文献阅读专题：系统调用加速

系统调用是用户获取内核服务的一种途径。一般的系统调用过程为：

- 由用户程序使用一条指令触发同步异常
- 陷入内核，根据系统调用号进行分发
- 内核运行具体的处理函数
- 从内核返回

系统调用的开销主要包括两个部分：

- 直接开销：包括用户与内核之间的上下文切换，以及系统调用分发的代码
- 间接开销：地址空间切换造成的缓存污染，包括 cache、TLB 等

为了减小系统调用的开销，有以下的思路：

- 将多个系统调用收集起来一同提交给内核，即批处理系统调用
  - Livio Soares 等人的文章 FlexSC: Flexible System Call Scheduling with Exception-Less System Calls 中，实现了一个内核与用户共享的虚拟页面，用户将所有系统调用请求统一写入该页面，内核收集后进行批处理，而后把处理结果写回该页面
  - Mohan Rajagopalan 等人的文章 Cassyopia: Compiler Assisted System Optimization 中提出了一种利用编译器优化将系统调用聚集起来的思路，在编译时减小应用程序中的系统调用次数
- 降低用户与内核上下文切换的开销
  - uni-kernel 是一种特别的内核结构，整个内核只运行一个应用程序，因此可以把应用程序直接放在内核态从而消除上下文切换的开销，Hsuan-Chi Kuo 等人的文章 A Linux in Unikernel Clothing 中，讨论了将通用 Linux 内核通过补丁、配置项修改的方式变化为 uni-kernle 的方法，实现了接近于一般 uni-kernel 的性能
  - Zhe Zhou 等人的文章 Userspace Bypass: Accelerating Syscall-intensive Applications 中，实现了用户旁路系统，该系统可以动态地检测当前应用程序的热门系统调用，而后通过二进制翻译的方法将两个频繁发生的系统调用之间的代码安全地移入内核态运行，从而降低了上下文切换次数，达到了与以往系统调用加速方法相近的性能，同时对应用程序实现了二进制兼容
- 异步的系统调用
  - Zach Brown 的文章 Asynchronous System Calls 讨论了 Linux 中异步系统调用实现的历史技术路径

Userspace Bypass: Accelerating Syscall-intensive Applications

1 已有的系统调用加速方法

- 异步系统调用
- 批处理系统调用
- unikernel
- 核内沙盒
- 内核旁路

优势：性能佳，但都需要程序员修改应用程序

2 传统系统调用的开销:

- 直接开销

  上下文切换开销：“A no-op system call with KPTI enabled can cost 431 CPU cycles, as measured by Mi et al. on Intel Skylake and seL4” (Zhou et al., p. 35) (pdf)

- 间接开销

  缓存污染开销：“Also on our platform, a pwrite syscall can degrade the IPC of the following userspace instructions from 2.9 to 0.2 (indirect costs). The IPC slowly goes back to 2.1 after executing 20,000 instructions. Figure 2 shows the trend of IPC by time elapsed.” (Zhou et al., p. 35) (pdf)

3 UB 设计概览

1. 最小化开发者的负担：二进制兼容
2. 最小化对内核架构的修改
3. 尽量达到其他方法的性能

- “Hot syscall identifier.” (Zhou et al., p. 37) (pdf)
- “BTC translator.” (Zhou et al., p. 37) (pdf)

4 Hot Syscall Identifier

- UB 标准

    - Tpath = 1000 instructions
- 模块实现
    - 系统调用采样：系统调用密集型每秒 100K 次，采样 10%，每分钟小于 500K 次，不会带来显著开销
    - 粗糙剖析：对于每秒系统调用小于 50K 次的线程，不进行下一步精细剖析
    - 精细剖析：每轮监控 15K 次，并维护表格记录
        - 系统调用的 RIP
        - 该系统调用接下来 4us（Tpath时间）内产生系统调用的次数

      若大于 900 次，则认为是频繁的
- 实验证明该模块对平台参数不敏感

5 BTC Runtime and Translator

思考1：如果存在一种运行时检查机制，保证应用程序的安全性，是否还需要特权级？

思考2：如何对应用程序加以约束（尽量小地影响功能）以保证程序员在编程时就避免危险操作？

思考3：内核架构之间的相互转换关系，能否实现动态转换？是否需要硬件的支持？

6 实验

性能与其他相当，逊于 eBPF，这是因为 UB 仅优化了上下文切换的开销。

结论：它的好处在与对应用程序的编写没有约束，无需重构

局限：

- 安全
    1. 侧信道攻击，因为用户空间代码被提升为了内核代码，可能用于窃取内核内存数据
    2. BTC translater 可能无法清除未知特权记录的 x86 指令，可通过指定指令白名单，遇到不在名单内的指令就停止提升
    3. 内核竞争导致的 TOCTOU 攻击
- 性能还不够好（相比于其他优化方法）
- 异步 IO 不是 UB 优化的目标，如：线程划分，一些线程进行运算密集型任务，另一些进行 IO，可能被阻塞

A Linux in Unikernel Clothing

1 披着 unikernel 外衣的 Linux

- 系统调用开销消除
- 通过配置进行裁减和定制

特点：融入了 Linux 软件生态，并且具有 unikernel 的特征

2 unikernel

两类：

- language-based：特定的编程语言运行时
  - 小镜像
  - 基于编译的检查和优化
  - 基于语言的安全性
- POSIX-like：尝试提供 POSIX 兼容，单地址空间、单特权级

问题：基于语言的 unikernel 需要重构应用，而类 POSIX 则是在重复实现 Linux（却无法融入 Linux 社区）

3 “Lupine Linux” (Kuo et al., 2020, p. 3) (pdf)

- specialization
  - Lupine 仅保留了283 个内核配置项，作为基础
- system call overhead elimination

应用程序清单：

- 内核配置
- 启动脚本

4 实现

- config 裁减
  - 应用程序特定的配置

    - 可以精确到应用使用了哪些系统调用
    - 精确到使用哪些内核服务：proc 文件系统、压缩、加密
    - 无需配置内核的 debug 功能

    unikernel 的最小化哲学，以应用需要为中心

  - 非必要配置
    - unikernel 面向单核场景
    - unikernel 面向特定平台

      因此可以针对性地对配置项进行缩减

- KML patch
  - 对内核的补丁：将唯一一个程序提升到内核态运行
  - 对 libc 的补丁，将 syscall 指令替换为 call

5 评估

结论：

- 首先，我们确认**内核专业化（定制）**非常重要：与最先进的 VM 相比，Lupine 的映像大小减少了 73%，启动时间加快了 59%，内存占用减少了 28%，吞吐量提高了 33%。然而，我们发现**应用程序级粒度的专业化可能并不重要**：只有 19 个特定于应用程序的选项涵盖了 20 个最流行的应用程序（占所有下载量的 83%），并且我们发现使用通用的应用程序配置最多会降低 4% 的性能。
- 其次，我们发现，虽然在相同权限域中运行应用程序在微基准测试中性能提高了 40%，但在宏基准测试中仅提高了 4%，这表明系统调用开销不应成为unikernel 开发人员的主要关注点。
- 最后，我们表明 Lupine 避免了类 POSIX unikernels 的主要缺陷，这些缺陷源于不基于 Linux，包括缺乏对未经修改的应用程序的支持以及高度优化的代码的性能。

配置

- 20 个热门应用占据了 83% 的下载量
- 手动测试需要打开哪些配置选项 `CONFIG`

6 超越 unikernel

- unikernel 的限制：多处理器、特权级等，导致不是所有应用都能运行，往往导致其丧失了通用性
- 实验一：后台控制进程对系统调用延迟几乎没有影响
- 实验二：资源竞争且有上下文切换
  - 线程切换：模拟 unikernel 的情景

    - 进程切换

  进程切换不比线程切换慢!

- 实验三：SMP 的影响

7 讨论

- 有效性威胁
    - 构成 Lupine 的最小配置可能不唯一，如 -O2 和 -Os
    - 如何获得配置清单？
    - 有些语言的包管理器可以分析依赖，从而得到“配置清单”
- 未达到的一些 unikernel 特点
    - 无法在 unikernel 监视器上运行
    - 不可变的基础设施（？）
    - 缺乏足够的编译时优化
- 未来工作
    - “Future research efforts should focus on making Linux specialization more effective and accessible.” (Kuo et al., 2020, pp. -) (pdf)

Cassyopia: Compiler Assisted System Optimization

1 简介

本质上是通过编译手段将多个系统系统调用合为一个，从而减少开销。

只需要修改编译器，而不需要修改应用程序代码

2 实现

如何发掘这些可以优化的机会?

- 系统调用流图，类似于程序流图
    - 每个结点代表一次系统调用
    - 相邻的系统调用使用箭头连接

    - 边的权重表示序列重复的次数
- 目标：找到图中高频率的调用序列，如果不相邻，则尝试通过编译技术重构代码让它们相邻

实现：在内核中添加了 240 号系统调用与可加载的内核模块，对编译器进行针对性修改以达到将系统调用聚集的目的

3 测试

测试：

- 拷贝程序提升不大，因为受到 IO 速度的限制
- 视频软件解码器提升很大（25%左右），因为属于 CPU 密集型