# Turning Linux into a High-Performance Library OS with Flux

Kaifu Tian
Tsinghua University

Youjie Zheng
Tsinghua University

Yiren Zhang
Tsinghua University

Yuyang You
Tsinghua University

Keyang Hu
Tsinghua University

Kang Chen
Peking University
Tsinghua University

Yu Chen
Tsinghua University

## Abstract

Modern high-performance applications increasingly bypass the host kernel for I/O and scheduling and rely on library OSes for operating-system services. High-performance library OSes are often built from scratch, but supporting unmodified applications requires reimplementing mature Linux semantics. Running Linux itself as a library OS avoids this cost, yet existing Linux-based designs remain slow. Our analysis of LKL shows that the main bottlenecks lie in the surrounding user-space runtime rather than Linux subsystems.

We present Flux, a Linux-based library OS that reduces the runtime bottlenecks while keeping Linux subsystems largely intact. Flux redesigns scheduling, user-space interrupt delivery, and syscall paths, and introduces a new process abstraction that supports Linux `fork` semantics without giving up direct-call syscalls. Flux improves Nginx throughput by up to 18.5× over Linux, and approaches Junction's Memcached throughput under a 100$\mu$s p99 latency target, showing that Linux can provide a high-performance library OS without rebuilding Linux functionality from scratch.

***CCS Concepts:*** • **Software and its engineering → Operating systems**; *Process management*; *Scheduling*.

***Keywords:*** Operating systems, Scheduling, Process management

**ACM Reference Format:**
Kaifu Tian, Youjie Zheng, Yiren Zhang, Yuyang You, Keyang Hu, Kang Chen, and Yu Chen. 2026. Turning Linux into a High-Performance Library OS with Flux. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3830418.3843881



*SOSP '26, Prague, Czech Republic*

ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843881

## 1 Introduction

Modern high-performance applications increasingly bypass the underlying operating systems. Datacenter services such as in-memory transactional databases [15, 36, 55], key-value stores [17, 35, 43, 56, 72], and analytics engines [2, 3] rely on DPDK [19] and RDMA [51] for networking, SPDK [13] for storage, and custom schedulers [14, 21, 62, 66] to achieve microsecond-scale tail latency and high throughput. By moving I/O datapaths and scheduling decisions into guest kernels [6, 31, 34, 66] or user space [14, 21, 33, 62, 65], these systems avoid kernel traps, steer work to specific cores, and integrate tightly with modern devices.

Previous kernel-bypass systems are often built from scratch and run with applications as library OSes. For example, Demikernel [84] allows applications to leverage different kernel-bypass devices through a unified I/O abstraction. Caladan [21] implements a high-performance TCP/IP stack and a cooperative scheduling framework. Building from scratch allows these systems to provide a lightweight compatibility layer that minimizes application syscall and datapath overhead, making full use of high-performance kernel-bypass frameworks. To support more unmodified applications, these systems generally need to adapt existing Linux features such as syscall handling, thread scheduling, process management, and file systems. Junction [20], built on Caladan, implements over 170 Linux syscalls and runs many unmodified cloud applications, demonstrating the feasibility of workload-specific Linux-compatible layers.

However, extending these systems to fully support more applications is expensive because it requires reimplementing and validating mature Linux semantics [8, 40, 80]. For example, Nginx [61] uses `fork` to create workers and Redis [72] uses it for background persistence. Extending Junction's current single-address-space model to support multi-process management would require substantial engineering effort.

Thus, a more natural approach is to run Linux itself as a library OS in user space and directly reuse Linux's mature codebase (e.g., syscalls and schedulers) as a compatibility layer. For example, Linux Kernel Library (LKL) [67] and its derivatives [52, 77, 78] are linked with applications as libraries, allowing them to execute Linux syscalls through function calls. In principle, this direction offers a well-tested

kernel-bypass system without reimplementing Linux interfaces from scratch. In practice, however, existing implementations still fall short on performance. LKL lacks multicore scalability, and rkt-io [78], while integrating kernel-bypass frameworks and supporting multicore scheduling, still incurs overhead from multi-level scheduling with lthreads [1] and internal LKL threads. The key question is whether this approach is inherently slow or can achieve both high performance and close-to-Linux compatibility.

We present Flux (Fast Linux User eXecution)[1], which turns Linux into a high-performance library OS. Flux shares an address space with applications, allowing unmodified applications to access Linux syscall interfaces through function calls. Flux reuses Linux subsystems largely unchanged, such as schedulers and file systems. Flux integrates kernel-bypass frameworks such as DPDK and SPDK to leverage the microsecond-scale performance of modern devices. For network I/O, Flux further reduces overhead in the critical path through a fast TCP path. To achieve both high performance and close-to-Linux compatibility, Flux needs to address two major challenges.

The first challenge is how to make Linux scheduling better suited to a user-space runtime to improve performance. The Linux scheduler relies on timer interrupts and inter-core interrupts (IPIs) to support preemption and wakeup mechanisms, but when Linux runs in user space, these mechanisms cannot directly rely on traditional hardware and kernel execution paths. Existing Linux-based library OSes typically simulate these mechanisms through host synchronization interfaces, polling threads, or additional user-space schedulers, resulting in multi-level scheduling, delayed event delivery, and extra context switches.

To efficiently run Linux in user space, Flux adopts a single-level scheduling model. Flux makes scheduling decisions on its own and does not rely on host schedulers. Flux uses Intel User Interrupts (UINTR) [30] to deliver timer and inter-core interrupts (IPIs). Flux further optimizes syscall entry and exit paths for user-space execution.

The second challenge is to support multi-process semantics for compatibility. Multi-process support means that a library OS can internally create and schedule multiple processes that run atop the same library OS instance. Multi-process support is important because many widely deployed server applications depend on `fork`-based multi-process semantics [61, 72]. Existing systems either offer only restricted `vfork`-based multi-process support or keep lightweight processes in a shared address space [20, 44, 64].

Flux introduces a multi-process abstraction that decouples host address space management from host thread scheduling for a library OS. Flux supports page-table-isolated multi-process management and copy-on-write (COW) `fork`. Relying on the host to perform privileged operations, Flux shares its state across multiple processes, while all decisions regarding multi-process management and address space switching are made by Flux. This approach allows Flux to maintain the performance benefits of single-level scheduling and function calls while providing rich multi-process compatibility.

Flux has two limitations. First, it targets Intel x86 platforms with UINTR. Without UINTR, it falls back to host-mediated timer and interrupt delivery, incurring additional host transitions and higher wakeup latency. Second, device interrupts and page faults still rely on the host. Flux forwards device interrupts through `eventfd`, incurring additional host transitions. It injects page faults through host signals, which adds substantial overhead to workloads that frequently fault on `mmap`-backed files (§7).

We evaluate Flux on microbenchmarks and real applications. For compatibility, Flux achieves a Linux Test Project (LTP) pass rate of 92.2%, which is close to native Linux and much higher than other systems (§6.2). For storage, Flux improves ext4 Filebench throughput by up to 47% over Linux (§6.3). For latency-sensitive network workloads, we use a service-level objective (SLO) requiring the 99th-percentile (p99) latency to remain below 100$\mu$s. Flux achieves throughput within 4% of Junction's on Memcached while satisfying this p99 latency target, and improves Nginx throughput by up to 18.5× over Linux while supporting normal multi-worker execution (§6.4, §6.5). For hardware preemption, Flux achieves microsecond-scale wakeup latency (§6.6). For microbenchmarks, Flux outperforms Linux by 3× on `hackbench` (§6.8). Overall, our results show that Linux running in user space is not inherently slow; Linux as a library OS can achieve performance close to state-of-the-art from-scratch systems without sacrificing Linux's mature functionality.

In summary, this paper makes the following contributions:

- We perform an in-depth analysis of LKL's performance and find that the performance bottleneck of Linux running in user space comes from the surrounding runtime, rather than Linux itself.
- We propose Flux as an efficient library OS through architectural design and modularized optimizations, while supporting Linux-style multi-process semantics through host-assisted address space management.
- Our results show that Flux achieves compatibility close to Linux and performance close to state-of-the-art kernel-bypass systems.

## 2 Background

### 2.1 Library OSes from Scratch

For compatibility, kernel-bypass systems usually choose to build library OSes from scratch [6, 21, 32–34, 44, 62, 65, 84]. This approach provides a lightweight compatibility layer for applications, allowing them to use the functionalities supported by the library OS with minimal modifications. For

[1]We have open-sourced Flux at https://github.com/THU-OSLAB/flux.

| Systems | Sched. Overhead | Multicore Sched. | Hardware Preempt.[1] | Multi-process | Syscalls/APIs[2] | LTP[3] |
|---|---|---|---|---|---|---|
| **LKL** [67] | High | ✗ | ✗ | ✗ | 303 | 9.4% |
| **rkt-io** [78] | High | ✓[4] | ✗ | ✓[5] | 273 | 14.6% |
| **Demikernel** [84] | Low | ✗ | ✗ | ✗ | 18 | N/A |
| **Junction** [20] | Low | ✓ | ✓ | ✓[5] | 172 | 10.0% |
| **Flux** | Low | ✓ | ✓ | ✓ | 345 | 92.2% |

**Table 1.** Comparison of library OSes. ✓: Supported. ✗: Not supported. [1] Preemptive scheduling with hardware interrupts. [2] Number of non-stub Linux syscalls or APIs in the evaluated versions. Demikernel exposes 18 custom PDPIX interfaces. [3] We use the Linux Test Project (LTP) pass rate as a measure of compatibility (see §6.2 for details). We modified the suite to start in a single-process environment, allowing systems with incomplete `fork` support to run tests. LTP targets the Linux syscall ABI and is therefore not applicable to Demikernel's custom interfaces. [4] rkt-io enables LKL symmetric multiprocessing (SMP) but depends on a single run queue. [5] rkt-io and Junction support restricted process creation based on `vfork`, but do not yet provide complete `fork`/`clone`/`exec`/`wait` semantics for unmodified LTP test cases.

example, Demikernel [84] unifies access to kernel-bypass devices through an I/O abstraction and coroutine scheduling. Caladan [21] implements a high-performance TCP/IP protocol stack and a cooperative scheduling framework. Skyloft [33] emphasizes the flexibility of scheduling policies, supporting customized schedulers for different workloads. These systems require applications to make minor modifications to use their specific APIs and scheduling mechanisms. To further support unmodified applications, Junction [20] builds upon Caladan and implements over 170 syscalls and Linux features such as signals and sockets, enabling the direct execution of numerous cloud application binaries, further demonstrating the feasibility of building a library OS for kernel bypass from scratch.

However, fully supporting a broader range of applications in these systems requires re-implementing and verifying many mature Linux semantics, incurring significant development costs. Tsai et al. [80] analyzed the Linux API dependencies of Ubuntu packages and found that 145 syscalls cover half of Ubuntu applications and 272 cover all. Moreover, applications depend not only on these syscalls but also on complex operation parameters (e.g., `fcntl`, `ioctl`) and system files in the virtual file system (VFS) (e.g., `/proc`, `/sys`). Loupe [40] further illustrates that applications trigger different OS functionalities under different testing scenarios. For example, Junction can run Nginx and Redis, but when Nginx enables multi-worker mode or Redis enables background persistence, Junction fails to run due to its single-address-space design, which has difficulty supporting multi-process interfaces and functionalities. Additionally, the development experience of Graphene [8] shows that language runtimes and support libraries rely on less frequently used but complex Linux semantics in interfaces like `futex`, while Junction currently implements a simplified version of `futex`. Therefore, enhancing the compatibility of current library OSes built from scratch requires substantial engineering effort. This prompts us to consider another design choice: **Turning Linux into a Library OS**.

### 2.2 Linux-based Library OSes

A complementary line of work pursues compatibility by running Linux itself as a library OS in user space. LKL and related systems aim to directly reuse Linux subsystems, including syscalls, scheduling, drivers, file systems, and network stacks, exposing standard Linux behaviors for applications to use. Through a host interface, LKL [67] runs as a library on Linux, Windows, and other operating systems. The LKL container [77] packages it as a lightweight container runtime. rkt-io [78] runs it in an enclave with DPDK, SPDK, and a user-space multicore scheduler.

In principle, reusing Linux code in a library OS can avoid redundant implementations of Linux functionalities. However, in practice, existing implementations still face significant performance overheads. Although LKL is linked with applications, allowing them to use Linux syscalls through function calls, LKL and related systems rely on the scheduling of the host operating system, which prevents them from fully leveraging the performance advantages of kernel bypass. We will analyze this issue in depth in §3.1.

### 2.3 A Library OS as a Guest Kernel

Some systems use hardware virtualization to run a library OS in guest kernel mode. This approach directly exposes privileged operations, including interrupt and exception handling, to the library OS and uses SR-IOV [63] for I/O virtualization. For example, Dune [5] exposes privileged CPU features to applications through VT-x [30], and subsequent systems use this model for high-performance networking, scheduling, and paging [6, 31, 34, 66, 83].

This paper focuses on running a library OS in user space for two reasons. First, **deployability**: virtualization-based approaches require hardware and platform support. Cloud providers generally do not expose hardware virtualization directly to tenants because it is reserved for provider-side use [28]. A user-space library OS, in contrast, can run as an ordinary process both on a conventional OS and inside a VM. Nested virtualization could support the virtualization-based

approach, but adds complexity and requires substantial engineering to achieve high performance. Second, **performance**: prior work reports that user interrupts outperform posted interrupts [31]. Virtualization also introduces two-level address translation and may require VM exits for privileged operations and address-space management. A VM exit is roughly an order of magnitude slower than a syscall [7]. Hardware mechanisms such as Intel VMFUNC [30] can accelerate address-space switching, but achieving both high performance and isolation still requires careful EPT- and CR3-level memory-layout and switching designs [39, 57].

# 3 Motivation

## 3.1 Analysis of LKL Performance

Linux-based library OSes inherit broad syscall support from Linux, yet existing systems still incur high runtime overhead. This raises a fundamental question: *is Linux as a Library OS inherently slow, or are current systems bottlenecked mainly by the runtime around Linux?* We answer this question by measuring LKL, a representative Linux-based library OS.

**Scheduling.** As shown in Figure 1, LKL adopts a single-core, non-preemptive scheduling model where all threads contend for one global CPU lock through host synchronization interfaces. Each thread created by the host application (referred to as a host thread) is paired with an LKL-managed thread (referred to as a stub thread). Stub threads exist only within LKL's scheduling context and are not visible to the host scheduler. LKL must ensure that the running context on the virtual LKL CPU always corresponds to the currently active stub thread. Otherwise, LKL performs an internal task switch to update the `current` pointer to the correct stub thread before entering the syscall path.

To illustrate the overhead introduced by this two-level scheduling model, we walk through an example involving two host threads in Figure 2. After host thread A completes an LKL syscall, the LKL `current` pointer is set to its associated stub thread (stub thread A). If host thread B subsequently invokes an LKL syscall, it first attempts to acquire the global CPU lock and then invokes an LKL-internal task switch, updating `current` to stub thread B. Finally, it executes the LKL syscall handler and releases the global CPU lock. On our platform, the whole process takes more than 2 $\mu$s.

Later work added SMP to LKL [52, 78]. rkt-io [78] uses an M:N model in which multiple core-pinned host threads share a global lock-free queue to cooperatively schedule lthreads [1]. Its scalability is limited by contention on the cache lines containing the queue head and tail, as well as by a global futex lock. Moreover, rkt-io still schedules LKL threads internally, as discussed in the overhead analysis.

**Interrupt Handling.** To emulate asynchronous events, LKL relies on host-provided timer interfaces (e.g., `timer_create`). The host invokes a callback that merely sets an internal pending bit, forcing LKL to actively poll this bit and handle the interrupt synchronously when execution yields or interrupts are re-enabled. This polling delays scheduling and degrades timer accuracy, with timer wakeup latencies exceeding 80$\mu$s (§6.8). For multicore scheduling, rkt-io further creates a dedicated IPI `lthread` for each LKL CPU. The `lthread` blocks on a semaphore until another CPU posts an IPI, then invokes the LKL interrupt path. This indirection adds a user-level wakeup and scheduling step before the target CPU handles the IPI. As we show in §6.6, rkt-io's cross-core wakeup tail latency exceeds 10$\mu$s.

## 3.2 Multi-process Support

**Why `fork` Matters.** Unix applications use `fork` [46] to create a child process with a logically independent copy of the parent's address space. Linux initially backs the two address spaces with the same physical pages and copies a page only when either process writes it, a mechanism known as copy-on-write (COW). Both processes then continue independently, and the child may either keep running the inherited program or replace it through `execve`. By contrast, `vfork` temporarily shares the parent's address space and suspends the parent until the child calls `execve` or exits, making it suitable only for the restricted `fork`-then-`exec` pattern [50].

This distinction affects widely deployed server applications. The Nginx master process creates workers that continue executing the inherited server image, while Redis uses COW `fork` for background persistence so that the parent can continue serving requests as the child writes a point-in-time snapshot [60, 71]. These workloads require the parent and child to execute concurrently and therefore cannot be supported by a `vfork`-only interface.

**Why `fork` Is Hard.** A Linux-based library OS such as LKL obtains low syscall overhead by colocating the application, the Linux kernel, and its runtime in one address space, allowing syscall handlers to be invoked as direct function calls. This model naturally supports threads that share an address space, but COW `fork` requires a distinct application address space for each child. Copying the entire address space duplicates Linux and runtime state that must remain shared and coherent, whereas keeping everything in one address space loses process-private memory and COW semantics. Kernel-bypass resources add another constraint: DMA-backed memory and device-facing state cannot be naively copied and often must remain shared across process creation.

**Existing Trade-offs.** Existing systems relax different requirements. Graphene creates a child by reconstructing its library OS and application state rather than using Linux-style page-table COW [8]. Junction provides a restricted `vfork`/`exec` path within a single application address space, which can launch a new image efficiently but cannot support cases in which the parent and child proceed independently [20]. Adjacent abstractions avoid independent address spaces

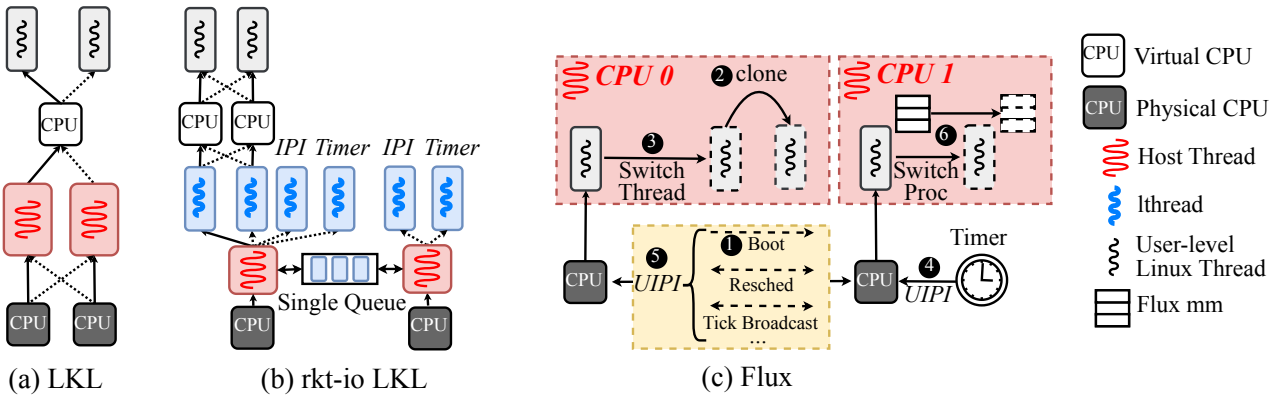


**Figure 1.** Comparison of scheduling models of Linux-based library OSes. (a) The original LKL also requires all application threads to contend for a single virtual CPU. (b) rkt-io enables LKL SMP but adds an `lthread` scheduling layer whose schedulers share a global run queue. (c) Flux uses a single-level scheduling model where Flux threads run directly on host threads. (§4.2)

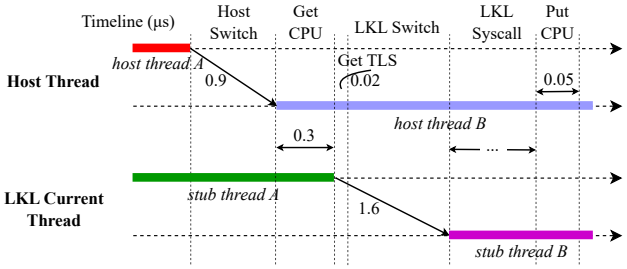


**Figure 2.** LKL syscall invocation path breakdown.

in different ways: Vessel keeps uProcesses in a shared address space for fast switching [44], while Pegasus transparently fuses multiple applications into a single process [64]. These designs use Memory Protection Keys (MPK) [30] for isolation, but the number of processes is bounded by the 16 hardware keys. Although they use key virtualization [24] to extend the number of processes, they cannot support Linux-style `fork` with independent address spaces.

### 3.3 Towards Fast Linux User eXecution: Challenges

Table 1 summarizes representative library OSes. They cover only subsets of efficient multicore scheduling, hardware preemption, multi-process support, and Linux compatibility. Our analysis identifies two key challenges to turning Linux into a high-performance library OS while supporting these capabilities:

- **Scheduling overhead**: Linux must block, wake up and preempt running tasks to maintain fairness and responsiveness across cores. Existing Linux-based library OSes layer Linux scheduling over host scheduling and rely on polling threads to deliver the timer interrupts and IPIs needed for preemption. The extra context switches and delayed event delivery increase runtime overhead. Flux addresses this challenge with single-level scheduling and hardware-assisted user interrupts (§4.2).
- **The lack of multi-process support**: Supporting Linux-style `fork` while retaining direct function calls into an in-address-space kernel requires privileged address-space operations unavailable to user space, such as switching page tables and handling page faults. Flux addresses this challenge with host-assisted address-space operations that preserve the shared state of the Flux runtime and the direct-call syscall path (§4.3).

## 4 Design

### 4.1 Overview

Flux turns Linux into a high-performance library OS, running applications without modifications. As shown in Figure 3, Flux keeps Linux subsystems largely intact and adds three surrounding components: kernel-bypass frameworks, an external runtime (the Flux runtime), and architecture-specific (`arch`) paths. The Flux runtime boots the system and bridges it to the host kernel. This split improves reuse across different platforms and eases rebasing across Linux releases. §5 describes the modularized implementation in detail. For privileged operations, Flux uses a lightweight kernel module to provide host-assisted mechanisms.

### 4.2 Single-Level Scheduling

Flux achieves single-level scheduling, where only the Linux scheduler inside Flux makes scheduling decisions and schedules Flux threads. As shown in Figure 1(c), each host thread is pinned to a physical core and serves as a bootstrap entry point. After boot, thread selection, wakeup, and context switching remain entirely inside Flux.

**Startup.** Flux takes over startup to enable multicore boot and single-level scheduling. The Flux runtime initializes required libraries (e.g., DPDK) and hardware resources, then launches the configured host threads and pins them to physical cores to minimize host-scheduler interference. Each thread invokes the primary or secondary startup entry of the Flux kernel and

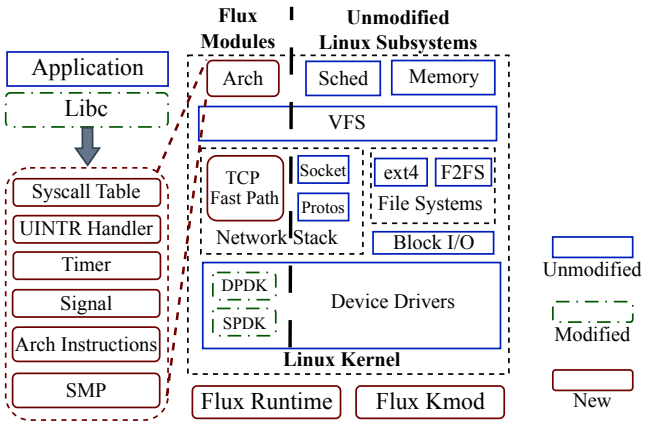


**Figure 3.** Flux overview. We mark SPDK and DPDK as modified because we implement drivers inside the Flux kernel to use their kernel-bypass interfaces (§5.3, §5.4).

enters the Linux SMP boot (❶). Flux does not need a virtual CPU abstraction because each host thread maps directly to one physical core. After SMP boot, the Flux kernel creates `kthreadd` and other internal kernel threads, then starts a dedicated Flux thread for the application's main entry point.

**Thread Creation.** Flux creates threads in the `arch`-defined `copy_thread` path (❷), which initializes the new context. A function entry point distinguishes kernel threads from application threads. For application threads, Flux copies general-purpose registers and prepares the stack state needed to restore execution (line 6). Kernel threads simply return to the specified entry function.

**Thread Switch.** Flux switches threads inside the Flux kernel (❸). `switch_to` implements the switch path. Listing 1 shows key steps: updating per-CPU state (line 11), conditionally saving extended processor state, including floating-point and vector registers (lines 13-15), and switching the thread-local-storage base with `WRFSBASE` to preserve the Linux application binary interface (line 23). At a synchronous function-call boundary, the x86-64 System V calling convention does not require the callee to preserve volatile floating-point and vector registers, so Flux does not save those registers again on the direct-call path. A user interrupt, however, may arrive at an arbitrary instruction while values in these registers are still live. If the interrupt triggers a thread switch, Flux saves the interrupted thread's extended processor state before scheduling the next thread. For multi-process execution, `switch_to` also switches address spaces when needed (§4.3).

**Thread Preemption and Wakeup.** Intel User Interrupts (UINTR) allow user-space programs to directly generate and handle interrupts without kernel-mediated delivery [30]. Flux uses UINTR for low-latency timers and IPIs. During initialization, the Flux runtime registers each host thread as a UINTR receiver and installs the interrupt handler. A thread

```
1  copy_thread(p, flags) {
2    if (flags & CLONE_VM)
3      p.proc_key = current.proc_key;
4    else
5      p.proc_key = flux_dup_mm(current.mm);
6    copy_thread_context(p, flags);
7  }
8
9  switch_to(prev, next) {
10   switch_stack_and_callee(prev, next);
11   cpu.current = next;
12   /* UINTR may leave xstate live. */
13   if (prev.from_uintr && !prev.xstate_loaded) {
14     save_xstate(prev);
15     prev.xstate_loaded = true;
16   }
17   /* Switch mm only for user threads. */
18   if (is_user(next)) {
19     if (cpu.proc_key != next.proc_key) {
20       flux_switch_mm(cpu.proc_key, next.proc_key);
21       cpu.proc_key = next.proc_key;
22     }
23     switch_tls_base(prev, next);
24   }
25 }
```

**Listing 1.** Pseudo-code of the clone and switch path in Flux. Red marks interfaces provided by the kernel module. `cpu` denotes per-CPU state.

running on a dedicated core periodically sends timer interrupts to target cores (❹). For better CPU efficiency, the same design could also map the local Advanced Programmable Interrupt Controller (APIC) into user space and program the timer directly [33].

During SMP boot, each host thread also becomes a sender to every other host thread. Flux assigns separate UINTR vectors to timers and Linux IPI types such as `CALL_FUNCTION` and `RESCHEDULE` (❺). On receipt, the Flux kernel decodes the vector and runs the corresponding handler, updating accounting state and reschedule flags as needed. Combined with single-level scheduling, this yields microsecond-scale preemption and cross-core wakeups.

### 4.3 Multi-Process Support

Flux stays in the application's address space, so syscalls remain direct function calls. Supporting `fork`, however, still requires a separate child address space with copy-on-write. Existing library OSes typically avoid this by reconstructing child state [8], offering only restricted `vfork`/`exec` [20], or keeping lightweight processes in a shared address space [44, 64]. Flux instead preserves Linux process semantics without giving up direct function calls. Flux introduces a process

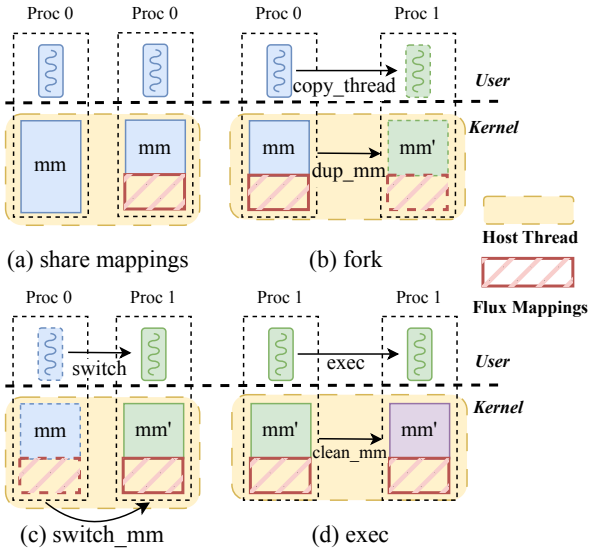


**Figure 4.** Flux multi-process support.

abstraction and uses a host kernel module for privileged address-space operations.

**Process Abstraction.** In traditional operating system designs, an address space can be shared among multiple threads or between user and kernel modes, yet distinct processes must maintain independent address spaces. Linux, for example, employs `task_struct` and `mm_struct` to govern thread and address space abstractions separately. Flux decouples this address space abstraction from host threads. Combined with its single-level scheduling architecture (§4.2), Flux allows a single host thread to seamlessly transition across multiple independent address spaces. Building on this abstraction, Flux supports the process lifecycle as shown in Figure 4:

**(a) Share mappings.** All writable Flux state, including the Flux runtime and heaps, kernel stacks, thread-local storage (TLS), and data segments, is shared across address spaces. Consequently, each address space managed by Flux is structurally partitioned into two parts: private application memory and shared Flux memory. This layout enables process switches through address-space switching.

**(b) Fork.** As shown in Listing 1, Flux uses an integer, `proc_key`, as the process identity shared with the host kernel. `copy_thread` either inherits `proc_key` for threads (line 3) or allocates a new host `mm_struct` for a real `fork` (line 5).

**(c) Switch (❻).** Flux uses dedicated kernel stacks for syscall and UINTR handling and shares these stacks across all address spaces. This design serves two purposes. First, context switches may occur on both paths. Because Flux switches kernel stacks before switching address spaces, the target stack must be accessible from the current address space, while the suspended stack must remain accessible when execution later switches back (line 10). Second, MPK isolation requires the Flux kernel to execute on a protected stack that application code cannot access (§4.4).

Stacks of kernel threads also stay in the shared mappings, so Flux switches address spaces only before switching to user threads. A per-CPU `proc_key` records the currently active address space, allowing Flux to compare it with the target before switching instead of consulting the previous thread's `proc_key` (lines 18-21).

**(d) Exec.** Flux preserves shared mappings of the Flux runtime, loads the new Executable and Linkable Format (ELF) binary into the current address space, and removes private mappings of the old application.

**Kernel Module.** The `flux_mm` kernel module is a lightweight host component that enables Flux's multi-process execution model. It provides a small `ioctl` surface for shared-mapping policy and host `mm_struct` lifecycle management. Flux retains control over process scheduling and lifecycle, while the module performs only the privileged operations needed to create, switch, and release page tables.

`flux_mm` establishes a common region for the Flux kernel and runtime in every process address space while keeping application memory private. It hooks into the startup of the Flux runtime and intercepts `mmap` and `brk` syscalls, converting writable private mappings (`MAP_PRIVATE`) of the Flux kernel and runtime into shared mappings (`MAP_SHARED`). A `fork` creates a new address space through `COPY_MM`, whereas an `execve` cleans the old mappings through `CLEAN_MM` and loads a new image.

`flux_mm` manages address spaces independently, allowing a pinned host thread to switch among multiple Flux processes. It maintains a hash table that maps integer `proc_keys` to reference-counted process structures (denoted as `proc`), thereby preventing user space from passing invalid pointers that could otherwise lead to kernel crashes. `proc` binds the lifetime of a Linux `mm_struct` to Flux's multi-process lifecycle. Linux itself uses two reference counts to manage the lifetime of an `mm`, while we focus on increments and decrements of `mm_users`, which tracks active use of the address space. `mm_users` can drop to zero only *after* the reference count of the corresponding `proc` has dropped to zero. By maintaining this reference count, operations on `proc` never encounter a use-after-free on the wrapped `mm`. Additionally, `flux_mm` interfaces are protected by the reference count of a global context, preventing attempts to exit and release all resources while it is still held in an intermediate state during creation or switching.

### 4.4 Isolation

**Threat Model.** Direct-call syscalls require the application and Flux to execute at the same host privilege level, with the Flux kernel and runtime mapped into each application address space. We trust the hardware, the host kernel and its Flux modules, and Flux itself, but treat application code and inputs as malicious. Our goal is to prevent the application

| | LOC | ioctl Interfaces |
|---|---|---|
| Shared Mappings | 265 | START_MAP_SHARED<br>END_MAP_SHARED<br>EXECVE |
| MM Management | 316 | COPY_MM<br>SWITCH_MM<br>RELEASE_MM<br>CLEAN_MM |

**Table 2.** Key functionalities and interfaces of the `flux_mm` kernel module.

from reading or corrupting Flux state or hijacking Flux control flow; otherwise, it could abuse Flux's host interfaces and direct device access to affect the host, other processes, or device data. Compromised hardware or host kernels, denial-of-service attacks, and user-interrupt side channels [68] are beyond the scope of this paper.

**Isolation Boundaries.** Flux separates process isolation from protection of its shared state. Independent host page tables isolate the private memory of different Flux processes, whereas Intel MPK protects the shared Flux kernel and runtime from the application within each address space. MPK associates pages with protection keys and uses the per-thread PKRU register to control data reads and writes [30]. Updating PKRU does not enter the host kernel, allowing Flux to retain direct function calls for syscalls.

**Memory Layout.** Each address space contains three MPK domains. The **Application Domain** contains the application and its libraries. The **Flux Domain** contains the Flux kernel, runtime, and supporting libraries, and cannot be read or written while application code is running. The **Shared Domain** contains exchange state that the application may read but only Flux may modify, such as the syscall entry address and virtual dynamic shared object (vDSO) regions. Consequently, although MPK provides at most 16 keys, it does not limit the number of Flux processes.

**Secure Transitions.** Flux uses a call gate on both syscall and UINTR entry to switch PKRU permissions via WRPKRU. To prevent malicious PKRU changes, it enforces W$\bigoplus$X[2] permissions, scans application binaries for illegal WRPKRU and XRESTOR instructions, and uses safe trampolines that verify EAX before execution to thwart control-flow hijacking. To harden control flow, Flux also resolves syscall symbols at fixed addresses in the Shared Domain, avoiding Procedure Linkage Table (PLT) poisoning, and runs kernel code on isolated stacks to prevent return-address tampering.

**Host-Interface Confinement.** MPK controls memory accesses but does not prevent an application from invoking host syscalls that operate on protected mappings or device state [10]. A modified `glibc` routes supported application syscalls into Flux as direct calls, and the ELF loader rewrites SYSCALL instructions in loaded binaries. However, malicious code may still bypass this path by jumping to a gadget containing a SYSCALL instruction not rewritten by the loader (Table 3). The `flux_mpk` kernel module therefore installs a syscall filter that intercepts every host syscall and admits it only when the current PKRU state grants access to the Flux Domain [74].

[2]W and X denote write and execute permissions, respectively. $\bigoplus$ denotes exclusive OR, so a memory page cannot be both writable and executable at the same time.

# 5 Implementation

## 5.1 Modularized Implementation

Flux is implemented on Linux v6.6. Most changes stay in the architecture (`arch`) layer, letting Flux rework dedicated execution paths without modifying unrelated subsystems. Table 3 summarizes the code added or modified by Flux. This implementation also preserves portability. By separating ISA-specific logic from the rest of the runtime, `arch` code can be reused across future ports and rebased onto newer kernels with modest effort. For example, it took only one person-day and roughly 300 lines of code to adapt the x86 and SMP modules to Linux v6.17.

Under the same kernel configuration as native Linux v6.6, native Linux exposes 349 valid syscalls, while Flux supports 345 of them. The four unsupported syscalls are `modify_ldt`, `kexec_load`, `map_shadow_stack`, and `perf_event_open`. They require segmentation, kexec, CET, and performance-monitoring support. Of the 345 syscalls supported by Flux, 314 directly reuse the standard Linux v6.6 implementations and execute inside the Flux kernel. Another 8 use Flux-specific implementations but operate only on state maintained by Flux. The remaining 23 require host cooperation to complete the actual operation after Flux finishes argument checking and state maintenance. For example, `mmap` requires Flux to establish the mappings in the host address space.

## 5.2 Syscall and Signal Handling

**Linux Syscall.** The default Linux syscall execution path consists of the following steps: (1) saving general-purpose registers and switching to the kernel stack; (2) performing pre-syscall preparation, including enabling interrupts and checking for pending work; (3) looking up the corresponding syscall handler in the syscall table and invoking it; (4) performing post-syscall cleanup, including disabling interrupts and checking the TIF (thread info flag) in a loop; (5) switching back to the user stack, restoring general-purpose registers, and returning to user mode.

**Syscall Optimizations.** Flux shortens the syscall path. First, it skips the interrupt-enable step because applications jump directly to the syscall entry with UINTR already enabled. Second, it bypasses the generic entry/exit machinery and checks only signal-related flags after the handler returns. Signals observed later can still be deferred to the next syscall

| | Modules | LOC | Description |
|---|---|---|---|
| **Arch** | `x86/sched` (§4.2) | 7,443 | x86 support for the clock, interrupt handling, and thread switching. |
| | `x86/crypto` (§5.4) | 3,564 | x86 implementations of hashing, encryption, and other generic operations. |
| | `x86/syscalls` (§5.2) | 2,317 | x86 syscall entry, syscall dispatch, and signal processing. |
| | `x86/mmu` (§7) | 2,431 | x86 page table configurations, memory mappings and NUMA support. |
| | `timer` (§4.2) | 1,004 | Timer-source registration and configuration. |
| | `smp` (§4.2) | 694 | Single-level-scheduling-based multicore support. |
| | `fastnet` (§5.3) | 3,730 | TCP-stack fast path and packet processing. |
| | `hostfs` (§5.4) | 768 | Host file-system access and mount-point support. |
| | `mpk` (§4.4) | 532 | Call gates and memory isolation using Intel MPK. |
| **Drivers** | `dpdk` (§5.3) | 2,783 | DPDK-based network device driver and I/O daemon support. |
| | `spdk` (§5.4) | 1,426 | SPDK-based block device driver and runtime integration. |
| **Runtime** | Environment Setup | 4,806 | Environment initialization, configuration, and kernel bootstrap. |
| | I/O Daemon | 2,717 | Centralized control, packet dispatching and timer support. |
| | ELF Loader | 1,005 | Loading the interpreter, the modified `glibc`, and mapping the application's code and data segments. For binaries that contain `SYSCALL` instructions, Flux rewrites these instructions to redirect them to its syscall entry [82]. |
| | Host Operations | 1,868 | Host-specific thread, signal, timer, memory, and virtual dynamic shared object (vDSO) operations. |
| | glibc | 84 | Modified entry points issue direct calls into Flux's syscall interface. |
| **Kernel Module** | `flux_uintr` (§4.2) | 680 | UINTR context setup, per-CPU state tracking, and IPI delivery. |
| | `flux_mm` (§4.3) | 1,764 | Multi-process address-space management and device hooks. |
| | `flux_mpk` (§4.4) | 656 | Syscall filtering and MPK enforcement for signal handling. |

**Table 3.** Modules of Flux. (40,272 LOC in total)

or UINTR exit without violating Linux semantics. Third, it returns directly with `ret` after syscall-table dispatch. It saves the user context and restores it only in special cases: clone-related syscalls restore registers for child setup, while signal delivery and syscall restart fall back to the default exit path to build `ucontext`, restore register state, and resume correctly.

**Signal Handling.** For host exceptions such as synchronous faults (`SIGSEGV` or `SIGILL`), the host signal handler switches domain when MPK isolation is enabled and copies the signal information into a per-CPU queue. To accelerate delivery, Flux rewrites the host `ucontext` to synthesize a UINTR stack frame. The fault then re-enters through the same UINTR path used for timers and IPIs. Flux also ensures that UINTR is disabled when control re-enters the UINTR handler after restoring the user-space context. Flux's UINTR handler then drains the per-CPU queue and builds the signal frame.

### 5.3 Network Fast Path

**Native Linux Path.** By default, Flux uses the native Linux network stack and driver. When the driver allocates Message Signaled Interrupts Extended (MSI-X) vectors, Flux registers one host `eventfd` for each active vector. A host interrupt thread waits on these descriptors and forwards each notification as the corresponding Flux device interrupt. The regular interrupt handler schedules completion-queue polling and continues through Linux's unmodified network stack.

**DPDK and Fast Path.** Flux integrates DPDK through a dedicated network driver. A dedicated core polls NIC queues and dispatches packets to other cores via IPC. Flux runs one polling thread per core to fetch received packets from IPC queues and process completed packet transmissions. The dedicated core could be eliminated for better scalability in future work [27]. To cut latency, Flux integrates a TCP fast path inspired by prior systems [20, 21, 62], while leaving protocols such as ARP and ICMP on the default Linux stack. This fast path also supports pollable sockets for compatibility with the Linux VFS layer, enabling reuse of existing mature mechanisms such as epoll.

**Buffer Management.** Flux keeps packet payloads in DMA-mapped memory throughout the network fast path and reclaims each buffer only after its final consumer releases it. For packet reception, DPDK allocates packet buffers from a shared mempool, while Flux allocates only per-packet metadata through Linux `kmem_cache` and processes the payload in place. For packet transmission, Flux allocates packet buffers from a dedicated DMA zone through `kmem_cache`. These buffers are returned to the allocator after completion. To reduce allocation overhead, each CPU caches recently released metadata for received packets and buffers for outgoing packets. New allocations first draw from the local cache, while cache misses and excess released objects fall back to the underlying `kmem_cache`. This design avoids intermediate payload copies while reducing allocator contention and preserving CPU locality.

**Network Thread Scheduling.** Polling-thread priority is critical. Under the default fair class, polling threads consume excessive CPU time, preventing application threads

from promptly processing incoming packets. Flux therefore assigns them `SCHED_IDLE`, which avoids unnecessary transitions to the default idle thread when polling blocks on an empty receive queue while still prioritizing application threads. Under idle conditions, the polling thread uses `UMWAIT` [30], a low-power instruction that blocks until a memory write occurs, after several unsuccessful polling attempts. Flux also tries to process incoming network packets before entering a pre-blocking slow path such as `epoll`, thereby avoiding subsequent scheduling overhead.

### 5.4 Storage Integration

**Host File System.** Flux implements a shim compatible with its internal VFS, enabling direct access to the host file system through Linux file-system interfaces. Host directories can be configured and mounted into Flux at startup.

**SPDK Integration.** Flux also integrates SPDK [13] through a custom block driver so Linux file systems can submit I/O directly to SPDK queue pairs. To handle completions efficiently, Flux uses one polling thread per core, matching SPDK's lock-free queues. Flux uses completion-aware polling: it checks for rescheduling after processing completions and briefly backs off when outstanding I/O makes no progress, instead of yielding on every poll [41]. Because SPDK uses pinned huge pages for DMA, Flux allocates file pages from DMA-capable memory and submits them without extra copies. File systems such as ext4 frequently compute checksums for journal blocks and metadata. Flux therefore ports x86-specific implementations of operations such as `CRC32` into the `arch` layer so these hot paths avoid generic software fallbacks.

MPK isolation and the call gate further protect both access paths, preventing applications from bypassing the configured mounts to modify arbitrary host files or crash the storage subsystem.

## 6 Evaluation

We evaluate Flux from an OS perspective to address the following questions:

- **Compatibility Analysis**: How closely does Flux match Linux behavior on the Linux Test Project (LTP) compared with other library OSes? (§6.2)
- **I/O Performance**: Can Flux deliver high throughput and low tail latency for file systems and network applications relative to native Linux and from-scratch library OSes? (§6.3 and §6.4)
- **Multi-process Support**: Does Flux maintain its performance advantages when running complex multi-process applications? (§6.5)
- **Hardware Preemption**: Can Flux efficiently preempt user tasks and achieve low wakeup latency? (§6.6)
- **Performance Breakdown**: How do Flux design choices contribute to removing bottlenecks? (§6.7, §6.8)

| **Systems** | Kernel Version | Scheduling Policy |
|---|---|---|
| **Linux** | v6.6 | EEVDF[1] [12] |
| **UML** [16] | v6.6 | EEVDF |
| **LKL** [67] | v6.6 | EEVDF |
| **rkt-io** [78] | v4.17 | CFS[2] [58] |
| **gVisor** [23] | - | G-M-P[3] [79] |
| **Junction** [20] | - | FIFO[4] (work stealing [62]) |
| **Flux** | v6.6 | EEVDF |

**Table 4.** Evaluated systems and their Linux kernel versions and scheduling policies. [1] Earliest Eligible Virtual Deadline First (EEVDF). [2] Completely Fair Scheduler (CFS). [3] Go's goroutine-machine-processor (G-M-P) scheduling model. [4] First in, first out (FIFO).

### 6.1 Experimental Setup

**Testbed.** The server has two 24-core Intel Xeon Gold 5418Y processors running at 2.0 GHz and 256 GB of RAM. The client has one 16-core AMD Ryzen 9 5950X CPU running at 4 GHz. To ensure accurate and stable results, we configure both machines according to best practices, disabling TurboBoost, CPU idle states, CPU frequency scaling, transparent huge pages, and hyperthreading. The testbed includes an Intel DC P5801X SSD and 200GbE Mellanox ConnectX-6 NICs. We use DPDK v23.11 and SPDK v24.09 for kernel-bypass I/O. We clear the file-system cache before each run.

**Systems.** As shown in Table 4, we compare Flux with several representative Linux-based library OSes, including LKL [67] and rkt-io [78]. We compare Flux with Junction [20], a state-of-the-art from-scratch library OS. Table 4 summarizes the evaluated systems and their scheduling policies.

We also compare Flux with gVisor [23] and User-mode Linux (UML) [16]. Both rely on separate host processes and intercept application syscalls through the host kernel. gVisor is a container runtime for isolation and compatibility. We enable the `systrap` optimization for gVisor. UML runs Linux in user space as a single process and uses `ptrace` [47] to intercept application syscalls. In our experiment, UML based on Linux v6.6 supports only one virtual CPU.

Unless otherwise stated, server-side core counts in network experiments refer to application-worker cores, with one worker thread per core, consistent with prior work [20, 33]. Flux and Junction each use one additional core for packet dispatch and timer-interrupt generation. However, our experiments show that this additional core does not materially affect performance or the relative results for Linux and gVisor. For experiments involving multiple applications (§6.2) and storage devices (§6.3, §6.5), we enable Flux's MPK-based isolation.

### 6.2 Compatibility Analysis

We use the LTP 20260130 release to evaluate Linux compatibility. We select 2,563 x86-64 test cases and apply a strict

| Metrics | Linux | Flux | gVisor | UML |
|---|---|---|---|---|
| **Passed** | 2,450 | 2,362 | 1,066 | 1,948 |
| **Pass rate** | 95.6% | 92.2% | 41.6% | 76.0% |

**Table 5.** LTP compatibility results over 2,563 test cases.

| Workloads | Avg. File Size | Files | Dir. Width | I/O Size | R/W Ratio |
|---|---|---|---|---|---|
| **Varmail** | 16KB | 1K | 1M | 1MB | 1:1 |
| **Fileserver** | 128KB | 10K | 20 | 1MB | 1:2 |
| **Webserver** | 16KB | 1K | 20 | 1MB | 10:1 |
| **Webproxy** | 16KB | 10K | 1M | 1MB | 5:1 |

**Table 6.** Filebench [18] workload characteristics.

passing criterion: a test is counted as passed only if it exits successfully, reports explicit passing evidence, and completes without runtime, cleanup, or residual-process errors. Native Linux v6.6 passes 2,450 tests (95.6%), followed by Flux with 2,362 tests (92.2%) under the same configuration.

Native Linux skips 99 tests, whereas Flux skips 131. The two systems share all 99 native Linux skips, which mainly require optional kernel configurations or modules, unavailable hardware facilities, legacy interfaces, or subsystem ABIs introduced after Linux v6.6. The additional 32 tests skipped by Flux pass on native Linux. Most of these exercise the cgroup v1 debug controller, which Flux does not expose; the remainder depend on optional cryptographic algorithms, kernel-testing modules, file-system features, memory-management preconditions, or unsupported host-integration operations.

Among the 2,432 non-skipped Flux executions, 2,362 pass, 22 fail their functional assertions, 18 terminate as broken, 6 time out, and 24 lack explicit passing evidence. The failures mainly concern cgroup and memory controller accounting, CPU hotplug and cpuset lifecycle handling, corner cases in security-related syscalls, file-system and credential semantics, timekeeping behavior, and kernel module integration. The broken and timed-out executions are primarily caused by guest startup and CPU-binding failures, incomplete resource cleanup, unresolved memory faults, and long-running memory or I/O stress paths.

The remaining 24 legacy tests exit with status zero and may print test-specific success messages, but do not emit an LTP `TPASS` record or a valid summary with a positive passed count. We therefore report them separately and do not count them as passed.

### 6.3 File-System Performance

Filebench [18] is a scalable benchmark for evaluating file-system performance under realistic workloads. Figure 5 shows the throughput results. Flux with SPDK outperforms Linux, Junction, and gVisor across all workloads. Compared with Linux ext4, Flux ext4 achieves improvements of 47%, 6%, 7%, and 10% on the varmail, fileserver, webserver, and webproxy workloads, respectively. For read/write-intensive

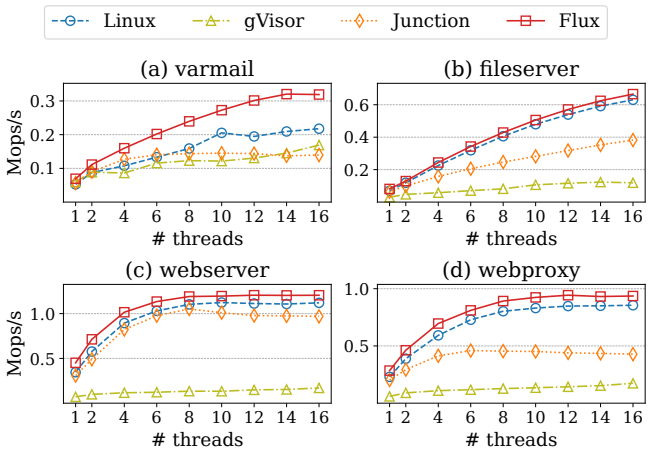


**Figure 5.** Multi-threaded workloads for Filebench.

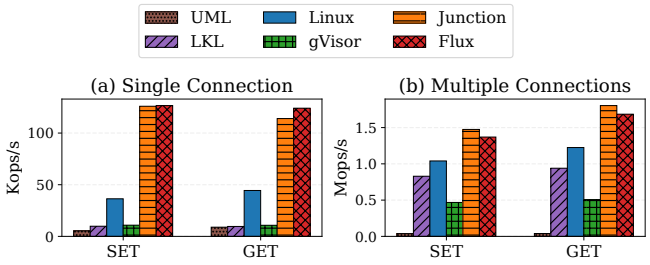


**Figure 6.** Redis throughput. (a) Single connection without pipelining (`-P 1 -c 1`). (b) Multiple connections with pipelining (`-P 16 -c 50`).

workloads, these improvements come from Flux avoiding kernel traps and using host-provided `memcpy` to quickly copy data from file pages to user buffers. For varmail, which stresses synchronous write operations, Flux can fully utilize SPDK interfaces to efficiently flush dirty pages and commit journal transactions.

Compared with Junction ext4, Flux ext4 achieves improvements of 121%, 74%, 15%, and 105% on the four workloads, respectively. This is because Junction invokes host kernel file systems through a shim layer, introducing additional overhead from acquiring locks and looking up directory entries. gVisor delivers the lowest throughput among all systems, primarily because it forwards file requests via 9P IPC to a separate *gofer* process on the host OS, paying heavy context-switch and serialization costs.

### 6.4 Network Applications

We evaluate the networking performance of Flux using two typical in-memory key-value stores, Redis [72] and Memcached [56], together with Nginx [61], a representative multi-process web server. Flux can run their unmodified binaries directly. We disable kernel preemption for a run-to-completion model as in previous work [20, 21, 62, 66]. We compare Flux against native Linux, Junction, and gVisor:

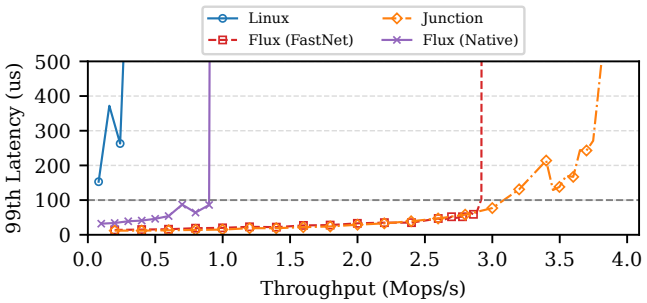


**Figure 7.** Memcached p99 latency and throughput.

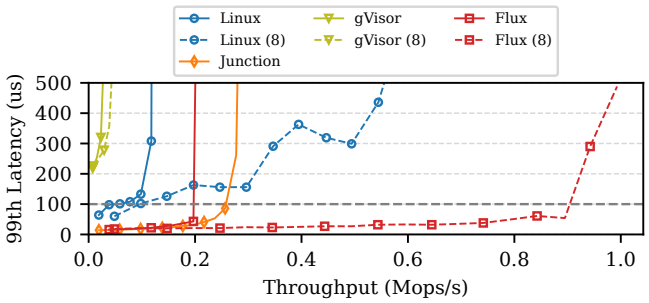


**Figure 8.** Nginx p99 latency and throughput. Junction only supports the single-worker configuration.

- **gVisor**. We use its default Docker setup with port forwarding.
- **UML and LKL**. Because the evaluated UML and LKL configurations do not support SMP, we only run Redis on these systems. UML and LKL cannot access the physical NIC directly. We create a TAP [45] device on the server and attach it, together with the physical NIC, to a software bridge, allowing UML and LKL to communicate with a remote client via the TAP interface.
- **rkt-io**. We also omit rkt-io from the network evaluation. Although rkt-io provides kernel-bypass networking support, it uses a modified, legacy version of DPDK (v19.02.0-rc4), which only supports dedicated NICs and does not support our ConnectX-6 NICs.
- **Demikernel**. Demikernel fails to run Redis successfully. Despite extensive efforts, we could not run Redis on Demikernel, using both a modified version of Redis that directly invokes Demikernel (as described in the Demikernel paper) and a POSIX shim merged into the mainline Demikernel repository.

**Redis.** We use `redis-benchmark` to evaluate the throughput of GET and SET requests over TCP connections. For all systems, we disable periodic Redis snapshots. To minimize the impact of the client-side network stack on latency, we run redis-benchmark inside Junction. Figure 6 presents the results. For a single connection, Flux outperforms Linux and LKL by 3.5× and 12.9× on SET, and by 2.8× and 12.9× on GET. Flux achieves throughput comparable to Junction's. For multiple connections, Flux outperforms Linux and LKL by 1.3× and 1.7× on SET, and by 1.4× and 1.8× on GET. Flux trails Junction by only about 7%, mainly because Junction employs a shorter wakeup path for polled events.

**Memcached.** We use Caladan's synthetic load generator to evaluate Memcached performance under USR workloads [4]. The server is configured to run on 4 cores, while the client runs on 16 cores to fully saturate the server. As shown in Figure 7, Linux experiences sharply increased latency at moderate throughput due to the overhead of the network stack, and the default scheduling cannot meet tail-latency requirements. At the maximum throughput with p99 latency below 100$\mu$s, Flux exhibits only 4% lower throughput than Junction. Flux with the network fast path also outperforms Flux with the original Linux driver by 3.2×. We omit gVisor from the Memcached evaluation because it cannot meet the latency requirements.

### 6.5 Multi-process Support

**Nginx.** Nginx [61] is a widely used web server that typically runs with multiple worker processes to handle concurrent requests. We use Caladan's synthetic load generator as the client and compare Flux with Linux and gVisor. For the single-worker configuration, we disable `master_process`, leaving one Nginx process to handle requests. We also include Junction in this restricted single-worker configuration because its process support does not extend to the normal multi-worker setup. As shown in Figure 8, Flux improves the maximum throughput of single-worker Nginx by 5.1× over Linux. With eight worker processes, it improves maximum throughput by 18.5× over Linux while keeping p99 latency strictly below 100$\mu$s. Flux thus supports practical multi-process applications without giving up its low-latency datapath.

**Redis Persistence.** We use Redis persistence to evaluate whether Flux's multi-process and storage support can sustain a `fork`- and copy-on-write-intensive workload. We compare native Linux, gVisor, Flux with HostFS, and Flux with SPDK on both ext4 and XFS. The workload replays a deterministic Yahoo! Cloud Serving Benchmark (YCSB) Workload A trace [11] with 50% reads, 50% updates, and a scrambled Zipfian key distribution. Every system loads the same trace into memory and replays it directly from the Redis main thread, excluding network and control-socket overhead from the timed path.

We break end-to-end persistence time into Redis processing, `write`, durability synchronization, `fork`, and residual time in Figure 9. With SPDK, Flux finishes in 2.86 s on ext4 and 2.84 s on XFS, reducing completion time by 6.1% and 3.0% compared to Linux, respectively. SPDK reduces the combined write and synchronization time by 17.6% on ext4 and 22.0% on XFS. The end-to-end gains remain modest because Redis's object traversal, serialization, and checksum computation account for most of the snapshot time. HostFS is

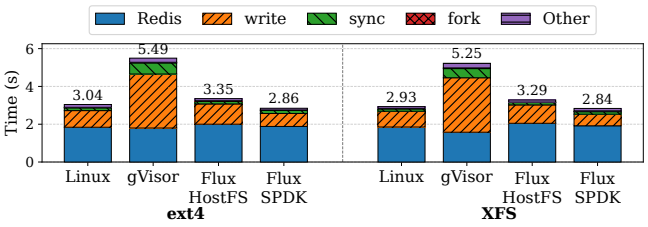


**Figure 9.** Redis persistence breakdown on ext4 and XFS.

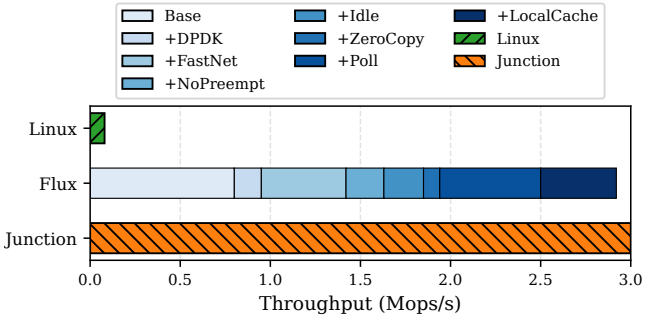


**Figure 10.** Performance breakdown of Flux using Memcached.

slower because each synchronous write traverses both Flux's VFS and the host VFS.

### 6.6 Hardware Preemption

We use schbench [54] to evaluate the scheduling performance of Flux. Schbench emulates the wakeup behavior of web workloads using $M$ message threads and $T$ worker threads. Message threads monitor workers and wake them when they sleep. Workers perform synthetic tasks and then block themselves, waiting to be awakened again. We configure schbench with 16 cores. One dedicated message thread is bound to a single core, while each of the remaining cores runs exactly one worker thread. We omit UML because the evaluated UML based on Linux v6.6 does not support SMP. We patch Junction's scheduling framework to support thread affinity. Table 7 shows the 99th-percentile wakeup latency. Compared with native Linux, Flux reduces the latency by 78%. Junction achieves fast wakeups because each core has only one thread, which is scheduled immediately after being placed on the run queue.

### 6.7 Understanding the Performance

Figure 10 shows a stepwise Memcached breakdown of our optimizations under a 100$\mu$s p99 latency target:

- **Base:** Running the unmodified Linux network stack with the Linux NIC driver (detailed discussion in §5.3).
- **+DPDK:** Running the unmodified Linux network stack with the DPDK dispatcher and polling threads.
- **+FastNet:** Integrating a kernel-bypass TCP fast path to accelerate packet processing.
- **+NoPreempt:** Disabling preemptive scheduling to prevent scheduling thrashing.
- **+Idle:** Adopting an idle polling mechanism to mitigate the overhead of frequent context switches between idle and application threads.
- **+ZeroCopy:** Enabling zero-copy I/O by directly allocating kernel payload buffers from DMA-mapped regions.
- **+Poll:** Speculatively polling for incoming packets before scheduling the application thread, reducing wakeup latency.
- **+LocalCache:** Using a per-CPU cache for incoming packet metadata and outgoing packet buffers to reduce contention and improve cache locality.

Collectively, these optimizations improve the baseline Flux performance by 3.65× (from 0.8 Mops to 2.92 Mops). These optimizations bring Flux close to Junction's performance while maintaining Linux compatibility.

### 6.8 Microbenchmarks

Table 7 summarizes the microbenchmark results with the call gate (§4.4) disabled. Enabling it adds about 100 cycles of overhead to each syscall on our machine, which has a minimal impact on our end-to-end experimental results.

**Single Process.** For a simple syscall (`gettid`), Flux demonstrates direct-call overhead comparable to that of other library OSes. For thread switching (*yield*), Flux tracks closely behind Junction's minimal FIFO scheduler and outperforms other systems with its single-level scheduling. For sleep, gVisor exhibits a massive ~1 ms delay because its short sleeps rely on Go's timer and goroutine channel notifications, which lack precision. Flux utilizes UINTR to deliver timer interrupts, achieving lower latency than native Linux. For condition variables, Flux is much faster than gVisor and UML.

**Multiple Processes.** We evaluate `fork` by spawning processes serially on one core and concurrently across 4 cores. Although Flux's `fork` is slower than native Linux on four cores due to the overhead of maintaining two layers of `mm_struct` (Flux's and the host kernel's) and additional reference counting, Flux significantly outperforms other systems. We also use hackbench [73] (-pPl 10000), an IPC and scheduler stress test that repeatedly wakes up many sender-receiver groups via signals and Unix sockets. Flux outperforms Linux by 3× (8s vs. 25s) because of its fast syscall handling and scheduling.

## 7 Discussion

**Hardware Requirements.** The current Flux implementation targets Intel x86 servers with UINTR and MPK support. UINTR enables low-latency timer and IPI delivery, while MPK protects the Flux kernel and runtime without adding a host transition to every syscall. On machines without UINTR, Flux can instead use host-mediated timer and interrupt delivery, at the cost of additional host transitions and higher

| Systems | yield | gettid | sleep | condvar | spawn | yield (process) | fork (1c/4c) | schbench ($\mu$s) | hackbench (s) |
|---|---|---|---|---|---|---|---|---|---|
| LKL | 610 | 191 | 377,514 | 6,757 | 14,199 | - | - | - | - |
| rkt-io | 1,812 | 21 | 363,302 | 2,441 | 176,890 | - | - | 11 | - |
| Junction | 394 | 19 | 221,110 | 1,013 | 12,848 | - | - | 3 | - |
| UML | 26,459 | 17,306 | 339,947 | 65,661 | 158,844 | 22,715 | 337,038/530,353 | 32,080 | >1,000 |
| gVisor | 5,710 | 3,924 | 2,148,485 | 18,282 | 74,452 | 4,407 | 406,845/815,499 | 2,363 | 739 |
| Linux | 2,548 | 296 | 307,880 | 6,893 | 13,826 | 3,310 | 119,086/139,177 | 18 | 25 |
| **Flux** | 573 | 34 | 220,732 | 1,266 | 3,055 | 2,872 | 115,641/176,632 | 4 | 8 |

**Table 7.** Microbenchmark results (*cycles/op* unless otherwise specified). *Sleep* results report total cycles for `usleep(100)`. A dash indicates that the corresponding system cannot run the test.

wakeup latency. Other architectures need counterparts to UINTR and MPK. For example, Donky provides MPK-like isolation on RISC-V [75].

**Device Interrupts and Drivers.** Flux shows how Linux device drivers can be reused when Linux runs as a library OS. The current implementation forwards host device interrupts through `eventfd`, allowing unmodified Linux interrupt handlers and driver code to execute inside Flux. This approach may also benefit operating systems with limited driver support, such as HongMeng [9]: rather than porting each driver into the target OS, a Linux application-kernel layer could provide access to the existing Linux driver ecosystem. The remaining challenge is performance. Host-mediated interrupts add transitions, while efficient polling and DMA integration still require device-specific runtime support.

**Memory Mapping and Page Faults.** Memory-mapped file access is important to databases [22, 25, 26, 29, 59, 69]. Flux creates virtual-address aliases for its initial virtual mappings to support mechanisms such as `mmap` and demand paging. It injects page faults through a host signal handler and uses the kernel module for privileged operations, including mapping pages and flushing the Translation Lookaside Buffer (TLB). Both signal-based fault injection [48] and `userfaultfd` [49] take tens of microseconds on our machine. Consequently, applications that frequently fault on `mmap`-backed files can run substantially slower under Flux. Reducing this cost requires hardware or host support for lower-overhead fault delegation and TLB invalidation.

## 8 Related Work

We discuss from-scratch library OSes related to our work in §2.1.

**Compatible Secure Containers.** gVisor [23] prioritizes broad Linux ABI compatibility and container isolation over I/O performance, running as either a process or a VM. Recent systems [7, 53] explore similar design points with different service-decomposition mechanisms. But these systems perform worse than library OSes, and achieving Linux compatibility requires significant engineering effort compared to using Linux directly.

**Linux in Other Environments.** Existing work ports Linux to other environments. UML [16] runs Linux as a user-space process, using `ptrace` to intercept syscalls and host signals to handle interrupts. L4Linux [38] runs Linux on the L4 microkernel to provide Linux compatibility. Lupine [37] and UKL [70] run trimmed Linux images as unikernels in non-root mode. X-Containers [76] runs Linux on a specialized kernel. UIEE [81] combines LKL with a TrustZone-oriented isolated execution environment. Flux uses small host-kernel modules only for privileged operations.

**User-Space Preemptive Scheduling.** Recent research has explored user-space preemptive scheduling enabled by Intel User Interrupts. Libpreemptible [42] introduces timer abstractions to facilitate preemptive schedulers. Skyloft [33] manages hardware timer interrupts in user space and supports multi-application scheduling. Vessel [44] enables fast core allocation among applications through its uprocess abstraction, and Vessel performs comparably to Junction. SBB [27] handles both timer and NIC interrupts and removes centralized overhead. Flux is the first Linux-based library OS with multicore preemptive scheduling.

## 9 Conclusion

This work shows that Linux can serve as a high-performance library OS when its user-space runtime is designed appropriately. Flux keeps Linux subsystems largely intact while reducing runtime overheads and supporting Linux `fork` semantics. Our evaluation shows substantial gains over existing Linux-based systems and performance close to that of from-scratch designs on representative workloads, demonstrating that Linux as a library OS can be both fast and immediately ready for real applications.

## Acknowledgments

We thank the anonymous reviewers and our shepherd for their constructive suggestions. We also thank the reviewers of ASPLOS 2026 and OSDI 2026 for their helpful comments. This work was supported by the Smart Grid-National Science and Technology Major Project (2024ZD0803000) and the National Natural Science Foundation of China (92467102). Correspondence to: Kang Chen (chenkang@pku.edu.cn), Yu Chen (yuchen@tsinghua.edu.cn).

## References


[1] Hasan Alayli. 2012. lthread 0.5: A Multithreaded Coroutine Library in C. https://github.com/halayli/lthread.

[2] Apache Flink PMC. 2026. Apache Flink 2.3.0. https://flink.apache.org. Stream and batch data analytics engine.

[3] Apache Software Foundation. 2026. Apache Spark 4.2.0. https://spark.apache.org. Unified analytics engine for large-scale data processing.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. doi:10.1145/2254756.2254766

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[7] Xiaohu Chai, Keyang Hu, Jianfeng Tan, Tiwei Bie, Guotao Tan, Tianyu Zhou, Anqi Shen, Dawei Shen, Xinyao Yang, Xin Chen, Xu Wang, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. 2026. SKernel: An Elastic and Efficient Secure Container System at Scale with a Split-Kernel Architecture. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 605–623. doi:10.1145/3767295.3769332

[8] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[9] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. 2024. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 465–485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo

[10] Emma Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Virtual Event, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russ Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[12] Jonathan Corbet. 2023. An EEVDF CPU Scheduler for Linux. https://lwn.net/Articles/925371/. Linux Weekly News.

[13] Intel Corporation. 2016. Storage Performance Development Kit (SPDK). https://spdk.io.

[14] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. doi:10.1145/3477132.3483571

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. doi:10.1145/2463676.2463710

[16] Jeff Dike. 2001. User-mode Linux. In *5th Annual Linux Showcase & Conference (ALS 01)*. USENIX Association, Oakland, CA, 12 pages. https://www.usenix.org/publications/library/proceedings/als01/dike.html

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[18] Filebench Project. 2016. Filebench 1.4.9.1: A Filesystem and Storage Benchmark. https://github.com/filebench/filebench.

[19] Linux Foundation. 2015. Data Plane Development Kit (DPDK). https://www.dpdk.org.

[20] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 55–73. https://www.usenix.org/conference/nsdi24/presentation/fried

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Virtual Event, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[22] Sanjay Ghemawat and Jeff Dean. 2021. LevelDB 1.23: A Fast Key-Value Storage Library. https://github.com/google/leveldb.

[23] Google. 2026. gVisor: Application Kernel for Containers. https://gvisor.dev/.

[24] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[25] Gavin Henry. 2019. Howard Chu on Lightning Memory-Mapped Database. *IEEE Software* 36, 6 (Nov. 2019), 83–87. doi:10.1109/MS.2019.2936273

[26] D. Richard Hipp. 2022. SQLite Memory-Mapped I/O. https://www.sqlite.org/mmap.html.

[27] Kang Hu, Shuqi Dong, Chuandong Li, Ran Yi, Zonghao Zhang, Yiming Yao, Bo An, Jie Zhang, Xiaolin Wang, Yingwei Luo, Zhenlin Wang, and Diyu Zhou. 2026. SBB: Eliminating Centralized Bottlenecks in Userspace Network Runtime. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 473–489. https://www.usenix.org/conference/osdi26/presentation/hu-kang

[28] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. 2023. PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environment. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 515–530. doi:10.1145/3600006.3613158

[29] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering*

*Bulletin* 35, 1 (March 2012), 40–45.

[30] Intel Corporation. 2026. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Relevant sections include Volume 2B (UMWAIT), Volume 2C (VMFUNC), and Volume 3A Chapters 4 (Paging) and 9 (User Interrupts); https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[31] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 466–481. doi:10.1145/3600006.3613136

[32] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[33] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 265–279. doi:10.1145/3694715.3695973

[34] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 295–306. doi:10.1145/2619239.2626299

[36] Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. 2025. No Cap, This Memory Slaps: Breaking through the Memory Wall of Transactional Database Systems with Processing-in-Memory. *Proc. VLDB Endow.* 18, 11 (July 2025), 4241–4254. doi:10.14778/3749646.3749690

[37] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. doi:10.1145/3342195.3387526

[38] L4Linux Project. 2026. L4Linux 7.1: Running Linux on the L4Re Microkernel. https://l4linux.org.

[39] Jiangshan Lai, Hang Huang, Quan Xu, Zhen Ren, Wenlong Hou, Wei Guo, Jia Rao, Hui Lu, Weidong Han, Jiesheng Wu, Jiang Liu, Naixuan Guan, Yibin Shen, Feng Yu, Xu Wang, Shiqiang Zhang, Zhiheng Tao, Yisheng Xie, Song Wu, and Hai Jin. 2026. JANUS: CrossWorld, Cooperative Nested Virtualization for Secure Containers. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 1015–1031. https://www.usenix.org/conference/osdi26/presentation/lai

[40] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. 2024. Loupe: Driving the Development of OS Compatibility Layers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 249–267. doi:10.1145/3617232.3624861

[41] Chuandong Li, Ran Yi, Zonghao Zhang, Jing Liu, Changwoo Min, Jie Zhang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. 2025. Aeolia: A Fast and Secure Userspace Interrupt-Based Storage Stack. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 479–495. doi:10.1145/3731569.3764816

[42] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Edinburgh, United Kingdom, 922–936. doi:10.1109/HPCA57654.2024.00075

[43] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[44] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. 2024. Fast Core Scheduling with Userspace Process Abstraction. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 280–295. doi:10.1145/3694715.3695976

[45] Linux Kernel Documentation. 2026. Universal TUN/TAP Device Driver. https://docs.kernel.org/networking/tuntap.html.

[46] Linux man-pages project. 2026. fork(2): Create a Child Process, Linux man-pages 6.18. https://man7.org/linux/man-pages/man2/fork.2.html.

[47] Linux man-pages project. 2026. ptrace(2), Linux man-pages 6.18. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[48] Linux man-pages project. 2026. signal(7): Overview of Signals, Linux man-pages 6.18. https://man7.org/linux/man-pages/man7/signal.7.html.

[49] Linux man-pages project. 2026. userfaultfd(2): Create a File Descriptor for Handling Page Faults in User Space, Linux man-pages 6.18. https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[50] Linux man-pages project. 2026. vfork(2): Create a Child Process and Block the Parent, Linux man-pages 6.18. https://man7.org/linux/man-pages/man2/vfork.2.html.

[51] linux-rdma organization. 2026. rdma-core 64.0: RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core.

[52] LSDS. 2019. SGX-LKL: Secure Linux Kernel Library for Intel SGX, SMP Support Snapshot. https://github.com/lsds/sgx-lkl.

[53] Kaesi Manakkal, Nathan Daughety, Marcus Pendleton, and Hui Lu. 2025. LITESHIELD: Secure Containers via Lightweight, Composable Userspace μKernel Services. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 973–985. https://www.usenix.org/conference/atc25/presentation/manakkal

[54] Chris Mason. 2016. schbench 1.0. https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench/+/refs/tags/v1.0.

[55] Syed Akbar Mehdi, Deukyeon Hwang, Simon Peter, and Lorenzo Alvisi. 2023. ScaleDB: A Scalable, Asynchronous In-Memory Database. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 361–376. https://www.usenix.org/conference/osdi23/presentation/mehdi

[56] Memcached Project. 2025. Memcached. https://memcached.org.

[57] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany, 2019-03-25) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 15 pages. doi:10.1145/3302424.3303946

[58] Ingo Molnar. 2007. Completely Fair Scheduler (CFS). https://docs.kernel.org/scheduler/sched-design-CFS.html. Linux Kernel Documentation.

[59] MongoDB, Inc. 2024. WiredTiger 11.3.1 API Reference. https://source.wiredtiger.com/11.3.1/group__wt.html. Documents the memory-mapped file access configuration.

[60] Nginx Project. 2026. Controlling nginx. https://nginx.org/en/docs/control.html.

[61] Nginx Project. 2026. nginx 1.30.4. https://nginx.org.

[62] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[63] PCI-SIG. 2010. *Single Root I/O Virtualization and Sharing Specification, Revision 1.1*. PCI-SIG. https://pcisig.com/PCIExpress/Specs/IOV/SingleRootIOVirtualizationandSharing_1.1

[64] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. 2025. Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 360–378. doi:10.1145/3689031.3696083

[65] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[66] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. doi:10.1145/3132747.3132780

[67] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux Kernel Library. In *9th RoEduNet IEEE International Conference*. IEEE, Sibiu, Romania, 328–333.

[68] Fabian Rauscher and Daniel Gruss. 2024. Cross-Core Interrupt Detection: Exploiting User and Virtualized IPIs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 94–108. doi:10.1145/3658644.3690242

[69] RavenDB Documentation. 2025. RavenDB 7.1 Storage Engine: Voron. https://ravendb.net/docs/article-page/7.1/csharp/server/storage/storage-engine.

[70] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 590–605. doi:10.1145/3552326.3587458

[71] Redis Project. 2026. Redis 8.10 Persistence Documentation. https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/.

[72] Redis Project. 2026. Redis 8.10.0. https://redis.io.

[73] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. 2026. hackbench(8): Scheduler Benchmark and Stress Test, rt-tests 2.10. https://man.archlinux.org/man/hackbench.8.en.

[74] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[75] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Virtual Event, 1677–1694. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[76] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 121–135. doi:10.1145/3297858.3304016

[77] Hajime Tazaki, Akira Moroo, Yohei Kuga, and Ryo Nakamura. 2021. How to Design a Library OS for Practical Containers?. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/3453933.3454011

[78] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. rkt-io: a Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 490–506. doi:10.1145/3447786.3456255

[79] The Go Authors. 2026. The Go Programming Language: Runtime Scheduler Source Code. https://go.dev/src/runtime/proc.go.

[80] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. doi:10.1145/2901318.2901341

[81] Huaiyu Yan, Zhen Ling, Xuandong Chen, Xinhui Shao, Yier Jin, Haobo Li, Ming Yang, Ping Jiang, and Junzhou Luo. 2026. UIEE: Secure and Efficient User-space Isolated Execution Environment for Embedded TEE Systems. In *Proceedings 2026 Network and Distributed System Security Symposium (NDSS 2026)*. Internet Society, San Diego, CA, USA, 18 pages. doi:10.14722/ndss.2026.230208

[82] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 293–300. https://www.usenix.org/conference/atc23/presentation/yasukata

[83] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 266–282. doi:10.1145/3552326.3567488

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. doi:10.1145/3477132.3483569