BEAM.SMP: From Single-Core Isolation to Mandatory Symmetric Multiprocessing

Namdak Tonpa Norbu Ketaka

August 11, 2026

Abstract

Erlang/OTP was designed from the outset for massive concurrency through lightweight processes and message passing. For more than a decade the runtime system existed in two parallel forms: a single-core (non-SMP) emulator and an SMP emulator. In OTP 21 the non-SMP runtime was permanently removed. This article examines the two architectures, the engineering and economic motivations for the transition, and the concrete consequences for core subsystems including the cryptographic application, ASN.1, drivers, and native-code interfaces. A dedicated section analyses how these architectural choices are realised in the synrc/hv project: the frozen single-core OTP 20.3.8.26 release versus the SMP-oriented main branch, both targeting the seL4 microkernel with a minimal Synrc syscall provider and an Alpine Linux driver domain. The internal structure of the Synrc BEAM host is examined file-by-file, showing how the single-core and SMP lines share a common codebase while admitting different degrees of concurrency.

Contents

1 Introduction

Erlang's process model is fundamentally sequential from the point of view of a single process: a process executes a sequence of reductions until it is suspended. Early implementations mapped this model onto a single operating-system thread. With the arrival of multi-core commodity hardware the OTP team introduced an SMP runtime (first stable in OTP R11B, 2006) that retained the same language-level semantics while allowing multiple schedulers to run Erlang processes in parallel.

For twelve years both runtimes were maintained. In April 2017 the OTP team announced the deprecation of the non-SMP emulators; they were removed entirely in OTP 21 (2018). From that release onward every supported Erlang system is an SMP system, even when started with a single scheduler (`+S 1`).

The synrc/hv project embodies both ends of this history. It ships a production-usable single-core BEAM based on the last OTP that still permitted a true non-SMP build, and it simultaneously develops an SMP path on the main branch. This dual-track approach makes the abstract architectural transition concrete.

2 Architectural Comparison

2.1 Single-core (non-SMP) architecture (OTP $\leq$ 20)

- One scheduler thread executes all Erlang processes.
- No operating-system threads are required for the core emulator (`--disable-threads` was legal).
- Global data structures (atom table, process table, ETS meta-data, binary heap) need only single-threaded mutual exclusion or none at all.
- Garbage collection is strictly sequential.
- Drivers and NIFs run to completion on the single scheduler; long-running native code blocks the entire virtual machine.
- The resulting binary can be linked against a minimal C library (newlib) and run as a single thread of control on a bare-metal or capability-based kernel such as seL4.

2.2 Symmetric Multiprocessing architecture (OTP 21+)

1. N normal schedulers (default = number of logical cores).
2. Separate dirty CPU and dirty I/O scheduler pools for long-running or blocking native code.
3. Auxiliary OS threads for timers, signals, asynchronous drivers and system management.

4. Lock-free and fine-grained locking protocols for run queues, process migration, message passing and memory allocators.

5. Concurrent garbage collection of process heaps and the shared binary heap.

6. Mandatory thread support at build time; the configure options that produced a pure single-threaded emulator were deleted.

The language-level semantics remain identical: processes do not share mutable data and communicate only by message passing. The difference lies entirely in the runtime realisation of that model.

3 Motivation for the Transition and Removal

Three interlocking reasons drove the decision.

3.1 Maintenance cost

Every new feature—dirty schedulers, true asynchronous signalling, improved ETS concurrency, the later JIT—had to be implemented and tested twice. The non-SMP path became a permanent tax on development velocity.

3.2 Dirty schedulers

Long-running NIFs had always been a latent reliability problem. Dirty schedulers solve it, but only in an SMP setting. Implementing an equivalent mechanism for the non-SMP emulator was judged disproportionately expensive.

3.3 Hardware reality

By 2017 single-core performance had stagnated while core counts continued to rise. The performance advantage once claimed for the non-SMP emulator on certain workloads had largely disappeared, and the operational simplicity of a single binary that automatically exploits available cores became more valuable than the residual single-threaded optimisations.

Consequently the non-SMP emulators were deprecated in OTP 20 and removed in OTP 21. The only remaining way to obtain single-scheduler behaviour is the run-time flag `+S 1`, which still executes inside the SMP codebase and retains the auxiliary threads required by that codebase.

4 Consequences for Core Subsystems

4.1 Cryptographic application (`crypto`)

OTP 20 linked against OpenSSL (or LibreSSL) through a set of NIFs that assumed a single-threaded caller. In the SMP world the same NIFs must be re-entrant or protected by locks; OpenSSL itself must be initialised for multi-threaded use. Static linking of a carefully configured LibreSSL remains feasible on both architectures, but the bare-metal single-core build can use a simpler, non-threaded LibreSSL configuration, while the SMP build must guarantee thread safety and supply entropy from multiple schedulers.

4.2 ASN.1 compiler and runtime

The ASN.1 application is largely unaffected at the language level. The generated Erlang code is pure and therefore safe on both runtimes. The difference appears in the supporting C code used by the PER/BER codecs when linked as NIFs or linked-in drivers: any global state must be made thread-local or protected under SMP.

4.3 Drivers and the I/O subsystem

Classic drivers ran exclusively on the single scheduler. Under SMP they execute on a scheduler thread and may be migrated; the driver API therefore gained explicit thread-safety obligations. The older asynchronous-thread pool (`+A`) was largely superseded by dirty I/O schedulers.

4.4 Native Implemented Functions (NIFs)

The NIF API acquired the ability to schedule work on dirty schedulers. A NIF written for OTP 20 may still run on a modern system, but a long-running NIF will block a scheduler and degrade latency. The recommended style therefore changed from “finish quickly or accept the consequences” to “off-load to a dirty scheduler.”

4.5 Memory management and atomics

The single-core runtime could use ordinary C loads and stores for many internal structures. The SMP runtime relies on C11 atomics (or compiler built-ins) and carefully designed lock-free algorithms. This is the principal reason a pure newlib/bare-metal port becomes substantially harder after OTP 20.

5 synrc/hv: Two Concrete Realisations

The synrc/hv project makes the abstract transition tangible by maintaining two parallel lines of work on the seL4 microkernel.

5.1 Single-core line — release `single-core-otp-20.3.8.26`

This release freezes the last OTP that still permitted a true non-SMP build:

- OTP 20.3.8.26 configured with `--disable-smp-support --disable-threads`.
- Patches for bare-metal operation (`ttsl_drv`, system drivers).
- Static linkage against newlib (or a minimal musl subset) via the `aarch64-none-elf` toolchain.
- Working interactive shell, kernel and stdlib; no ncurses, no Rust, no LING-derived code.
- Single protection domain hosting the entire BEAM; one seL4 thread of control.
- Cryptography supplied by a statically linked LibreSSL configured for a single-threaded environment.
- Device access mediated through a separate Alpine Linux guest (via sDDF) so that the BEAM domain itself remains minimal.

The resulting artefact is a small, auditable, single-threaded BEAM that can be reasoned about as one sequential machine running inside a formally verified isolation boundary.

5.2 SMP line — main branch

The main branch targets a modern OTP (currently aligned with recent releases) and accepts the full SMP runtime:

- Multiple normal schedulers (intended to map onto the four cores of the target platform).
- A constrained set of auxiliary OS threads (dirty schedulers, timer, etc.), kept deliberately small (the project has experimented with a fixed budget of approximately six OS threads).
- Requirement for C11 atomics, memory barriers and seL4 notifications to implement cross-core wake-ups and run-queue operations.
- Richer Synrc syscall surface and more elaborate mapping of BEAM threads onto seL4 TCBs and notification objects.
- Same hybrid architecture (Synrc BEAM domain + Alpine driver domain) but now with concurrency inside the BEAM domain itself.

5.3 Comparative summary

Aspect	Single-core (OTP 20.3.8.26)	SMP (OTP 29.0.5)
OTP base	20.3.8.26 (frozen)	recent / 29-era
Schedulers	1	N (typically 4)
OS threads	1 (or none extra)	fixed small set (≈ 6)
Atomics / locks	unnecessary	mandatory
LibreSSL	single-threaded build	thread-safe build + entropy
seL4 mapping	one TCB	multiple TCBs + notifications
TCB size	minimal	larger but still bounded
Primary goal	high-assurance, auditable	multi-core performance

The two lines share the same overall system architecture (seL4 + Synrc + Alpine) and the same philosophy of keeping device drivers outside the BEAM's protection domain. They differ precisely in the degree of concurrency admitted inside the BEAM itself, thereby illustrating the cost and benefit of the transition that OTP made mandatory in version 21.

6 Internal Structure of the Synrc BEAM Host

The Synrc BEAM host is a thin, seL4-native environment that runs an (almost) unmodified Ericsson BEAM. Its source tree splits into four logical layers: entry and orchestration, the Synrc syscall/VFS ABI, the scheduler and worker infrastructure, and the classic BEAM emulator pieces. The same sources serve both the single-core and the SMP lines; compile-time and run-time configuration determine how much concurrency is admitted.

6.1 Entry and top-level orchestration

`synrc_main.c`
: Program entry point for the Synrc protection domain. Initialises the seL4 environment (CSpace, VSpace, notifications), sets up the minimal heap/`sbrk`, creates the primary BEAM scheduler thread (and any fixed worker threads), then hands control to the emulator. Contains the main event loop that waits on seL4 notifications (timer, IPC from other PDs, console, etc.).

`beam_aux.c`
: Auxiliary initialisation and glue: environment variables, command-line parsing, early memory setup, registration of BIFs/NIFs that need special Synrc treatment, and platform-specific boot logic.

6.2 Syscall trap and virtual file system (the Synrc ABI)

`syscall_trap.h` / `syscall_trap.c`
: The heart of the Synrc host. Implements the small set of Linux/musl

syscalls that a statically-linked OTP actually needs: `write`, `read`, `open/close` (limited), `mmap/munmap`, `brk/sbrk`, `clock_gettime`, `getpid`, `exit`, etc. On a trap the code either services the call locally or forwards it via seL4 IPC / shared-memory rings to the Alpine driver domain (sDDF). This layer replaces a full libc + kernel.

`vfs.h / vfs.c`
: Minimal in-memory / capability-based VFS. Provides the file-descriptor table and the handful of file-like objects the BEAM expects (console, null, simple tmpfs or sDDF device nodes). Keeps the BEAM from needing a real POSIX file system inside its own protection domain.

`worker_syscall.h`
: Declarations for the syscall-related entry points that worker threads may call. Allows a dirty-scheduler or auxiliary worker to perform a restricted set of Synrc syscalls without going through the full trap path.

6.3 Scheduler, run-queues and worker threads

These files implement the multi-core / multi-thread story that the modern BEAM expects.

`beam_sched.c`
: Core scheduler logic for the Synrc host. Owns the mapping between BEAM schedulers and seL4 threads/TCBs, the reduction-counting / timeslice logic, and the interaction with seL4 notifications for cross-core wake-ups. In the single-core OTP-20 build this file is essentially a thin wrapper; on the SMP main branch it becomes the central load-balancer.

`beam_rq_super.c`
: "Super" run-queue management. Handles the global or hierarchical run-queues, work-stealing, and migration of Erlang processes between the normal schedulers.

`beam_poll.c`
: I/O and timer polling. Integrates seL4 notifications / sDDF rings with the BEAM's `select/poll`-style waiting. Used by both the main scheduler and dirty I/O paths.

`beam_dirty_cpu.c`
: Implementation of a dirty CPU scheduler. Long-running CPU-bound NIFs are off-loaded here so they do not block a normal scheduler. Mapped to one (or a few) dedicated seL4 threads.

`beam_dirty_io.c`
: Implementation of a dirty I/O scheduler. Blocking or long I/O operations run here. Again mapped to a small number of seL4 threads.

`beam_worker6.c ... beam_worker15.c`
: Concrete worker-thread entry points. The numbering indicates a fixed pool (workers 6–15) that can be assigned roles: normal scheduler, dirty CPU, dirty I/O, async, timer, etc. Each file is typically a thin `void *workerN(void *arg)` that calls into the common run-loop defined in `worker_run.h`. Keeping them as separate compilation units makes it easy to control exactly how many OS threads are created and what each one is allowed to do—essential for a fixed thread budget.

`worker_run.h`
: Common run-loop and state for all worker threads. Contains the shared data structures, the "what kind of worker am I?" dispatch, and the interaction with the central scheduler / run-queues.

6.4 Classic BEAM emulator pieces

`beam_emulator.h / beam_emulator.c`
: The main BEAM instruction interpreter / `process_main` loop. Lightly adapted so that it can be driven by the Synrc scheduler instead of the stock ERTS scheduler.

`beam_loader.h / beam_loader.c`
: BEAM file loader and code-server support. Responsible for loading `.beam` files (from the embedded rootfs or from the VFS) and installing the code into the runtime.

6.5 How the pieces fit together

```
synrc_main.c
|
+-- initialises seL4 + heap
+-- starts fixed worker threads (beam_worker*.c)
|   +--> worker_run.h  -->  beam_sched.c / beam_dirty_*.c
+-- installs syscall_trap (syscall_trap.c + vfs.c)
+-- enters beam_emulator.c  (process_main / scheduler loop)
    +--> beam_rq_super.c, beam_poll.c, beam_loader.c
```

- **Single-core OTP-20 build.** Only one real scheduler thread is active. Most of the `beam_worker*`, `beam_dirty_*` and `beam_rq_super` machinery is either compiled out or reduced to stubs. `syscall_trap` and `vfs` remain essential.
- **SMP main-branch build.** The full set of workers is started (subject to the fixed thread budget). `beam_sched` and `beam_rq_super` become live, dirty schedulers are used for NIFs, and cross-core wake-ups go through seL4 notifications.

6.6 Summary of file roles

File	Layer	Single-core	SMP
`synrc_main.c`	Boot	Entry point; creates the single scheduler thread and enters the emulator	Entry point; creates the fixed worker pool and the primary scheduler
`beam_aux.c`	Boot / glue	Early init, environment, one-time BIF/NIF registration	Same role, plus any SMP-specific early setup
`syscall_trap.c/.h`	Synrc ABI	Implements the minimal set of traps needed by a single-threaded BEAM	Same traps; now callable from every worker thread
`vfs.c/.h`	Synrc ABI	In-PD file-descriptor table and console/null nodes	Shared by all workers; remains single-writer or locked
`beam_sched.c`	Scheduler	Thin wrapper around the single run queue	Central load-balancer, reduction accounting, cross-core wake-ups
`beam_rq_super.c`	Scheduler	Stub / unused	Hierarchical/global run-queues and work-stealing logic
`beam_dirty_cpu.c`	Dirty CPU	Compiled out	Off-loads long-running CPU NIFs to a dedicated worker
`beam_dirty_io.c`	Dirty I/O	Compiled out	Off-loads blocking I/O and driver work to a dedicated worker
`beam_worker6.c` … `15.c`	OS threads	At most one worker (or none)	Concrete entry points for the fixed pool (sched, dirty, timer, …)
`worker_run.h`	Worker infra	Minimal / unused	Common run-loop and per-worker state shared by all `beam_worker*`
`worker_syscall.h`	Worker infra	Unused	Restricted syscall helpers callable from dirty/async workers
`beam_emulator.c/.h`	Classic BEAM	Interpreter and `process_main` driven by the single scheduler	Same interpreter, now driven by multiple schedulers
`beam_loader.c/.h`	Classic BEAM	Loads `.beam` files into the single address space	Same loader; code becomes visible to all schedulers
`beam_poll.c`	I/O / timer	Single-threaded poll of timers and seL4 notifications	Poll integrated with the notification set used by all workers

This layout keeps the trusted computing base of the BEAM protection domain small while still giving the modern SMP BEAM the threading and dirty-scheduler services it requires.

6.7 Implications for High-Assurance and Bare-Metal Deployments

Organisations that maintain a frozen OTP 20 single-core tree for seL4 or other minimal environments gain a dramatically smaller trusted computing base: one thread, no OS-thread synchronisation inside the emulator, and a cryptographic library that need not be thread-aware. The modern SMP tree, while mandatory for any system that must scale beyond a single core, imposes a richer set of requirements on the host (threads, notifications, atomics, multi-core memory consistency). The two architectures are therefore best treated as distinct products with a common language front-end rather than as two configurations of a single runtime.

synrc/hv demonstrates that both products can be realised on the same formally verified foundation, giving system designers a concrete choice between maximal simplicity and multi-core scalability. The shared source tree and the clear separation between the Synrc ABI layer and the scheduler/worker layer make that choice a matter of configuration rather than a complete rewrite.

7 Later Language and Runtime Features: Porting Difficulties

Freezing a runtime at OTP 20 (or any pre-21 baseline) buys a simple, single-threaded execution model, but it also freezes the language surface and the set of runtime services that application code may assume. Subsequent OTP releases have introduced features that are either unavailable on an older runtime or that impose new requirements on any attempt to port a modern BEAM to a minimal environment such as Synrc/seL4. This section surveys the most consequential of those features.

7.1 The `maybe` expression

The `maybe ... end` construct (EEP 49) was introduced as an experimental selectable feature in OTP 25, became enabled by default in OTP 27, and is scheduled to become permanent (no longer disableable) in OTP 29.

- It adds the keywords `maybe` and `else`.
- Any existing code that used the atom `maybe` (or `else` in certain contexts) without single quotes becomes a syntax error once the feature is active.
- The feature is implemented primarily in the compiler; the runtime must still recognise the new opcodes or AST forms that the compiler emits.

For a frozen OTP 20 tree the consequences are immediate:

- Application code written against OTP 25+ cannot be compiled by the OTP 20 compiler.
- Even if one back-ports only the compiler, the resulting BEAM files contain instructions or metadata that the OTP 20 loader and emulator do not understand.
- Conversely, a modern SMP Synrc build that tracks recent OTP must implement (or at least tolerate) the new feature flags, the updated loader, and the keyword-related changes in the parser.

The selectable-feature mechanism itself (EEP 60) adds further surface area: the runtime and the code server must honour per-module and command-line feature flags, increasing the complexity of any minimal loader.

7.2 JIT (Just-In-Time compiler)

OTP 24 introduced a JIT for x86_64; OTP 25 extended it to AArch64. The JIT replaces the classic threaded-code interpreter with native code generation at load time.

- A bare-metal or newlib-based port must either implement a comparable JIT (including the assembler back-end and the necessary memory-protection primitives) or retain an interpreter path.
- The JIT assumes a richer OS environment (writable+executable memory transitions, certain cache operations, signal handling for debugging aids).
- On seL4 the required permission dances and the interaction with the capability system are non-trivial; many minimal ports simply disable the JIT and stay with the interpreter.

Thus the single-core OTP 20 line naturally stays with the classic emulator, while an SMP mainline that wishes to track upstream performance must confront the JIT's platform requirements.

7.3 Atomics, persistent terms and modern memory primitives

- The `atomics` module (OTP 21.2) and the underlying native atomic operations became part of the expected runtime toolbox.
- Persistent terms, improved binary reference counting, and more aggressive use of C11 atomics inside ERTS itself all assume a memory model and a set of hardware primitives that a pure newlib/single-threaded build never needed.
- Any SMP Synrc port must supply working C11 atomics (or a faithful emulation) and must correctly implement the memory-ordering constraints that the BEAM's lock-free algorithms rely on.

7.4 Time model and timers

OTP 18 already introduced the modern time API and time-warp modes; later releases made multi-time-warp the default and tightened the requirements on monotonic clocks and timer implementation. A minimal seL4 host must provide a sufficiently accurate monotonic clock source and must service the BEAM's timer wheel without the luxury of a full POSIX timer subsystem. The single-core OTP 20 code base has far fewer expectations in this area than a contemporary OTP 27+ runtime.

7.5 Dirty schedulers and the NIF interface

Although dirty schedulers appeared before OTP 21, their use became pervasive only after the non-SMP emulator disappeared. Modern NIFs are routinely written with `enif_schedule_nif` and the dirty-CPU / dirty-I/O calling conventions. A port that lacks dirty-scheduler support must either:

- reject such NIFs at load time, or
- provide a faithful implementation (exactly what the `beam_dirty_cpu.c` / `beam_dirty_io.c` and worker-thread files in Synrc exist to do).

7.6 Distribution, term format and external interfaces

Later OTP releases made Unicode atoms the default, evolved the external term format, and tightened distribution-protocol capabilities. While a single-node embedded system can often ignore distribution, any residual use of `term_to_binary`/`binary_to_term`, ETS transfer, or port communication can be affected by format changes. Keeping a frozen OTP 20 term-format implementation is simple; tracking upstream requires continuous updates to the encode/decode paths.

7.7 Consequences for the two Synrc lines

Feature / change	Single-core OTP 20.3.8.26	SMP OTP 29.0.5
`maybe` expression	Unavailable; code must be written to the older language surface	Must be supported (compiler + loader) once the tracked OTP enables it by default
JIT	Not present; classic interpreter only	Optional but desirable; requires executable-memory and cache management on seL4
C11 atomics / `atomics`	Unnecessary	Mandatory for correct SMP operation
Dirty schedulers	Disabled / stubbed	Implemented via the fixed worker pool
Modern time API	Minimal subset sufficient	Full monotonic + time-warp support expected
New opcodes / beam format	Frozen at OTP 20	Must track the OTP version being followed

In short, every language or runtime feature added after OTP 20 widens the gap between a deliberately frozen single-core artefact and a mainline SMP port that tries to stay close to upstream. The `maybe` keyword is merely the most visible recent example of a recurring pattern: convenience and expressiveness for application writers are purchased at the cost of additional surface area that a minimal, high-assurance host must either implement or permanently forgo.

7.8 ASN.1 and `public_key` representation changes

Later OTP releases, culminating in the large rework that shipped with OTP 28, altered the Erlang terms produced and expected by the ASN.1 compiler and by the `public_key` application. Two changes are especially visible to application code that manipulates X.509 names, certificates or other CHOICE-heavy structures.

1. **Modernised ASN.1 modules in `public_key`.** The ancient hand-maintained ASN.1 specifications used by `public_key` were replaced by more contemporary modules. While the documented high-level API was kept as stable as possible, the concrete Erlang terms that appear for many CHOICE and OPEN TYPE positions changed.

2. **Explicit {`:correct, Val`} wrappers for CHOICE values.** In OTP 28 a successful decode of certain CHOICE types (most noticeably the DirectoryString / attribute-value choices that appear inside an `RDNSequence`) is often returned as `:correct, :uTF8String, <<"...">>` or the analogous form for `printableString`, `utf8String`, etc. Earlier releases emitted the bare tuple `:uTF8String, <<"...">>` (or the older "otp" format that sometimes wrapped the value in an extra list).

Consequently any code that pattern-matches on the older shapes must be extended. A typical defensive encoder now contains clauses for all of the forms that may appear across the supported OTP range:

In OTP 28 the ASN.1 decoder started wrapping certain `CHOICE` values (especially the string types that appear inside an RDNSequence) in an extra `:correct, ...` tuple. Earlier releases emitted the bare `:uTF8String, ...` or `:printableString, ...` form, and some versions also placed the value inside an extra list. Consequently any encoder that walks an RDNSequence must recognise all of these historic and current shapes—both the new `:correct, ...` wrappers and the older plain or list-wrapped tuples—before it can extract the actual character data and turn it into the proper ASN.1 string encoding.

Additional related incompatibilities observed with the OTP 28 ASN.1/`public_key` rework include:

- Changes in the representation of `OPEN TYPE` and some `AlgorithmIdentifier` parameters (e.g. `'NULL'` becoming {`asn1_OPENTYPE, <<5,0>>`}).
- A narrower set of type names accepted by the low-level `der_encode/2` / `der_decode/2` entry points; some previously usable constructors are no longer exposed.
- Subtle differences in the handling of default values and extensions that can affect round-tripping of certificates and CRLs.

For a frozen OTP 20 single-core tree these changes are invisible: the application simply continues to see the old term shapes. For any Synrc (or other) port that tracks a modern OTP the ASN.1 layer becomes another moving target—exactly the same class of problem already illustrated by the `maybe` keyword, the JIT and the atomics requirements. The practical consequence is that high-assurance or long-lived embedded runtimes must either

- stay on a deliberately frozen OTP (and accept that they cannot run newer application code that depends on the new ASN.1 shapes), or
- continuously maintain a compatibility veneer that normalises all of the historic and current term formats, as the example above does.

8 Conclusion

The removal of the non-SMP emulator in OTP 21 marked the end of an era in which Erlang could be regarded as an essentially sequential abstract machine that happened to support concurrency. From that release onward the runtime is a genuine multi-core virtual machine whose sequential appearance is an emergent property of the language, not of the implementation. The transition was driven by maintenance economics, the necessity of dirty schedulers, and the realities of contemporary hardware. Subsystems such as `crypto`, ASN.1, drivers and NIFs adapted with varying degrees of visible change; the deepest impact is felt in the memory model and the threading assumptions that any new port must satisfy.

The synrc/hv project preserves both sides of this history: a frozen, single-core OTP 20.3.8.26 release that maximises auditability and minimises the TCB, and an SMP-oriented main branch that accepts the full complexity of the modern runtime in exchange for multi-core performance. The internal structure of the Synrc BEAM host—with its explicit separation of syscall ABI, scheduler, dirty workers and classic emulator—shows how a single codebase can serve both goals. Together they demonstrate that the architectural decision made by the OTP team in 2017–2018 remains a live engineering choice when the target platform is a high-assurance microkernel rather than a general-purpose operating system.

References

[1] L. Larsson. Deprecation and removal of the non-smp emulator. Erlang Questions mailing list, April 2017.

[2] Erlang/OTP 21.0 Release Notes, OTP-14518.

[3] K. Lundin et al. Historical accounts of SMP development. Erlang User Conference presentations, 2008–2010.

[4] Official OTP documentation on schedulers, dirty schedulers and the NIF API (OTP 24–29).

[5] synrc/hv. Type-1 Hypervisor for ARMv8. `https://github.com/synrc/hv`, release `single-core-otp-20.3.8.26`.