先来回顾一下系统调用的过程

- 首先用户程序执行一条指令触发同步异常，如：IA-32中的int 0x80，riscv中的ecall

- 而后陷入内核，保存用户空间的上下文，内核根据系统调用号进行分发，找到用户传入系统调用号对应的具体处理函数

- 接下来内核开始运行具体的处理函数

- 最后恢复用户上下文，从内核返回

在这个过程中，系统调用的开销可以分为直接开销和间接开销：

- 直接开销就包括了上下文切换中保存恢复上下问文的代码，以及在找到具体的系统调用处理函数前的分发代码

- 间接开销包括了由于地址空间等切换造成缓存污染所导致的用户程序IPC下降

系统调用加速的核心就是减少用户内核上下文之间的切换，根据以往的工作可以分成这几类：

第一篇：

- 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

第二篇：

- 优势：性能与其他方法相当的情况下实现了应用程序的二进制兼容。

- 不足：开销还是大了一些，安全性

第三篇：

unikernel：唯一的，整个系统只运行一个应用程序，没有用户态和内核态，单一的地址空间，系统调用开销转换为函数调用开销。

- 系统调用开销消除

- 通过配置进行裁减和定制

优势：融入了 Linux 软件生态，并且具有 unikernel 的特征

局限：如何生成应用程序的配置清单？如何获得支持一个应用程序的最小化配置？