哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

AstrancE 操作系统设计与实现

曾熙晨　滕奇勋　杨纯懿
哈尔滨工业大学（深圳）

2025 年 6 月 30 日

目录

摘要：项目概览

项目核心

AstrancE：基于 Rust 的模块化宏内核操作系统。

- **目标**：突破 ArceOS 在用户态支持和系统调用方面的限制。
- **继承**： ArceOS 的模块化设计和内存安全优势。
- **创新**：从微内核扩展为支持完整用户态交互的宏内核系统。
- **成果**：显著提升系统功能完整性和实用性。

核心转变

引入 `axsyscall` 与 `axmono` 模块，为传统 C/Rust 用户程序提供运行接口。**重点优化**：进程管理、内存管理、文件系统、信号机制、用户态支持。

截至 6 月 29 日，取得 14 名的成绩。

摘要：各模块完成情况

表: 各模块完成情况概览

模块	状态	关键进展
进程管理	基本完成	用户态 ELF 加载、独立虚拟地址空间、用户态线程创建与调度。
内存管理	基本完成	COW、懒分配、多段映射、RAII 风格页表、mmap 跨区域、共享内存。
文件系统	基本完成	procfs（动态系统信息）、动态挂载、devfs 扩展（设备节点）。
信号机制	基本完成	POSIX 信号全流程、备用信号栈、sigreturn_trampoline、与调度器联动。
用户态支持	基本完成	宏内核式 syscall 分发、统一用户态运行环境、独立虚拟地址空间隔离。

内核开发背景

项目背景：基于 ArceOS

本项目基于 **ArceOS** 开展开发工作。ArceOS 是一个采用 Rust 语言编写的实验性模块化操作系统（或单内核），其核心设计理念强调模块化与安全性。

ArceOS 核心特点

- **语言优势**： Rust 语言的内存安全特性，规避常见安全漏洞。
- **模块化设计**：系统架构细致划分为独立功能模块（Crate 形式），便于定制与维护。
- **跨平台支持**：良好硬件抽象层（`axhal`）与驱动模块（`axdriver`）。
- **异步并发**：高效任务调度与同步机制（`axsync`, `axtask`）。
- **POSIX 兼容**： `arceos\posix_api` 提供一定程度兼容性。

- **模块示例**：内存与驱动（`axalloc`, `axdriver`, `axhal`）、文件系统（`axfs`）、基础组件（`axsync`, `axtask`, `axlog`）。

ArceOS 局限与项目创新

- **ArceOS 现有局限**：
  - **功能最小化**：许多模块仅实现基本功能。
  - **设备支持不足**：文件系统仅支持抽象设备（Null, Zero）。
  - **核心组件待完善**：信号处理、页表操作、系统调用、网络协议栈等。

本项目目标

基于 ArceOS 框架，推进系统功能完善与增强：

- **重点优化**：内存管理、信号处理、文件系统模块。
- **核心增强**：用户态程序支持能力。
- **开发理念**：围绕“功能完整性”，利用 ArceOS 模块化优势。

- **核心创新与价值**：
  - **架构转换**：引入 `axsyscall` 和 `axmono`，实现 **微内核** → **宏内核**转换。
  - **用户态支持**：为传统用户态应用提供完整系统调用接口。
  - **平台优势**： ArceOS 对 RISC-V 和 LoongArch 等多架构的良好支持。

内核总体设计架构概述

AstrancE 总体设计架构概述

AstrancE 操作系统采用模块清晰、抽象统一的设计原则，其五大核心子系统协同支撑调度、内存、文件、信号与用户态功能。

核心特性

具备高可移植性、可维护性与良好的拓展价值。

(a) **进程管理：用户态程序支持与线程管理**基于 ArceOS 的 `axtask` 模块，新增用户态程序支持功能。

- 通过 `axmono` 实现 ELF 文件加载，支持独立虚拟地址空间运行。
- 完善线程支持，通过修改 `axmm` 模块帧管理机制，支持用户态多线程。
- 扩展用户态程序生命周期管理。

AstrancE 总体设计架构概述 (续)

(b) **内存管理：跨区域映射与共享内存**以 `AddrSpace` 为核心，实现完整虚拟地址空间管理。

- 新增跨区域内存映射（`mmap`）支持。
- 支持写时复制（COW）机制，优化内存使用效率。
- 实现共享内存映射功能，支持进程间共享内存。
- 引入 RAII 思想重写内存区域和页表管理，提升内存安全性和健壮性。
- 支持内存保护、页面错误处理等高级特性。

AstrancE 总体设计架构概述 (续)

(c) **文件系统**：**VFS 框架与动态挂载**基于统一的虚拟文件系统（VFS）接口，实现模块化文件系统架构。

- 新增 `procfs` 支持，实现动态生成系统信息文件。
- 实现动态挂载机制，支持将不同文件系统挂载到指定目录。
- 扩展 `devfs` 功能，支持设备文件动态创建和管理。
- 通过 `axfs_crates` 模块实现文件系统可插拔设计。

AstrancE 总体设计架构概述 (续)

(d) **信号机制：POSIX 兼容的异步通知**实现完整的 POSIX 信号处理机制。

- 包括信号注册、屏蔽、触发、处理的全流程。
- 通过 `SignalContext` 管理信号处理表、阻塞信号集和待处理信号集。
- 新增默认信号处理器，支持常用信号默认处理行为。
- 支持备用信号栈和上下文恢复（`sigreturn_trampoline` 机制）。
- 信号机制与调度器深度融合，确保异步通知的及时可靠性。

AstrancE 总体设计架构概述 (续)

(e) **用户态支持：系统调用与进程隔离**通过 `axsyscall` 模块实现宏内核架构系统调用分发机制。

- 支持超过 50 个 POSIX 系统调用，涵盖文件操作、进程管理、内存管理等。
- 新增 `axmono` 模块，提供统一接口支持（ELF 加载器、进程创建、多线程）。
- 实现独立虚拟地址空间管理，通过 `copy_from_kernel` 确保用户态与内核态隔离。
- 支持 C 和 Rust 两种开发语言，提供完整的 POSIX API 兼容性。

项目的技术特色

本项目技术特色主要体现在以下几个方面：

1. **架构转换创新**：成功从 ArceOS 微内核向宏内核转换，保持模块化优势并提供完整系统调用支持。
2. **内存管理增强**：实现跨区域映射、共享内存等高级功能，引入 RAII 提升内存安全性和健壮性。
3. **文件系统扩展**：新增 `procfs` 支持，实现动态挂载和文件系统可插拔机制。
4. **信号处理完善**：实现完整的 POSIX 信号机制，包括默认信号处理器，提升异步通知能力和稳定性。
5. **用户态支持**：通过 `axmono` 模块提供完整用户态程序运行环境，支持 ELF 加载、进程与线程管理。
6. **构建工具创新**：开发 `acbat` 模块，实现批量 ELF 文件处理和链接脚本生成，优化开发流程。

团队贡献概述

本开发团队成员在以下关键领域做出主要贡献：

- **系统调用实现**：初始提交实现基本系统调用框架。
- **内存管理优化**：修复跨区域 `mmap` 问题，优化内存映射对齐，引入 RAII 重构内存区域和页表管理。
- **线程支持**：新增用户态线程支持功能。
- **信号处理**：实现默认信号处理器，完善信号处理机制。
- **构建工具**：开发 `acbat` 模块，用于批量处理 ELF 文件。
- **用户态程序支持**：实现加载程序并在用户态运行的基本功能。

成果

这些改进使得 AstrancE 在保持 ArceOS 模块化优势的基础上，具备了运行传统用户态应用的能力，为操作系统的实际应用提供了坚实的基础。

个人贡献分工

曾熙晨：

- 主导内存管理优化，包括修复跨区域 `mmap` 问题（`3b693df`），完善内存映射对齐、`brk` 机制、默认信号处理器等（`060c8d8`）。
- 实现用户态线程支持（`9227965`），扩展系统并发能力。
- 维护和更新依赖（如 `axns`、`lwext4_rust`），保障主干代码可用性和兼容性。
- 参与系统日志、构建脚本、依赖管理等基础设施维护（如 `d25a4dc`、`d23c8f8`）。

个人贡献分工 (续)

滕奇勋:

- 主要负责分支管理、代码合并与集成，推动多项功能主干同步。
- 参与资源 API 和文件/目录 inode xattr 支持的开发与合并（1b72401）。
- 参与系统调用相关模块的调试与维护。

杨纯懿:

- 负责 futex（用户态同步原语）功能的实现（7461639）。
- 参与 busybox 兼容性修复、文件系统接口完善（baae0c3）。
- 参与 lua 支持、用户态接口完善、代码合并与分支管理。
- 参与 axmono 用户态支持相关代码的开发与维护（如 435472a）。

内存管理

物理与虚拟内存管理

AstrancE 采用全局分配器 axalloc 实现物理页帧的高效分配与回收，结合 Rust 的 RAII 机制，自动管理内存生命周期，避免泄漏。每个进程拥有独立的虚拟地址空间（AddrSpace），支持多区域映射和权限隔离。

```rust
pub struct FrameTrackerImpl {
    pub pa: PhysAddr,
    tracking: bool,
}
impl Drop for FrameTrackerImpl {
    fn drop(&mut self) {
        self.dealloc_frame();
    }
}
```

通过 Rust 所有权系统，FrameTrackerImpl 生命周期自动管理，保障物理内存安全。

写时复制（COW）与懒分配

为优化进程复制和内存共享，系统实现了高效的写时复制（COW）机制。多个地址空间可共享同一物理页，写入时自动分离。懒分配策略下，内存区域首次访问才分配物理页，提升资源利用率。

```
pub(crate) fn handle_page_fault_cow(
    vaddr: VirtAddr,
    orig_flags: MappingFlags,
    aspace: &mut AddrSpace,
) -> bool {
    if !orig_flags.contains(MappingFlags::WRITE) { return false; }
    if let Ok((_, pte_flag, _)) = aspace.pt.query(vaddr) {
        if !pte_flag.contains(MappingFlags::COW) { return false; }
    } else { return false; }
    // ... 省略 ...
}
```

该机制保证了进程间数据隔离与高效内存复用。

mmap 与共享内存

系统支持 Linux 风格的 mmap，允许用户空间灵活绑定匿名内存或文件映射。支持跨区域映射、共享内存、权限与对齐优化，提升了安全性和兼容性。

```
pub fn mmap(
    &mut self,
    start: VirtAddr,
    size: usize,
    perm: MmapPerm,
    flags: MmapFlags,
    mmap_io: Arc<dyn MmapIO>,
    populate: bool,
) -> AxResult<VirtAddr> { /* ... */ }
```

通过 RAII 机制，内存资源分配与释放自动化，提升系统健壮性。

进程与线程管理

模块化进程/线程管理

AstrancE 采用三层解耦架构：axtask（调度）、axprocess（进程/线程树管理）、axmono（用户态扩展）。用户态进程/线程建模、资源隔离、信号处理、系统调用、futex 等功能均由 axmono 实现。

```
pub struct TaskExt {
    pub time: RefCell<time::TimeStat>,
    pub thread: Arc<Thread>,
}
impl TaskExt {
    pub fn thread_data(&self) -> &ThreadData {
        self.thread.data().unwrap()
    }
    pub fn process_data(&self) -> &ProcessData {
        self.thread.process().data().unwrap()
    }
}
```

TaskExt 结构体实现了内核任务与用户态线程/进程的桥接。

用户进程/线程生命周期

用户进程/线程的创建流程包括：axtask 任务创建、页表配置、ProcessData/ThreadData 构建与 axprocess 管理、TaskExt 绑定。

```rust
pub fn spawn_user_task(
    exe_path: &str,
    aspace: Arc<Mutex<AddrSpace>>,
    uctx: UspaceContext,
    pwd: String,
    parent: Option<Arc<Process>>,
) -> AxTaskRef {
    let mut task = spawn_user_task_inner(exe_path, uctx, pwd, None);
    task.ctx_mut().set_page_table_root(aspace.lock().page_table_root());
    let tid = task.id().as_u64() as Pid;
    let process_data = ProcessData::new(exe_path.into(), aspace, spawn_signal_ctx(), None);
    let parent = parent.unwrap_or(init_proc());
    let process = parent.fork(tid).data(process_data).build();
    let thread_data = ThreadData { /* ... */ };
    let thread = process.new_thread(tid).data(thread_data).build();
    task.init_task_ext(TaskExt::new(thread));
    task.task_ext().process_data().ns_init_new();
    task.into_arc()
}
```

创新点与实际成效

独立实现用户态进程/线程建模、信号、系统调用、同步等关键能力。TaskExt 桥接三大核心模块，用户态扩展集中管理，便于维护与拓展。

```
pub struct ProcessData {
    pub exe_path: RwLock<String>,
    pub aspace: Arc<Mutex<AddrSpace>>,
    pub ns: AxNamespace,
    pub child_exit_wq: WaitQueue,
    pub exit_signal: Option<Signal>,
    pub signal: Arc<Mutex<SignalContext>>,
    pub signal_stack: Box<[u8; 4096]>,
}
pub struct ThreadData {
    pub clear_child_tid: AtomicUsize,
    pub signal: Arc<Mutex<SignalContext>>,
}
```

结构清晰、模块解耦、功能丰富、易于扩展。

虚拟文件系统（VFS）

VFS 架构设计

AstrancE 的 VFS 设计目标是实现“统一抽象、灵活扩展、透明访问”。支持多后端文件系统（FAT/EXT4/RAMFS/DEVFS/PROCFS），所有节点统一建模为 `VfsNode`，trait 多态分发，路径解析、权限管理、属性查询等高度解耦。

```
pub trait VfsOps: Send + Sync {
    fn mount(&self, _path: &str, _mount_point: VfsNodeRef) -> VfsResult { ... }
    fn umount(&self) -> VfsResult { ... }
    fn format(&self) -> VfsResult { ... }
    fn statfs(&self, _path: *const c_char, fs_info: *mut FileSystemInfo) -> VfsResult<usize> { ... }
    fn root_dir(&self) -> VfsNodeRef;
}
```

通过 trait 机制实现多态分发，便于后端文件系统灵活接入与替换。

VFS 节点操作与属性

所有文件系统节点统一建模为 VfsNode，其属性通过 VfsNodeAttr 结构体描述，支持标准权限、类型、大小、时间戳等字段。

```rust
pub trait VfsNodeOps: Send + Sync {
    fn open(&self) -> VfsResult { ... }
    fn release(&self) -> VfsResult { ... }
    fn get_attr(&self) -> VfsResult<VfsNodeAttr> { ... }
    fn read_at(&self, _offset: u64, _buf: &mut [u8]) -> VfsResult<usize> { ... }
    fn write_at(&self, _offset: u64, _buf: &[u8]) -> VfsResult<usize> { ... }
    // ... 省略 ...
}
#[derive(Debug, Clone, Copy)]
pub struct VfsNodeAttr {
    dev: u64, mode: VfsNodePerm, ty: VfsNodeType, size: u64,
    blocks: u64, st_ino: u64, nlink: u32, uid: u32, gid: u32,
    atime: u32, ctime: u32, mtime: u32,
}
```

属性字段丰富，兼容多种后端文件系统需求。

procfs 与 devfs 扩展

procfs 支持以文件系统方式导出内核与系统运行时信息，节点可在运行时动态生成。
devfs 支持设备节点的动态注册与管理，支持多级目录和自定义设备。

```rust
pub struct ProcFileSystem {
    parent: Once<VfsNodeRef>,
    root: Arc<ProcDir>,
}
impl VfsOps for ProcFileSystem {
    fn mount(&self, _path: &str, mount_point: VfsNodeRef) -> VfsResult { ... }
    fn root_dir(&self) -> VfsNodeRef { self.root.clone() }
}
pub struct DeviceFileSystem {
    parent: Once<VfsNodeRef>,
    root: Arc<DirNode>,
}
impl DeviceFileSystem {
    pub fn add(&self, name: &'static str, node: Arc<dyn VfsNodeOps>) { self.root.add(name, node); }
    pub fn mkdir(&self, name: &'static str) -> Arc<DirNode> { self.root.mkdir(name) }
}
```

路径查找采用最长前缀匹配，支持动态挂载与多后端协同。

信号子系统

信号机制设计

```rust
#[repr(usize)]
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
pub enum Signal {
    NONE = 0,
    SIGHUP = 1,
    SIGINT = 2,
    // ...
    SIGKILL = 9,
    SIGUSR1 = 10,
    SIGSEGV = 11,
    // ...
}
```

AstrancE 信号子系统完全自主设计，兼容 POSIX，采用 Rust 枚举实现信号类型，类型安全且易扩展。

支持信号注册、屏蔽、递送、处理、上下文恢复等全流程。

信号关键结构

```
bitflags! {
    pub struct SignalSet: u64 {
        const SIGHUP = 1 << 0;
        const SIGINT = 1 << 1;
        // ...
    }
}
pub enum SigHandler {
    Ignore,
    Handler(unsafe extern "C" fn(i32)),
    Default(fn(Signal, &mut SignalContext)),
}
pub struct SigAction {
    pub handler: SigHandler,
    pub mask: SignalSet,
    pub flags: SigFlags,
}
```

信号集合采用 bitflags 实现，支持高效批量操作。每个信号可独立配置处理动作，支持默认、忽略、自定义函数。

结构设计保证线程隔离和生命周期管理。

信号递送与处理流程

```
impl SignalContext {
    pub fn send_signal(&mut self, sig: Signal) {
        if !self.pending.contains(sig) {
            self.pending =
            ↪ self.pending.union(sig);
        }
    }
}
pub fn handle_pending_signals() {
    while ctx.has_pending() {
        let sig = ctx.pending.get_one().unwrap();
        let sig_action = &ctx.handlers[sig as
        ↪ usize];
        match sig_action.handler {
            SigHandler::Default =>
            ↪ handle_default_signal(sig, regs),
            SigHandler::Ignore => {},
            SigHandler::Handler(handler) => {
                unsafe { enter_signal_handler(&mut
                ↪ ctx, curr_sp, handler, sig) };
```

信号通过 ‘send_signal‘ 递送到目标线程/进程，‘handle_pending_signals‘ 主循环处理所有待处理信号。

支持备用栈、trap/调度点自动递送，与调度器、trap 框架深度集成。

信号信息结构 siginfo

```rust
#[derive(Debug, Clone, Copy)]
pub struct SigInfo {
    pub signo: Signal,
    pub errno: i32,
    pub code: SigCode,
    pub data: SigInfoData,
}

pub enum SigInfoData {
    Generic { pid: i32, uid: u32 },
    Child { pid: i32, uid: u32, status: SigStatus,
    ↪  utime: u64, stime: u64 },
    MemoryAccess { addr: VirtAddr },
    None,
}
```

‘siginfo‘ 结构体兼容 POSIX，支持丰富的信号附加信息（如来源进程、错误码、内存地址等），便于用户态精细处理信号。

不同信号类型可携带不同数据，极大提升灵活性和可扩展性。

信号帧与备用栈管理

```rust
pub enum SignalStackType { Primary, Alternate,
↪  Emergency }

pub struct SignalFrame {
    loaded: AtomicBool,
    range: VirtAddrRange,
    data: SignalFrameData,
}

pub struct SignalFrameManager {
    frames: [Option<SignalFrame>; 3],
    current: SignalStackType,
    scratch: AtomicUsize,
}
```

支持主栈、备用栈、紧急栈三类信号处理栈，自动切换，防止主栈溢出导致系统崩溃。

每个信号处理帧独立管理上下文和掩码，确保信号处理的安全性和可恢复性。

信号系统调用与接口设计

```rust
// 注册/查询信号处理动作
pub fn sys_sigaction(...);
// 屏蔽/恢复信号
pub fn sys_sigprocmask(...);
// 发送信号
pub fn sys_kill(pid, signo);
// 线程信号
pub fn sys_tkill(tid, signo);
pub fn sys_tgkill(tgid, tid, signo);
// 等待信号
pub fn sys_sigtimedwait(...);
// 信号挂起
pub fn sys_rt_sigsuspend(...);
// 信号返回
pub fn sys_sigreturn();
```

完整实现 POSIX 标准信号相关系统调用，支持进程/线程级信号注册、屏蔽、递送、等待与恢复。

接口设计与 Linux 兼容，便于用户态程序移植和系统集成。

信号分发与 Trap 集成

```rust
#[register_trap_handler(POST_TRAP)]
fn post_trap_handler(trap_frame: &mut TrapFrame,
↪  from_user: bool) -> bool {
    if from_user {
        time_stat_from_kernel_to_user();
        handle_pending_signals(trap_frame);
    }
    true
}

fn handle_pending_signals(tf: &TrapFrame) {
    // 进程级、线程级信号处理
    // 自动切换信号栈，递送信号
}
```

信号递送与 trap/调度点深度集成。每次从内核返回用户态前自动检查并递送待处理信号，支持进程级与线程级信号独立处理，保证信号响应的实时性和安全性。

用户态支持与系统调用

axsyscall 宏内核系统调用

AstrancE 通过 axsyscall 模块实现宏内核风格的系统调用分发，支持 50+ POSIX 标准系统调用，自动化分发表，错误码与返回值严格对齐 Linux/POSIX，涵盖进程/线程管理、文件操作、内存管理、信号、网络等。

```rust
#[macro_export]
macro_rules! syscall_handler_def {
    ($($(#[$attr:meta])*$sys:ident=>$args:tt$body:expr $(,)?)*)=>{
#[axhal::trap::register_trap_handler(axhal::trap::SYSCALL)]
pub fn handle_syscall(tf: &mut axhal::arch::TrapFrame, syscall_num: usize) -> Option<isize> {
    use syscalls::Sysno;
    let args = [tf.arg0(), tf.arg1(), tf.arg2(), tf.arg3(), tf.arg4(), tf.arg5()];
    let sys_id = Sysno::from(syscall_num as u32);
    let result: Option<SyscallResult> = match sys_id {
        // ...
        _ => None
    };
    result.map(|r| r.as_isize())
}
    };
}
```

系统调用分发表采用宏自动生成，便于维护和扩展。

axmono 用户态 ELF 加载与隔离

axmono 模块实现完整 ELF 加载器，支持静/动态链接。每进程独立虚拟地址空间，COW/懒分配，copy_from_kernel 方法保障用户/内核隔离。

```rust
pub fn load_elf_to_mem(
    elf_file: OwnedElfFile,
    uspace: &mut AddrSpace,
    args: Option<&[String]>,
    envs: Option<&[String]>,
) -> AxResult<(VirtAddr, VirtAddr, Option<VirtAddr>)> {
    // ...
}
#[cfg(any(feature = "mm", feature = "process"))]
pub fn copy_from_kernel(aspace: &mut axmm::AddrSpace) -> AxResult {
    use axmm::kernel_aspace;
    if !cfg!(target_arch = "aarch64") && !cfg!(target_arch = "loongarch64") {
        aspace.copy_mappings_from(&kernel_aspace().lock(), false)?;
    }
    Ok(())
}
```

该机制确保用户/内核隔离与安全。

模块化与多语言支持

AstrancE 支持 C/Rust 两种开发语言，POSIX API 兼容，主流程序无缝移植。feature flags 灵活裁剪功能，便于模块化扩展。

```
[features]
smp=["axfeat/smp"]
irq=["axfeat/irq"]
alloc=["dep:axalloc","axfeat/alloc"]
multitask=["axtask/multitask","axfeat/multitask","axsync/multitask"]
fd=["alloc","dep:axns"]
fs=["dep:axfs","dep:axfs_vfs","axfeat/fs","fd"]
net=["dep:axnet","axfeat/net","fd"]
pipe=["fd"]
select=["fd"]
epoll=["fd"]
uspace=["axns/thread-local"]
```

通过 feature flags 灵活裁剪功能，适应不同场景需求。

工作总结与展望

工作总结

AstrancE 继承并发扬了 ArceOS 的模块化思想，通过 feature flags 和条件编译机制，实现了内核各子系统（调度、内存、文件、信号、API 等）的灵活裁剪与组合。系统支持多种功能模块的按需启用，极大提升了代码的可维护性和适应性。

- 宏内核架构与模块化设计，灵活裁剪与组合
- 用户态支持与系统调用机制，POSIX 兼容
- 虚拟内存与内存安全，RAII 自动管理
- 文件系统与设备管理，VFS 动态挂载
- 信号机制与异步事件，调度器深度融合
- 工程实践与生态兼容，C/Rust 支持

未来计划

未来将持续完善内核各子系统，加强用户态生态兼容性，深化多核/多平台支持，丰富性能测试与优化。

- 完善网络协议栈，实现完整 TCP/IP 支持
- 扩展 axdriver，支持更多外设驱动
- 增强文件系统功能，优化性能，引入缓存
- 实现高级内存管理功能，如内存回收、压缩
- 完善用户态工具链与库，提升开发环境
- 持续性能优化与基准测试
- 引入更多安全机制，如 Capabilities、SELinux/AppArmor