Fuchsia Internals — Volume IV

Starnix: Running Unmodified Linux on Zircon

Fuchsia Internals Series

***Abstract*—Starnix is Fuchsia's Linux runtime. It reimplements the behaviour of the Linux kernel in Rust, as an ordinary Fuchsia component that runs in user space. Starnix does not boot Linux in a virtual machine. It runs the user code of unmodified Linux and Android binaries directly on the CPU, using a Zircon feature called *restricted mode*, and it emulates the Linux system calls, file system, signals, namespaces and IPC. Each Linux thread is a Zircon thread that alternates between restricted mode, where Linux code runs, and normal mode, where Starnix handles the system call. Linux address spaces are built from Zircon VMOs and VMARs. Linux files come from a virtual file system inside Starnix, which reaches native Fuchsia file systems through the `fuchsia.io` protocol. For Android, Starnix provides its own Binder and ashmem drivers and passes GPU work through to the Fuchsia graphics stack. This volume covers each of these parts in turn: the execution loop, tasks and namespaces, memory, system call dispatch, the file system and its bridge to Fuchsia, signals, networking, the Android drivers, containers and boot, diagnostics, and performance. Each part is tied to the Zircon primitives of Volume III. The volume is written for hardware and systems engineers who want an exact model of how a second operating system can run on a capability microkernel without a hypervisor.**

***Index Terms*—Starnix, Fuchsia OS, Linux compatibility, restricted mode, system-call emulation, Binder, virtual file system, namespaces, Android, Zircon, OS personality.**

I. Introduction

STARNIX lets Fuchsia run existing Linux and Android software. Fuchsia's native system call interface looks nothing like Linux, so a Linux binary cannot run on it unaided. The usual answer is a virtual machine that boots a real Linux kernel as a guest. Fuchsia can do that as well; Volume V of the *Fuchsia Subsystems* series covers its hypervisor. Starnix takes a different path. It reimplements what a Linux program can see of the Linux kernel: the system call table, the file system, signals, namespaces, the process model and the Android drivers. That code is a Rust program that runs as a normal Fuchsia component, called the **starnix kernel**. The Linux program's own machine code runs unmodified on the CPU. Only its system calls go to Starnix.

A program built this way is called a *user-space kernel*. gVisor and User-Mode Linux are two others. On Fuchsia the feature that makes it work is Zircon's **restricted mode** (Volume III). Restricted mode lets one thread switch between two modes, each with its own register state. In normal mode the thread runs trusted Starnix code. In restricted mode it runs untrusted Linux code, and every Linux system call or unhandled fault returns it to normal mode. Linux threads share the Zircon scheduler, address spaces and memory with the rest of the system. So there is no second kernel scheduling guests, no emulated MMU, and no virtual devices. What a virtual machine spends on those, Starnix spends instead on one round trip between the two modes for each system call.

TABLE I
Where the "Linux kernel" lives, by approach.

Approach	Guest kernel	Syscall path	Isolation / cost
Virtual machine	Real Linux, in guest	Guest kernel (no exit)	Strong (stage-2); heavy
Starnix	Re-implemented in host userspace	Restricted-mode bounce to host	Process + capabilities; light
gVisor	Re-implemented (Go)	systrap or KVM trap	Process; medium
Containers	Shared host kernel	Native syscalls	Namespaces only

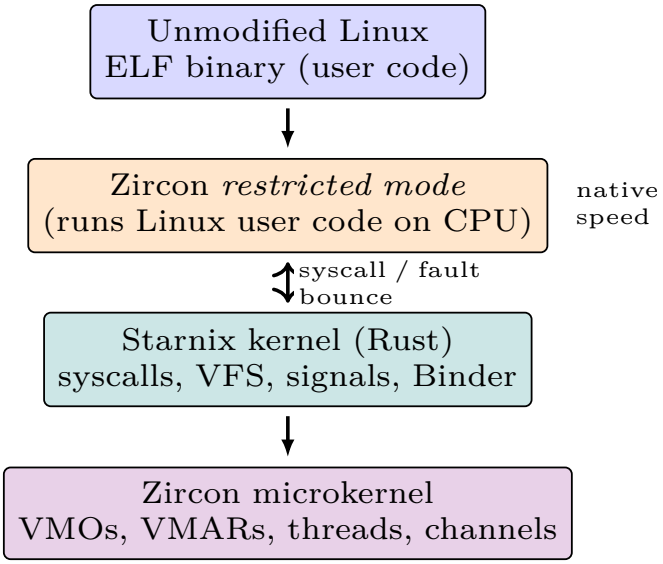


Fig. 1. Starnix runs Linux user code natively in restricted mode, and implements the Linux kernel in a Fuchsia component.

Scope. Starnix implements the interface that Linux offers to user programs. That covers system calls, `/proc` and `/sys`, signals, namespaces, the rules for loading ELF files and the dynamic linker, and the vDSO. It does not implement the interface that Linux offers to its own modules, so you cannot load a `.ko` file. Linux programs also expect some devices: Binder, ashmem, the GPU, a framebuffer and input. Starnix reimplements some of these and passes the rest to native Fuchsia services. The rest of this volume follows a Linux process from its launch through each part of Starnix it uses. It closes with a performance model

Volume IV of the fourteen-volume *Fuchsia Internals* series (I: Graphics & Display Pipeline; II: Build System & Toolchains; III: The Zircon Kernel; IV: Starnix; V: Graphics FIDL APIs in Practice; VI: The Component Framework; VII: Driver Development; VIII: Storage; IX: Power Management; X: Networking; XI: FIDL; XII: Software Delivery & OTA; XIII: Diagnostics & Observability; XIV: Security & Verified Boot). Compiled 2026 from public Fuchsia OS source and documentation at `fuchsia.dev` and `cs.opensource.google/fuchsia`.

and a walk through the life of one process (§XVI).

II. Architecture Overview

Starnix joins Fuchsia's Component Framework (Volume VI) as a **runner**. A runner is a component that starts other components of one kind. The `starnix` runner starts Linux programs. It implements the `fuchsia.component.runner/ComponentRunner` protocol. Component Manager sends it a `ComponentStartInfo`, which holds the program's resolved package, its `program` stanza, its namespace, and the server end of its outgoing directory. The runner then runs the binary, by turning it into Linux tasks inside a starnix kernel.

One kernel, many tasks. One starnix kernel runs many Linux processes and threads, as one Linux kernel does. Its central Rust object is the `Kernel`. The `Kernel` holds the pid table, the first mount namespace, the device registry, the namespaces for abstract sockets and System V IPC, the power and uptime state, and the services that all tasks share. The state of each process and thread lives in `ThreadGroup` and `Task` objects (§IV). Starnix does not schedule Linux threads. Each Linux thread is a Zircon thread, and Zircon's fair scheduler runs it like any other thread (Volume III). Where it can, Starnix translates a Linux `nice` value or scheduling class into a Zircon profile.

Capabilities Starnix holds. The starnix kernel is a component, so it holds only the Zircon capabilities routed to it. It holds a job in which to create processes and threads, and the VMAR and VMO rights to build Linux address spaces. It holds channels to the Fuchsia file systems mounted into the container. When graphics are enabled, it also holds handles to the magma and sysmem services (Volume I). A Linux process inside Starnix therefore has no more authority than its container was given. This holds even for a process running as Linux `root`: the capability rules of Volume III still decide what it can reach.

III. The Restricted-Mode Execution Loop

Every Linux thread in Starnix runs one loop. Volume III describes restricted mode in general; this section shows how Starnix uses it. When a task starts, Starnix calls `zx_restricted_bind_state`. Zircon creates a **restricted-state VMO**, binds it to the calling thread, and returns a handle to it. The VMO holds the Linux register state: the general purpose registers, the instruction pointer, the stack pointer and the thread pointer register (`fs_base` on x86-64, `TPIDR_EL0` on arm64). Starnix writes the Linux entry point and the initial stack pointer into that state. The stack already holds `argv`, `envp` and the ELF auxiliary vector. Starnix then calls `zx_restricted_enter`. It passes the address where the thread should resume in normal mode, which the syscall calls the *vector table*, and a context value that comes back in the first argument register.

Reason codes. When a thread leaves restricted mode, Zircon resumes it in normal mode at the vector table, with a *reason code* that says why (Table II). `ZX_RESTRICTED_REASON_SYSCALL` is the common case. `ZX_RESTRICTED_REASON_EXCEPTION` means the Linux code faulted and Zircon could not resolve the fault itself. Zircon places an exception report in the state VMO. Starnix turns the report into a Linux signal: `SIGSEGV` for a bad memory access, `SIGILL` for an illegal instruction, `SIGFPE` for a divide error. A page fault on a valid mapping that is not yet populated never reaches Starnix. Zircon pages it in and lets the Linux code continue, as it would for any other thread. `ZX_RESTRICTED_REASON_KICK` means that some thread called `zx_restricted_kick` on this one. Starnix kicks a thread when something needs its attention while it runs Linux code: a pending signal, a `ptrace` stop, or the thread's exit. A kick sent while the thread is already in normal mode is kept, and the next `zx_restricted_enter` returns at once with the same reason.

TABLE II
Restricted-mode exit reasons and Starnix's response.

Reason	Starnix action
`SYSCALL`	Decode the number and arguments, run the handler, write the result register.
`EXCEPTION` (page fault)	Zircon has already paged in any fault on a valid mapping; raise `SIGSEGV` or `SIGBUS` for the rest.
`EXCEPTION` (other)	Raise `SIGILL`, `SIGFPE` or `SIGTRAP`, according to the fault.
`KICK`	Check for deliverable signals, `ptrace` stops or exit; act, then re-enter.

Two kernel entries per Linux system call. Restricted mode is a mode of a user thread, but the CPU cannot switch between the two modes without the kernel. When Linux code executes a `syscall` instruction, the CPU traps into Zircon. Zircon sees that the thread is in restricted mode, saves its registers into the state VMO, and returns to normal mode at the vector table. When Starnix has handled the call, it calls `zx_restricted_enter`, which is a second Zircon system call, to go back. So each Linux system call costs two passes through the Zircon kernel, one Rust dispatch, and any Zircon system calls the handler makes, such as mapping memory or writing to a channel. Ordinary instructions cost nothing extra. Code that only computes runs at native speed; code that makes many system calls pays the round trip on each one. §XV returns to this cost.

IV. Task and Process Model

Linux draws only a loose line between a process and a thread, and Starnix keeps the same line. Four Rust types hold the state:

- `Kernel`: one for each starnix instance, in the role of the whole Linux kernel. It holds the pid allocator, the first mount namespace, the device registry and the shared subsystems.
- `ThreadGroup`: what Linux calls a *process*. Its tasks share a `tgid`, which is the pid that user programs see, as well as a memory manager, a file table, signal handlers, credentials, and a session and process group.
- `Task`: one Linux *thread*. It has a `tid`, a signal mask, the `set_child_tid` and `clear_child_tid` pointers, and a Zircon thread that runs it.
- `CurrentTask`: a reference to the running task, passed to every system call handler. It plays the part of `current` in Linux kernel code, and it gives the handler the registers and access to the Linux process's memory.

clone and fork. A Linux `clone()` call takes a word of flags. Each flag decides whether the child shares one resource with its parent or gets a copy of it (Table III). With `CLONE_THREAD`, Starnix makes a new `Task` in the same `ThreadGroup`, which is a new thread. Without it, Starnix makes a new `ThreadGroup`, which is a new process. With `CLONE_VM`, the child shares the parent's `MemoryManager`. Without it, Starnix gives the child a copy-on-write clone of the address space (§VI). Starnix supports the newer `clone3` call, with its `clone_args` structure, as well as the older `clone`, `fork` and `vfork`. For `vfork`, Starnix blocks

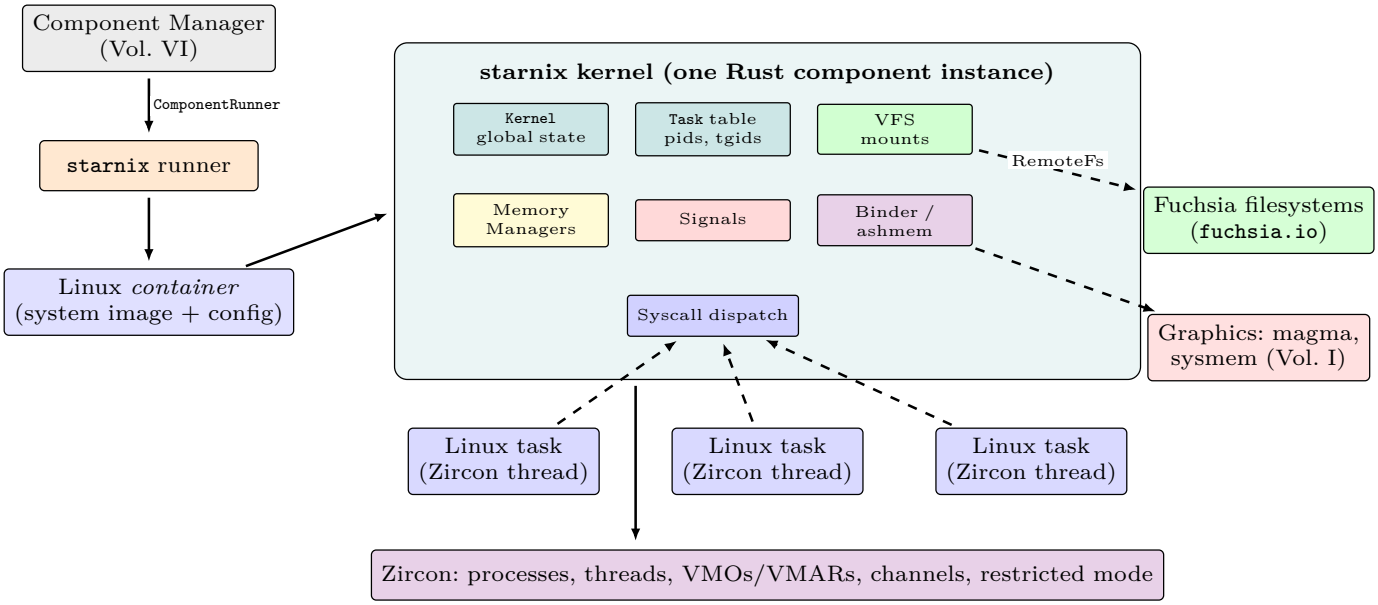


Fig. 2. Starnix architecture. The `starnix` runner launches a container. One starnix kernel component holds all of the container's Linux tasks and subsystems, builds them from Zircon objects, and reaches out to native Fuchsia file systems and the graphics stack.

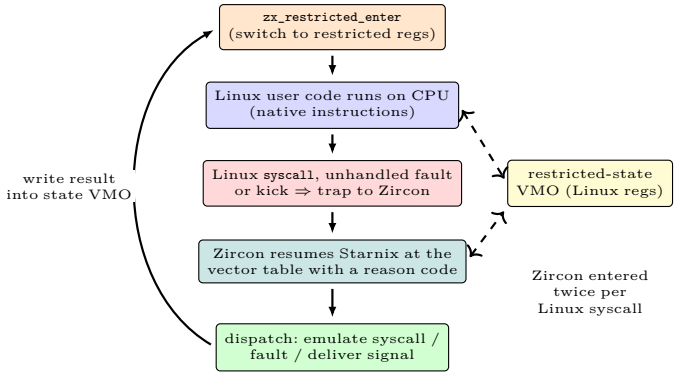


Fig. 3. The Starnix run loop. One Zircon thread alternates between Linux user code in restricted mode and the Starnix dispatcher in normal mode. The two exchange the Linux registers through the restricted-state VMO.

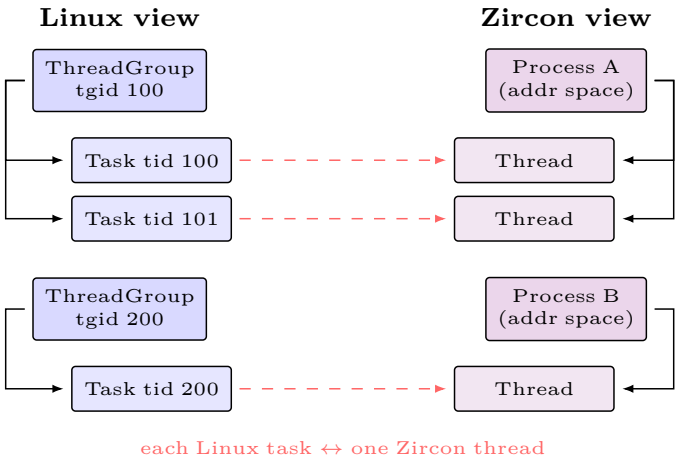


Fig. 4. Linux tasks mapped onto Zircon. Each `ThreadGroup` is one Zircon process with its own restricted region. Each `Task` is one Zircon thread.

TABLE III
SELECTED `CLONE()` FLAGS AND THEIR STARNIX EFFECT.

Flag	Effect in Starnix
`CLONE_THREAD`	New `Task` in the same `ThreadGroup` (shared tgid).
`CLONE_VM`	Share the `MemoryManager`; else COW-clone the address space.
`CLONE_FILES`	Share the file-descriptor table; else copy it.
`CLONE_FS`	Share `FsContext` (root, cwd, umask).
`CLONE_SIGHAND`	Share the signal-handler table.
`CLONE_VFORK`	Parent blocks until child execs/exits.
`CLONE_SETTLS`	Set the child's TLS/thread-pointer register.
`CLONE_NEW*`	Create a new namespace (pid, mount, net, uts, ipc, user).

the parent until the child calls `execve` or exits, so the usual pair of `vfork` and `execve` works without copying the address space.

`execve`. `execve` throws away the old address space and loads a new program. Starnix parses the ELF file and maps its `PT_LOAD` segments from the file's VMO (§VI). If the file names a dynamic linker in `PT_INTERP`, Starnix maps that as well, usually `ld.so`. It then maps the Linux vDSO and builds the initial stack. The stack holds `argv`, `envp` and the auxiliary vector: `AT_PHDR`, `AT_ENTRY`, `AT_RANDOM`, and `AT_SYSINFO_EHDR`, which points at the vDSO, among others. The thread then enters restricted mode at the dynamic linker's entry point. For a script that starts with `#!`, Starnix runs the interpreter that the first line names.

Exit and reaping work as they do on Linux. A task that has ended stays a zombie until its parent collects it with `wait4` or `waitid`, and the parent receives `SIGCHLD`. An orphaned process is handed to the container's `init`, or to the nearest ancestor that called `prctl(PR_SET_CHILD_SUBREAPER)`. The exit status records the exit code or the signal that ended the process, in the same encoding Linux uses.

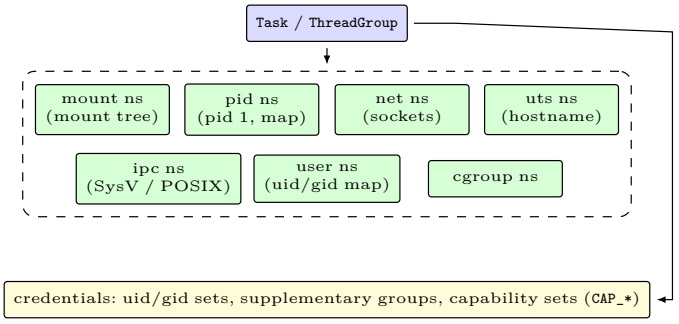


Fig. 5. A task refers to a set of namespaces and a set of credentials. `clone(CLONE_NEW*)`, `unshare` and `setns` create or switch the namespaces.

V. Namespaces and Credentials

Linux containers are built from *namespaces* and *credentials*. Android and container runtimes depend on both, so Starnix implements both.

Namespaces. Each `Task` refers to seven namespaces: mount, pid, network, UTS (the host and domain names), IPC, user and cgroup. `clone` with a `CLONE_NEW*` flag creates a new one, and `unshare` and `setns` create or switch them. Programs use the mount namespace most. It is a process's own view of the file tree (§VIII), so two containers in one starnix kernel can have entirely different roots. The pid namespace lets a container number its own processes, with its `init` as pid 1. Starnix keeps those numbers on top of the `Kernel`'s global pid table, so a task has a different pid at each level of namespace it belongs to.

Credentials. Each `ThreadGroup` holds a set of credentials. They are the real, effective, saved and file system uid and gid; the list of supplementary groups; and the five Linux *capability* sets: permitted, effective, inheritable, bounding and ambient. The capability sets hold bits such as `CAP_NET_ADMIN` and `CAP_SYS_ADMIN`, which allow privileged operations. `setuid`, `setgid` and `capset` change the credentials under the usual Linux rules, and the file system and network code check them. Linux `root`, uid 0, inside Starnix is not a Fuchsia administrator. It can use only the Zircon capabilities its container holds, so a Linux process with `CAP_SYS_ADMIN` still cannot reach outside the container.

VI. Memory Management

Each Linux process is one Zircon process, created with `zx_process_create_shared`. Such a process has two halves of address space. The upper half, the *shared region*, holds the starnix kernel's own code and data. It is the same in every Linux process of one container, and only normal mode can reach it. The lower half, the *restricted region*, belongs to that process alone and holds the Linux program. Linux programs expect to live near address zero, which is why the restricted region is the low one.

A `MemoryManager` describes one Linux address space. It keeps a tree of *memory regions*, the equivalent of Linux VMAs, and backs each one with a Zircon VMO mapped into the restricted region (Volume III). The Linux memory system calls translate directly:

- An anonymous `mmap` creates an anonymous VMO and maps it with `zx_vmar_map`, with the protections the caller asked for.
- A file `mmap` maps the file's VMO, often the same VMO that `RemoteFs` already holds, so no data is copied. With `MAP_SHARED`, writes go back to the file's VMO. With `MAP_PRIVATE`, the mapping gets a copy-on-write child VMO.
- `mprotect` becomes `zx_vmar_protect`.
- `munmap` becomes `zx_vmar_unmap` of the range, and Starnix splits memory regions as needed.
- `mremap` maps a region again, with a new size or at a new address.
- `brk` grows or shrinks an anonymous mapping.
- `madvise` with `MADV_DONTNEED` decommits pages with `zx_vmo_op_range`; `MADV_WILLNEED` faults pages in ahead of use.

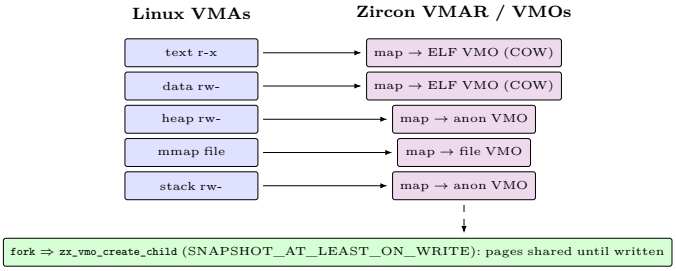


Fig. 6. A Linux address space is a set of VMO mappings in the restricted region. On `fork`, Starnix copies the map and shares the pages copy-on-write through child VMOs.

The rules for executable memory from Volume III still apply. To make a Linux mapping executable, Starnix needs a VMO with the Zircon execute right, as any Fuchsia program does.

Usercopy. System call handlers read and write the Linux program's memory: arguments behind pointers, `iovec` buffers and path strings. The handler runs in normal mode on the same Zircon thread, inside the same shared process, so the restricted region is in its own address space. Starnix reads and writes it with ordinary loads and stores, through copy routines that recover from a fault. These play the part of Linux's `copy_from_user` and `copy_to_user`. A bad pointer makes the call fail with `EFAULT`, and a path is copied with a length limit.

Stack growth and memory limits. The main thread's stack grows down, like a `MAP_GROWSDOWN` mapping. A fault just below it extends the mapping, up to the `RLIMIT_STACK` limit. Starnix enforces the `setrlimit` limits on memory and on file descriptors. When a Linux program goes past a limit, it sees the failures Linux would give: `ENOMEM`, or its `ThreadGroup` killed for running out of memory. Starnix also watches Fuchsia's memory pressure signals, which Volume VIII of the *Fuchsia Subsystems* series describes.

Futexes. Starnix implements Linux `futex()` on the Linux program's memory. A lock that nobody else holds costs one atomic compare in the Linux program and never reaches Starnix. Only a contended lock does, and then Starnix puts the waiting `Task`s to sleep and wakes them. The design follows the Zircon futex of Volume III, but the key is an address in the Linux address space. Starnix supports the `FUTEX_WAIT`, `FUTEX_WAKE`, `FUTEX_REQUEUE` and priority inheritance operations, and robust futex lists.

VII. System-Call Dispatch

When a thread returns with the `SYSCALL` reason, Starnix reads the Linux system call number from the saved registers: `rax` on x86-64, `x8` on arm64. It uses the number to index a **system call table**, one for each architecture. The arguments come from the registers each architecture's convention names: `rdi`, `rsi`, `rdx`,

TABLE IV
REPRESENTATIVE LINUX SYSCALL AREAS AND HOW STARNIX IMPLEMENTS THEM.

Area	Implementation
Process	`clone/fork/execve/exit/wait/prctl` over `Task/ThreadGroup`.
Memory	`mmap/mprotect/brk/madvise/futex` over VMOs/VMAR.
File I/O	`openat/read/write/close/statx/getdents64` through the VFS.
Signals	`rt_sigaction/sigprocmask/kill/tgkill/sigreturn/signalfd`.
Sockets	`socket/bind/connect/sendmsg/recvmsg` per family.
Polling	`epoll_create1/ctl/wait, poll, ppoll, eventfd`.
Time	`clock_gettime/nanosleep/timerfd`; vDSO fast paths.
ioctl	Per-device handlers (`TIOC*`, Binder, ashmem, DRM/magma).

`r10`, `r8` and `r9` on x86-64, and `x0` to `x5` on arm64. Each table entry is a typed Rust handler. The handler takes `&CurrentTask` and the decoded arguments, and returns a `Result` of either a value or an `Errno`. The dispatcher writes the value, or the negated `errno`, into the result register.

Listing 1. Shape of a Starnix syscall handler (illustrative).

```
// sys_* handlers receive the current task + decoded args.
pub fn sys_getpid(current: &CurrentTask) -> Result<pid_t, Errno> {
    Ok(current.thread_group.leader)
}

pub fn sys_write(current: &CurrentTask, fd: FdNumber,
                 addr: UserAddress, len: usize) -> Result<usize, Errno> {
    let file = current.files.get(fd)?;            // fd table lookup
    let buf  = current.read_memory_to_vec(addr, len)?; // copy_from_user
    file.write(current, &buf)                      // VFS write
}
// Dispatch: table[nr](current, args...) -> set result reg = ret or -errno
```

The dispatcher also reproduces four finer points of Linux. First, a handler can return `ERESTARTSYS`. If a signal interrupted the call, Starnix runs the signal handler and then retries the call, when the handler was installed with `SA_RESTART`. Second, a call such as `nanosleep` that must resume with changed arguments uses the `restart_block` mechanism. Third, 32-bit `i386` and `armhf` binaries go through a second table of `compat` entry points, which decode the narrower arguments. Fourth, Starnix runs `seccomp-bpf` filters before the handler, so a sandboxed Linux program is allowed, denied, given an error or trapped exactly as on Linux.

Some calls never reach the dispatcher. Starnix maps a Linux **vDSO** into each process, and the vDSO answers `clock_gettime`, `gettimeofday` and `getcpu` by reading a shared time page. These calls leave restricted mode no more than an ordinary function call does. The Zircon vDSO of Volume III serves time the same way.

VIII. THE VIRTUAL FILE SYSTEM

Almost everything in Linux is a file, so Starnix has a complete virtual file system (VFS) in the Linux style. Its main types match the Linux kernel's:

- `FileSystem`: one mounted file system, with a `FileSystemOps` that implements it.
- `FsNode`: an inode. It holds the type, mode, owner and size, and an `FsNodeOps` table of functions for lookups and metadata.
- `FileObject`: an open file description. It holds the offset, the status flags and a `FileOps` table. File descriptors point to it.

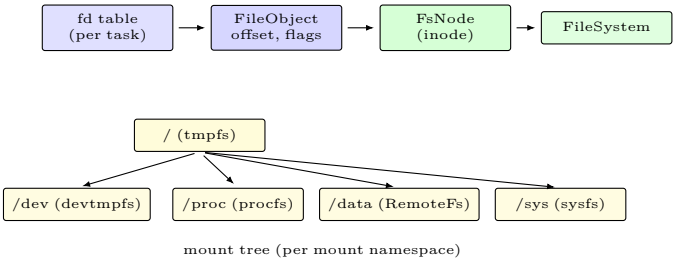


Fig. 7. The Starnix VFS objects and an example mount tree. The `/data` mount is a `RemoteFs` bridged to a Fuchsia file system.

- `DirEntry` and `NamespaceNode`: the path layer, which Linux calls dentries. They bind names to nodes within a mount and record the mount tree.
- `FsContext`: the root, current directory and umask of a process.

Each task has a file table, an `FdTable`. A file descriptor in it points to a `FileObject` and keeps the close-on-exec flag.

Path resolution walks the path one component at a time, from the task's root or current directory. At a mount point, the `DirEntry` leads to the root of another `FileSystem`. Starnix follows symbolic links up to the loop limit, and checks permissions and namespace boundaries at each step. This is the job of `path_lookup` in Linux. A cache of `DirEntry` objects remembers recent lookups of a name, so Starnix does not call `FsNodeOps::lookup` for them again. Starnix supports the `*at` family of calls such as `openat`, and the `O_PATH`, `O_NOFOLLOW` and `RESOLVE_*` flags.

Built-in file systems. Starnix has its own implementations of these:

- `tmpfs`, files in RAM backed by VMOs, the usual root of a container;
- `devtmpfs` for `/dev`, `procfs` for `/proc`, built from `Task` and `Kernel` state, and `sysfs` for `/sys`;
- `selinuxfs`, `cgroupfs`, `functionfs`, `configfs` and `fuse`;
- `overlayfs`, which stacks an upper directory on a lower one for layered Android and distribution images;
- a reader for `ext4` images.

`epoll`, `inotify`, `fanotify`, `eventfd`, `timerfd`, `pipe` and `memfd_create` all produce `FileObject`s too. They share one wait queue mechanism, so any of them can be passed to `poll` or `epoll`. The file system that ties Starnix to the rest of Fuchsia is `RemoteFs`.

IX. BRIDGING TO FUCHSIA: REMOTEFS AND FUCHSIA.IO

A Linux program expects POSIX files. A Fuchsia component offers directories and files through the `fuchsia.io` FIDL protocol (Volume XI covers FIDL). **RemoteFs** connects the two. When Starnix mounts a Fuchsia directory into a Linux mount namespace, RemoteFs wraps the directory's `fuchsia.io` handle in `FsNode`s. From then on, RemoteFs turns each Linux file operation on that mount into a `fuchsia.io` call over a Zircon channel (Table V).

Some Fuchsia files give out a VMO, as package files and blobs do. For those, RemoteFs maps the VMO directly. A Linux `mmap` of the file then shares pages with the Fuchsia blob, and so does the mapping of an ELF file for `execve`. The demand paging of Volume III fills in the pages on first touch. This lets a Linux process load its libraries and run its binaries straight from Fuchsia's content-addressed storage, without copying them

TABLE V
LINUX FILE OPERATIONS MAPPED TO FUCHSIA.IO BY REMOTEFS.

Linux operation	fuchsia.io call
openat	Directory.Open (new connection per node)
read/pread	File.Read/ReadAt, or read the file's VMO
write/pwrite	File.Write/WriteAt
lseek	File.Seek
getdents64	Directory.ReadDirents
stat/statx	Node.GetAttributes
mmap	File.GetBackingMemory → map the VMO

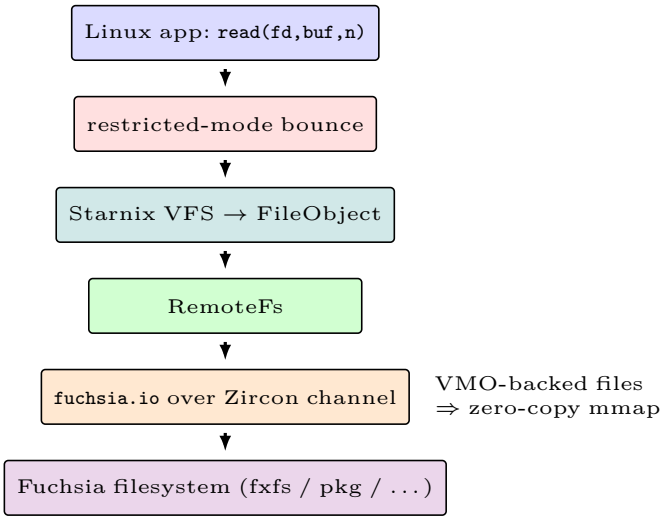


Fig. 8. A Linux `read()` on a bridged mount becomes a `fuchsia.io` call. When the file is backed by a VMO, a Linux `mmap` maps the same pages.

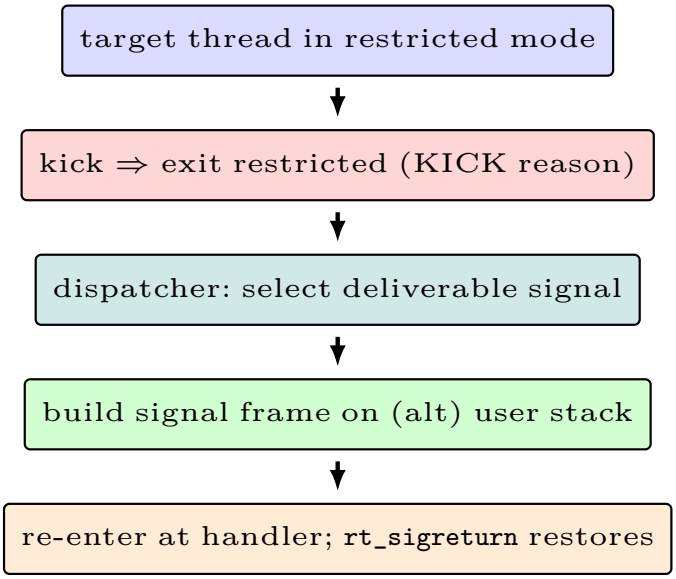


Fig. 9. Signal delivery. A kick brings the thread out of restricted mode. Starnix builds a Linux signal frame and re-enters the program at its handler.

TABLE VI
SOCKET FAMILIES AND THEIR STARNIX BACKENDS.

Family	Backend
AF_UNIX	In-process; stream / datagram / seqpacket + SCM_RIGHTS.
AF_INET[6]	Bridged to Fuchsia netstack via fuchsia.posix.socket.
AF_VSOCK	Host/guest virtio-vsock bridge.
AF_NETLINK	Synthesized route / uevent interfaces.
AF_PACKET	Limited; raw frames where supported.

(Volume VIII covers that storage). A container's writable state, such as `/data`, usually comes from `fxfs` through the same bridge.

X. SIGNALS

Starnix implements the whole Linux signal model. That covers the standard signals, numbers 1 to 31, and the real-time signals from `SIGRTMIN` to `SIGRTMAX`, for which Starnix queues each `siginfo`. Starnix keeps a set of pending signals for each task and for each thread group. It implements the blocked mask set by `sigprocmask`, the handlers set by `rt_sigaction`, and the default actions: terminate, dump core, stop, continue and ignore. It also implements alternate signal stacks, set by `sigaltstack`, and the `signalfd` and `sigwaitinfo` interfaces.

To deliver a signal, Starnix often has to interrupt a thread that is running Linux code in restricted mode. It does this with a **kick**: it calls `zx_restricted_kick`, and the thread returns to the dispatcher with the `KICK` reason. The dispatcher picks a signal that is pending and not blocked, choosing the target thread in the group by the Linux rules. If the program catches that signal, Starnix builds a Linux **signal frame** on the program's stack, or on the alternate stack if the handler was installed with `SA_ONSTACK`. The frame saves the interrupted registers, a `ucontext` and a `siginfo`. Starnix then points the program counter at the handler, puts the signal number in the first argument register, and sets the return address to the `rt_sigreturn` trampoline. When the handler returns, the trampoline calls `rt_sigreturn`, which removes the frame and restores the interrupted registers. A restartable system call that a signal interrupted is issued again if the handler has `SA_RESTART`, and otherwise returns `EINTR`.

XI. NETWORKING

A socket is a `FileObject`. Starnix picks its implementation by address family (Table VI).

- `AF_UNIX` sockets live entirely inside Starnix. They connect Linux tasks directly, in stream, datagram and seqpacket modes. Starnix also handles abstract namespace sockets, and passes file descriptors sent with `SCM_RIGHTS` through its VFS.
- `AF_VSOCK` sockets connect to the host.
- `AF_NETLINK` sockets offer the `NETLINK_ROUTE` and `uevent` interfaces that Linux tools and Android's `netd` expect. Starnix builds the answers itself.
- `AF_INET` and `AF_INET6` sockets go to the Fuchsia network stack, through the `fuchsia.posix.socket` protocols (Volume X). A Linux TCP connection therefore uses Fuchsia's netstack; there is no second IP stack inside Starnix. Starnix updates the Linux socket from the events on the Fuchsia socket.

`poll`, `epoll` and `select` work on all of these through Starnix's wait queues, which are built on Zircon signals and ports (Volume III).

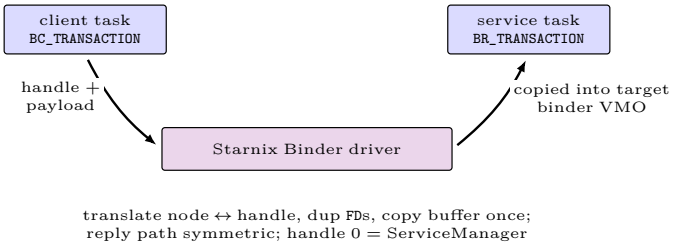


Fig. 10. An Android Binder transaction. The Starnix Binder driver translates object references, duplicates the file descriptors passed, and copies the buffer once into the target's binder region.

XII. Android Support: Binder, ashmem, and the GPU

Android is the main reason Starnix exists, and Android needs more than POSIX. Its middleware is built on a few Linux drivers, and Starnix reimplements them inside the starnix kernel. Binder matters most.

A. The Binder driver

Android's IPC goes through the `/dev/binder` character device, and through its siblings `hwbinder` and `vndbinder`. Each process maps a binder region with `mmap`; in Starnix the region is a VMO. The driver copies incoming transactions into that region. A process talks to the driver with the `BINDER_WRITE_READ` `ioctl`. The call passes a *write* buffer of `BC_*` commands, such as `BC_TRANSACTION`, `BC_REPLY`, `BC_FREE_BUFFER`, `BC_ACQUIRE`, `BC_RELEASE` and `BC_ENTER_LOOPER`. It also passes a *read* buffer, which the driver fills with `BR_*` replies such as `BR_TRANSACTION`, `BR_REPLY`, `BR_TRANSACTION_COMPLETE` and `BR_DEAD_BINDER`.

The driver tracks three kinds of thing. A **binder node** stands for an object that a process exports, and lives in that process. A **binder ref** is another process's reference to a node. A **handle** is the integer by which a client names a ref. The driver's main job is to *translate references*. Each object in a transaction is described by a `flat_binder_object` entry, and the driver rewrites each entry from the sender's form to the receiver's:

- A `BINDER_TYPE_BINDER`, where the owner passes its own node, becomes a `BINDER_TYPE_HANDLE` in the receiver.
- A `BINDER_TYPE_HANDLE` that names the receiver's own node becomes a `BINDER_TYPE_BINDER`.
- A `BINDER_TYPE_FD` makes the driver duplicate the file descriptor into the receiver, through the VFS.

The driver then reserves space in the *target's* binder region and copies the buffer there once, gathering its pieces if it has several. It posts a `BR_TRANSACTION` to a thread in the target's binder thread pool. The reply comes back the same way. Handle 0 is special: it names the **context manager**, Android's `ServiceManager`, which registers with `BINDER_SET_CONTEXT_MGR`. A client that sends `BC_REQUEST_DEATH_NOTIFICATION` receives `BR_DEAD_BINDER` when the service's process exits. Starnix implements all of this on its own task and VFS state. Android's `ServiceManager`, its HALs, its `binder` libraries and its apps make calls exactly as they would on Linux.

B. ashmem and shared memory

Android shared memory, `/dev/ashmem`, is built on Zircon VMOs. The `ASHMEM_SET_SIZE` and `ASHMEM_SET_NAME` `ioctl`s set up a VMO, and `mmap` shares it between processes. `ASHMEM_UNPIN` marks a range that the system may discard under memory pressure, and `ASHMEM_PIN` claims it back. Android uses ashmem for `gralloc` buffers and for the shared regions of `libcutils`.

C. GPU, graphics, and input

Starnix does not emulate the GPU; it passes GPU work through. The Linux or Android Vulkan or GLES driver talks to Fuchsia's **magma** interface (Volume I) through a device node that Starnix provides. Graphics buffers are allocated as sysmem buffer collections and shared with the Fuchsia compositor, so a frame the Linux program renders reaches the compositor without a copy. Input events and the display reach the Linux program through more device nodes and a Wayland bridge. A graphical Linux or Android app therefore presents through the same display path that Volume I describes. The split follows a simple rule. Binder and ashmem are pure software interfaces, so Starnix reimplements them. The GPU is a real device that Fuchsia already drives, so Starnix passes work through to it.

XIII. Devices, Containers, and Boot

Devices. Starnix keeps a registry of devices by major and minor number, and fills `/dev` from it through `devtmpfs`. The character devices include:

- `tty` and `pty`, with a complete terminal and line discipline layer and `/dev/pts`;
- `null`, `zero`, `full`, `random` and `urandom`;
- `loop` and `fuse`;
- the Android devices, Binder and ashmem;
- the device nodes that pass through to Fuchsia graphics and input.

An `ioctl` on a device goes to that device's handler. Starnix sends `uevent`s over `AF_NETLINK`, so Android's `ueventd` and `udev`-style tools see devices arrive and leave.

Containers and boot. A container is a component that names three things: its Linux system image; a set of *features*, such as `android`, `selinux`, `network` or `magma`; and its *mounts*. It also gives the command line for the Linux `init`. Starting a container takes four steps (Fig. 11). First, Starnix creates the `Kernel`. Second, it builds the first mount namespace: a root of `tmpfs` or an overlay, then `/dev`, `/proc` and `/sys`, then a `RemoteFs` mount for each Fuchsia directory routed to the component. Third, it enables the features asked for, and loads the SELinux policy if there is one. Fourth, it starts `init` as pid 1, and `init` boots the rest of Linux user space. The `starnix` runner can also run a single Linux program as a one-shot component. This is how `fx test` runs Linux tests.

XIV. Diagnostics, /proc, and ptrace

Linux tools read `/proc` and `/sys`, so Starnix builds both from the live state of its `Task`, `ThreadGroup` and `Kernel` objects:

- `/proc/<pid>/maps` lists the `MemoryManager`'s regions, and `/proc/<pid>/fd` the file table.
- `/proc/<pid>/status` and `/proc/<pid>/stat` show the task's details.
- `/proc/<pid>/cmdline` and `/proc/<pid>/environ` show how the process started.
- `/proc/cpuinfo`, `/proc/meminfo` and `/proc/mounts` describe the machine and the container.

Starnix also implements `ptrace`: attaching, stops, reading and writing registers and memory, single stepping, stops at system calls, and the `PTRACE_*` options. So Linux `strace` and `gdb` work on processes inside Starnix. To stop a traced thread, Starnix

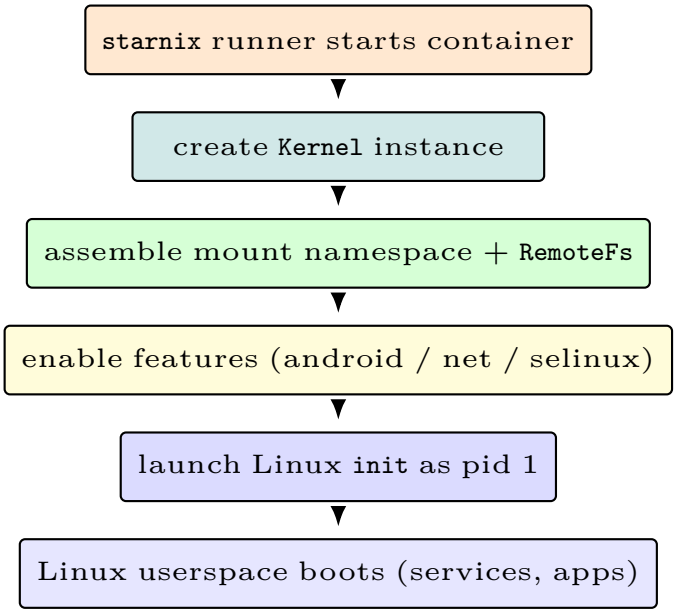


Fig. 11. Container boot, from the `starnix` runner to a running Linux `init` and user space.

kicks it out of restricted mode, as it does to deliver a signal. On the Fuchsia side, the starnix kernel publishes Inspect data and trace events (Volume XIII). The time spent switching modes and handling each system call appears in the same Perfetto timeline as the rest of the system. A crash in a Linux program can be examined with the native Fuchsia tools as well.

XV. Performance and Limitations

Cost model. Say a program executes N_i instructions and makes N_s Linux system calls. Restricted mode runs on the real CPU, not in an interpreter, so the N_i instructions run at native speed. The extra cost is all in the N_s calls. Each call costs a trap into Zircon and a return to normal mode, then a Rust dispatch. Then comes the handler's own work, including any Zircon calls it makes, and a `zx_restricted_enter` to go back:

$$T \approx t_{\text{native}}(N_i) + N_s\,(t_{\text{switch}} + t_{\text{dispatch}} + t_{\text{handler}}).$$

Here t_{switch} covers both passes through Zircon. RFC-0261 expects this switch to cost more than a native system call, because the CPU passes through the kernel in each direction, and it expects the difference to matter little for Starnix. Three things follow. Code that mostly computes, including the just-in-time compiled code of the Android runtime, runs close to native speed. Tight loops of `getpid` or `read` show the switch cost most clearly. The vDSO removes the switch altogether for the most frequent time calls. Mapping files without a copy (§IX) and copying a Binder buffer only once (§XII) keep the cost of moving data low.

No second scheduler and no second MMU. Linux threads are Zircon threads, so Zircon's fair scheduler (Volume III) schedules Linux and Fuchsia work together, with no scheduler inside another. Linux memory is made of VMOs mapped into a real address space, so the hardware MMU translates Linux addresses directly. There is no shadow page table and no software walk of a guest page table. These two facts are why Starnix costs less than a virtual machine.

Limitations. Starnix implements the interface Linux offers to programs, not the one it offers to its own modules. Loadable kernel modules, drivers that program hardware directly, and behaviour that depends on the internals of one Linux version are out of scope. Starnix covers much of the Linux system call, `ioctl` and `/proc` surface, but not all of it, and the coverage keeps growing. A program that probes a rarely used corner may get `ENOSYS` or `EINVAL`. Real-time scheduling and `SCHED_DEADLINE` work only as far as Zircon profiles can express them. None of these limits weakens security. A Linux program is confined to its container's Zircon capabilities, whatever it believes about being `root`.

XVI. Putting It Together: Life of a Linux Process

This section follows one Android or Linux program from its launch to a system call and back, and names the part of Starnix that handles each step.

1) **Launch.** Component Manager resolves the program's component and calls `ComponentRunner.Start` on the `starnix` runner. The runner asks the container's starnix kernel to create a new `ThreadGroup` (§IV).
2) **Address space.** Starnix builds a `MemoryManager`. It maps the ELF's `PT_LOAD` segments from the program's VMO, which `RemoteFs` got from the Fuchsia package without a copy (§IX, §VI). It maps `ld.so` and the Linux vDSO, and lays out the initial stack with the auxiliary vector.
3) **First entry.** Starnix binds a restricted-state VMO, writes the dynamic linker's entry point and the stack pointer into it, and calls `zx_restricted_enter`. `ld.so` runs natively and loads the program's libraries. Each `openat` and `mmap` it makes is one round trip to Starnix.
4) **A system call.** The program calls `write(2)`. The CPU traps into Zircon, which returns the thread to normal mode with the `SYSCALL` reason (§III). The dispatcher reads the registers, looks up the file descriptor in the `FdTable`, copies the buffer out of the program's memory, and calls `write` on the `FileObject` (§VIII). If the file is on a bridged mount, RemoteFs sends a `fuchsia.io File.Write` over a channel. Starnix writes the result into the result register and enters restricted mode again.
5) **An IPC call.** In an Android app, a method call on a remote object becomes a Binder `BC_TRANSACTION` `ioctl`. The Binder driver translates the references, copies the payload once into the service's binder VMO, and wakes a service thread (§XII).
6) **A signal.** Another task sends `SIGTERM`. Starnix kicks the thread out of restricted mode, builds a signal frame, and enters the program again at its handler (§X).
7) **Exit.** On `exit_group`, Starnix takes the `ThreadGroup` apart. It unmaps its VMOs, closes its files, which releases the Zircon handles behind them, sends `SIGCHLD` to the parent, and tells the component framework that the program has ended.

Each step uses only primitives from Volume III: VMOs and VMARs for memory, threads for tasks, channels for files, and restricted mode for running the program. Only the GPU path reaches into Volume I.

XVII. Conclusion

Starnix runs Linux programs on Fuchsia without a second scheduler and without an emulated MMU. Zircon's restricted mode makes this possible. Linux user code runs natively on the CPU. The Linux kernel is replaced by a Rust program that implements the system calls, the file system, signals, namespaces and the Android drivers. It builds all of them from the Zircon

primitives of Volume III. VMOs and VMARs become Linux address spaces. Channels and `fuchsia.io` become Linux files. Zircon threads become Linux tasks, and Zircon's scheduler runs all of them. The cost is that of any user-space kernel. Instructions run at native speed, and each Linux system call adds two passes through Zircon and a dispatch, plus whatever Zircon calls the handler makes.

The hardware analogies used across this series apply here too. The switch out of restricted mode on a system call is a trap and emulate handshake. The Binder driver is a message router that rewrites capability references as it forwards them, like an address translation table on a bus. RemoteFs is a protocol adapter between two file interconnects. Starnix builds on Volume III and uses Volume I for graphics.

References

[1] Google, "Starnix," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/components/v2/starnix

[2] Google, "RFC-0261: Fast and efficient user space kernel emulation," *Fuchsia RFCs*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0261_fast_and_efficient_user_space_kernel_emulation

[3] Google, "`zx_restricted_enter`," *Zircon system calls*, 2026. [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/syscalls/restricted_enter

[4] Google, "`src/starnix` source tree," *Fuchsia*, 2024. [Online]. Available: https://cs.opensource.google/fuchsia/fuchsia/+/main:src/starnix/

[5] Google, "`fuchsia.io` protocol," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/reference/fidl/fuchsia.io

[6] Google, "Component runners (`fuchsia.component.runner`)," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/runner

[7] Android Open Source Project, "Binder IPC," 2024. [Online]. Available: https://source.android.com/docs/core/architecture/hidl/binder-ipc

[8] Fuchsia Internals Series, "Volume III: The Zircon Kernel," 2026.

[9] Fuchsia Internals Series, "Volume I: Graphics & Display Pipeline," 2026.

[10] Google, "Component Framework," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/components/v2