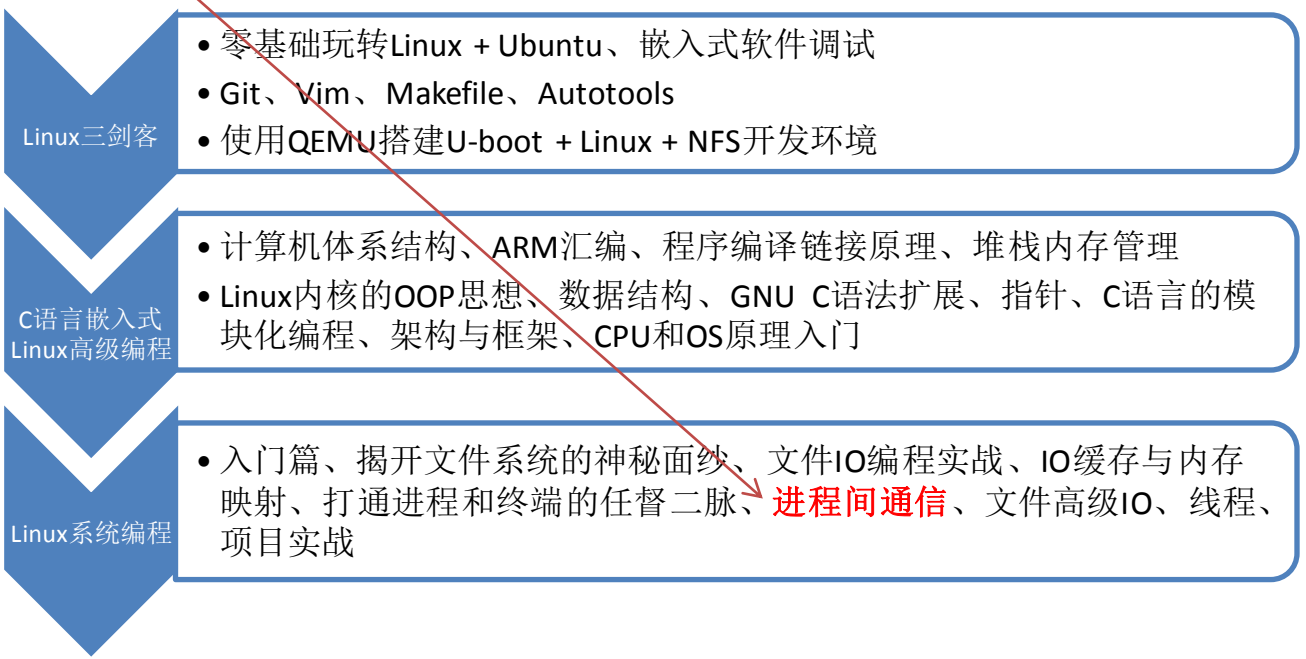
文档版本	说明	作者	创建日期
V0.1	Linux系统编程：入门篇视频配套PPT	王利涛	2018年10月14日
V0.2	第01期：揭开文件系统的神秘面纱	王利涛	2018年11月07日
V0.3	第02期：文件IO编程实战	王利涛	2018年11月25日
V0.4	第03期：IO缓存与内存映射	王利涛	2018年12月11日
V0.5	第04期：打通进程和终端的任督二脉	王利涛	2019年03月15日
V0.6	第05期：进程间通信	王利涛	2019年07月24日

学习路线图

- We are here…



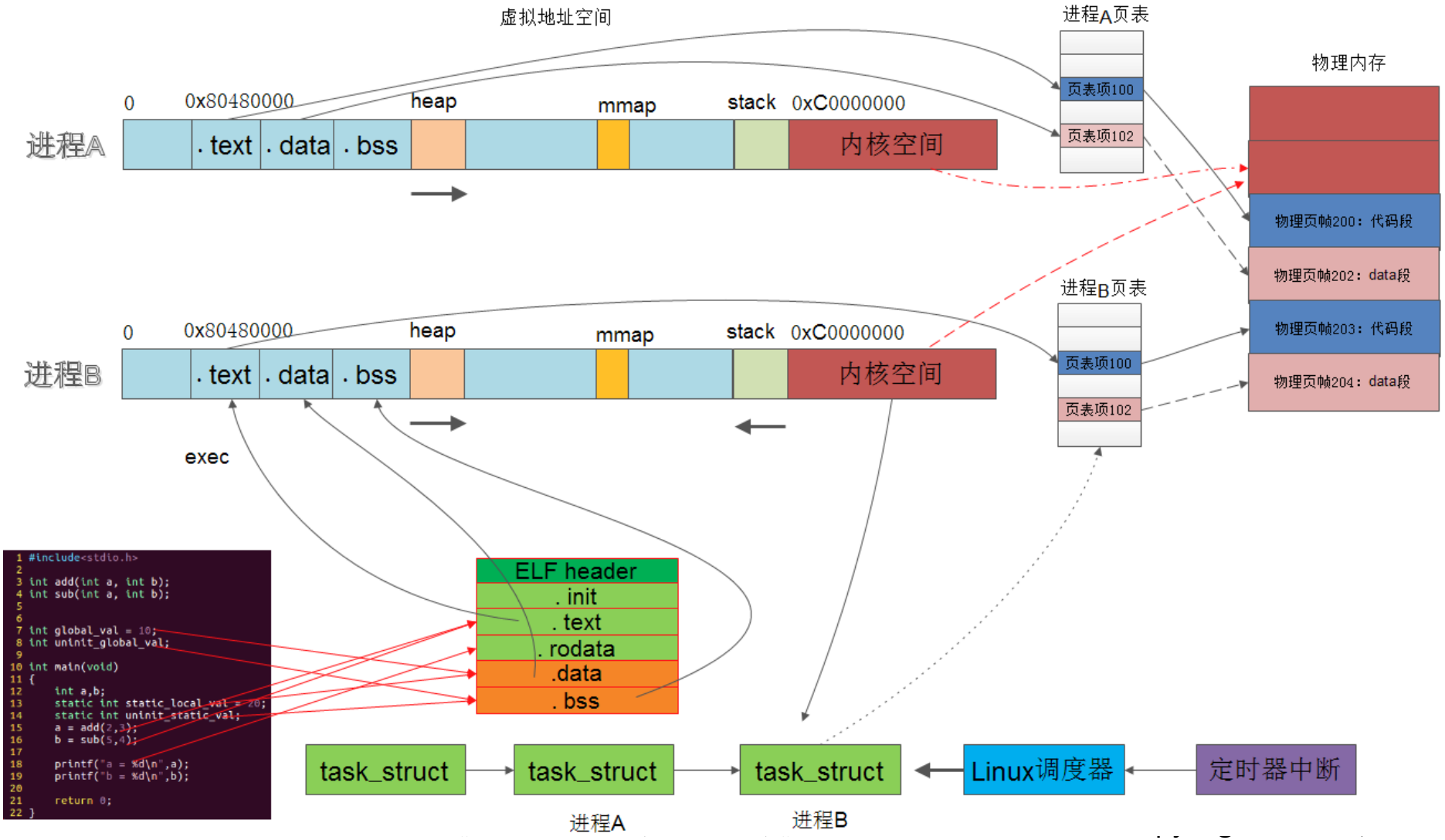
《嵌入式工程师自修养》视频教程

- 第00步：Linux三剑客
- 第01步：C语言嵌入式Linux高级编程
- 第03步：Linux系统编程
- 第04步：Linux内核编程
- 第05步：嵌入式驱动开发
- 第06步：项目实战
- ----------
- 详情咨询QQ：3284757626
- 视频淘宝店：wanglitao.taobao.com
- 博客：www.zhaixue.cc
- 微信公众号：

Linux系统编程第05期

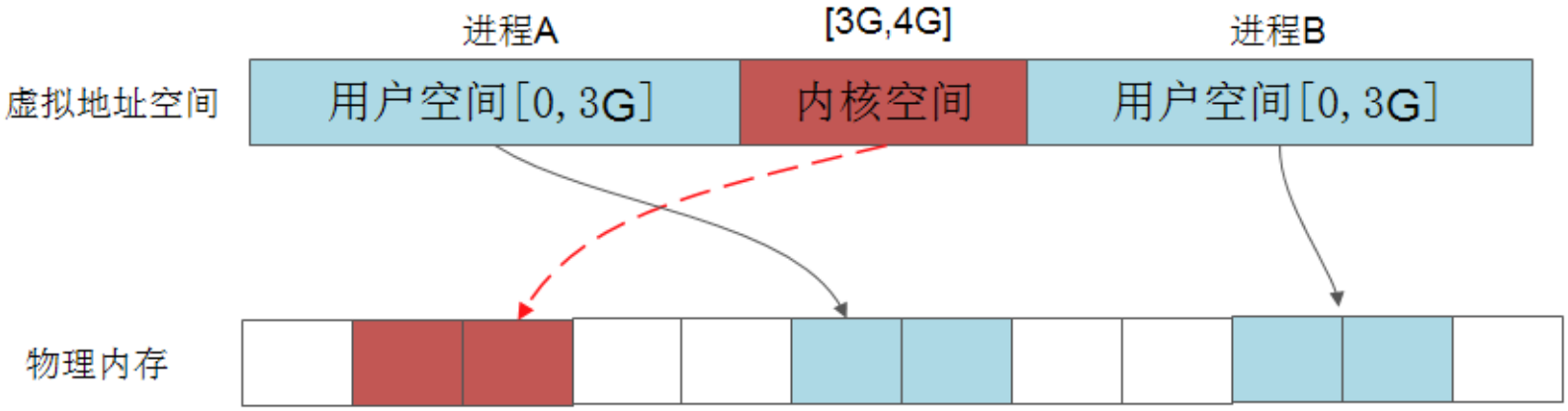
进程间通信IPC

进程的地址空间



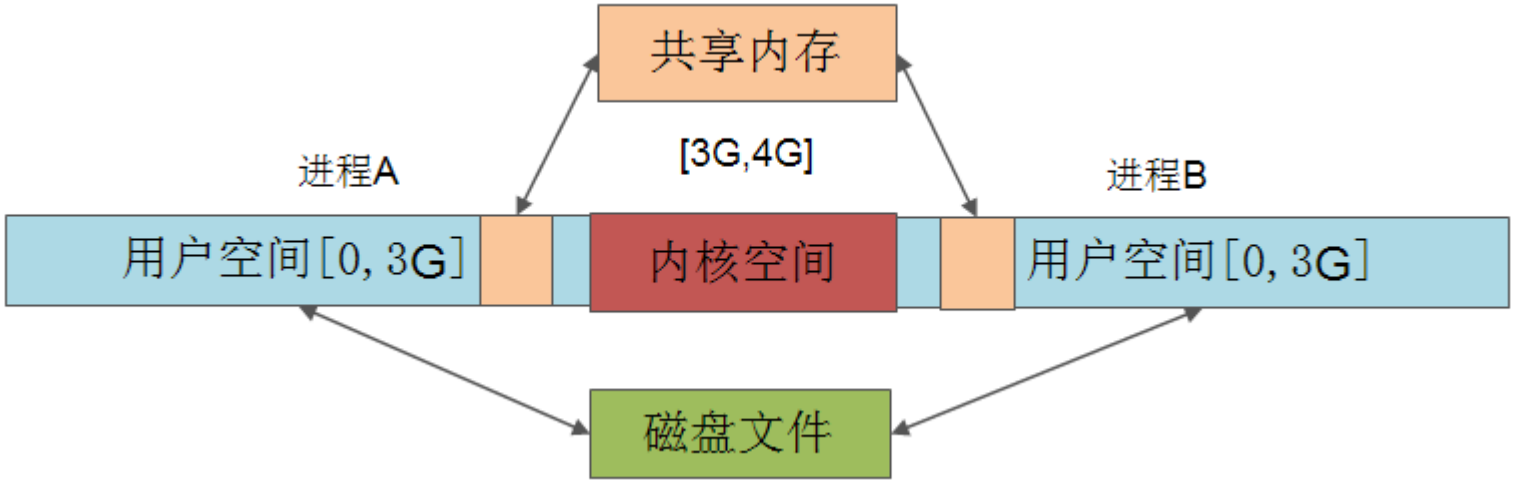

```c
#include<stdio.h>

int add(int a, int b);
int sub(int a, int b);

int global_val = 10;
int uninit_global_val;

int main(void)
{
    int a,b;
    static int static_local_val = 20;
    static int uninit_static_val;
    a = add(2,3);
    b = sub(5,4);

    printf("a = %d\n",a);
    printf("b = %d\n",b);

    return 0;
}
```

进程的物理空间



进程间通信

- 通过文件
- 通过内核
- 共享内存
- 实验：两个进程通过磁盘文件交换数据



进程间通信

- 什么是IPC?
  - IPC：inter-process communication
  - Pipe、FIFO
  - System V IPC： message queue、semaphore、 share-memory
  - POSIX IPC： message queue、semaphore、 share-memory
  - Signal
  - Socket IPC
  - D-BUS
  - …

Linux进程间通信

- IPC工具的分类
  - 通信
    - 进程之间的数据传输、交换
    - 管道、FIFO、socket、消息队列、共享内存、内存映射
  - 同步
    - 进程或线程操作之间的同步
    - 信号量、条件变量、文件锁、读写锁
  - 异步通信
    - 信号

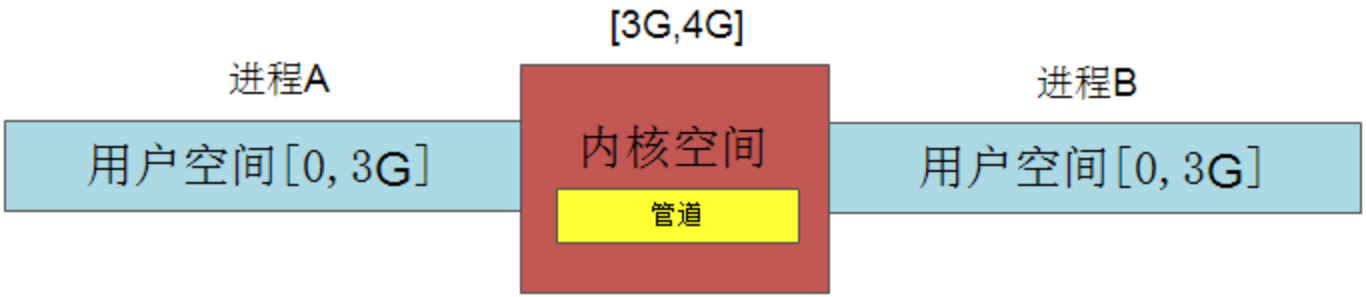
Linux进程间通信

- 不同IPC的应用场合
  - 无名管道：只能用于亲缘关系的进程
  - 有名管道：任意两进程间通信
  - 信号量：进程间同步，包括system V 信号量、POSIX信号量
  - 消息队列：数据传输，包括system V 消息队列、POSIX消息队列
  - 共享内存：数据传输，包括system V 共享内存、POSIX共享内存
  - 信号：主要用于进程间异步通信
  - Linux新增API：signalfd、timerfd、eventfd
  - Socket IPC：不同主机不同进程之间的通信
  - D-BUS：用于桌面应用程序之间的通信

无名管道：PIPE

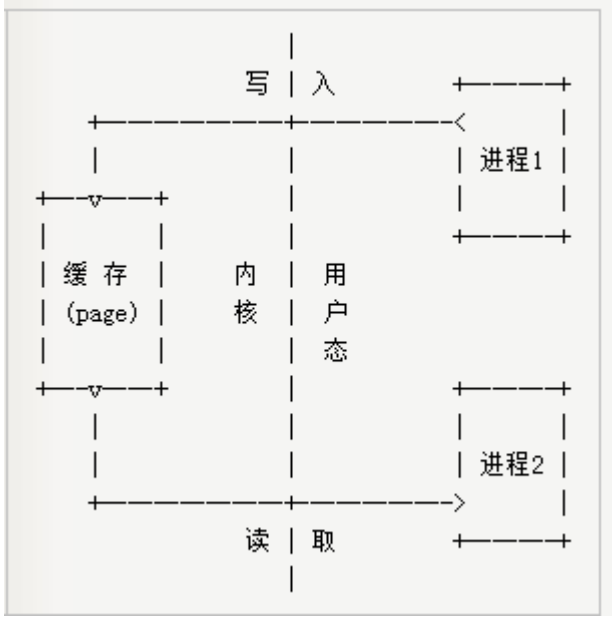
无名管道：PIPE

- Linux内核中的管道



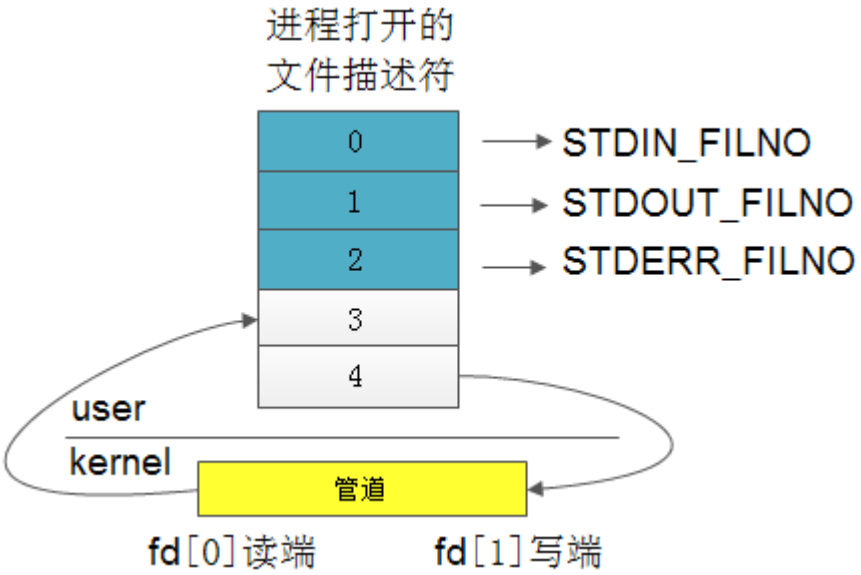
PIPE的内核层实现

```
// $ locate  pipe_fs_i.h
struct pipe_inode_info {
wait_queue_head_t wait;
char *base;                 //指向管道缓存首地址
unsigned int len;           //管道缓存使用的长度
unsigned int start;         //读缓存开始的位置
unsigned int readers;
unsigned int writers;
unsigned int waiting_writers;
unsigned int r_counter;
unsigned int w_counter;
struct fasync_struct *fasync_readers;
struct fasync_struct *fasync_writers;
};
```



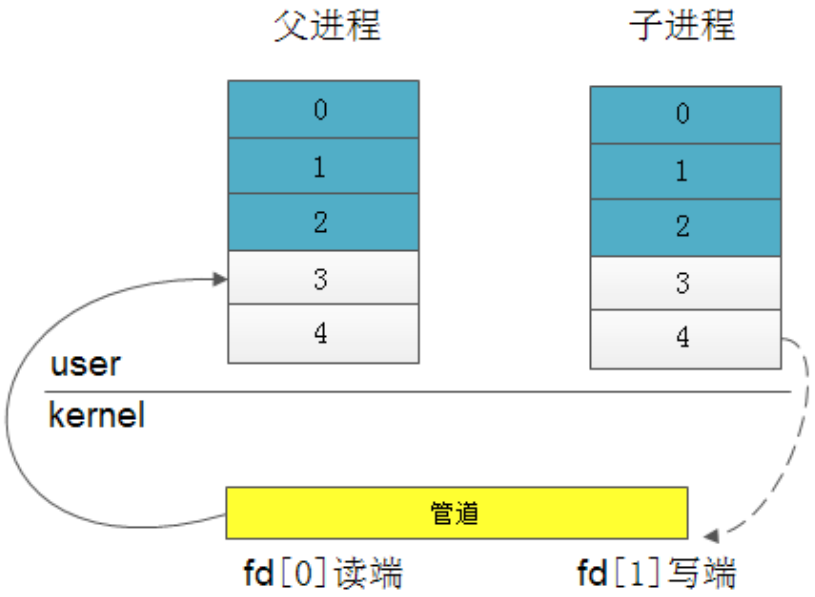
PIPE的内核层实现

- 通信原理
  - 管道是一个文件(pipefs)：
    - 内核将一个缓冲区与管道文件进行关联、封装
    - 用户可通过open/read/write/close等I/O接口进行读写



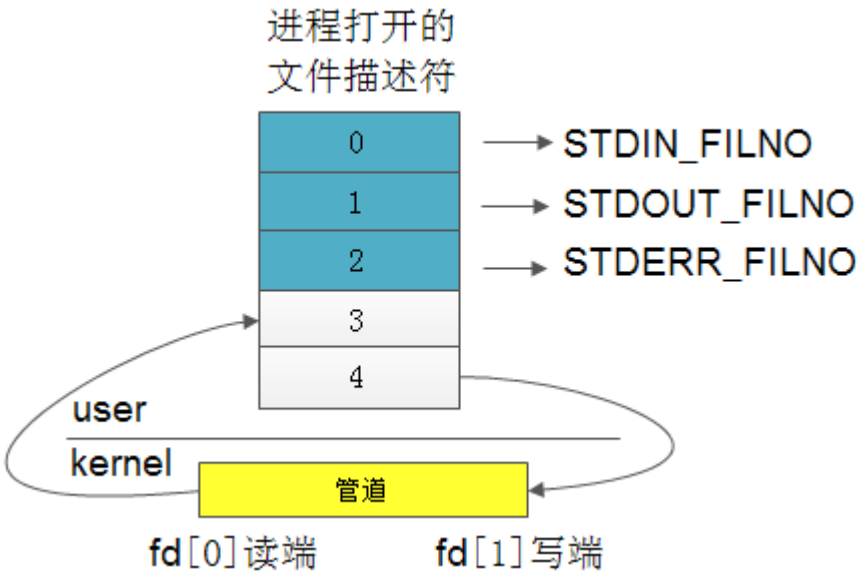
PIPE的内核层实现

- 通信原理
  - 像一个管道连接两个进程
  - 一个进程的输出作为另一个进程的输入
  - 用于亲缘进程之间的通信：共享资源



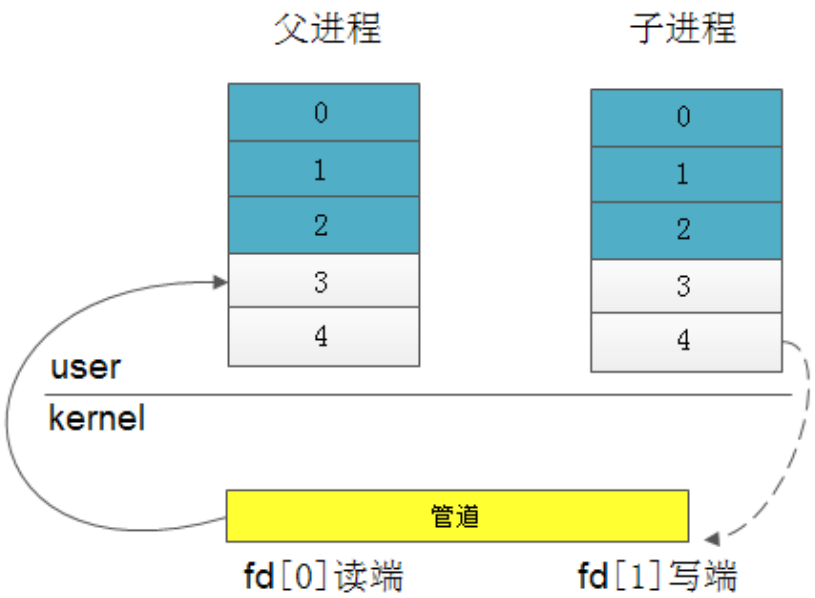
PIPE管道编程

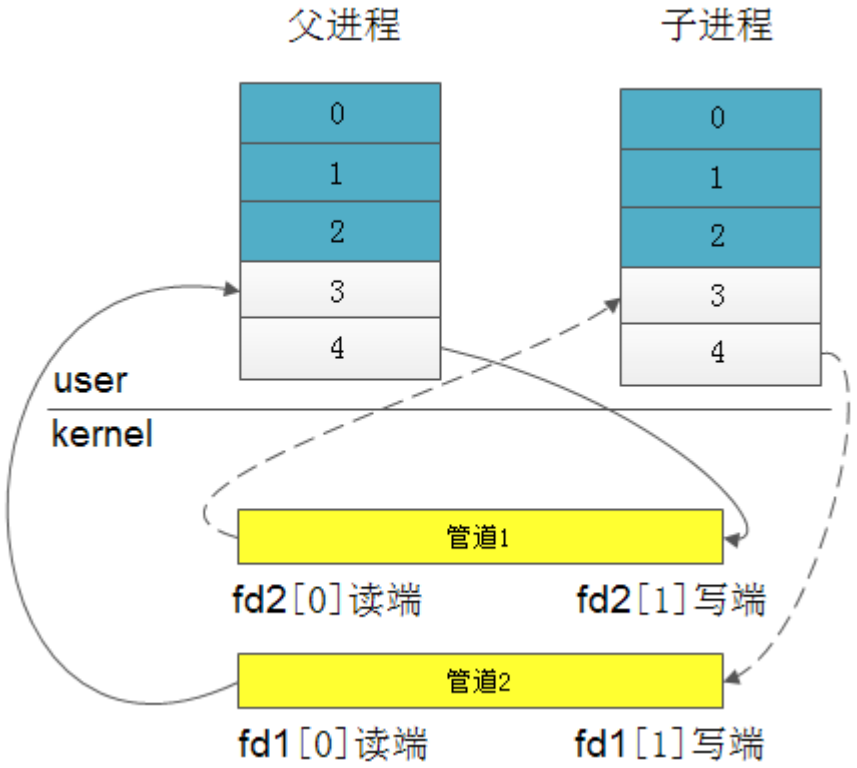
- 创建一个管道
  - 函数原型：
    - int pipe (int pipefd[2]);
    - int pipe2(int pipefd[2], int flags);
  - 函数参数：管道的两个文件描述符：一个用来读、一个用来写



Pipe管道编程

- 编程实例
  - 单向通信
  - 双向通信

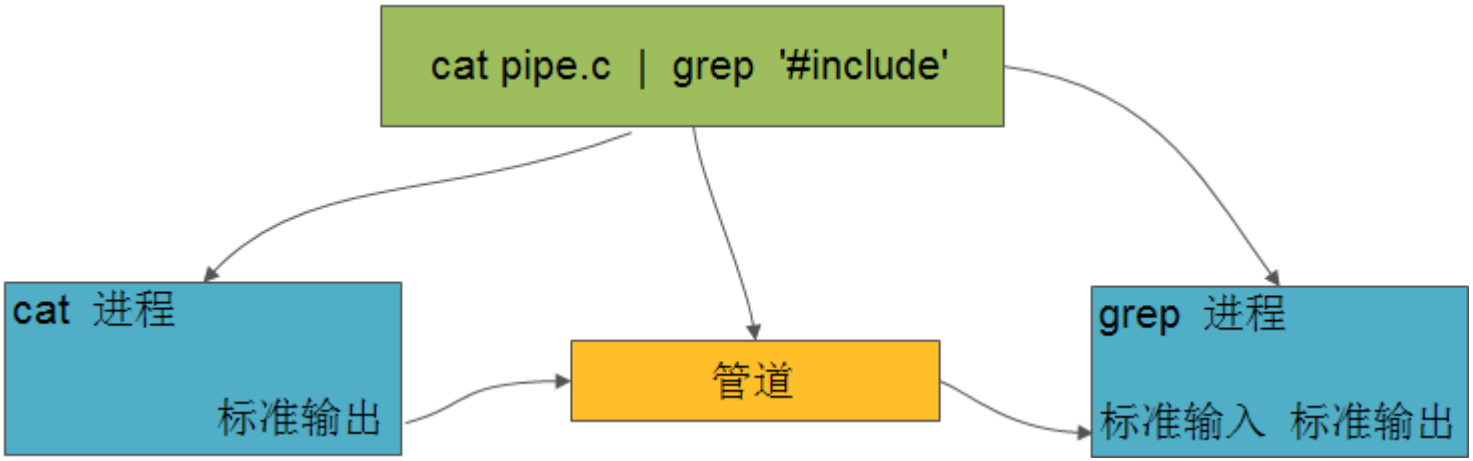




无名管道(2)：
shell管道的实现

Shell中的管道通信

- 管道的作用
  - Shell中具有亲缘关系的进程之间传递消息
  - 管道的本质是一个字节流



在shell中运行命令

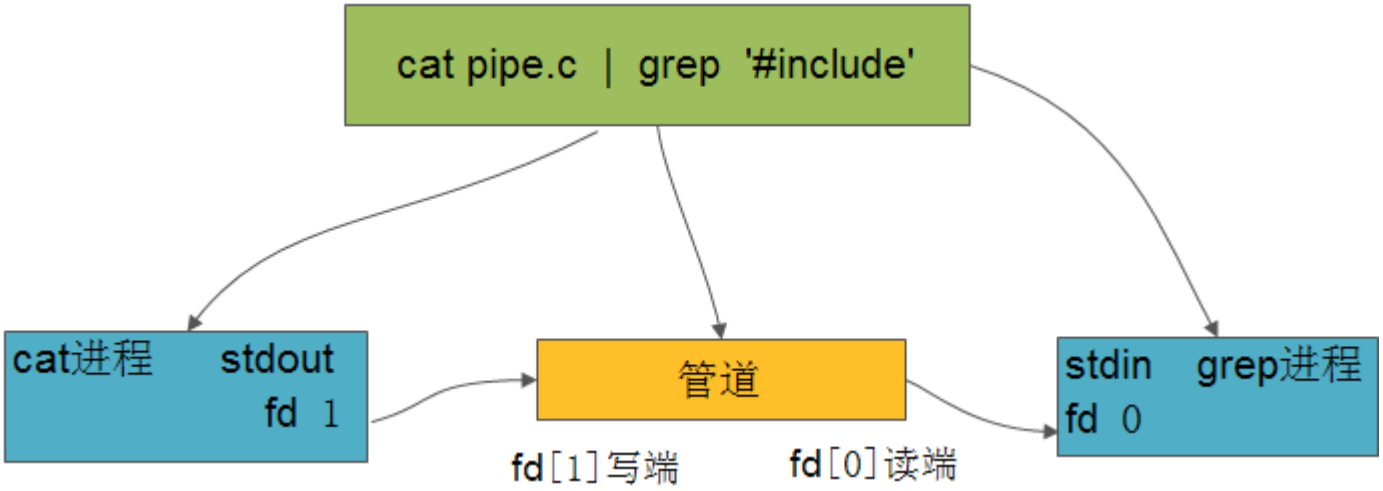
- 基本流程和重定向功能
  - 封装成进程：fork/exec系统调用
  - 该进程默认打开的stdin 、stdout 、stderr 连接在终端上
  - 运行的命令从键盘读取数据并且把输出和错误消息写到屏幕上
  - 通过重定向，可以从指定文件读取数据，或将数据输出到指定文件
  - 重定向I/O的功能是由shell本身实现的：标准流与文件的连接
  - 程序本身并不知道数据最后流向哪里：只跟标准流打交道
  - 通过命令：cmd > file 告诉shell将 stdout 定位到文件file，于是shell就将文件描述符与指定的文件连接起来，程序的输出到file，而不是默认的屏幕

输入输出重定向

- dup函数和dup2函数
  - 将管道和输入输出设备联系起来
  - 输入、输出重定向到某个设备、文件
  - #include <unistd.h>
  - int dup(int oldfd);
  - int dup2(int oldfd, int newfd);
  - int dup3(int oldfd, int newfd, int flags);

SHELL管道的实现

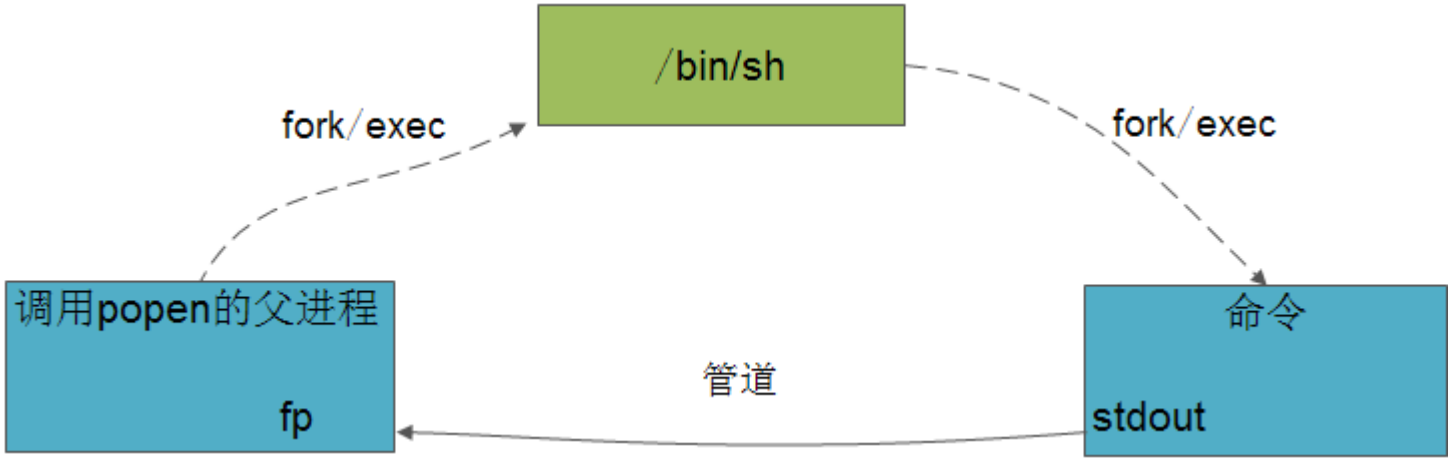
- 实现原理
  - 复制文件描述符：dup2
  - 一个程序的标准输出重定向到管道中
  - 而另一个程序的标准输入从管道中读取



无名管道(3)：
通过管道与shell命令进行通信

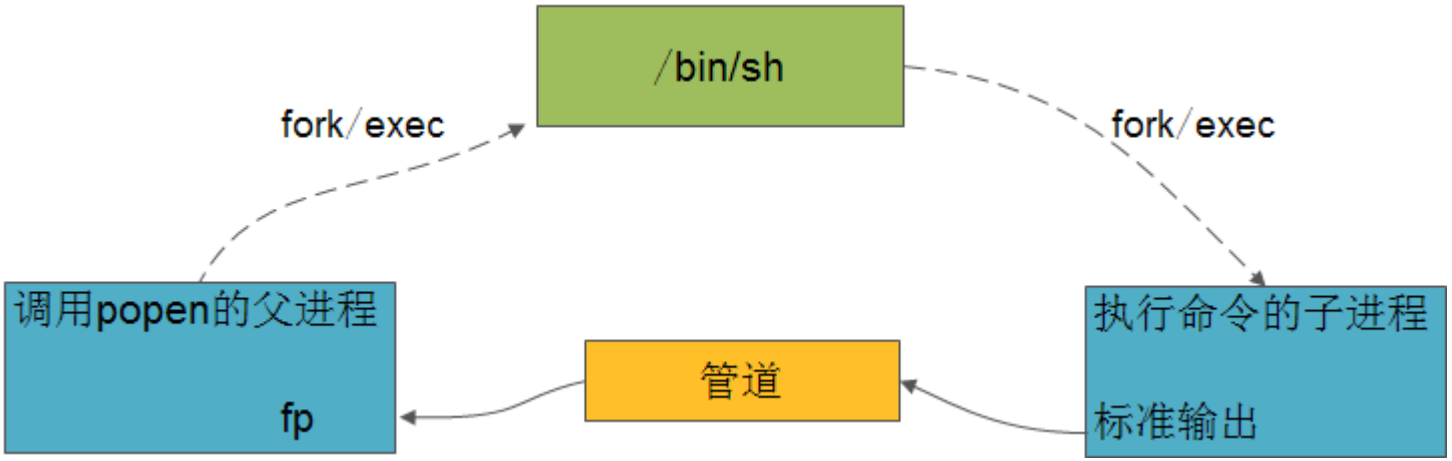
通过管道与shell命令进行通信

- popen函数
  - FILE *popen(const char *command, const char *type);
  - 创建一个管道，并创建一个子进程来执行shell，shell会创建一个子进程来执行command
  - 将父子进程的输入/输出重定向到管道，建立一个单向的数据流
  - 返回一个fp文件指针给父进程，父进程可根据fp对管道进行读写
  - 向管道中读数据：读命令的标准输出
  - 向管道中写数据：写入该命令的标准输入



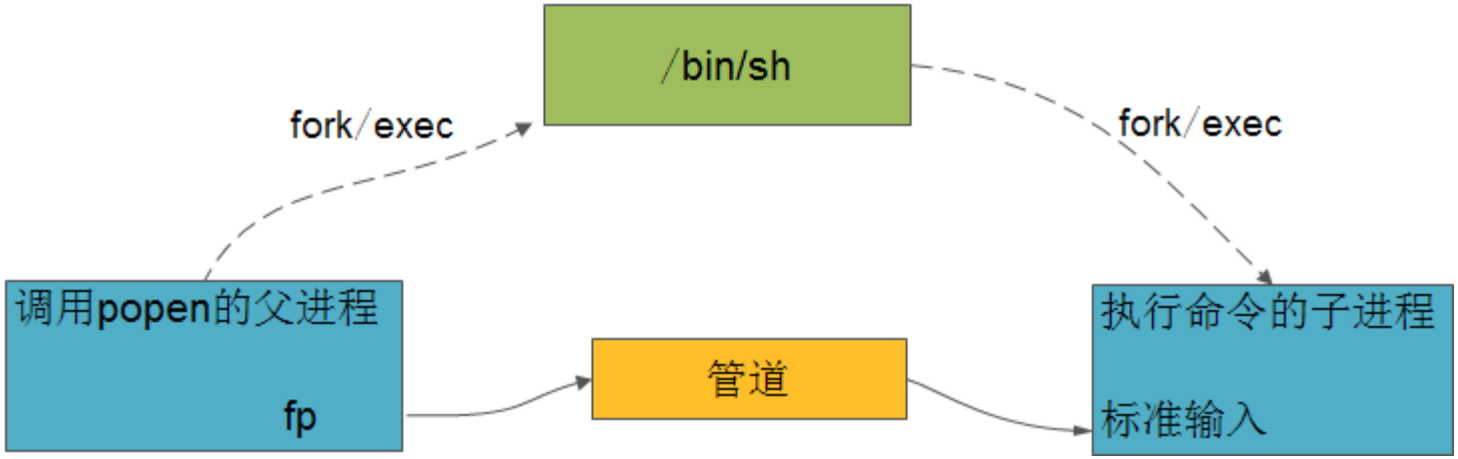
通过管道与shell命令进行通信

- 读模式调用popen
  - fp = popen(command, “r”);
  - popen函数创建子进程执行command，创建管道
  - 子进程的标准输出写入到管道，被调用popen的父进程读取。
  - 父进程通过对popen返回的fp指针读管道，获取执行命令的输出



通过管道与shell命令进行通信

- 写模式调用popen
  - fp = popen(command, “w”);
  - popen函数创建子进程执行command，创建管道
  - 调用popen的父进程，通过fp进行对管道进行写操作
  - 写入的内容通过管道传给子进程，作为子进程的输入



无名管道(4)：
通过管道同步进程

PIPE通信应用

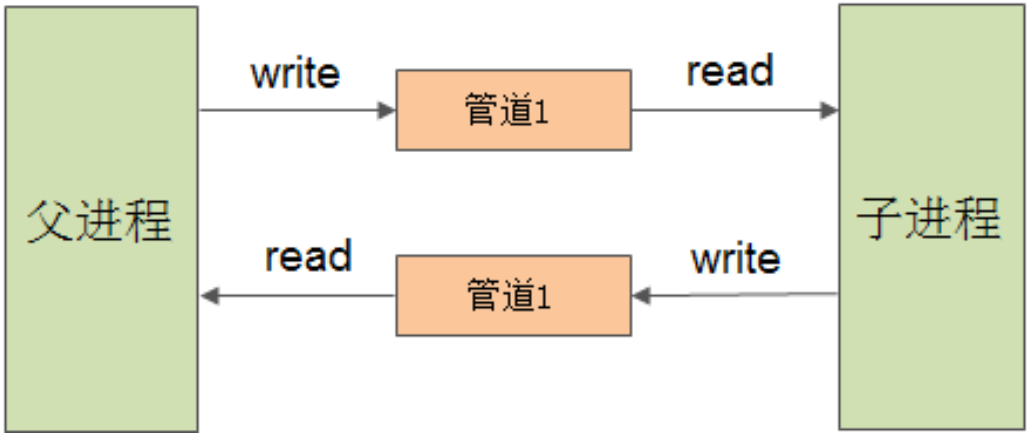
- 通过管道同步进程
  - 管道自带同步互斥机制：
    - 管道的内核实现：fs/pipe.c
    - 通过内核的锁、等待队列等机制实现
  - Write操作可能会阻塞进程
    - 当内存缓冲区已满或被读进程锁定
    - 直到所有数据被写入到管道为止
  - Read操作进程可能会阻塞进程
    - 读进程可以休眠在等待队列，
    - 直到所有子进程都关闭了管道的写入端描述符为止
    - 父进程的写入端描述符也要关闭，否则父进程读管道时也会被阻塞
    - 只有当所有的写端描述符都已关闭，且管道中的数据都被读出，对读端描述符调用read函数才会返回0（即读到EOF标志）
    - 当所有的读取端和写入端都关闭后，管道才能被销毁

管道缓冲区设置

- 管道缓冲区
  - 管道对应的内存缓冲区大小
    - PIPE_BUF的容量是有限的：默认是65536字节
    - 在不同OS下的PIPE_BUF大小设置不同：在 limits.h 头文件中定义
    - 写入管道的数据超过PIPE_BUF大小，内核会分割几块传输
    - 最大值 /proc/sys/fs/pipe-maxsize
    - 查看打开的管道文件：$ cat /proc/PID/fd
  - 设置缓冲区大小
    - 特权用户：可以修改上限值
    - 设置缓冲区大小：fcntl (fd, F_SETPIPE_SZ, size)

小结

- PIPE通信特点
  - 无名管道(匿名管道)是一个字节流
  - 可通过文件I/O接口读写、但无法lseek
  - 单向通信：一端用于写入、一端用于读出
  - 通信简单、性能单一、只能在近亲进程间通信



命名管道：FIFO(named pipe)

命名管道：FIFO

- FIFO通信特点
  - FIFO文件，有文件名字
    - 可以像普通文件一样存储在文件系统之中
    - 可以像普通文件一样使用open/read/write读写
    - 跟PIPE一样，属于流式文件，不能使用lseek定位
  - 具有写入原子性、可同时对FIFO进行写操作，如日志系统/var/log
  - First In First Out：最先被写入FIFO的数据，最先被读出来
    - 默认阻塞读、阻塞写的特性，可以在open的时候进行设置
    - 当一个进程打开FIFO的一端时，如果另一端没有打开，该进程会被阻塞

FIFO编程示例

- 系统调用接口
  - shell命令：mkfifo pathname
  - 函数接口： int mkfifo (const char *pathname, mode_t mode);
  - 函数功能：创建一个FIFO有名管道
  - 函数参数：
    - pathname：FIFO管道文件名
    - mode：读写权限
- 进程通信示例
  - 亲缘关系进程之间通信
  - 非亲缘关系进程之间的通信

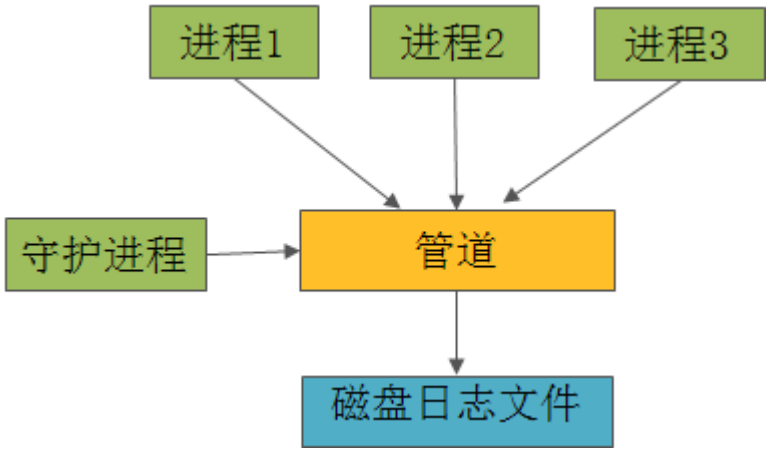
FIFO的内核实现

- FIFO与PIPE的区别和联系
  - 联系
    - 在内核中的实现：fs/pipe.c，本质上都是内存中的一块page cache
    - 通过向内核注册pipefs来实现，可以通过I/O接口read、write等访问
  - 区别
    - 匿名管道pipe通过自己的两端读写描述符进行读写操作
    - 命名管道有自己的文件名，可以通过文件名直接进行读写操作
    - 匿名管道pipe一般用于亲缘进程间通信
    - 命名管道FIFO可用于非亲缘进程间通信

FIFO应用：LOG日志系统的实现

FIFO通信应用

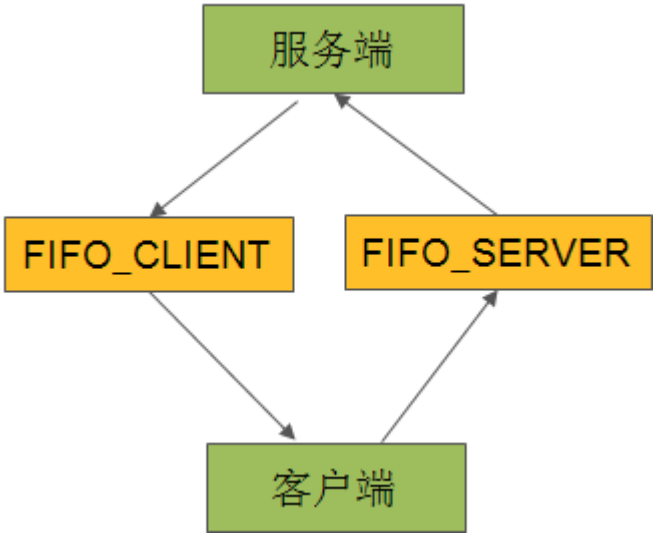
- Log日志系统
  - 各个进程往FIFO管道写入数据
  - 守护进程使用FIFO接收各个进程的输出日志信息
  - 并将FIFO中的数据写到对应的日志文件中



FIFO应用：服务端与客户端通信

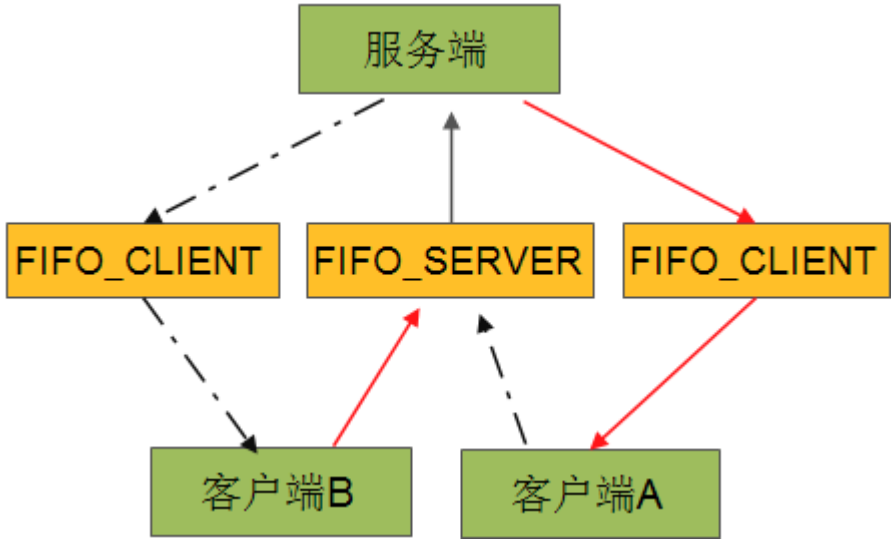
FIFO通信应用

- 服务器/客户端应用程序
  - 服务端、客户端进程通过FIFO实现双向通信



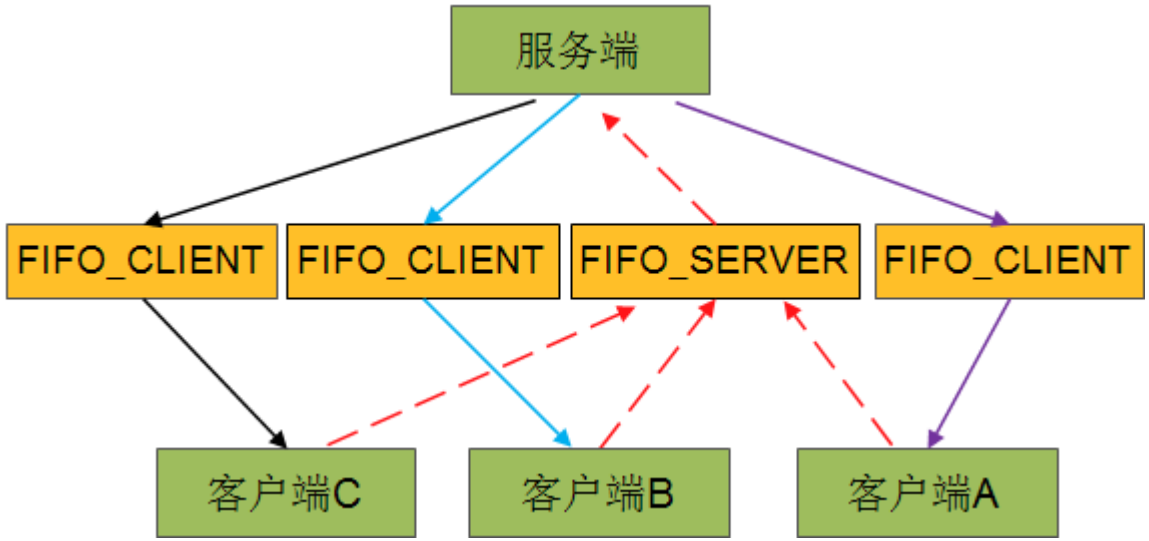
FIFO通信应用

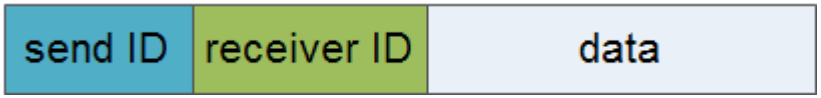
- 服务器/客户端应用程序
  - 2个客户端进程通过服务端实现双向通信



FIFO通信应用

- 服务器/客户端应用程序
  - 不同客户端进程通过服务端实现通信

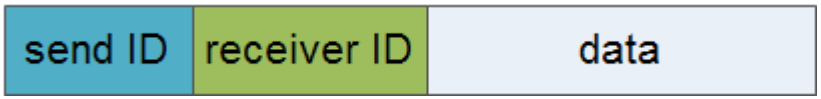




IPC 对象与IPC key

进程间通信对象：IPC

- 什么是IPC对象？
  - IPC：Inter-process communication
  - 管道通信：FIFO、PIPE，流式数据
  - 消息队列：message queue
  - 信号量：semaphore
  - 共享内存：share memory
  - …



IPC分类

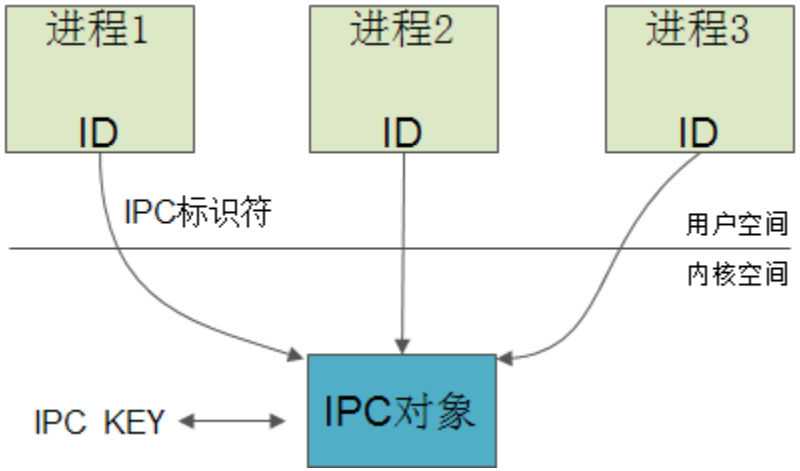
- System V IPC
  - 消息队列：system V message queue
  - 信号量： system V semaphore
  - 共享内存： system V share memory
- POSIX IPC
  - 消息队列：POSIX message queue
  - 信号量： POSIX semaphore
  - 共享内存： POSIX share memory

系统调用接口标准发展史

- System V 与 POSIX
  - UNIX/Linux的分支众多： AT&T公司的System V、BSD、Mac OS
  - System V release 4：简称SVR4，春秋一霸，1983年起发布的4个版本
  - System V 标准：在SV系统中被广泛使用，然后移植到大多数UNIX中，并加入到了各种标准中
  - POSIX：portable operating system interface， 可移植操作系统接口
  - POSIX.1标准：IEEE于1988年制定的第一个标准
  - POSIX后续标准：兼容了system V的大部分标准，扩展了自己的标准
  - 支持情况：
    - 大部分 Linux/Unix 都支持system V 和 POSIX标准接口
    - 有些Unix系统对POSIX标准的支持可能不太好

IPC对象

- IPC对象的基本概念
  - 不同进程通过IPC对象通信，IPC对象存储在内核中，全局可见
  - 每个IPC对象在内核中有自己的数据结构，定义在各自头文件中
  - 对IPC对象的引用
    - 普通文件：文件名 --- 文件描述符
    - IPC 对象 ： IPC key --- IPC 标识符



IPC标识符和IPC key

- 对IPC对象的标识
  - 类似于文件描述符，可通过一个IPC标识符来引用一个IPC对象
  - IPC标识符类似于文件描述符，是一个整数，是IPC对象的内部名
  - 当多个进程引用同一个IPC对象时，需要一个统一的外部命名。
  - 类似于文件名，每个IPC对象与一个键(key)相关联
  - IPC key，是IPC对象的外部名，是一个独一无二的整数，确保唯一性
    - 该键数据类型为key_t，在sys/types.h中被定义为长整型。
    - 普通文件：通过open打开一个文件名，获得文件描述符
    - IPC对象：通过get可根据给定的key去创建一个IPC对象，并返回IPC标识符

IPC key

- 创建IPC key的三种方法
  - 随机选取一个整数值作为key值。
    - 所有整数放到一个头文件中，使用IPC对象的程序包含这个头文件即可。
  - 在get系统调用中将IPC_PRIVATE常量作为key值。
    - 每个调用都会创建一个全新的IPC对象
    - 从而确保每个对象都拥有一个唯一的key
  - 使用ftok函数生成一个(接近唯一)key

IPC对象的引用

- 使用基本流程
  - 通过get系统调用创建或打开一个IPC 对象。
    - 给定一个整数key，get调用会返回一个整数标识符，即IPC 标识符
  - 通过这个标识符来引用IPC对象、进行各种操作
  - 通过ctl系统调用获取或设置IPC对象的属性、或者删除一个对象
  - IPC对象具有的权限定义在 /Linux/ipc.h文件中

接　口	消息队列	信号量	共享内存
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
关联数据结构	msqid_ds	semid_ds	shmid_ds
创建/打开对象	msgget()	semget()	shmget() + shmat()
关闭对象	（无）	（无）	shmdt()
控制操作	msgctl()	semctl()	shmctl()
执行 IPC	msgsnd()——写入消息 msgrcv()——接收消息	semop()——测试/调整信号量	访问共享区域中的内存

各种IPC对象的标识符和句柄

工 具 类 型	用于识别对象的名称	用于在程序中引用对象的句柄
管道	没有名称	文件描述符
FIFO	路径名	文件描述符
UNIX domain socket	路径名	文件描述符
Internet domain socket	IP 地址+端口号	文件描述符
System V 消息队列	System V IPC 键	System V IPC 标识符
System V 信号量	System V IPC 键	System V IPC 标识符
System V 共享内存	System V IPC 键	System V IPC 标识符
POSIX 消息队列	POSIX IPC 路径名	mqd_t (消息队列描述符)
POSIX 命名信号量	POSIX IPC 路径名	sem_t * (信号量指针)
POSIX 无名信号量	没有名称	sem_t * (信号量指针)
POSIX 共享内存	POSIX IPC 路径名	文件描述符
匿名映射	没有名称	无
内存映射文件	路径名	文件描述符
flock()文件锁	路径名	文件描述符
fcntl()文件锁	路径名	文件描述符

System V 消息队列

System V 消息队列

- 通信方法
  - 支持不同进程之间以消息(message)的形式交换数据。
  - 发送者
    - 获取消息队列的ID(IPC标识符)
    - 将数据放入一个带有标识的消息结构体，发送到消息队列
  - 接收者
    - 获取消息队列的ID
    - 将指定标识的消息从消息队列中读出，然后进一步后续处理

消息队列

- 编程接口
  - key_t ftok (const char *pathname, int proj_id);
  - int msgget(key_t key, int msgflg);
    - 创建或打开一个消息队列
    - 首先从既有消息队列中搜索与指定key对应的队列，返回该对象的标识符
    - 若没找到，msgflg指定了IPC_CREAT，则创建一个队列，并返回IPC标识符
  - int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
    - 发送消息到消息队列
  - ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
    - 从消息队列中接收信息
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
    - IPC_STAT：获取消息队列的属性信息
    - IPC_SET：设置消息队列的属性
    - IPC_RMID：删除消息队列

Linux中的消息队列

- 内核实现：消息队列模型
  - 相关数据结构：/usr/include/linux/msg.h、/ipc/msg.c
  - msqid_ds：标识整个消息队列的基本情况：消息队列权限、所有者、操作权限，和2个指针，分别指向消息队列中的第一和最后一个消息
  - msg：整个消息队列的主体，一个消息队列有若干个消息，每个消息数据结构的基本信息包括消息类型、消息大小、消息内容指针和下一个消息数据结构位置
  - 消息队列是消息的链表，存储在内核中，由消息队列标识符标识。
  - IPC标识符：消息队列的ID

消息队列应用：点对点通信

消息队列应用

- 不同进程之间的点对点通信
  - 不同进程之间通过各自指定的消息类型点对点通信
  - 不需要经过服务端“中转”分发，由内核充当“代理人”角色
  - 不同的进程可以操作同一个消息队列
  - 各自发送、接收自定义类型的消息，互不影响

消息队列应用：多人聊天室

消息队列应用

- 多用户本地聊天室
  - 支持多人同时聊天(3人以上、可以修改程序设置)
  - 每个用户端进程以ID登录，ID作为服务端要发送的消息类型
  - 服务端实现消息的广播转发功能

小结

- 消息队列与FIFO比较
  - 引用方式
    - 用来标识消息队列的是一个key，而不是普通文件所用的pathname
    - 用来引用消息队列的句柄是一个由msgget()调用返回的标识符。
    - 这些标识符类似于普通文件I/O通过open返回的文件描述符。
  - 传输的数据
    - FIFO发送的数据是流式数据、raw数据
    - 通过消息队列进行的通信是面向消息的
    - 除了包含数据之外，每条消息还有一个用整数表示的类型。
  - 消息队列比FIFO优越的地方
    - 消息队列双方通过消息通信，无需花费精力从字节流中解析出完整信息
    - 每条消息都有type字段，read进程可通过消息类型选择自己感兴趣的消息

小结

- 消息队列的优点
  - 降低系统耦合：
    - 生产者-消费者模式，自助餐模式，多个读写进程通过容器建立联系、互不影响，实现解耦
    - 消息是跟平台和语言无关的。
  - 提速系统性能：
    - 非核心流程异步化，非阻塞模式节省时间，不需要双方同时在线
  - 广播：
    - 一个消息可以发送给多个进程，只需要发送到队列就可以了
  - 削峰：
    - 生产者-消费者的负载平衡、秒杀活动

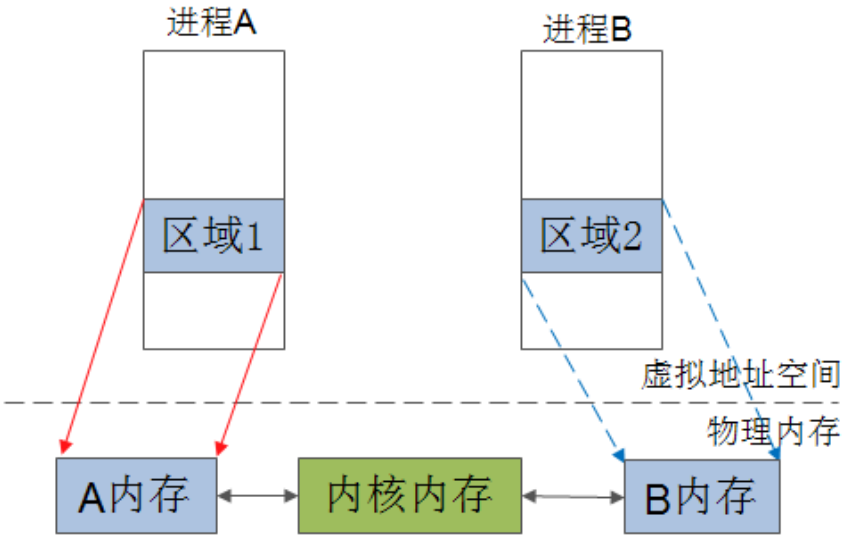
小结

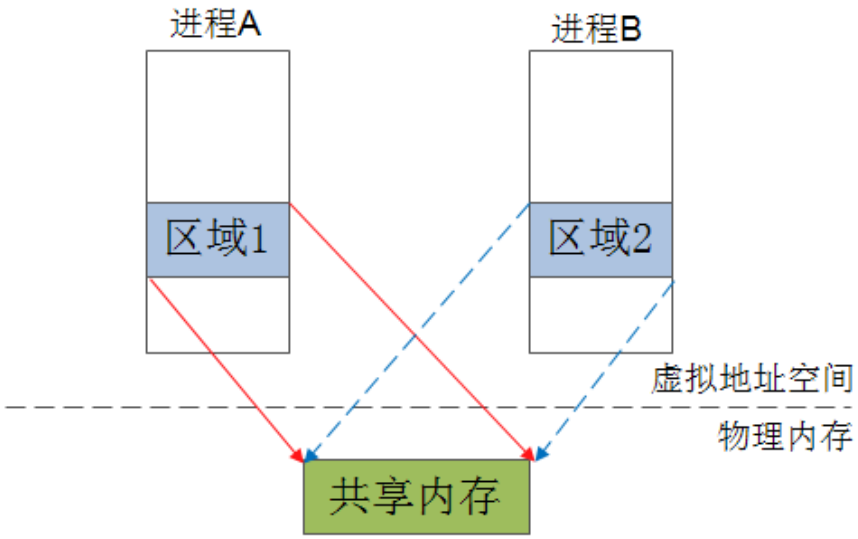
- 消息队列的缺陷
  - 效率低：“代理人”通信机制
  - 内核：为使用者分配内存、检查边界、设置阻塞、权限监控
  - 消息队列的的总数、消息的大小、单个队列的容量是有限制的
    - 注：Linux没有限制
    - MSGMNI：系统中所能创建的消息队列标识符
    - MSGMAX：单条消息中最多可写入的字节数
    - MSGMNB：一个消息队列中一次最多保存的字节数（mtext）
    - MSGTQL：系统中所有消息队列所能存放的消息总数
    - MSGPOOL：消息队列中用于存放数据的缓冲区的大小
  - 使用标识符而不是文件描述符来引用，使用键而不是文件名来标识消息队列，使用复杂
  - 消息队列是无连接的，内核不会像对待管道、FIFO、socket那样维护引用队列的进程数

system V 共享内存

System V 共享内存

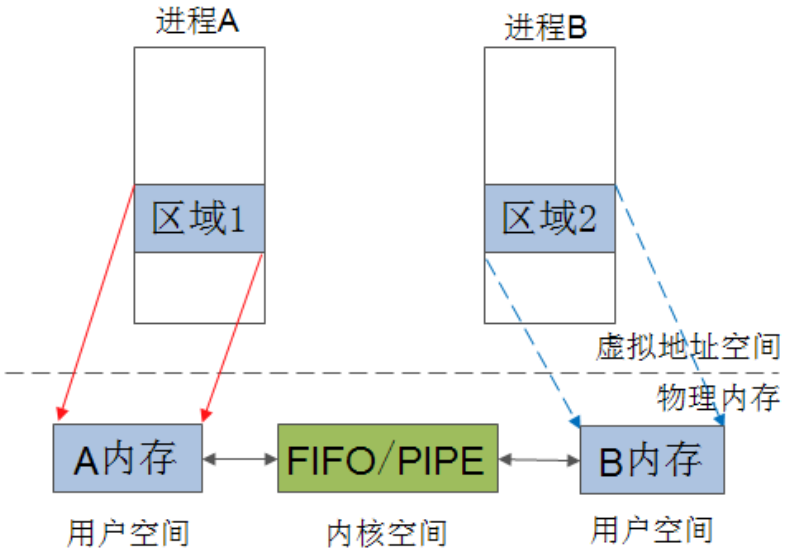
- 通信原理
  - 多个进程共享物理内存的同一块区域(通常被称为“段”：segment)
  - 抛弃了内核“代理人”角色，让两个进程直接通过一块内存通信

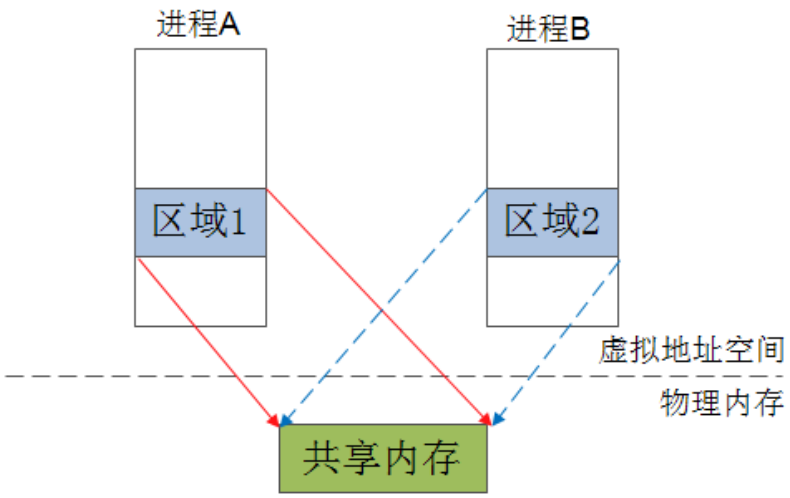




共享内存 VS 消息队列/管道

- 优势
  - 减少了内存拷贝(从用户拷贝到内核、从内核拷贝到用户空间)
  - 减少了2次系统调用，提高了系统性能





使用System V 共享内存

- 操作流程
  - 获取共享内存对象的ID
  - 将共享内存映射至本进程虚拟内存空间的某个区域
  - 不同进程通过对这块共享内存进行读写、传输数据
  - 当进程不再使用这块共享内存时，解除映射关系
  - 当没有进程再需要这块共享内存时，删除它

使用共享内存

- 相关API
  - 获取共享内存对象的ID： int shmget(key_t key, size_t size, int shmflg);
  - 映射共享内存： void *shmat(int shmid, const void *shmaddr, int shmflg);
  - 解除内存映射： int shmdt(const void *shmaddr);
  - 设置内存对象： int shmctl(int shmid, int cmd, struct shmid_ds *buf);
  - 查看IPC对象信息：$ ipcs -m

相关API

- 获取共享内存对象的ID
  - 函数原型： int shmget(key_t key, size_t size, int shmflg);
  - 函数功能：创建或打开一个共享内存对象
  - 所需头文件：sys/types.h　　sys/shm.h
  - 函数参数
    - Key：IPC对象的键值，一般为IPC_PRIVATE或ftok返回的key值
    - Size：共享内存大小，一般为内存物理页的整数倍
    - shmflg：
      - » IPC_CREAT：如果不存在与指定的key对应的段，那么就创建一个新段
      - » IPC_EXCL：若key指定的内存存在且指定了IPC_CREAT，返回EEXIST错误
      - » SHM_HUGETLB：使用巨页(huge page)
  - 返回值：共享内存的标识符ID

相关API

- attach共享内存
  - 函数原型： void *shmat (int shmid, const void *shmaddr, int shmflg);
  - 函数功能：将shmid标识的共享内存引入到当前进程的虚拟地址空间
  - 所需头文件：sys/shm.h
  - 函数参数
    - shmid ：共享内存的IPC对象 ID
    - shmaddr
      - » 若为NULL：共享内存会被attach到一个合适的虚拟地址空间，建议使用NULL
      - » 不为NULL：系统会根据参数及地址边界对齐等分配一个合适的地址
    - shmflg：
      - » IPC_RDONLY：附加只读权限，不指定的话默认是读写权限
      - » IPC_REMAP：替换位于shmaddr处的任意既有映射：共享内存段或内存映射
      - » SHM_RND：将shmaddr四舍五入为SHMMLBA字节的倍数
  - 返回值：共享内存段的地址

相关API

- detach共享内存
  - 函数原型：int shmdt(const void *shmaddr);
  - 函数功能：解除内存映射，将共享内存分离出当前进程的地址空间
  - 所需头文件：sys/shm.h
  - 函数参数
    - shmaddr：共享内存地址
  - TIPS：
    - 通过fork创建的子进程会继承父进程所附加的共享内存段，父子进程可以通过共享内存进行IPC通信。在exec系统调用中，所有附加的共享内存段都会被分离
    - 函数shmdt仅仅是使进程和共享内存脱离关系，将共享内存的引用计数减1，并未删除共享内存。
    - 当共享内存的引用计数为0时，调用shmctl的IPC_RMID命令才会删除共享内存

相关API

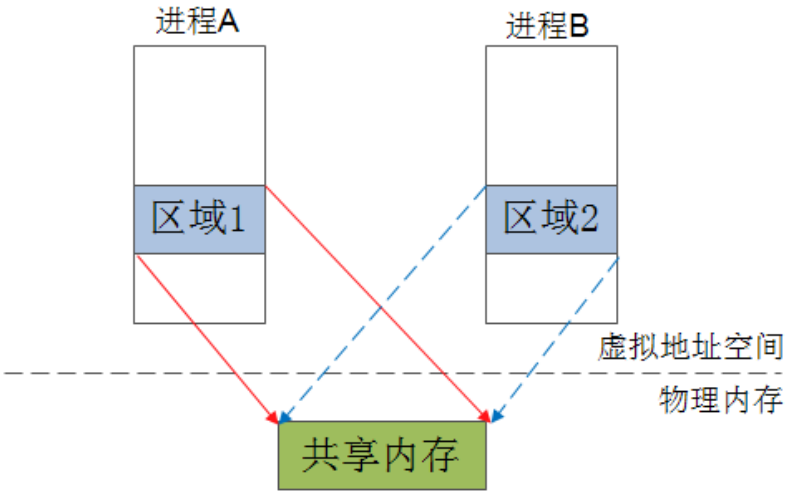
- 设置共享内存属性
  - 函数原型： int shmctl (int shmid, int cmd, struct shmid_ds *buf);
  - 函数功能：获取/设置 共享内存对象属性
  - 所需头文件：sys/shm.h
  - 函数参数
    - shmid:：共享内存的对象ID
    - cmd：
      - IPC_RMID：删除共享内存段及关联的shmid_ds数据结构
      - IPC_STAT：将该内存对象关联的shmid_ds数据结构拷贝到参数buf中
      - IPC_SET：使用buf中的数据更新与该共享内存对象相关联的shmid_ds
      - SHM_INFO：获取系统共享内存的相关信息
      - SHM_LOCK：将一个共享内存段锁进内存，防止被swap出去
      - SHM_UNLOCK：将一个共享内存段解锁

小结

- 共享内存的通信限制
  - SHMMNI：系统所能创建的共享内存的最大个数：IPCMNI 32768
  - SHMMIN：一个共享内存段的最小字节数 4096
  - SHMMAX：一个共享内存段的最大字节数 33554432
  - SHMALL：系统中共享内存的分页总数 2097152
  - SHMSEG：一个进程允许attach的共享内存段的最大个数

小结

- 共享内存通信特点
  - 共享内存抛弃了“内核代理人”角色，提升了系统性能
  - 需要进程本身维护共享内存的各种问题：同步、互斥...
  - 一般需要信号量、互斥锁、文件锁等配合使用，在各个进程之间在高效通信的同时，防止发生数据的践踏、破坏。



思考

- 内存映射mmap和共享内存shmat之间有什么关系？
  - 内存映射mmap可应用与IPC及其他很多方面

system V 信号量

信号量的基本概念

- 什么是信号量
  - 英文：semaphore，简称SEM，主要用来进程间同步
  - 本质：内核维护的一个正整数，可对其进行各种操作+/-操作
  - 分类：system V 信号量、POSIX 有名信号量、POSIX 无名信号量
  - 用途：用来标识系统中可用的共享资源的个数，协调各进程有序地使用这些资源，防止发生冲突
  - 信号量类似于酒店房间的房卡，房间资源是有限的、房卡也是有限的
  - P操作：程序在进入临界区之前要先对资源进行申请
  - V操作：程序离开临界区后要释放相应的资源，如房卡交给房东

信号量的基本概念

- 通信原理
  - 类似于房卡，不是单个值，而是一组(实际上是数组)信号量元素构成
  - 将信号量设置成一个绝对值
  - 在信号量当前值的基础上加上一个数量
  - 在信号量当前值的基础上减去一个数量，降到0以下可能会引起阻塞
  - 阻塞进程一直等待其它进程修改信号量的值，直到恢复正常运行
  - 信号量本身无意义，通常会与一块临界资源(如共享内存)关联使用

301	302	303	304
201	202	203	204
101	102	103	104

酒店房间

房卡：代表每个房间资源的信号量

使用system V 信号量

- 相关API
  - 获取信号量ID：int semget (key_t key, int nsems, int semflg);
  - P/V操作：int semop (int semid, struct sembuf *sops, size_t nsops);
    - 操作术语：荷兰语中的首字母，由荷兰计算机科学家Edsger Dijkstra确定
    - 其它操作术语：down(减小信号量)、up(增大信号量)
    - POSIX标准：wait、post
  - 信号量设置：int semctl (int semid, int semnum, int cmd, ...);

System V信号量相关API

- 创建或打开一个信号量
  - 函数原型： int semget (key_t key, int nsems, int semflg);
  - 包含头文件： sys/ipc.h sys/sem.h
  - 函数参数：
    - key：用来表示信号量的键，通常使用值IPC_PRIVATE或由ftok创建
    - nsems：信号的数量，所有的信号放在一个数组内
    - semflg：位掩码，用来设置信号量的权限或检查一个已有信号量的权限
      - » IPC_CREAT：如果找不到指定key相关联的信号量，创建一个新信号量集合
      - » IPC_EXCL：若指定了IPC_CREAT且指定key关联的信号量存在，报EEXIST错误
  - 函数返回值
    - 成功：返回用于操作信号量的句柄ID
    - 失败：-1，并设置errno全局变量

System V信号量相关API

- 信号量设置
  - 函数原型： int semctl (int semid, int semnum, int cmd, ...);
  - 包含头文件：
    - #include <sys/ipc.h>
    - #include <sys/sem.h>
  - 函数参数：
    - semid ：用于操作信号量的句柄ID、标识符
    - semnum ：信号的数量，所有的信号放在一个数组内
    - cmd：
      - IPC_RMID：删除信号量集及相关联的内核semid_ds数据结构
      - IPC_STAT：获取 semid_ds 副本
      - IPC_SET：设置 semid_ds 数据结构
      - GETVAL：获取信号集中第semnum个信号量的值
      - GETALL：获取所有的信号量的值
      - SETVAL：设置信号集中的第semnum个信号量的值
  - 函数返回值
    - 成功：根据cmd命令，返回不同的值
    - 失败：-1，并设置errno全局变量

System V信号量相关API

- 信号量P/V操作
  - 函数原型： int semop(int semid, struct sembuf *sops, size_t nsops);
  - 包含头文件：
    - #include <sys/ipc.h>
    - #include <sys/sem.h>
  - 函数参数：
    - semid ：用于操作信号量的IPC标识符
    - sops ：指向数组的指针，数组中包含了需要执行的操作
    - nsops ：数组的大小
  - 函数返回值
    - 成功：根据cmd命令，返回不同的值
    - 失败：-1，并设置errno全局变量

信号量值的P/V操作

- 结构体：sembuf
  - sem_num：用来标识要操作的信号集中的信号量
  - sem_op：
    - 若大于0：将sem_op的值加到信号量值上
    - 若等于0：对信号量值进行检查，确定其当前值是否为0，若为0操作结束，若不为0，则一直阻塞，直到信号量的值变为0为止
    - 若小于0：将信号量值减去sem_op。最后结果大于或等于0，操作立即结束；若最后结果小于0，则当前进程会阻塞
  - sem_flag：
    - SEM_UNDO
    - IPC_NOWAIT

```
struct sembuf {
        unsigned short  sem_num;    /* semaphore index in array */
        short           sem_op;     /* semaphore operation */
        short           sem_flg;    /* operation flags */
};
```

使用system V 信号量

- 使用流程
  - 使用semget创建或打开一个信号量集
  - 使用semctl SETVAL或SETALL操作初始化集合中的信号量(其中一个进程操作即可，内核中维护，对其它进程是全局可见的)
  - 使用semop操作信号量值。多个进程通过多信号量值的操作来表示一些临界资源的获取和释放
  - 当所有进程不再需要信号量集时，使用semctl IPC_RMID操作删除这个信号量集(其中一个进程操作即可)

信号量编程应用：
对共享内存的同步访问

system V 信号量编程应用

- 对共享内存的同步访问
  - 通过读、写信号量实现对共享内存的同步互斥访问
  - 实现一个二元信号量协议
    - 读信号量：当为1时，读进程才能进行P操作、读取数据，否则会阻塞
    - 写信号量：当为1时，写进程才能进行p操作、写入数据，否则会阻塞

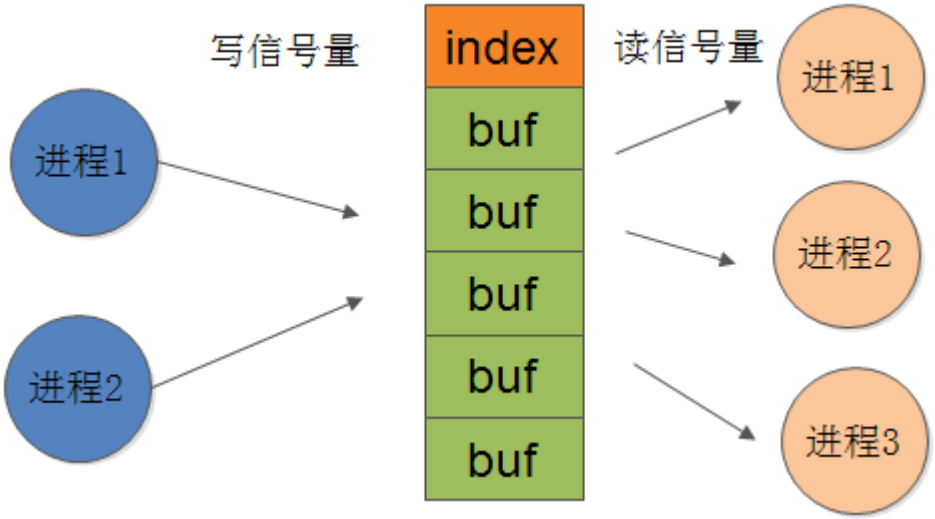
信号量编程应用：
生产者-消费者模型

编程示例

- 生产消费者模型
  - 有若干个缓冲区，生产者不断往里填数据，消费者不断从里面取数据
  - 如何使两者不产生冲突呢？
    - 缓冲区只有若干个，且有固定大小，而生产者和消费者则有多个进程
    - 生产者往缓冲区填数据前要判断缓冲区是否满了，满了就会等，直到有空间
    - 消费者从缓冲区拿数据之前要判断缓冲区是否为空，空了就会等，直到缓冲区内有数据为止
    - 在某一个时刻，缓冲区只允许有一个操作者进行读或写操作

编程示例

- 生产消费者模型
  - 2个生产者，每1S、5S往缓冲区写一次数据
  - 3个消费者，每2S、2S、5S往缓冲区读一次数据
  - 2个写进程分别对写信号量做P操作、对读信号量做V操作
  - 3个读进程分别对写信号量做V操作、对读信号量做P操作



小结

- System V 信号量的通信特点
  - 信号量是通过标识符而不是常用的文件描述符来引用的
  - 使用键而不是文件名来标识信号量
  - 创建和初始化信号量需要使用单独的系统调用
  - 内核不会维护引用一个信号量集的进程数量。很多操作需要开发者自己控制
  - 信号量的操作存在诸多限制

POSIX IPC简介

POSIX IPC对象编程接口

接　口	消息队列	信号量	共享内存
头文件	<mqueue.h>	<semaphore.h>	<sys/mman.h>
对象句柄	mqd_t	sem_t *	int（文件描述符）
创建/打开	mq_open()	sem_open()	shm_open() + mmap()
关闭	mq_close()	sem_close()	munmap()
断开链接	mq_unlink()	sem_unlink()	shm_unlink()
执行 IPC	mq_send(), mq_receive()	sem_post(), sem_wait(), sem_getvalue()	在共享区域中的位置上操作
其他操作	mq_setattr()——设置特性 mq_getattr()——获取特性 mq_notify()——请求通知	sem_init()——初始化未命名信号量 sem_destroy()——销毁未命名信号量	无

POSIX 与 system V IPC

- POSIX IPC
  - POSIX接口更简单：使用类似于文件I/O的open、close、unlink等接口
  - POSIX使用名字代替键来标识IPC对象
  - 对 IPC 对象引用计数，简化了对 IPC 对象的删除
    - 跟文件类似，删除操作也仅仅是删除了IPC对象的名字
    - 只有当IPC对象的引用计数变成0之后才真正销毁IPC对象
- System V IPC
  - System V IPC 可移植性更好：几乎所有的UNIX系统都支持system V，POSIX在UNIX系统中只是一个可选组件，有些UNIX系统并不支持
  - Linux系统一般都会支持system V
  - Linux 2.6开始陆续支持POSIX…

TIPS

- POSIX 编程注意事项
  - 使用POSIX 消息队列和共享内存时，需要实时库librt链接，编译时需指定 $ -lrt
  - 使用POSIX 信号量时，需要和线程库libpthread链接起来，编译时需指定$ -lpthread

POSIX 消息队列(上)：API编程实例

POSIX 消息队列

- 相关API
  - mq_open： 创建或打开一个消息队列
  - mq_send： 向消息队列写入一条消息
  - mq_receive：从消息队列中读取一条消息
  - mq_close： 关闭进程打开的消息队列
  - mq_unlink：删除一个消息队列
  - mq_setattr：设置消息队列一些额外的属性
  - mq_getattr：获取消息队列一些额外的属性
  - mq_nofity： 异步通知

POSIX消息队列的相关API

- 创建或打开 IPC 对象
  - 函数原型：
    - mqd_t mq_open (const char *name, int oflag);
    - mqd_t mq_open (const char *name, int oflag, mode_t mode,
      struct mq_attr *attr);
  - 函数功能：使用指定名字创建或打开一个对象，返回该对象的句柄
  - 函数参数:
    - name：用来标识要创建或打开的对象
    - Oflag：O_CREAT/O_EXCL /O_RDONLY /O_WRONLY /O_RDWR /O_NONBLOCK
    - Mode：位掩码，权限设置
    - Attr：设置消息队列的属性，若为NULL，使用默认属性。Linux3.5以后版本也可通过/proc查看设置
  - 函数返回值
    - 成功：返回消息队列的IPC对象描述符
    - 失败：返回-1，并设置errno

POSIX消息队列的相关API

- 关闭POSIX 消息队列
  - 函数原型：int mq_close(mqd_t mqdes);
  - 函数功能：通过描述符关闭消息队列
  - TIPS：
    - POSIX 消息队列在进程终止或执行exec()时会自动被关闭

POSIX消息队列的相关API

- 删除一个POSIX 消息队列
  - 函数原型：int mq_unlink (const char *name);
  - 函数功能：
    - 删除通过 name 标识的消息队列
    - 在所有进程使用完该队列之后销毁该队列。
    - 若打开该队列的所有进程已经关闭该队列，立即删除

POSIX消息队列的相关API

- 向 POSIX 消息队列写入消息
  - 函数原型：int mq_send(mqd_t mqdes, const char *msg_ptr,
    size_t msg_len, unsigned int msg_prio);
  - 函数功能：将msg_ptr指向的缓冲区中的消息添加到描述符mqdes所引用的消息队列中
  - 函数参数：
    - mqdes： 消息队列描述符
    - msg_ptr：指向存放消息的缓冲区指针
    - msg_len：消息的长度[10,8192]
    - msg_prio：消息对队列中按优先级排列，设置为0表示无需优先级

POSIX消息队列的相关API

- 从 POSIX 消息队列读取消息
  - ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
  - 函数功能：
    - 从mqdes引用的消息队列中删除一条优先级最高、存放时间最长的消息
    - 将删除的消息保存在msg_ptr指针指向的缓冲区
  - 函数参数：
    - mqdes： 消息队列描述符
    - msg_ptr：指向存放消息的缓冲区指针
    - msg_len：msg_ptr所指向的缓冲区长度，要大于消息队列的mq_msgsize
    - msg_prio：如不为空，接收到的消息的优先级会被复制到指针指向处
  - 函数返回值
    - 成功：返回接收的消息的字节数
    - 失败：-1，并设置errno

消息队列(中)：异步通知

POSIX消息队列：异步通知

- 异步通知API介绍
  - 函数原型：int mq_notify(mqd_t mqdes, const struct sigevent * sevp);
  - 函数功能：
    - 当空的消息队列到来消息时给进程发送一个通知
    - 当执行完相关处理，通知机制结束，可以重新调用mq_notify注册
  - 函数参数：
    - mqdes：消息队列的ID
    - sevp：通知方式设置

POSIX消息：异步通知

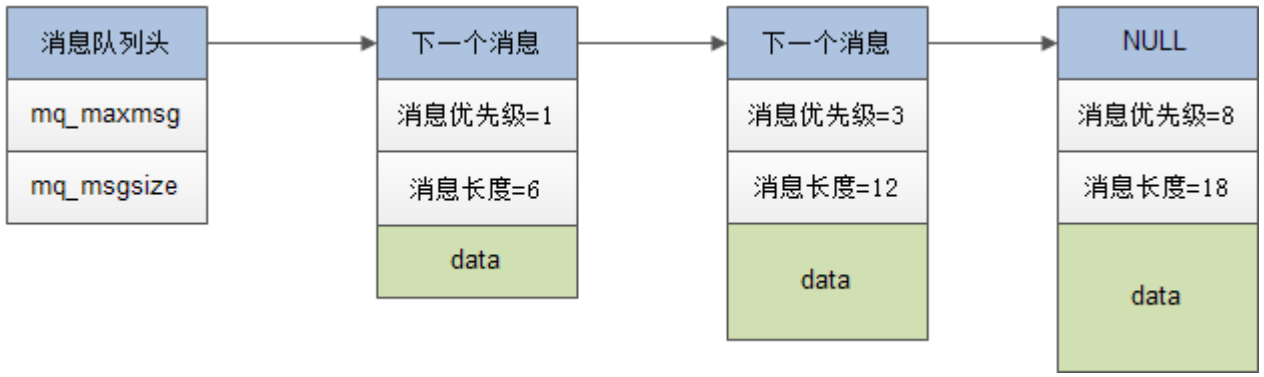
- 关键结构体：sigevent
  - sigev_notify
    - SIGEV_NONE：有通知时什么也不做
    - SIGEV_SIGNAL：给进程发送一个信号来通知进程
    - SIGEV_THREAD/ SIGEV_THREAD_ID
  - sigev_signo：要发送的信号

```
struct sigevent {
        int          sigev_notify;                /* Notification method */
        int          sigev_signo;                 /* Notification signal */
        union sigval sigev_value;                 /* Data passed with notification */
        void (*sigev_notify_function) (union sigval);/* Function used for thread notification (SIGEV_THREAD) */
        void         *sigev_notify_attributes;    /* Attributes for notification thread (SIGEV_THREAD) */
        pid_t        sigev_notify_thread_id;      /* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

消息队列(下)：内核实现

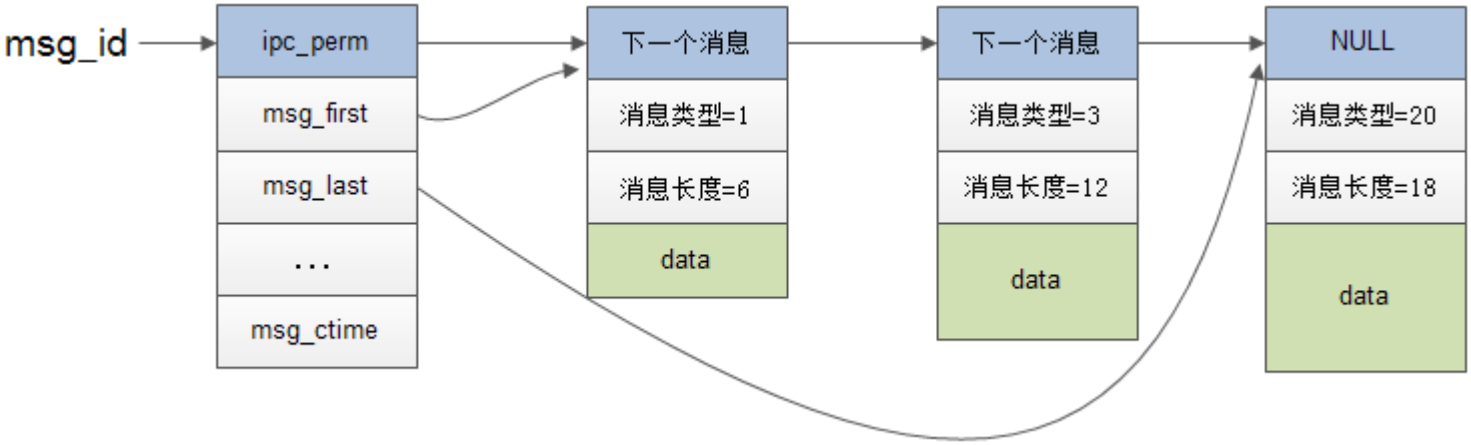
Linux内核中的消息队列

- POSIX消息队列内核实现
  - 相关数据结构：/usr/include/linux/mqueue.h、/ipc/mqueue.c
  - 消息队列是消息的链表，存储在内核中，由消息队列标识符标识。
  - 标识符成为消息队列的ID，程序通过这个句柄可以操作消息队列
  - 消息属性：
    - 一个无符号整数优先级
    - 消息的数据长度(可以为0)
    - 消息的数据本身



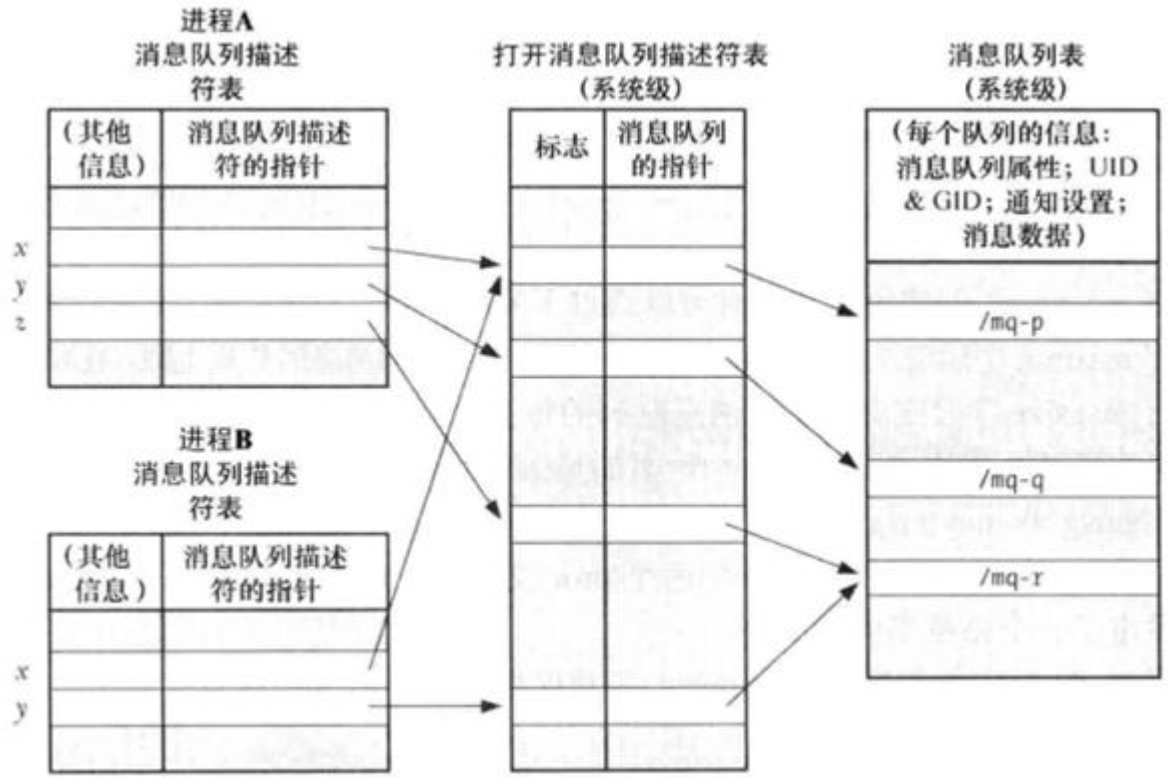
Linux内核中的消息队列

- system V消息队列内核实现
  - 相关数据结构：msgid_ds、/usr/include/linux/msg.h、/ipc/msg.c
  - 消息队列是消息的链表，存储在内核中，由消息队列标识符标识。
  - 标识符成为消息队列的ID，程序通过这个句柄可以操作消息队列
  - 消息属性：
    - 一个长整数类型(System V)
    - 消息的数据长度(可以为0)
    - 消息的数据本身



消息队列描述符

- 内核中的消息队列描述符
  - 用来标识打开的消息队列，类似于文件描述符，用来标识打开的文件
  - 是一个进程级句柄，在内核中实现类似于文件描述符



消息队列的属性

- 查看设置消息队列的属性
  - 通过proc文件系统
    - $ cat /proc/sys/fs/mqueue/queues_max
    - $ cat /proc/sys/fs/mqueue/msg_max
    - $ cat /proc/sys/fs/mqueue/msgsize_max
  - 通过POSIX系统调用接口
    - mq_setattr：设置mq_flags
    - mq_open：设置mq_maxmsg、mq_msgsize

Linux特性

- POSIX消息队列在Linux上实现的特性
  - 提供了mqueue类型的虚拟文件系统
    - 使用另一种I/O模型操作消息队列、获取消息队列的相关信息
    - 可通过挂载、ls、rm命令来列出和删除消息队列
    - $ mkdir /dev/mqueue
    - $ mount -t mqueue none /dev/queue
    - $ cat /dev/mqueue/my_mqueue
    - $ hexdump /dev/mqueue/my_mqueue

POSIX与system V 消息队列

- 区别和联系
  - POSXI消息队列通过设置优先级，总是返回优先级最高的最早消息
  - System V 消息队列可以通过消息类型返回指定优先级的任意消息
  - POSIX消息队列可以实现异步事件通知
    - 当有一个消息放置到某个空消息队列中时，这种通知有两种方式可以选择：产生一个信号，或者创建一个线程来执行一个指定的函数
    - msgrcv函数接受信息时，若队列为空会阻塞，若设置了NOBLOCK标志，则会不停地调用msgrcv轮询是否有消息到来，非常消耗CPU资源

小结

- POSIX 消息队列的优势
  - 允许一个进程能够在一条消息进入之前的空队列时异步地通过信号或线程的实例化来接收通知
  - 在Linux上可以使用poll、select、epoll来监控POSIX消息队列
  - POSIX的可移植性很差
  - POSIX消息队列严格按照优先级排序，而system V可以根据类型来选择消息，灵活性更强

POSIX 信号量

POSIX 信号量

- 相关的API
  - sem_t *sem_open (const char *name, int oflag);
  - sem_t *sem_open (const char *name, int oflag,mode_t mode, unsigned int value);
  - int sem_close (sem_t *sem);
  - int sem_post (sem_t *sem);
  - int sem_wait (sem_t *sem);
  - int sem_trywait (sem_t *sem);
  - int sem_timedwait (sem_t *sem, const struct timespec *abs_timeout);
  - int sem_unlink (const char *name);
  - int sem_getvalue (sem_t *sem, int *sval);

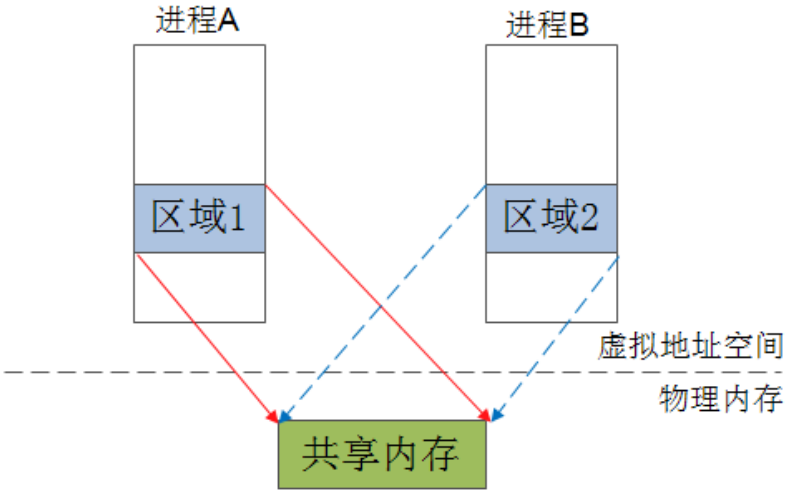
POSIX信号量

- 使用指南
  - 包含头文件：#include <semaphore.h>
  - 编译时要指定：-lpthread
  - Pthread：
    - POSIX threads，操作线程的API标准
    - 适用于 Unix、Linux、Mac OS

POSIX 共享内存

POSIX共享内存

- 相关的API函数
  - int shm_open (const char *name, int oflag, mode_t mode);
  - int shm_unlink (const char *name);
  - int ftruncate (int fd, off_t length);
  - int fstat (int fd, struct stat *buf);
  - void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
  - int munmap (void *addr, size_t length);



文件锁

共享内存

- 优点
  - 在所有的IPC通信中效率最高
- 缺点
  - 所有共享内存带来的问题：同步
  - 解决方法
    - System V 信号量
    - POSIX 信号量
    - 文件锁：
      - 专门为文件设计的同步技术
      - 适用于操作共享文件映射

文件锁

- 什么是文件锁
  - 英文名：file lock，在同一时刻只允许一个进程对文件进行访问
  - 建议性锁：advisory locking，又称协同锁
    - 内核只提供加减锁以及检测是否加锁，不提供锁的控制与协调工作
    - 需要多进程相互检测确认的加锁机制
    - A进程对一个操作的文件加了锁
    - B进程同样可以对该文件进行读写操作
    - 只有当B进程也对该文件加锁，文件锁才能起到同步作用
    - Linux一般使用建议锁，而Windows一般使用强制性锁
  - 强制性锁： mandatory locking
    - 进程对文件进行I/O操作是，内核内部会检测该文件是否被加锁
    - A进程对一个操作的文件加了锁
    - 当B进程对该文件进行I/O操作时，内核若检测该文件加了强制锁，B进程的操作则会失败

文件锁使用接口

- 系统调用：flock
  - 函数原型：int flock (int fd, int operation);
  - 函数功能：给整个文件添加或解除一个建议锁
  - 函数参数：operation
    - LOCK_SH：共享锁
    - LOCK_EX：独占锁、排他锁
    - LOCK_UN：移除锁
  - TIPS
    - Flock只提供加锁、解锁机制，不提供锁检查
    - 需要用户自己检测，达到多进程同步操作
    - 用户若不自己检测，同样可以对一个已经加锁的文件进行读写操作

文件锁使用接口

- 系统调用：fcntl
  - 函数原型： int fcntl (int fd, int cmd, ... /* arg */);
  - 函数功能：给文件(部分文件)进行加锁、解锁操作
  - 函数参数：cmd
    - F_SETLK：非阻塞式申请锁
    - F_SETLKW：阻塞式申请锁
    - F_GETLK：获取锁的相关信息
  - 记录锁
    - 读锁F_RDLCK，写锁F_WRLCK，释放锁F_UNLCK

文件锁使用接口

- 系统调用：lockf
  - 函数原型：int lockf (int fd, int cmd, off_t len);
  - 函数功能：
    - 可以更细粒度地对文件进行加锁、解锁操作
    - 库函数lockf是对系统调用fcntl的封装
  - 函数参数：operation
    - F_LOCK：对文件某一区域添加独占锁
    - F_TLOCK：非阻塞式申请锁
    - F_ULOCK：对文件某一区域解锁

作业

- POSIX共享内存+ 文件锁

信号机制：signal

信号的基本概念

- 信号(signal)
  - 又叫：软中断信号，是一种异步通信的IPC
  - 类似于硬件中断，可以将一个事件以信号形式通知给进程
  - 给一个指定进程发送一个信号
    - 信号只是告诉进程发生了什么事，并不传递数据
    - 进程表的表项中有一个软中断信号域，有信号发给该进程，对应位置位
    - 进程根据接收信号类型作相应的处理

信号的基本概念

- 信号的来源
  - 来自shell终端用户输入的各种信号：ctrl + C/D
  - 来自其它进程或者进程本身发送的信号
  - 来自系统内部的信号
    - 硬件异常：如SIGBUS表示总线错误、SIGSEGV表示段错误
    - 终端相关的信号
    - 软件事件相关的信号

内核对信号的处理

- 一个进程对信号的处理方式
  - 缺省行为
    - 忽略信号：如SIGIGN、SIGCHLD
      - SIGKILL/SIGSTOP比较特殊，不能忽略，所有进程都要在OS管控之下
    - 终止进程：SIGTERM、SIGINT、SIGHUP
    - 终止进程并内核转储：SIGBUS、SIGABRT、SIGQUIT
  - 捕获信号并执行信号注册的handler
    - 通过signal系统调用可以改变信号的处理行为，即注册新的handler
    - 当有信号到来时，信号的处理类似于中断程序
    - 暂停当前进程正在执行的代码、跳到注册的回调函数handler执行
    - 函数返回，回到当前进程捕获信号的地方继续执行
    - 若该信号没有注册回调函数，采用默认操作：忽略或终止进程

信号系统调用接口

- 信号相关API
  - typedef void (*sighandler_t)(int);
  - sighandler_t signal (int signum, sighandler_t handler);
  - int kill (pid_t pid, int sig);
    - 通过signal注册信号处理函数
    - 进程之间通过kill发送软中断信号
    - 内核也可以因内部异常等事件给进程发信号

注册信号处理函数

- 系统调用：signal
  - 函数原型：sighandler_t signal (int signum, sighandler_t handler);
  - 函数功能：注册一个信号处理函数
  - 函数参数
    - signum：信号值，定义在：asm/signal.h 头文件中，很多信号跟体系相关
    - handler：信号对应的处理函数
    - Linux支持的信号列表

信号相关的系统调用

- 系统调用：kill
  - 函数原型：int kill (pid_t pid, int sig);
  - 函数功能：给指定进程发送一个信号
- 系统调用：pause
  - 函数原型：int pause (void);
  - 函数功能：将当前进程挂起睡眠，等待某一个信号，直到信号到来，恢复运行
  - 返回值：该函数总是返回-1

定时发送信号

- 系统调用：alarm
  - 函数原型：unsigned int alarm(unsigned int seconds);
  - 函数功能：给当前进程在指定的seconds秒后发送一次SIGALRM信号
- 系统调用：setitimer
  - int getitimer (int which, struct itimerval *curr_value);
  - int setitimer (int which, const struct itimerval *new_value,
    struct itimerval *old_value);
  - 函数功能：获取定时器状态、设置定时器，周期发送信号
  - 函数参数：which，指定三个内部定时器中的一个
    - ITIMER_REAL：按实际时间计时，计时到达给进程发送SIGALRM信号
    - ITIMER_VIRTUAL：当进程执行时才计时，到期后发送SIGVTALRM信号
    - ITIMER_PROF：当进程执行或系统为该进程执行动作时都计时，如统计进程在用户态和内核态所花的时间，到期后发送SIGPROF信号给进程

编写安全的信号处理函数

信号处理函数

- 信号的本质
  - 是一种软中断，中断有优先级，信号也有优先级
  - 信号处理函数类似于中断处理函数
  - 信号也可以随时打断当前正在运行的进程，去运行信号处理函数

信号处理函数

- 编程要点
  - 重入：可能在任何时刻、任意地点打断当前进程的执行
  - 尽量不要在处理函数中修改全局数据
  - 尽量使用可重入函数，被打断的进程可能正在调用不可重入函数
  - 难点：很难写出一个安全地、可重入的信号处理程序
  - int sum ;
  - int add (int count)
  - {

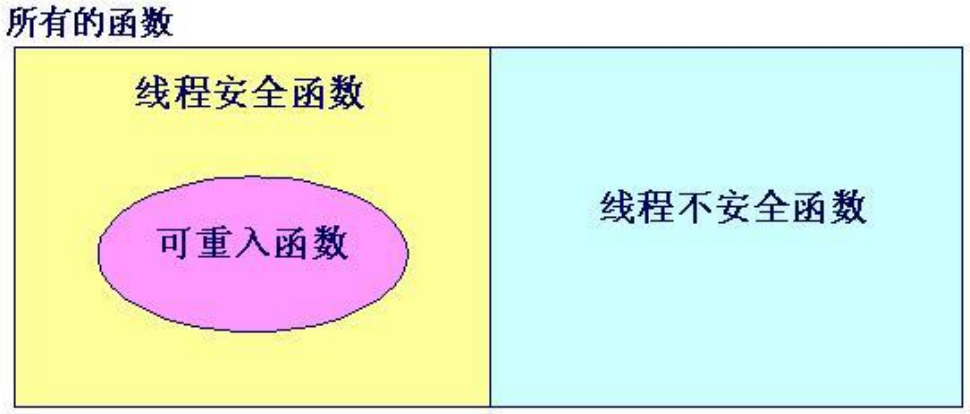
    static int sum = 0;

    For （i from 0 to count）

    sum = sum + i;
  - }

可重入函数

- 可重入与线程安全
  - 可重入函数一定是线程安全的
  - 不可重入函数通过加锁访问全局变量，也是线程安全的，但仍是不可重入的
  - 如果一个函数对于信号处理来说是可重入的，则称其为异步信号安全函数，可重入函数跟信号安全函数可以看做等价的
  - 线程安全的函数，不一定是异步信号安全的



可重入安全函数列表

表 10-3　在信号中确保可重入的安全函数列表

abort()	accept()	access()
aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
clock_gettime()	close()	connect()
creat()	dup()	dup2()
execle()	execve()	_Exit()
_exit()	fchmod()	fchown()
fcntl()	fdatasync()	fork()
fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()
getpgrp()	getpid()	getppid()
getsockname()	getsockopt()	getuid()
kill()	link()	listen()
lseek()	lstat()	mkdir()
mkfifo()	open()	pathconf()
pause()	pipe()	poll()

可重入安全函数列表

posix_trace_event()	pselect()	raise()
read()	readlink()	recv()
recvfrom()	recvmsg()	rename()
rmdir()	select()	sem_post()
send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()
setsockopt()	setuid()	shutdown()
sigaction()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()
sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()
socketpair()	stat()	symlink()
sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	timer_getoverrun()	timer_gettime()
timer_settime()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

思考

- 是不是一定不能调用不可重入函数
- 是不是一定不能调用信号不安全的函数

信号底层API：sigaction

Linux中的信号分类

- 标准信号及其不可靠性
  - 标准信号
    - 1~31号信号，也叫不可靠信号，继承UNIX信号，采用位图管理
    - 如果同时来相同的信号来不及处理，内核会丢弃掉
  - 实时信号
    - 32~64号信号，是可靠的，采用队列管理
    - 来一次，处理一次，转发一次

信号处理机制

- 内核对信号的处理
  - A进程向B进程发送一个信号，内核会首先收到该信号，然后发给B进程，在发送给B进程之前，内核负责管理这些信号
  - 对于不可靠信号，内核采用位图标记，给该信号分配sigqueue结构体，挂入链表之中，并将位图中的对应位置一；此时若有相同的信号发来，因为对应位已经置一，因此内核会丢弃该信号
  - 对于可靠信号，内核采用队列管理：给该信号分配一个sigqueue结构体，并挂入到链表队列之中
  - 队列中信号的个数也是有限制的，超过默认值，可靠信号也会丢失，也就变得不可靠了。

信号底层API：sigaction

- 信号底层注册函数
  - 函数原型：int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);
  - 函数功能：给信号设置新的注册函数act，同时保留原有的信号处理函数在oldact
    - 执行某些信号时屏蔽某些信号，直接给sa_mask赋值即可
    - 处理带参数的信号
    - 一次注册，长期有效

```
struct sigaction
        {
            void        (*sa_handler)(int);
            void        (*sa_sigaction)(int,  siginfo_t *,  void *);
            sigset_t    sa_mask;
            int         sa_flags;
            void        (*sa_restorer)(void);
         };
```

高级信号管理

- 新的信号发送函数
  - 函数原型：int sigqueue (pid_t pid, int sig, const union sigval value);
  - 函数功能：
    - 用法与kill类似
    - 与kill不同之处：kill可以将 pid 设置为指定负值，向整个进程发送信号
    - 可以给指定进程传递一个int型数据
    - sigaction 和 sigqueue 是一对 CP

小结

- 一旦给信号安装了handler，它就一直有效
- 信号的handler存在并发访问、可重入问题
- 在信号的handler运行期间，会阻塞掉当前本身该信号
- 在handler运行期间，当前信号的多次提交可能被丢弃，只保留一次
- 除了本身信号是被阻塞的，可以通过设置，阻塞设定的一些信号
- signal是标准C定义的函数，而sigaction是POSIX接口函数
- Signal是对sigaction的封装
- 不同的架构、操作系统对信号的value、default action 可能不一样
- 特殊的两个信号：SIGKILL和SIGSTOP
  - 不能被忽略
  - 不能安装signal handler、不能被捕捉
  - 不能被阻塞

Linux新增API：signalfd

信号通信机制小结

- 优缺点
  - 在软件层次上是对中断机制的一种模拟
  - 信号是进程间通信中唯一的“异步通信机制”
  - 带来的弊端：数据的并发访问、可重入问题
  - 解决方案：将信号抽象为文件，将信号转化为I/O文件操作

Linux新增API

- signalfd
  - 将信号抽象为一个文件描述符
  - 将信号的异步处理转换为文件的I/O操作
  - 当有信号发生时，可以对其read
  - 每次read都会阻塞、直到signalfd指定的信号到来
  - 也可以将信号的监听放到select、poll、epoll等监听队列中

Linux新增API

- signalfd
  - 函数原型： int signalfd(int fd, const sigset_t *mask, int flags);
  - 函数功能：创建一个可以对信号进行I/O访问的文件描述符
  - 函数参数：
    - mask：进程想通过文件描述符接收的信号集
    - flags：
      - » SFD_NONBLOCK：
      - » SFD_CLOEXEC：

Linux新增API：timerfd

Linux新增API

- timerfd
  - 函数功能：通过文件I/O方式去获取定时器的通知事件
  - 相关API：
    - int timerfd_create(int clockid, int flags);
    - int timerfd_settime(int fd, int flags, const struct itimerspec *new_value, struct itimerspec *old_value);
    - int timerfd_gettime(int fd, sturct itimerspec *curr_value);

Linux新增API：eventfd

进程间等待通知机制

- eventfd API
  - int eventfd(unsigned int initval, int flags);
    - 用来创建一个用于事件通知的eventfd对象
    - 返回值为一个int型fd，用来引用打开的对象
    - 打开的对象在内核中是一个64位的无符号整型计数器，初始化为initval
    - 计数器数据类型：typedef uint64_t eventfd_t
  - flags:
    - EFD_CLOEXEC：类似于open的flags，fork子进程时不继承
    - EFD_NONBLOCK：一般会设置成非阻塞
    - EFD_SEMAPHORE：信号量语义的read，每次读操作，计时器值减1
  - 内核实现：fs/eventfd.c

进程间等待通知机制

- eventfd操作API
  - read/eventfd_read
    - 读操作将64位的计数器置0
    - 如果有EFD_SEMAPHORE标记，计数器值减1
    - 如果计数器值为零，继续读的话，可能会阻塞或非阻塞
  - write/eventfd_write：
    - 设置计数器的值

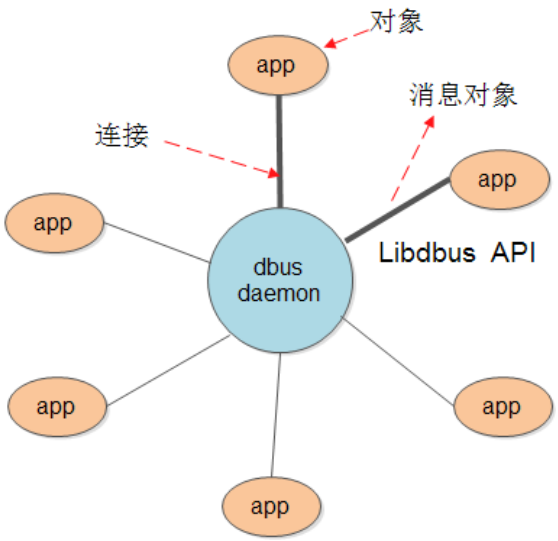
D-BUS总线简介及小结

D-BUS总线基本概念

- 什么是D-BUS？
  - 针对桌面环境，用于本地进程间通信的一种IPC机制
  - 主要用于同一桌面会话中，不同桌面应用程序之间的通信
  - 同时支持桌面会话与操作系统之间的通信
  - 系统总线：system bus，用于内核和应用进程之间的通信和消息传递
  - 会话总线：session bus，用于桌面(GNOME/KDE等)用户进程之间的通信。D-BUS一般由一个系统总线和几个会话总线构成
  - 总线权限：只有Linux内核、Linux桌面环境和权限较高的应用进程才有权限向系统总线写入消息，安全性高

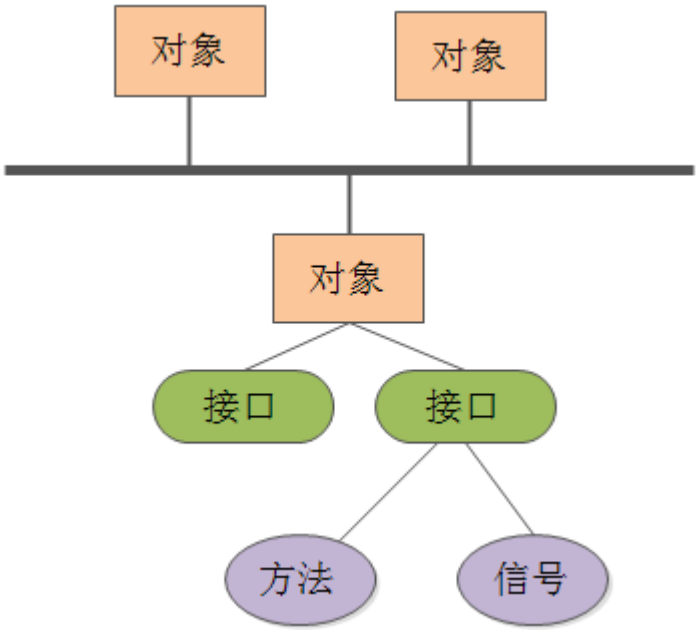
D-BUS总线基本概念

- D-BUS组成
  - Libdbus
    - 点对点的通信支持库，提供C语言API
    - 不同进程可引用库的API进行通信
  - dbus daemon
    - D-BUS服务进程，基于libdbus，作用类似于总线
    - 不同进程可以通过API连接它，发送和接收消息
    - 支持一对一、一对多的通信
  - 基于libdbus的封装库或框架：
    - Libdbus-glib、libdbus-qt
    - 通过dbus binding，支持多种语言：C/C++、Java/Python、Perl、Ruby…
    - 只要应用程序兼容dbus通信协议，可以支持任何语言



D-BUS总线基本概念

- 对象
  - 每个使用D-BUS的进程都包括一组对象，消息发送到或者发送自某一个对象，这些对象由对象路径唯一标识
  - D-BUS本质上是一个对等(p2p)协议：每个消息都有一个源和目的，这些地址被称为对象路径
  - 每个对象支持一个或多个接口，一个接口是多个方法和信号的集合



D-BUS在行业中的应用

- 在PC桌面环境中的应用
  - 给Linux桌面环境(GNOME、KDE等)提供的服务标准化
  - Android中的Dbus
  - Qt中的Dbus
  - 越多越多的GUI开始支持/兼容Dbus…
  - 例：Ubuntu16.04桌面环境
    - Systemd—lightdm—upstart
      - » Dbus-daemon
      - » Dcanf-service
      - » Gnome-session-b
      - » Gnome-terminal-bash-su

IPC进程间通信小结

- 各种工具的比较
  - 无名管道：只能用于亲缘进程通信
  - 有名管道：可用于任意两进程间通信，但只能传输流数据、缓冲区大小受限
  - 消息队列：可以传输有格式字节流，但是效率低：系统调用产生的用户空间、内核空间转换的开销
  - 共享内存：通信效率最高最快，解决了进程间通信运行效率低等开销问题，但是可能会带来同步问题
  - 信号量：用来不同进程、线程之间的同步，与共享内存结合使用
  - 文件锁：可以对整个文件、或者文件的一部分区域进行加锁
  - 信号：唯一的异步通信、但是存在一系列的问题
  - Linux特有API：将异步通信操作转换为I/O操作
  - Dbus：桌面进程之间的通信
  - 套接字：适用于不同机器进程间的通信，目前应用最广泛的