



进程间通信

中国科学院大学计算机学院

2025-11-10





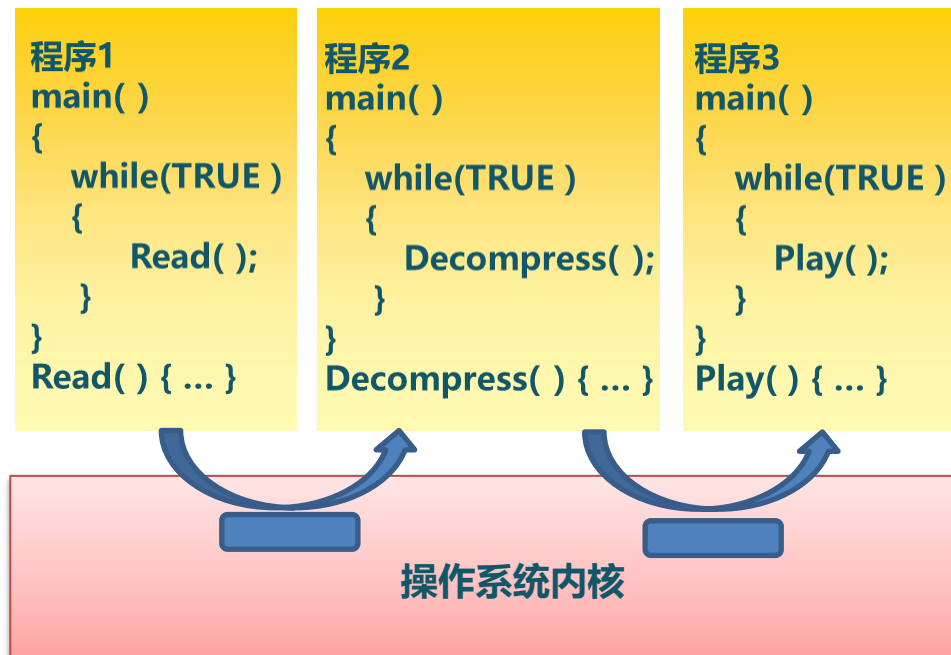
内容

- 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)
 - 不同进程协作完成任务
 - 不同进程访问共享数据

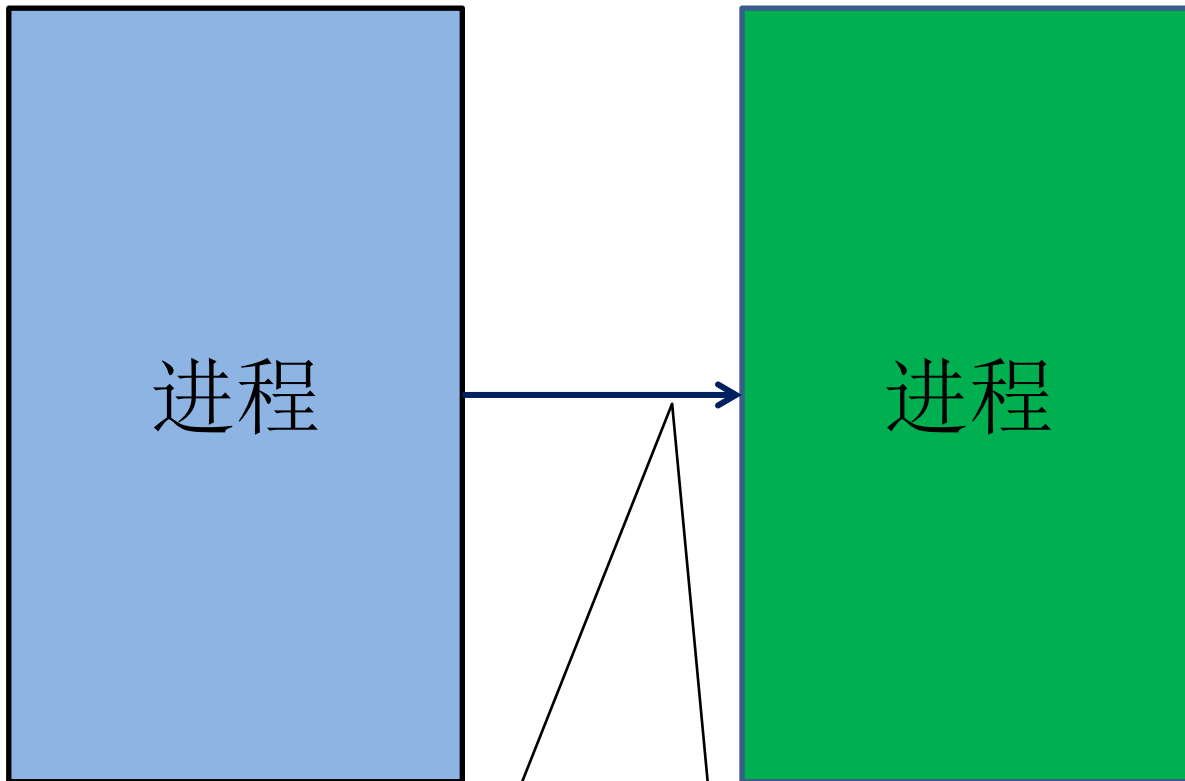




整体结构

发送者

接收者



信号，管道，消息队列，共享内存，套接字等



信号 (Signal)

- 一种用于通知进程的机制
 - 通知接收进程异常事件或错误发生
 - 可以在任一时刻发给某一进程，无需知道进程状态
 - 用于进程间简单通信，不用于大量数据传输
 - 内核可以暂存信号，等进程能处理信号时，再传递给进程
 - 信号被屏蔽时
 - 进程处于不可中断的睡眠状态
 - 例如
 - SIGKILL：强制中止进程
 - SIGTERM：请求进程终止，进程可以优雅退出
 - SIGSTOP/SIGCONT：暂停进程执行/恢复进程执行



信号 (Signal)

- 信号的产生
 - 硬件方式
 - 键盘Ctrl+C发送SIGINT信号
 - CPU检测到硬件非法访问（例如进程读取未分配的内存区域或写入只读的内存区域），通知内核生成信号（例如SIGSEGV），发送给发生事件的进程
 - 软件方式
 - 通过系统调用，发送信号
 - 常用函数：kill
 - shell命令: kill -#SigNum PID
 - kill(1234, SIGTERM)

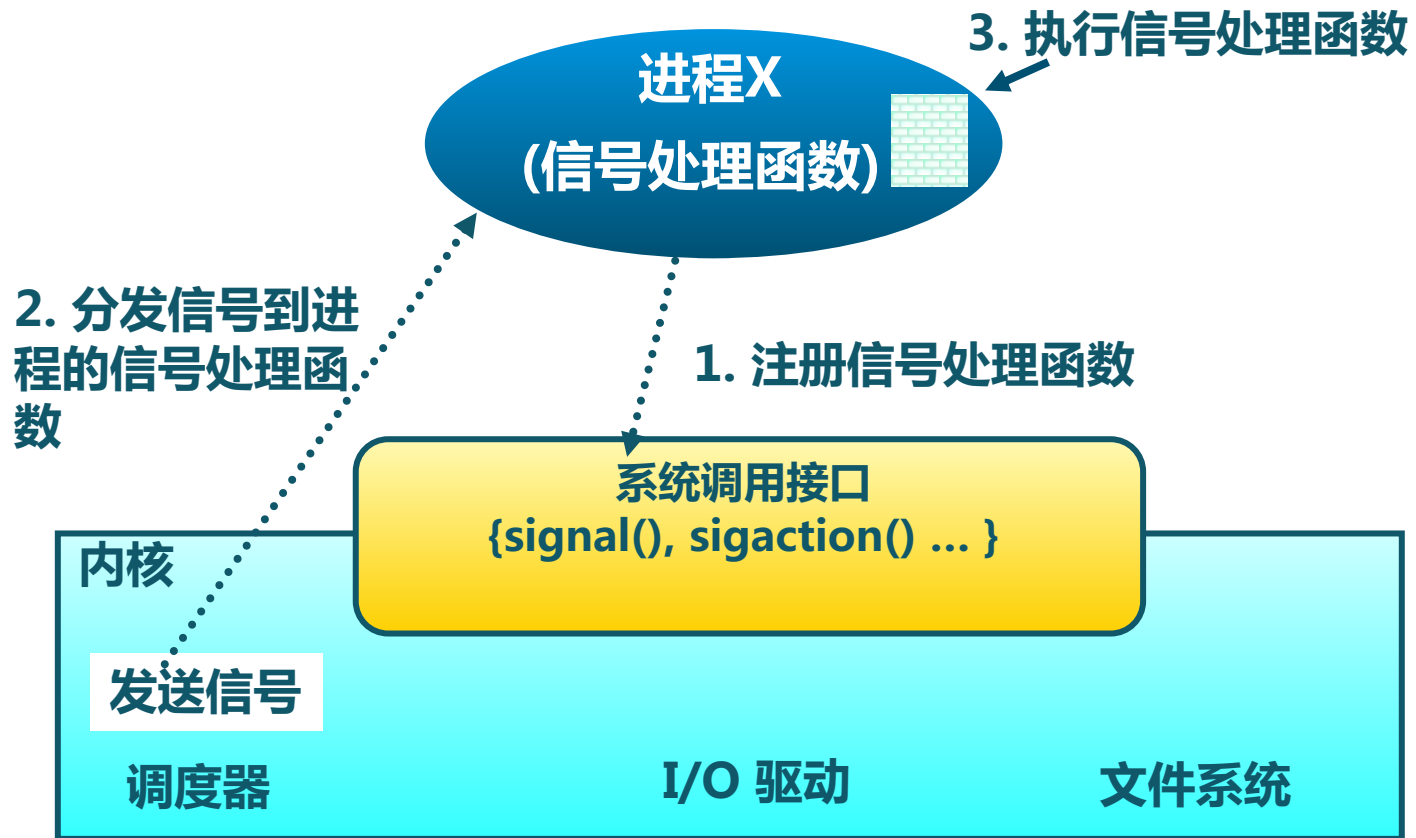


信号 (Signal)

- 信号的接收处理
 - 信号有默认行为
 - SIGTERM : 默认终止进程
 - SIGSTOP : 默认暂停进程
 - SIGSEGV : 默认终止进程
 - SIGCHLD : 默认忽略
 - 捕获
 - 自定义信号处理函数 : 执行进程指定的信号处理函数
 - 忽略
 - 对信号不做任何处理
 - 屏蔽
 - 禁止进程接收和处理信号
 - 内核暂存信号
 - 解除屏蔽后可以接收和处理信号



信号捕获的实现





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 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(;;);
}
```



信号 (Signal)

- 信号的接收处理
 - 有些信号不能捕获
 - 例如 , SIGKILL, SIGSTOP
 - 有些信号不能忽略
 - 例如 , SIGKILL, SIGSTOP, SIGSEGV
 - 有些信号不能屏蔽
 - 例如 , SIGKILL, SIGSTOP



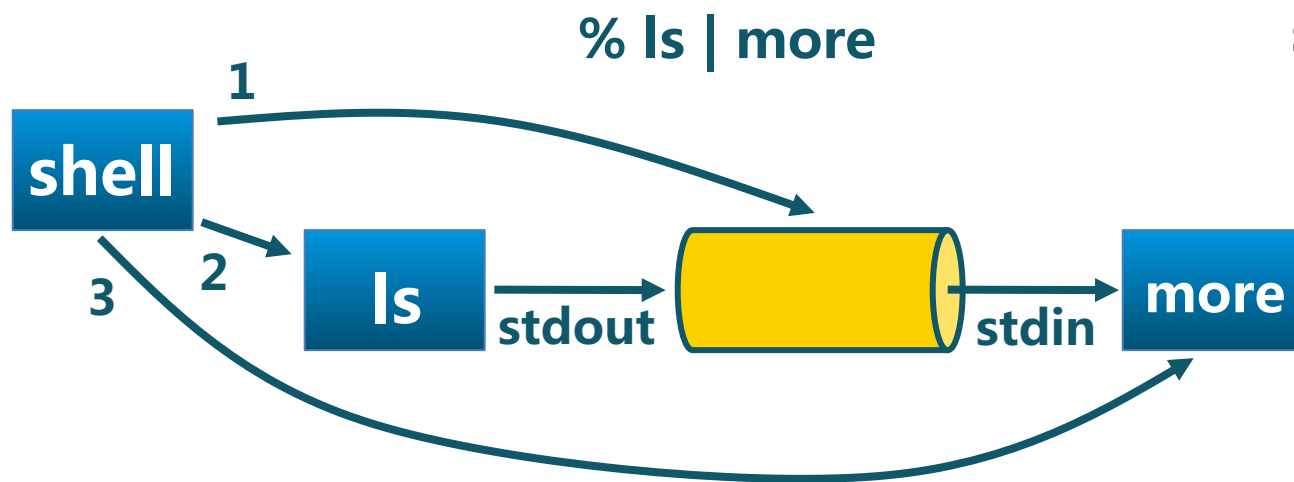
内容

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



管道示例

- shell通过管道重定向输入输出流
 - 从键盘、文件、程序读取
 - 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 (Pipe)

- 通常用于进程间进行数据流处理或命令链
- 无名管道
 - 进程间基于内核缓冲区的通信机制
 - 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
 - 使用管道的最后一个进程退出，则管道被删除
 - 数据单向流动，简化通信模型
 - 管道一端为写端，一端为读端
 - 避免同时读写引入的同步问题
 - 例如，父进程写数据，子进程读数据
 - 示例：`|, pipe(int fd[2])`



管道 (Pipe)

- 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
 - 可以持久存在，除非被显式删除
 - 支持多个生产者和消费者，需要使用同步机制保存数据的一致性
 - 示例：`mkfifo(pathname, mode)`



与管道相关的系统调用

- 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
- 读管道：read(fd, buffer, nbytes)
- 写管道：write(fd, buffer, nbytes)



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])
}
```



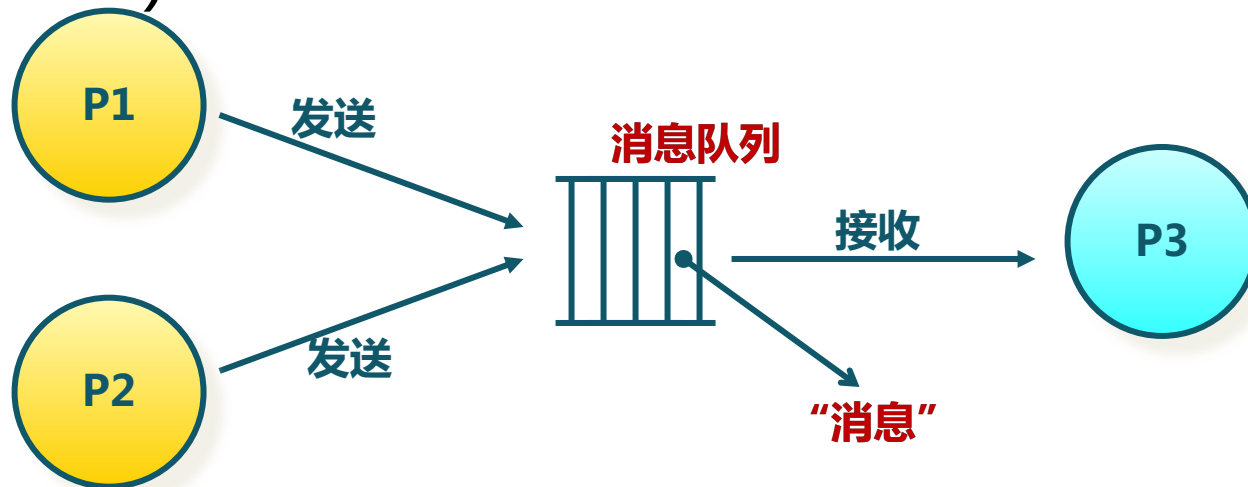
内容

- 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)包含一个消息类型和一个字节序列
- 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)
```



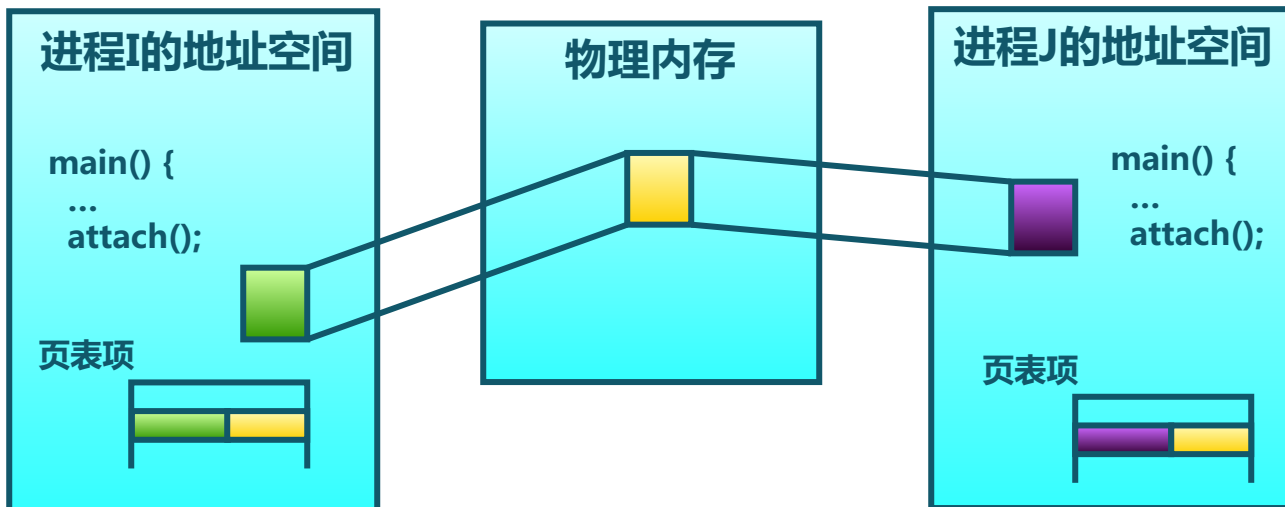
内容

- 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
- 必须用额外的同步机制来协调数据访问
- 优点：快速、方便地共享数据





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))
```



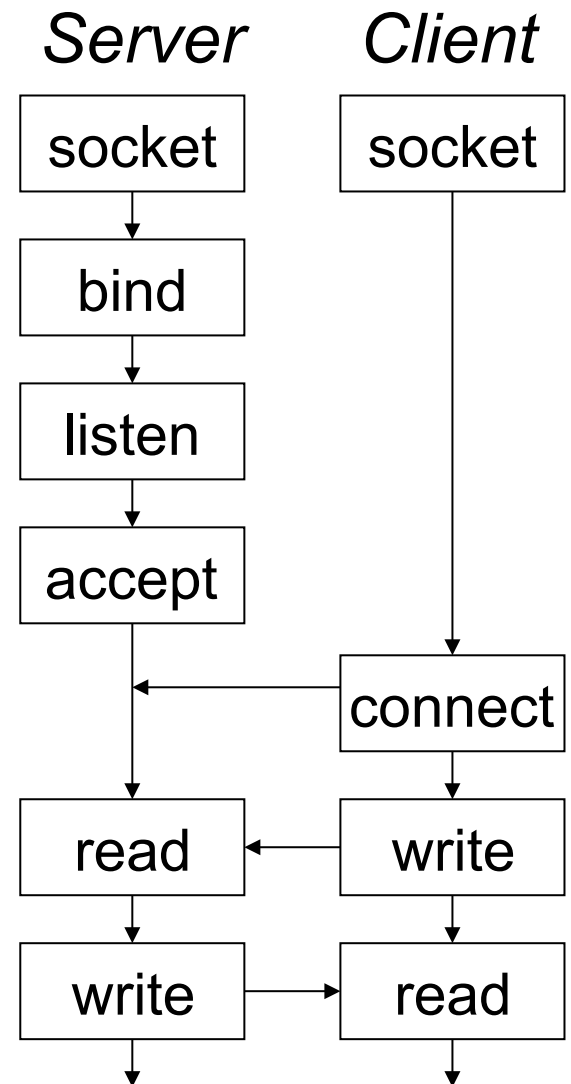
内容

- 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
 - 单向通信；无格式字节流；默认大小64KB
 - 需进程打开、关闭管道
- 消息队列
 - 支持多生产者-多消费者模型
 - 可以设置队列支持的单条消息大小和消息数量
- 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并发访问
- 套接字（Socket）
 - 主要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

- 基本原语
 - 管道：read/write
 - 消息队列：msgsnd, msgrcv
- 通信链路
 - 内存、设备文件、网络
 - Send/Recv交换数据
 - 直接通信 vs. 间接通信
 - 阻塞 vs. 非阻塞
 - 例外处理



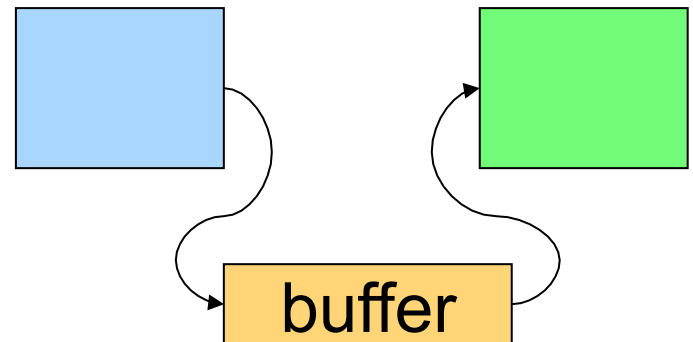
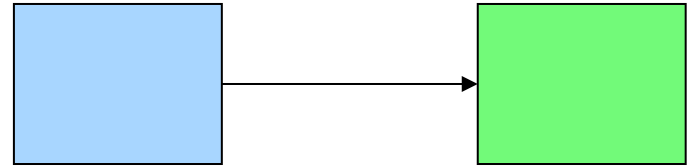
直接通信

- 进程明确指定接收者或发送者

- `Send(P, msg)`
- `Recv(C, msg)`

- 示例

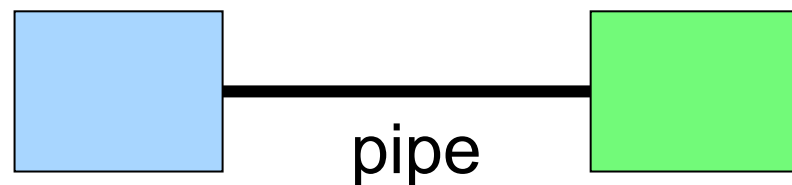
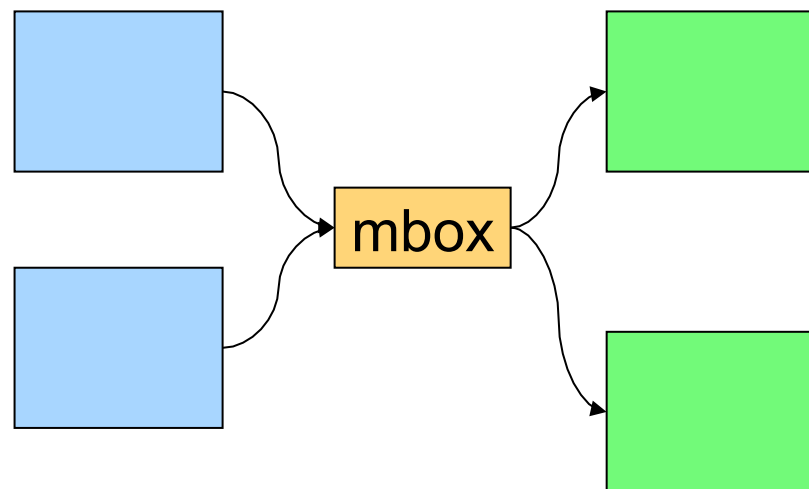
- `int kill(pid_t pid, int sig)`
- `int sigqueue(pid_t pid, int sig, const union sigval value)`





间接通信

- 使用信箱（消息队列、共享内存）
 - 允许多对多通信
 - 需要打开/关闭信箱
 - `Send(A, msg)/Recv(B, msg)`
- 缓冲区
 - 信箱需要有一个缓冲区
 - 生产者-消费者模型，使用条件同步机制
- 消息长度
 - 不确定，可以把大消息切成多个小消息发送
- 信箱和无名管道对比
 - 信箱允许多对多通信
 - 无名管道隐含一个发送一个接收





阻塞读写

- 阻塞发送
 - 当缓冲区满时，发送进程阻塞，直到消息由接收进程收到，缓冲区重新可用
-
- 阻塞接收
 - 如果缓冲区没有消息，则接收进程阻塞，直到有消息可用



非阻塞读写

• 非阻塞发送

- 当缓冲区满时，发送/写函数返回错误信息。发送进程可以不等待缓冲区可用继续执行其他操作
- 发送方需要再次检查能否发送

```
int fd[2];  
pipe(fd);
```

```
int flags = fcntl(fd[0], F_GETFL, 0);  
, fcntl(fd[0], F_SETFL, flags | O_NONBLOCK);
```

```
flags = fcntl(fd[1], F_GETFL, 0);  
fcntl(fd[1], F_SETFL, flags | O_NONBLOCK);
```

• 非阻塞接收

- 如果缓冲区有消息，则返回数据
- 如果缓冲区无消息，则接收/读函数返回错误信息，接收方继续执行其他操作

```
if (msgsnd(msgid, &msg, msgsz, IPC_NOWAIT) == -1)  
{ if (errno == EAGAIN)  
  { // 处理队列满的情况 } }
```

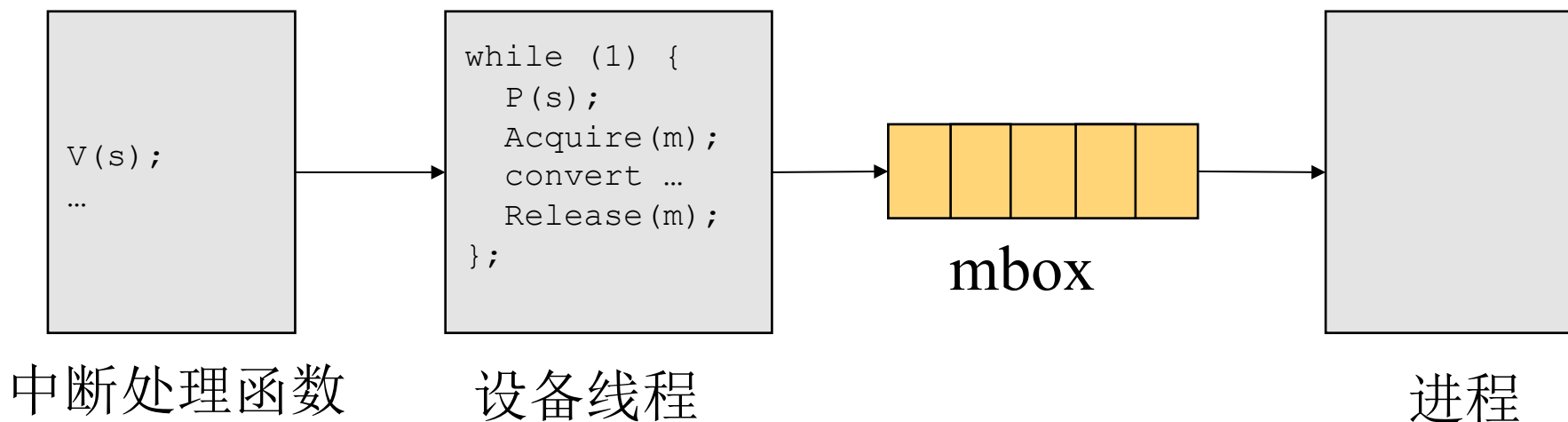
```
if (msgrcv(msgid, &msg, msgsz, msgtype,  
IPC_NOWAIT) == -1) {  
  if (errno == ENOMSG)  
    { // 处理队列空的情况 } }
```



例子：基于条件同步和IPC实现键盘输入

- 实现键盘输入

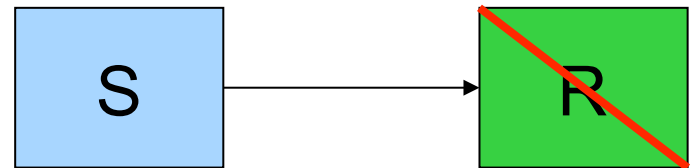
- 键盘按键按下时会产生中断信息
- 键盘中断处理函数识别按键码，使用信号量通知设备线程
- 设备线程基于按键码生成mbox消息，发送给进程





例外处理：进程结束

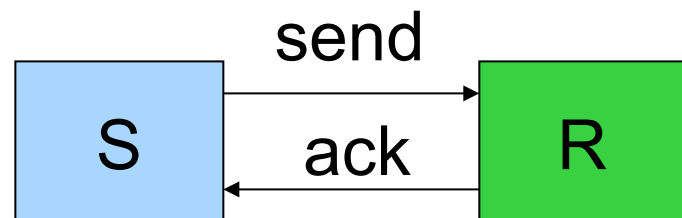
- R等待S发来的消息但S已经结束
 - 问题：R使用读阻塞模式时，会永久阻塞
 - 解决：等待超时
- S发送一个消息给R，但R已经结束了
 - 问题：S使用写阻塞模式时，会永久阻塞
 - 解决：发送超时





例外处理：消息丢失

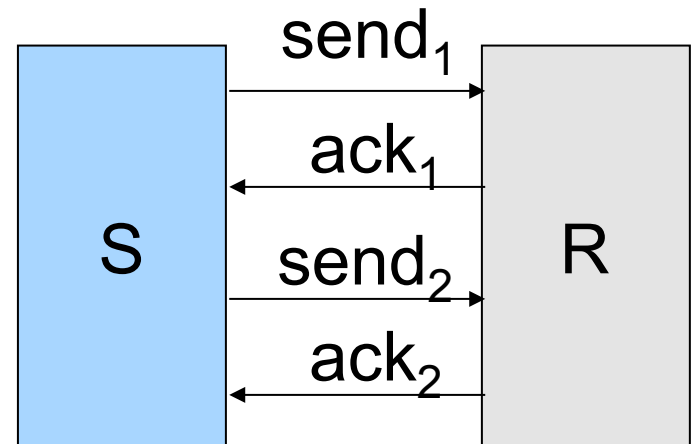
- 使用确认(ACK)和超时(timeout)检测和重传消息
 - 需要接收者每收到一个消息发送一个确认
 - 发送者阻塞直到ACK到达或者超时
 - `status = send(dest,msg,timeout);`
 - 如果超时发生且没收到确认，重发消息
- 问题：
 - 如果消息没有丢失，重复发送了消息怎么办？





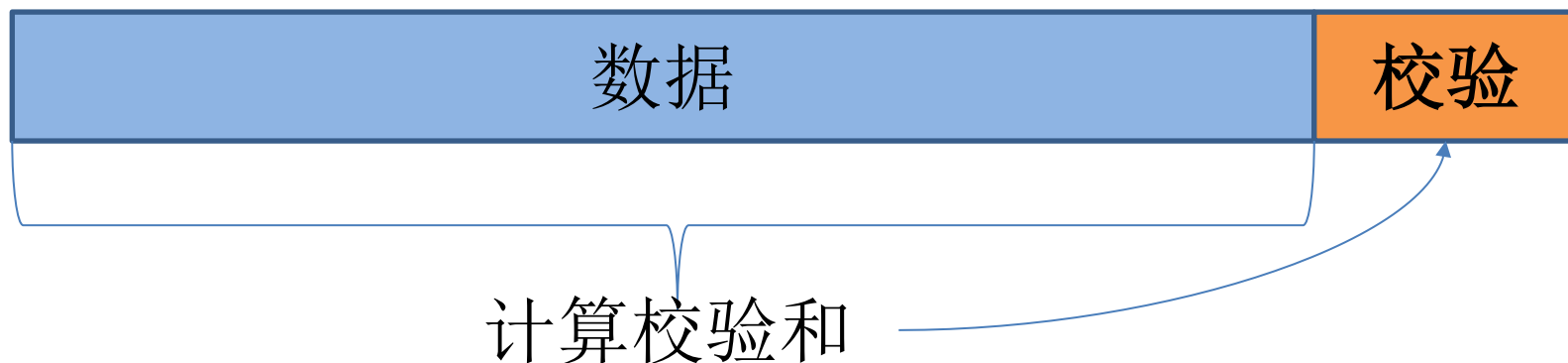
例外处理：消息丢失（续）

- 重传必须处理
 - 接收端收到的消息重复
- 重传
 - 使用序列号确认是否重复
 - 在接收端删掉重复消息
- 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

- 进程间通信
 - 信号、管道、消息队列、共享内存、套接字
 - 多进程协作执行任务、共享访问数据、数据传输
 - 典型的生产者-消费者模型，需要同步机制支持
 - 根据传输数据大小、数据格式等，不同机制适用不同场景
- 设计考虑
 - 间接通信可以实现更灵活的进程间协作、数据传输
 - 根据应用需求，决定采用阻塞模式或非阻塞模式
 - 例外需要仔细处理