合肥工業大學

Serein OS

操作系统设计文档

参赛队名：宇宙警备队

参赛人员：翁振昊，张尚昆，刘奕博

指导教师：田卫东、周红鹃

全国大学生计算机系统能力大赛
操作系统赛
内核实现赛道
2026 年 1 月

目录

目录 1
1. 介绍 3
1.1 项目背景及意义 3
1.2 操作系统内核研究的演进与趋势 4
1.3 项目的主要工作 4
2. 设计目标 5
2.1 提升内存使用效率 5
2.2 实现可控的进程调度 5
2.3 丰富进程间通信手段 5
2.4 优化 I/O 性能 6
2.5 支持线程编程模型 6
2.6 初步具备网络能力 6
2.7 改善交互体验 6
2.8 设计原则 6
3. 系统设计与实现 7
3.1 进程管理 7
3.1.1 进程状态管理 7
3.1.2 进程调度策略 8
3.1.3 进程同步 11
3.1.4 I/O 多路复用与定时器 14
3.1.5 小结 17
3.2 内存管理 18
3.2.1 共享内存 18
3.2.2 Copy-on-Write 21
（b）写入触发缺页：cow_handle 负责“分裂”页面 22
3.2.3 Lazy Allocation 23
3.2.4 mmap/munmap 26
3.2.5 小结 29
3.3 文件系统 30
3.3.1 FAT32 优化 30
3.3.2 缓冲区分桶 35
3.3.3 非阻塞 I/O 36
3.3.4 特殊设备文件 38
3.3.5 小结 39
3.4 网络功能 39
3.4.1 BSD Socket API 39
3.4.2 Unix Domain Sockets 42
3.4.3 IPv4/UDP 43
3.4.4 sockviz 可视化监控 45
3.4.5 小结 46
3.5 用户交互 46

3.5.1 分级日志.....46
3.5.2 溢出监控.....50
3.5.3 中断优化.....53
3.5.4 用户态命令集拓展.....57
3.5.5 可视化演示工具.....61
3.5.6 小结.....64
4. 系统调用的设计实现.....65
4.1 系统调用的流程.....65
4.1.1 调用发起与寄存器约定.....65
4.1.2 陷入处理与上下文保存.....65
4.1.3 参数提取与调用分发.....66
4.1.4 返回用户态.....67
4.2 部分系统调用分类与实现.....68
4.2.1 内存管理.....68
4.2.2 文件系统操作.....70
4.2.3 信号系统.....73
4.2.4 网络通信.....75
4.2.5 进程控制与线程.....77
4.2.6 系统管理.....79
4.3 添加新系统调用.....79
5. 总结和展望.....80
5.1 工作总结.....80
5.1.1 版本演进.....80
5.1.2 各子系统的实现.....81
5.1.3 性能数据.....82
5.1.4 测试情况.....82
5.2 经验总结.....83
5.2.1 踩过的坑.....83
5.2.2 设计上的取舍.....83
5.2.3 开发过程.....85
5.3 未来计划.....85
5.3.1 真实网络支持.....85
5.3.2 用户态线程库.....85
5.3.3 文件系统改进.....86
5.3.4 性能分析工具.....86
5.3.5 更多硬件支持.....86
5.3.6 小结.....86

1. 介绍

Serein 是一个基于 Kendryte K210 开发板的 xv6 操作系统内核增强项目。它基于华中科技大学 HUST-OS/xv6-k210 开源项目，并经过多个版本的迭代开发，以支持 RISC-V 架构，特别是 K210 芯片上的运行，同时兼容 QEMU 模拟器。该项目旨在将一个简化的教学操作系统扩展为功能更全面的嵌入式内核，支持现代操作系统特性，如高级调度、内存优化和网络支持。

1.1 项目背景及意义

操作系统是计算机系统的核心，负责对硬件资源进行统一管理并为上层应用提供运行环境。然而，在传统的操作系统教学中，相关内容往往侧重于原理和模型的讲解，学生虽然能够理解进程、内存和文件系统等基本概念，但将这些知识应用到真实系统中的机会较少，容易停留在“知道原理而不了解实现”的层面。

基于这一背景，我们以华中科技大学开源的 xv6-k210 为起点展开。该项目将 MIT xv6 教学操作系统移植至 RISC-V K210 开发板，具备较好的教学与实验基础。在此之上，我们对系统进行了持续的扩展与完善，从 V1.0 迭代至 V3.1，逐步将其发展为一个功能相对完整的嵌入式操作系统内核，并命名为 Serein。

在开发过程中，项目覆盖了操作系统的多个核心子系统。在内存管理方面，实现了 Copy-on-Write、Lazy Allocation 和 mmap 等机制，显著改善了进程创建和内存使用效率；在进程管理方面，引入 Stride 步长调度算法，以替代简单的轮转调度，实现更稳定的 CPU 份额分配；在进程间通信方面，实现了信号量、共享内存以及较为完整的 POSIX 信号机制；在文件系统方面，通过分配游标、稀疏索引和缓冲区分桶等手段，对 FAT32 的读写性能进行了优化；在网络功能方面，实现了 BSD Socket 接口，支持 Unix Domain Socket 和 IPv4/UDP 协议。

在用户交互方面，对原有的 Shell 进行了改进，补充了命令历史、行内编辑和自动补全等基本功能。相关实现涉及终端转义序列解析、内核与用户态协作以及输入输出缓冲区管理等问题，也暴露出一些在教学系统中容易被忽视的细节。为辅助理解系统行为，项目还实现了若干可视化演示程序，用于展示调度、内存分配、I/O 多路复用、进程协作和网络状态等机制。

系统主要在 QEMU 模拟的 RISC-V 环境中进行开发与测试，在保持与 K210 硬件兼容的同时，提高了调试与迭代效率。目前，系统共实现 75 个系统调用，并通过了 70 余项测试用例。

Serein 的开发过程是一种将操作系统理论逐步落实到工程实现中的实践尝试。通过实现和调试各类核心机制，我们对操作系统中一些关键设计和实现细节有了更加具体和直观的认识，也为后续进一步学习和研究相关内容奠定了实践基础。

1.2 操作系统内核研究的演进与趋势

操作系统内核研究从上世纪 60 年代的 Multics 和 UNIX 开始，经历了从宏内核到微内核的演变。早期内核如 UNIX 采用宏内核设计，所有服务集成在内核空间，效率高但稳定性差。随着复杂度增加，研究者转向微内核，如 Mach，将非核心服务移到用户空间，提高安全性。

现代商业系统中，Windows NT 内核融合宏微内核优势，支持多平台运行，并在安全性上持续优化。macOS 的 XNU 内核结合 Mach 和 BSD，提供优秀的图形和多媒体支持。开源领域，Linux 内核主导服务器市场，其模块化设计允许自定义，广泛用于云和嵌入式设备。国内如鸿蒙和麒麟，也基于 Linux 扩展，聚焦安全和国产化。

当前趋势包括了：

- **微内核复兴**：如 Fuchsia 的 Zircon，强调隔离以提升可靠性。
- **安全性增强**：面对网络威胁，内核引入自我保护机制，如地址空间布局随机化。
- **硬件适配**：多核、非易失性内存和量子计算推动内核优化，利用新特性提高性能。
- **嵌入式与边缘计算**：针对 IoT，内核需轻量、低功耗，支持实时性和网络。
- **AI 集成**：未来内核可能内置 AI 模块，优化资源分配。

这些趋势指导我们的项目，我们在 xv6-k210 中融入现代特性，如线程支持和网络栈，预见未来需求。

1.3 项目的主要工作

xv6-k210 项目主要致力于在多个关键方面对操作系统进行深度优化与功能拓展，以适配 K210 芯片并满足多样化应用需求。

从华中科技大学 Baseline 出发，我们的开发历程如图 1.3-1 所示：

图 1.3-1 开发历程

在进程管理方面，实现 Stride 调度、信号系统、clone/futex 线程支持等。内存管理引入 Copy-on-Write、Lazy Allocation 和 mmap/munmap。文件系统优化 FAT32 和缓冲区，提升 I/O 性能。网络功能添加 BSD Socket API、Unix Domain Sockets、IPv4/UDP 和 sockviz 可视化。IPC 包括 POSIX 信号量和共享内存。V3.0 新增 poll、alarm、fcntl 和 Shell 增强。

此外，我们添加 QEMU 支持、PID 哈希表、UID/GID 权限等，累计贡献约 10,000 行代码。

2. 设计目标

Serein 操作系统的设计围绕一个核心思路展开：在保持 xv6 简洁性的基础上，引入现代操作系统的关键特性，使其具备实用价值。基于华科 xv6-k210 提供的多核启动、FAT32 文件系统和基础进程管理，我们确定了以下设计目标。

2.1 提升内存使用效率

xv6 原版的内存管理采用直接复制和立即分配的策略，对于大内存进程效率较低。我们的设计目标是引入按需分配机制，让系统只在真正需要的时候才消耗物理内存资源。

具体而言，fork 操作应当延迟页面复制，只有在父子进程某一方真正写入时才分配新页面；堆扩展应当延迟物理页分配，只有在程序首次访问时才真正分配。此外，系统需要支持文件到内存的映射，让程序可以像访问内存一样读写文件内容。

2.2 实现可控的进程调度

原版 xv6 的轮转调度无法区分进程优先级。我们的设计目标是让系统能够按照权重分配 CPU 时间，高权重进程获得更多运行机会，同时保证调度结果的确定性和可验证性。

调度器需要在多核环境下安全运行，避免与进程状态变更操作形成死锁。考虑到系统稳定性，我们选择 Stride 步长调度作为目标方案，它的调度结果具有确定性，便于测试验证。

2.3 丰富进程间通信手段

xv6 只有管道一种 IPC 机制，无法满足复杂的进程协作需求。我们的设计目标是补充三类 IPC 机制：

（1）信号量：提供进程同步原语，支持阻塞等待和唤醒操作。

（2）共享内存：允许多个进程访问同一块物理内存，实现高效的大块数据共享。

（3）信号系统：提供异步通知机制，让进程能够响应外部事件和异常情况

对于共享内存，我们采用固定地址窗口的设计思路，确保多进程映射同一段

共享内存时获得一致的虚拟地址，简化指针处理。

2.4 优化 I/O 性能

在多核场景下，I/O 子系统的锁争用会成为瓶颈。我们的设计目标是降低锁粒度、支持非阻塞操作，并提供 I/O 多路复用能力。

缓冲区管理需要采用分桶设计，让访问不同扇区的请求能够并行处理。管道和文件需要支持非阻塞模式，在数据不可用时立即返回而非阻塞调用者。系统需要提供类似 poll 的接口，让应用程序能够同时等待多个文件描述符的就绪事件。

2.5 支持线程编程模型

单进程多线程是现代应用程序常用的并发模型。我们的设计目标是在内核层面提供线程支持的基础设施，包括创建共享地址空间的轻量级执行流，以及用户态快速锁机制。

2.6 初步具备网络能力

网络功能是现代操作系统的标配。考虑到项目的教学定位，我们的设计目标不是实现完整的 TCP/IP 协议栈，而是让系统具备正确的 Socket 编程接口。

Socket 需要与现有的文件抽象统一，支持 read、write、close 等通用操作，体现"一切皆文件"的设计理念。在传输层，需要支持本地进程间通信的 Unix Domain Socket，以及本地回环的 IPv4 地址，能够验证网络编程模型的正确性。

2.7 改善交互体验

作为一个面向学习和演示的系统，日常使用的便利性同样重要。Shell 需要具备命令历史浏览和 Tab 补全功能，让用户不必每次都完整输入命令。光标应当能够在命令行中自由移动，方便修改已输入的内容。

2.8 设计原则

在追求功能扩展的同时，我们遵循以下设计原则：

（1）稳定性优先：宁可采用简单但可靠的方案，也不追求复杂但容易出错的实现。

（2）渐进式开发：每个功能都需要通过完整的回归测试，确保不破坏已有功能。

（3）务实取舍：对于超出项目定位的复杂需求，选择满足核心目标的简化

方案。

3. 系统设计与实现

3.1 进程管理

在 xv6-k210 中，进程管理模块是操作系统的核心部分，负责进程的创建、初始化、调度以及资源回收。进程管理包括多个重要子模块，如 Stride 调度、进程状态、clone/futex 等。下面我们逐一解析这些功能模块。

3.1.1 进程状态管理

本系统采用基于进程控制块（PCB, struct proc）的进程管理模型。每个进程在生命周期内均处于某一种明确的状态，状态信息由 PCB 中的 state 字段记录。系统通过对该状态字段的维护，完成进程的创建、调度、阻塞与回收等核心管理流程，从而保证调度逻辑清晰、状态转换可控并便于调试与扩展。

系统定义的进程状态包括：UNUSED、SLEEPING、RUNNABLE、RUNNING、ZOMBIE、STOPPED。

UNUSED 表示进程表槽位未被占用，可用于分配新进程；当内核创建进程并完成必要的资源初始化（如 trapframe、页表、内核栈与上下文等）后，进程将进入可被调度的状态集合。

RUNNABLE 表示进程已具备运行条件，等待调度器分配 CPU；当调度器选择某个可运行进程并进行上下文切换后，该进程进入 RUNNING 状态并实际占用处理器执行。

运行中的进程可能因等待某些事件或资源（例如 I/O 完成、同步原语、特定条件满足等）而主动阻塞，此时通过 sleep()进入 SLEEPING 状态，并在相应事件发生后由 wakeup()唤醒重新转为 RUNNABLE。

当进程结束执行并调用退出流程后，它不会立即释放全部结构，而是进入 ZOMBIE 状态保留必要信息，等待父进程通过 wait()等机制回收，从而避免资源泄漏并保证父子进程语义正确。

此外，为支持信号与作业控制相关机制，系统引入 STOPPED 状态，用于表示进程因信号等原因被暂停执行；处于该状态的进程不会参与调度，直到收到继续执行的事件后再恢复为可调度状态。

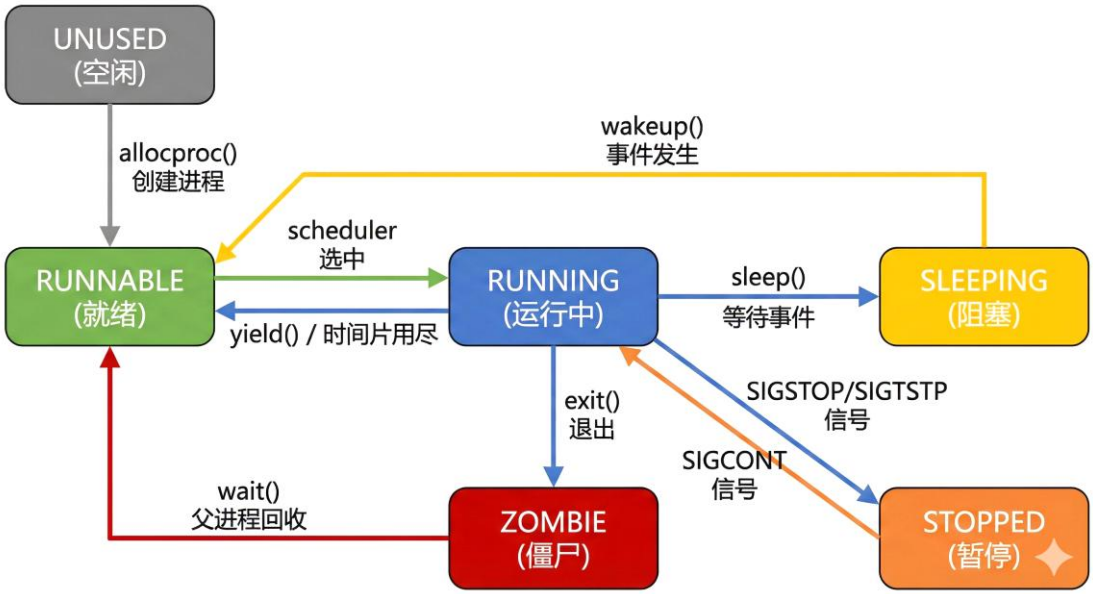

图 3.1-1 进程状态转换图

3.1.2 进程调度策略

（1）设计背景与目标：

在我们的操作系统Serein的早期版本中，我们采用了彩票调度思路，其优点是实现简单，但其本质依赖随机抽签，导致调度顺序不确定：即使某个进程票数更高，也可能因为“运气不好”连续多轮得不到CPU，从而引起响应时间抖动。

但随着我们后来在 Serein 中引入信号量、共享内存等同步/通信机制，这个问题会被进一步放大：例如高优先级（高票数）进程持有关键资源但短时间内未被调度，会使其他等待该资源的进程长时间阻塞，系统交互体验与并发性能明显下降。因此，我们将最终的调度策略调整为 Stride 步长调度。

（2）核心思想与算法说明：

Stride Scheduling：用确定性的比例调度替代随机抽签，从而达到以下目标：

（a）确定性比例分配：进程获得 CPU 时间的比例与其权重（tickets）近似成正比；

（b）降低抖动：避免彩票调度中的短期随机性导致的“高权重进程长时间不被调度”；

（c）多核可用且不引入死锁：在多核场景下保证调度器稳定运行，避免与**exit()/reparent()**等路径产生循环等待。

Stride 调度为每个进程维护一个累计量 pass，每次调度选择 pass 最小的可运行进程执行；进程每运行一次，就让它的 pass 增加一个步长 stride。其中：

（a）stride=STRIDE_LARGE/tickets；

（b）tickets 越大→stride 越小→pass 增长越慢→更频繁被选中；

（c）STRIDE_LARGE 是一个大常数，用于控制精度并避免溢出；

在 Serein 中，我们在 kernel/include/sched.h 中定义：

```
#define STRIDE_LARGE       (1 << 20)    // 2^20
#define DEFAULT_TICKETS 10
#define MAX_TICKETS         100
#define MIN_TICKETS         1
```

这里我们选择 2^20 作为大常数的原因是：在 tickets∈[1,100]的范围内，stride 仍有足够精度，并且 pass 使用 uint64 保存，长期运行也不易溢出。

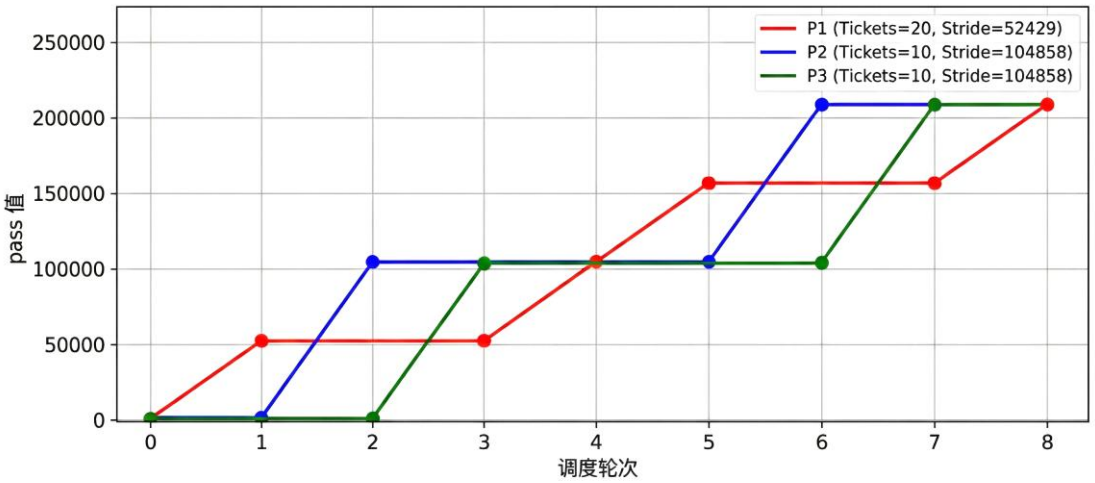

图 3.1-2 Stride 调度算法原理图

票数越多 → stride 越小 → pass 增长越慢 → 更频繁被选中，8 轮调度之后，P1 获得 50% CPU，P2、P3 各获得 25%"

（3）数据结构设计：

为支持 Stride 调度，我们在 PCB（kernel/include/proc.h 的 struct proc）中引入调度相关字段：

```
int tickets;              // 票数权重(1-100)
uint64 stride;            // stride=STRIDE_LARGE/tickets
uint64 pass;              // 累计 pass 值，每次运行后+stride
uint64 runticks;          // 累计运行 ticks
uint64 schedcount;        // 被调度次数
```

这些字段在 allocproc()中完成初始化，并在 fork()中继承父进程参数，以保证公平性和比例分配的连续性。

（4）实现过程：

阶段一：无锁扫描选取候选:

调度器首先遍历进程表，寻找 state==RUNNABLE 且 pass 最小的进程作为候选 minp：

该阶段不加锁扫描的目的，是避免在多核环境下长时间持锁，从而降低锁竞争，并为后续“单锁验证”创造条件。

```
struct proc *minp = 0;
```

```
2.  uint64 min_pass = ~0ULL;
3.
4.  for(p = proc; p < &proc[NPROC]; p++) {
5.  if(p->state == RUNNABLE && p->pass < min_pass) {
6.  min_pass = p->pass;
7.  minp = p;
8.  }
9.  }
```

该阶段不加锁扫描的目的，是避免在多核环境下长时间持锁，从而降低锁竞争，并为后续“单锁验证”创造条件。

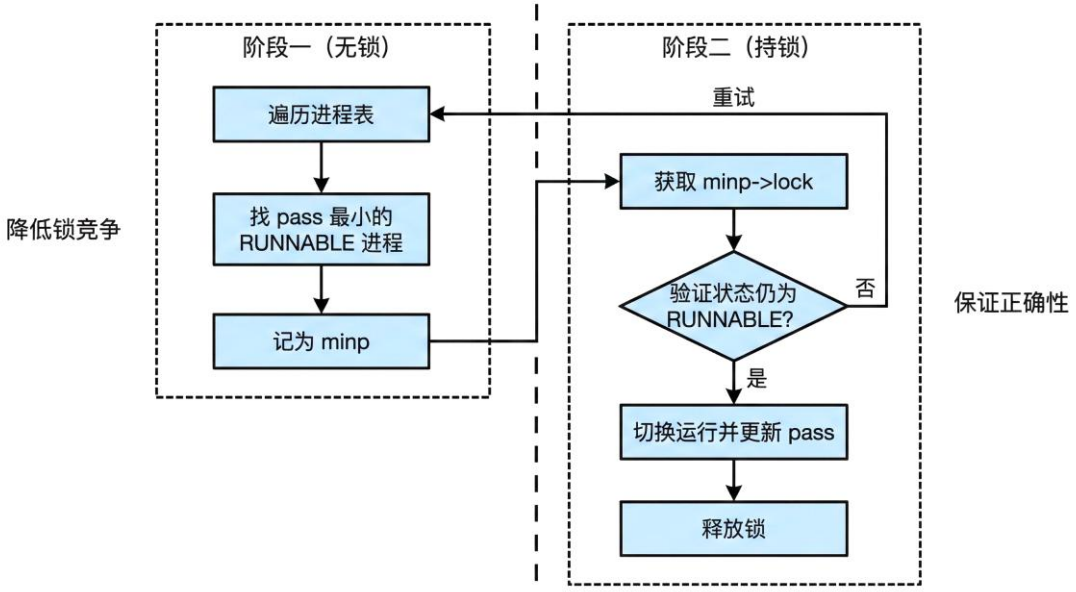

图 3.1-3 两阶段调度流程图

阶段二：单锁验证与切换：

找到候选进程后，调度器只获取该进程一个锁（minp->lock），并再次验证其仍处 RUNNABLE 状态：

```
1.  acquire(&minp->lock);
2.  if(minp->state == RUNNABLE) {
3.  minp->state = RUNNING;
4.  minp->schedcount++;
5.  minp->pass += minp->stride;
6.  swtch(&c->context, &minp->context);
7.  }
8.  release(&minp->lock);
```

当验证通过后，Serein 将进程状态切换为 RUNNING，更新 schedcount 与 pass，再进行上下文切换 swtch()。

(5)测试与可观测性：

我们 Stride 调度的实现已经通过 usertests，并解决了早期版本 reparent2 上出现的卡死问题。此外，系统中保留了调度统计字段，并可通过用户态工具观察调度效果；当不同票数进程同时竞争 CPU 时，运行时间比例能够更稳定地接近票数比例，调度抖动显著小于彩票调度，部分测试情况在我们的 docs 文档中也有相关说明。

3.1.3 进程同步

（1）设计背景与目标：

在原 xv6 版本中，用户态并发程序虽然可以通过 fork()创建多个执行流，但系统缺少一个可复用、可阻塞等待的同步原语。在未修改之前只提供内核内部的 sleep/wakeup 机制，用户态难以直接组织出“资源计数+阻塞等待+唤醒”的通用同步语义，这在实现父子进程严格顺序、生产者/消费者等并发场景时非常不便。

随着我们在 Serein 中逐步引入更丰富的进程机制，缺少同步原语会进一步放大并发问题：

没有可靠同步，多个进程对共享资源访问会出现竞态；无法阻塞等待，用户程序只能忙等，影响系统整体吞吐与交互体验；同步语义缺失也会让后续 IPC 无法“真正可用”。

因此，我们在 Serein 中新增计数信号量（Counting Semaphore）子系统，并通过系统调用开放给用户态，目标包括：

（a）为用户态提供标准的 P/V 阻塞式同步原语；

（b）复用内核 sleep/wakeup，实现简单且符合 xv6 风格；

（c）支持多进程竞争资源时的公平阻塞/唤醒语义；

（d）处理“进程在 wait 阻塞期间退出/被 kill”等异常路径，避免系统卡死或资源计数被破坏。

（2）核心思想与算法说明：

Serein 的信号量实现采用经典计数语义，并以 xv6 的 sleep/wakeup 为基础实现阻塞与唤醒。

P 操作（wait / down）：尝试获取资源，若资源不足则阻塞。在 Serein 内核中对应 semwait()，其核心逻辑为：

1.count--;

2.若 count < 0，则调用 sleep() 阻塞。

```
1.  acquire(&sem->lock);              // 获取该信号量的锁，保护后续操作
2.  sem->count--;                     // 将信号量计数值减 1，表示尝试获取一个资源
3.  if(sem->count < 0) {              // 如果计数值小于 0，说明资源不足
4.  // V2.0: 此处会将当前进程加入等待队列...
5.  sleep(sem, &sem->lock);    // 进程进入睡眠（阻塞），并原子性地释放锁
```

```
6.  // V2.0: 进程被唤醒后，会从等待队列移除...
7.  }
8.  release(&sem->lock);                // 释放信号量的锁
```

其中 sem 指针本身作为 sleep channel，使得 wakeup(sem) 能唤醒等待该信号量的进程。

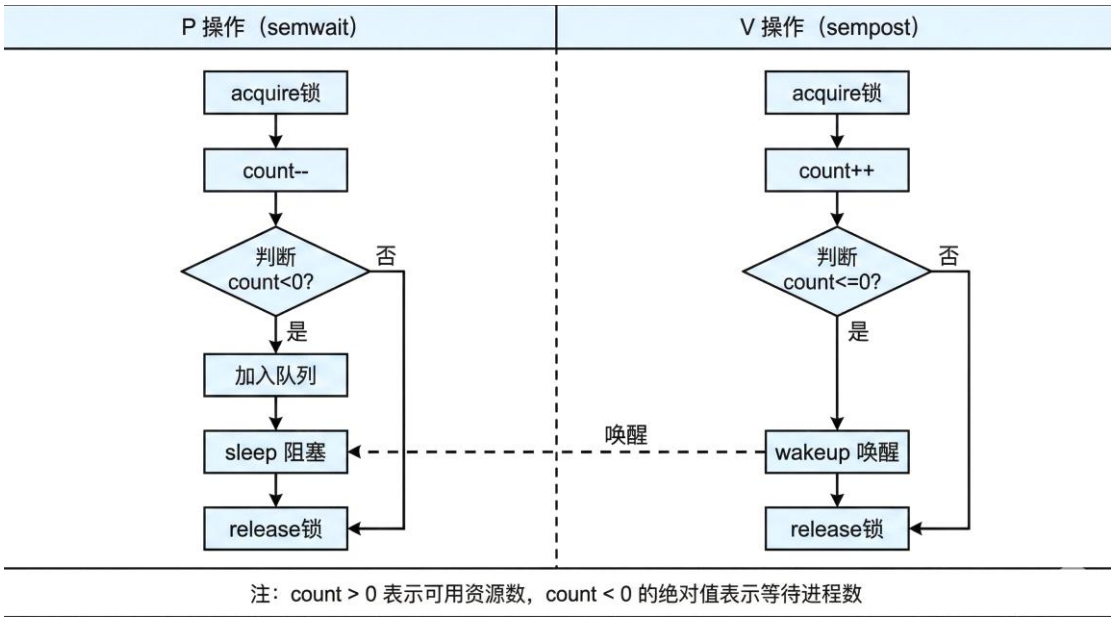

图 3.1-4 信号量 P/V 操作流程图

V 操作（post/up）：释放资源，若有等待者则唤醒一个。在 Serein 内核中对应 sempost()，其核心逻辑为：

（a）count++;

（b）若 count <= 0，则调用 wakeup()唤醒一个等待者。

```
1.  acquire(&sem->lock);          // 获取该信号量的锁，保护后续操作
2.  sem->count++;                 // 将信号量计数值加 1，表示释放一个资源
3.  if(sem->count <= 0) {         // 如果计数值小于等于 0，说明有进程正在等待
4.  wakeup(sem);                  // 唤醒一个正在该信号量上等待的进程
5.  }
6.  release(&sem->lock);          // 释放信号量的锁
```

“退出安全”的关键改进：

为防止进程在 semwait()阻塞期间被 kill 或 exit()导致信号量计数永久出错，我们在 exit()流程中加入了清理逻辑 sem_cleanup_proc()。该函数会遍历所有信号量，将退出的进程从等待队列中移除，并补偿性地将 count++，从而修复信号量状态，避免其他进程永久阻塞。

（3）数据结构设计：

Serein 维护一个全局信号量表，限制系统中同时存在的信号量数量，符合 xv6 的“静态表管理”风格。

信号量对象结构：

在 kernel/include/sem.h 的 struct sem 中定义了信号量的核心字段：

```
struct sem {
struct spinlock lock;          // 自旋锁，用于保护该信号量内部数据的一致性
int count;                     // 信号量计数值，可以为负数，表示等待者数量
int ref;                       // 引用计数，用于标记该信号量槽位是否被使用
// ... (其他字段，如 name)
struct proc *waiters[NPROC];   // 等待队列，显式记录所有阻塞在该信号量上的进程
int nwaiters;                  // 等待队列中的进程数量
};
```

其中 waiters[]/nwaiters 的引入是本模块的关键工程化改进：它使得 exit()清理路径能够准确识别“谁在等待哪个信号量”，避免等待状态残留破坏全局同步语义。

（4）实现过程：

Serein 的信号量实现主要围绕初始化、分配/回收、P/V 操作和退出清理四个环节。

（a）初始化与分配：

seminit()在系统启动时初始化全局信号量表。semalloc()则从表中寻找一个未被使用（ref==0）的槽位分配给用户，并返回其 sem_id。

（b）P/V 操作：

即 semwait()和 sempost()，它们通过对 count 的原子性修改，并结合 sleep/wakeup 机制实现阻塞与唤醒，如（2）中所述。

（c）退出清理：

这是 Serein 相比“最朴素信号量实现”的关键增强点。在 exit() 函数中，我们会调用 sem_cleanup_proc()，其核心逻辑如下：

```
for(int i = 0; i < sem->nwaiters; i++) {   // 遍历该信号量的等待队列
if(sem->waiters[i] == p) {                 // 如果找到了正在退出的进程 p
// 1. 从 waiters[] 队列移除 p (通过数组移位实现)
// ...
// 2. 修复信号量计数
sem->count++;                              // 将计数值加 1 作为补偿
break;                                     // 找到并处理后即可退出内层循环
}
}
```

这一补偿语义非常关键：因为该进程当初在 semwait()中已经执行了 count--，但它退出后不再会“真正消耗到资源并在未来归还”，因此必须把 count 加回去，保证系统同步计数保持正确。

（5）测试与可观测性：

为验证信号量模块的正确性与可用性，Serein 在用户态提供了系列测试程序：

- semtest.c：单进程语义测试，覆盖 sem_open/wait/post/close 等基本计数变化。
- semtest2.c：父子进程阻塞/唤醒测试，使用 fork() 构造一个进程 wait 阻塞、另一个进程 post 唤醒的同步流程。
- semstress.c：压力与并发测试，用于更强并发下的稳定性验证。
- 这些测试确保了信号量在单进程和多进程场景下的语义正确性与健壮性。

3.1.4 I/O 多路复用与定时器

（1）设计背景与目标：

在 Serein 的早期版本中，用户程序进行 I/O 等待主要依赖阻塞 read/write，很难写出“同时等待多个事件”的程序结构；同时也缺少简单的“定时触发机制”，不利于实现超时控制、周期性任务等能力。因此在我们的 SereinV3.0 中，我们补充了两个和“进程可用性/交互性”强相关的能力：

（a）poll()多路复用：一个进程可同时监控多个文件描述符（pipe/普通文件/socket 等），等待其中任意一个就绪；

（b）alarm()定时器：进程可以设置定时到期时间，到期后由内核发送 SIGALRM 信号触发用户态处理逻辑。

目标是：

支持事件循环/服务器等典型程序结构（poll）；支持超时与定时任务（alarm）；避免 poll 忙等导致 CPU100%占用；与 Serein 的信号系（sig_pending、sigaction）自然集成。

（2）核心思想与算法说明：

（a）poll 轮询检查+tick 级睡眠避免忙等:

poll 的核心做法是：

- 每轮遍历用户传入的 pollfd[]，对每个 fd 计算当前就绪事件 revents；
- 只要发现任意 fd 就绪就返回；
- 若都不就绪且未超时，则 sleep(&ticks)等一个时钟 tick 后继续下一轮，避免纯忙等待。

关键事件位定义：

```
1.  #define POLLIN      0x0001    // 可读
2.  #define POLLOUT     0x0004    // 可写
3.  #define POLLERR     0x0008    // 错误
4.  #define POLLHUP     0x0010    // 挂起
5.  #define POLLNVAL 0x0020    // 无效 fd
```

不同文件类型的“就绪判断”在 file_poll()中分支处理，例如 pipe：

```
1.  if(pi->nread!=pi->nwrite)// pipe 缓冲区有数据->可读
2.      revents|=POLLIN;
3.  if(pi->nwrite<pi->nread+PIPESIZE)// pipe 缓冲区有空间->可写
4.      revents |= POLLOUT;
5.  if(!pi->writeopen)          // 写端关闭->HUP
6.      revents |= POLLHUP;
```

poll 的主循环关键逻辑：

```
1.  while(1) {                          // 反复检查直到就绪或超时
2.  int ready = 0;                  // 统计本轮就绪 fd 数量
3.  for(int i = 0; i < nfds; i++) {     // 遍历用户传入的 pollfd 数组
4.  fds[i].revents = 0;                 // 每轮先清空返回事件
5.  if(fds[i].fd < 0) continue;       // 负 fd 忽略
6.          struct file *f = p->ofile[fds[i].fd]; //从进程打开文件表取 file
7.          if(f == 0) {                        // fd 不存在
8.              fds[i].revents = POLLNVAL; // 标记无效
9.              ready++;                    // 算作“就绪”（需要立即返回给用户）
10.             continue;
11.         }
12.   short rev = file_poll(f);    //根据文件类型计算就绪事件
13.   fds[i].revents = rev & (fds[i].events|POLLERR|POLLHUP| POLLNVAL);
14.   if(fds[i].revents != 0) ready++;       // 有事件则计入 ready
15.    }
16.
17.   if(ready > 0) return ready;           // 有就绪 fd：立即返回
18.
19. if(timeout==0||(ticks-start_ticks)>= timeout_ticks) // 立即返回
20.         return 0;
21.    acquire(&tickslock);                 // 保护 ticks 的 sleep channel
22.    sleep(&ticks, &tickslock);           // 睡眠到下一个 tick，避免忙等
23.    release(&tickslock);                 // 醒来后释放锁并继续下一轮
24. }
```

（b）alarm：进程级到期时间 + 时钟中断触发 SIGALRM

alarm 的核心是：为每个进程维护一个“到期 tick”，在时钟中断中检查并触发信号：

进程调用 alarm(seconds) 设置 p->alarm_ticks；

每次 timer tick 时遍历进程表，若 ticks >= p->alarm_ticks 则设置 sig_pending 对应位并清除 alarm（单次定时）；

信号系统在返回用户态前处理 sig_pending，最终让用户态 handler 执行。

进程字段（文档中的关键代码）：

```
1.  if(p->alarm_ticks > ticks)                          // 若旧 alarm 尚未到期
2.  remaining = (p->alarm_ticks - ticks) / 10;        // 计算剩余秒数（tick -> 秒）
3.  if(seconds == 0)                    // seconds=0 表示取消定时器
4.  p->alarm_ticks = 0;                                 // 禁用 alarm
5.  else
6.  p->alarm_ticks = ticks + seconds * 10;     // 秒转 tick，设置到期时间
7.  时钟中断检查并触发信号（逐行注释）：
8.
9.  // kernel/timer.c - timer_tick() 核心
10. if(p->state != UNUSED && p->alarm_ticks != 0) {
11. if(ticks >= p->alarm_ticks) {                  // 到期判断
12. p->sig_pending |= (1 << SIGALRM);     // 标记 SIGALRM 待处理
13. p->alarm_ticks = 0;                  // 单次 alarm：触发后清空
14. }
15. }
```

（3）数据结构设计：本模块涉及两类结构：

poll 的用户态参数结构 struct pollfd（在内核中用于 copyin/copyout）：

```
1.  struct pollfd {
2.  int fd;              // 文件描述符
3.  short events;     // 用户请求关注的事件
4.  short revents;   // 内核返回的就绪事件
5.  };
```

alarm 的进程字段：struct proc 增加 alarm_ticks，并复用信号系统字段 sig_pending 来触发信号。

（4）实现过程：

（a）poll：

定义事件位与 pollfd；

实现 file_poll()：按 FD_PIPE/FD_ENTRY/FD_SOCKET 等类型判定就绪；

sys_poll 主循环：每轮检查 -> 就绪返回 / 超时返回 / tick sleep 再检查。

（b）alarm：

在 struct proc 中新增 alarm_ticks 字段；

sys_alarm 完成“设置/取消 + 返回旧剩余时间”；

在 timer_tick() 中遍历进程，触发 SIGALRM：设置 sig_pending 位并清空 alarm；

确保信号系统在返回用户态时处理 sig_pending（与文档“SIGALRM 与信号系统集成”一致）。

（5）测试与可观测性：

- ❖ polltest：验证 pipe 的可读、关闭后 POLLHUP、timeout 返回 0 等行为；
- ❖ alarmtest：设置 1 秒 alarm，验证约 1000ms 后触发 SIGALRM；
- ❖ usertests 通过，说明与文件系统/pipe/信号系统/调度等路径集成稳定。

3.1.5 小结

Serein 的进程管理相关改进，核心目标是让系统在并发场景下更“确定、可控、可用”，并为后续更复杂的交互程序与网络/IPC 能力打基础。

调度策略确定化：Stride 替代 Lottery

我们将早期依赖随机抽签的彩票调度改为 Stride 步长调度，同时采用“两阶段调度（无锁扫描+单锁验证）”避免多核下与 exit()/reparent() 等路径发生锁循环等待，解决了早期 reparent2 卡死问题。这使调度顺序更稳定、抖动更小，尤其在引入同步原语后能显著减少“高优先级持锁但运气不好不被调度”造成的放大效应。

阻塞式同步原语：计数信号量（Semaphore）

为解决用户态缺少通用同步机制的问题，Serein 在内核中引入计数信号量表 sems[NSEM]，并通过系统调用暴露 sem_open/wait/post/close/getvalue。实现复用 sleep/wakeup，并在 V2.0 引入 waiters[]/nwaiters 等待队列，配合 sem_cleanup_proc() 在 exit() 清理阻塞等待，避免进程在 sem_wait 阻塞期间退出导致的计数破坏与“永久等待”问题，使同步机制在真实并发下更健壮。

进程可用性增强：poll + alarm（事件循环与超时基础）

poll() 为用户态提供同时等待多个 fd 的能力，并通过 sleep(&ticks) 避免忙等；alarm() 为进程引入定时触发机制：在 struct proc 维护 alarm_ticks，在时钟 tick 中到期后置位 sig_pending(SIGALRM)，与信号系统融合，为超时控制、交互增强和网络事件循环奠定基础。

Serein 的进程管理模块围绕“确定性调度 + 可用同步原语 + 高效 IPC + 事件/超时机制”逐层补齐能力，并在多核与异常退出路径上做了工程化加固，让并发系统更稳定、更可扩展。

3.2 内存管理

3.2.1 共享内存

（1）设计背景与目标：

在 xv6 原版中仅提供 pipe 作为 IPC 机制，适合字节流传输，但在以下场景下明显不足：

（a）需要在多个进程之间高效共享大量数据；

（b）需要“共享数据+信号量同步”这样的经典组合；

（c）需要进程之间能够直接读写同一块内存以提升吞吐。

因此我们在 Serein 中新增共享内存（Shared Memory）子系统，目标包括：

（a）支持多个进程映射同一组物理页，实现零拷贝数据共享；

（b）不同进程对同一共享段 attach 时映射到相同虚拟地址，避免“地址不一致导致数据不可见”；

（c）支持引用计数与 unlink 延迟释放，保证生命周期安全；

（d）处理进程异常退出时的清理，避免引用计数泄漏导致共享段无法回收。

（2）核心思想与算法说明：

Serein 的共享内存实现采用“共享段对象+物理页数组+固定虚拟地址映射”的设计：

每个共享段（struct shm）拥有若干物理页 pa[]。

当进程 shmattach(shmid)时，将这些物理页映射到进程页表中的一段连续虚拟地址区间。

为避免不同进程 attach 得到不同地址导致访问不一致，Serein 使用固定 VA 区间，并按 shmid 计算偏移：

基地址：SHM_VA_BASE=0x40000000

步长：SHM_VA_STRIDE=SHM_MAX_PAGES* PGSIZE

对应关系：同一个 shmid 在任何进程中 attach 都得到同一个 va。

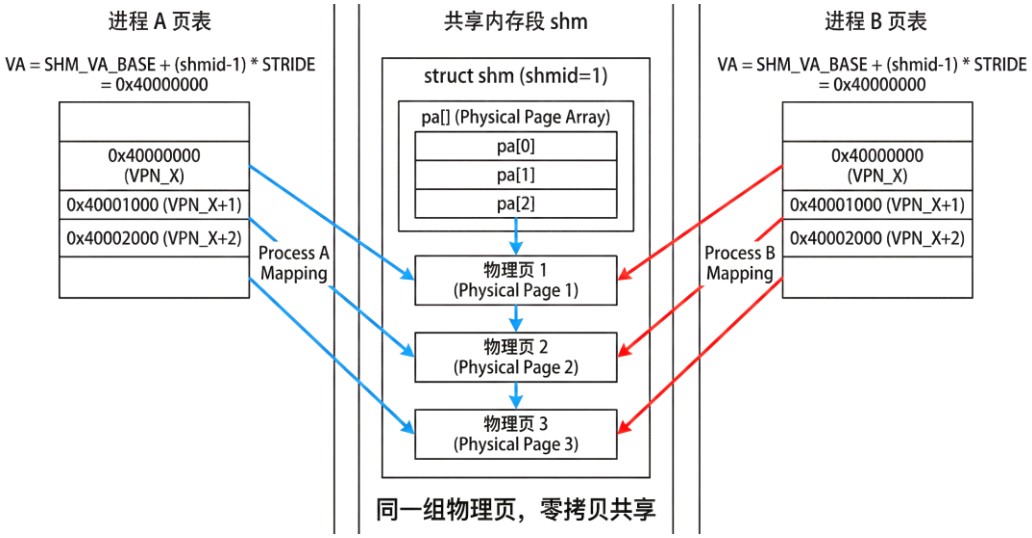

图 3.2-1 共享内存映射结构图

（3）数据结构设计：kernel/include/shm.h

```
#define NSHM 16                    // 最多 16 个共享段
#define SHM_MAX_PAGES 16           // 每段最多 16 页 (64KB)

struct shm {
char name[SHM_NAME_MAX];           // 共享段名字
void *pa[SHM_MAX_PAGES];           // 物理页数组
int npages;                        // 页数
uint64 size;                       // 字节大小
int ref;                           // 引用计数
int shmid;                         // 唯一标识
struct spinlock lock;              // 保护该 shm 段
};
```

这里的关键点是：pa[] 保存共享段拥有的物理页，多个进程 attach 只是“增加映射”，并不复制物理页。

（4）实现过程：

（a）shmcreate：创建共享段并分配物理页

核心流程是：先确定需要多少页，然后逐页 kalloc()分配并清零，最后写入元数据。

```
int npages = PGROUNDUP(size) / PGSIZE;   // 计算需要多少页（向上取整）
if(npages > SHM_MAX_PAGES)               // 超过上限则失败
return -1;

for(int i = 0; i < npages; i++) {        // 循环分配每一页
void *pa = kalloc();                     // 分配物理页
shm->pa[i] = pa;                         // 记录到物理页数组
memset(pa, 0, PGSIZE);                   // 清零，避免脏数据泄露
}

shm->size = size;                        // 记录大小
shm->npages = npages;                    // 记录页数
shm->ref = 1;                            // 创建者占用一个引用
shm->shmid = next_shmid++;               // 分配唯一 shmid
```

（b）shmattach：映射到固定虚拟地址

Serein 的关键改动在于：不用“从 p->sz 往上分配 VA”，而是对每个 shmid 计算固定 VA，从而保证一致性。

```
1.  uint64 va = SHM_VA_BASE + (shmid - 1) * SHM_VA_STRIDE;    // 计算固定虚拟地址起
    点
2.
3.  for(int i = 0; i < shm->npages; i++) {   // 逐页建立映射
4.  mappages(pagetable,                                  // 当前进程页表
5.  va + i * PGSIZE,                  // 每页连续 VA
6.  PGSIZE,                                  // 映射 1 页
7.  (uint64)shm->pa[i],                 // 对应共享物理页
8.  PTE_W|PTE_R|PTE_U);                 // 用户可读写
9.  }
10.
11. shm->ref++;                              // attach 成功，引用计数 +1
12. return va;                                 // 返回映射起始地址给用户态
```

（c）shmdetach：解除映射+引用计数回收

detach 的语义是：只解除页表映射（vmunmap(..., do_free=0)），物理页仍归共享段对象所有。

```
1.  vmunmap(pagetable, va, shm->npages, 0); // 解除映射，不释放物理页
2.  shm->ref--;                                          // 引用计数 -1
3.  if(shm->ref == 0 && shm->name[0] == '\0') // 已 unlink 且无人引用
4.  shmfree(shm);              // 释放物理页并回收槽位
```

（d）进程退出清理：shmdetach_all 防止引用泄漏

为避免进程异常退出时映射残留导致 ref 永远不归零，Serein 在 exit() 路径中加入统一清理：

```
1.  shmdetach_all(p->pagetable);// 退出时清理当前进程的所有 shm 映射
```

其作用是保证：无论用户态是否显式调用 detach，进程退出时共享内存引用计数都能正确减少，避免共享段“永远无法回收”。

（5）测试与可观测性：

共享内存功能通过用户态测试验证：

shmtest：验证父子进程通过共享内存互相读写字符串，确保“写入可见、地址一致、映射正确”。

同时与信号量结合时，可构造生产者/消费者模型：共享内存负责数据区，信号量负责同步（这是我们在 Serein 中引入这两者的典型用法组合）。

3.2.2 Copy-on-Write

（1）设计背景与目标：

在原版 xv6 中，fork()会完整复制父进程的用户地址空间：父进程有多少页，子进程就会分配并拷贝多少页。随着进程内存增长，fork() 代价快速上升；而在很多典型场景中（例如 fork() 后立刻 exec()），这部分拷贝几乎全部被浪费。

因此 Serein 在内存管理中引入 Copy-on-Write（写时复制，CoW） 优化，目标是：

（a）fork()时尽量不复制物理页，而是让父子进程共享页面；

（b）仅当某一方发生写入时，才为该进程单独复制出新页；

（c）保证语义与传统 fork 一致：父子进程写入互不影响；

（d）在多核环境下，引用计数维护必须正确，避免竞态导致提前释放或泄漏。

（2）核心思想与算法说明：

CoW 的基本思路是：

（a）fork 时：父子进程页表都指向同一物理页，但把原本可写页改成“只读+ CoW 标记”；同时该物理页引用计数 +1。

（b）运行时：当某进程尝试写入该页，CPU 会因“写只读页”触发缺页异常；内核在缺页处理中识别到该页是 CoW 页：

（c）若该物理页引用计数为 1：说明已经不再共享，直接恢复可写即可；

（d）若引用计数 ＞1：分配新页、复制内容、更新当前进程映射到新页，并对旧页 ref--。

（e）Serein 选择使用 RISC-V 页表的保留位作为 PTE_COW（bit 8）来区分 CoW 页面。

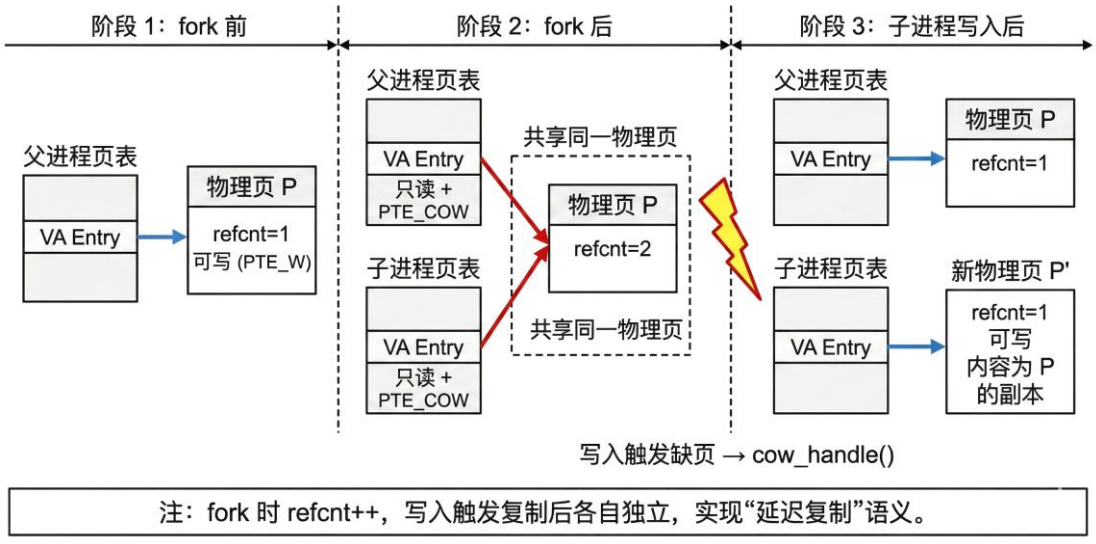

图 3.2-2 Copy-on-Write 工作流程图

（3）数据结构设计：

为了知道一个物理页是否仍被多个进程共享，我们在 kernel/kalloc.c 中维护每页引用计数数组（文档中的关键代码如下），并用锁保护避免多核竞态：

```
1.  static uint refcnt[PHYSTOP / PGSIZE];              // 每个物理页一个引用计数槽
2.  void krefget(void *pa) {                           // 增加引用计数
3.  acquire(&reflock);                                 // 加锁，避免多核并发更新出错
4.  refcnt[(uint64)pa / PGSIZE]++;                     // 对应物理页 ref++
5.  release(&reflock);                                 // 解锁
6.  }
7.  void krefput(void *pa) {                           // 减少引用计数，必要时释放
8.  acquire(&reflock);                                 // 加锁保护 refcnt
9.  if(--refcnt[(uint64)pa / PGSIZE] == 0) {           // ref-- 后若变 0
10. release(&reflock);                                 // 先释放锁（避免 kfree 内部再用锁）
11. kfree(pa);                                         // 真正释放物理页
12. return;                                            // 结束
13. }
14. release(&reflock);                                 // ref 仍 >0，仅解锁
15. }
```

（4）实现过程：

（a）fork 阶段：uvmcopy 改为共享页+标记 CoW

原版 uvmcopy() 会为子进程分配新页并拷贝内容；Serein 的 CoW 修改点是：对“原本可写页”清除写权限并设置 CoW 标记，然后让子进程映射到同一物理页，并增加引用计数。

```
1.  if (flags & PTE_W) {                               // 如果该页原本可写
2.      flags = (flags & ~PTE_W) | PTE_COW;            // 清除写位 + 打上 CoW 标记
3.      *pte = PA2PTE(pa) | flags;                     // 同步更新父进程页表为只读 CoW
4.  }
5.  mappages(new, i, PGSIZE, pa, flags);               // 子进程映射同一物理页（只读 CoW）
6.  krefget((void*)pa);                                // 该物理页共享者+1，引用计数增加
```

（b）写入触发缺页：cow_handle 负责“分裂”页面

当用户态写 CoW 页触发 fault，内核进入 CoW 处理函数。它先确认该页确实是 CoW，然后根据引用计数决定“直接改权限”还是“复制新页”。

```
1.  pte_t *pte = walk(pagetable, va, 0);               // 找到 va 对应的页表项
```

```
2.  if ((*pte & PTE_COW) == 0)                          // 若没有 CoW 标记
3.      return -1;                                      // 说明不是 CoW fault，交给别的路径
4.  uint64 pa = PTE2PA(*pte);                           // 取出旧物理页地址
5.  if (krefcnt((void*)pa) == 1) {                      // 如果该页已经只剩 1 个引用
6.      *pte = (*pte | PTE_W) & ~PTE_COW;               // 直接恢复可写，并清除 CoW 标记
7.      return 0;                                       // 完成
8.  }
9.  char *mem = kalloc();                               // 仍有多个引用：分配新物理页
10. memmove(mem, (char*)pa, PGSIZE);                    // 把旧页内容复制到新页
11. *pte = PA2PTE(mem) | (flags | PTE_W) & ~PTE_COW;// 更新页表
12. krefput((void*)pa);                                 // 原物理页引用计数减少（当前进程不再共享它）
13. return 0;                                           // 完成
```

这里的关键保证是：写入发生时才真正复制，因此 fork 代价大幅下降；同时通过 refcnt 保证多进程共享/释放的正确性。

（5）测试与可观测性：

从文档记录的结果来看，CoW 对 fork 性能提升显著（10MB 进程）：

原版 fork：约 80ms

引入 CoW：约 5ms（约 16 倍提升）

同时 usertests 通过，说明：

写时复制不破坏用户态读写语义；页表权限与缺页处理流程能够正确协作；引用计数逻辑在并发与退出释放路径中保持一致性。

3.2.3 Lazy Allocation

（1）设计背景与目标：

在原版 xv6 中，进程调用 sbrk(n) 扩展堆空间时，内核通常会立即为新增的虚拟地址范围分配物理页并建立映射。但在实际应用中，堆扩展后往往并不会马上访问所有新增空间（例如一次性预留大块内存、或逐步增长的数据结构），立即分配会带来两类问题：

（a）浪费：分配了但长期不使用的页占用宝贵物理内存；

（b）变慢：sbrk/growproc 需要分配并映射多页，系统调用耗时上升。

因此 Serein 引入 Lazy Allocation：

sbrk() 扩展堆时只扩大逻辑大小，不立刻分配物理页；当进程第一次访问新增区域触发缺页异常时，再由内核分配并映射该页；与 vmunmap/uvmdealloc 等回收路径兼容：允许虚拟区间存在“未映射空洞”，不应 panic。

（2）核心思想与算法说明：

Lazy Allocation 将“分配时机”从 sbrk()调整为“首次访问时”。实现上分两步：

（a）扩展阶段（不分配）：

growproc(n>0) 仅更新进程大小 p->sz，不调用 uvmalloc 分配页。

（b）访问阶段（缺页触发分配）：

当用户访问到尚未映射的地址，会触发 page fault。缺页处理路径识别该地址是否属于合法的用户地址范围（va < p->sz 等），若合法则：

kalloc() 分配物理页并清零；同时映射到用户页表 pagetable 与内核页表 kpagetable（Serein 双页表结构）；返回用户态后指令重试，即可正常访问。

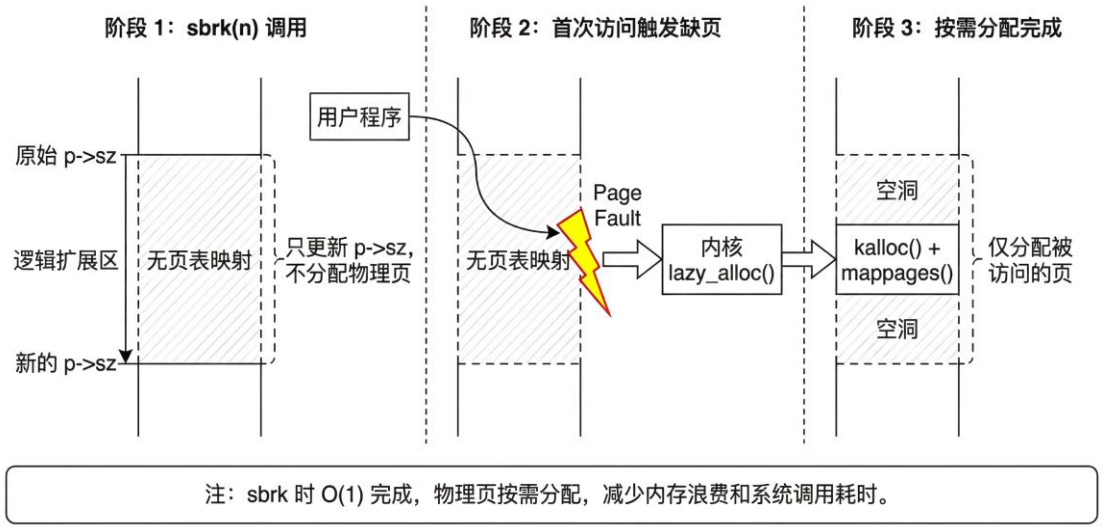

图 3.2-3 Lazy Allocation 工作流程图

（3）数据结构设计：

Lazy Allocation 不需要新增复杂结构，主要复用：

（a）struct proc 的 p->sz：表示进程用户地址空间上界；

（b）页表结构：允许某些 VA 尚未映射（存在空洞）。

关键在于系统的回收/解除映射函数必须支持“页表项不存在或无效”的情况。

（4）实现过程：

（a）growproc：sbrk 扩堆时只更新 p->sz:

```
1.  int growproc(int n) {
2.      struct proc *p = myproc();                    // 获取当前进程
3.      uint64 sz = p->sz;                            // 取出当前进程大小
4.
5.      if(n > 0){                                    // 扩展堆：sbrk(n), n>0
6.          uint64 newsz = sz + n;                    // 计算新的地址空间大小
7.          if (newsz >= MAXVA)                       // 超过最大用户地址则失败
```

```
8.              return -1;
9.          p->sz = newsz;                          // 关键：只更新大小，不分配物理页
10.     } else if(n < 0){                           // 收缩堆：sbrk(n), n<0
11.         sz = uvmdealloc(p->pagetable,           // 回收映射页（存在则回收）
12.                         p->kpagetable,          // 同时回收内核页表映射
13.                         sz,                      // 原大小
14.                         sz + n);                // 新大小（更小）
15.         p->sz = sz;                             // 更新进程大小
16.     }
17.     return 0;                                   // 成功
18. }
```

（b）缺页时 lazy_alloc：真正分配并映射：

当访问未映射 VA 触发缺页，进入 lazy_alloc()。

```
1.  va = PGROUNDDOWN(va);                   // 对齐到页边界（fault addr 可能非对齐）
2.  if (va >= sz || va < PGSIZE)            // 合法性检查：必须在用户空间且 < p->sz
3.      return -1;                          // 非法访问：交给上层 kill 或返回错误
4.  char *mem = kalloc();                   // 分配一页物理内存
5.  memset(mem, 0, PGSIZE);                 // 清零，避免泄露旧数据
6.
7.  mappages(pagetable,                     // 建立用户页表映射
8.               va, PGSIZE, (uint64)mem,
9.               PTE_W|PTE_R|PTE_X|PTE_U);  // 用户态可访问（按实现给出权限）
10. mappages(kpagetable,                    // 建立内核页表映射
11.              va, PGSIZE, (uint64)mem,
12.              PTE_W|PTE_R|PTE_X);        // 内核态访问权限
13.
14. return 0;                               // 分配完成，返回后用户态可继续执行
```

（c）vmunmap 兼容“空洞”：跳过未映射页

Lazy Allocation 的副作用是：一段虚拟区间里可能有“尚未触发访问”的页，这些页没有页表项。如果 vmunmap 仍假设“每页必然映射”，就会 panic。

因此 Serein 在 vmunmap() 做了关键修改：

```
1.  if((pte = walk(pagetable, a, 0)) == 0)  // 页表项不存在
2.      continue;                           // 直接跳过（空洞）
3.  if((*pte & PTE_V) == 0)                 // 页表项无效
```

```
4.      continue;                                   // 同样跳过
```

（5）测试与可观测性：

Lazy Allocation 的正确性主要体现在：

sbrk 扩展后立刻不访问不会分配页；

首次访问能正确 fault 并补齐映射；

收缩堆/exit 等路径回收不会因空洞 panic；

usertests 中 sbrkbasic / sbrkmuch / mem 等用例可覆盖这类行为。

3.2.4 mmap/munmap

（1）设计背景与目标：

在 Serein 的早期版本中，用户态进程只能通过 read/write 做文件 I/O，或通过 sbrk 申请匿名内存。对于“把文件内容当作内存来访问”这类常见需求（例如加载大文件、共享只读数据、对文件进行随机访问并减少拷贝），缺少统一机制。

因此我们在 Serein 中引入 mmap/munmap，目标包括：

（a）支持把文件或匿名内存映射到进程虚拟地址空间；

（b）mmap 区域与堆（p->sz）完全分离，避免与 sbrk 冲突；

（c）支持 MAP_SHARED 时在 munmap 做写回，保证文件映射可持久化；

（d）实现尽量保持简单：先支持“整段 unmap（完全解除映射）”，减少边界情况。

（2）核心思想与算法说明：

（a）基于 VMA（虚拟内存区间）管理映射元数据

Serein 为每个进程维护 VMA 槽位数组（文档中已给出结构），mmap() 本质上只是：

在 p->vmas[] 中找一个空槽位；选择一段不会冲突的虚拟地址 [va, va+len)；记录映射信息（prot/flags/file/offset)；

返回 va 给用户态。

（b）固定 mmap 起始区间：MMAPBASE，不修改 p->sz

在 kernel/sysfile.c 的实现里明确规定：

#define MMAPBASE 0x40000000

mmap 区域从 MMAPBASE 向高地址增长

不再修改 p->sz（代码注释里也写明：mmap 区域与堆分离）

这正对应你 docs 中提到的“最初放在 p->sz 之上会与 sbrk 冲突，后来改成固定区域”。

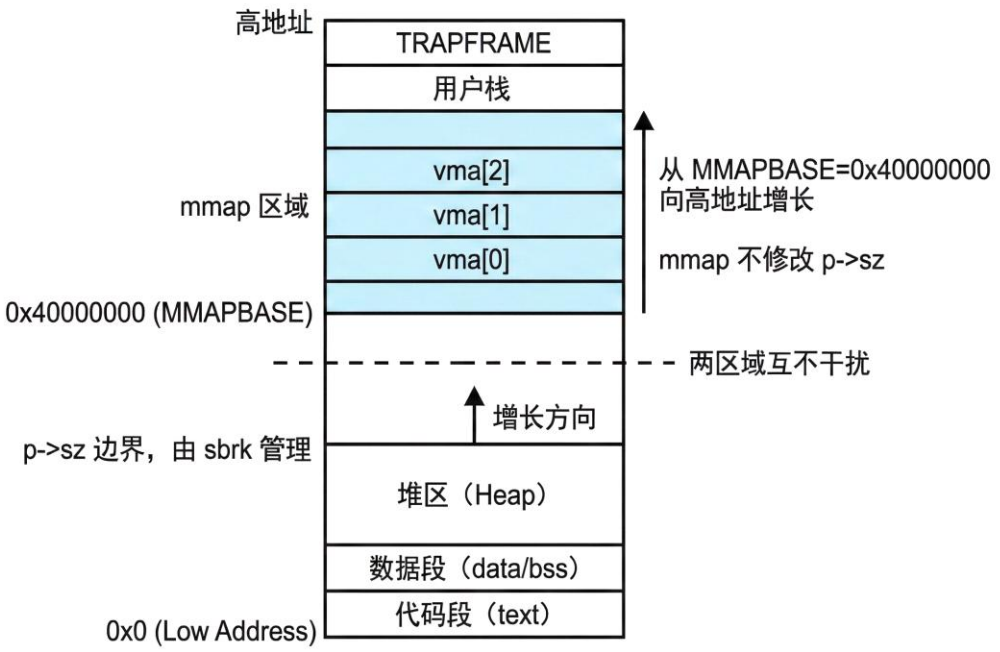

图 3.2-4 mmap 地址空间布局图

（3）数据结构设计：

mmap/munmap 的核心数据结构是 VMA（Virtual Memory Area），用于描述一段映射区间：

```
1.  struct vma {
2.      uint64 addr, len;          // 映射虚拟地址范围
3.      int prot, flags;           // PROT_xxx / MAP_xxx
4.      struct file *f;            // 文件映射对应的 file（匿名映射则为 0）
5.      uint64 offset;             // 文件偏移
6.      int valid;                 // 槽位是否有效
7.  };
8.
9.  struct proc {
10.     struct vma vmas[MAX_VMA]; // VMA 槽位数组
11. };
```

（4）实现过程：

（a）sys_mmap：分配 VMA 槽位+选择地址+记录元数据

Serein 的 sys_mmap()实现在 kernel/sysfile.c，关键流程如下：

```
1.  // 1. 参数校验
2.  if(len == 0) return -1;   // 长度不能为 0
3.
4.  // 2. 文件映射检查
5.  if(!(flags & MAP_ANONYMOUS)) {   // 非匿名映射
```

```
    if(fd < 0 || fd >= NOFILE || (f = p->ofile[fd]) == 0) return -1;
    if((prot & PROT_READ) && !f->readable) return -1;
    if((prot & PROT_WRITE) && (flags & MAP_SHARED) && !f->writable) return -1;
}

// 3. 分配 VMA 槽位
for(int i = 0; i < MAX_VMA; i++)
    if(!p->vmas[i].valid) { vma = &p->vmas[i]; break; }
if(!vma) return -1;   // 无可用槽位

// 4. 计算映射地址（从 MMAPBASE 开始，向高地址增长）
len = PGROUNDUP(len);
uint64 va = MMAPBASE;
for(int i = 0; i < MAX_VMA; i++)
    if(p->vmas[i].valid && p->vmas[i].addr + p->vmas[i].len > va)
        va = p->vmas[i].addr + p->vmas[i].len;
va = PGROUNDUP(va);

// 5. 地址越界检查
if(va + len > TRAPFRAME) return -1;

// 6. 初始化 VMA
vma->addr = va;
vma->len = len;
vma->prot = prot;
vma->flags = flags;
vma->offset = offset;
vma->valid = 1;
vma->f = f;
if(f) filedup(f);   // 持有文件引用
```

return va; // 返回映射的虚拟地址> 你现在这份实现是“登记 VMA + 返回地址”的风格：不在 sys_mmap 里立即建立所有页映射（这也与 docs 中“mmap 区域缺页由 fault 处理”一致的设计方向相吻合）。

（b）sys_munmap：整段解除映射 + MAP_SHARED 写回

Serein 当前 sys_munmap() 做了简化：只支持完全解除（addr==vma->addr && len==vma->len），并在 MAP_SHARED 时写回所有已映射页到文件：

```
// 找到包含 addr 的 vma
```

```
2.    for(int i = 0; i < MAX_VMA; i++)
3.      if(p->vmas[i].valid && addr >= p->vmas[i].addr && addr < p->vmas[i].addr + p->vmas[i].len)
4.        { vma = &p->vmas[i]; break; }
5.    if(!vma) return -1;
6.
7.    // 仅支持整段 unmap
8.    if(!(addr == vma->addr && len == vma->len)) return -1;
9.
10.   // MAP_SHARED: 将已映射页写回文件
11.   if((vma->flags & MAP_SHARED) && vma->f)
12.     for(uint64 a = vma->addr; a < vma->addr + vma->len; a += PGSIZE)
13.       if((pte = walk(p->pagetable, a, 0)) && (*pte & PTE_V)) {
14.         uint64 pa = PTE2PA(*pte);
15.         uint64 off = vma->offset + (a - vma->addr);
16.         elock(vma->f->ep); ewrite(vma->f->ep, 0, pa, off, PGSIZE); eunlock(vma->f->ep);
17.       }
18.
19.   // 清理映射并释放页（用户页表 + 内核页表）
20.   for(uint64 a = vma->addr; a < vma->addr + vma->len; a += PGSIZE) {
21.     if((pte = walk(p->pagetable, a, 0)) && (*pte & PTE_V)) { kfree((void*)PTE2PA(*pte)); *pte =
      0; }
22.     if((pte = walk(p->kpagetable, a, 0)) && (*pte & PTE_V)) *pte = 0;
23.   }
24.   if(vma->f) fileclose(vma->f);   // 释放文件引用
25.   vma->valid = 0;                          // 释放 VMA
26.   return 0;（5）测试与可观测性：
```

从 docs 的描述，mmaptest 覆盖了：

文件映射后读写是否正确；MAP_SHARED 下 munmap 写回是否生效；并且 usertests 通过，说明 mmap 区域与堆分离、以及与 Lazy/vmunmap 等路径的兼容性没有破坏原有内存测试。

3.2.5 小结

Serein 的内存管理改进，核心目标是降低 fork 与内存分配的成本、减少无效内存占用，并提供更接近现代 OS 的内存抽象接口。

Serein 的内存管理整体围绕“更少拷贝、更少立即分配、更强映射/共享能力”做了增强，主要包括四块：

命名多页共享内存（SHM）：实现
shmcreate/shmopen/shmattach/shmdetach/shmunlink，支持多页共享段与引用计数、unlink 延迟释放语义；采用固定虚拟地址窗口保证不同进程 attach 地址一致，并在进程退出路径 shmdetach_all 自动清理，避免引用泄漏。

Copy-on-Write（CoW）：

fork() 不再全量复制地址空间，而是父子进程共享物理页、页表标记为只读 + PTE_COW；写入触发缺页后再复制新页。配套在 kalloc.c 加入物理页引用计数，保证多核下共享/释放正确。

Lazy Allocation：

sbrk/growproc 扩堆时只更新 p->sz，不立即分配物理页；首次访问触发缺页再分配并映射。并修改 vmunmap 等回收路径以兼容“未映射空洞”，避免 panic。

mmap/munmap：

引入基于 VMA 的映射管理，mmap 区域从 MMAPBASE 起单独划分，避免与堆冲突；munmap 支持整段解除映射，并在 MAP_SHARED 时将已映射页写回文件。

Serein 的内存管理通过不同机制分别优化“fork 拷贝成本”“堆内存即时分配浪费”“文件映射与共享访问能力”，并配套修正回收/解除映射路径以支持空洞和复杂映射，整体上让系统更高效、更接近现代 OS 的内存行为。

3.3 文件系统

在 Serein 的开发过程中，文件系统不仅要解决"能用"的问题，更要解决"好用"的问题。由于 K210 开发板主要依赖 SD 卡作为存储介质，其随机读写性能相对较弱。如果直接照搬原始的 FAT32 实现，在大文件读写和高并发场景下，系统会变得极度迟钝。为此，我们在标准 FAT32 的基础上，通过对 fat32.c 和 bio.c 的深度改造，进行了一系列针对性的优化。

3.3.1 FAT32 优化

FAT32 虽然古老，但其简单的链表式结构使其成为嵌入式设备的默认选择。然而，在真实工程场景中，这种链表结构带来了两个严重的 O(N) 性能瓶颈：空闲簇查找和随机读写寻址。

（1）分配游标与环形扫描

在原始实现中，每次需要分配新的磁盘簇时，内核总是从 FAT 表的第 2 个簇开始逐个遍历，直到找到空闲位置。随着文件越来越多，这个过程会变得极其漫长。

为了解决这个问题，我们在 fat32.c 中引入了一个静态全局变量 next_free_clus_hint，用于记住上次分配簇的位置。优化后的分配算法采用 Next-Fit

策略，不再使用简单的线性扫描，而是从上次成功分配的位置继续向后查找。具体流程如下：

（a）检查游标位置。首先检查全局游标 next_free_clus_hint 是否指向一个合法区域，如果有效则直接从该位置开始向后扫描。

（b）线性推进。向后逐个检查 FAT 表项，一旦发现某个簇标记为 0 表示空闲，则立即占用该簇，更新游标为当前位置加 1，并对簇内容清零后返回。

（c）环回扫描。如果扫描到磁盘末尾仍未找到空闲簇，算法会折返到磁盘开头的第 2 个簇，继续向后扫描直到再次遇到起始位置。

（d）最终判定。只有转完一整圈还没找到空闲簇，才抛出磁盘已满的错误。

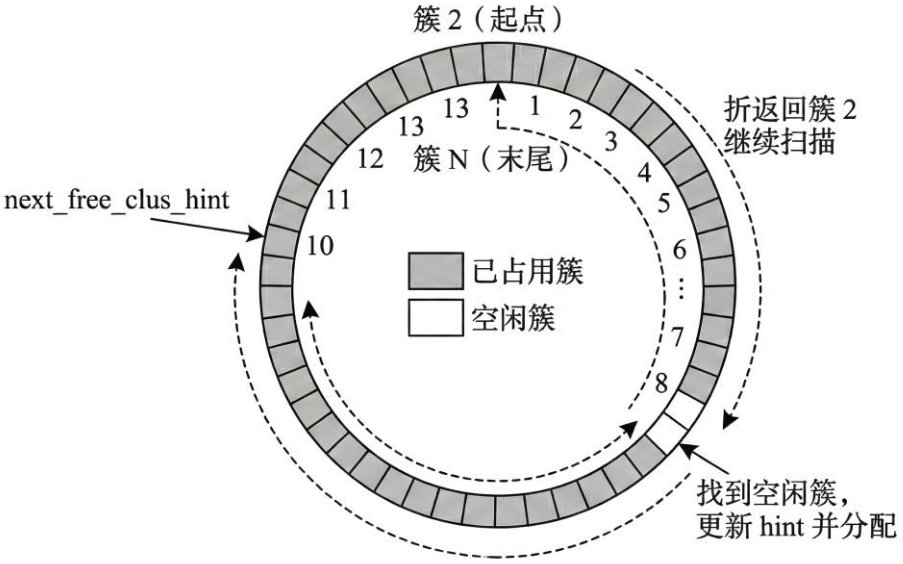

图 3.3-1 分配游标环形扫描示意图

核心实现代码如下：

```
// kernel/fat32.c
// V2.0 Optimization: Remember last allocated cluster position
static uint32 next_free_clus_hint = 2;

static uint32 alloc_clus(uint8 dev) {
    uint32 start_clus = next_free_clus_hint;
    if (start_clus < 2 || start_clus >= total_clus) {
        start_clus = 2;
    }

    // Search from hint to end
    for (uint32 i = start_sec_idx; i < fat.bpb.fat_sz; i++) {
        // ... scan FAT entries ...
```

```
            if (((uint32 *)(b->data))[j] == 0) {
                ((uint32 *)(b->data))[j] = FAT32_EOC + 7;
                next_free_clus_hint = clus + 1;    // Update hint
                zero_clus(clus);
                return clus;
            }
        }

        // Wraparound: search from beginning to hint
        for (uint32 i = 0; i < start_sec_idx; i++) {
            // ... same logic ...
        }
        panic("no clusters");
    }
```

这一改进利用了文件写入时的空间局部性：如果刚刚在位置 X 分配了一个簇，那么位置 X+1 很大概率也是空闲的。这种接力棒式的分配策略，将大文件连续写入时的分配开销从 O(N) 降低到了接近 O(1)。

（2）稀疏索引加速随机访问

FAT32 的另一个痛点是随机访问。文件的物理存储是不连续的，为了读取文件的第 100MB 数据，内核必须从文件的第一个簇开始，沿着 FAT 链表跳转数万次才能到达目标位置。对于视频播放器这种需要频繁 Seek 的应用，这是灾难性的。

我们在内存中的目录项结构 struct dirent 中引入了稀疏索引机制。在 fat32.h 中扩展了结构定义：

```
// kernel/include/fat32.h
struct dirent {
    // ... 标准字段 ...
    uint32 *index;        // 动态分配的索引表
    int index_cnt;        // 索引表有效条目数
    // ...
};
```

索引构建的核心逻辑很简单：每遍历 128 个簇，就在索引表中记录一次当前位置。这相当于给文件建立了一份跳表：第 0 层是原始的 FAT 链表，每个簇指向下一个簇；第 1 层是稀疏索引层，每隔 128 个簇就立一块路牌。

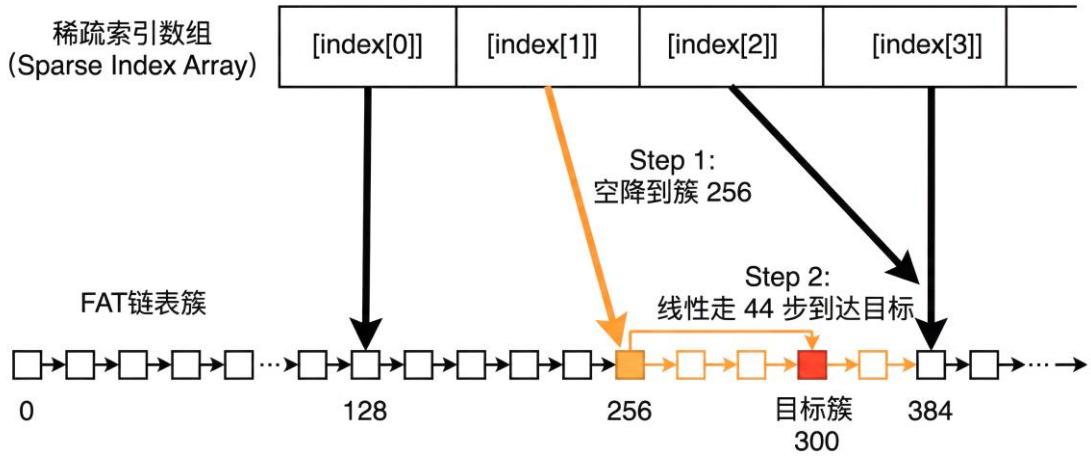

图 3.3-2 稀疏索引结构示意图

```
1.  // kernel/fat32.c - reloc_clus 函数中的索引构建
2.  if (entry->clus_cnt % 128 == 0) {
3.      int idx = entry->clus_cnt / 128 - 1;
4.      if (!entry->index) {
5.          entry->index = (uint32*)kalloc();
6.          if (entry->index) {
7.              memset(entry->index, 0, PGSIZE);
8.              entry->index_cnt = 0;
9.          }
10.     }
11.     if (entry->index && idx < 1024) {
12.         entry->index[idx] = clus;
13.         if (idx >= entry->index_cnt) entry->index_cnt = idx + 1;
14.     }
15. }
```

当调用 reloc_clus 进行寻址时，算法不再盲目遍历，而是先查找索引表，找到最近的路牌，然后从该路牌空降，仅需走最后几步即可到达终点。索引表使用 kalloc 动态分配一个 4KB 页面，可容纳 1024 个索引项，按每簇 4KB 计算，单个索引表可支持最大 512MB 文件的 O(1) 级随机访问。

（3）批量截断

在删除大文件时，我们需要释放它占用的成千上万个簇。如果每释放一个簇就写一次磁盘更新 FAT 表，那么删除一个 1GB 的文件可能需要几分钟。

我们在 V2.0 中重构了 etrunc 函数，加入了批量更新逻辑：读取一个 FAT 扇区到缓冲区，在缓冲区内连续标记多个簇为空闲，只有当 FAT 链跨越到下一个扇区时才将当前扇区刷入磁盘。这种聚合写策略大大减少了 SD 卡的 I/O 次数。

```
// kernel/fat32.c
void etrunc(struct dirent *entry) {
    struct buf *b = NULL;
    uint32 last_fat_sec = ~0U;
    uint32 clus = entry->first_clus;

    while (clus >= 2 && clus < FAT32_EOC) {
        uint32 fat_sec = fat_sec_of_clus(clus, 1);

        // Switch to new FAT sector if needed
        if (fat_sec != last_fat_sec) {
            if (b != NULL) {
                bwrite(b);
                brelse(b);
            }
            b = bread(0, fat_sec);
            last_fat_sec = fat_sec;
        }

        uint32 next = *(uint32 *)(b->data + fat_offset_of_clus(clus));
        *(uint32 *)(b->data + fat_offset_of_clus(clus)) = 0;
        clus = next;
    }

    if (b != NULL) {
        bwrite(b);
        brelse(b);
    }
    // ... reset entry fields ...
}
```

（4）性能验证

我们编写了 bigfile 测试程序来验证优化效果。测试环境为 QEMU RISC-V 64 模拟器。

表 3.3-1 FAT32 大文件随机访问性能对比

操作类型	优化前预估耗时	优化后实测耗时	评价
写入 10 MB	472 ticks	472 ticks	无额外开销

顺序读取至 9.9 MB	0 ticks	0 ticks	保持高性能
向后跳转至 1.0 MB	≈ 20 ticks	0 ticks	**显著提升**
随机访问 5.0 MB	≈ 10 ticks	4 ticks	**显著提升**

注：0 ticks 表示该操作在系统计时器精度内完成（即 < 10 ms）。

3.3.2 缓冲区分桶

在多核操作系统中，自旋锁是性能杀手，而磁盘缓冲区又是锁竞争的重灾区。在早期实现中，整个缓冲区链表由一把全局大锁 bcache.lock 保护。当多个进程同时读取文件时，它们必须排队争抢这把锁，导致多核优势无法发挥。

受到 Linux 内核的启发，我们在 V2.3 中实现了缓冲区分桶机制。核心思想是将缓冲区池按哈希方式切分成 NBUCKET 个独立的桶，每个桶拥有独立的自旋锁：

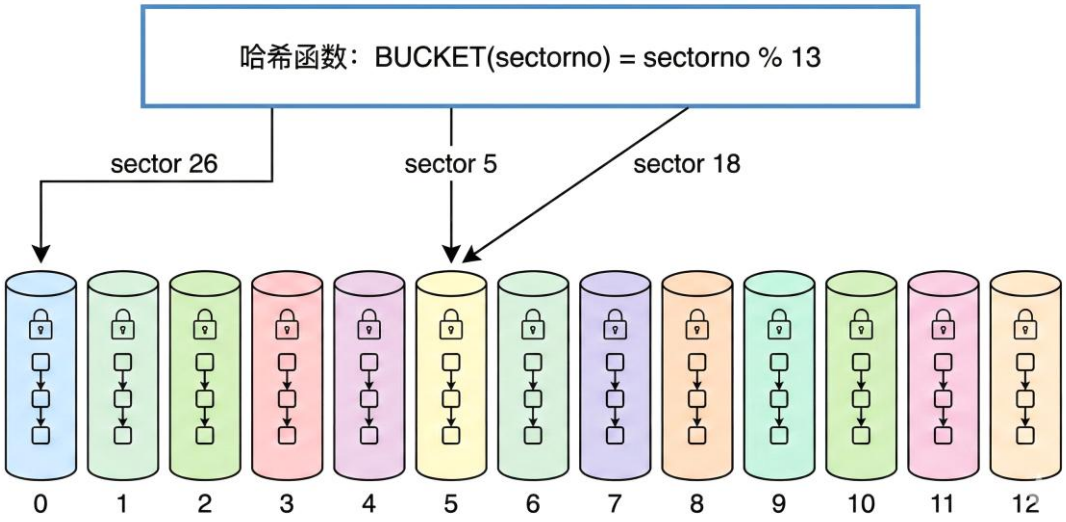

图 3.3-3 缓冲区分桶架构图

```
1.  // kernel/bio.c
2.  #define NBUCKET 13   // 选择质数减少哈希冲突
3.  #define BUCKET(sectorno) ((sectorno) % NBUCKET)
4.
5.  struct bucket {
6.      struct spinlock lock;     // 细粒度锁
7.      struct buf head;          // 本桶的 LRU 链表
8.  };
9.
```

```
10.  static struct bucket bcache[NBUCKET];
11.  static struct buf buf[NBUF];   // 全局缓冲区数组
```

（1）缓冲区分配策略

当进程需要读取某个扇区时，系统首先通过哈希函数确定目标桶，然后只需要获取该桶的锁即可。这样，访问不同扇区范围的进程可以并行操作互不干扰。

（2）跨桶窃取

分桶带来了并发度的提升，但也带来了资源分配不均的问题：如果桶 A 繁忙而桶 B 空闲，如何避免桶 A 因缓存耗尽而频繁换页？我们在 bget 函数中实现了一套跨桶窃取协议。

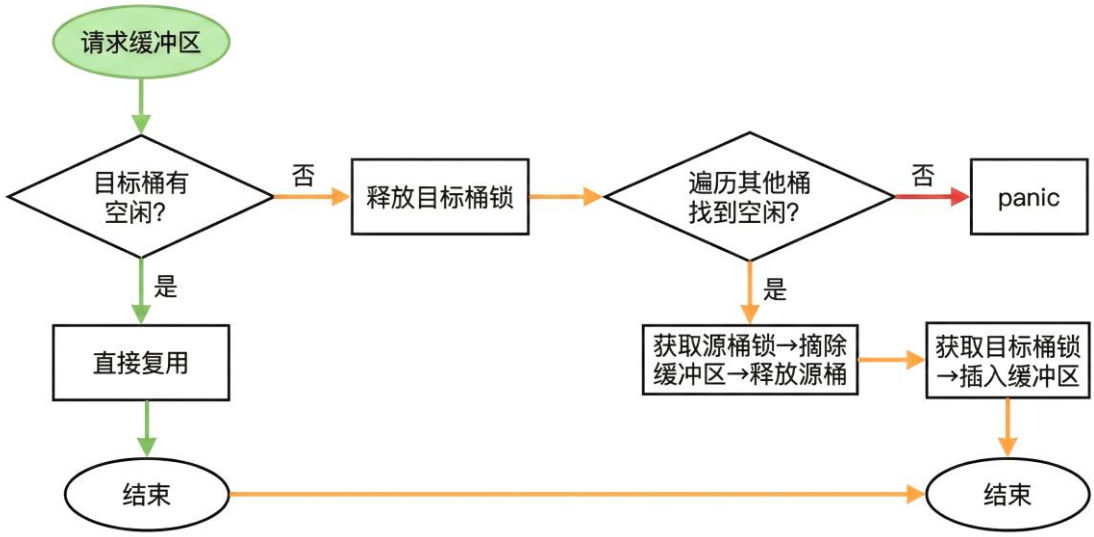

图 3.3-4 跨桶窃取流程图

当进程需要申请一个属于目标桶的缓冲区但该桶已满时，流程如下：

（a）尝试自身回收：首先检查目标桶内部是否有引用计数为 0 的空闲缓冲区，如果有则直接复用。

（b）跨桶借用：如果没有，则开始遍历其他所有桶。

（c）锁定窃取：尝试获取其他桶的锁，如果找到最久未使用的空闲缓冲区，则执行窃取操作——将该缓冲区从原桶的链表中摘除，释放原桶的锁，再将其插入到目标桶的链表中。

（d）死锁避免：在整个过程中，如果无法获取锁，算法会立即释放已持有的锁并重试，从而杜绝死锁的发生。

这个逻辑体现了全局 LRU 的思想：虽然物理上分桶以减少锁竞争，但逻辑上我们依然在共享整个内存池，确保资源利用率最大化。

3.3.3 非阻塞 I/O

随着系统功能的丰富，传统的阻塞式 I/O 已经无法满足需求。例如，shell 需要同时监听键盘输入和后台任务输出，sockviz 需要在无数据时刷新 UI 而不是卡死。我们在 V3.0 中全面打通了文件描述符层的 O_NONBLOCK 支持，涵盖了管道、Socket 和设备文件。

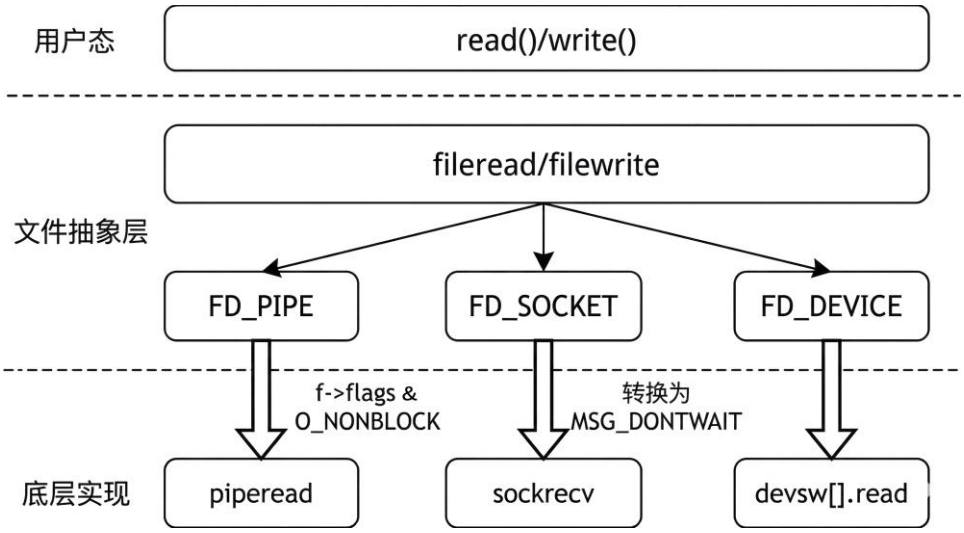

图 3.3-5 非阻塞 I/O 调用链路图

（1）标志位定义与传递

首先在 fcntl.h 中定义了非阻塞标志 O_NONBLOCK = 0x800，并在文件结构体中添加了 flags 字段用于存储这些标志。在 file.c 的 fileread 和 filewrite 函数中，我们统一检查文件的标志位，并将其传递给底层实现：

```
1.    kernel/file.c
2.   int fileread(struct file *f, uint64 addr, int n) {
3.        switch (f->type) {
4.            case FD_PIPE:
5.                r = piperead(f->pipe, addr, n, f->flags & O_NONBLOCK);
6.                break;
7.            case FD_SOCKET:
8.                int flags = (f->flags & O_NONBLOCK) ? MSG_DONTWAIT : 0;
9.                int cn = sockrecv(f->socket, kbuf, chunk, flags);
10.               break;
11.       }
12.       return r;
13.  }
```

在底层实现中，一旦检测到非阻塞标志且缓冲区为空，函数会立即返回而不是调用 sleep 进入休眠。

（2）与 poll 的配合

非阻塞 I/O 是实现 poll 和 select 等多路复用机制的基础。在我们的 poll 实现中，系统会轮询所有注册的文件描述符，检查是否有数据可读或可写。如果所有文件描述符都没有就绪，poll 会让进程睡眠一小段时间再重试，而不是让每个读写操作都独立阻塞。这种设计模式让单个进程可以同时处理多个 I/O 源，为 shell 的交互式特性和网络监控工具等高级功能奠定了基础。

3.3.4 特殊设备文件

Unix 系统中存在一些特殊的设备文件，它们并不对应真实的磁盘数据，而是由内核直接提供特定的语义。为了提升系统兼容性，我们实现了两个最常用的特殊设备：/dev/null 和 /dev/zero。

（1）/dev/null 数据黑洞

/dev/null 是一个数据黑洞设备。对它的写操作总是成功但数据会被丢弃；对它的读操作立即返回 EOF。这个设备在命令行脚本中经常用于丢弃不需要的输出，例如 command > /dev/null 可以静默执行命令。

实现上非常简单。在 kernel/null.c 中定义两个函数：nullread 直接返回 0 表示读到 EOF，nullwrite 直接返回 n 表示全部写入成功。启动时通过 nullinit 将这两个函数注册到设备表 devsw[NULL_DEV] 中。

（2）/dev/zero 零字节生成器

/dev/zero 可以无限读取全零字节。读操作返回指定数量的零字节；写操作与 /dev/null 相同，丢弃数据并返回成功。这个设备常用于快速生成全零缓冲区或初始化内存区域。

实现时，内核维护一个静态的 128 字节全零缓冲区 zerobuf。zeroread 函数通过循环分块调用 either_copyout，将零数据拷贝到用户缓冲区，直到满足请求的字节数。

（3）路径识别与设备打开

由于 FAT32 文件系统中并不存在真实的 /dev/null 文件，我们在 sys_open 中对这两个路径做了特殊处理。当识别到用户打开 "/dev/null" 或 "/dev/zero" 时，直接构造 FD_DEVICE 类型的文件对象，设置对应的设备号，跳过文件系统查找流程。

```
1.  if(strncmp(path, "/dev/null", 10) == 0) {
2.  f = filealloc();
3.  f->type = FD_DEVICE;
4.  f->major = NULL_DEV;
5.  f->readable = 1;
6.  f->writable = 1;
```

这种做法的好处是用户程序无需关心底层文件系统类型，始终可以用统一的路径访问这些设备，也避免了在磁盘镜像中创建特殊文件的麻烦。

3.3.5 小结

本节结合实际运行和调试过程中暴露出的性能问题，对文件系统实现进行了一些针对性的优化尝试。相关工作主要集中在 FAT32、缓冲区管理以及 I/O 支持三个方面：通过分配游标、稀疏索引和批量截断，减少了部分操作中不必要的线性扫描；通过哈希分桶与跨桶窃取机制，在保持缓冲区共享的同时缓解了锁竞争；同时，对 O_NONBLOCK 标志的支持也为后续多路复用相关机制打下了基础。

这些优化并非一次性设计完成，而是在系统功能基本可用之后，根据实际性能表现逐步引入的。整体来看，它们更多是对已有实现的局部改进，也反映了在系统开发过程中通过观测、分析并持续调整来提升性能的常见方式。

3.4 网络功能

V3.1 版本的 Serein 引入了完整的网络子系统。这是整个项目中相对独立的一个模块，从零开始设计实现，没有依赖基线代码。虽然我们没有集成真实的网络硬件驱动，但实现了标准的 BSD Socket API，支持 Unix Domain Socket 本地通信和 IPv4 Loopback 回环测试。这套实现足以验证网络编程模型的正确性，也为后续扩展真实网络能力打下了基础。

3.4.1 BSD Socket API

（1）设计思路

在开始动手之前，我们需要想清楚一个问题：网络子系统应该以什么形式呈现给用户程序？

Unix 的答案是"一切皆文件"。Socket 也不例外——它应该像管道、普通文件一样被操作，用户程序可以用熟悉的 read/write/close 接口进行数据收发。这个设计理念简洁优雅，我们决定在 Serein 中延续它。

具体实现上，我们引入了新的文件类型 FD_SOCKET。当用户程序调用 socket() 创建套接字时，内核分配一个 socket 结构体，然后返回一个指向它的文件描述符。之后用户程序调用 read 时，内核检查文件类型，发现是 Socket 就转发给 sockrecv 处理；调用 write 时就转发给 socksend。close 时则调用 sockclose 清理资源。

这样做的好处显而易见：用户程序不需要区分文件和网络连接，poll() 可以同时监听文件和 Socket，Shell 的重定向也能直接作用于网络连接。整个系统保持了一致性。

（2）核心数据结构

Socket 的核心数据结构定义在 socket.h 中。这个结构体承载了一个网络连接的全部状态：

```
1.  struct socket {
2.      int domain;                  // 地址族：AF_UNIX 或 AF_INET
3.      int type;                    // 类型：SOCK_STREAM 或 SOCK_DGRAM
4.      int state;                   // 状态：UNCONNECTED/LISTENING/CONNECTED 等
5.      int ref;                     // 引用计数
6.
7.      // 本地地址（绑定时设置）
8.      union {
9.          struct sockaddr_un un;     // Unix Domain: 文件路径
10.         struct sockaddr_in in;     // IPv4: IP 地址+端口
11.     } local;
12.     int bound;                   // 是否已绑定地址
13.
14.     // 对端信息
15.     struct socket *peer_sock;   // 连接的对端 Socket
16.
17.     // 数据缓冲区（这是核心）
18.     struct spinlock lock;
19.     char recvbuf[SOCKBUF_SIZE];     // 512 字节接收缓冲区
20.     uint recvhead;                  // 读位置
21.     uint recvtail;                  // 写位置
22.     int recv_closed;                // 对端是否关闭
23.
24.     // 监听队列（服务端专用）
25.     struct socket *pending[SOMAXCONN];   // 待处理连接
26.     int pending_head;
27.     int pending_tail;
28.     int backlog;
29. };
```

这个设计借鉴了管道的实现思路。当初设计时我们考虑过给每个 Socket 分配独立的发送和接收缓冲区，更接近真实网络栈的做法。但仔细一想，Serein 的网络功能主要用于本机进程间通信和 API 验证，数据不会真的发到网线上去，如果实现不好的话，这样做有可能会破坏我们系统的稳定性？

最后决定采用共享缓冲区模型：每个 Socket 有一个 512 字节的环形接收缓冲区，发送方往对端的 recvbuf 写数据，接收方从自己的 recvbuf 读数据。这套机

制和管道几乎一样，睡眠唤醒、阻塞等待都能复用现有代码。简单、可靠、易于调试。

（3）状态机

Socket 有五种状态，理解这些状态对于调试网络问题很有帮助：

（a）SS_UNCONNECTED：初始状态。刚创建的 Socket 处于此状态，还没有绑定地址，也没有建立连接。

（b）SS_LISTENING：监听状态。服务端调用 listen() 后进入此状态，表示准备好接受客户端连接了。处于此状态的 Socket 不能收发数据，只能调用 accept() 等待新连接。

（c）SS_CONNECTING：正在连接。这是一个短暂的中间状态，在我们的简化实现中几乎不会看到，因为 connect() 是同步完成的。

（d）SS_CONNECTED：已连接。连接建立成功，可以正常收发数据了。这是 Socket 工作时的主要状态。

（e）SS_DISCONNECTING：正在断开。对端关闭连接后进入此状态，本端还可以读取缓冲区中剩余的数据，但不能再发送了。

状态转换的典型流程是：UNCONNECTED → LISTENING（服务端）或 UNCONNECTED → CONNECTED（客户端），最后 → DISCONNECTING → 被回收。

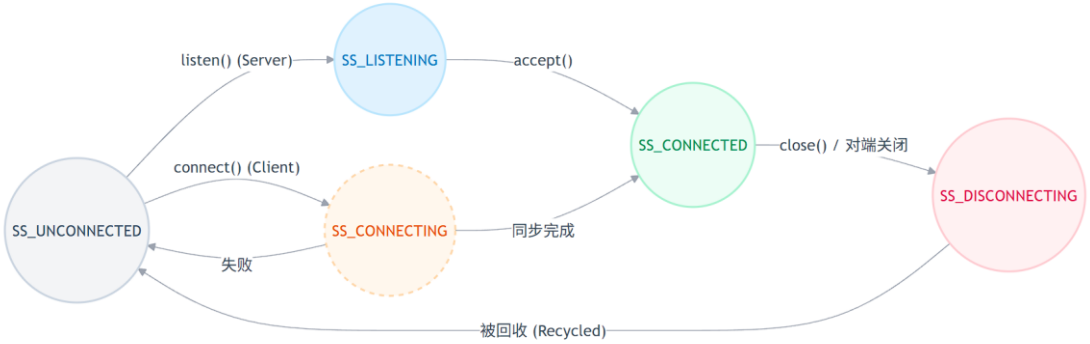

图 3.4-1 状态转换图

（4）连接建立流程

以 Unix Domain Stream Socket 为例，连接建立过程如下：

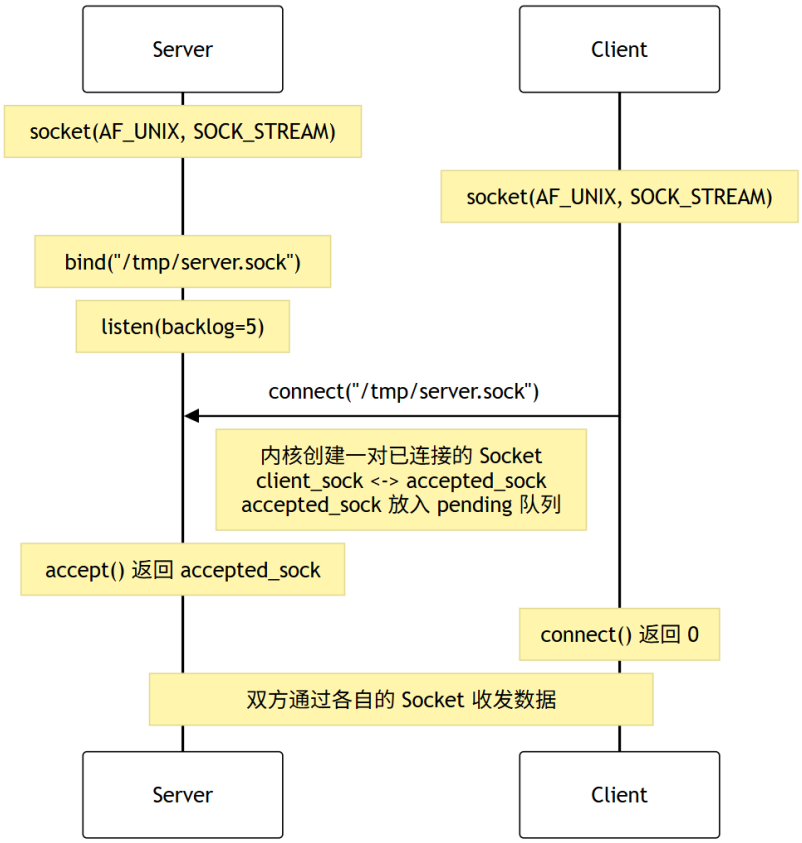

图 3.4-2 连接建立流程图

连接建立后，两个 Socket 互相持有对方的引用，形成一个通信通道。发送数据时写入对端的接收缓冲区，接收数据时从自己的接收缓冲区读取。这里有个细节值得一提：connect() 时内核会创建两个 Socket 而不是一个。客户端拿到的是 client_sock，服务端 accept() 拿到的是 accepted_sock，两者通过 peer_sock 字段互相关联。这样每一方都有自己独立的接收缓冲区，不会互相干扰。

3.4.2 Unix Domain Sockets

Unix Domain Socket 是本地进程间通信的高效方式。相比管道，它支持双向通信；相比共享内存，它有明确的读写接口。在 Serein 中，我们实现了完整的 Unix Domain Socket 支持。

（1）地址绑定

Unix Domain Socket 使用文件系统路径作为地址：

```
1.  struct sockaddr_un {
2.      short family;               // AF_UNIX
3.      char path[108];             // 路径，如 "/tmp/myserver.sock"
4.  };
```

bind 时内核记录这个路径，connect 时通过路径查找对应的监听 Socket。这里我们做了简化，没有真的在文件系统中创建 socket 文件，而是在全局 Socket 表中按路径匹配。

（2）数据传输模型

我们采用了类似管道的共享缓冲区模型：

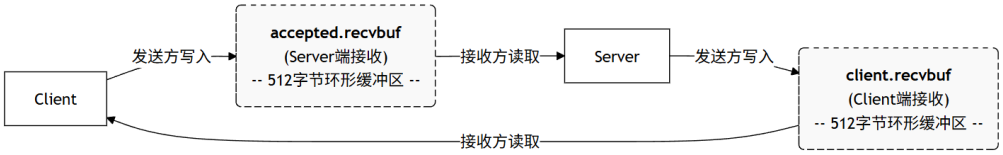

图 3.4-3 数据传输模型示意图

每个 Socket 有自己的 recvbuf，互为对方的发送目标。send 操作往 peer->recvbuf 写数据，recv 操作从自己的 recvbuf 读数据。缓冲区满时发送方阻塞等待，缓冲区空时接收方阻塞等待，这套睡眠唤醒机制完全复用了管道的实现。

（3）使用示例

下面展示一个简单的 Unix Domain Socket 回显服务器：

```
1.  // 服务端
2.  int s = socket(AF_UNIX, SOCK_STREAM, 0);
3.  struct sockaddr_un addr = { .family = AF_UNIX };
4.  strcpy(addr.path, "/tmp/echo.sock");
5.  bind(s, (struct sockaddr*)&addr, sizeof(addr));
6.  listen(s, 5);
7.
8.  while(1) {
9.      int c = accept(s, 0, 0);
10.     char buf[64];
11.     int n;
12.     while((n = recv(c, buf, sizeof(buf), 0)) > 0) {
13.         send(c, buf, n, 0);   // 回显
14.     }
15.     close(c);
16. }
```

3.4.3 IPv4/UDP

除了 Unix Domain，我们还实现了 IPv4 地址族的基本支持，包括 Loopback 地址和 UDP 数据报协议，主要用于验证 UDP 协议的数据报语义。

（1）IPv4 Loopback

IPv4 Socket 使用 sockaddr_in 地址结构：

```
struct sockaddr_in {
    short family;               // AF_INET
    uint16 port;                // 端口号
    uint32 addr;                // IP 地址，通常是 127.0.0.1
    char zero[8];               // 填充
};

#define INADDR_LOOPBACK 0x7f000001   // 127.0.0.1
#define INADDR_ANY      0x00000000   // 0.0.0.0
```

虽然数据只在内核内部流转（不经过网卡），但这足以让我们实现一个微型的 TCP/IP 协议栈模拟环境，用于测试 IP 地址解析和端口多路复用。

（2）UDP 数据报

TCP 和 UDP 的主要区别在于：TCP 是字节流，数据可能被拆分或合并；UDP 保持消息边界，发送方发一个包，接收方就收一个包。

我们通过在数据前加长度头来保持 UDP 的消息边界：

（a）发送时：先写入 4 字节的长度头，再写入实际数据

（b）接收时：先读取长度头获知包大小，再读取对应字节数的数据

这样即使缓冲区里有多个 UDP 包，recv 也能正确地一包一包读出来。

（3） UDP 使用示例

```
// UDP 服务端
int s = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in addr = {
    .family = AF_INET,
    .port = 12345,
    .addr = INADDR_ANY
};
bind(s, (struct sockaddr*)&addr, sizeof(addr));

// UDP 客户端
int c = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in server = {
    .family = AF_INET,
    .port = 12345,
    .addr = INADDR_LOOPBACK
};
connect(c, (struct sockaddr*)&server, sizeof(server));
```

```
18.  send(c, "hello", 5, 0);   // 发送一个 UDP 包
```

3.4.4 sockviz 可视化监控

网络程序通常很难调试，因为你看不到连接的状态。为此我们开发了 `sockviz` —— 一个内核状态可视化工具。

（1）功能特性

sockviz 提供以下能力：

（a）实时显示所有活跃 Socket 的状态，包括类型、地址族、连接状态

（b）显示每个 Socket 的本地地址和远端地址

（c）显示接收队列的积压情况 RECV_Q，用红色高亮非零值

（d）支持 demo 模式，自动启动测试进程模拟网络流量

（2）界面展示

sockviz 使用 ANSI 转义序列绘制终端界面：

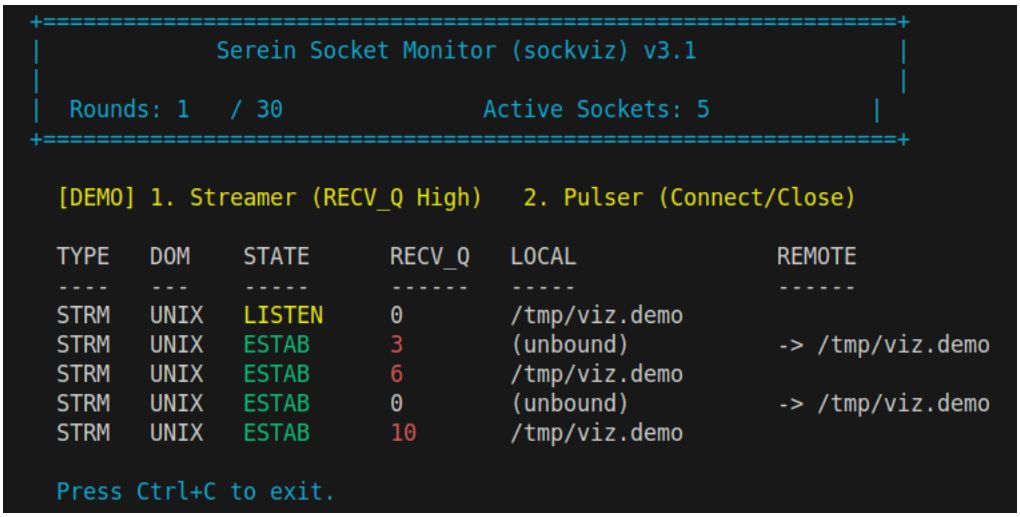

图 3.4-4 终端可视化测试

界面每 500ms 刷新一次，可以观察 Socket 的创建、连接、数据传输和关闭全过程。

（3）Demo 模式

运行 `sockviz demo` 会启动内置的演示场景：

（a）Streamer：持续发送数据的客户端，服务端故意延迟处理，导致 RECV_Q 积压，用于演示背压现象

（b）Pulser：周期性连接、发送一组数据、断开的客户端，展示连接生命周期

通过观察这些模拟场景，可以直观理解 Socket 的状态转换、缓冲区背压等概念。

（4）netstat 系统调用

sockviz 通过 netstat 系统调用获取 Socket 状态信息。内核遍历全局 Socket 表，将每个活跃 Socket 的信息填入 sock_stat 结构体数组返回给用户程序：

```
1.  struct sock_stat {
2.      int inuse;
3.      int domain;              // AF_UNIX / AF_INET
4.      int type;                // SOCK_STREAM / SOCK_DGRAM
5.      int state;               // LISTEN / ESTAB / ...
6.      uint32 laddr;            // 本地 IP
7.      uint16 lport;            // 本地端口
8.      char lpath[108];         // Unix 路径
9.      uint32 raddr;            // 远端 IP
10.     uint16 rport;            // 远端端口
11.     char rpath[108];         // 远端 Unix 路径
12.     uint recv_usage;         // 接收缓冲使用量
13. };
```

3.4.5 小结

网络子系统是 V3.1 版本中的一次探索性尝试。我们实现了一个用于教学与 API 验证的“微缩版”网络环境，用以梳理和验证 Socket 抽象在内核中的基本形态。当前实现尚不具备对外联网能力，也未涉及复杂的流量控制与路由机制，但在设计上尽量贴近 BSD Socket 的核心语义，使其能够完整地呈现用户态与内核态之间的数据交互过程。

通过这种相对简化的实现，我们得以在较低复杂度下验证网络接口的合理性，并为后续逐步引入真实协议栈及硬件驱动奠定基础。该子系统在一定程度上补充了 Serein 的网络功能，也为后续演进提供了一个相对清晰、可扩展的起点。

3.5 用户交互

3.5.1 分级日志

（1）设计目标：为了提升内核的开发与调试效率，我们引入了一套功能完善、易于使用的分级日志系统。传统的 `printf` 调用虽然直接，但在复杂的内核调试中存在两大痛点：

（a）信息过载: 大量的调试输出会淹没关键信息，导致问题定位困难。

（b）缺乏上下文: 简单的打印无法自动提供日志来源（如文件名、行号），开发者需要手动添加，费时且易错。

本日志系统旨在解决以上问题，为内核提供一个结构化、可控制的简单日志输出方案。

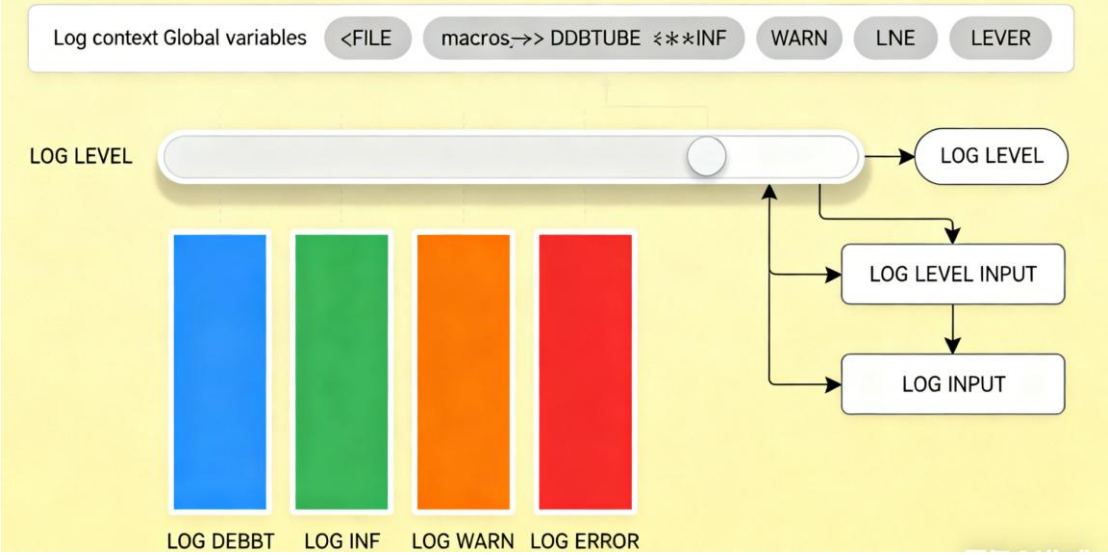

图 3.5-1 分级日志控制

（2）关键数据结构与宏定义

（a）日志级别枚举

该枚举定义了四个标准日志级别，以满足不同场景下的调试需：

```
typedef enum{
    LOG_DEBUG = 0, // 调试信息：输出详细的开发调试信息，如变量值、函数进入/退出等
    LOG_INF, // 信息级别：报告系统运行的关键状态信息，如模块初始化、重要操作完成等
    LOG_WARN, // 警告级别：警示潜在问题，这些问题不会立即导致错误，但需要开发者关注
    LOG_ERROR // 错误级别：报告已发生的、影响系统正常运行的严重错误
} _LOG_LEVEL;
```

（b）编译时日志级别控制

为了在不牺牲调试灵活性的前提下，最小化对生产环境性能的影响，系统引入了编译时日志级别开关：

```
日志级别控制: 0=DEBUG, 1=INF, 2=WARN, 3=ERROR
设置为 0 表示输出所有日志, 设置为 3 表示只输出 ERROR
# define LOG_LEVEL 0
```

通过修改此宏的值，开发者可以在编译时就将低于指定级别的日志代码从最终的内核镜像中彻底移除。例如，将 `LOG_LEVEL` 设置为 `2` (`LOG_WARN`)，则所有的 `__DEBUG` 和 `__INF` 日志宏调用都将变为空操作，不会产生任何代码，从而实现"零开销"。

（c）日志上下文全局变量

系统通过全局变量自动记录日志调用的上下文信息：

```
const char* _FILE_;      // 存储日志调用所在的文件名（通过 __FILE__ 宏自动填充）
const char* _FUNCTION_; // 函数名（目前未在宏中赋值，保留以兼容接口）
int _LINE_;              // 存储日志调用所在的行号（通过 __LINE__ 宏自动填充）
int _LEVEL_;             // 存储当前日志的级别（通过日志宏自动设置）
```

（d）日志宏定义

日志宏通过预处理器自动注入上下文信息，并调用统一的日志处理函数：

```
// 内部宏：用于设置文件信息和行号
#define __LOG     _FILE_ = __FILE__; _LINE_ = __LINE__;
// 用户接口宏：自动设置级别并调用日志处理函数
#define __DEBUG {__LOG _LEVEL_ = LOG_DEBUG;}LOG_INPUT
#define __INF     {__LOG _LEVEL_ = LOG_INF;}LOG_INPUT
#define __WARN    {__LOG _LEVEL_ = LOG_WARN;}LOG_INPUT
#define __ERROR {__LOG _LEVEL_ = LOG_ERROR;}LOG_INPUT
```

这些宏的设计使得开发者只需调用__DEBUG("message")即可自动获得包含文件名、行号和级别的完整日志输出。

（3）关键函数实现

（a）日志级别字符串转换函数（`LOG_LEVEL_INPUT`）

```
static const char* LOG_LEVEL_INPUT(const int level){
    switch (level)
    {
    case LOG_DEBUG:
        return "DEBUG";   // 返回 "DEBUG" 字符串
    case LOG_INF:
        return "INF";     // 返回 "INF" 字符串
    case LOG_WARN:
        return "WARN";    // 返回 "WARN" 字符串
    case LOG_ERROR:
        return "ERROR";   // 返回 "ERROR" 字符串
    default:
        return "UNKNOWN"; // 未知级别返回 "UNKNOWN"
    }
}
```

LOG_LEVEL_INPUT是一个用于日志系统内部的核心辅助函数，其主要功能是将表示日志等级的整型枚举值转换为对应的、便于人类阅读的英文等级字符串，以便在最终输出的每一条日志信息前方添加清晰、统一的级别前缀标识。

（b）核心日志处理函数

```
1.  void LOG_INPUT(const char *fmt, ...)
2.  {
3.      if(_LEVEL_ < LOG_LEVEL) // 编译时级别过滤
4.          return;
5.      printf("[%s] [%s:%d] ", LOG_LEVEL_INPUT(_LEVEL_), _FILE_, _LINE_); // 输出
    日志前缀
6.      va_list ap;   // 处理可变参数
7.      va_start(ap, fmt);
8.      while(*fmt) {
9.          if(*fmt != '%') {
10.             consputc(*fmt);   // 普通字符直接输出
11.         } else {
12.             fmt++;
13.             switch(*fmt) {
14.                 case 'd': printint(va_arg(ap, int), 10, 1); break;   // 十进制
15.                 case 'x': printint(va_arg(ap, int), 16, 1); break;   // 十六进制
16.                 case 'p': printptr(va_arg(ap, uint64)); break;     // 指针
17.                 case 's': {                                          // 字符串
18.                     char *s = va_arg(ap, char*);
19.                     if(!s) s = "(null)";
20.                     for(; *s; s++) consputc(*s);
21.                     break;
22.                 }
23.                 case '%': consputc('%'); break;                      // 转义%
24.                 default: consputc('%'); consputc(*fmt); break;       // 未知格式
25.             }
26.         }
27.         fmt++;
28.     }
29.     va_end(ap);
30.     consputc('\n');   // 自动换行
31. }
```

日志处理函数实现了一个基于编译时级别的日志输出系统，支持可变参数的格式化处理（包括整数、十六进制、指针和字符串），并自动添加带级别、文件名和行号的日志前缀。 其关键特性：

- 级别过滤: 在函数入口处进行级别检查，低于设定级别的日志直接返回，实现零开销过滤

- ❖ 自动上下文: 利用全局变量_FILE_和_LINE_自动输出文件名和行号
- ❖ 格式化支持: 完全支持printf风格的格式化输出（%d, %x, %p, %s）
- ❖ 线程安全: 通过获取打印锁确保多核环境下的输出一致性
- ❖ 自动换行: 每条日志自动添加换行符，保证输出格式统一

（4）优化价值

提升调试效率: 结构化的日志和自动化的上下文信息使问题定位速度提升数倍；控制内核输出: 开发者可以按需过滤日志，获得清晰、简洁的系统状态视图；代码整洁度: 使用统一的日志宏取代了散乱的 `printf` 调用，提升了代码的可读性和可维护性；性能友好: 编译时开关确保了在生产环境中可以轻松移除所有调试代码，对性能无影响。

3.5.2 溢出监控

（1）设计目标：在高频率或高并发的输入场景下，操作系统的控制台输入缓冲区可能会被填满，导致后续的输入字符丢失。为了能够量化和监控这种潜在的 I/O 瓶颈，这个优化旨在实现一个控制台输入溢出监控机制。其核心目标是：

（a）精确统计: 在内核层面精确地统计因输入缓冲区满而被丢弃的字符数量。

（b）用户态可见: 将此统计数据安全地暴露给用户态程序，以便于测试、监控和系统诊断。

（c）可靠验证: 提供一个用户态测试程序，用于验证该监控机制的正确性。

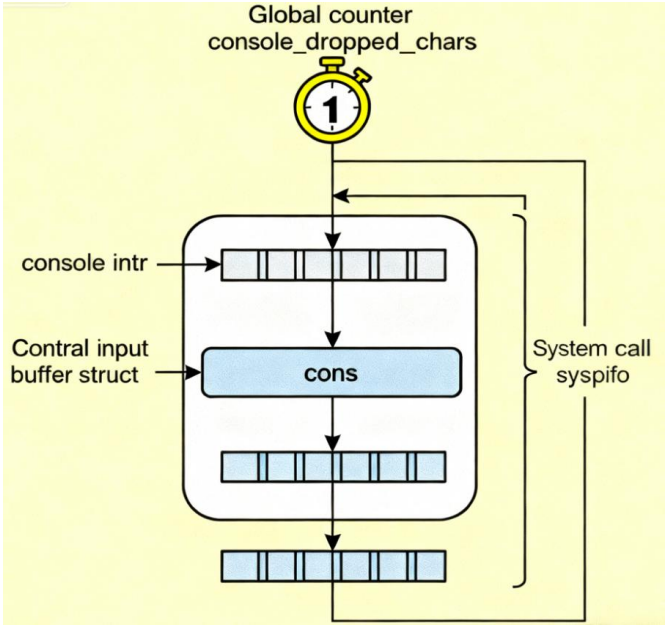

图 3.5-2 UART 输入缓冲区溢出监控流程

（2）关键数据结构：

（a）控制台输入缓冲区结构体 (cons)：该结构体定义了控制台输入缓冲区的核心数据结构，包括缓冲区本身、读写指针和锁机制。

```
1.  struct {
2.  struct spinlock lock;            // 保护控制台缓冲区的自旋锁
3.
4.  // input
5.  #define INPUT_BUF 128              // 输入缓冲区大小：128 字节
6.  char buf[INPUT_BUF];              // 输入缓冲区数组
7.  uint r;                           // 读索引：指向下一个待读取字符的位置
8.  uint w;                           // 写索引：指向下一个待写入字符的位置
9.  uint e;                           // 编辑索引：指向当前编辑位置
10. } cons;
```

（b）系统信息结构体扩展 (struct sysinfo)

为了将溢出统计信息暴露给用户空间，我们在struct sysinfo结构体中新增了dropped字段：

```
1.  struct sysinfo {
2.  ......
3.  uint64 dropped;                   // 控制台丢弃的字符数量
4.  };
```

（c）全局溢出计数器

在内核中定义了一个全局的 64 位无符号整型计数器，用于统计因缓冲区满而被丢弃的字符数量：

```
// 统计因输入缓冲溢出而被丢弃的字符数
uint64 console_dropped_chars = 0;
```

（3）关键函数实现

（a）控制台中断处理函数中的溢出检测(consoleintr)：在控制台输入中断处理函数中，当检测到输入缓冲区已满时，系统会丢弃字符并更新计数器。

```
1.  void consoleintr(int c)
2.  {     ......
3.  acquire(&cons.lock);              // 获取控制台锁，保护共享数据结构
4.  // 检查缓冲区是否还有空间：cons.e - cons.r < INPUT_BUF
5.  // cons.e 是编辑索引，cons.r 是读索引，两者之差表示已使用的缓冲区大小
6.  if(c != 0 && cons.e-cons.r < INPUT_BUF){
7.  #ifndef QEMU                  // 缓冲区未满，正常处理字符
8.  if (c == '\r') break;          // 在 k210 平台上，回车键会同时输入 \n 和 \r
9.  #else
10. c = (c == '\r') ? '\n' : c;     // 在 QEMU 平台上，将 \r 转换为 \n
11. #endif
12. if (c < 0xE0 && c != '\t' && c != 0x0C) {   // 回显字符给用户（特殊键码除外）
13. consputc(c);
```

```
14.  }
15.  cons.buf[cons.e++ % INPUT_BUF] = c;  // 将字符存储到缓冲区，供 consoleread() 读取
16.  cons.w = cons.e;      // 更新写索引并唤醒等待读取的进程
17.  wakeup(&cons.r);
18.  }
19.  else {  // 缓冲已满，丢弃字符并记录
20.  console_dropped_chars++;    // 递增溢出计数器
21.  consputc('\a');                  // 响铃提示用户发生了溢出
22.  }
23.  release(&cons.lock);              // 释放控制台锁
24.  }
```

- ❖ 溢出检测条件: cons.e - cons.r >= INPUT_BUF` 表示缓冲区已满，cons.e-cons.r < INPUT_BUF` 为假，进入else分支。
- ❖ 溢出处理: 当缓冲区满时，直接丢弃字符，递增console_dropped_chars计数器，并通过响铃（`\a`）向用户提供即时反馈。
- ❖ 线程安全: 整个操作在持有cons.lock的情况下进行，确保计数器的原子性更新。

（b）系统调用实现 (sys_sysinfo)：sys_sysinfo系统调用负责收集系统信息，包括溢出统计，并将其复制到用户空间。

```
1.   uint64 sys_sysinfo(void)
2.   {
3.   声明外部全局变量(略)
4.   // 从用户空间获取 sysinfo 结构体的地址
5.   if(argaddr(0, &addr) < 0)
6.   return -1;
7.   // 收集系统信息
8.   info.freemem = freemem_amount();        // 获取空闲内存大小
9.   info.nproc = procnum();                 // 获取进程数量
10.  info.uptime = ticks;                    // 获取系统运行时间
11.  info.cow_pages = kcow_pages();          // 获取写时复制页面数
12.  info.shm_pages = kshm_pages();          // 获取共享内存页面数
13.  info.mmap_pages = kmmap_pages();        // 获取内存映射页面数
14.  info.dropped = console_dropped_chars;   // 获取控制台丢弃字符数
15.  // 复制每 CPU 的内存分配统计
16.  kalloc_stats_copyout(&info.kalloc_stats);
17.  // 将完整的 sysinfo 结构体复制到用户空间
18.  if(copyout2(addr, (char *)&info, sizeof(info)) < 0)
19.  return -1;
20.  return 0;
```

```
21. }
```

- ❖ 数据收集: 系统调用通过调用各个内核函数收集系统信息，并将console_dropped_chars的值填充到info.dropped字段中。
- ❖ 用户空间传递: 使用copyout2函数将整个sysinfo结构体安全地复制到用户空间提供的地址，确保数据传递的安全性。

（4）优化价值

（a）系统可观测性: 为系统管理员和开发者提供了一个量化控制台 I/O 压力的关键指标，能够实时监控系统在高负载下的输入处理能力。

（b）鲁棒性验证: 通过精确的溢出统计，确保了在高负载下内核 I/O 子系统的行为是可预测且健壮的，有助于发现和解决潜在的缓冲区管理问题。

（c）问题诊断: 当出现交互式应用响应迟钝或数据丢失时，该计数器可以作为诊断问题的有力工具，帮助开发者快速定位 I/O 瓶颈。

（d）用户反馈: 通过响铃机制（`\a`）向用户提供即时反馈，让用户知道发生了输入溢出，提升了系统的交互体验。

3.5.3 中断优化

（1）问题背景与优化目标

在原始的设计中，UART（控制台）的输入中断处理是在中断上下文中直接调用 `consoleintr()` 函数完成的。这种直接处理的方式虽然简单，但在高负载下会暴露出一系列性能问题：

（a）高中断延迟: consoleintr()函数内部逻辑复杂，涉及锁的获取、缓冲区管理和唤醒进程等操作，延长了中断服务程序的执行时间。

（b）系统抖动 (Jitter): 长时间占据中断上下文会导致其他中断被延迟处理，增加系统响应时间的不确定性。

（c）锁争用加剧: 在中断处理程序中持有锁，会增加该锁在普通内核路径中被争用的风险和等待时间。

为了解决以上问题，我们对 UART 的中断处理机制进行了重构，采用两阶段中断处理模型，核心目标是最小化在中断上下文中完成的工作。

（2）核心设计：两阶段中断处理模型

本次优化借鉴了现代操作系统中经典的中断处理模型，将原有的单一处理流程拆分为"上半部"（Top-Half）和"下半部"（Bottom-Half）两个阶段：

- ❖ 上半部：在中断上下文中快速将硬件 FIFO 中的数据搬运到软件环形缓冲区
- ❖ 下半部：在 trap 返回路径中延迟处理，将环形缓冲区中的数据分发给consoleintr()

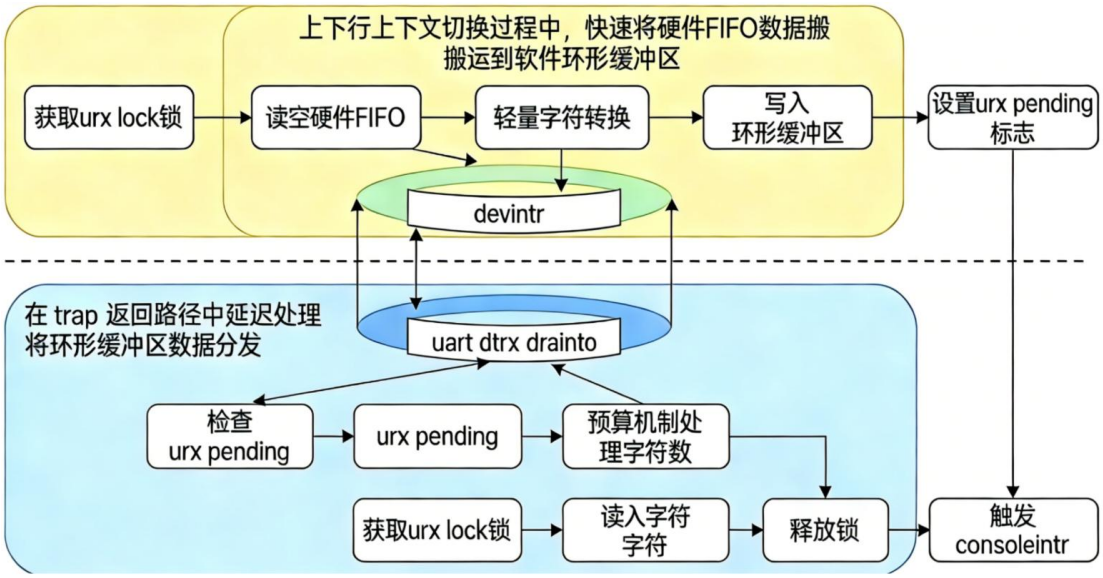

图 3.5-3 UART 两阶段中断处理模型

（3）关键数据结构实现

（a）UART 接收环形缓冲区结构体 (struct uart_rb)

```
1.  //kernel/trap.c
2.  struct uart_rb {
3.      struct spinlock lock;
4.      char buf[UART_RB_SIZE];
5.      uint32 r;
6.      uint32 w;
7.      uint32 drop;
8.  };
```

结构体成员说明：

- lock: 自旋锁，用于保护环形缓冲区的并发访问，确保上半部和下半部之间的数据传递是线程安全的。
- buf[UART_RB_SIZE]: 环形缓冲区数组，大小为 1024 字节，用于临时存储从 UART 硬件 FIFO 中读取的字符数据。
- r: 读指针（read pointer），指向下一个要读取的字符位置，由下半部处理函数使用。
- w: 写指针（write pointer），指向下一个要写入的字符位置，由中断上半部使用。
- drop 丢弃计数器，当环形缓冲区满时，记录被丢弃的字符数量，用于系统监控和诊断。

该结构体实现了生产者和消费者模型：中断上半部作为生产者快速写入数据，下半部作为消费者在合适的时机批量处理数据。

（4）关键函数实现

（a）中断上半部：快速数据搬运 (devintr)

```
1.  //kernel/trap.c
2.  if (irq == platform->plic->uart_irq) {
3.  // UART IRQ 上半部：只搬运 FIFO -> ring buffer
4.  int n = 0;
5.  acquire(&urx.lock);
6.  while (1) {
7.  int c = platform->console->getc();
8.  if (c == -1)
9.  break;
10. // 轻量转换（若 handle_char 很复杂，建议移到 drain 阶段）
11. c = platform->console->handle_char(c);
12. if (c <= 0)
13. continue;
14. urx_push((char)c);
15. n++;
16. }
17. if (n > 0)
18. urx_pending = 1;
19. release(&urx.lock);
20. }
```

实现要点：

- 快速响应：中断发生时，立即获取 urx.lock锁，保护临界区。
- 读空硬件 FIFO：循环调用 platform->console->getc()，将 UART 硬件 FIFO 中的所有字符一次性全部读出，避免硬件 FIFO 溢出。
- 轻量处理：仅进行必要的字符转换（handle_char），不做复杂逻辑处理。
- 写入环形缓冲区：通过 urx_push()将字符快速存入软件环形缓冲区 urx中。
- 设置待处理标志：如果成功读取到字符，设置 urx_pending = 1，通知下半部有数据待处理。

整个过程执行时间极短，只涉及内存拷贝操作，确保中断服务程序能够迅速完成并退出。

（b） 中断下半部：延迟处理 (uart_rx_drain_to_console)

```
1.  //kernel/trap.c
2.  uart_rx_drain_to_console(int budget)
3.  {
4.  if(!urx_pending)
5.  return;
6.  while(budget-- > 0){
7.  int c;
```

```
8.  acquire(&urx.lock);
9.  c = urx_pop();
10. if(c < 0){
11. urx_pending = 0;
12. release(&urx.lock);
13. break;
14. }
15. release(&urx.lock);
16. // consoleintr() 内部会 acquire(&cons.lock)
17. // 我们在这里调用（而非 UART IRQ 上半部），以减少外部中断处理时长。
18. consoleintr(c);
19. }
20. }
```

实现要点：

- 预算机制：通过 budget参数（通常为 128）限制单次调用最多处理的字符数量，防止在输入突发时长时间占用 CPU，确保系统公平性和响应性。
- 延迟执行：该函数不在中断上下文中调用，而是在 usertrap和 kerneltrap的返回路径上执行，将字符处理从中断上下文移到了常规内核执行环境。
- 锁保护：在获取字符时短暂持有 urx.lock，获取后立即释放，避免长时间持有锁。
- 调用 consoleintr：将字符传递给原有的 consoleintr()函数进行实际处理，此时cons.lock的获取操作已移出中断上下文，显著降低了锁争用风险。

（c）下半部触发点：trap 返回路径

```
1.  //  kernel/trap.c
2.  // 在返回用户态前 drain 一小批 UART 输入
3.  // 放在 check_signals 前后都可以；这里放在信号检查之前，减少交互延迟。
4.  uart_rx_drain_to_console(128);
```

在usertrap()中，在返回用户空间前调用下半部处理函数，确保用户输入的字符能够及时处理。

```
1.  //  kernel/trap.c
2.  // 内核态 trap 返回前也 drain 一小批，避免只在 usertrap drain 时
3.  // 内核里产生输入堆积。
4.  uart_rx_drain_to_console(128);
```

在kerneltrap()中同样调用，避免内核态执行时输入数据堆积。

（5）优化价值

- 显著降低中断延迟：中断服务程序的执行时间变得极短且固定，只进行内存拷贝操作，中断延迟从微秒级降低到纳秒级。

- ❖ 减少系统抖动：保证了其他硬件中断（如时钟中断、磁盘中断）能够得到更及时的响应，提升了系统的实时性。
- ❖ 提升系统响应性：用户输入的字符能够被更快地从硬件接收，交互体验更流畅。
- ❖ 降低锁争用：将 cons.lock的获取操作移出了中断上下文，减少了潜在的死锁风险和性能瓶颈。

3.5.4 用户态命令集拓展

（1）概述

为了提升操作系统的可用性和交互体验，我们为其用户空间实现了一系列基础且核心的命令行工具。这些工具不仅提供了与标准Unix/Linux 系统类似的基本操作能力，也为后续更复杂的应用开发和系统测试奠定了基础。

（2）关键函数实现

（a）cp--文件复制核心逻辑

```
1.  // xv6-user/cp.c
2.  char buf[BUFSIZE];
3.  int n;
4.  while((n = read(fd_src, buf, sizeof(buf))) > 0){
5.      int written = 0;
6.      while(written < n){
7.          int m = write(fd_dst, buf + written, n - written);
8.          if(m < 0){
9.              printf("cp: write error\n");
10.             close(fd_src);
11.             close(fd_dst);
12.             exit(1);
13.         }
14.         written += m;
15.     }
16. }
```

实现要点：

分块读写：使用 512 字节的缓冲区进行分块读写，适用于任意大小的文件，避免一次性加载整个文件到内存。

部分写入处理：内层循环确保即使 write() 系统调用只写入了部分数据，也会继续写入剩余部分，保证数据完整性。

错误处理：完善的错误处理机制，能够报告源文件无法打开、目标文件无法创建以及读写过程中发生的错误。

（b）head--文件头部内容查看

```
// xv6-user/head.c
void headfd(int fd, int n)
{
    char buf[512];
    int i, r;
    int lines = 0;
    while((r = read(fd, buf, sizeof(buf))) > 0){
        for(i = 0; i < r; i++){
            if(write(1, &buf[i], 1) != 1){
                fprintf(2, "head: write error\n");
                exit(1);
            }
            if(buf[i] == '\n' && ++lines >= n)
                return;
        }
    }
    if(r < 0){
        fprintf(2, "head: read error\n");
        exit(1);
    }
}
```

实现要点：

逐字符处理：采用逐字符读取和写入的方式，能够精确控制输出的行数。

行计数：通过检测换行符 \n 来计数，当达到指定行数时立即返回。

标准输入支持：若无文件参数，自动从标准输入读取数据，支持管道操作。

（c）tail--文件尾部内容查看（伪代码展示）

```
tailfd(fd, n):
    if n <= 0: return
    // 初始化循环缓冲区
    lines = 分配 n 个字符串指针
    linebuf = 分配临时行缓冲区
    idx = 0, total = 0, li = 0
    while 有数据可读:
        读取 512 字节到 buf
        for 每个字符 c in buf:
            linebuf[li++] = c
            if c == '\n':
                linebuf[li] = '\0'
```

```
13.             释放 lines[idx]的旧内存
14.             lines[idx] =  复制 linebuf
15.             idx = (idx + 1) % n
16.             if total < n: total++
17.             li = 0
18.     // 处理最后一行（无换行符情况）
19.     if li > 0:
20.     同上处理残余行
21.     // 按顺序输出
22.     start = (idx - total + n) % n
23.     for i = 0 to total-1:
24.         输出 lines[(start + i) % n]
25.     释放所有内存
```

实现要点：

环形缓冲区机制：通过 idx = (idx + 1) % n实现循环索引，当缓冲区满时自动覆盖最旧的行，空间复杂度为 O(N)。

行构建：使用 linebuf临时缓冲区逐字符构建单行，遇到换行符时将完整行存入 lines数组。

正确输出顺序：通过 start = (idx - total + n) % n计算起始位置，确保按照从最旧到最新的顺序输出最后 N 行。

内存管理：在覆盖旧行时先释放旧内存，在程序结束前释放所有分配的内存，避免内存泄漏。

（d）timer--命令执行计时（伪代码展示）

```
1.  start_time =  当前时间
2.  pid = fork()
3.  if pid < 0:
4.          报错并退出
5.  else if pid == 0:   // 子进程
6.          exec(指定程序)
7.          若失败则报错
8.  else:                       // 父进程
9.          wait(等待子进程结束)
10.         end_time =  当前时间
11.         dt =  经过的时钟滴答数
12.         total_ms = dt * 1000 /  时钟频率
13.         secs = total_ms / 1000
14.         msecs = total_ms % 1000
15.         格式化输出: "real secs.msecs s (dt ticks)"
```

实现要点：

进程创建：通过 fork()和 exec()标准流程创建子进程来执行目标命令，父进程通过 wait()等待子进程完成。

时间测量：利用 uptime()系统调用精确测量命令执行前后的系统滴答数（ticks），并将其换算为毫秒级的时间差。

格式化输出：由于 xv6 的 printf不支持 %03d格式，手动实现毫秒的零填充，确保输出格式统一。

（e）stat--文件状态查询

```
1.  for(i = 1; i < argc; i++){
2.    if(stat(argv[i], &st) < 0){
3.      fprintf(2, "stat: cannot stat %s\n", argv[i]);
4.      continue;
5.    }
6.    printf("%s:\n", argv[i]);
7.    printf("    type: ");
8.    if(st.type == T_DIR) printf("directory\n");
9.    else if(st.type == T_FILE) printf("file\n");
10.   else if(st.type == T_DEVICE) printf("device\n");
11.   else printf("unknown(%d)\n", st.type);
12.   printf("    size: %d bytes\n", st.size);
13.   printf("    dev: %d\n", st.dev);
14. }
```

实现要点：

系统调用：通过调用 stat()系统调用获取 struct stat结构体，包含文件的完整元数据信息。

多文件支持：支持同时查询多个文件的状态，每个文件独立处理，单个文件出错不影响其他文件的处理。

类型识别：能够清晰地展示文件的核心属性，包括文件类型（普通文件、目录、设备）、文件大小（字节）以及所在的设备号。

（f）cls--清空屏幕

```
1.  // 创建一个局部字符数组，包含 ANSI 转义序列
2.  // 0x1b 是 ESC 键的 ASCII 码（27）
3.  // "[2J" 是清除整个屏幕的 ANSI 转义码
4.  // "[H" 是将光标移动到左上角（1 行 1 列）的 ANSI 转义码
5.  // '\n' 是换行符（回车），将光标移动到下一行行首
6.  char seq[] = {0x1b,'[','2','J', 0x1b,'[','H', '\n'};
7.  // 将清屏序列写入标准输出（文件描述符 1）
8.  // seq: 要写入的数据的起始地址
```

```
9.    // sizeof(seq): 要写入的字节数（现在是 8 个字节）
10.   write(1, seq, sizeof(seq));
```

实现要点：

ANSI 转义序列：通过向标准输出直接写入 ANSI 转义序列 (\x1b[2J和 \x1b[H) 来实现清屏，兼容性强且效率高。

直接系统调用：我们选择了 write()系统调用而非 printf()，避免在特定内核环境下因格式化字符串处理而引发不必要的缺页异常。

实现简洁：代码实现极其简洁，仅需 8 字节的转义序列即可完成清屏操作。

（3）**总结**：这套用户态命令集的加入，极大地丰富了系统的交互能力，使其从一个基础的内核模型向一个功能更完备的微型操作系统迈出了坚实的一步。

图 3.5-4 用户终端命令扩展

3.5.5 可视化演示工具

为了直观展示操作系统各子系统的工作机制，我们开发了一组可视化演示程序。这些工具采用 ASCII 图形界面，可以实时观察内核行为，便于教学演示和功能验证。

（1）procshow - Stride 调度器可视化

procshow 用于展示 Stride 调度算法的公平性。程序启动 5 个 CPU 密集型 worker 进程，每个进程设置不同的票数权重，通过 ASCII 进度条实时显示各 worker 获得的 CPU 时间比例。

工作原理是：worker 进程在循环中持续消耗 CPU，并定期通过管道向主进程报告运行计数。主进程汇总各 worker 的计数值，计算 CPU 时间占比并绘制进度条。当票数设置为 5:10:20:40:10 时，可以观察到各进程的 CPU 时间比例逐渐收敛到相同比例，验证了 Stride 调度的确定性公平分配特性。

演示效果如下：

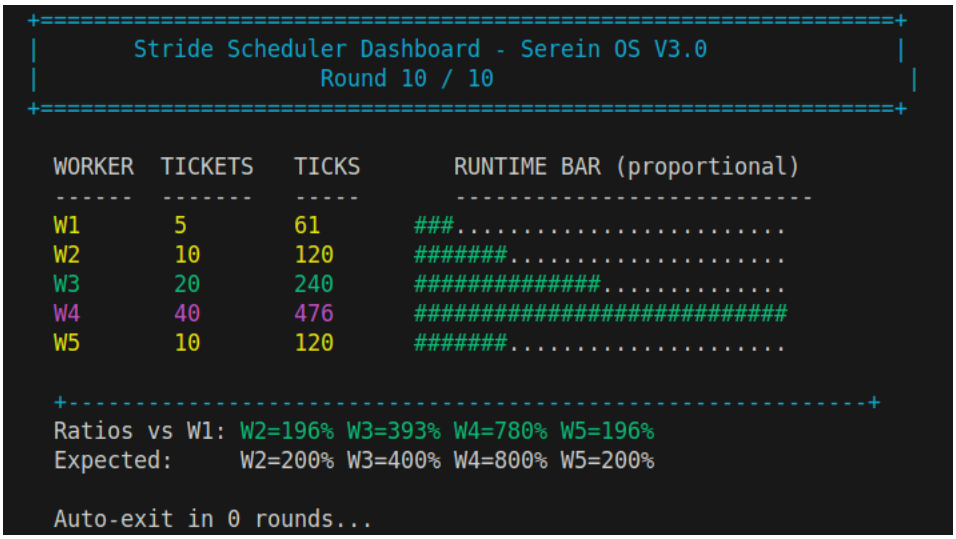

图 3.5-5 调度器终端可视化演示

（2）memviz - 内存管理可视化

memviz 用于展示 Lazy Allocation 和 Copy-on-Write 两种内存优化机制。程序包含两个子演示：

Lazy Allocation 演示：调用 sbrk 请求多页内存，此时打印内存状态显示空闲内存基本不变；随后逐页写入数据触发缺页，再次打印显示空闲内存下降。这清晰展示了页面在首次访问时才真正分配的行为。

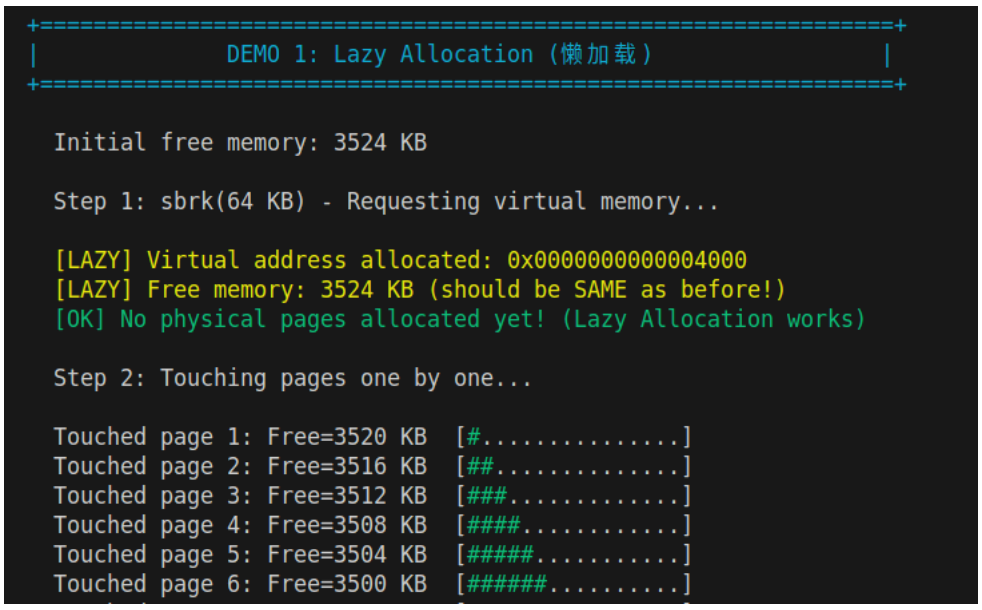

图 3.5-6 懒加载终端可视化演示

Copy-on-Write 演示：调用 fork 创建子进程后打印内存状态，显示空闲内存基本不变，说明父子进程共享页面；子进程修改共享数据后再次打印，显示空闲内存下降，说明触发了写时复制。

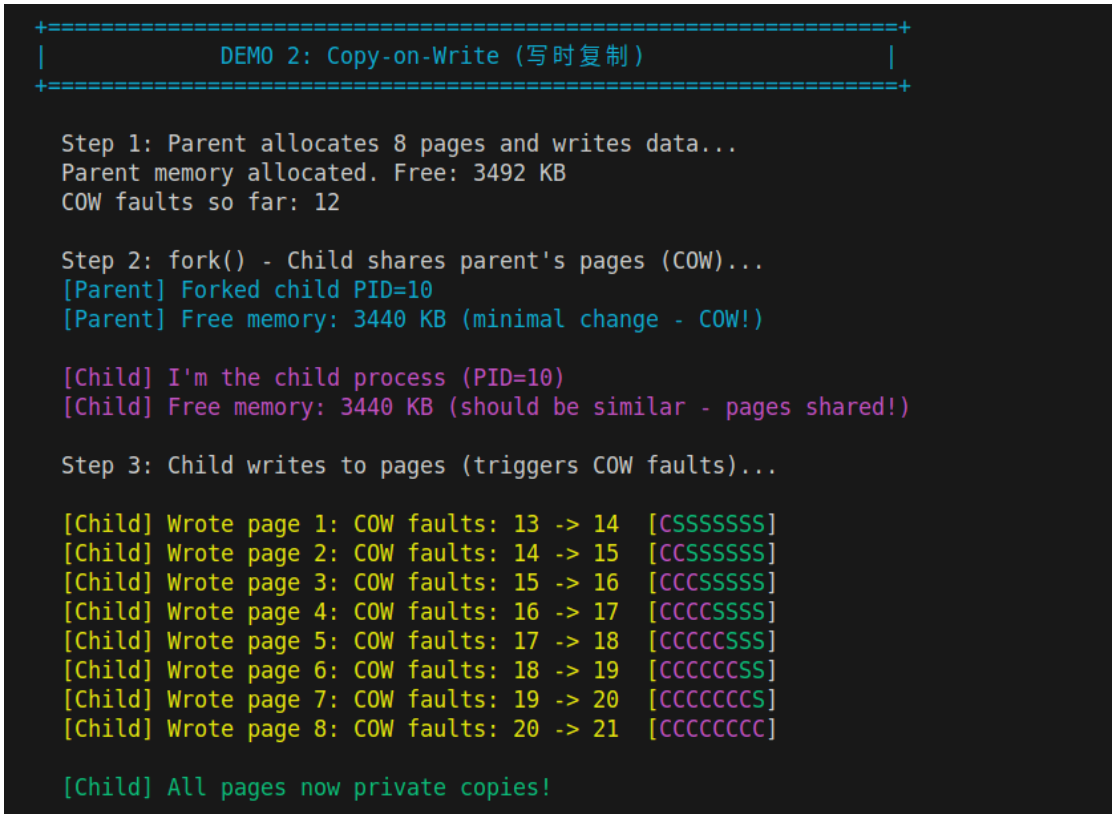

图 3.5-7 CoW 终端可视化演示

（3）pollwatch - I/O 多路复用可视化

pollwatch 用于展示 poll 系统调用同时监控多个文件描述符的能力。程序创建 4 个管道，每个管道有独立的 writer 进程按随机间隔写入数据。主进程使用 poll 同时监控所有管道的读端，实时显示哪些管道有数据可读、哪些处于等待状态、哪些已关闭。

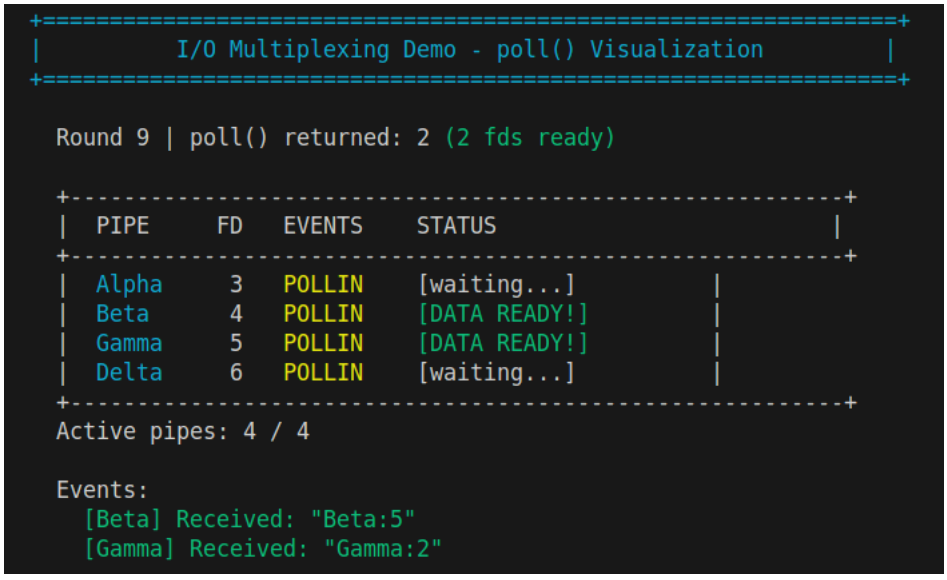

图 3.5-8 多路复用终端可视化演示

（4）ipcband - IPC 综合演示

ipcband 用于展示多种 IPC 机制协同工作。程序使用共享内存实现一个 8 槽的环形缓冲区，通过信号量实现生产者-消费者同步，并使用管道传递状态报告。

演示运行时，可以观察到生产者和消费者交替操作缓冲区，环形队列的 head 和 tail 指针不断移动，信号量控制着双方的节奏。这个演示很好地展示了共享内存与信号量配合使用的典型模式。

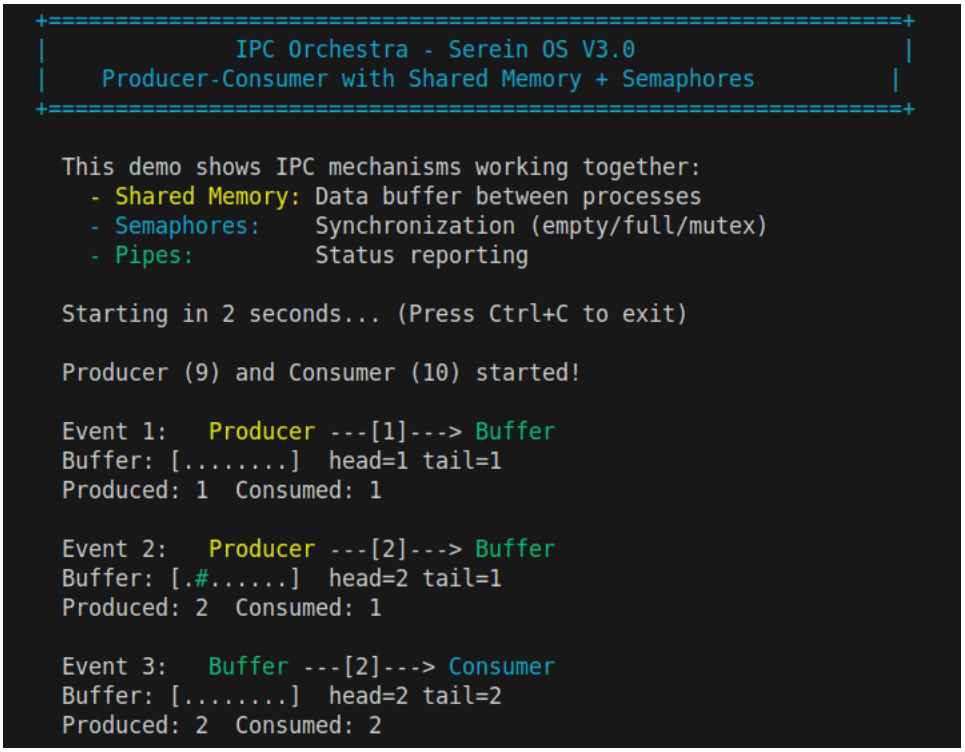

图 3.5-9 IPC 综合演示

以上演示程序均支持 Ctrl+C 中断，并会自动清理创建的子进程和 IPC 资源。在开发过程中，我们发现 sys_sleep 的循环只检查 killed 标志而不检查 sig_pending，导致 Ctrl+C 信号无法被及时处理。通过在 sleep 循环中增加对 sig_pending 的检查，解决了这个问题。

3.5.6 小结

我们在“用户交互”上的工作，系统性地优化了内核与用户的交互机制，在功能增强与性能优化两方面取得了显著进展。在功能层面，引入了分级日志系统，通过编译时开关实现零开销调试，显著提升了开发与调试效率。同时，新增了丰富的用户态命令集（cp、head、tail、timer、stat、cls），极大增强了系统的实用性和操作便捷性。在性能层面，核心改进在于重构了 UART 中断处理。通过采用“两阶段中断处理”模型，将繁重的 consoleintr()处理延迟到 trap 返回路径，显著降低了中断延迟和系统抖动，并减轻了锁争用，大幅提升了系统在高负载下的实时响应能力。此外，新增的输入溢出监控机制，通过精确统计并暴露缓冲区丢弃字符

数量，增强了对系统 I/O 行为的可观测性，为系统调试与性能分析提供了关键数据支持。

总的来说，本小节的优化使系统在保持轻量级的同时，交互体验、开发效率和实时性能均得到质的提升，为构建更健壮、可用的操作系统奠定了基础。

4. 系统调用的设计实现

4.1 系统调用的流程

4.1.1 调用发起与寄存器约定

RISC-V 架构使用 ecall 指令触发从用户态到内核态的模式切换。用户程序在发起系统调用前，需要按照 RISC-V 调用约定设置寄存器。a7 寄存器存放系统调用号，a0 至 a5 依次存放最多 6 个参数。所有系统调用号定义在 kernel/include/sysnum.h 中，从 1 开始编号。

以 fork 为例，用户态库函数的汇编存根由 usys.pl 脚本生成。执行 ecall 后，处理器检测到 Environment Call From U-mode 异常，将 scause 寄存器置为 8，同时保存当前 pc 到 sepc 寄存器，然后跳转到 stvec 指向的陷入处理入口。

4.1.2 陷入处理与上下文保存

用户态陷入的入口是 trampoline.S 中的 uservec 代码段。这段汇编代码执行以下关键操作：

（1）交换 sscratch 和 a0，获取 trapframe 结构的地址。这个地址在进程切换时由内核预先设置到 sscratch 寄存器。

（2）将全部 31 个通用寄存器保存到 trapframe 结构中。trapframe 定义在 kernel/include/trap.h，包含 epc、ra、sp、gp、tp、t0-t6、s0-s11、a0-a7 等字段，确保用户态上下文可以完整恢复。

（3）从 trapframe 读取内核页表、内核栈指针、内核陷入处理函数地址和当前 hart ID，切换到内核执行环境。

（4）跳转到 trap.c 中的 usertrap 函数。

usertrap 函数首先检查陷入原因。对于系统调用，核心处理逻辑如下：

```
1.  void usertrap(void) {
2.      struct proc *p = myproc();
3.
```

```
    // 保存用户程序计数器
    p->trapframe->epc = r_sepc();

    if(r_scause() == 8){
      // 系统调用：跳过 ecall 指令
      p->trapframe->epc += 4;

      // 开启中断，允许系统调用被抢占
      intr_on();

      // 分发到具体处理函数
      syscall();
    }
    // ... 其他异常处理
}
```

epc 加 4 是为了让用户程序在返回后从 ecall 的下一条指令继续执行，否则会陷入无限循环的系统调用。

4.1.3 参数提取与调用分发

syscall.c 中的 syscall 函数负责分发。它从 trapframe 的 a7 字段读取调用号，查表调用对应的处理函数，并将返回值写入 a0：

```
void usertrap(void) {
    struct proc *p = myproc();
    // 保存用户程序计数器
    p->trapframe->epc = r_sepc();
    if(r_scause() == 8){
      // 系统调用：跳过 ecall 指令
      p->trapframe->epc += 4;
      // 开启中断，允许系统调用被抢占
      intr_on();
      // 分发到具体处理函数
      syscall();
    }
    // ... 其他异常处理
}
```

syscalls 是一个函数指针数组，每个系统调用号作为索引对应一个 sys_xxx 处理函数。这种设计使得添加新系统调用非常简洁：在 sysnum.h 定义编号，在对应的 sys*.c 实现函数，最后在 syscall.c 注册即可。

参数提取由三个辅助函数完成。argint 获取 32 位整数参数，argaddr 获取 64 位地址参数，argstr 获取用户空间字符串。它们的核心是 argraw 函数，根据参数序号从 trapframe 读取对应寄存器值：

```
1.  static uint64 argraw(int n) {
2.      struct proc *p = myproc();
3.      switch (n) {
4.          case 0: return p->trapframe->a0;
5.          case 1: return p->trapframe->a1;
6.          case 2: return p->trapframe->a2;
7.          case 3: return p->trapframe->a3;
8.          case 4: return p->trapframe->a4;
9.          case 5: return p->trapframe->a5;
10.     }
11.     panic("argraw");
12.     return -1;
13. }
```

对于字符串参数，argstr 调用 fetchstr 从用户空间逐字节复制，直到遇到空字符或达到最大长度限制，同时验证地址是否在进程合法范围内。

4.1.4 返回用户态

系统调用处理完成后，usertrap 调用 usertrapret 准备返回用户态。这个函数执行以下操作：

（1）关闭中断，防止返回过程中被打断。

（2）设置 stvec 指向用户态陷入入口 uservec，为下次陷入做准备。

（3）将内核页表、内核栈、usertrap 函数地址等信息写入 trapframe，供下次陷入时使用。

（4）设置 sstatus 寄存器的 SPP 位为 0 表示返回用户模式，SPIE 位为 1 使能用户态中断。

（5）将返回地址写入 sepc 寄存器。

（6）计算用户页表的 satp 值，跳转到 userret 汇编代码执行最后的寄存器恢复和模式切换。

userret 使用 csrw 指令切换页表，从 trapframe 恢复所有用户寄存器，最后执行 sret 指令返回用户态。此时 a0 寄存器已经包含了系统调用的返回值。

4.2 部分系统调用分类与实现

Serein-OS 实现了 75 个系统调用，按功能可分为内存管理、文件系统、信号处理、网络通信、进程控制等类别。下面详细介绍各类别中的关键实现。

4.2.1 内存管理

sbrk 堆空间调整

调用号 12，参数为字节调整量 n，返回调整前的堆顶地址。这是用户程序动态分配内存的基础接口。

实现采用 Lazy Allocation 策略。sbrk 仅修改 p->sz 的值，完全不涉及物理内存分配。当进程实际访问新地址时，触发缺页异常，由 usertrap 中的 lazy_alloc 函数分配物理页并建立映射。这种设计使得 sbrk 调用本身变得非常轻量级，只需要修改一个整数字段：

```
1.  uint64 sys_sbrk(void) {
2.      int n;
3.      argint(0, &n);
4.      int addr = myproc()->sz;
5.      if(growproc(n) < 0)
6.          return -1;
7.      return addr;
8.  }
9.  // kernel/proc.c
10. int growproc(int n) {
11.     struct proc *p = myproc();
12.     if(n > 0){
13.         // 仅更新大小，不分配物理页
14.         p->sz = p->sz + n;
15.     }
16.     // ...
17.     return 0;
18. }
```

lazy_alloc 在 trap 处理中检测到合法的堆地址缺页时调用，分配一个物理页，清零后映射到用户页表和内核页表。这个优化减少了不必要的内存占用，尤其对 fork 后立即 exec 的场景效果显著。

mmap 内存映射

调用号 40，是虚拟内存管理中最复杂的系统调用之一。参数包括地址提示、映射长度、保护标志、映射标志、文件描述符和偏移量。

mmap 的地址空间与 sbrk 管理的堆完全分离，使用固定基址 MMAPBASE 开始分配，避免了地址冲突问题。这是在调试过程中总结出的经验，早期版本将 mmap 放在 p->sz 之上导致了与堆增长的冲突。

每个进程维护一个包含 16 个槽位的 VMA 数组。sys_mmap 首先查找空闲 VMA 槽，然后计算可用的虚拟地址，最后初始化 VMA 结构但不立即分配物理页：

```
1.  uint64 sys_mmap(void) {
2.    // 找到空闲 VMA 槽
3.    struct vma *vma = 0;
4.    for(int i = 0; i < MAX_VMA; i++) {
5.      if(!p->vmas[i].valid) {
6.        vma = &p->vmas[i];
7.        break;
8.      }
9.    }
10.   if(vma == 0)
11.     return -1;
12.
13.   // 计算虚拟地址：从 MMAPBASE 开始，避开已有映射
14.   uint64 va = MMAPBASE;
15.   for(int i = 0; i < MAX_VMA; i++) {
16.     if(p->vmas[i].valid && p->vmas[i].addr + p->vmas[i].len > va) {
17.       va = p->vmas[i].addr + p->vmas[i].len;
18.     }
19.   }
20.   va = PGROUNDUP(va);
21.
22.   // 初始化 VMA，不分配物理页
23.   vma->addr = va;
24.   vma->len = len;
25.   vma->prot = prot;
26.   vma->flags = flags;
27.   vma->f = f;
28.   vma->valid = 1;
29.
30.   if(f) filedup(f);   // 增加文件引用计数
```

```
31.
32.     return va;
33. }
```

实际的页面分配发生在缺页时，由 mmap_handle_fault 函数处理。对于文件映射，从文件读取数据填充新页；对于匿名映射则直接分配清零页。munmap 时如果是 MAP_SHARED 映射需要将脏页写回文件。

mincore 页面状态查询

调用号 57，用于查询指定地址范围的虚拟页是否驻留在物理内存。这个系统调用主要用于内存监控工具 memviz 展示进程的内存使用情况。

实现遍历地址范围内的每一页，通过 walk 函数查询页表项，检查 PTE_V 有效位是否置位。由于 xv6 没有实现 RISC-V 的 Accessed 位，这里用页表项是否有效作为驻留判断依据：

```
1.  uint64 sys_mincore(void) {
2.      uint64 npages = (length + PGSIZE - 1) / PGSIZE;
3.
4.      for(a = addr, i = 0; i < npages; a += PGSIZE, i++) {
5.          pte_t *pte = walk(p->pagetable, a, 0);
6.          if(pte && (*pte & PTE_V)) {
7.              // 页面有效，标记为驻留
8.              vec_buf[i/8] |= (1 << (i % 8));
9.          }
10.     }
11.
12.     copyout2(vec, vec_buf, vec_size);
13.     return 0;
14. }
```

结果以位图形式返回，每字节的最低位对应第一个页面，依次类推。

4.2.2 文件系统操作

open 文件打开

调用号 15，是文件操作的核心入口。参数包括路径、打开标志和权限模式。这个系统调用需要协调文件系统查找、权限检查、文件对象分配等多个步骤。

根据标志执行不同逻辑。O_CREATE 标志触发文件创建，调用 create 函数在文件系统中创建新条目。否则调用 ename 查找现有文件。找到文件后，分配 struct file 结构并加入进程的 ofile 数组：

```
1.  uint64 sys_open(void) {
2.      struct dirent *ep;
```

```
3.      struct file *f;
4.      int fd;
5.
6.      if(omode & O_CREATE){
7.          ep = create(path, T_FILE, mode);
8.      } else {
9.          ep = ename(path);
10.     }
11.
12.     // 分配文件对象和描述符
13.     f = filealloc();
14.     fd = fdalloc(f);
15.
16.     // 设置文件属性
17.     f->type = FD_ENTRY;
18.     f->ep = ep;
19.     f->readable = !(omode & O_WRONLY);
20.     f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
21.
22.     return fd;
23. }
```

O_TRUNC 标志触发文件截断，清空现有内容。O_APPEND 标志将写入位置设置到文件末尾。O_NONBLOCK 标志设置非阻塞模式，影响后续的 read 和 write 行为。

lseek 文件定位

调用号 45，修改文件的当前读写偏移。参数包括文件描述符、偏移量和基准位置。

whence 参数决定偏移计算方式：SEEK_SET 从文件头计算，SEEK_CUR 从当前位置计算，SEEK_END 从文件尾计算。实现直接修改 f->off 字段，并验证结果的合法性：

```
1.  uint64 sys_lseek(void) {
2.      struct file *f;
3.      int offset, whence;
4.
5.      if(argfd(0, 0, &f) < 0 || argint(1, &offset) < 0 || argint(2, &whence) < 0)
6.          return -1;
7.
8.      int newoff;
```

```
    switch(whence) {
        case SEEK_SET:
            newoff = offset;
            break;
        case SEEK_CUR:
            newoff = f->off + offset;
            break;
        case SEEK_END:
            newoff = f->ep->file_size + offset;
            break;
        default:
            return -1;
    }

    if(newoff < 0)
        return -1;

    f->off = newoff;
    return newoff;
}
```

rename 文件重命名

调用号 26，支持文件和目录的重命名及跨目录移动。这是一个需要操作多个目录项的复杂系统调用。

实现时需要同时锁定源目录和目标目录的 dirent 结构。为防止死锁，代码采用基于地址的锁排序策略，总是先锁定地址较小的 dirent。这个方案是在调试过程中发现偶发死锁后总结出来的：

```
uint64 sys_rename(void) {
    // 地址比较，确定加锁顺序
    if((uint64)src_dp < (uint64)dst_dp) {
        elock(src_dp);
        elock(dst_dp);
    } else {
        elock(dst_dp);
        elock(src_dp);
    }

    // 执行重命名操作
    // ...
```

```
13.
14.     // 按相反顺序解锁
15.  }
```

fcntl 描述符控制

调用号 60，用于获取和修改文件描述符的属性。支持的命令包括 F_GETFL 获取状态标志、F_SETFL 设置状态标志、F_DUPFD 复制描述符到指定起始位置。

F_SETFL 常用于设置 O_NONBLOCK 标志。非阻塞模式下，read 和 write 在无数据或缓冲区满时立即返回 EAGAIN 而非阻塞。这个功能对于实现事件驱动的程序至关重要：

```
1.    uint64 sys_fcntl(void) {
2.        int fd, cmd, arg;
3.        struct file *f;
4.
5.        // ...获取参数...
6.
7.        switch(cmd) {
8.            case F_GETFL:
9.                return f->flags;
10.
11.           case F_SETFL:
12.               f->flags = (f->flags & ~O_NONBLOCK) | (arg & O_NONBLOCK);
13.               return 0;
14.
15.           case F_DUPFD:
16.               // 复制到 >= arg 的最小空闲 fd
17.               for(int i = arg; i < NOFILE; i++) {
18.                   if(p->ofile[i] == 0) {
19.                       p->ofile[i] = f;
20.                       filedup(f);
21.                       return i;
22.                   }
23.               }
24.               return -1;
25.       }
26.   }
```

4.2.3 信号系统

信号系统是 V2.0 版本的核心新增功能，实现了完整的 POSIX 信号机制，支持 32 种信号。

sigaction 信号处理器注册

调用号 43，设置进程对特定信号的处理方式。参数为信号编号和处理函数地址。

handler 参数可取三种值：SIG_DFL 表示默认处理，SIG_IGN 表示忽略信号，其他值表示用户自定义处理函数的地址。SIGKILL 和 SIGSTOP 是特殊信号，不允许被捕获或忽略，尝试设置会返回错误：

```
1.  uint64 sys_sigaction(void) {
2.      int sig;
3.      uint64 handler;
4.      struct proc *p = myproc();
5.
6.      if(argint(0, &sig) < 0 || argaddr(1, &handler) < 0)
7.          return -1;
8.
9.      // 验证信号范围
10.     if(sig < 1 || sig >= NSIG)
11.         return -1;
12.
13.     // SIGKILL 和 SIGSTOP 不可捕获
14.     if(sig == SIGKILL || sig == SIGSTOP)
15.         return -1;
16.
17.     p->sig_handlers[sig] = (void (*)(int))handler;
18.     return 0;
19. }
```

处理器地址存储在进程的 sig_handlers 数组中，最多支持 32 个信号槽位。

kill2 信号发送

调用号 42，向指定进程发送指定信号。与传统只发送 SIGKILL 的 kill 不同，kill2 支持发送任意信号：

```
1.  uint64 sys_kill2(void) {
2.      int pid, sig;
3.
4.      if(argint(0, &pid) < 0 || argint(1, &sig) < 0)
5.          return -1;
6.
7.      return send_signal(pid, sig);
```

```
8.  }
```

send_signal 函数查找目标进程，设置其 sig_pending 位图中对应信号的位。实际的信号处理发生在目标进程从内核返回用户态之前，由 check_signals 函数检查 pending 位图并调用相应处理函数。

sigreturn 信号帧恢复

调用号 44，在信号处理函数执行完毕后调用，用于恢复被中断的执行上下文。

内核在派发信号时会将原始的 trapframe 保存到 p->sig_saved_tf，并构造一个新的用户栈帧用于执行信号处理器。处理器返回时通过 sigreturn 恢复原始上下文：

```
1.  uint64 sys_sigreturn(void) {
2.      struct proc *p = myproc();
3.
4.      // 恢复保存的 trapframe
5.      memmove(p->trapframe, &p->sig_saved_tf, sizeof(struct trapframe));
6.
7.      // 清除正在处理的信号
8.      p->sig_handling = 0;
9.
10.     return p->trapframe->a0;   // 返回原始系统调用的返回值
11. }
```

信号帧的构建和恢复是信号系统中最复杂的部分，需要精确处理栈对齐、寄存器保存和返回地址设置，调试过程中曾多次因栈帧错误导致用户程序崩溃。

4.2.4 网络通信

V3.1 版本引入了完整的 BSD Socket API，将"一切皆文件"的理念扩展到网络通信领域。

socket 套接字创建

调用号 68，创建通信端点。参数为地址族、套接字类型和协议。当前支持 AF_UNIX 和 AF_INET 两种地址族，SOCK_STREAM 和 SOCK_DGRAM 两种类型。

实现调用 sockalloc 从全局套接字池分配 struct socket 结构，初始化 domain 和 type 字段，然后通过 sockfdalloc 包装为文件描述符。套接字文件的 f->type 设置为 FD_SOCKET，使得后续可以通过标准 read/write/close 操作：

```
1.  uint64 sys_socket(void) {
2.      int domain, type, protocol;
3.      struct socket *so;
4.
```

```
    // 验证参数
    if(domain != AF_UNIX && domain != AF_INET)
        return -1;
    if(type != SOCK_STREAM && type != SOCK_DGRAM)
        return -1;

    so = sockalloc();
    if(so == 0)
        return -1;

    so->domain = domain;
    so->type = type;

    int fd = sockfdalloc(so, 1, 1);  // 可读可写
    if(fd < 0){
        sockfree(so);
        return -1;
    }

    return fd;
}
```

bind/listen/accept 服务端流程

bind 将套接字绑定到指定地址。对于 AF_UNIX，地址是文件系统路径字符串；对于 AF_INET，是 IP 地址和端口的组合结构。

listen 将套接字状态从 SS_UNCONNECTED 变为 SS_LISTENING，准备接受连接。backlog 参数指定等待队列长度。

accept 是一个阻塞调用，等待客户端连接请求。连接建立后分配新的套接字和文件描述符用于数据通信，原套接字继续监听。在调试过程中发现 accept 的地址缓冲区大小必须足够容纳 sockaddr_un 结构，否则会导致栈溢出：

```
uint64 sys_accept(void) {
    // 使用足够大的缓冲区，避免栈溢出
    char addr[128];  // sockaddr_un 需要 110 字节
    int addrlen = sizeof(addr);

    struct socket *newso = sockaccept(so, (struct sockaddr*)addr, &addrlen);

    int fd = sockfdalloc(newso, 1, 1);
    return fd;
```

```
10. }
```

send/recv 数据传输

send 将用户缓冲区数据写入套接字发送队列。实现分块复制，每次最多 256 字节，循环直到全部数据发送完毕或遇到错误。

recv 从接收缓冲区读取数据。缓冲区采用环形结构，head 和 tail 指针标记有效数据范围。若缓冲区为空且为阻塞模式则睡眠等待数据到达。

数据传输模型类似管道：发送端写入对端的 recvbuf，接收端读取自己的 recvbuf。这种设计复用了管道的实现经验，简化了缓冲区管理。

netstat 状态查询

调用号 75，遍历全局套接字表收集状态信息。返回的 sock_stat 结构包含地址族、套接字类型、连接状态、本地和远端地址、接收队列使用量等信息。这个系统调用支持 sockviz 可视化监控工具实时展示网络连接状态。

4.2.5 进程控制与线程

clone 轻量级进程创建

调用号 63，类似 Linux 的 clone 系统调用，支持通过标志位控制父子进程共享的资源。这是用户态线程库的基础。

当前实现支持以下标志：

- CLONE_VM：共享虚拟内存空间
- CLONE_FILES：共享文件描述符表
- CLONE_THREAD：加入同一线程组

实现基于 fork 框架，根据标志决定是否深拷贝相关资源。CLONE_THREAD 标志使新进程与父进程共享 PID：

```
1.  uint64 sys_clone(void) {
2.      int flags;
3.      uint64 stack;
4.
5.      int pid = fork();
6.      if(pid == 0) {
7.          struct proc *np = myproc();
8.
9.          // 设置新栈
10.         if(stack != 0) {
11.             np->trapframe->sp = stack;
12.         }
13.
14.         // 标记为线程
```

```
15.         if(flags & CLONE_THREAD) {
16.             np->is_thread = 1;
17.             np->thread_group = p;
18.         }
19.     }
20.
21.     return pid;
22. }
```

完整的 CLONE_VM 实现需要修改 fork 核心逻辑让父子共享页表，当前版本是简化实现。

futex 快速用户空间互斥锁

调用号 64，提供用户态同步原语的内核支持。futex 是 Fast Userspace muTEX 的缩写，大多数情况下在用户态通过原子操作完成锁的获取和释放，只有发生竞争时才需要进入内核。

支持两种操作：FUTEX_WAIT 检查指定地址的值是否等于预期值，若相等则睡眠等待；FUTEX_WAKE 唤醒等待在指定地址上的进程：

```
1.  uint64 sys_futex(void) {
2.      uint64 addr;
3.      int op, val;
4.
5.      switch(op) {
6.          case FUTEX_WAIT: {
7.              int current;
8.              copyin(p->pagetable, (char*)&current, addr, sizeof(int));
9.
10.             if(current != val)
11.                 return -1;   // 值已改变，不需要等待
12.
13.             sleep((void*)addr, 0);
14.             return 0;
15.         }
16.
17.         case FUTEX_WAKE: {
18.             wakeup((void*)addr);
19.             return 0;
20.         }
21.     }
22. }
```

这个接口配合用户态的原子比较交换指令，可以实现高效的互斥锁和条件变量。

exit_group 线程组退出

调用号 65，终止整个线程组内的所有进程。遍历进程表，将同一线程组的所有成员标记为 killed，然后调用 exit 终止当前进程。这是多线程程序正确退出的基础。

4.2.6 系统管理

sysinfo 系统信息

调用号 19，返回系统运行状态。填充 struct sysinfo 结构，包含空闲内存页数、运行进程数、系统运行时间等信息。vmstat 工具使用这个调用展示系统整体资源使用情况。

halt 和 reboot 电源管理

调用号 66 和 67。halt 执行关机操作，reboot 重启系统。实现通过 SBI 调用请求 M-mode 固件执行相应操作。这两个调用只有 root 用户可以执行。

trace 系统调用追踪

调用号 18，设置进程的追踪掩码 p->tmask。掩码的每一位对应一个系统调用号，置位表示追踪该调用。在 syscall 函数返回前，若对应位被设置则打印调用名称、参数和返回值。这个功能用于调试和性能分析，帮助定位系统调用层面的问题。

4.3 添加新系统调用

在 Serein 中添加自定义系统调用需要修改以下文件：

（1）kernel/include/sysnum.h：定义新的系统调用号

（2）kernel/sys*.c：实现 sys_xxx 处理函数

（3）kernel/syscall.c：添加函数声明和注册

（4）serein-user/usys.pl：添加用户态存根生成

（5）serein-user/user.h：添加用户态函数声明

以获取进程票数为例，完整的添加步骤可参考 docs/用户使用-系统调用扩展.md 文档。

5. 总结和展望

5.1 工作总结

从最初基于华科 xv6-k210 开始动手，到现在 V3.1 版本的完成，Serein 经历了五个版本的迭代。这个过程中，我们把一个基础的教学操作系统逐步改造成了一个功能相对完善的嵌入式内核。

5.1.1 版本演进

回顾整个开发过程，每个版本都有明确的目标：

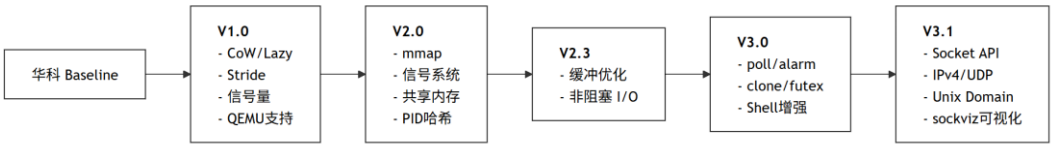

图 5.1-1 版本演进图

V1.0 主要解决内存效率问题，CoW 和 Lazy Allocation 让系统不再像原版那样"大手大脚"地使用内存。同时引入 Stride 调度和信号量，让进程管理更加灵活。这个版本还完成了从 K210 到 QEMU 的移植，方便后续开发调试。

V2.0 的重点是进程间通信和信号系统。mmap 让程序可以把文件直接映射到内存，共享内存让多个进程能够高效交换数据，信号系统则提供了异步通知机制。这些组合起来，Serein 开始有了"现代操作系统"的样子。

V2.3 相对小一些，主要是 I/O 层面的优化。缓冲区分桶减少了多核争用，非阻塞 I/O 让程序处理多个数据源变得更容易。

V3.0 面向用户体验，poll 和 alarm 让应用程序能够高效地等待多个事件，clone/futex 为线程支持打下基础，Shell 的命令历史和 Tab 补全则让日常使用方便了很多。

V3.1 是网络功能的突破。虽然没有真正的网络硬件驱动，但 Socket API 的完整实现让 Serein 具备了网络编程的基础框架。

各版本的代码增量大致如下：

表 5.1-1 内核版本演进与主要功能扩展

版本	主要新增功能	新增代码量（约）
V1.0	CoW、Lazy Allocation、Stride 调度器、信号量、QEMU 移植	3,000 行
V2.0	mmap、信号系统、共享内存、PID 哈希	4,000 行

V2.3	缓冲区分桶、非阻塞 I/O、FAT32 优化	1,000 行
V3.0	poll、alarm、fcntl、clone / futex、Shell 增强	1,500 行
V3.1	Socket API、Unix Domain Socket、IPv4 Loopback、UDP	2,500 行

累计下来，我们新增了约 10,000 行内核代码，实现了 75 个系统调用。

5.1.2 各子系统的实现

（1）内存管理

CoW 的效果非常直观：fork 一个占用 10MB 内存的进程，以前需要 80ms，现在只要 5ms。这个 16 倍的提升来自于一个简单的思路——不复制不需要复制的东西。父子进程共享同一份物理页面，只有当其中一方真正要写入时才分配新页面。

Lazy Allocation 的想法类似：sbrk 申请的内存不立即分配，等程序真正访问时再给。很多程序申请了内存但不一定全部用到，这种延迟策略省下了不少物理内存。

mmap 则是另一种内存使用方式。把文件映射到地址空间后，读写文件就像读写内存一样简单，内核负责在后台处理数据的加载和写回。

（2）进程调度

Stride 调度解决的问题是"如何让高优先级进程获得更多 CPU 时间"。每个进程有一个票数，票数越高，每次运行后累加的 pass 值越小，下次被选中的机会就越大。这比彩票调度的随机性更可预测，测试时更容易验证正确性。

实现时遇到一个麻烦：调度器遍历进程表时如果持有锁，很容易和其他 CPU 上的进程形成死锁。最后采用的方案是两阶段扫描，先无锁找候选，再加锁验证，避开了死锁。

（3）进程间通信

信号量是最基础的同步原语，sem_wait 等待、sem_post 释放，配合共享内存就能实现生产者消费者模型。

共享内存的设计有个细节值得一提：我们采用固定地址窗口，每个 shmid 对应一个确定的虚拟地址。这样做限制了同时可用的共享内存段数量，但好处是不用处理复杂的地址分配逻辑，多进程映射同一段共享内存时地址也能保持一致。

（4）信号系统

信号系统的核心难点在于上下文切换。当信号到达时，内核需要在用户栈上构建一个信号帧，保存当前寄存器状态，然后跳转到用户注册的处理函数。处理函数执行完毕后通过 sigreturn 恢复原来的上下文继续执行。

这个过程对栈对齐要求很严格，RISC-V 需要 16 字节对齐。早期版本在这个地方踩了不少坑，信号处理函数返回后程序就崩溃了，后来仔细检查才发现是对齐问题。

（5）网络子系统

Socket 的实现复用了管道的环形缓冲区机制。两个相连的 socket 共享一块缓冲区，send 写入对方的接收缓冲区，recv 从自己的接收缓冲区读取。这种模型简单有效，对于 Loopback 和 Unix Domain 场景足够用了。

调试过程中遇到一个隐蔽的 bug：accept 函数的临时缓冲区只有 16 字节，但 sockaddr_un 结构需要 110 字节，结果栈上的返回地址被覆盖，函数一返回就崩溃。这种问题在用户态很常见，但在内核里遇到时还是花了不少时间才定位到。

（6）Shell 增强

命令历史和 Tab 补全看起来是小功能，但实现起来比预想的复杂。Tab 补全后光标位置和内部缓冲区的同步需要仔细处理，否则退格删除时会出现位置不一致的问题。

5.1.3 性能数据

几个关键优化的效果：

表 5.1-2 核心机制性能优化对比

优化项	优化前	优化后	性能提升
fork 创建 10MB 进程	80 ms	5 ms	**16×**
kill /wait 进程查找	O(N) 线性遍历	O(1) 哈希查找	**50×（N = 50）**
FAT32 大文件定位	O(n) 簇链遍历	O(n / 128) 索引定位	**128×**

5.1.4 测试情况

usertests 套件有 70 多个用例，覆盖了基本的进程、内存、文件操作。在此基础上，我们为新增功能编写了 20 多个专项测试：

表 5.1-3 用户态测试程序分类与覆盖

类别	对应测试程序
内存管理	cowstress, mmaptest, mincore_test
信号系统	sigtest, sigtest2, alarmtest
信号量	semtest, semtest2, semstress
共享内存	shmtest1, shmtest2, shmstress
线程 / 并发	threadtest, polltest, nbtest
文件系统	fcntltest, chmodtest, testdev

目前所有测试在 QEMU 双核环境下稳定通过。

5.2 经验总结

5.2.1 踩过的坑

有几个印象深刻的问题：

（1）引用计数竞态

CoW 实现初期，多核同时 fork 会导致引用计数错误。两个 CPU 同时读取计数值，各自加一后写回，结果只增加了一而不是二。

解决方法很直接：加锁。虽然会有一点性能开销，但正确性是第一位的。

```
1.  void kaddref(uint64 pa) {
2.  acquire(&refcnt_lock);
3.  refcnt[PA2IDX(pa)]++;
4.  release(&refcnt_lock);
5.  }
```

（2）调度器死锁

调度器需要遍历所有进程找下一个运行的，如果遍历时持有锁，刚好被遍历的进程在另一个 CPU 上也想获取锁，就会死锁。

最后采用两阶段方案：先不加锁扫描一遍找候选，再加锁验证候选是否还符合条件。如果不符合就重新扫描。这样虽然可能多扫几次，但避免了死锁。

（3）信号帧对齐

sigreturn 恢复上下文时，从信号帧读取保存的寄存器值。如果信号帧的位置没有对齐，读取的位置就错了，恢复出来的寄存器值也是错的。

这个问题的表现非常诡异，程序有时候能正常返回，有时候会崩溃，取决于原来栈指针的位置。后来在构建信号帧时强制对齐到 16 字节才解决。

（4）Socket 缓冲区溢出

accept 系统调用中，用 struct sockaddr 作为临时缓冲接收客户端地址。这个结构只有 16 字节，但 Unix Domain Socket 的地址结构有 110 字节。结果超出的部分覆盖了栈上的其他变量，包括返回地址。

找这个 bug 花了不少时间。现象是 accept 在内核里执行得好好的，一返回用户态就崩溃。最后靠逐步简化测试用例，用 simplesock 单进程测试才定位到问题。

5.2.2 设计上的取舍

几个需要做决定的地方：

（1）共享内存地址分配

共享内存有个麻烦的问题：进程 A 把共享段映射到地址 0x40000000，进程 B 映射到 0x50000000，那它们在共享段里存的指针就全乱了。

理想的做法是像 mmap 那样动态分配地址，每个进程自己找空闲区域。但这样一来，多进程访问同一段共享内存时地址不一致，还得额外处理指针重定位的问题，复杂度一下子上去了。

我们最后选了一个"偷懒"的办法：固定地址窗口。每个 shmid 对应一个预定义的虚拟地址范围，所有进程映射同一段共享内存时拿到的地址都一样。代价是同时可用的共享内存段数量有限，但对于我们这样的教学演示场景，8 个共享段绰绰有余了。

有时候简单粗暴的方案反而最实用。

（2）Socket 缓冲区模型

真正的网络栈里，每个 socket 有独立的发送缓冲区和接收缓冲区，还要处理 TCP 的滑动窗口、拥塞控制、超时重传等一大堆复杂逻辑。如果我们也这么做，光网络子系统的代码量就能超过目前整个项目。

我们退了一步，Serein 的网络功能主要是验证 Socket API 能不能正常工作，又不是真的要跑在互联网上。Loopback 和 Unix Domain 都是本机内部通信，数据在内核里转一圈就回来了，根本不会丢包，要什么重传？

于是就复用了管道那套环形缓冲区的机制。两个连接的 socket 共享一块 4KB 的缓冲区，send 往里写，recv 往外读，写满了就阻塞等着，简单直接。

这个模型跑 socktest、inettest、udptest 都没问题，sockviz 监控工具也能正常显示队列状态。当然，这套东西拿去跑真正的网络应用肯定不行，但那本来也不是我们的目标。

（3）调度算法的选择

这个决策背后有一段曲折的经历。最开始我们的目标是实现 MLFQ 多级反馈队列调度，这是一个相当复杂但功能强大的算法，能够同时兼顾交互式进程的响应速度和批处理进程的吞吐量。

MLFQ 的实现过程中遇到了严重的竞态问题。在多核环境下，进程在不同优先级队列之间迁移时，如果两个 CPU 同时操作同一个进程的队列状态，就会出现各种诡异的问题。更麻烦的是 usertests 中的 reparent 测试始终无法通过。这个测试涉及父进程退出后子进程被 init 收养的场景，在 MLFQ 的队列操作和进程状态变更之间存在微妙的时序依赖，稍有不慎就会导致进程丢失或者队列损坏。

我们花了相当长的时间试图修复这些问题，尝试了不同的加锁策略，但始终无法在保证正确性的同时让所有测试通过。最后考虑到系统稳定性是第一位的，决定退而求其次，采用 Stride 步长调度。

Stride 调度虽然没有 MLFQ 那样的自适应能力，但它有两个明显的优点：

1.实现相对简单，只需要维护每个进程的 tickets 和 pass 值

2.调度结果是确定性的，同样票数的进程最终获得的 CPU 时间几乎相等，非常方便测试验证

相比之下，彩票调度虽然实现更简单，但结果具有随机性，不太适合需要精确验证调度公平性的场景。

我们设计 Serein 的思想是："能跑通所有测试"比"功能看起来更强大"更重要。如果有后来者对调度算法感兴趣，可以尝试在我们的代码基础上实现 MLFQ，解决掉我们未能解决的 reparent 竞态问题。这会是一个很好的挑战。测试时这个确定性很有用，可以直接验证"票数比例是否等于运行时间比例。

5.2.3 开发过程

（1）每个版本做完后跑一遍完整的 usertests，确认没有破坏已有功能。这个习惯在后期省了不少事，有几次改动意外影响了其他模块，多亏回归测试及时发现。

（2）复杂功能出问题时，写一个最简化的测试用例比在完整程序里调试效率高得多。Socket 的 accept bug 就是通过 simplesock 单进程测试定位的。

（3）分级日志在调试阶段很有用。出问题时把 LOG_DEBUG 打开，能看到关键路径的执行情况。发布时调成 LOG_WARN 就不会有额外开销。

5.3 未来计划

项目做到现在，功能上已经比较完整了，但还有不少可以继续探索的方向。

5.3.1 真实网络支持

目前的网络功能其实只是"假装有网络"。Loopback 是自己跟自己通信，Unix Domain 是同一台机器上的进程互相发消息，都不需要真正把数据发到网线上去。

如果想让 Serein 真正具备联网能力，K210 可以通过 SPI 接口外接 ESP8266 这类 Wi-Fi 模块。具体要做的事情包括：

（1）写一个 SPI 接口的 Wi-Fi 模块驱动，跑通 AT 指令集，能够连接 Wi-Fi 热点

（2）集成 lwIP 这类轻量级协议栈，支持真正的 TCP/IP 通信

（3）实现 DHCP 客户端自动获取 IP 地址，实现 DNS 解析域名

做完这些，Serein 就能作为物联网边缘节点接入网络，跑一些简单的数据采集和上报应用。这会是一个相当有意思的扩展方向。

5.3.2 用户态线程库

clone 和 futex 系统调用都已经实现了，内核层面支持线程的基础设施有了。但目前用户想创建线程还得直接调用这些底层接口，不太方便。

下一步可以在用户态封装一个 pthread 兼容库：pthread_create 和 pthread_join 用 clone 实现，pthread_mutex 用 futex 实现，pthread_cond 条件变量在 mutex 基础上再封装一层。

有了这个库，现有的多线程 C 程序移植到 Serein 上就方便多了，基本改改头文件就能跑。

5.3.3 文件系统改进

FAT32 目前只支持 8.3 短文件名，文件名长一点就会被截断，挺别扭的。可以考虑实现 VFAT 长文件名扩展，让用户可以用正常的文件命名。

另外一个痛点是目录遍历性能。现在 ls 一个大目录时每次都要读 SD 卡，明显能感觉到卡顿。如果加一层目录项缓存，把最近访问过的目录内容留在内存里，性能会好很多。

5.3.4 性能分析工具

内核调试目前主要靠 printf 大法，哪里有问题就往哪里加打印。这种方式简单直接，但效率不高，而且打印太多会影响系统行为，打印太少又看不出问题。

如果能有一套像样的性能分析工具就好了：

（1）系统调用追踪，记录每次调用的类型、参数和耗时

（2）中断延迟统计，监测从中断触发到处理程序执行之间的时间分布

（3）内存分配追踪，记录每次 kalloc 和 kfree 的调用栈，方便发现内存泄漏

这些工具对于把 Serein 用到实际嵌入式产品开发会很有帮助。

5.3.5 更多硬件支持

现在开发测试主要在 QEMU 虚拟环境下进行，虽然方便调试，但毕竟不是真实硬件。后续可以像原版 xv6-k210 项目那样，在 K210 真实开发板上做更多验证，确保代码在真机上也能稳定运行。

另外 RISC-V 生态这几年发展很快，出了不少新的开发板。比如 StarFive VisionFive 系列性能比 K210 强不少，也许可以尝试移植过去，看看 Serein 在更强的硬件上能跑出什么效果。

5.3.6 小结

Serein 一词源自法语，意为“傍晚的晴空雨”，感谢朋友的同名电影所带来的启发。

回顾整个项目，从最初拿到华科 xv6-k210 的代码，到现在 V3.1 版本的完成，Serein 已经从一个基础的教学系统成长为具备 Copy-on-Write、Stride 调度、信号系统、共享内存、Socket 网络等现代操作系统核心功能的嵌入式内核。代码规模从基线的几千行增长到约 16,000 行内核代码，系统调用数量达到 75 个。

这个过程中踩过的坑、做过的取舍、积累的调试经验，可能比最终的代码本身更有价值。多核竞态怎么处理、栈对齐为什么重要、什么时候该坚持理想方案什么时候该务实妥协，这些都不是看书能学会的，只有自己动手做一遍才能真正理解。

希望 Serein 的代码和文档能对后来的学习者有所帮助。如果你在这个基础上做出了什么有意思的东西，或者解决了我们没能搞定的 MLFQ 问题，欢迎交流。