中国科学院大学
University of Chinese Academy of Sciences

操作系统课程 **B0911010Y-01**

进程间通信

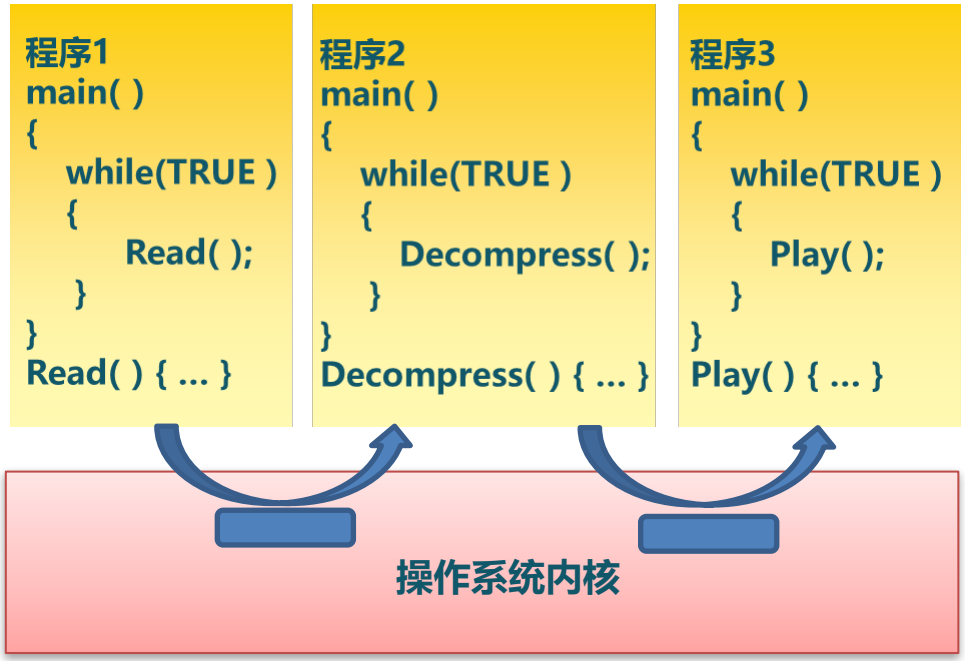
中国科学院大学计算机与控制学院
中国科学院计算技术研究所
2019-11-04

内容

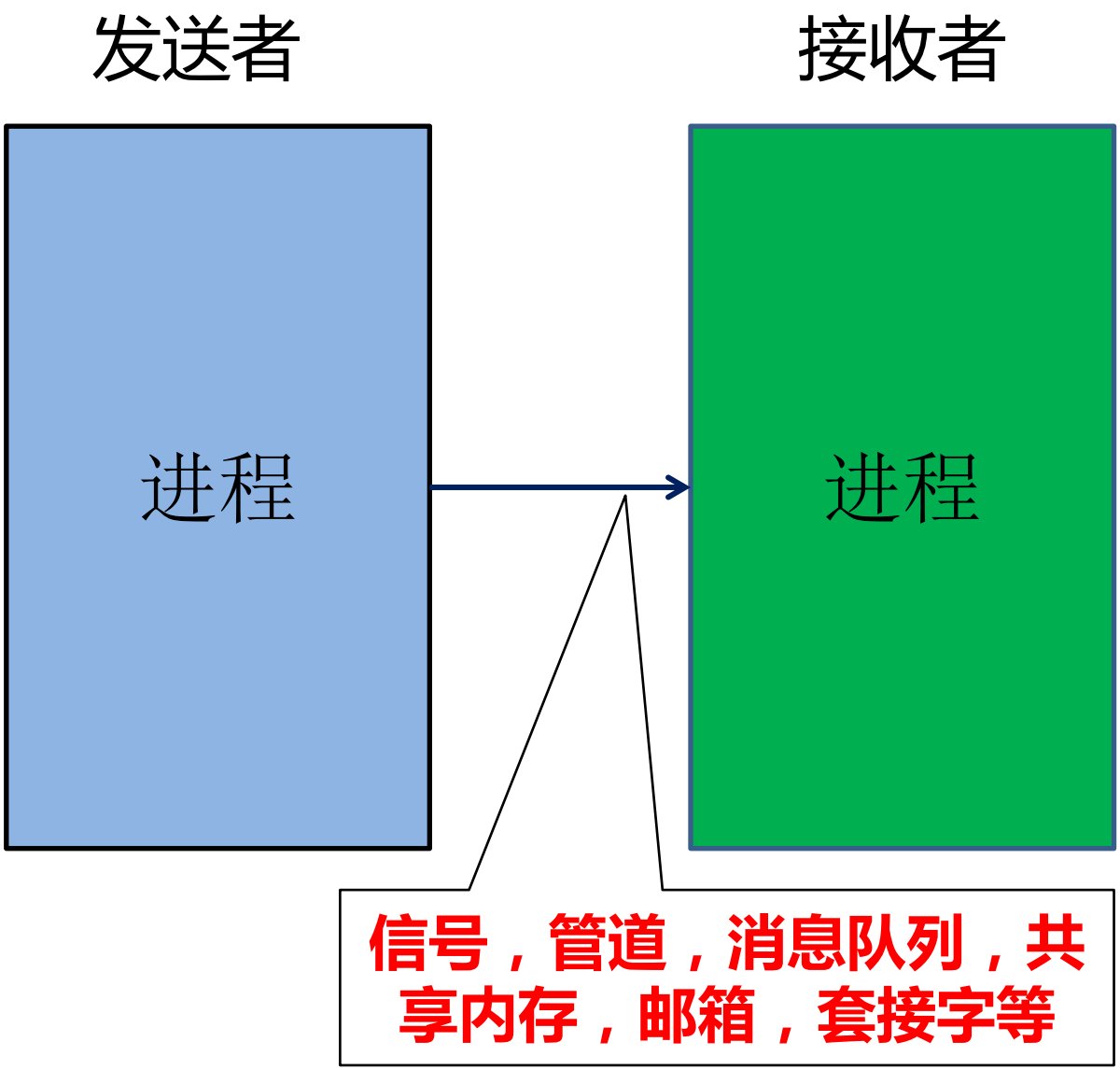
- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

基本概念

- 进程间通信（IPC, Inter-Process Communication）
  - 同步
    - 共享资源互斥访问
    - 事件通知，条件同步
  - 传输
    - 数据传输



整体结构



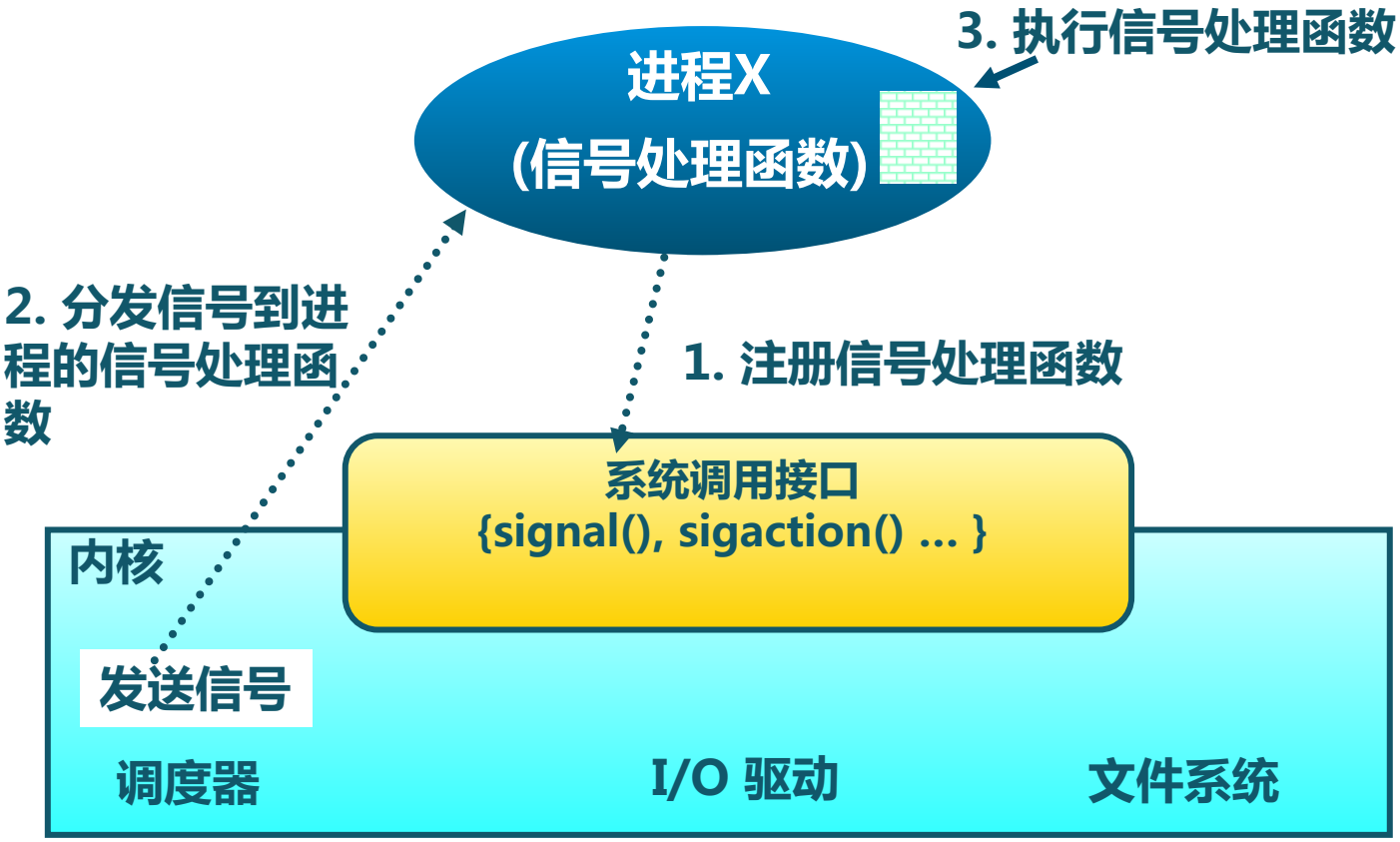
信号（Signal）

- 进程间的中断通知和处理机制
  - 如：SIGKILL, SIGSTOP, SIGCONT等
  - 可以在任一时刻发给某一进程，无需知道进程状态
  - 内核可以保存信号，再传递给进程
  - 信号可以被阻塞
- 信号的产生
  - 硬件方式
    - 键盘Ctrl+C发送SIGINT信号
    - 硬件非法访问（例如内存）
  - 软件方式
    - 通过系统调用，发送signal信号

信号（Signal）

- 信号的接收处理
  - 捕获（catch）
    - 默认处理：执行操作系统指定的缺省处理，例如进程终止、进程挂起等
    - 自定义处理：执行进程指定的信号处理函数
  - 忽略(Ignore)
    - 对信号不做任何处理
  - 屏蔽（Mask）
    - 禁止进程接收和处理信号
    - 解除屏蔽后可以接收和处理信号

信号的实现


3. 执行信号处理函数
进程X
(信号处理函数)
2. 分发信号到进程的信号处理函数
1. 注册信号处理函数
系统调用接口
{signal(), sigaction() ... }
内核
发送信号
调度器
I/O 驱动
文件系统

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 for
                             * 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 for
                             * 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(;;);
}
```

内容

- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

管道（Pipe）

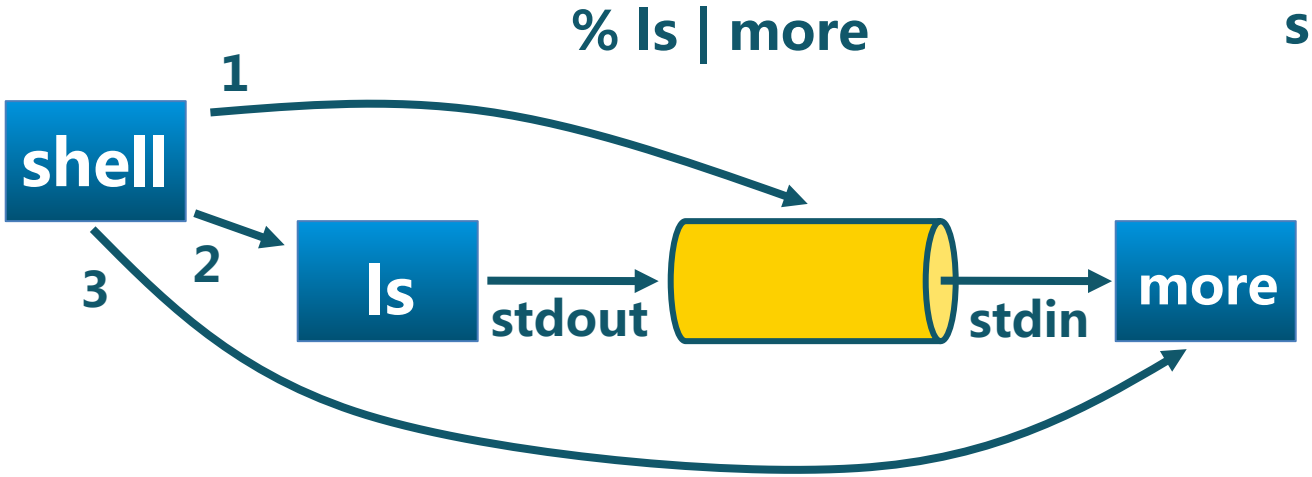
- 无名管道
  - 进程间基于内存文件的通信机制
    - 内核缓冲区，数据单向流动，半双工通信
    - 读空或写满时，需有相应并发机制控制
    - 可以用read, write函数读写
  - 只能在具有亲缘关系的进程间使用，例如父子进程，子进程可以从父进程继承文件描述符
  - 示例：pipe(int fd[2])
- 命名管道
  - 基于一种特殊设备文件的进程通信机制
  - 允许无亲缘关系的进程间通信
  - 示例：mkfifo(pathname, mode)

与管道相关的系统调用

- 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(fd[2])
  - fd是2个文件描述符组成的数组
  - fd[0]是读文件描述符
  - fd[1]是写文件描述符

管道示例

- shell通过管道重定向输入输出流
  - 可能从键盘、文件、程序读取
  - 可能写入到终端、文件、程序



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more 创建一个进程，设置 stdin 为管道读端

管道示例

- 示例

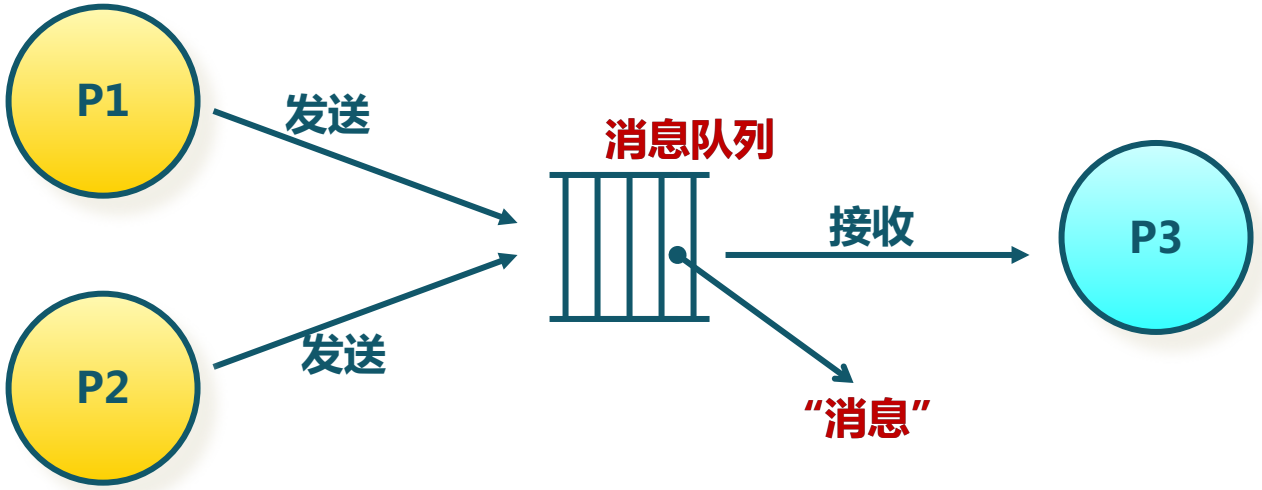
```
int fd[2]
pipe(fd)
pid = fork()
if(pid > 0) {
    close(fd[0])
    write(fd[1], ”hello world”, 20)
    close(fd[1])
}
if(pid == 0) {
   close(fd[1])
   read(fd[0], buff, 20)
   close(fd[0])
}
```

内容

- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

消息队列

- 消息队列是由操作系统维护的以字节序列为基本单位的间接通信机制
  - 独立于发送和接收进程
- 每个消息(Message)是一个字节序列
- 相同标识的消息按先进先出顺序组成一个消息队列（Message Queues）



消息队列的系统调用

- 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

消息队列示例

- 示例

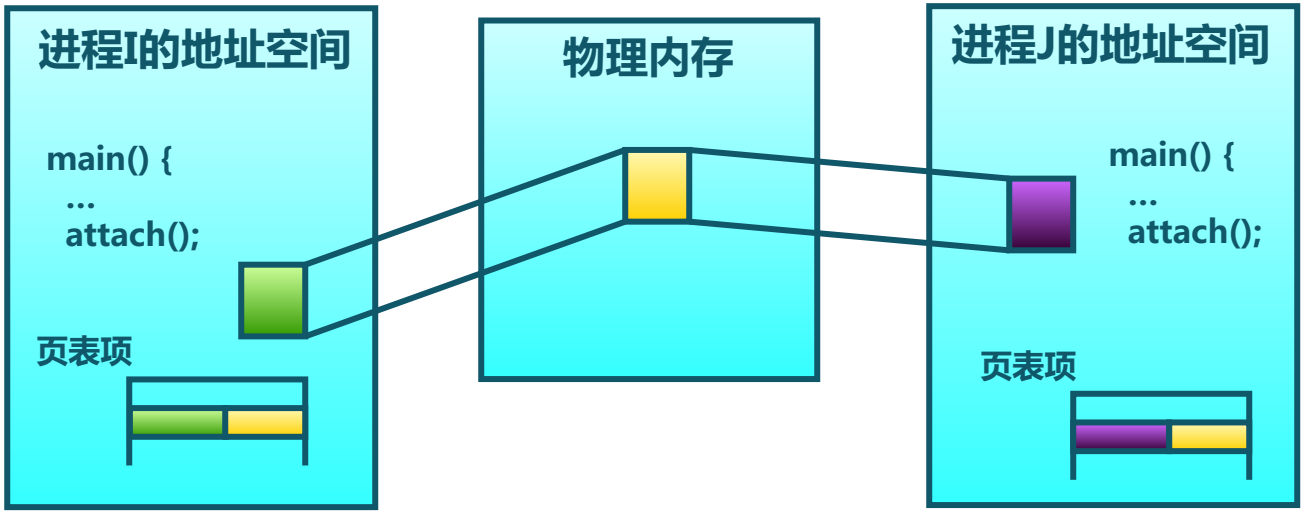
```
struct msg {
    long mtype;
    char mtext[1024];
} msgs;
//发送者
qid=msgget(MSGKEY,IPC_CREATE)
msgsnd(qid, &msgs, sizeof(struct msg), IPC_NOWAIT)
//接收者
qid=msgget(MSGKEY,IPC_CREATE)
msgrcv(qid, &msgs, sizeof(struct msg), 0, 0)
```

内容

- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

共享内存

- 共享内存是操作系统把同一个物理内存区域同时映射到多个进程的内存地址空间的通信机制
- 每个进程将共享内存区域映射到私有地址空间
- 优点：快速、方便地共享数据
- 缺点：必须用额外的同步机制来协调数据访问



共享内存系统调用

- shmget(key, size, flags)
  - 创建或获取一个共享段
- shmat(shmid, *shmaddr, flags)
  - 把共享段映射到进程地址空间
  - 返回指向共享内存的指针
- shmdt(*shmaddr)
  - 取消共享段到进程地址空间的映射
- shmctl(…)
  - 共享段控制

共享内存示例

- 示例

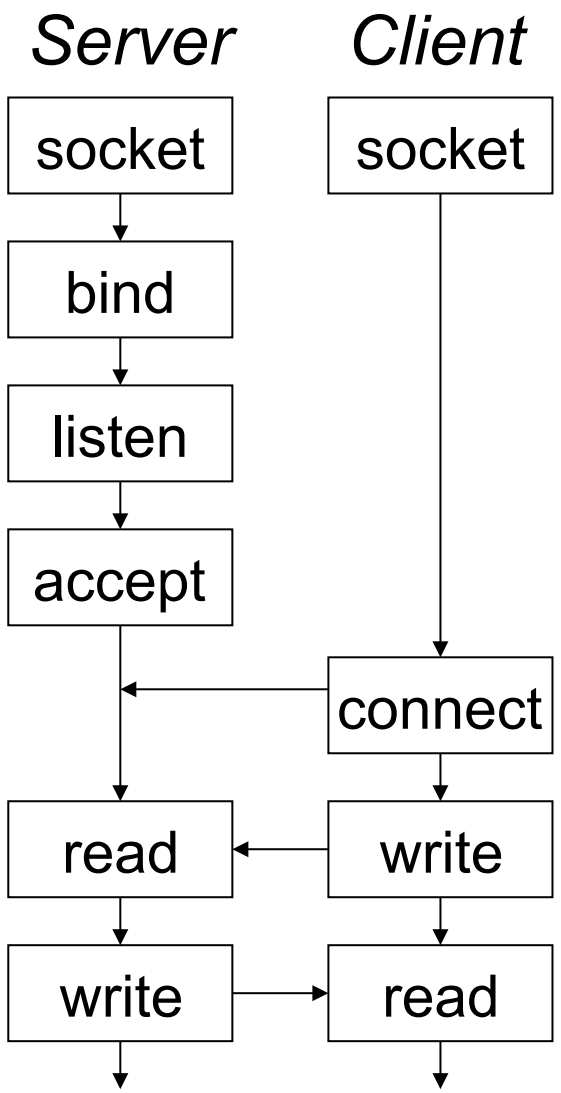
```
int shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT)

int *pi = (int *)shmat(shmid, 0, 0);
//写入内存，P、V操作？
*pi = 42;
*(pi + 1) = 33;
shmdt(pi);

//读取内存，P，V操作？
int *pi = (int *)shmat(shmid, 0, 0);
printf("%d,%d", *pi, *(pi+1))
```

内容

- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

Socket API

- TCP/UDP的抽象
- 寻址：IP地址和端口
- 创建和关闭socket
  - sockid=socket(af,type,proto);
  - sockerr = close(sockid);
- 绑定socket到本地地址
  - sockerr = bind (sockid, localaddr,addrlen);
- 监听与接收
  - listen(sockid,len);
  - accept(sockid,addr,len);
- 连接socket到目标地址
  - connect(sockid,destaddr,addrlen);



IPC方式对比

- 信号
  - 传送的信息量小，只有一个信号类型
- 管道
  - 半双工方式，单向通信；无格式字节流
  - 需进程打开、关闭管道
- 消息队列
  - 缓冲区大小受限
- 共享内存
  - 通信效率高，但需要信号量等机制协调共享内存的访问冲突
- 套接字（Socket）
  - 用于不同机器的进程间通信
  - 需要对数据进行封包和解包操作

内容

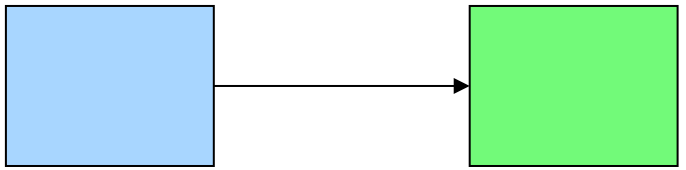
- 信号
- 管道
- 消息队列
- 共享内存
- 套接字
- IPC设计考虑

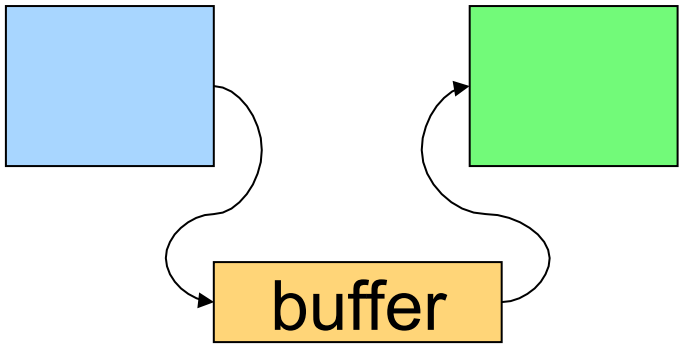
设计考虑

- 进程间通信基本原语
  - Send（msg），Receive（msg）
- 进程通信流程
  - 建立通信链路
    - 内存、设备文件、网络
  - Send/Recv交换数据
    - 是否缓冲消息
    - 直接通信 vs. 间接通信
    - 同步 vs. 异步
    - 单向通信 vs. 双向通信
    - 例外处理

缓冲消息

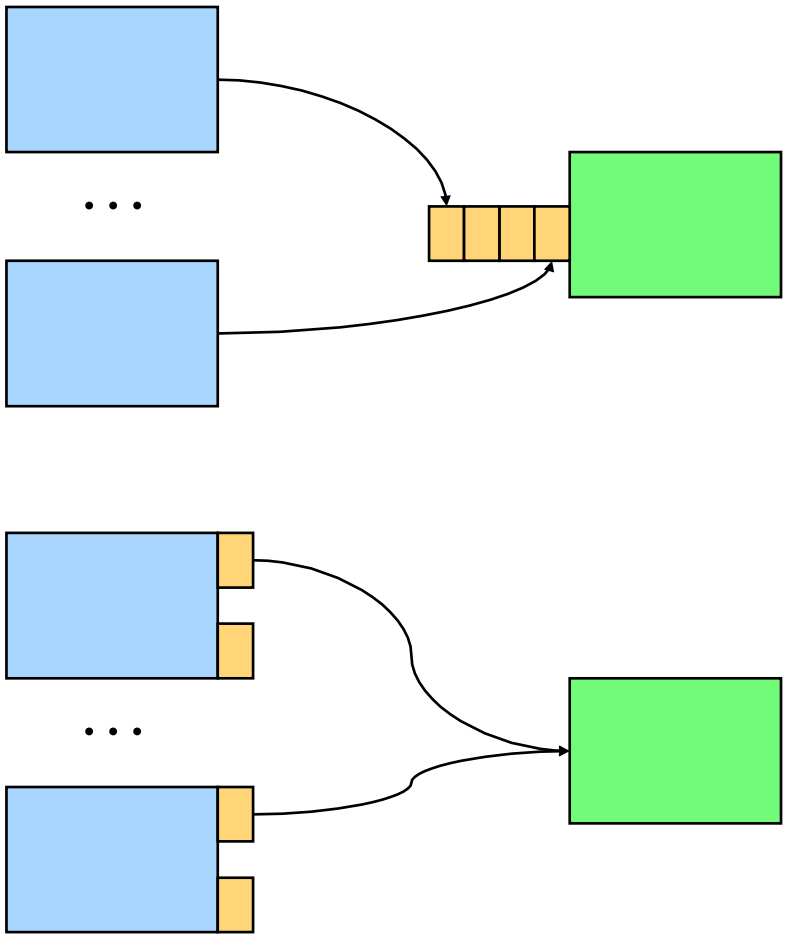
- 无缓冲
  - 发送方必须等待接收方接收消息
  - 每个消息都要握手
- 有界缓冲
  - 缓冲区长度有限
  - 缓冲区满则发送方阻塞
  - 可以使用管程
- 无界缓冲
  - “无限”长度
  - 发送方永远不阻塞
  - 实际应用受限





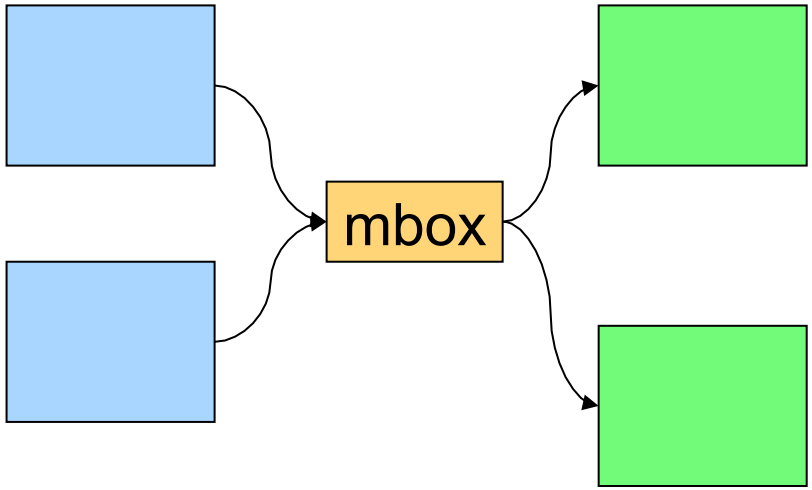
直接通信

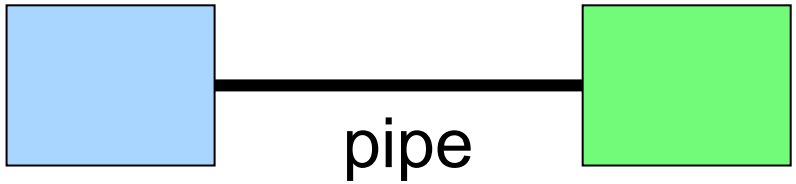
- 通信进程明确指定接收者或发送者
  - Send(P, msg)
  - Recv(Q, msg)
- 接收端有缓冲
  - 多个进程可能向接收方发送消息
  - 接收方从特定的进程接收消息，需要遍历整个缓冲区
- 发送者有缓冲
  - 每个发送者发送给多个接收者



间接通信

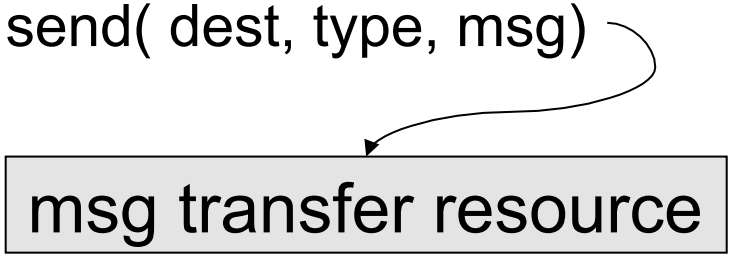
- 使用信箱
  - 允许多对多通信
  - 需要打开/关闭信箱
  - Send(A, msg)/Recv(A, msg)
- 缓冲区
  - 信箱需要有一个缓冲区以及互斥锁和条件变量
- 消息长度
  - 不确定，可以把大消息切成多个包发送
- 信箱和管道对比
  - 信箱允许多对多通信
  - 管道隐含一个发送一个接收





同步和异步：发送

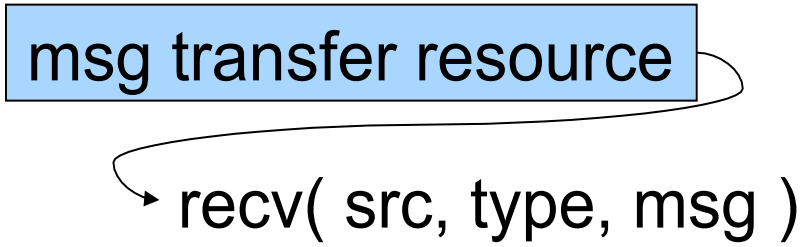
- 同步发送
  - 发送进程阻塞，直到消息由接收进程收到
  - 启动数据传输，直到源缓冲用完后再阻塞
- 异步发送
  - 发送进程调用send()启动数据传输后，继续执行其他操作
  - 结束
    - 需要应用检查状态
    - 通知或者向应用发信号




```
status = async_send( dest, type, msg )
…
if !send_complete( status )
    wait for completion;
…
use msg data structure;
…
```

同步和异步：接收

- 同步接收
  - 接收进程阻塞，直到有消息可用
  - 如果有消息，则返回数据

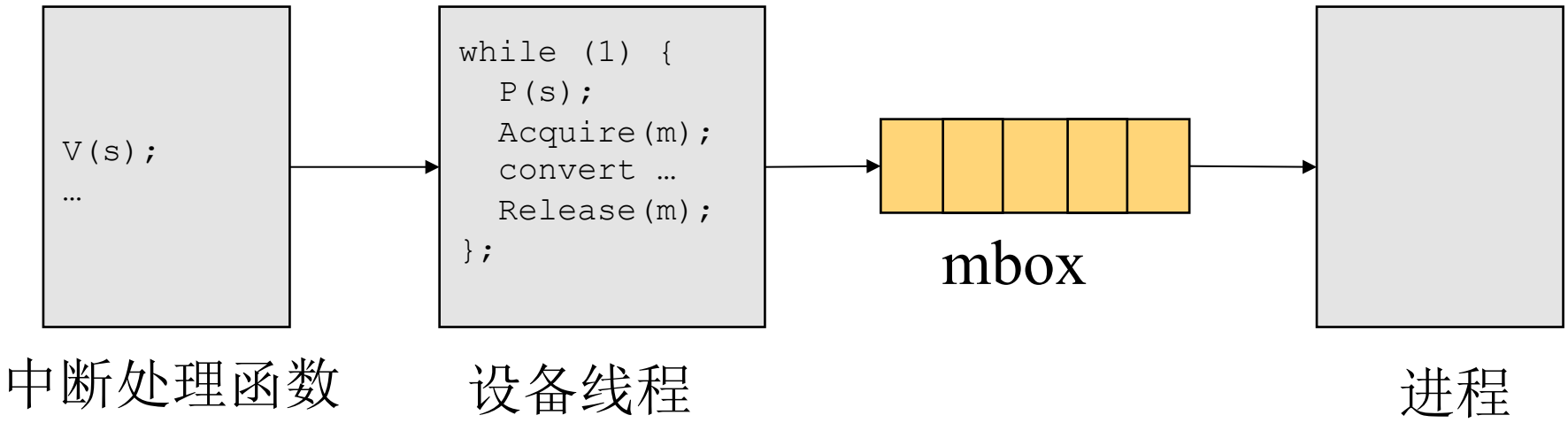


- 异步接收
  - 如果有消息，则返回数据
  - 如果无消息，则返回状态

```
status = async_recv( src, type, msg );
if ( status == SUCCESS )
    consume msg;

while ( probe(src) != HaveMSG )
    wait for msg arrival
recv( src, type, msg );
consume msg;
```

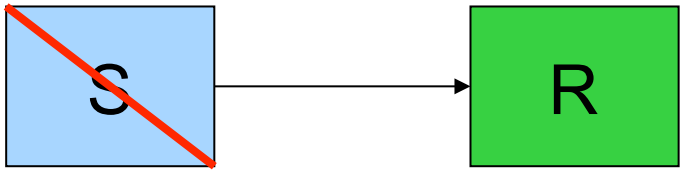
例子：键盘输入

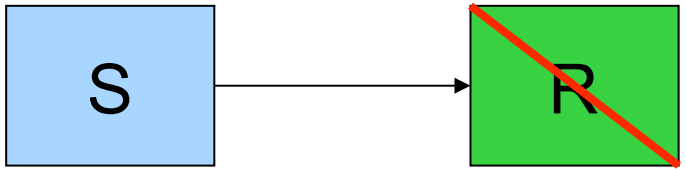
- 实现键盘输入
  - 需要一个中断处理器
  - 从中断处理函数中生成一个mbox消息
- 假设键盘设备线程把每个输入字符转换成一个mbox消息
  - 键盘中断函数怎么和设备线程同步？



例外处理：进程结束

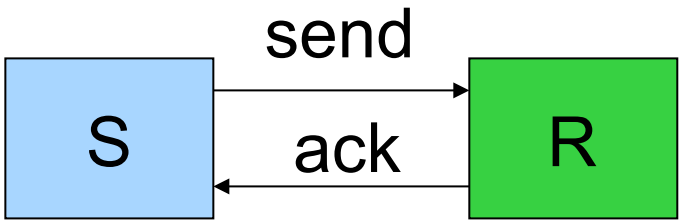
- R等待S发来的消息但S已经结束
  - 问题：同步接收时，R会永久阻塞
  - 解决：等待超时
- S发送一个消息给R，但R已经结束了
  - 问题：同步发送时，S会永久阻塞
  - 解决：发送超时





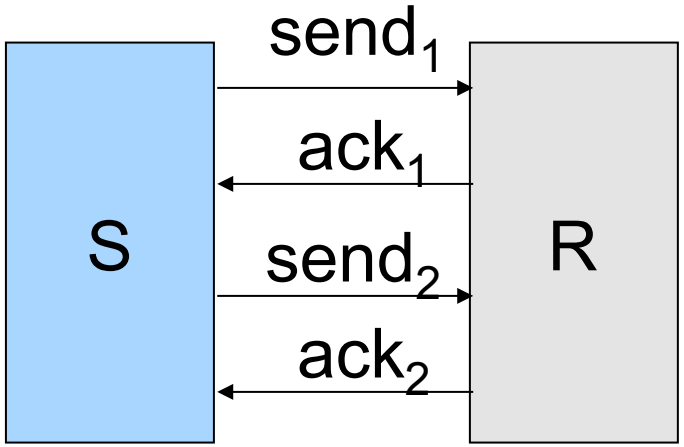
例外处理：消息丢失

- 使用确认(ACK)和超时(timeout)检测和重传消息
  - 需要接收者每收到一个消息发送一个确认
  - 发送者阻塞直到ACK到达或者超时
    - status = send(dest,msg,timeout);
  - 如果超时发生且没收到确认，重发消息
- 问题：
  - 确认消息丢失
  - 重复发送消息

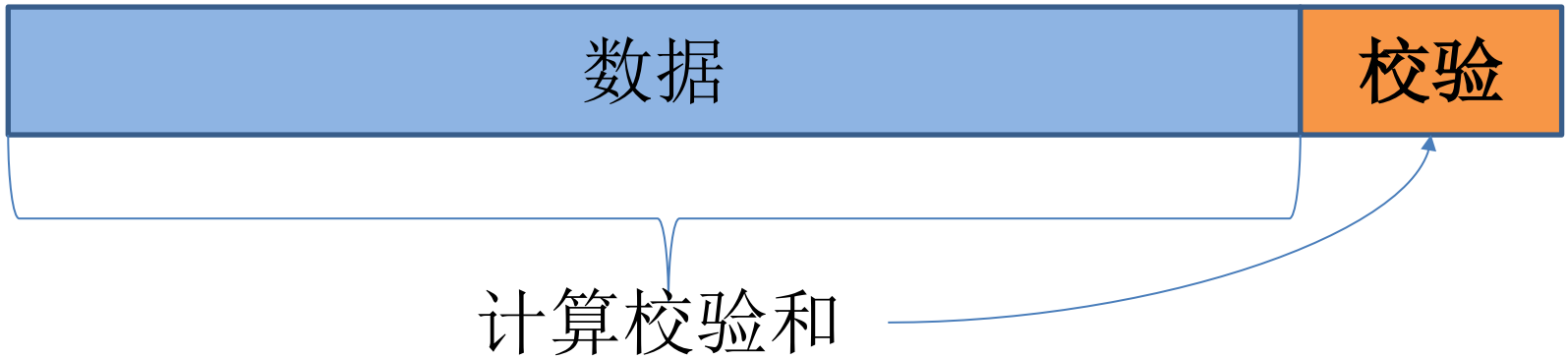


例外处理：消息丢失（续）

- 重传必须处理
  - 在接收端消息重复
  - 发送端确认乱序
- 重传：
  - 使用序列号确认是否重复
  - 在接收端删掉重复消息
  - 发送者收到乱序确认时重传
- 减少确认消息：
  - 批量传送确认
  - 接收者发送noack



例外处理：消息损坏



- 检测方法
  - 发送端计算整个消息的校验和，并随消息发送校验和（CRC）
  - 在接收端重新计算校验和，并和消息中的校验和对比
- 纠正方法
  - 重传
  - 使用纠错码恢复

总结

- 数据传输
  - 进程间移动数据
  - 隐含同步机制
  - 不同机制适用不同场景
- 设计考虑
  - 最常见的是同步方式
  - 异步方式允许并发，但需要仔细的设计
  - 间接通信实现更灵活
  - 例外需要仔细处理