项目申请书：基于 Occlum 实现 System V 信号量系统调用

1. 项目简述

Occlum 是一个面向英特尔 SGX 的内存安全、多进程库操作系统（LibOS）。作为 LibOS，它能让传统应用程序在 SGX 上运行，几乎无需或完全无需修改源代码，从而实现保护。

Occlum有以下显著特点：

- 高效多任务处理。Occlum 提供轻量级的 LibOS 进程：从某种意义上说，这些进程是轻量级的，因为所有 LibOS 进程共享同一个 SGX 隔离区。与每个隔离区都有独立的重载型 LibOS 进程相比，Occlum 的轻量级 LibOS 进程在启动时速度可快达 1000 倍，在进行 IPC（进程间通信）时速度可快 3 倍。此外，Occlum 还提供了一个可选的 PKU（用户空间保护密钥）功能，以便在需要时增强 Occlum 的 LibOS 与用户空间进程之间的故障隔离。
- 多种文件系统支持。Occlum 支持各种类型的文件系统，例如只读哈希文件系统（用于完整性保护）、可写加密文件系统（用于保密保护）、不可信主机文件系统（用于 LibOS 与主机操作系统之间方便的数据交换）。
- 内存安全性。Occlum 是第一个用内存安全编程语言（Rust）编写的 SGX LibOS。因此，Occlum 更不容易出现底层的内存安全漏洞，更值得信赖地运行安全关键型应用程序。
- 易用性。Occlum 提供用户友好的构建和命令行工具。在 SGX 隔离区内运行 Occlum 应用程序可以简单到只需输入几个 shell 命令。

2. 申请理由

我有一定基于 rust 开发操作系统内核的基础。我参加了 2025 春夏季开源操作系统训练营，完成了 rCore 实验并进入宏内核阶段的项目实习。

在实习中我主要开发 starry-next OS，贡献如下：

1. 我首次在 starry-next 实现了System V共享内存机制 github 链接，帮助很多参加全国大学生 OS 比赛的同学通过了 iozone 测例，收获老师同学的好评。
2. 首次在 starry-next 实现了文件页缓存机制 github 链接，并与文件读写和 mmap 相配合。目前在单进程场景下已经较为健壮，正在尝试适配高并发场景。
3. 预计会在训练营结束前（6月20日之前）将完整的 PageCache、文件操作、mmap、shm 机制互相配合，一同合并到 starry-next 主线。

在这段经历中，我收获如下技能：

1. 熟悉 Linux IPC 机制相关源码。
2. 熟练使用 Rust 语言进行OS开发。

我对rust 语言以及操作系统都拥有浓厚的兴趣，而 occlum 是一个开源的、用rust 编写的商用级操作系统，我非常渴望能够为其做出贡献，特别是在我已经较为熟悉的 IPC 部分。

3. 项目方案

基于 Occlum 实现完整的 System V 信号量（`semget`, `semop`, `semctl`）系统调用涉及多个层面的设计和安全考量。Occlum 作为 SGX LibOS，核心挑战在于：**如何在隔离的 Enclave 内安全、高效地模拟共享内存状态和跨进程同步原语。**

3.1 设计目标

1. POSIX 兼容性：兼容标准 System V 信号量语义。
2. 进程内兼容性：同一 Enclave 内进程间信号量操作必须正确。
3. 跨 Enclave 限制：不同 Enclave 间信号量默认不共享（符合 SGX 隔离模型）。
4. 安全性：
    - 保护信号量状态机密性（加密存储）。
    - 保证信号量状态完整性（防篡改）。
    - 防止 TOCTOU 攻击。
5. 性能：最小化 Enclave 切换和加密开销。

3.2 关键组件设计

1. 信号量核心数据结构（Enclave 内）

```
// Occlum 内部表示 (存储在 Enclave 内存)
struct OcclumSemaphore {
    sem_id: i32,                    // 全局唯一 ID (Enclave 内)
    key: key_t,                     // 用户提供的 key
    creator_pid: pid_t,             // 创建者 PID
    permissions: mode_t,            // 权限位 (rwx)
    nsems: usize,                   // 信号量集合大小
    semaphores: Vec<SemValue>,      // 信号量值数组 (加密存储)
    last_pid: pid_t,                // 最后操作者 PID (用于 IPC_STAT)
    last_op_time: timespec,         // 最后操作时间
    last_change_time: timespec,     // 最后值变更时间
    // 同步原语 (保护内部状态)
    internal_lock: Mutex<()>,       // 粗粒度锁 (简化设计) 或 RWLock
    // 等待队列 (用于 semop 阻塞操作)
    wait_queues: Vec<WaitQueue>,    // 每个信号量一个等待队列
}

// 信号量值 (加密存储)
struct SemValue {
    encrypted_value: [u8; 16], // AES-GCM 加密的值 + Tag
    // 或使用 Occlum 提供的安全存储 API (如 protected_fs)
}

// 等待队列项
struct Waiter {
    pid: pid_t,
    op: SemOp,                  // 请求的操作 (用于检查是否可唤醒)
    wake_signal: Condvar,       // 条件变量用于唤醒
}
```

2. 全局信号量管理器 (Enclave 内)

```
struct SemaphoreManager {
    // 全局信号量表 (Key -> OcclumSemaphore) 或 (ID -> OcclumSemaphore)
    semaphores: HashMap<i32, Arc<OcclumSemaphore>>, // 使用 ID 索引
    key_to_id: HashMap<key_t, i32>,                 // Key 到 ID 映射
    next_id: AtomicI32,                             // 下一个可用 ID
    global_lock: Mutex<()>,                         // 保护管理器状态
}

impl SemaphoreManager {
    fn new() -> Self { ... }
    fn semget(&self, key: key_t, nsems: i32, semflg: i32) -> Result<i32> { ... }
    fn semctl(&self, semid: i32, semnum: i32, cmd: i32, arg: *mut c_void) -> Result<i32> { ... }
    // semop 通常在信号量对象自身上实现
}
```

3.3 系统调用实现

`semget` 系统调用流程：

1. 参数检查：验证 `nsems` > 0，`key` 有效，`semflg` 合法。
2. 查找/创建：
   - 获取 `SemaphoreManager` 全局锁。
   - 根据 `key` 查找 `key_to_id`：
     - 存在：检查权限（`semflg` 中的 `IPC_CREAT | IPC_EXCL`）和 `nsems` 大小。
     - 不存在：若 `IPC_CREAT` 被设置，则创建新信号量集（`next_id` 分配 ID，初始化 `OcclumSemaphore`，插入 `semaphores` 和 `key_to_id`）。
3. 返回：返回信号量 ID 或错误。

`semop` 系统调用流程：

1. 参数检查：验证 `semid` 有效，`sops` 指针有效，`nsops` > 0。
2. 查找信号量：通过 `semid` 从 `SemaphoreManager` 获取 `Arc<OcclumSemaphore>`。
3. 获取信号量锁：获取目标信号量对象的 `internal_lock`。
4. 模拟原子操作：
   - 预检查 (Try Phase)：遍历所有 `sops`：
     - 检查信号量索引 `semnum` 有效。
     - 解密 `semaphores[semnum]` 得到当前值 `cur_val`。

        - 根据 `sem_op` 类型执行检查：
            - `> 0` (V 操作)：总是允许。
            - `0` (Z 操作)：检查 `cur_val == 0`。
            - `< 0` (P 操作)：检查 `cur_val >= |sem_op|`。
        - 如果所有操作都允许：继续到执行阶段。
        - 如果任何操作不允许且 `sem_flg` 包含 `IPC_NOWAIT`：释放锁，返回 `EAGAIN`。
        - 否则 (需要阻塞)：将当前进程加入对应信号量的 `wait_queues[semnum]`，在 `Condvar` 上阻塞（释放锁等待唤醒）。
    5. 执行操作 (Undo Phase)：
        - 再次检查信号量值（防止唤醒后状态改变）。
        - 遍历 `sops`：
            - `cur_val += sem_op`。
            - 加密新值写回 `semaphores[semnum]`。
            - 更新 `last_pid`, `last_op_time`, `last_change_time`。
        - 唤醒等待者：遍历所有信号量的等待队列，检查是否有进程的请求现在可以满足。如果有，唤醒它们。
    6. 释放锁：释放 `internal_lock`。
    7. 返回：成功返回 0。

`semctl` 系统调用流程：

1. 参数检查：验证 `semid` 有效，`cmd` 合法，`semnum` 在范围内（对于需要它的命令）。
2. 查找信号量：通过 `semid` 获取 `Arc<OcclumSemaphore>`。
3. 获取锁：获取信号量的 `internal_lock`。
4. 处理命令：
    - `IPC_STAT`：将信号量元数据复制到用户提供的 `semid_ds` 结构（`arg`）。
    - `IPC_SET`：更新权限和所有者（需权限检查）。
    - `IPC_RMID`：标记删除。释放资源（从管理器移除），唤醒所有等待者并返回 `EIDRM`。
    - `GETVAL`：解密返回指定 `semnum` 的值。
    - `SETVAL`：设置指定 `semnum` 的值（需权限），加密存储。
    - `GETPID` / `GETNCNT` / `GETZCNT`：返回元数据。
    - `GETALL` / `SETALL`：批量获取/设置所有值（需加解密）。
    - `SEM_INFO` / `SEM_STAT`：返回系统级信息（Occlum 可能简化）。
5. 释放锁。
6. 返回：根据命令返回结果或 0。

3.4 核心挑战与解决方案

状态存储安全

- 问题：信号量值必须保密（可能隐含业务信息）和防篡改。
- 方案：
    - 加密存储：使用 Enclave 密钥（如密封密钥）或每个信号量的密钥加密 `semaphores` 数组（如 AES-GCM）。
    - 内存隔离：信号量结构体本身存储在 Enclave 内存中，受 SGX 保护。
    - Occlum 安全存储：复用 Occlum 的文件系统加密机制（如 `protected_fs`）持久化（如果需持久化）。

原子性与一致性

- 问题：`semop` 要求原子地执行一组操作。
- 方案：
    - 粗粒度锁：使用单个 `internal_lock` 保护整个信号量对象。简单但可能影响并发性能。
    - 细粒度锁：为每个信号量值或等待队列配锁。更复杂，需防死锁。
    - Enclave 内原子性：锁和操作在 Enclave 内执行，天然原子（无抢占）。

阻塞与唤醒 (`semop`)

- 问题：如何在 Enclave 内安全高效地实现进程阻塞/唤醒。
- 方案：
    - 条件变量 (`Condvar`)：与 `Mutex` 配合使用。进程在检查条件失败时在 `Condvar` 上等待（`wait`），释放锁。执行 `semop` 的进程在操作完成后，遍历等待队列，对满足条件的 `Condvar` 执行 `notify_one` / `notify_all`。
    - 等待队列：为每个信号量值维护一个队列，存储等待的进程上下文（PID）和其操作请求（用于唤醒时重试检查）。
    - 超时处理：`semtimedop` 需支持。`Condvar::wait_timeout` 实现。

跨进程兼容性

- 问题：同一 Enclave 内不同进程需通过 `semid` 访问同一信号量对象。

- 方案：
  - 全局管理器：`SemaphoreManager` 作为 Enclave 单例，所有进程共享其地址空间（Occlum 进程模型）。
  - 共享内存映射：Occlum 内进程共享 LibOS 数据结构。管理器及其管理的 `OcclumSemaphore` 对象存储在共享内存区域。

密钥管理 (`semctl SETVAL/GETVAL`)

- 问题：用户直接读写值，需透明加解密。
- 方案：
  - 拦截系统调用：在 `semctl` 实现内部，对 `SETVAL` / `GETVAL` / `SETALL` / `GETALL` 等涉及值的命令，在读写 `OcclumSemaphore.semaphores` 数组前后进行加解密操作。
  - 密钥存储：密钥存储在 `OcclumSemaphore` 内部或由全局管理器管理（使用 Enclave 主密钥派生）。

资源管理与销毁 (`IPC_RMID`)

- 问题：安全释放资源，处理等待进程。
- 方案：
  - 标记删除：设置删除标志。
  - 唤醒所有等待者：遍历所有等待队列，唤醒所有进程，它们将收到 `EIDRM` 错误。
  - 引用计数：使用 `Arc` 管理 `OcclumSemaphore`。当最后一个引用（来自进程或管理器）被释放时，安全擦除内存。

3.5 进一步优化

与 Occlum 架构集成

1. 系统调用入口：
   - 修改 Occlum 的 `syscall_table.rs`，注册 `semget`, `semop`, `semtimedop`, `semctl` 的处理函数。
   - 处理函数调用上述 `SemaphoreManager` 的方法。
2. 进程管理：
   - 进程退出时 (`exit` 系统调用)，需清理该进程在所有信号量等待队列中的条目（防止僵尸等待）。
3. 时间管理：
   - 使用 Occlum 的时钟源 (`clock_gettime`) 更新 `last_op_time` / `last_change_time`。
4. 安全增强：
   - 检查用户指针有效性（`user_check.rs`）。
   - 敏感操作（如 `semctl(IPC_RMID)`）进行权限检查（`capability.rs`）。
5. 持久化 (可选)：
   - 若需 Enclave 重启后恢复信号量状态，需将 `SemaphoreManager` 状态（或关键信号量）通过 Occlum 的 `protected_fs` 或密封存储 API 持久化。

性能优化考虑

1. 锁粒度：
   - 评估锁竞争。如果信号量操作频繁，考虑细粒度锁（如每个信号量值一个锁）。
2. 批处理唤醒：
   - `semop` 操作完成后，批量检查等待队列，减少不必要的唤醒。
3. 共享内存优化：
   - 确保信号量核心数据结构位于 Enclave 物理内存 (EPC) 的高效区域。

测试策略

1. 单元测试：针对 `OcclumSemaphore` 和 `SemaphoreManager` 的核心逻辑（加解密、锁、队列）。
2. POSIX 一致性测试：
   - 移植 Linux 的 `ltp` (Linux Test Project) 中 IPC 信号量测试用例。
   - 使用标准库测试 (如 `glibc` 的 `tst-sysvipc`)。
3. 并发压力测试：模拟多进程高频竞争操作信号量。
4. 安全测试：
   - 尝试伪造信号量 ID/key。
   - 尝试越权访问 (`semctl` 权限检查)。
   - 测试资源耗尽（创建大量信号量）。
5. 性能基准测试：对比原生 Linux 与 Occlum 内信号量操作的延迟和吞吐量。

4. 项目时间规划

时间段	任务
7月之前	仔细阅读 occlum 源码，熟悉项目组织架构

时间段	任务
7.1 ~ 7.30（4个周）	为 Occlum 支持 System V 信号量机制，包括 semget、semctl、semop、semtimedop 等系统调用。
8.1~8.7（1个周）	测试驱动，编写一些简单的测试应用程序，验证实现的正确性，给出性能测试结果，并依此进一步完善系统。
8.8~8.30（3个周）	调研一到两个使用了System V信号量的真实世界应用程序，并尝试在Occlum中正确运行。
9.1~9.15（2个周）	编写实现与测试文档，输出报告，尝试实现更多扩展任务。