per-host, and reversible:

```
echo 0 > /ukfs/kernel/drivers/ixgbe/tier    # Run at Tier 0 (max performance)
echo 1 > /ukfs/kernel/drivers/ixgbe/tier    # Run at Tier 1 (crash containment)
echo 2 > /ukfs/kernel/drivers/ixgbe/tier    # Run at Tier 2 (full isolation)
```

The kernel validates the request against signing certificate constraints, license ceilings, and architecture capability — an unsigned driver can't be set to Tier 0.

Automatic demotion: If a Tier 1 driver crashes repeatedly (3 times in 60 seconds), the kernel automatically sets it to Tier 2 — stronger isolation at the cost of some performance. The admin is notified via FMA. If the driver stabilizes, the admin can set it back to Tier 1 or Tier 0. This means a buggy driver update doesn't take down a production server; it gracefully degrades to a slower but safer execution mode.

What Linux can't do: Linux has no equivalent of Tier 1, and no concept of runtime tier mobility. A Linux driver is either a kernel module (Ring 0, full trust) or a VFIO userspace driver (Ring 3, with heavy IPC overhead). The decision is architectural and permanent — you can't move a driver between privilege levels without rewriting it. UmkaOS makes this a deployment knob.

Performance budget: The cumulative overhead for an nginx-class workload (TCP RX + NVMe read + TCP TX + context switches) is ~2.2% on x86-64 with all security features enabled. Seven optimizations — register shadowing, NAPI batch amortization, doorbell coalescing, capability token caching, RCU deferred quiescent state, CpuLocal register access, and debug-only PerCpu CAS — are built into the design from day one, not deferred optimizations.

For non-batched RPC workloads (single-request gRPC, database transactions), per-request overhead is higher (1.2-3.5% depending on cache temperature) because NAPI and doorbell batching can't amortize across requests. The tail latency target is under 8% at p99 on fast-isolation architectures. This is documented honestly — the 5% throughput budget and the 8% tail latency budget are separate targets.

---

2. Live Kernel Evolution: Never Reboot Again

Linux kernel updates require a reboot (or kexec, which still drops all process state). kpatch/livepatch can hot-fix individual functions but cannot replace entire subsystems or change data structures.

UmkaOS can replace any kernel subsystem — the scheduler, the TCP stack, the filesystem layer, the memory allocator — while the system is running, without dropping connections or losing state.

How it works: The kernel is split into two domains based on replaceability. The **Nucleus** (~2-3 KB of verified code) handles capability table lookups and the atomic swap mechanism — it is never replaced. Everything else is **Evolvable**: factored into non-replaceable data structures and replaceable, stateless policy algorithms.

Replacement is a four-phase protocol: prepare (load new code, export state), validate (import state with dry-run), swap (IPI halts all CPUs for ~1-10 µs, atomic vtable pointer swap), resume (all CPUs continue with new code). For stateless policies like congestion control or page reclaim heuristics, replacement is a single atomic pointer swap — under 1 µs, invisible to applications.

The state-spill problem: The hardest part of live evolution is preventing developers from accidentally storing mutable state in a replaceable module. If someone puts a counter or a cache in the swappable code, it's silently lost on replacement. UmkaOS addresses this with three layers of defense:

1. **Link-time**: The module loader rejects modules with any mutable global state (static variables, thread-locals). This check operates on ELF sections and has zero false negatives.
2. **Compile-time**: A proc macro verifies that the module entry point is a zero-sized type with no mutable fields.
3. **SDK types**: The driver SDK provides state handles that make it natural to put data in the right place (Nucleus-owned memory) without thinking about the split.

State migration across versions: When a new module version adds fields to its persistent state, the SDK generates automatic in-place migration from version annotations. For breaking changes (type conversions, unit changes), developers write explicit migration functions. Per-connection state (e.g., TCP congestion control) migrates lazily on next packet — no thundering herd.

Cross-language support: The KABI IDL compiler generates identical state definitions and migration scaffolding for both Rust and C from a single source file. A C driver can be hot-swapped for a Rust implementation (or vice versa) with binary-compatible persistent state. The choice of language is a developer preference, not an architectural constraint.

What Linux can't do: Linux has no framework for replacing subsystems with state transfer. kpatch replaces individual functions; it cannot change struct layouts, swap data structures, or migrate per-connection state. The Nucleus/Evolvable split is architecturally impossible to retrofit into Linux's monolithic code organization.

---

3. KABI: Stable Driver ABI

Linux has no stable kernel ABI. Internal APIs change every release. Out-of-tree drivers (NVIDIA, ZFS) break regularly and require per-version adaptation layers.

UmkaOS provides KABI (Kernel ABI) — a stable, versioned interface between the core kernel and drivers:

- **KABI IDL**: A declarative interface definition language. Describes vtable methods, struct layouts, and capability requirements.
- **Code generation**: The KABI compiler produces both Rust and C bindings from the same source — binary-compatible across languages.
- **Versioned vtables**: Append-only design. New methods are added at the end; removed methods get tombstone stubs. The core kernel detects which methods a driver supports from the vtable size.

- **5-release compatibility window**: Drivers compiled against any of the last 5 KABI versions work without recompilation.
- **Post-quantum signing**: All driver modules are signed with ML-DSA-44 or SLH-DSA-128f. Verified against a Kernel Revocation List at load time.

What Linux can't do: Linux's internal API stability policy is "we don't guarantee it." DKMS recompiles drivers on every kernel update. UmkaOS's versioned KABI means a driver compiled today works unchanged for years across kernel updates.

---

4. Tier M and the Multikernel Peer Model

A modern server contains multiple autonomous processors: DPUs running embedded Linux, SmartNICs with their own RTOS, GPUs with dedicated thread schedulers. Linux treats all of them as dumb peripherals requiring million-line host-side drivers in Ring 0.

UmkaOS introduces **Tier M** (Multikernel Peer) — a fourth protection class alongside Tier 0/1/2. A Tier M device runs its own kernel (UmkaOS or a compatible shim) and communicates with the host via a standardized wire protocol over PCIe P2P or RDMA ring buffers. The host-side code for any Tier M peer is ~2,000 lines of generic transport — no device-specific driver code.

Three paths to the same protocol:

- **Path 1 — Traditional driver**: For legacy hardware (USB controllers, simple AHCI). Standard Tier 0/1/2 driver model. Nothing changes.
- **Path 2 — Full UmkaOS port**: The device runs a full UmkaOS kernel instance. Host and device speak the peer protocol natively. Best for DPUs with general-purpose ARM/RISC-V cores (NVIDIA BlueField, AMD Pensando).
- **Path 3 — Firmware shim**: The device keeps its vendor firmware and implements the UmkaOS peer protocol as a thin shim (~8K-12K lines). The host cannot distinguish a shim from a full port. Best for devices with constrained firmware environments.

All three paths coexist on the same system. A host can have traditional NVMe drivers (Path 1), a BlueField DPU running UmkaOS (Path 2), and an Intel IPU with a protocol shim (Path 3) — all managed through the same capability system, the same topology graph, the same service discovery.

Isolation is physical, not just software: The Tier M boundary is a hardware fabric (PCIe BAR + IOMMU domain). A compromised peer cannot access host memory outside its granted capabilities. The host retains unilateral reset (FLR/SBR) over any peer. Capability scoping ensures each peer gets only the services it needs — a network-processing DPU has no path to the storage subsystem.

Practical consequence: When a new DPU supports the peer protocol, it works with UmkaOS immediately. No host kernel patch, no driver development cycle, no vendor coordination. The device advertises its capabilities, the host binds services, and data flows through shared ring pairs.

One protocol, multiple transports: The peer protocol that Tier M devices speak is the exact same protocol that cluster nodes use for distributed coordination (next section). A BlueField DPU connected via PCIe and a remote server connected via RDMA Ethernet use identical message formats, capability negotiation, and service discovery — the only difference is the physical transport. This means a service running on a local DPU can transparently migrate to a remote cluster node (or vice versa) without any protocol translation. The topology graph treats local PCIe peers and remote RDMA peers as entries in the same namespace, differing only in measured latency and bandwidth.

What Linux can't do: Linux's driver model assumes host-side control of all device logic. There is no concept of a device as a peer with its own kernel. Supporting a new DPU in Linux requires a massive host-side driver (NVIDIA's BlueField driver is >100K lines). UmkaOS replaces that with ~2,000 lines of generic transport code.

---

5. Distributed Kernel: RDMA, DSM, and Cluster Coordination

Linux treats each machine as an isolated node. Cluster coordination — distributed locking, shared memory, device sharing — is entirely in userspace.

UmkaOS makes the cluster a first-class kernel abstraction using the same peer protocol as Tier M devices — a remote server and a local DPU are both "peers" in the topology graph, differing only in transport latency:

Distributed Shared Memory (DSM): Page-granular MOESI coherence over RDMA. When a process faults on a remote page, the kernel fetches it transparently via one-sided RDMA read (~2-5 µs). The Owned state enables direct peer-to-peer data transfer — Node A sends a dirty page directly to Node B without writing back to the home node first, reducing home-node bottleneck by 40-60% for read-heavy workloads. Multiple consistency modes: strict MOESI, causal consistency with vector clocks, relaxed with anti-entropy reconciliation.

Distributed Lock Manager (DLM): Kernel-native locking with cluster-wide wait-for-graph deadlock detection. Unlike timeout-based DLMs, UmkaOS detects and breaks deadlocks in real time.

CXL 3.0 integration: CXL-attached memory is a first-class tier in the memory hierarchy. The page cache migrates cold pages to CXL pools and hot pages to local DRAM. Combined with RDMA tiers, the memory hierarchy spans local DRAM, CXL, compression tier, swap, and RDMA remote memory — managed by a single policy.

Application-visible DSM: Beyond kernel-internal use, DSM is exposed to userspace via syscalls. Applications can create shared memory regions that span cluster nodes, with choice of consistency model. This enables distributed databases and HPC applications to share memory across machines without MPI or custom RDMA verbs code.

Distributed futex: `futex_wait` on a DSM page works transparently across nodes. If the futex word is on a remote page, the kernel handles the RDMA fault, the remote wakeup, and the coherence protocol. POSIX mutexes in shared DSM memory work as expected — the application doesn't need to know the memory is distributed.

What Linux can't do: Linux has no kernel-native DSM (previous attempts like kerrighed/MOSIX failed due to coherence overhead on 1GbE — RDMA changes the cost structure by 1000x). Linux's DLM (used by GFS2) has no real-time deadlock detection. CXL support in Linux exists but isn't integrated into page cache tiering. Linux has no distributed futex — cross-node synchronization requires explicit network messaging.

---

6. Scheduling: EEVDF, CBS Guarantees, and Energy-Aware Placement

UmkaOS uses EEVDF (Earliest Eligible Virtual Deadline First), matching the algorithm Linux adopted in 6.6. The implementation adds several features Linux doesn't have:

- **CBS (Constant Bandwidth Servers)**: Hard CPU bandwidth guarantees per cgroup. A container guaranteed 30% CPU actually gets 30%, regardless of how busy the system is. This is a minimum guarantee, not just a maximum limit.
- **Deferred dequeue**: Over-served tasks stay in the tree with negative lag, decaying passively. No unfair penalty for bursting.
- **Latency-nice**: Shifts eligibility window rather than granting more CPU — latency-sensitive tasks preempt immediately without monopolizing bandwidth.
- **Energy-Aware Scheduling (EAS)**: Uses firmware-provided power data (ACPI, device tree) to place tasks on the most energy-efficient cores. Separate from RAPL power budgeting (which caps total watts).
- **Core provisioning**: Dedicated cores with tickless operation, offloaded RCU callbacks, and zero softirq interference for latency-critical workloads.

What Linux can't do: Linux 6.6+ has EEVDF but not CBS guarantees. Linux's EAS is ARM-focused; x86 support is limited. UmkaOS's CBS + EAS + core provisioning provides a unified resource model that Linux approximates with scattered mechanisms (cpusets, isolcpus, nohz_full, rcu_nocbs) that don't compose well.

---

7. Security: Capabilities, PQC, and Confidential Computing

Capability model: Every resource is accessed via capability tokens tracked in a global table. 16-level delegation hierarchy — a process can grant restricted subsets of its privileges to sub-services. O(1) revocation via generation counter increment. Capabilities are network-portable for cluster-wide access control, with a fast local-only path that doesn't block.

Post-quantum cryptography (PQC): - Boot chain: hybrid Ed25519 + ML-DSA-65 signatures (classical + lattice-based). - Driver signing: ML-DSA-44 or SLH-DSA-128f (stateless hash-based). - DSM and peer traffic: X25519 + ML-KEM-768 hybrid key exchange with automatic rotation. - The crypto API is aware of which algorithms perform internal hashing (FIPS 204/205) versus which expect pre-computed digests — this distinction is critical for correct PQC verification and is handled automatically.

Confidential computing: First-class support for Intel TDX, AMD SEV-SNP, and ARM CCA. Device passthrough via capability-gated VFIO/iommufd.

Hardware memory safety: ARM MTE tag-aware allocator, Pointer Authentication (PAC), Intel LAM. Tag faults trigger autonomous fault management rather than silent corruption.

What Linux can't do: Linux capabilities are a flat bitmask with no delegation hierarchy and no network portability. Linux has no kernel-native PQC — it's userspace-only via OpenSSL/GnuTLS. UmkaOS's PQC protects the kernel itself, from boot to driver loading to cluster communication.

---

8. Memory Management

Hot-pluggable metadata: Page descriptors are virtually mapped (vmemmap). Adding terabytes of CXL memory at runtime requires no relocation of existing metadata — just allocate new backing pages for the vmemmap range.

Replaceable allocator policy: The physical allocator's data (free lists, zones, watermarks) is separated from its policy (block selection, NUMA fallback heuristics). The policy can be hot-swapped at runtime via

the live evolution mechanism — useful for adapting to workload changes over a multi-year deployment.

Per-CPU magazine slab: Lock-free fast path using architecture-specific per-CPU registers. No global contention for common allocation sizes. NUMA-pinned backing.

Memory tiering: A single reclaim policy manages the full hierarchy — local DRAM, CXL, compression tier, swap, RDMA remote. Each tier has measured latency; the policy (evolvable, optionally ML-informed) migrates pages between tiers based on access heat.

Process memory hibernation: When memory pressure is severe, Linux kills processes (OOM killer). UmkaOS offers an alternative: hibernate a process's entire address space to swap and suspend it. When pressure subsides, the process is woken and its pages fault back in on demand. The application resumes where it left off — no data loss, no restart. The OOM killer still exists as a last resort, but hibernation handles the common case of transient pressure spikes.

What Linux can't do: Linux cannot hot-swap its memory allocator policy. Linux's OOM killer is kill-or-nothing — there is no hibernate-and-resume option. Linux's CXL support doesn't integrate into page cache tiering. UmkaOS's memory compression tier is a first-class NUMA-aware tier, not a bolt-on block device.

---

9. VFS and Storage

Ring buffer protocol: All communication between the core kernel and Tier 1 filesystem drivers uses shared-memory ring buffers. VFS operations are serialized as messages, enabling pipeline parallelism — the core validates the next request while the driver processes the current one.

Two-phase dirty extents: Reserve, Commit, Complete. If a driver crashes between reserve and commit, the core reconciles from the intent list. Abort path for repeated writeback failures.

Dual error reporting: Per-fd generation tracking for fsync correctness (each open fd sees each error exactly once), plus boolean error flags for fast writeback scanner checks. Both set atomically on I/O completion.

What Linux can't do: Linux's VFS is synchronous function calls — no pipeline parallelism across isolation boundaries. The ring buffer protocol is what makes isolated filesystem drivers practical at high IOPS.

---

10. Networking

NetBuf: Replaces Linux's sk_buff with a compact metadata header referencing DMA-accessible data pages. Zero-copy from NIC to socket. Fixed-size ring entries for cross-domain transport — no per-packet allocation on the isolation boundary.

NAPI batch amortization: Domain switches are amortized over batches of 64 packets. Per-packet isolation overhead is negligible. Adaptive interrupt coalescing dynamically trades latency for throughput based on packet rate.

XDP in dedicated domain: XDP programs run in their own isolation domain, separate from the NIC driver. The driver copies packet descriptors to a bounce buffer; the XDP program processes them without direct access to driver state (DMA rings, device registers).

Modular TCP: Congestion control is a hot-swappable policy module. Per-connection state lives in the socket; the algorithm is replaced atomically. Existing connections keep the old algorithm; new connections pick up the new one.

What Linux can't do: Linux's XDP runs in the driver's address space with no isolation. Linux's sk_buff is variable-sized with multiple allocations per packet. UmkaOS's fixed-size ring entries eliminate per-packet allocation on the cross-domain path.

---

11. Virtualization: KVM With Stronger VMM Isolation

UmkaOS includes a full KVM hypervisor backend supporting Intel VT-x/EPT, ARM VHE/Stage-2, and RISC-V H-extension. The performance-critical VM entry/exit path runs in Tier 0 (Ring 0), same as Linux. The VMM device model (MMIO emulation, virtio backends, migration logic) runs as a Tier 2 isolated service — which is functionally the same as Linux's model (QEMU/cloud-hypervisor are already Ring 3 processes). The performance difference is effectively zero.

The benefit is **isolation, not speed**: Linux's VMM runs in Ring 3 but with broad access to the KVM ioctl interface. A compromised VMM (QEMU has had many CVEs in device emulation code) can potentially leak host memory or escalate privileges. UmkaOS's Tier 2 containment means the VMM has no access to host kernel memory beyond what capabilities explicitly grant. A VMM bug kills the guest cleanly; the host is unaffected.

Live migration: Pre-copy iterative page transfer, post-copy on-demand faulting, and a convergence algorithm that throttles the guest when dirty rate exceeds network bandwidth.

Confidential computing: First-class Intel TDX, AMD SEV-SNP, and ARM CCA support. Device passthrough via capability-gated VFIO/iommufd.

What Linux can't do: Linux KVM's VMM isolation relies on process boundaries and seccomp filtering. UmkaOS adds capability-scoped access — the VMM can only interact with the specific devices and memory regions it was granted, enforced by the kernel's capability system, not by syscall filtering heuristics.

---

12. io_uring: Deep Integration

UmkaOS implements io_uring with deep integration into the ring buffer architecture. Because VFS and block I/O already communicate via shared-memory rings (for Tier 1 isolation), io_uring's submission/completion rings fit naturally — there is no impedance mismatch between the userspace ring and the kernel's internal data path.

SQPOLL mode recovers doorbell coalescing even for single-request workloads (the poll thread batches submissions from the ring), which is important for the non-batched RPC performance story.

Fixed file tables, registered buffers, and linked SQEs (including hardlinks for unconditional chains) are fully supported.

What Linux can't do differently: Linux's io_uring is excellent but lives in a monolithic kernel where VFS calls are synchronous function calls. UmkaOS's ring-native architecture means io_uring submissions flow through the same ring mechanism as all other cross-domain I/O — no special-case path.

---

13. eBPF: Isolated Execution Domain

UmkaOS runs all eBPF programs — XDP, TC classifiers, kprobes, tracepoints, cgroup hooks — in a dedicated isolation domain separate from both the core kernel and driver domains. On x86-64, BPF programs get their own MPK protection key. A logically flawed BPF program (e.g., one that corrupts its map data or enters an unexpected state) cannot touch core kernel memory or driver memory.

The verifier and JIT compiler match Linux's semantics for program compatibility. All 212 Linux 6.12 BPF helpers are supported (generated at build time from Linux headers). UmkaOS adds a small set of native helpers for per-CPU counters and kernel parameter access.

What Linux can't do: Linux runs verified BPF programs directly in Ring 0 with full kernel memory access. If the verifier has a bug (and verifier bugs are found regularly), the BPF program can read or write arbitrary kernel memory. UmkaOS's domain isolation provides defense-in-depth even if the verifier is imperfect.

---

14. Native Multi-Object Wait

Linux has no single call to wait on heterogeneous kernel objects. Waiting for "socket data OR timer OR child exit OR event" requires converting everything to file descriptors (timerfd, pidfd, eventfd, signalfd) and funneling through epoll.

UmkaOS provides native multi-object wait on mixed handle types — file descriptors, named events, process IDs, inline timers, and semaphores in a single call. Existing epoll code continues to work unchanged; the native wait is an additional option for cases where epoll's "everything must be an fd" requirement forces unnecessary complexity.

Named events (manual-reset or auto-reset) are visible in the kernel filesystem and work cross-process within a namespace.

What Linux can't do: futex_waitv (Linux 5.16+) waits on multiple futexes, but only futexes — not fds, not pids, not timers. UmkaOS's wait is truly heterogeneous.

---

15. Windows Emulation Acceleration (WEA)

Wine/Proton emulate Windows NT kernel behavior in userspace. The wineserver IPC for synchronization operations costs ~15-25 µs per operation. UmkaOS provides kernel-native NT primitives that Wine calls directly.

Key insight: WEA wraps native UmkaOS primitives. The kernel has one wait implementation (the multi-object wait from the previous section), and WEA translates NT handle types into it. NT Events use UmkaOS named events; NT Mutexes add ownership tracking; I/O Completion Ports wrap io_uring completion queues. There is no separate "Windows subsystem" — just a translation layer.

Projected speedups: ~10x for named event creation, ~16x for multi-object wait, ~5x for IOCP operations. Additional features: NT object namespace (hierarchical, named), in-kernel PEB/TEB mapping (avoids segment register swap overhead on every syscall), VirtualAlloc flag translation, structured exception handling via signal delivery.

WEA is opt-in via a capability — zero overhead for non-Wine processes.

What Linux can't do: Linux has no kernel-native NT object model. Wine's wineserver is an IPC bottleneck that cannot be eliminated without kernel support.

---

16. Observability and Fault Management

FMA (Fault Management Architecture): A unified sense-think-act remediation loop. Hardware errors, driver crashes, and ECC events are ingested as structured events. Rule-based diagnosis (with optional ML inference) identifies root causes. Autonomous remediation: driver reload, page retirement, device quarantine, node eviction. The escalation policy is consistent across all subsystems.

umkafs (`/ukfs/`): UmkaOS's native kernel filesystem. Separate from sysfs (which remains for Linux compatibility). Provides structured access to kernel parameters, driver state, tracing, health events, and power management. Lowercase paths, admin-friendly.

Stable tracepoints: ABI-stable tracepoint IDs with versioned argument schemas. Tracepoint ABI survives kernel updates — monitoring tools don't break on upgrade.

EDAC: Proactive page retirement driven by correctable-error rate trending. Degrading DIMMs are detected before uncorrectable errors reach applications.

What Linux can't do: Linux's tracepoint IDs are not ABI-stable. Linux has no integrated fault remediation loop — EDAC, MCE, and driver recovery are separate, uncoordinated subsystems. UmkaOS unifies them under FMA.

---

17. ML Policy Framework

Static kernel heuristics (page reclaim watermarks, readahead windows, I/O reordering thresholds) are tuned for average workloads. UmkaOS replaces them with evolvable, integer-only ML models that adapt to specific hardware and workload patterns.

Every inference result is bounded by administrator-defined safety ranges. If the ML engine fails to refresh a parameter, the kernel automatically reverts to conservative fallback heuristics. Per-cgroup overrides allow different workloads to use different tuning. Policy updates are HMAC-validated to prevent model poisoning.

The data layer (parameter store, observation ring buffers) is non-replaceable. The orchestration layer (model registration, decay timers) is evolvable.

What Linux can't do: Linux has no in-kernel inference framework. Kernel tuning is manual (sysctl) or external (BPF programs poking at parameters). UmkaOS's closed-loop optimization adapts to hardware and workload in real time.

---

18. Containers, TTY, and User I/O

Containers: All 8 Linux namespace types. cgroups v2 with v1 compatibility shim. OCI runtime support. Overlayfs with metadata squashing for fast lookups in multi-layer images.

Lockless TTY: Per-session lock-free ring buffers replace Linux's global TTY lock. Scales linearly with session count. Separate control ring for signals — signal delivery never blocks on a full output buffer. Zero-copy logging mode for high-density container environments.

Audio, input, display: Full Linux-compatible ALSA, evdev, and DRM/KMS interfaces. Device drivers run in Tier 1 isolation.

---

19. Accelerators and AI

Unified accelerator framework: GPU, inference engine, FPGA, and custom accelerators share a common scheduling model with bandwidth servers. The same cgroup knobs control GPU time as control CPU time.

In-kernel inference: Tiny quantized integer-only models for kernel policy decisions. CPU-budget-limited. Cycle watchdog prevents runaway inference. Used by page reclaim, readahead, I/O scheduling, and thermal management.

P2P DMA: GPU-to-GPU and GPU-to-NVMe direct transfers using shared IOMMU domains.

---

20. Development Model: AI-Built From Day One

UmkaOS is developed using agentic programming — AI agents perform both architecture design and implementation directly from specification documents. This shapes the architecture itself:

- Every struct, every error path, every cross-subsystem interaction is explicitly specified in a 25-chapter, 325-section design document.
- Cross-references are bidirectional — implementing one section never requires discovering hidden dependencies in another.
- Hardware facts (syscall numbers, ISA features, crypto standards) are verified against current sources, not LLM training data.

The architecture document is the canonical reference. All implementation traces back to it.

---

21. The 50-Year Uptime Design Goal

The title says "50-Year Uptime." This is a design constraint, not a proven claim — UmkaOS hasn't been running for 50 years and neither has anything else. What it means is that every architectural decision is evaluated against the question: "does this survive decades of continuous operation?"

The mechanisms that address this, scattered across the sections above, connect into a single longevity story:

- **Live evolution** (§2) — kernel subsystems are replaced without reboot. Kernel updates that would require a reboot on Linux are applied live. Over decades, this is thousands of avoided reboots.
- **Driver crash recovery** (§1) — Tier 1 driver crashes are contained and hot-reloaded. The most common cause of Linux reboots (driver bugs) becomes a recoverable event.
- **FMA autonomous remediation** (§16) — hardware degradation is detected, diagnosed, and remediated automatically. Failing DIMMs are retired before uncorrectable errors reach applications.

- **EDAC proactive page retirement** (§16) — correctable ECC errors trigger page retirement before they become uncorrectable. The physical memory footprint gracefully shrinks over years as pages are retired.
- **Replaceable policies** (§2, §8) — the memory allocator, page reclaim, scheduler, congestion control, and I/O scheduler can all be swapped live. New algorithms developed in 2040 can be deployed to a kernel that booted in 2026.
- **u64 counters** — every kernel-internal identifier uses u64. At worst-case increment rates, the earliest counter exhaustion is ~584 billion years. Protocol-mandated u32 counters have documented wrap-safety analysis.
- **Process memory hibernation** (§8) — transient memory pressure pauses processes instead of killing them. Data loss from OOM kills is eliminated for the common case.

None of this proves 50-year uptime. It proves that the architecture doesn't contain mechanisms that *prevent* it — no counter that wraps in 5 years, no subsystem that can't be updated without reboot, no hardware failure that forces a restart. Whether the implementation achieves it is an engineering question that will take years to answer.

---

22. What Ships When

Phase	Goal	Key subsystems
1 (current)	Boot to hello-world on all 8 architectures	Concurrency primitives, allocators, EEVDF, capability system, KABI compiler, isolation infrastructure
2	Busybox shell with Tier 1 crash recovery demo	VFS, page cache, VMM, fork/exec, signals, futex, epoll, VirtIO-blk
3	systemd + Docker	ext4, TCP/IP, AF_UNIX, NVMe, cgroups, namespaces, io_uring, eBPF
4	Production ready	Top-20 hardware drivers, LTP test suite, performance parity
5	Full platform	All arch ports, GPU, distributed kernel (DSM, DLM, peer protocol)

---

23. Who Benefits and How

Different deployments care about different features. Not everything matters everywhere.

Cloud / data center (the primary target): - Driver crash isolation (§1) eliminates the most common cause of unplanned reboots. A NIC driver bug at 3 AM triggers auto-reload, not a page to the on-call engineer. - Live evolution (§2) means kernel security patches deploy without maintenance windows. Zero planned downtime for CVE remediation. - CBS bandwidth guarantees (§6) give tenants real SLAs, not "best effort." A noisy neighbor can't steal CPU from a guaranteed workload. - FMA (§16) automates the remediation that ops teams currently do manually — page retirement, driver reload, device quarantine. - Tier mobility (§1) lets the same kernel image serve both latency-sensitive bare-metal deployments (Tier 0 drivers) and multi-tenant isolation-first deployments (Tier 1/2 drivers) without rebuilding anything.

HPC / AI training clusters: - DSM (§5) eliminates the MPI-only bottleneck. Shared memory across nodes with tunable consistency — strict MOESI for correctness-critical data, relaxed for bulk training data. - CXL memory tiering (§5) treats disaggregated memory as a transparent tier. Model parameters that don't fit in local DRAM migrate to CXL or RDMA remote memory automatically. - Peer protocol (§4) lets GPU-equipped DPUs join the cluster fabric without host-side megadrivers. - P2P DMA (§19) enables GPU-to-NVMe and GPU-to-GPU transfers without CPU bounce.

Gaming / Steam Deck / Valve consoles (via Proton): - WEA (§15) is the headline feature here. 10-16x speedup for Windows game synchronization primitives means fewer frame drops in games that use `WaitForMultipleObjects` heavily (which is most of them). - Lockless TTY (§18) helps for game server hosting — thousands of concurrent sessions don't fight over a global lock. - Audio in Tier 1 (§18) means an ALSA driver crash doesn't kill the kernel — it reloads in ~100ms. You hear a click, not a hard reboot.

Desktop / laptop: - Driver crash recovery (§1) is the killer feature for desktop users. GPU, WiFi, and Bluetooth driver bugs are the top three causes of Linux desktop instability. With Tier 1, a GPU driver crash reloads the driver instead of freezing the display. - Stable KABI (§3) means hardware vendors can ship binary drivers that work across kernel updates. No more "update your kernel, lose your NVIDIA driver." - Process hibernation (§8) avoids OOM killing your browser. Under memory pressure, background tabs hibernate to swap instead of being killed.

Home lab / small office: - The same kernel runs on x86 NAS boxes, ARM SBCs (Raspberry Pi class), and RISC-V boards. Eight-architecture support (§intro) means one knowledge base for heterogeneous hardware. - Live evolution (§2) means the home server that runs backups, media, and DNS doesn't need scheduled reboots for kernel updates. - FMA (§16) handles disk degradation automatically — EDAC retires failing pages, the kernel quarantines a flaky USB drive, and you get a notification instead of discovering corruption later.

Mobile / Linux phones / tablets (non-Android): - UmkaOS targets glibc/systemd, not Bionic/Android. But Linux phones (PinePhone, Librem 5, postmarketOS) and non-Android ARM devices (industrial panels, in-vehicle infotainment, kiosks) are natural targets. - Process hibernation (§8) replaces aggressive OOM killing with pause-and-resume. When a phone runs low on memory, background apps hibernate to swap instead of being killed — the user switches back and the app is still there, not restarting from scratch. - Power budgeting (§6) manages battery life at the kernel level. Per-cgroup watt budgets let the system give the foreground app a power budget while throttling background services. EAS places tasks on the most efficient cores (big.LITTLE, hybrid), extending battery life without userspace heuristics.

Power-constrained deployments (cross-cutting — mobile, laptop, edge, data center): - Per-cgroup watt budgets (§6) are valuable across the board, not just data center. On a laptop: limit a compile job to 15W so the fan doesn't spin up. On an edge device: stay within a 5W solar/battery budget. In a data center: rack-level power capping — the kernel reports and enforces per-container power consumption via RAPL/SCMI telemetry. - Intent-based power hints let applications declare upcoming phases ("compute burst" or "idle wait"), so the kernel pre-adjusts voltage and frequency instead of reacting after the fact. - EAS (§6) on heterogeneous SoCs (ARM big.LITTLE, Intel hybrid P+E cores) places tasks on the most energy-efficient core for their current workload phase.

Embedded / IoT (ARMv7, RISC-V): - UmkaOS runs on ARMv7 (Cortex-A class) and RISC-V 64 with full driver isolation on ARMv7 (DACR provides Tier 1 at ~30-40 cycles). RISC-V lacks fast isolation but Tier 2 still works for untrusted drivers. - The capability model (§7) provides finer-grained access control than Linux's flat capability bitmask — useful for IoT devices that expose specific interfaces to specific services. - PQC (§7) protects firmware update signing against harvest-now-decrypt-later attacks on devices expected to be deployed for 10+ years.

Android phones — not blocked by architecture, blocked by engineering effort: - Android runs on Linux. Bionic libc makes standard Linux syscalls — it would work on UmkaOS. The kernel architecture supports everything Android needs (cgroups, namespaces, SELinux, DMA-BUF, memfd). - The gap is drivers: Binder (Android's IPC — ~15K lines, would be a KABI driver), and vendor SoC modules (GPU, modem, camera ISP — typically 100+ modules per SoC, currently shipped as Linux `.ko` files that would need KABI ports). - This is engineering work, not an architectural limitation. If someone did it, the benefits would be significant: driver crash isolation (GPU bugs don't reboot the phone), process hibernation (better than Android's kill-and-restart), live kernel updates (security patches without reboot), and most importantly — **stable KABI would solve Android's biggest kernel problem**: vendor drivers that break on every kernel version bump, which is why phones stop getting updates after 3-4 years.

What UmkaOS is NOT good for (today): - Real-time control systems requiring hard sub-microsecond guarantees. UmkaOS has CBS and latency-nice but not a formally verified real-time scheduler. PREEMPT_RT equivalent is Phase 4+. - Systems with only 2-4 MB of RAM. UmkaOS targets server/desktop/embedded-with-MMU, not microcontroller class.

Summary: What UmkaOS Adds Over Linux

Feature	Linux	UmkaOS
Implementation language	C (Rust modules experimental)	Entire kernel in Rust — no UB, compile-time data race prevention
Lock ordering	Runtime lockdep (detects when exercised)	Compile-time type-level enforcement (prevents at build time)
Driver crash isolation	All Ring 0 or all Ring 3	Tier 1: Ring 0 with memory domain isolation, <1% overhead
Multikernel peer model	Host-side megadrivers per device	Tier M: ~2K lines generic transport, same protocol as cluster
Live subsystem replacement	kpatch (function-level)	Full subsystem swap with state migration, <10 µs pause
State-spill prevention	None	3-layer enforcement: link-time, compile-time, SDK
Stable driver ABI	"We don't guarantee it"	5-release versioned vtable, C+Rust from single IDL
Distributed shared memory	Userspace only	Kernel-native MOESI/RDMA, ~2-5 µs remote page fault
Distributed futex	None	futex_wait on DSM pages works transparently across nodes
OOM handling	Kill process	Hibernate-and-resume (OOM kill as last resort)
KVM VMM isolation	Process boundary + seccomp	Capability-scoped Tier 2 (same perf, stronger containment)
eBPF isolation	Runs in Ring 0 (trusts verifier)	Dedicated MPK domain (defense-in-depth if verifier has bugs)

Feature	Linux	UmkaOS
Multi-object wait	futex_waitv (futexes only)	fd + event + pid + timer + semaphore in one call
Wine acceleration	Userspace wineserver IPC	Kernel-native NT primitives, 10-16x speedup
Fault remediation	Separate EDAC/MCE/recovery	Unified FMA sense-think-act loop
Kernel-native ML tuning	Manual sysctl	Closed-loop, safety-clamped, per-cgroup ML policy
PQC	Userspace only	Boot chain through cluster communication
Dropped compatibility	Everything, including /dev/mem	No .ko modules, no /dev/mem, no kexec, no ia32 compat mode
Counter longevity	Mixed u32/u64	u64 everywhere + documented wrap analysis
Architecture support	x86-64 + ARM primary	First-class x86, ARM, ARMv7, RISC-V, PPC, s390x, LoongArch