Library OS

TOC

1. 先介绍一些 OS 相关知识
2. LibOS 的起源
3. 当代 LibOS 的技术介绍
4. LibOS 在 SGX 领域的应用

OS Background

在座的都是大佬，我来班门弄斧了

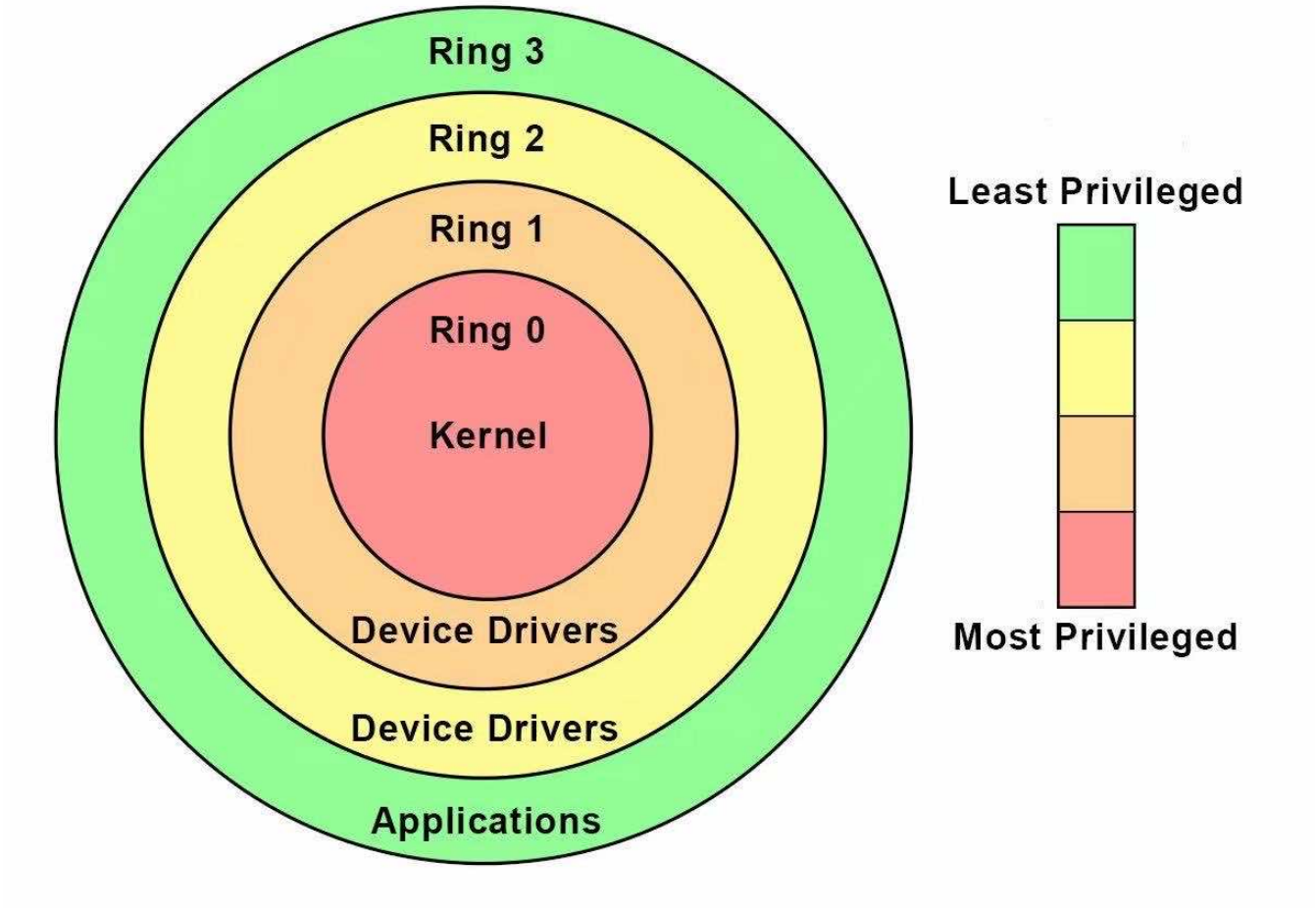
HPD

分级保护域，Hierarchical Protection Domains

CPU 提供了 RING 0～3 四个不同的保护域，不同域有不同的指令执行权限。

一般来说，OS 运行于 Ring 0，拥有操作设备的权限。 用户程序运行于 Ring 3，仅有使用 CPU 进行计算的权限。 通俗称为 内核/用户 态。

（HPD 有时候也称为 CPL(Current Privilege Level)、Protect Ring）



System Call

当用户程序需要调用硬件资源等特权操作时，需要通过 kernel 提供的接口来完成。 这些接口就称为 system call。

1. 将相关 syscall 号分别存入 %rax 和 %orig_rax。
2. 把最多六个参数存入 %rdi, %rsi, %rdx, %r10, %r8, %r9
3. 保存用户线程上下文
4. 切换到内核态，kernel 查找并执行相应的 syscall 函数
5. 执行结果存入 %rax，内核切换回用户态，恢复用户线程上下文

user/kernel 的切换可以通过 软中断 0x80 或 sysenter/sysexit、syscall/sysret 等指令进行切换

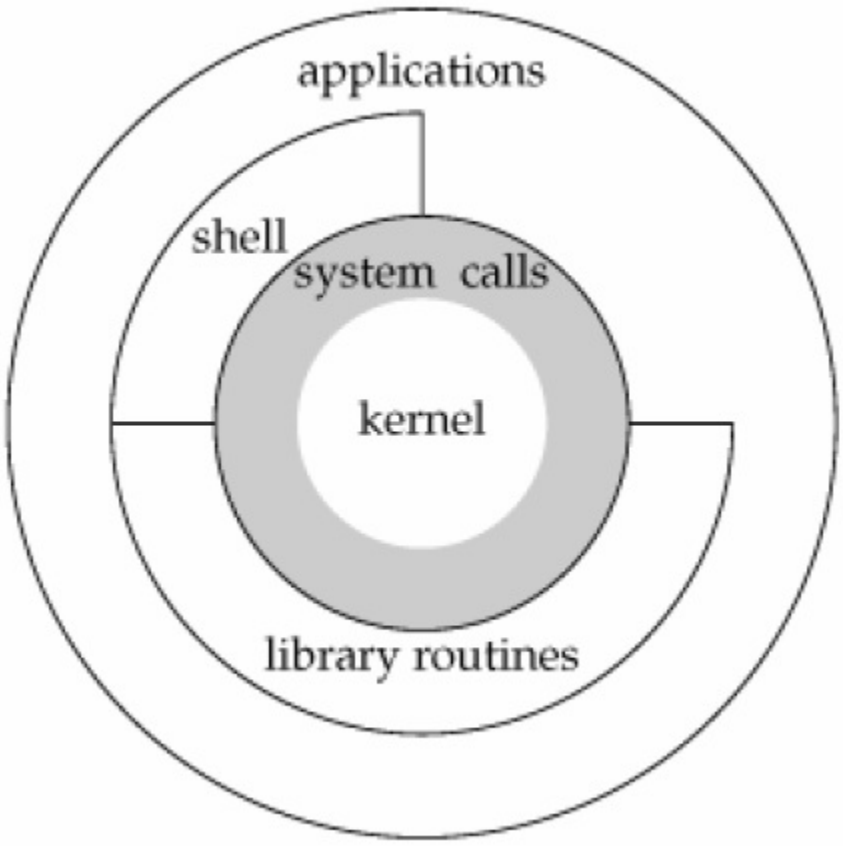
- x86-32：
    - Intel: 0x80、sysenter/sysexit
    - AMD: 0x80、syscall/sysret
- x86-64: syscall

LibC

但是实际上用户程序一般并不会直接和 kernel 交互（因为除了 syscall 外，还有大量其他相关工作）， 这些“幕后的相关工作”，一般由 libc 来完成。

常用的 libc 包括 glibc、musl

用户程序和 kernel 的边界究竟是 syscall 还是 libc 尚无定论



题外话，为什么 Go 不用 libc？

LibOS

以 syscall 隔离开 user 和 kernel，导致了大量的上下文切换和数据拷贝开销。

最早的 LibOS 正是一种试验性的探索方向，试图简化 kernel 的职责，kernel 只负责最基础的硬件保护， 而将对设备的操作直接放到用户态里，以 lib 的形式提供给应用程序。

但是该领域随着 VMM 的崛起而被人遗忘。

后来随着云时代的到来，传统虚拟机方案对性能的损耗引起人们的重视， LibOS 重新作为一种对 VM 的高性能替代方案再次被重视。

VMM & VMX

补充介绍一下 VMM 和 VMX。

VMM 和 VM 也就是我们俗称的虚拟机管理器和虚拟机

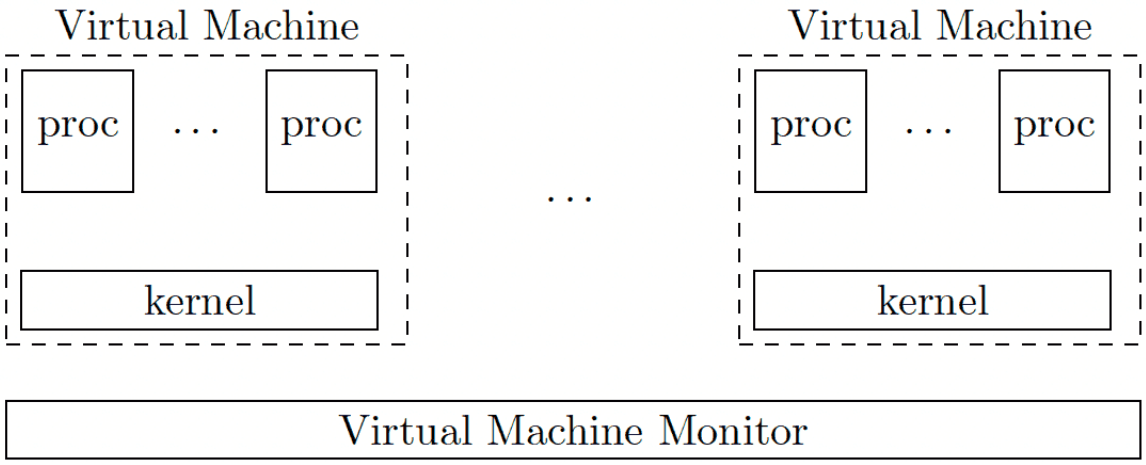
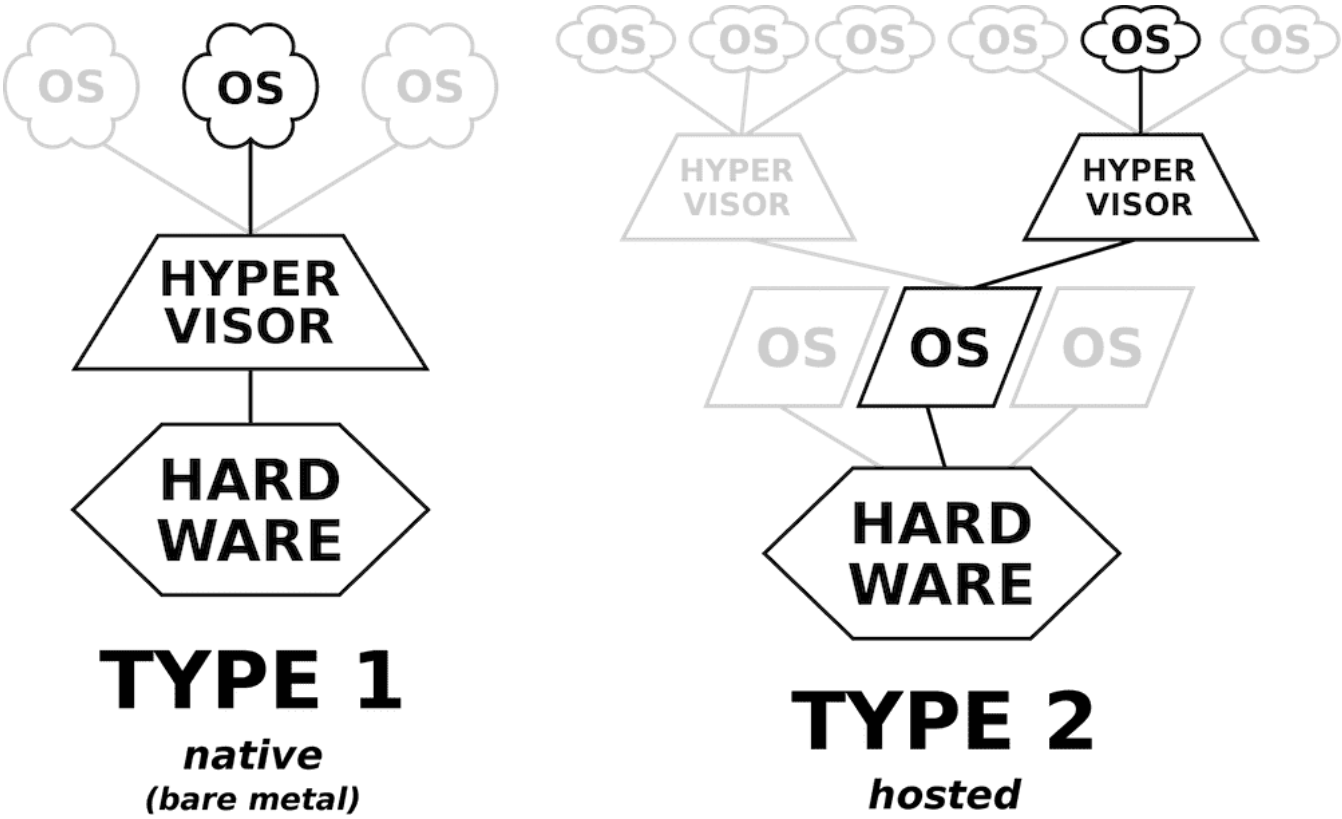


Figure 2: A Virtual Machine Monitor.

VMM 有时候也称为 hypervisor，根据其是运行于 Host OS 之上还是 Bare Metal 之上可以区分为两个类型



简而言之，最早的 VMM 就是试图用用户进程运行一整个操作系统。这有很大的难度，原因之一就是对指令权限的控制非常繁琐和困难。

后来各家芯片厂商提出了硬件虚拟化方案，Intel-VMX（或叫 VT-x）和 AMD-V。

VMX 为 CPU 新增了两种状态：root/non-root。 和原有的 user/kernel 组合后可得四种状态:

1. root/user: Intel VMM 运行于用户态
2. root/kernel: Intel VMM 运行于内核态
3. non-root/user: VM 运行于用户态
4. non-root/kernel: VM 运行于内核态（guest OS）

硬件支持 VMM 对 Ring0 的虚拟化操作。

可以设定让 VM CPU 在遇到指定指令时触发 `VM EXIT`， 将控制权切换给 VMM，从而得以实现对任意指令的拦截。 让 VM 完全意识不到自己运行于虚拟环境之中，有时候也被称为 `blue pill`。

一句话总结就是，硬件虚拟化极大地降低了 VMM 的实现难度，也提高了 VM 的性能。

LibOS & hypervisor

再回过头来看 LibOS，实际上它和 VM 的分野也不是那么的清晰。

共同点：都是将 App 封装于一个独立的“沙箱”中

不同点：

- VM 会提供一个完整的 OS，仍然有 user/kernel 的权限区别
- LibOS 完全运行于用户空间，提供部分 OS 的功能

双方也并没有那么泾渭分明，可以笼统地将 libos 认为是轻量级的 VM。

有些 libos 可能会直接和 OS 交互，有些 OS 可能会依托于 VMM。

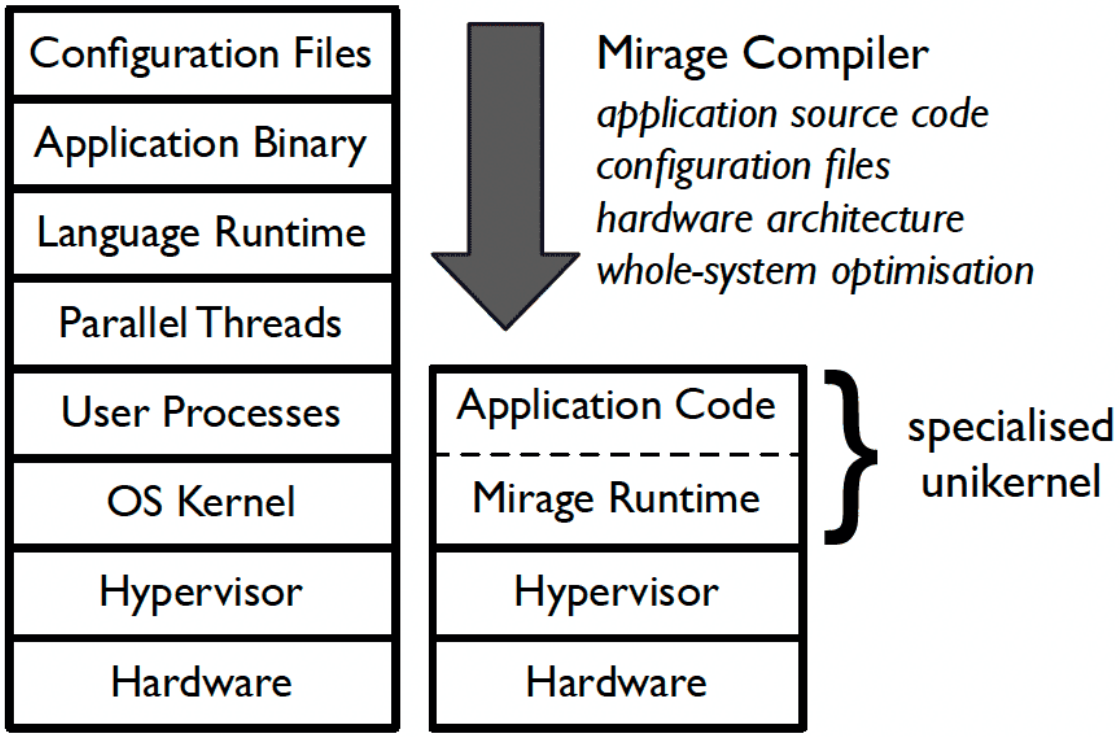


Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

LibOS & OS

LibOS 的发展方向非常多元，其不同的需求也导致了其非常多样化的实现路径

- 性能优先

为了降低 user/kernel 的切换开销，在 libos 中重新实现 OS 的设备功能（如 tcp 协议栈）， 让所有的数据处理都在用户空间完成，再基于零拷贝等技术直接和设备交互。

- 兼容性优先

  兼容主流 OS 接口，用最小功能集封装 App，实现 App 的运行时迁移。 可以保存 App 任意时刻的运行状态，将 App 连同 libos 一起打包迁移。

- 安全加固

  libos 通过拦截 app syscall 实现对 app 的沙箱， 对 syscall 进行监听和修改以实现安全加固的需求。

LibOS 在 SGX 领域的应用

我近期主要关注 LibOS 在 SGX 领域的应用

SGX 为什么需要 LibOS？

这要从 SGX 的运行时限制讲起。

SGX 将一个进程的内存分为可信区和不可信区。要进入不可信区需要通过专门的指令（ECall）启用专属的 SGX 线程。

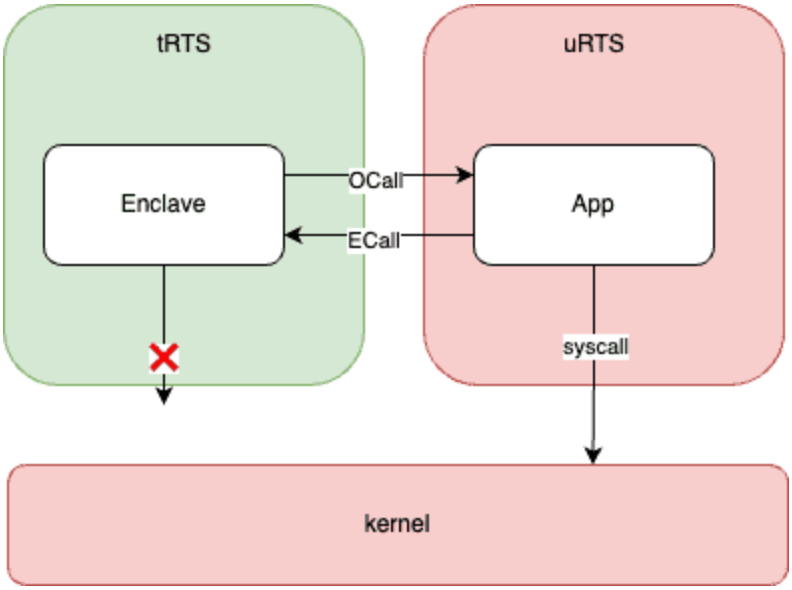
这些线程能够执行的 CPU 指令被严格限制，比如 syscall 就被禁止，会触发 #UD exception。

虽然 SGX 允许 Enclave 程序自定义 Exception Handler 去捕获 #UD 异常， 但是这些 Handler 并不允许发起 OCall 调用。

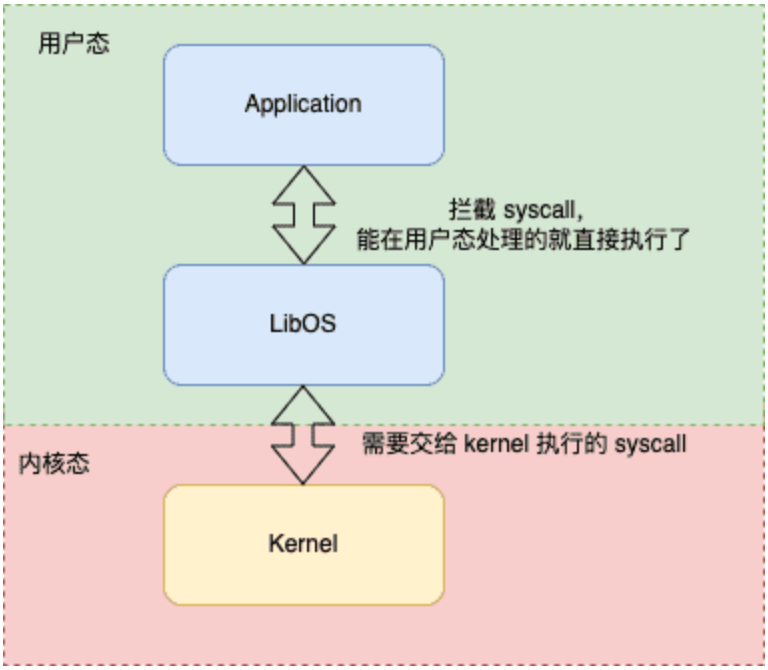
换句话说，仍然不能离开 enclave 区域，还是无法触及 kernel 的 syscall ABI。

SGX 最初是为计算型程序所设计，这些程序仅需要内存中的数据和用户态的 CPU 就可完成计算任务。

但是随着人们对通用型 TEE 的需求，会希望运行于可信区的程序同样能够完成如磁盘读取、网络 I/O 等业务， 而这些任务都需要 kernel syscall。



既然 SGX 原生无法满足 syscall 的需求，那么人们就将眼光转向拦截 syscall 的能手 LibOS

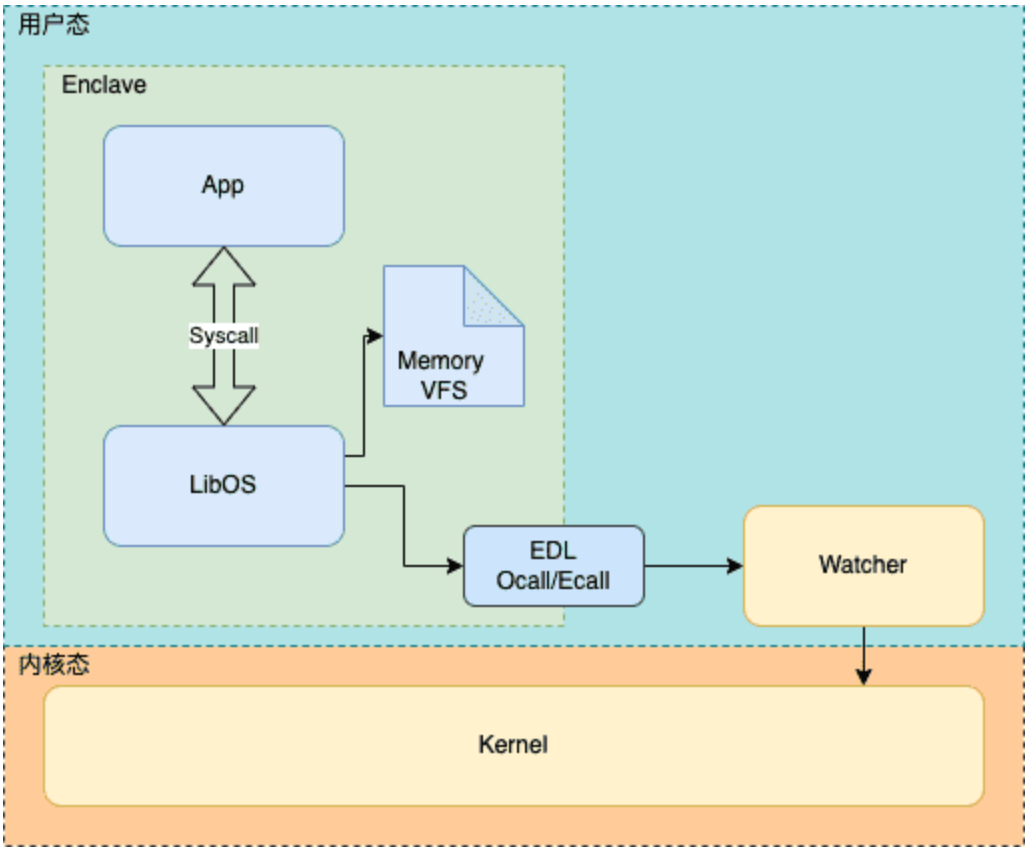


总结一下

SGX Enclave 不支持 syscall，但是应用程序需要 syscall。

所以用 LibOS 为应用程序提供 syscall，然后 LibOS 再自己想办法处理这些 syscall

一顿操作猛如虎后，最终架构就成了这样



trap and emulate

实际上，在 SGX 这种应用场景中，libos 的能力基础，全都建立于于对应用程序 syscall 的拦截上。

这种拦截并模拟 syscall 的操作，也称为 `Trap And Emulate`。

回顾一下用户程序发起 syscall 的几种方法：

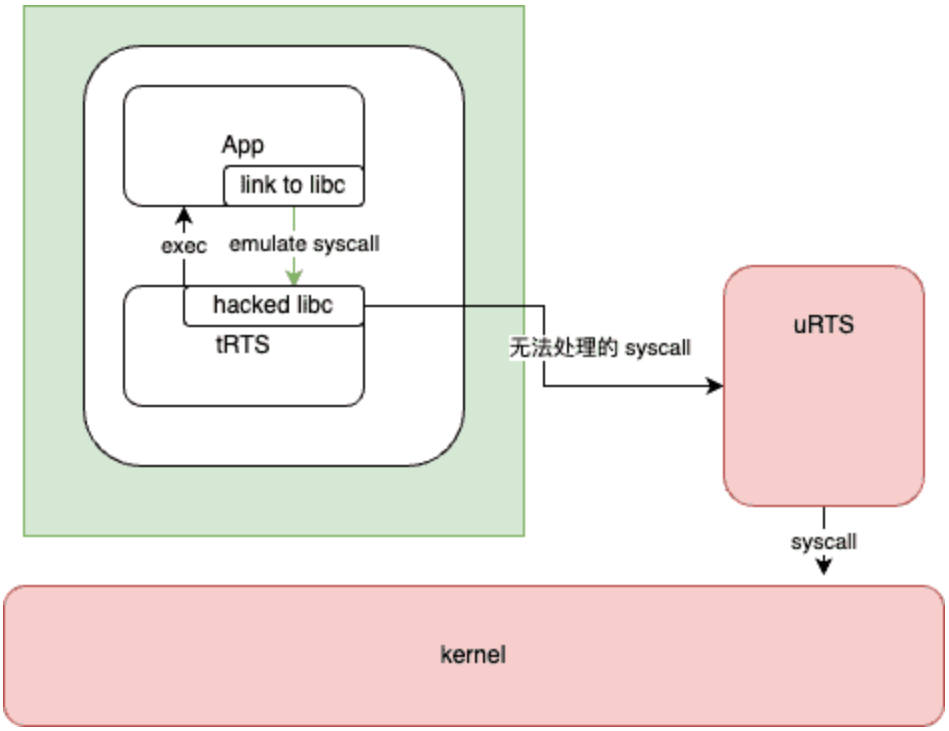
- interrupt 0x80
- syscall/sysenter instructions
- libc API

主流的应用程序，大多都是靠 libc 来调用 syscall，而且都能够编译为动态链接到 libc 的可执行程序。

即使是 python 脚本，它也能够通过一个动态链接到 libc 的解释器来执行。

那么 libos 只需要在装载阶段，将 libc 劫持为自己修改过的兼容 libc API 的库就行了。

然后在这些自定义的 libc 兼容库里，对 syscall 想做什么都可以。



那对于 Golang 这种不走寻常路，直接汇编调用 syscall 指令的怎么办？

目前看到的常见做法是，改 Go 的源码，让它用 libc …

```
54  55      TEXT runtime·exit(SB),NOSPLIT,$0-4
55  56          MOVL    code+0(FP), DI
56  57          MOVL    $SYS_exit_group, AX
57      -       SYSCALL
    58  +       CALL    runtime·invoke_libc_syscall(SB)
58  59          RET
59  60
```

拦截后能做什么？

最直接的好处是，即使 libos 什么也不做，仅仅是转发 syscall 给 kernel， 也让 enclave 程序拥有了 syscall 的能力， 这就让 enclave 程序可以读写文件，可以通过 socket 发起网络请求， 或者运行网络服务器。

Defense-in-Depth 其次，libOS 可以对 syscall 进行限制和监督，仅允许最小限度的 syscall， 减少暴露面，实现加固。

除此之外还可以干很多增强安全性或功能的能力。

比如，通过改写 socket 的 `listen, connect, recv, write` syscall，可以实现 TTLS，即在 syscall 这一层对连接启用 TLS 加密，应用程序完全无感知。

类似的，也可以通过改写 file 的 `read, write`，实现对文件系统的透明加密。

Trap And Emulate

前文介绍的 syscall 拦截都基于拦截动态链接的 libc。

那如果是没法改源码的静态编译可执行程序呢？

对于 SGX Enclave 程序而言，目前看来是没办法了。

那抛开 SGX，在 OS 领域，有其他拦截方式吗？

ptrace 大法好

syscall 在 kernel 中有两个 hook 点：

- `syscall-enter-stop`: 进入 syscall 前
- `syscall-exit-stop`: syscall 结束以后。

拦截 syscall 的技巧是，在 `syscall-enter-stop` 后， 把记录 syscall 编号的 `%orig_rax` 改写为一个不存在的值（如 -1 或 MAX_UINT64）。

这样通过 PTRACE_SYSCALL/PTRACE_SYSEMU 恢复 tracee 运行后， kernel 无法查找到合法的 syscall handler。

tracee 就会进入 syscall-exit-stop 再次触发 tracer。

此处可以通过 `%rax` 设置 syscall 的返回值。

同理，一些返回参数可以通过 `%rdi %rsi %rdx` 等寄存器来设置。这样就实现了一次 syscall 的劫持/模拟。

```
for (;;) {
    /* Enter next system call */
    ptrace(PTRACE_SYSCALL, pid, 0, 0);
    waitpid(pid, 0, 0);  // <- tracee 被切出，返回给 tracer

    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, 0, &regs);  // <- 获取 tracee 的寄存器

    // 此处模拟的是 syscall 黑名单
    // 如果判断一个 syscall 不被允许，就将其标记为 blocked
```

```
    int blocked = 0;
    if (is_syscall_blocked(regs.orig_rax)) { // <- 通过 %orig_rax 获取系统调用号
        blocked = 1;
        regs.orig_rax = -1; // 将 %orig_rax 设置为一个非法的数来拒绝改次 syscall
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }

    ptrace(PTRACE_SYSCALL, pid, 0, 0);  // <- 恢复 tracee 执行
                                        // 因为 %orig_rax 是非法的，所以 tracee
会跳过 syscall，
                                        // 直接进入 `syscall-exit-stop`
    waitpid(pid, 0, 0);  // <- 触发 syscall-exit-stop，回到 tracer

    if (blocked)
        regs.rax = -EPERM; // 通过 %rax 将 sycall 返回设置为错误
        ptrace(PTRACE_SETREGS, pid, 0, &regs);
    }
}
```

Ps. 为减少 context switch，`PTRACE_SYSEMU` 方式恢复的 tracee 会忽略 syscall-exit-stop。

LibOS? VMM?

除了 ptrace 外，另一条路径就是依赖 VMX 硬件虚拟化提供的指令拦截能力。

如 gVisor 就提供了两个实现方式，ptrace 或 KVM。

这也是为什么前面会说，LibOS 和 VM 的区分实际上并不很明确。笼统地说，LibOS 就是种轻量级 VM。

比如 LibOS + KVM 的用户空间负载应该远小于 QEMU + KVM。

Thanks