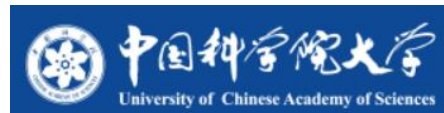




中国科学院软件研究所
Institute of Software, Chinese Academy
of Sciences



进程间通信

郑晨

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
 - <https://ipads.se.sjtu.edu.cn/courses/os/index.shtml>
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。

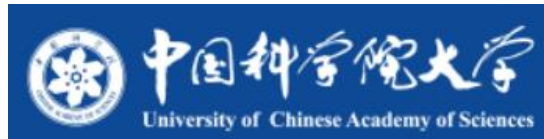


中国科学院软件研究所

Institute of Software, Chinese Academy of Sciences



上海交通大学
SHANGHAI JIAO TONG UNIVERSITY



使用多个进程的应用

- **一些应用程序选择使用不同进程来运行不同模块**
 - 优势-1：功能模块化，避免重复造轮子（如数据库、界面绘制）
 - 优势-2：增强模块间隔离，增强安全保障（敏感数据的隔离）
 - 优势-3：提高应用容错能力，限制故障在模块间的传播
- **然而不同进程拥有不同的内存地址空间**
 - 进程与进程之间无法直接进行通信和交互
 - 需要一种进程间通信的方式
 - **IPC**：Inter-Process Communication

常见IPC的类型

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

IPC的接口类型

- **已有接口**
 - 内存接口：共享内存；文件接口：管道（Pipe）、套接字（Socket）
- **新的接口**
 - 消息接口、信号接口等
- **简单IPC的消息接口**
 - 发送消息：Send(message)
 - 接收消息：Recv(message)
 - 远程方法调用：RPC(req_message, resp_message)
 - 回复消息：Reply(resp_message)

简单IPC的设计与实现

消息接口

- **最基本的消息接口**
 - 发送消息: `Send(message)`
 - 接收消息: `Recv(message)`
- **远程方法调用与返回 (RPC)**
 - 远程方法调用: `RPC(req_message, resp_message)`
 - 回复消息: `Reply(resp_message)`

简单IPC：消息的发送

```
1 // IPC 的发送者
2 int main(void)
3 {
4     Message msg;
5     // chan 表示发送者和消费者之间的一个“通信连接”
6     Channel chan = simple_ipc_channel(...);
7     // 按照语义生成请求消息
8     msg = construct_request(...);
9
10    // 通过通信连接发送一个消息出去
11    Send(chan, &msg);
12    ...
13 }
```

发送者和消费者需要依赖于一个通信连接chan（即channel），作为媒介进行消息传输

简单IPC：消息的接收

```
1 // IPC 的接收者
2 int main(void)
3 {
4     Message msg;
5     // chan 表示发送者和消费者之间的一个“通信连接”
6     Channel chan = simple_ipc_channel(...);
7
8     while (1) {
9         // 等待一个消息的到来，这里会收到 Send 发送的消息
10        Recv(chan, &msg);
11        // 处理消息
12        results = handle_msg(&msg);
13        ...
14    }
15    ...
16 }
```

简单IPC：消息的远程方法调用（发送者）

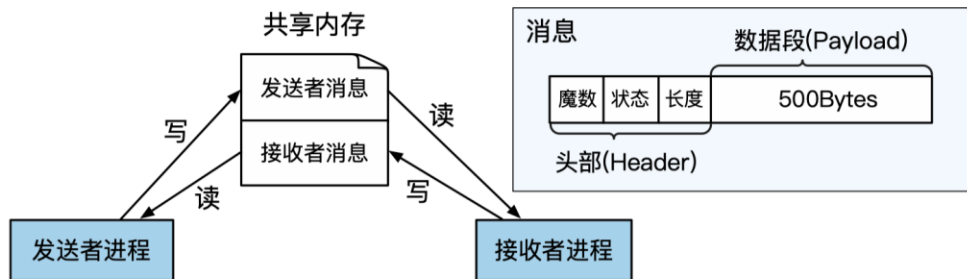
```
1 #include <simple-ipc.h> // 使用后续章节的简单 IPC 设计
2 ...
3
4 // IPC 的发送者
5 int main(void)
6 {
7     Message req_msg, resp_msg;
8     // chan 表示发送者和消费者之间的一个“通信连接”
9     Channel chan = simple_ipc_channel(...);
10    // 按照语义生成请求消息
11    req_msg = construct_request(...);
12
13    // 以 RPC 的方式调用接收者，并阻塞等待一个结果的返回
14    RPC(chan, &req_msg, &resp_msg);
15
16    printf("The response is: %s", msg_to_str(resp_msg));
17    ...
18 }
```

回顾RPC：Remote Procedure Call，远程方法调用

简单IPC：消息的远程方法调用（接收者）

```
1 #include <simple-ipc.h> // 使用后续章节的简单 IPC 设计
2 ...
3
4 // IPC 的接收者
5 int main(void)
6 {
7     Message req_msg, resp_msg;
8     // chan 表示发送者和消费者之间的一个“通信连接”
9     Channel chan = simple_ipc_channel(...);
10
11     while (1) {
12         // 等待一个消息的到来
13         Recv(chan, &req_msg);
14         // 处理消息并构建结果消息
15         resp_msg = handle_msg(&req_msg);
16         Reply(chan, &resp_msg);
17     }
18     ...
19 }
```

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**
 - 操作系统在通信过程中不干预数据传输
 - 操作系统仅负责准备阶段的映射
- **方法-2：基于操作系统辅助的数据传递**
 - 操作系统提供接口（系统调用）： Send、Recv
 - 通过内核态内存来传递数据，无需在用户态建立共享内存

两种数据传递方法的对比

- **基于共享内存的优势**

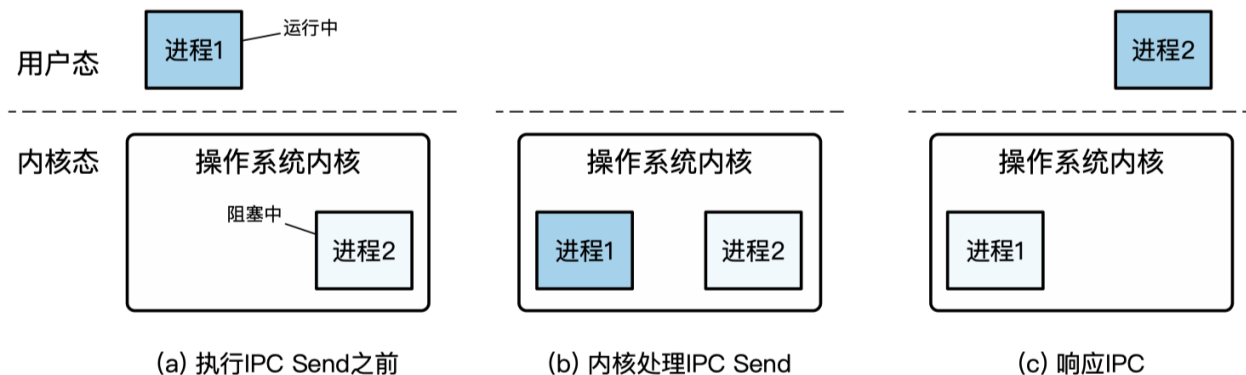
- 用户态无需切换到内核态即可完成IPC（多核场景下）
- 完全由用户态程序控制，定制能力更强
- 可实现零内存拷贝（无需内核介入）

- **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



IPC的方向：单向和双向

- **简单IPC的一次完整通信过程包含两个方向的通信**
 - 发送者传递一个消息（即请求）给接收者
 - 接收者返回一个消息（即结果）给发送者
- **通信的三种可能方向**
 - 仅支持单向通信
 - 支持双向通信（可基于单向通信实现）
 - 单向和双向通信均可（根据配置来选择）

IPC控制流：同步和异步

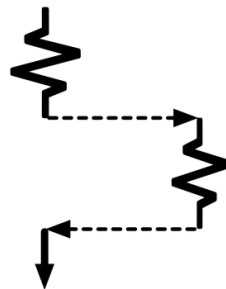
- **同步IPC**

- IPC操作会阻塞进程直到操作完成
- 线性的控制流
- 调用者继续运行时，返回结果已经ready

- **异步IPC**

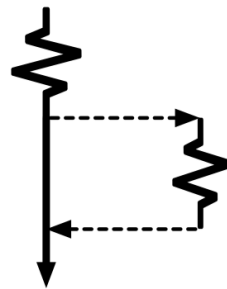
- 进程发起IPC操作后即可返回而不需要等待其完成
- 通过轮询或回调函数（需内核支持）来获取返回结果

调用者 被调用者



(a) 同步IPC

调用者 被调用者



(b) 异步IPC

IPC的超时机制

- **一种新的错误：超时**
 - 传统的函数调用不存在超时问题
 - IPC涉及两个进程，分别有独立的控制流
- **超时可能的原因**
 - 被调用者是恶意的：故意不返回
 - 被调用者不是恶意的：运行时间过长、调度时间过长、请求丢失等
- **超时机制**
 - 应用可自行设置超时的阈值，但如何选择合适的阈值却很难
 - 特殊的超时机制：阻塞、立即返回（要求被调用者处于可立即响应的状态）

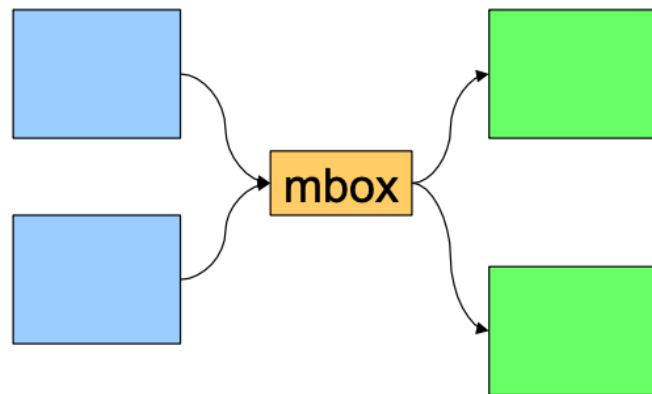
IPC的两种通信连接抽象

- **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



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

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

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`

```
# zhengchen @ zhengchendeMacBook-Pro in ~ [15:46:10]  
$ ls | grep Virtual  
VirtualBox VMs
```

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

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

Stdin, stdout, stderr

- **Unix谱系下，每个进程都有三个已经打开的“文件”**
 - 并非真实的文件，但在unix下，所有对象都可以看作一个文件
- **三个文件：**
 - Stdin: 标准输入流
 - Stdout: 标准输出流
 - Stderr: 标准错误流
- **E.g. printf 就输出到stdout**
- **每个文件都关联了一个整型的文件描述符**
 - 一个索引：该进程打开的文件
- **标准流的文件描述符 (see /usr/include/unistd.h):**
 - stdin: STDIN_FILENO = 0
 - stdout: STDOUT_FILENO = 1
 - stderr: STDERR_FILENO = 2

重定向输出

- **Ls > file.txt 是怎么工作的**
 - Ls:
 - `fprintf(stdout, "%s", filename);`
 - 他是怎么知道将输出写到文件里，而不是标准输出stdout？
- **在unix中，当打开一个新文件，这个文件将获得第一个可用的文件描述符**
- **若关闭stdout，再打开一个文件，该文件会有文件描述符 1**
- **因此，printf会以为这个文件是stdout，并将输出写入该文件**
- **且，不需要更改ls的任何代码！**

示例

- 下面示例会执行ls -ls 并将其输出写到文件/tmp/stuff

Example program fragment (should check for errors)

```
...
pid_t pid = fork();
if (!pid) { // child
    close(1); // close stdout
    FILE *file = fopen("/tmp/stuff", "w"); // open a new file, which gets file descriptor 1
    // exec the "ls -la" command
    char* const arguments[] = {"ls", "-la", NULL};
    execv("ls", arguments);
}
...
}
```

示例

- **前一个示例中，执行顺序如下：**
 - 关闭stdout
 - 打开一个文件，该文件获得描述符 1
- **如果已经打开了该文件，但他获得了其他的文件描述符？**
 - Dup()系统调用：文件描述符duplication
 - Dup()可以为一个已经打开的文件创建另一个文件描述符，并从低开始选择一个为使用的文件描述符号
 - Fileno()函数会返回一个打开文件的描述符
- **因此，新的执行如下：**
 - `FILE *some file = fopen(...);`
 - `close(1);`
 - `dup(fileno(some file));`

示例

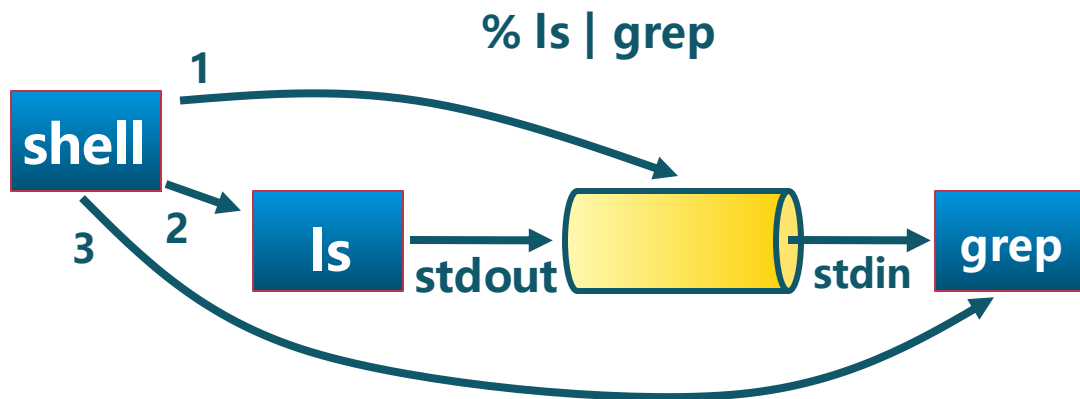
Example program fragment (should check for errors)

```
...  
FILE *file = fopen("/tmp/stuff", "w"); // open a new file  
pid_t pid = fork();  
if (!pid) { // child  
    close(1); // close stdout  
    dup(fileno(file)) // duplicate the file's file descriptor  
    // exec the "ls -la" command  
    char* const arguments[] = {"ls", "-la", NULL};  
    execv("ls", arguments);  
}  
...  
}
```

管道示例

- Shell

- 1. 创建管道
- 2. 为ls创建一个进程, 设置 stdout为管道写端
- 3. 为grep 创建一个进程, 设置 stdin 为管道读端

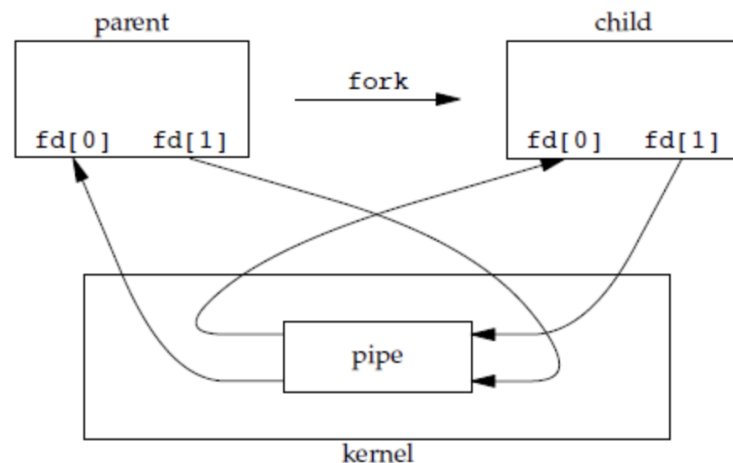


管道相关系统调用

- **读管道:** `read(fd,buffer,nbytes)`
 - C语言中的 `scanf()` 是基于它实现的
- **写管道:** `write(fd,buffer,nbytes)`
 - `printf()` 是基于它实现的
- **创建管道:** `pipe(rgfd)`
 - `rgfd` 是2个文件描述符组成的数组
 - `rgfd[0]` 是读文件描述符
 - `rgfd[1]` 是写文件描述符

Popen(): fork() with a pipe

- **Popen()**
 - 创建一个双向pipe
 - Forks and execs一个子进程 (e.g, "ls -a")
 - 返回管道, 其实是一个文件 (FILE *)
 - 父进程和子进程可以通过pipe通信



- 通过fork, waitpid, pipe, close, open, dup等系统调用来实现

示例

- 下面示例代码会打印出ls -la的输出

Example program fragment (should check for errors)

```
...
// fork/exec a child process and get a pipe to READ from
FILE *pipe = popen("/usr/bin/ls -la", "r");

// Get lines of output from the pipe, which is just a FILE *, until EOF is reached
char buffer[2048];
while (fgets(buffer, 2048, pipe)) {
    fprintf(stderr, "LINE: %s", buffer);
}

// Wait for the child process to terminate
pclose(pipe);
}
```

► 共享内存（内存接口的IPC）

共享内存

- 基础实现: 共享区域

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```

```
} item;
```

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

```
volatile int buffer_write_cnt = 0;
```

```
volatile int buffer_read_cnt = 0;
```

```
volatile int empty_slot = BUFFER_SIZE;
```

```
volatile int filled_slot = 0;
```

共享数据区域, 容量为10

共享状态

基于共享内存的生产者消费者问题实现

- 基础实现: 发送者 (生产者)

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

当没有空间时，发送者盲等

```
    while (empty_slot == 0)  
        ; /* do nothing -- no free buffers */
```

```
    empty_slot --;
```

```
    buffer[buffer_write_cnt] = msg; ----- 发送者放置消息
```

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

```
    filled_slot ++;
```

```
    ...
```

```
}
```

基于共享内存的生产者消费者问题实现

- 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 (filled_slot == 0)
```

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

```
    filled_slot--; // remove an item from the buffer
```

```
    item = buffer[buffer_read_cnt]; ----- 接收者获取消息
```

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

```
    empty_slot++;
```

```
    return item;
```

```
}
```

共享内存的问题

- **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

▶ 消息传递 (MESSAGE PASSING)

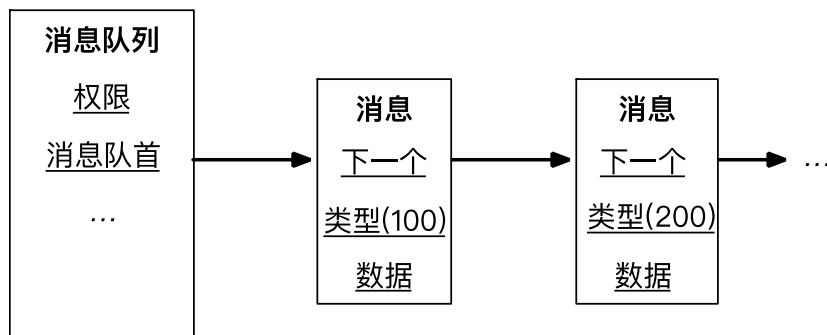
消息队列

- 一种消息传递机制
- 设计选择：
 - 间接通信方式，信箱为内核中维护的消息队列结构体
 - 有（有限的）缓存
 - 没有超时机制
 - 支持多个（大于2）的参与者进行通信
 - 通常是非阻塞的（不考虑如内核缓存区满等异常情况）

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);
```

- Ftok (pathname, proj_id)
 - ❖ 通过一个路径名和整数标识符生产IPC key键值
- msgget(key, flags)
 - ❖ 获取消息队列标识
- msgsnd(QID, buf, size, flags)
 - ❖ 发送消息
- msgrcv(QID, buf, size, type, flags)
 - ❖ 接收消息
- msgctl(...)
 - ❖ 消息队列控制

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)

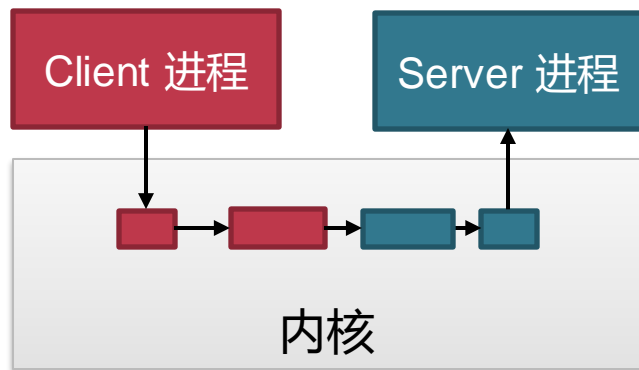
轻量级远程方法调用 (LRPC)

IPC通常会带来较大的性能损失

- **传统的进程间通信机制通常会结合以下机制：**
 - 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
 - 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

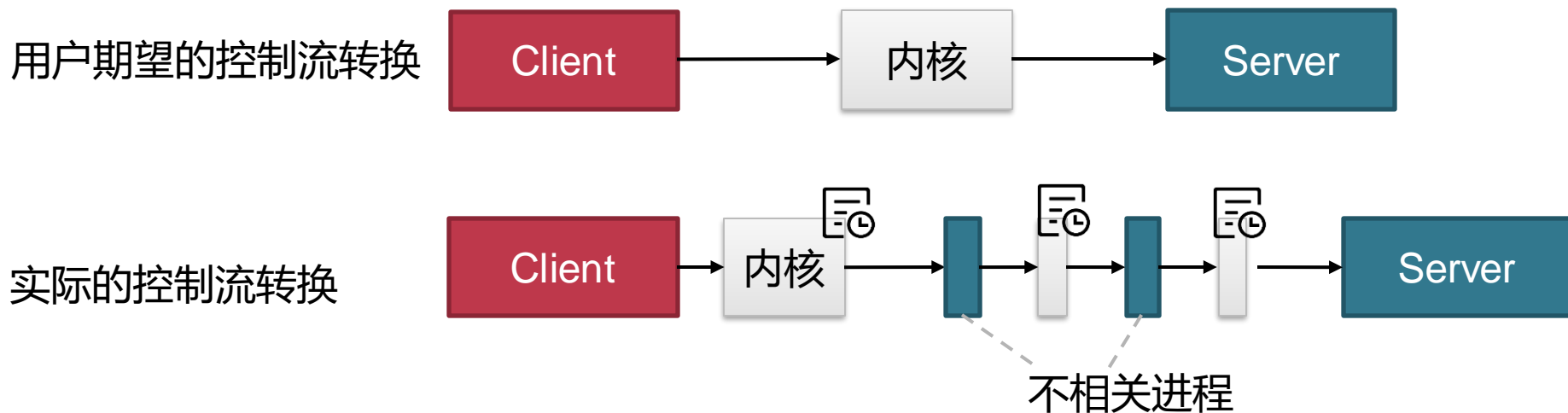
轻量级远程调用 (LRPC)

- Lightweight Remote Procedure Call (LRPC)
- 解决两个主要问题
 - 控制流转换: 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



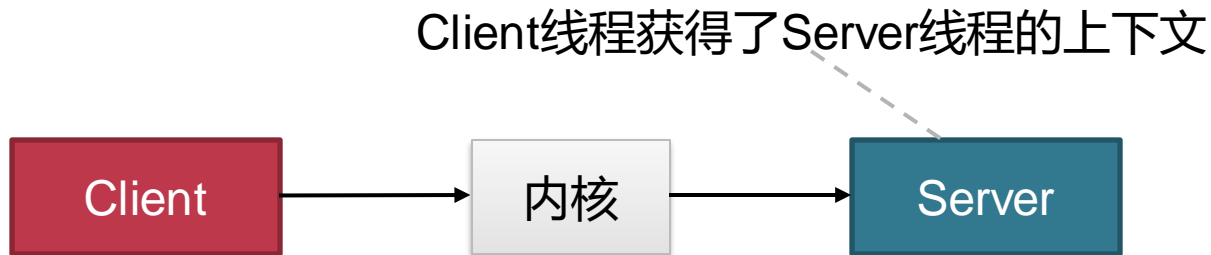
控制流转换：调度导致不确定时延

- 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
 - Client和Server之间可能会执行多个不相关进程



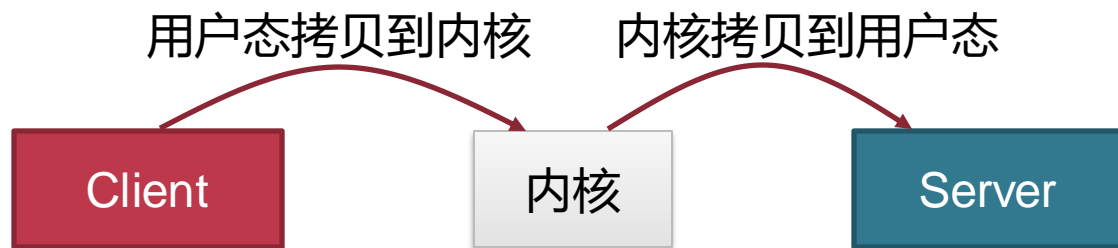
迁移线程: 将Client运行在Server的上下文

- 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
 - Client→内核→ Server
- 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

- **Server进程通过内核注册一个服务描述符**
 - 对应Server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)**
 - 用于从Server进程处返回（类似栈）
- **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

轻量远程调用：基本用户接口

• 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
9. 执行Server地址空间中的处理函数

轻量远程调用：通信调用实现

- 内核中的代码

```
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

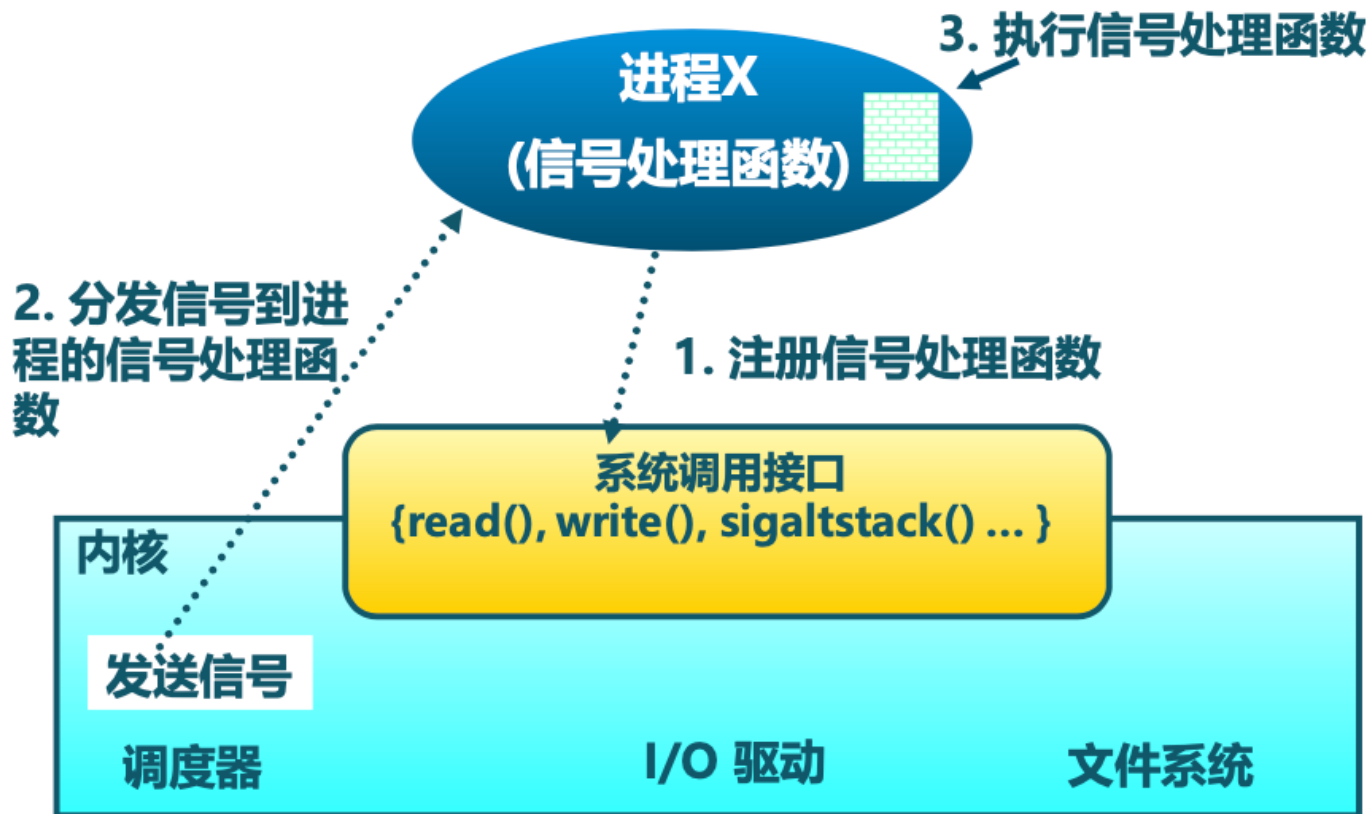
- 为什么需要将栈分成参数栈和运行栈？
- LRPC中控制流转换的主要开销是什么？
- 在不考虑多线程的情况下，共享参数栈是否安全？

▶ LINUX信号机制

信号 (Signal)

- **进程间的软件中断通知和处理机制**
 - 如:SIGKILL, SIGSTOP, SIGCONT等
- **信号的接收处理**
 - 捕获(catch):执行进程指定的信号处理函数被调用
 - 忽略(Ignore):执行操作系统指定的缺省处理
 - 例如:进程终止、进程挂起等
 - 屏蔽(Mask):禁止进程接收和处理信号
 - 可能是暂时的(当处理同样类型的信号)
- **不足**
 - 传送的信息量小, 只有一个信号类型

信号的实现



信号使用示例

```
#include <stdio.h>
#include <signal.h>

main()
{
    signal(SIGINT, sigproc);    /* DEFAULT ACTION: term */
    signal(SIGQUIT, quitproc); /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

信号使用示例

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
    signal(SIGINT, sigproc); /* NOTE some versions of UNIX will reset
                             * signal to default after each call. So
                             * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}

main()
{
    signal(SIGINT, sigproc); /* DEFAULT ACTION: term */
    signal(SIGQUIT, quitproc); /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

信号使用示例

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
    signal(SIGINT, sigproc);    /* NOTE some versions of UNIX will reset
                                * signal to default after each call. So
                                * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}

void quitproc()
{
    printf("ctrl-\\ pressed to quit\n");    /* this is "ctrl" & "\\ " */
    exit(0); /* normal exit status */
}

main()
{
    signal(SIGINT, sigproc);    /* DEFAULT ACTION: term */
    signal(SIGQUIT, quitproc); /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

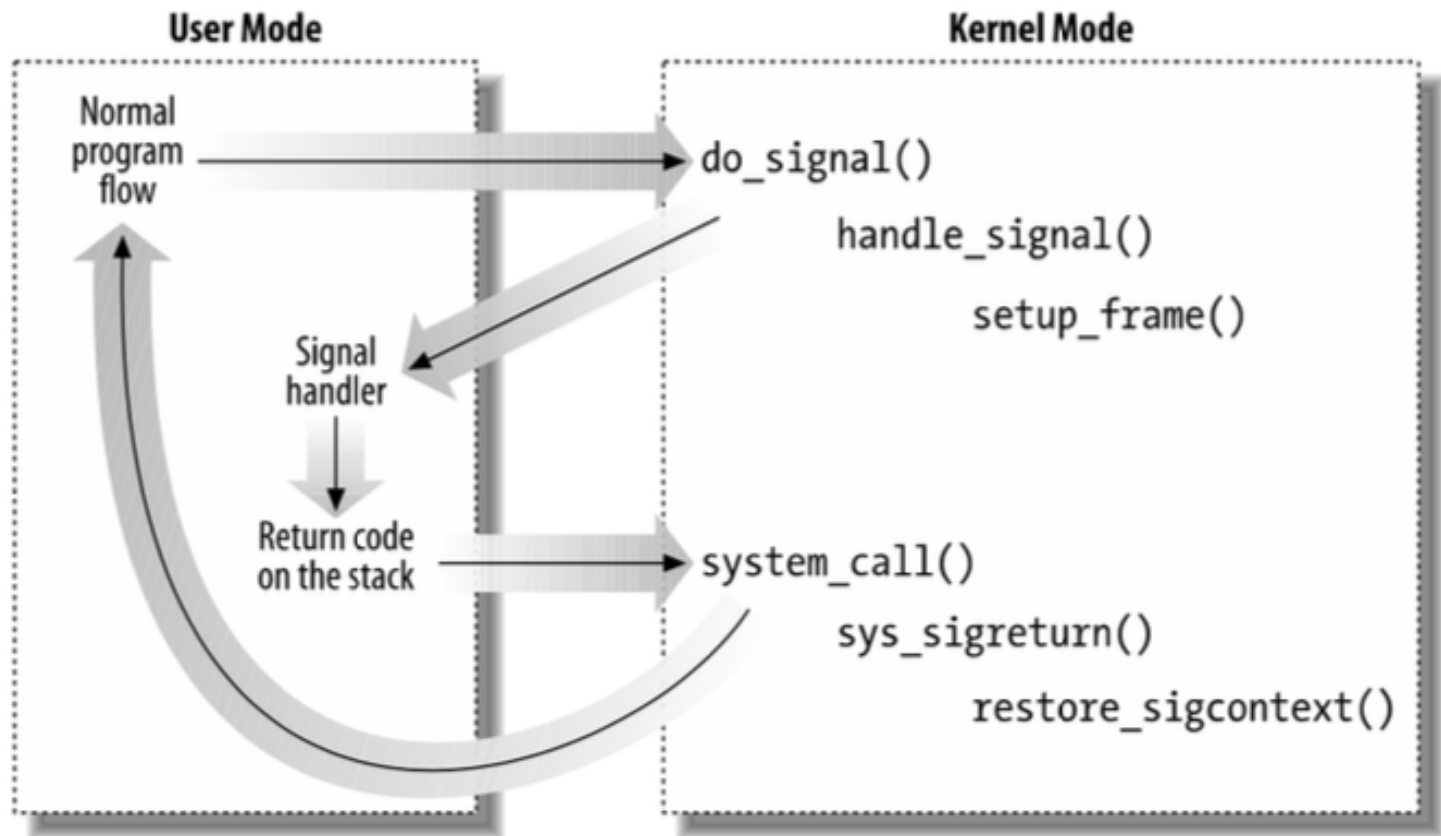
Signal model

- 应用通过signal()、sigaction()注册信号处理程序
- 通过kill()等函数传递信号
 - 或者由内核中硬件异常处理函数触发信号
- Signal传递跳转到signal处理函数
 - Irregular control flow, 类似于中断

语言异常

- **Signal是异常和catch blocks的底层机制**
- **JVM或其他的运行时系统会设置相关singal处理函数**

信号异常控制流程

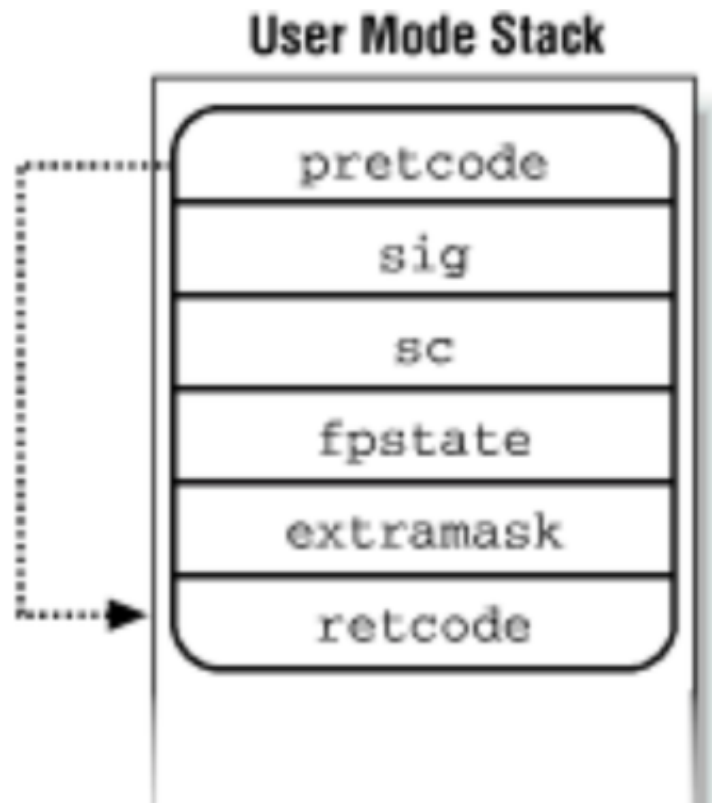


栈操作

- **信号处理函数可以执行在另外一个栈上**
 - 通过sigaltstack()系统调用
- **类似中断处理，内核将寄存器状态推送到栈上**
 - 通过sigreturn()系统调用返回到内核
 - 应用可以修改其私有的栈上寄存器状态

用户态栈段

- Pretcode: 信号处理函数的返回地址
- Sig: signal number
- Sc: 用户态进程的硬件上下文状态
- Fpstate: 用户态进程的浮点寄存器
- Extramask: 阻塞的实时信号
- Retcode: 8字节的代码, 触发sigreturn()系统调用



Signal trampoline & sigreturn()系统调用

- 内核通过一小段汇编代码来清理信号处理函数现场
 - Signal trampoline: sigreturn()
 - Sigreturn()将重制信号处理函数的所有操作
 - 改变进程的singal mask, 切换信号栈
 - 切换栈空间, 恢复进程上下文
 - Sigreturn()从不返回
 - Signal trampoline 代码一半存在于vDSO或者C库中
 - vDSO (virtual dynamic shared object) :内核自动映射到每个用户态应该地址空间的共享库

多线程异步信号处理

- **第一个可用线程获得信号**
 - 大部分处理例程跑在线程的栈上
 - 一个处理例程可以跑在指定的栈上
 - 内核态的线程不执行该处理例程，直到其返回用户态

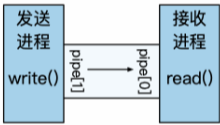
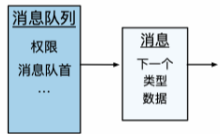
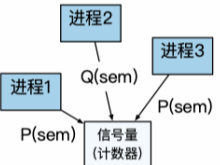
信号处理例程

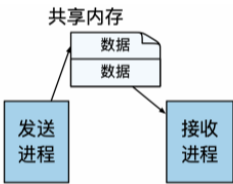
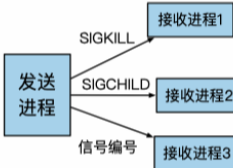
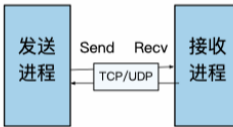
- **信号默认处理例程**
 - Ignore, kill, suspend, continue, dump core
 - 在内核执行
- **通过signal/sigaction注册新的信号处理例程，覆盖默认**
- **Sigkill, sigstop等不能被覆盖**

信号传递

- **发送信号==在任务中标示一个等待的信号**
 - 如果有TASK_INTERRUPTIBLE标示, 被阻塞
- **当中断或syscall返回时, 检查等待的信号, 如果有等待的信号, 发送。**
- **信号嵌套**
 - Sigaction, 检查哪一个signal被阻塞, 或在sigreturn时被传递
 - 类似禁止硬件中断
 - 在信号处理例程中的阻塞系统调用, 只有在小心的使用sigaction时才可使用

IPC小结

	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。