

# Lecture9 IPC

IPC: Inter-Process Communication (进程间通信)

## 1. Intro

### 使用多个进程的应用

- 优点
  1. 功能模块化，避免重复造轮子 (e.g. DB、UI)
  2. 增强模块间隔离，增强安全保障 (敏感数据的隔离)
  3. 提高应用容错的能力，限制故障在模块间的传播
- 难点：不同进程拥有不同的内存地址空间
  1. 进程与进程间无法直接进行通信和交互
  2. 需要一种进程间通信的方式

### 常见的IPC类型

IPC机制	数据抽象	参与者	方向
管道	文件接口	两个进程	单向
共享内存	内存接口	多进程	单向/双向
消息队列	消息接口	多进程	单向/双向
信号	信号接口	多进程	单向
套接字	文件接口	两个进程	单向/双向

消息队列：可以做优先级filter之类的新功能

### IPC的接口类型

- 已有接口
  1. 内存接口：共享内存
  2. 文件接口：管道(Pipe), 套接字(Socket)
- 新的接口  
消息接口、信号接口等

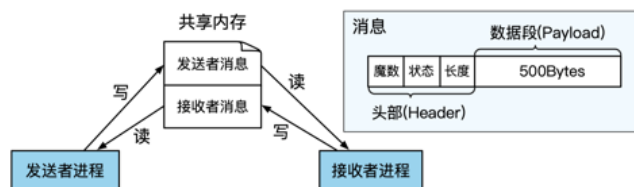
## 2. 简单IPC的设计与实现

## 消息接口

- 最基本的消息接口
  1. 发送: Send(msg)
  2. 接收: Recv(msg)
- 远程方法调用与返回(RPC)
  1. 远程方法调用: RPC(req\_msg, resp\_msg)
  2. 回复消息: Reply(resp\_msg)

## 简单IPC的两个阶段

### 简单IPC的两个阶段



- **阶段-1: 准备阶段**
  - 建立通信连接, 即进程间的信道
    - 假设内核已经为两个进程映射了一段共享内存
- **阶段-2: 通信阶段**
  - 数据传递
    - "消息"抽象: 通常包含头部(含魔数)和数据内容(500字节)
  - 通信机制
    - 两个消息保存在共享内存中: 发送者消息、接收者消息
    - 发送者和接收者通过**轮询**消息的状态作为通知机制

一般不包含指针

## 简单IPC数据传递的两种方法

- 方法-1: **基于共享内存的数据传递**
  1. 操作系统在通信过程中不干预数据传输
  2. 操作系统仅负责准备阶段的映射

"One-Copy"

- **基于共享内存的优势**
  - 用户态无需切换到内核态即可完成IPC (多核场景下)
  - 完全由用户态程序控制, 定制能力更强
  - 可实现零内存拷贝 (无需内核介入)
- 方法-2: **基于操作系统辅助的数据传递**
  1. 操作系统提供接口 (系统调用): Send, Recv
  2. 通过内核态传递数据, 无需在用户态建立共享内存

- **基于系统调用的优势**

- 抽象更简单，用户态直接调用接口，使用更方便
- 安全性保证更强，发送者在消息被接收时通常无法修改消息
- 多方（多进程）通信时更灵活、更安全

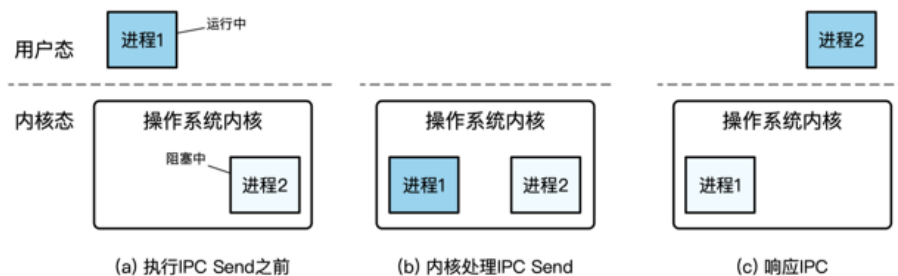
## 简单IPC的通知机制

- **方法-1：基于轮询（消息头部的状态信息）**

- 缺点：大量CPU计算资源的浪费

- **方法-2：基于控制流转移**

- 由内核控制进程的运行状态
- 优点：进程只有在条件满足的情况下才运行，避免CPU浪费



轮询的优点：时延较低 控制流转移的缺点：时延不能保证

## IPC两种通信方式的抽象

- **方法-1：直接通信**

- 通信的一方需要显示地标识另一方，每一方都拥有唯一标识
- 如：Send(P, message), Recv(Q, message)
- 连接的建立是自动完成的（由内核完成）

- **方法-2：间接通信**

- 通信双方通过“信箱”的抽象来完成通信
- 每个信箱有自己唯一的标识符
- 通信双方并不直接知道在与谁通信
- 进程间连接的建立发生在共享一个信箱时

间接通信类似于校长信箱，直接通信相当于直接给校长发邮件

## IPC的权限检查

- **宏内核**
  - 通常基于权限检查的机制实现
  - 如：Linux中与文件的权限检查结合在一起（以后介绍）
- **微内核**
  - 通常基于Capability安全检查机制实现
  - 如seL4将通信连接抽象为内核对象，不同进程对于内核对象的访问权限与操作有Capability来刻画
  - Capability保存在内核中，与进程绑定
  - 进程发起IPC时，内核检查其是否拥有对应的Capability

Capability可以转移，这是基于文件fd的实现方式所无法达到的

## IPC的命名服务

- **命名服务：一个单独的进程**
  - 类似一个全局的看板，协调服务端与客户端之间的信息
  - 服务端可以将自己提供的服务注册到命名服务中
  - 客户端可以通过命名服务进程获取当前可用的服务
- **命名服务的功能：分发权限**
  - 例如：文件系统进程允许命名服务将连接文件系统的权限任意分发，因此所有进程都可以访问全局的文件系统
  - 例如：数据库进程只允许拥有特定证书的客户端连接

## 3. 管道：文件接口的IPC

### Unix管道

管道是Unix等系统中常见的进程间通信机制

- 管道(Pipe)：两个进程间的一根通信管道

```
→ os-textbook git:(master) ls | grep ipc  
ipc.tex
```

## 优缺点

- 优点
  1. 设计和实现简单
- 缺点
  1. 缺少消息的类型，接收者需要对消息内容进行解析
  2. 缓冲区大小预先分配且固定
  3. 只能支持单向通信
  4. 只能支持最多两个进程间通信

## 匿名管道与命名管道

- **传统的管道缺乏名字，只能在有亲缘关系的进程间使用**
  - 也称为“匿名管道”
  - 通常通过fork，在父子进程间传递fd
- **命名管道：具有文件名**
  - 在Linux中也称为fifo，可通过mkfifo()来创建
  - 可以在没有亲缘关系的进程之间实现IPC
  - 允许一个写端，多个读端；或多个写端，一个读端

## 4. 共享内存（内存接口的IPC）

---

- **缺少通知机制**
  - 若轮询检查，则导致CPU资源浪费
  - 若周期性检查，则可能导致较长的等待时延
  - **根本原因**：共享内存的抽象过于底层；缺少OS更多支持
- **TOCTTOU（Time-of-check to Time-of-use）问题**
  - 当接收者直接用共享内存上的数据时，可能存在被发送者恶意篡改的情况（发生在接收者检查完数据之后，使用数据之前）
  - 这可能导致buffer overflow等问题

## 5. 消息传递

---

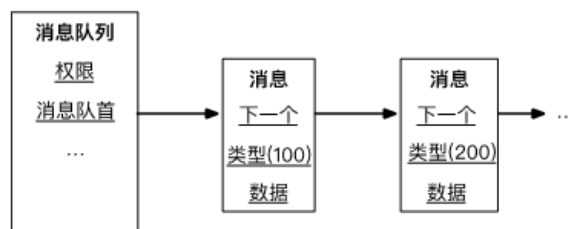
## 设计选择

1. 间接通信方式，信箱为内核中维护的消息队列结构体
2. 有（有限的）缓存
3. 没有超时机制
4. 支持多个（大于2）的参与者进行通信
5. 通常是非阻塞的（不考虑如内核缓冲区满等异常情况）

## 消息队列：带类型的消息传递

- **消息队列：以链表的方式组织消息**
  - 任何有权限的进程都可以访问队列，写入或者读取
  - 支持异步通信（非阻塞）
- **消息的格式：类型 + 数据**
  - 类型：由一个整型表示，具体的意义由用户决定
- **消息队列是间接消息传递方式**
  - 通过共享一个队列来建立连接

```
msgsnd();  
msgrcv();  
msgctl();
```



- **消息队列的组织**
  - 基本遵循FIFO (First-In-First-Out)先进先出原则
  - 消息队列的写入：增加在队列尾部
  - 消息队列的读取：默认从队首获取消息
- **允许按照类型查询: Recv(A, type, message)**
  - 类型为0时返回第一个消息 (FIFO)
  - 类型有值时按照类型查询消息
    - 如type为正数，则返回第一个类型为type的消息

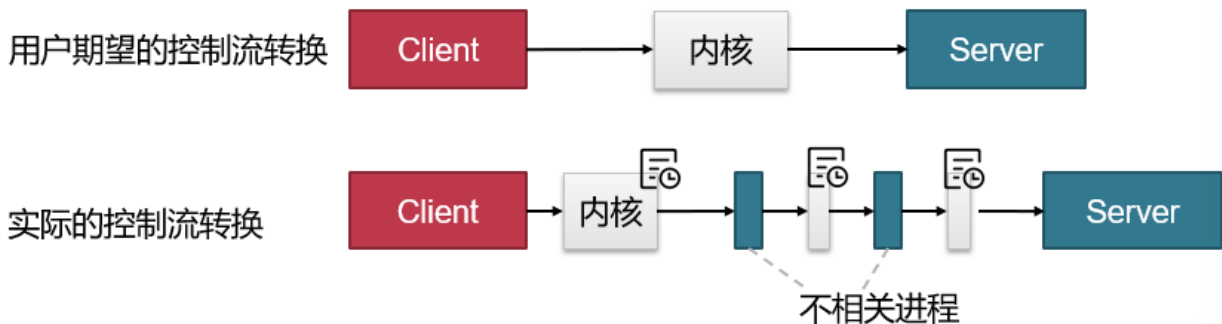
## 6. 轻量级远程方法调用（LRPC）

## 解决两个主要问题

- 控制流转换：Client进程快速通知Server进程
- 数据传输：将栈和寄存器参数传递给Server进程

### 控制流转换：调度导致不确定时延

控制流转换时需要下陷到内核，内核系统为了保证公平，会在内核中根据情况进行调度  
(也就是说下线后不会立马执行目标进程)



解决方案为迁移线程：将Client运行在Server的上下文中

- **为什么需要做控制流转换？**
  - 使用Server的代码和数据
  - 使用Server的权限 (如访问某些系统资源)
- **只切换地址空间、权限表等状态，不做调度和线程切换**



### 数据传输：数据拷贝的性能损失

大部分Unix类系统，经过内核的传输有(至少)两次拷贝

- **数据拷贝:**

- 慢: 拷贝本身的性能就不快 (内存指令)
- 不可扩展: 数据量增大10x, 时延增大10x



解决方案为共享参数栈和寄存器

- **参数栈 (Argument stack, 简称A-stack)**

- 系统内核为每一对LRPC连接预先分配好一个A-stack
- A-stack被同时映射在Client进程和Server进程地址空间
- Client进程只需要将参数准备到A-stack即可
  - 不需要内核额外拷贝

- **执行栈 (Execution stack, 简称E-stack)**

- **共享寄存器**

- 普通的上下文切换: 保存当前寄存器状态 → 恢复切换到进程寄存器状态
- LRPC迁移进程: 直接使用当前的通用寄存器
  - 类似函数调用中用寄存器传递参数

## 轻量远程调用: 通信连接建立

1. Server进程通过内核注册一个服务描述符SD(Service Descriptor)

对应于Server进程内部的一个处理函数

2. 内核为SD预先分配好参数栈

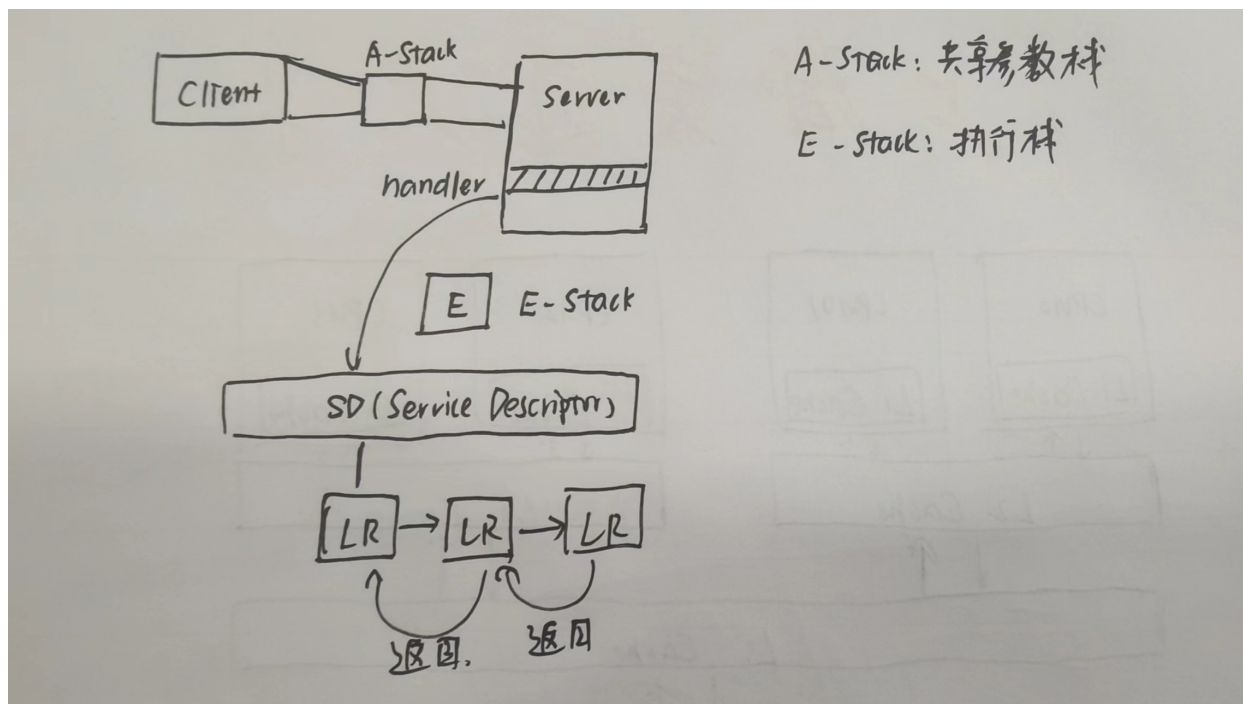
3. 内核为SD预先分配好调用记录(Linkage record)

用于从Server进程处返回(类似栈)

4. 内核将参数栈交给Client进程, 作为一个绑定成功的标志

在通信过程中, 通过检查A-Stack来判断Client是否正确发起通信





- Server进程 ① 注册服务描述符**

```

service_descriptor =
    register_service(handler_func, &A-stack, &E-stack);

void handler_func (A-stack, arg0, ...
    arg7) {
    u64 ret;
    //从寄存器和A-stack中获取参数
    //使用E-stack(运行栈)来处理
    ...
    //返回结果
    ipc_return (ret);
}

```
- Client进程 ② 连接服务并调用**

```

A-stack =
    service_connect(service_name);
/* 准备数据到A-stack */
...
/*arg0—7 为寄存器数据*/
ipc_call(A-stack, arg0, .. arg7);

```
- ③ 用A-stack和寄存器获取参数, 用运行栈来执行逻辑**

## 轻量远程调用：一次调用过程

- 1.内核验证绑定对象的正确性，并找到正确的服务描述符
- 2.内核验证参数栈和连接记录
- 3.检查是否有并发调用 (可能导致A-stack等异常)
- 4.将Client的返回地址和栈指针放到连接记录中
- 5.将连接记录放到线程控制结构体中的栈上 (支持嵌套LRPC调用)
- 6.找到Server进程的E-stack (执行代码所使用的栈)
- 7.将当前线程的栈指针设置为Server进程的运行栈地址
- 8.将地址空间切换到Server进程中

轻量远程调用：讨论

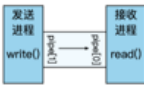





- Q1：为什么要将栈分为参数栈和运行栈？

A1：参数栈是为了共享传递参数，而执行栈是为了执行代码以及处理局部变量使用的
- Q2：LRPC中控制流转换的主要开销是什么？

A2：地址空间的切换（来自硬件限制）是主要的性能开销
- Q3：在不考虑多线程的情况下，共享参数栈是否安全？

A3：安全的。因为是同步IPC，所以在被调用者上下文执行的时候，没有其他人可以去读写A-stack

7. Summary

	IPC机制	数据抽象	参与者	方向	内核实现
	管道	字节流	两个进程	单向	通常以FIFO的缓冲区来管理数据。有匿名管道和命名管道两类主要实现。
	消息队列	消息	多进程	单向 双向	队列的组织方式。通过文件的权限来管理对队列的访问。
	信号量	计数器	多进程	单向 双向	内核维护共享计数器。通过文件的权限来管理对计数器的访问。
	共享内存	内存区间	多进程	单向 双向	内核维护共享的内存区间。通过文件的权限来管理对共享内存的访问。
	信号	事件编号	多进程	单向	为线程/进程维护信号等待队列。通过用户/组等权限来管理信号的操作。
	套接字	数据报文	两个进程	单向 双向	有基于IP/端口和基于文件路径的寻址方式。利用网络栈来管理通信。