XPC vs uintr

- xpc 简介
- ipc 简介
- uintr 实现 ipc
- 分析与讨论

XPC

目标：一种同步/高效/安全/易用的 IPC 机制

XPC = 用户态内存切换 + 特殊映射传递数据 = 高效 RPC

- 同步：使用新增指令，硬件完成进程切换
- 高效：bypass kernel + no memory copy
- 安全：传递内存，非共享（hand over）
- 易用：兼容常用内存使用模式 + 兼容同步接口

什么是 IPC ?

- fs service / database service / android binder
- IPC = RPC = 指定处理函数 + 数据传递 = 数据传递 + 同步机制

IPC 步骤

- 建立链接 (都比较麻烦，无法 bypass kernel)
- 发出请求
  - 传统：调用某一个系统调用，讲请求 pend 在 server 的请求队列（内存），通知 server (信号量等)
  - XPC: 新增指令 xcall，切换进程，特殊映射传递数据
- 响应请求
  - 传统: server 进程被唤醒，调用某一个系统调用 （查看某一块内存 buffer），响应请求。等待或者休眠。
  - XPC: 正常执行函数调用，完成后 xret（用户态切换进程）。
- 断开链接 (类似建立链接)

为什么慢?

- server 响应请求**不需要**内核介入（除了进程切换）
- client 发出请求**需要**内核介入
  - 没有完全基于 shmem 的 IPC 机制，无法同步

uintr

- 同步机制，6bit 信息（用来分辨来源）
- uintr + shmem == RPC ?
  - RPC = 数据传递 + 同步机制
  - RPC = mmap + uintr

简单设计

- server 注册时，默认注册 64 个匿名内存文件（可配置初始状态）
- client 建立链接(uitt) 时:
  - mmap 打开匿名文件，可以进行文件属性修改(如 size)。
  - 如果修改了文件属性，需要通知 server (可以通过另一个 uipi)
- client 发出 uipi 试做进行函数调用
  - 传参：将参数写入 mmap 映射的内存
  - 如果有必要，通过系统调用修改 mmap 文件属性

简单设计

- server 响应请求:
  - 特殊请求：默认，不可修改，最高优先级
    - 配置修改：通过系统调用更新对应的 mmap
  - 其他请求：从 mmap 段读取参数，处理请求，返回结果。

问题 / 分析

- 同步 vs 异步
- lantency / 进程 offline:
  - XPC: 不存在这个问题
  - uintr: 无解，对应 uintr_wait，依赖于 os 本身的调度。
- shmem 带来的安全问题
  - XPC: 不存在这个问题
  - uintr: 无解
    - 数据破坏：导致 server 崩溃
    - TOCTTOU：若对方不可信，可以进行一次额外拷贝（把这个问题丢给具体实现）。
    - 代码注入等：...

问题 / 分析

- 易用性 / 兼容性 / 软件成本
  - XPC: 兼容同步接口，需要大量内核修改和少量用户库修改。
  - uintr: 高度兼容当前的使用模式，只需要很少的内核和用户程序修改。
- 硬件成本
  - XPC: 挺大的，比较复杂的系统
  - uintr: 无
- 嵌套 RPC
  - XPC：完美
  - uintr: 等同于多次顺序的 RPC，但是本身发生的很少，不需要考虑

问题 / 分析

- 并行度
  - XPC: 完美，理论上并行度 = client 数量（受制于数据的同步互斥）
  - uintr: 类似传统方式，最大并行度 = MIN(server 所开的线程数量，64)
    - 协程：不影响并行度
- 返回值处理
  - XPC: 同步接口，处理很方便
  - uintr: 异步接口，pooling / uintr_wait

讨论

- 利用 (类dma)专属硬件 / 指令完成数据传输?
  - 可以解决 shmem 的问题?
- 调整内核调度，实现 uintr_wait 的快速响应?
  - 引入优先级，链接越多优先级越高
- 请求 pending ?
  - buffer 满之后无法 pending，需要调整 buffer 属性
  - 类似日志文件系统

讨论

- 退化?
  - 当负载过重时，server 无法常驻，退化为传统方式
- 异步内核?
  - 正交，似乎没啥联系
  - os ==> link mannger