

进程间通信

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

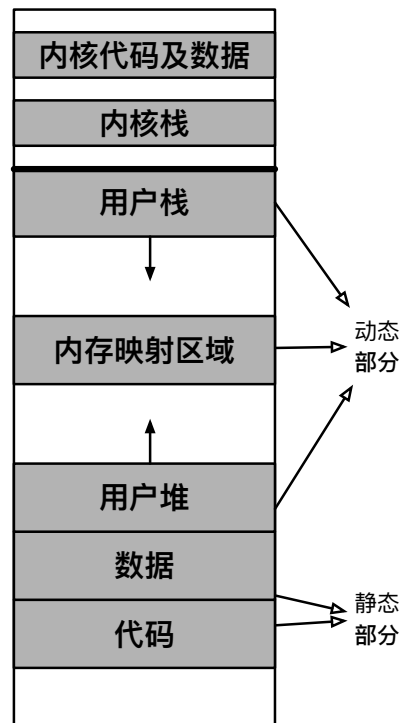
回顾: 进程

- 进程是计算机程序运行时的抽象

- 静态部分: 程序运行需要的代码和数据
- 动态部分: 程序运行期间的状态
(程序计数器、堆、栈.....)

- 进程具有独立的虚拟地址空间

- 每个进程都具有"独占全部内存"的假象
- 内核中同样包含内核栈和内核代码、数据



应用程序的功能非常复杂

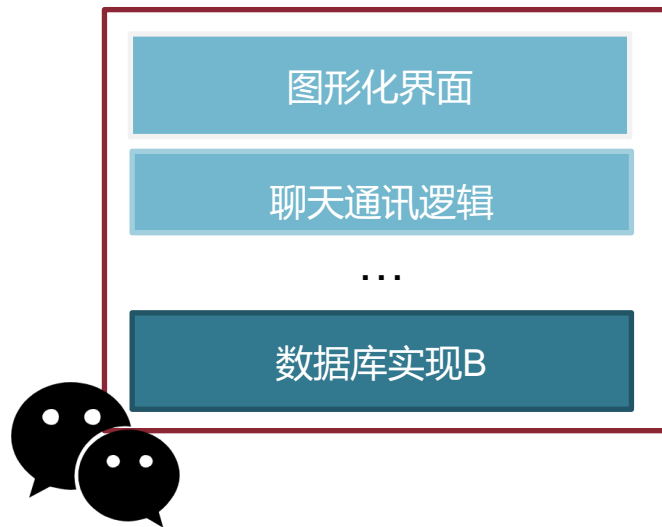
- **独立进程：一个进程就是一个应用**
 - 不会去影响其他进程的执行，也不会被其他进程影响

单个应用的功能非常复杂



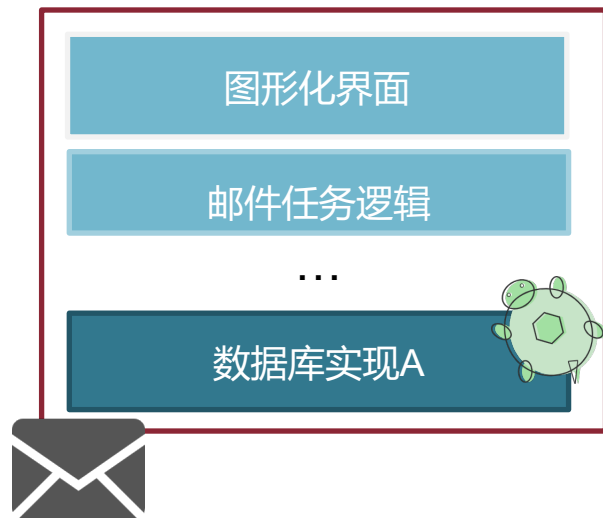
独立进程的问题1: 大量重复实现

- 聊天软件和邮件软件都依赖数据库
- 各自实现一份在自己的进程中



独立进程的问题2: 低效实现

- 聊天软件的数据库实现经过精心的优化
- 邮件软件团队的开发重心在其他组件上
 - 借用低效的某数据库开源实现



独立进程的问题3: 没有信息共享

- 邮件和聊天软件都需要监控系统资源信息
- 没有信息共享
 - 即使邮件软件已经完成了计算，聊天软件也要重新计算一遍

如果进程可以协作

- **协作进程**

- 和独立进程相反，可以影响其他进程执行或者被影响

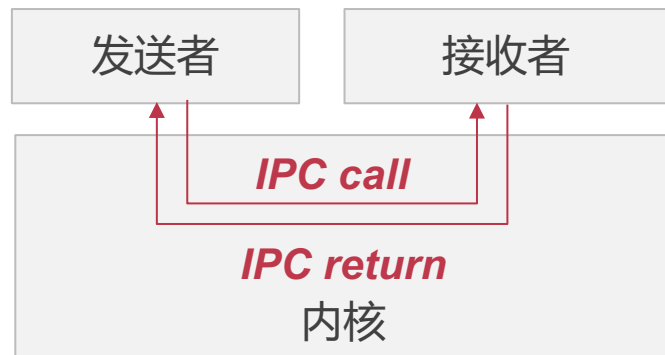
- **好处**

- 模块化: 数据库单独在一个进程中，可以被复用
- 加速计算: 不同进程专注于特定的计算任务，性能更好
- 信息共享: 直接共享已经计算好的数据，避免重复计算



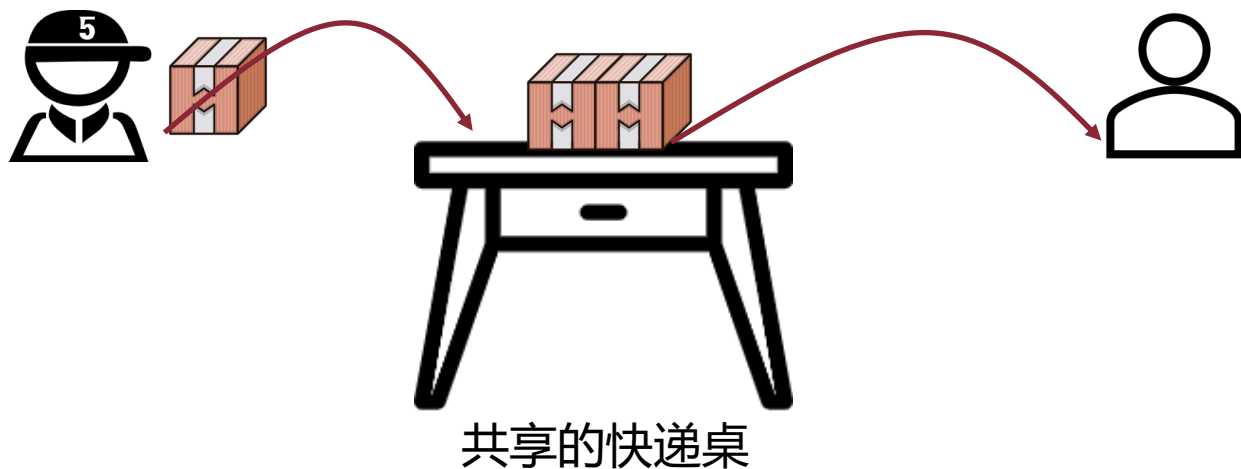
进程间通信 (Inter-process Communication, IPC)

- 进程协作的达成依赖于进程间通信
- 进程间通信: 两个(或多个)不同的进程, 通过内核或其他共享资源进行通信, 来传递控制信息或数据
 - 交互的双方: 发送者/接收者、客户端/服务端、调用者/被调用者
 - 通信的内容一般叫做“消息”



共享内存: 共享一块区域

- 小区门口的桌子上允许临时放置快递
- 快递员在桌上放置快递，小明从桌上取快递



共享内存

- 系统内核为两个进程映射共同的内存区域
- 挑战: 做好同步
 - 发送者不能覆盖掉未读取的数据
 - 接收者不能读取别的数据

共享内存

- 基础实现: 共享区域

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```

```
    } item;
```

```
    item buffer[BUFFER_SIZE];
```

```
    int in = 0;
```

```
    int out = 0;
```

共享数据区域，容量为10

共享状态

共享内存

- 基础实现: 发送者

```
while (new_package) {  
    /* Produce an item */
```

当没有空间时，发送者盲目等待

```
    while (((in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* do nothing -- no free buffers */
```

```
    buffer[in] = item;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
}
```

发送者放置消息

共享内存

- 基础实现: 接收者

当没有新消息时，接收者盲目等待

```
while (wait_package) {
```

```
    while (in == out)
```

```
        ; // do nothing -- nothing to consume
```

```
    // remove an item from the buffer
```

```
    item = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
    return item;
```

```
}
```

接收者获取消息

共享内存的问题

- 轮询导致资源浪费
- 固定一个检查时间，时延长

消息传递

- 消息系统

- 通过中间层(如内核)保证通信时延，仍可以利用共享内存传递数据



好处： 1) 低时延 (消息立即转发)
2) 不浪费计算资源

消息传递

- **基本操作:**
 - 发送消息 *Send(message)*
 - 接收消息 *Recv(message)*
- **如果两个进程 *P* 和 *Q* 希望通过消息传递进行通信，需要:**
 - 建立一个通信连接
 - 通过 *Send/Recv* 接口进行消息传递

直接通信

- **直接通信**

- 进程拥有一个唯一标识
- `Send(P, message)`: 给P进程发送一个消息
- `Recv(Q, message)`: 从Q进程接收一个消息

- **直接通信下的连接**

- 连接的建立是自动的 (通过标识)
- 一个连接唯一地对应一对进程
- 一对进程之间也只会存在一个连接
- 连接可以是单向的，但是在大部分情况下是双向的

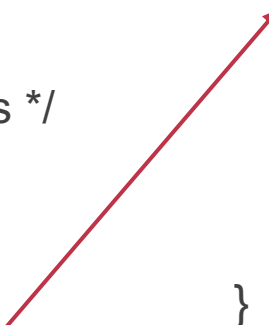
直接通信

- 发送者

```
while (new_package) {  
    /* Produce an item */  
    while (( (in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(XiaoMing, "Package");  
}
```

- 接收者

```
while (wait_package) {  
    Recv(Expressman, Msg);  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    return item;  
}
```



Recv会阻塞，直到Send发送消息过来

快递员有好多苦恼 (1)

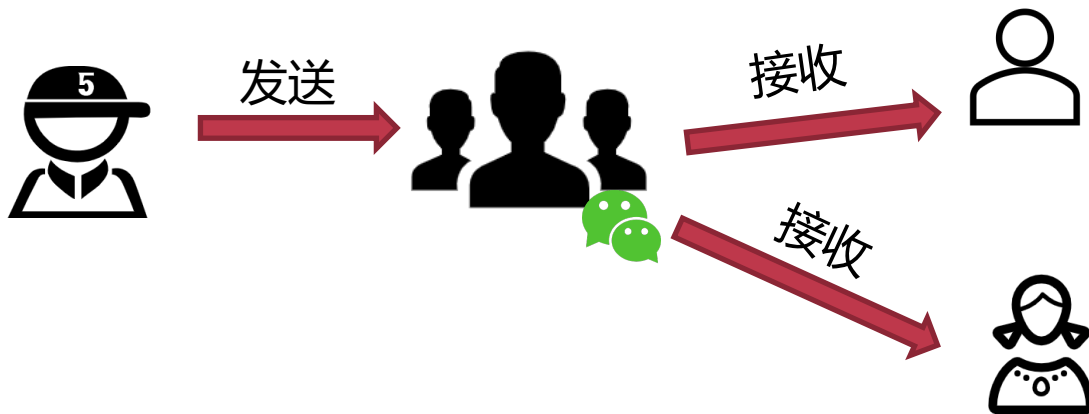


小明经常不接听电话，怎么办？

- 快递员执行Send的时候，小明还没有Recv
- 快递员知道小明妈妈经常在家，希望建立一个聊天群，在群里发布快递信息
- 小明不接听时可以拜托妈妈下来拿快递

间接通信：用聊天群发布快递信息

- 消息的发送和接收需要经过一个“信箱”
 - 聊天群 (所有在群内的人都可以接收消息)
 - 每个“信箱”有自己唯一的标识符 (这里的群号)
 - 发送者往“信箱”发送消息，接收者从“信箱”读取消息



间接通信

- **间接通信下的连接**
 - 进程间连接的建立发生在共享一个信箱时
 - 每对进程可以有多个连接 (共享多个信箱)
 - 连接同样可以是单向或双向的
- **间接进程间通信的操作**
 - 创建一个新的信箱
 - 通过信箱发送和接收消息
 - 销毁一个信箱
- **原语**
 - Send(**M**, message): 给信箱M发送一个消息
 - Recv(**M**, message): 从信箱M接收一个消息

间接通信: 用聊天群发布快递信息

- 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while (( (in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(Mailbox, "Package");  
}
```

- 接收者(小明)

Recv(Mailbox, Msg);

- 接收者(小明妈妈)

Recv(Mailbox, Msg);

小明或者妈妈任何一个人只要上聊天群，就能看到快递信息

快递员有好多苦恼 (2)



快递信息发布到群里，经常是小明和妈妈一起下来了。我都被投诉好几次了，这可怎么办好？

- 怎么解决“信箱”共享带来的多接收者的问题呢？

信箱共享的挑战

- **信箱的共享**

- 进程 P_1 、 P_2 和 P_3 共享一个信箱M
- P_1 负责发送消息， P_2 、 P_3 负责接收消息
- 当一个消息发出的时候，谁会接收到最新的消息呢？

- **可能的解决方案**

- 让一个连接(信箱)只能被最多两个进程共享，避免该问题
- 同一时间，只允许最多一个进程在执行接收信息的操作
- 让消息系统任意选择一个接收者 (需要通知发送者谁是最终接收者)

快递员有好多苦恼 (3)



小明和妈妈回复消息很慢，如果我发完消息等待回复可能要等很久；如果我放下快递直接走的话，事后又可能被投诉，怎么办呢？

- 快递员想要弄清楚自己是等待消息被确认呢(阻塞)，还是发送完消息就赶去送下一个快递呢(非阻塞)？

消息传递的同步与异步

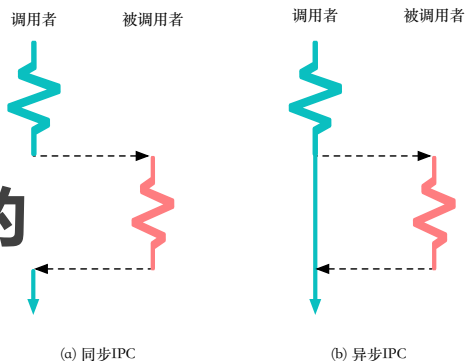
- 消息的传递可以是阻塞的，也可以是非阻塞的

- 阻塞通常被认为是同步通信

- 阻塞的发送/接收: 发送者/接收者一直处于阻塞状态，直到消息发出/到来
- 同步通信通常有着更低时延和易用的编程模型 (不会被投诉)

- 非阻塞通常被认为是异步通信

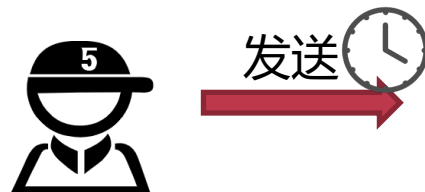
- 发送者/接收者不等待操作结果，直接返回
- 异步通信的带宽一般更高 (快递员可以送更多的快递)



超时机制

- 为了好评，快递员选择：

- 尽可能等待 (同步的通信)
- 但是一旦超过一个值 (如15分钟)，就先带走快递，等下再配送



- 超时机制的引入

- Send(A, message, **Time-out**)
 - 超过Time-out限定的时间就返回错误信息
- 两个特殊的超时选项: ① 一直等待 (阻塞) ; ② 不等待 (非阻塞)
- 避免由通信造成的拒接服务攻击等

同步通信和超时机制

- 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER_SIZE)  
           == out)  
          ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    if (Send(Mailbox, "Package",  
        "15min") == error)  
        goto retry;  
}
```

- 接收者(小明)

Recv(Mailbox, Msg, Time-out);

- 接收者(小明妈妈)

Recv(Mailbox, Msg, Time-out);



快递员决定等待最多15分钟，一旦超时就放弃这一次派送

快递员有好多苦恼 (4)



新冠疫情缓解后，为了避免快递丢失，小区保安不允许快递员将快递放在快递桌上。快递员不能放下快递就走了。

- 有手机后，快递桌的作用有什么变化吗？

通信连接的缓冲: 快递桌的作用

- **缓冲:** 通信连接可以选择保留住还没有处理的消息
- **常见的三种设计**
 - **零容量:** 通信连接本身不缓冲消息, 发送者只能阻塞等待接收者接收消息 (保安不提供快递桌)
 - **有限容量:** 连接可以缓冲最多N个消息, 当缓冲区满之后发送者只能阻塞等待 (快递只能放在桌上, 但是空间有限)
 - **无限容量:** 连接可以缓冲系统资源允许下的任意数量的消息, 发送者几乎不需要等待 (快递可以放在门口任何位置)

UNIX IPC

UNIX经典IPC

Unix 管道

- 管道是Unix等宏内核系统中非常重要的进程间通信机制
- 管道(Pipe): 两个进程间的一根通信通道
 - 一端向里投递，另一端接收
 - 管道是间接消息传递方式，通过共享一个管道来建立连接
- 例子: 我们常见的命令 `ls | grep`

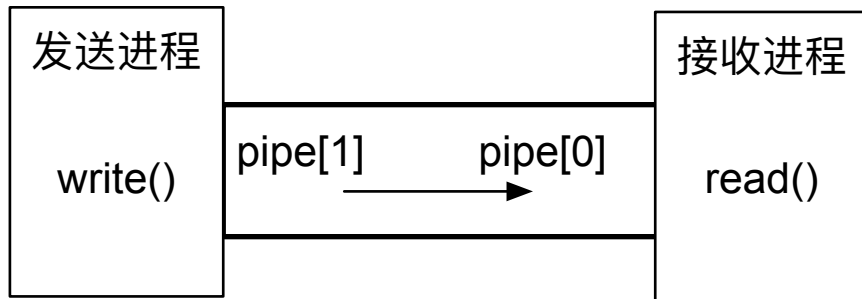
```
→ os-textbook git:(master) ls | grep ipc  
ipc.tex
```

Unix 管道

```
int fd[2];  
pipe(fd);  
fd[0]; // read side  
fd[1]; // write side
```

- 管道的特点:

- 单向通信，当缓冲区满时阻塞
- 一个管道有且只能有两个端口: 一个负责输入 (发送数据)，一个负责输出 (接收数据)
- 数据不带类型，即字节流
- 基于Unix的文件描述符使用



Unix 管道

- Xv6 (Unix V6)为例: 管道数据结构

连接缓冲区域, 定长

一个连接对应
最多两个进程

```
struct pipe {  
    struct spinlock lock;  
    char data[PIPESIZE];  
    uint nread;           // number of bytes read  
    uint nwrite;          // number of bytes written  
    int readopen;         // read fd is still open  
    int writeopen;        // write fd is still open  
};
```

Unix 管道

• Xv6为例: 管道写操作

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
    int i;

    acquire(&p->lock);
    for(i = 0; i < n; i++){
        while(p->nwrite == p->nread + PIPESIZE) { //
            ↪ pipewrite-full
            if(p->readopen == 0 || proc->killed){
                release(&p->lock);
                return -1;
            }
            wakeup(&p->nread);
            sleep(&p->nwrite, &p->lock); // pipewrite-sleep
        }
        p->data[p->nwrite++ % PIPESIZE] = addr[i];
    }
    wakeup(&p->nread); // pipewrite-wakeup1
    release(&p->lock);
    return n;
}
```

检查缓冲区域是否满

如果读者不存在，
那么返回错误信息

尝试唤醒读者去读消息，
并将自己阻塞住

往缓冲区域上放置消息

Unix 管道

- Xv6为例: 管道读操作

```
int
piperead(struct pipe *p, char *addr, int n)
{
    int i;

    acquire(&p->lock);
    while(p->nread == p->nwrite && p->writeopen){ //
        ↪ pipe-empty
        if(proc->killed){
            release(&p->lock);
            return -1;
        }
        sleep(&p->nread, &p->lock); // piperead-sleep
    }
    for(i = 0; i < n; i++){ // piperead-copy
        if(p->nread == p->nwrite)
            break;
        addr[i] = p->data[p->nread++ % PIPESIZE];
    }
    wakeup(&p->nwrite); // piperead-wakeup
    release(&p->lock);
    return i;
}
```

检查是否有消息

阻塞等待消息到来

读取消息

扩展: Sleep/Wakeup通信机制

- Xv6管道实现中依赖于sleep和wakeup两个接口
- Xv6中的sleep和wakeup是经典的进程间等待(wait)和通知(notify)的机制
- 信道(Channel)是等待和通知的媒介
- 一个进程可以通过sleep接口将自己等待在一个信道上
- 另外一个进程可以通过wakeup将等待在某个信道上的进程唤醒

扩展: Sleep/Wakeup通信机制

- Xv6中的sleep实现

- 进程结构体(curproc)中存在一项chan的一项

将chan设置为sleep
等待的channel指针

```
void sleep(void *chan, struct lock *lk):  
    //critical_section_begin
```

```
    curproc->chan = chan  
    curproc->state = SLEEPING  
    release(lk);  
    sched()  
    acquire(lk);
```

//curproc 为当前进程结构

//调度到其他进程

```
    //critical_section_end
```

把当前的进程状态设置为
阻塞，然后调度到其他进程，注意lock的释放和恢复

扩展: Sleep/Wakeup通信机制

- Xv6中的wakeup实现

- 查找所有的目前等待在chan上的进程，并唤醒他们 (设置为可执行)

```
void wakeup(void *chan):  
    //critical_section_begin
```

```
    foreach p in ptable[]:    //ptable 中存放所有进程结构  
        if p->chan == chan and p->state == SLEEPING:  
            p->state = RUNNABLE
```

```
    //critical_section_end
```

遍历所有的进程结构，如果chan
和参数一致，设为可运行

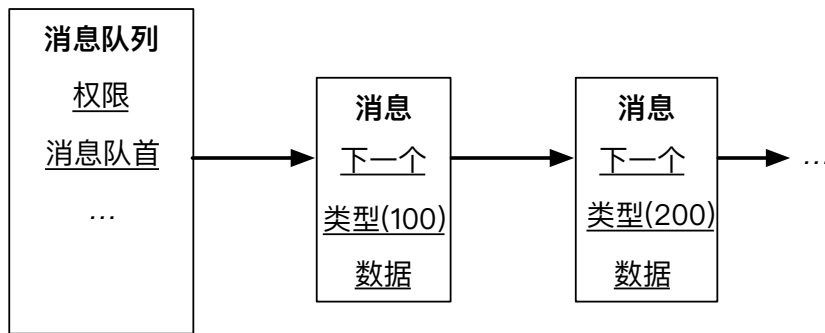
管道的优点与问题

- **优点: 设计和实现简单**
 - 针对简单通信场景十分有效
- **问题:**
 - 缺少消息的类型，接收者需要对消息内容进行解析
 - 缓冲区大小预先分配且固定
 - 只能支持单向通信（为什么？）
 - 只能支持最多两个进程间通信

消息队列：带类型的消息传递

```
Ftok();  
Msgget();  
Msgsnd();  
Msgrcv();  
Msgctl();
```

- **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



消息队列的例子

发送者

```
key = ftok("./msgque", 11);  
msgid = msgget(key, 0666 |  
IPC_CREAT);  
message.mesg_type = 1;  
msgsnd(msgid, &message,  
sizeof(message), 0);
```

接收者

```
key = ftok("./msgque", 11);  
msgid = msgget(key, 0666 |  
IPC_CREAT);  
msgrcv(msgid, &message,  
sizeof(message), 1, 0);  
msgctl(msgid, IPC_RMID,  
NULL);
```

消息队列：带类型的消息传递

- 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

消息队列 VS. 管道

- **缓存区设计:**

- 消息队列: 链表的组织方式, 动态分配资源, 可以设置很大的上限
- 管道: 固定的缓冲区间, 分配过大资源容易造成浪费

- **消息格式:**

- 消息队列: 带类型的数据
- 管道: 数据 (字节流)

- **连接上的通信进程:**

- 消息队列: 可以有多个发送者和接收者
- 管道: 两个端口, 最多对应两个进程

- **消息的管理:**

- 消息队列: FIFO + 基于类型的查询
- 管道: FIFO

消息队列更加灵活易用,
但是实现也更加复杂

Lightweight Remote Procedure Call (LRPC)

轻量级远程方法调用 (LRPC)

Unix进程间通信机制通常十分重量级

- **现有机制**

- 管道，信号，域套接字，共享内存，信号量，消息队列，等等

- **传统宏内核系统中的通信机制通常会结合：**

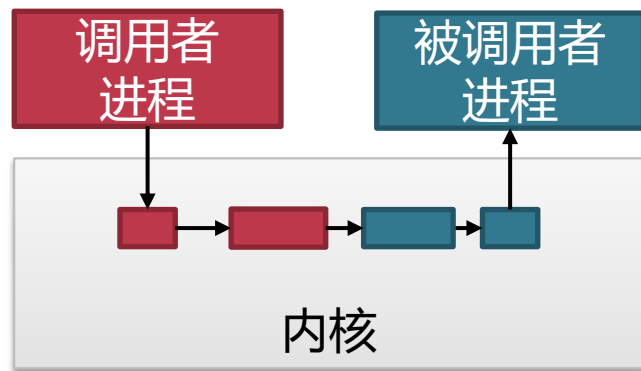
- 通知: 告诉目标进程事件的发生
- 调度: 修改进程的运行状态以及系统的调度队列
- 数据的传输: 传输一个消息的数据过去

- **缺少一个轻量的远程调用机制**

- 客户端进程切换到服务端进程，执行特定的函数 (Handler)
- 参数的传递和结果的返回

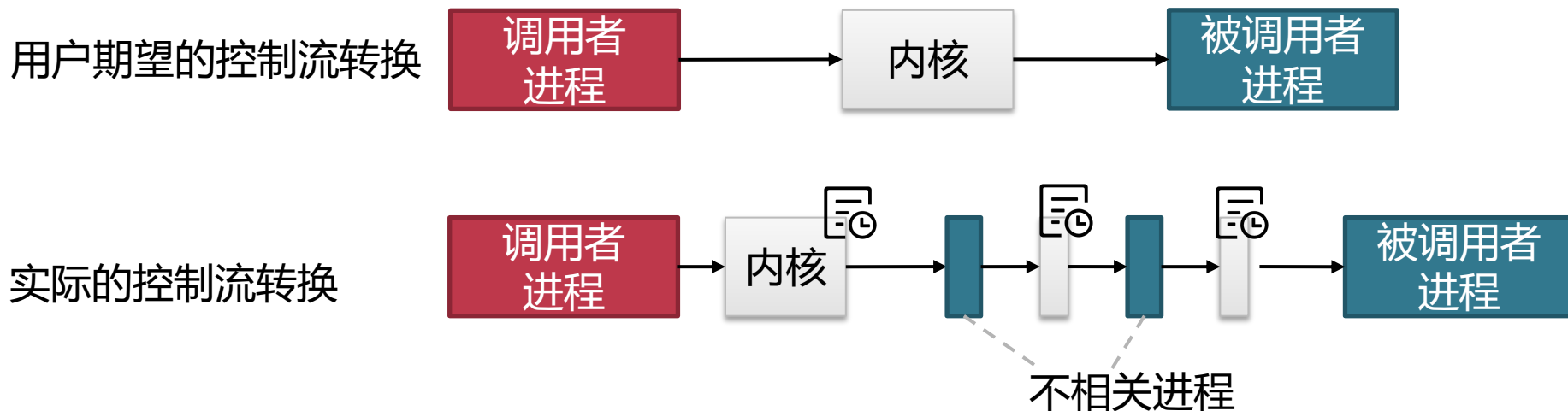
轻量级远程调用 (LRPC)

- Lightweight Remote Procedure Call (LRPC)
- 解决两个主要问题
 - 控制流转换: 调用者进程快速通知被调用者进程
 - 数据传输: 将栈和寄存器参数传递给被调用者进程



控制流转换：调度导致不确定时延

- 控制流转换需要下陷到内核
- 内核系统为了保证公平等，会在内核中根据情况进行调度
 - 调用者和被调用者之间可能会执行多个不相关进程



迁移线程：将调用者运行在被调用上下文

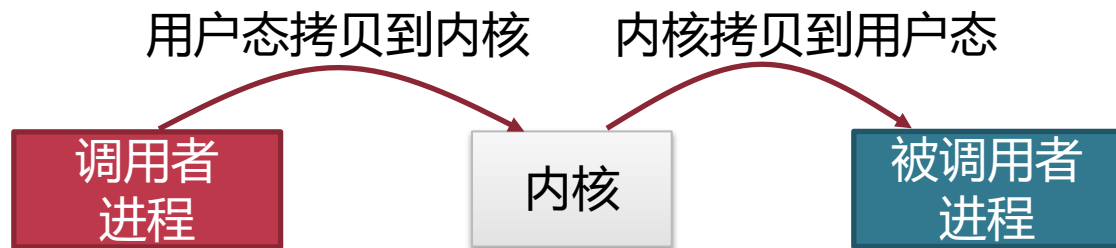
- 为什么需要做控制流转换？
 - 使用被调用者的代码和数据
 - 使用被调用者的权限 (如访问某些系统资源)
- 只切换地址空间、权限表等状态，不做调度和真正的线程切换

调用者进程获得了被调用者的上下文



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
 - 调用者→内核→被调用者
- 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



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被同时映射在调用者进程和被调用者进程地址空间
 - 调用者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 → 恢复切换到的进程寄存器状态
 - 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

- **被调用者进程通过内核注册一个服务描述符**
 - 对应被调用者进程内部的一个处理函数(Handler)
- **内核为服务描述符预先分配好参数栈**
- **内核为服务描述符分配好调用记录 (Linkage record)**
 - 调用记录被用作从被调用者进程处返回
- **内核将参数栈交给客户端进程，作为一个绑定成功的标志**
 - 在通信过程中，通过检查A-stack来判断调用者是否正确发起通信

轻量远程调用：基本用户接口

• 服务端进程

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);  
}
```

• 客户端进程

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. 将调用者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. 找到被调用者进程的*E-stack* (执行代码所使用的栈)
7. 将当前线程的栈指针设置为被调用者进程的运行栈地址
8. 将地址空间切换到被调用者进程中
9. 执行被调用者地址空间中的处理函数

轻量远程调用：通信调用实现

- 内核

```
ipc_call(A-stack, arg0, .. arg7):  
    verify_binding(A-stack); //验证A-stack正确性  
    service_descriptor = get_desc_from_A(A-stack);  
    /*其他安全检查: 是否存在并发调用? */  
    ...  
    save_ctx_to_linkage_record(); //保存调用信息到连接记录上  
    save_linkage_record();  
    ...  
    /* 切换运行状态 */  
    switch_PT(); //修改页表  
    switch_cap_table(); //修改权限表  
    switch_sp(); //修改栈地址  
    ....  
    //返回到用户态(服务端进程), 不修改参数寄存器  
    ctx_restore_with_args (ret);
```


轻量远程调用：讨论

- 为什么需要将栈分成参数栈和运行栈？
- LRPC中控制流转换的主要开销来自哪？
- 不考虑多线程的情况下，共享参数栈安全吗？

ChCore IPC

CHCORE进程间通信

ChCore进程间通信

- **通信进程直接切换**
 - 启发自LRPC和L4直接切换技术
- **同步的通信**
- **通过共享内存传输大数据**
- **基于Capability的权限控制**
 - 类似Unix文件描述符的权限机制，Capability表示一个线程/进程对于系统资源的具体权限

建立通信连接

1. 服务端进程在内核中注册服务
2. 客户端进程向内核申请连接目标服务端进程的服务
 - 可选: 设置共享内存
3. 内核将客户端请求转发给服务端
4. 服务端告诉内核同意连接 (或拒绝)
 - 可选: 设置共享内存
5. 内核建立连接, 并把连接的Capability返回给客户端 (或返回拒绝)

通信过程 (发起通信)

1. 客户端进程通过连接的Capability发起进程间通信请求
2. 内核检查权限，若通过则继续步骤3，否则返回错误
3. 内核直接切换到服务端进程执行 (不经过调度器)
 - 将通信请求的参数设置给服务端进程的寄存器中
4. 服务端处理完毕后，通过与步骤3相反的过程将返回值传回客户端

实现：用户态通信调用

```
void ipc_dispatcher(ipc_msg_t *ipc_msg) {  
    u64 ret;  
    char* data = ipc_get_msg_data(ipc_msg);  
  
    // handling ipc accordingly...  
  
    ipc_return(ret);  
}  
  
void server() {  
    // ...  
  
    ipc_register_server(ipc_dispatcher);  
  
    // ...  
}
```

```
void client() {  
    int server_thread_gap;  
    ipc_struct_t client_ipc_struct;  
    ipc_msg_t *ipc_msg;  
  
    // get server_thread_gap ①  
  
    ipc_register_client(server_thread_gap, &client_ipc_struct);  
  
    u64 ret = ipc_call(&client_ipc_struct, ipc_msg);  
    // ...  
}
```

实现：通信系统调用

- IPC_call

获取内核的连接对象，检查cap权限

```
u64 sys_ipc_call(u32 conn_cap, ipc_msg_t *ipc_msg)
{
    struct ipc_connection *conn = NULL;
    u64 arg;
    int r;

    conn = obj_get(current_thread->process, conn_cap, TYPE_CONNECTION);

    /* Message Processing */
    // do some thing...

    arg = get_srv_vmaddr(ipc_msg);
    thread_migrate_to_server(conn, arg);

    BUG("This function should never\n");
    return 0;
}
```

在传递之前需要将
ipc_msg在客户进程的
虚拟地址转换到服务
进程中的虚拟地址

将控制流移交给服务进程

实现：通信系统调用

- IPC_return

```
void sys_ipc_return(u64 ret) {  
    struct ipc_connection *conn = current_thread->active_conn;  
    thread_migrate_to_client(conn, ret);  
    BUG("This function should never\n");  
}
```

将控制流移交给客户进程