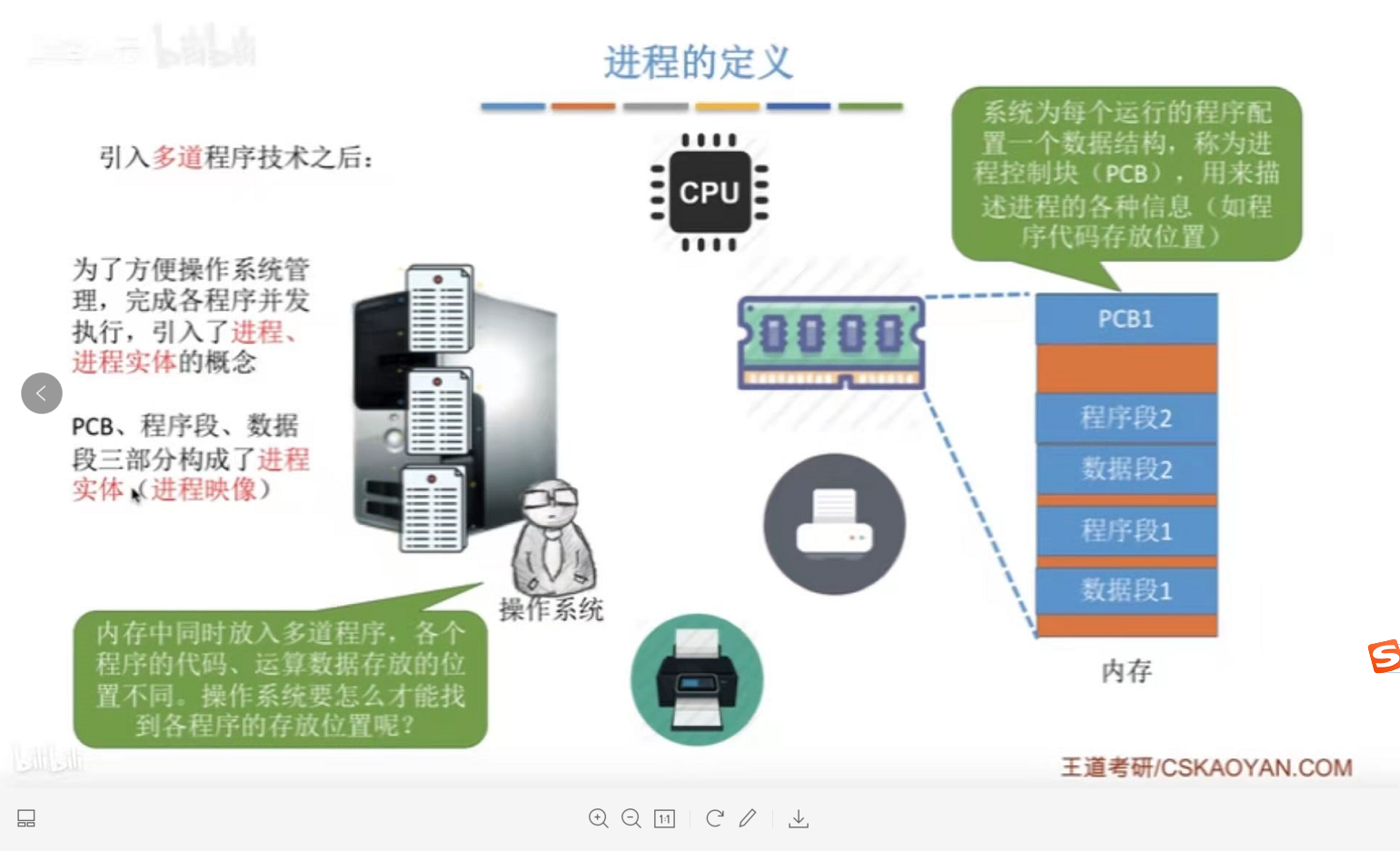
进程间通信方式

1. 管道 （匿名管道，命名管道）
2. 消息队列
3. 共享内存 + 信号量
4. 信号
5. Socket （网络 不同主机）

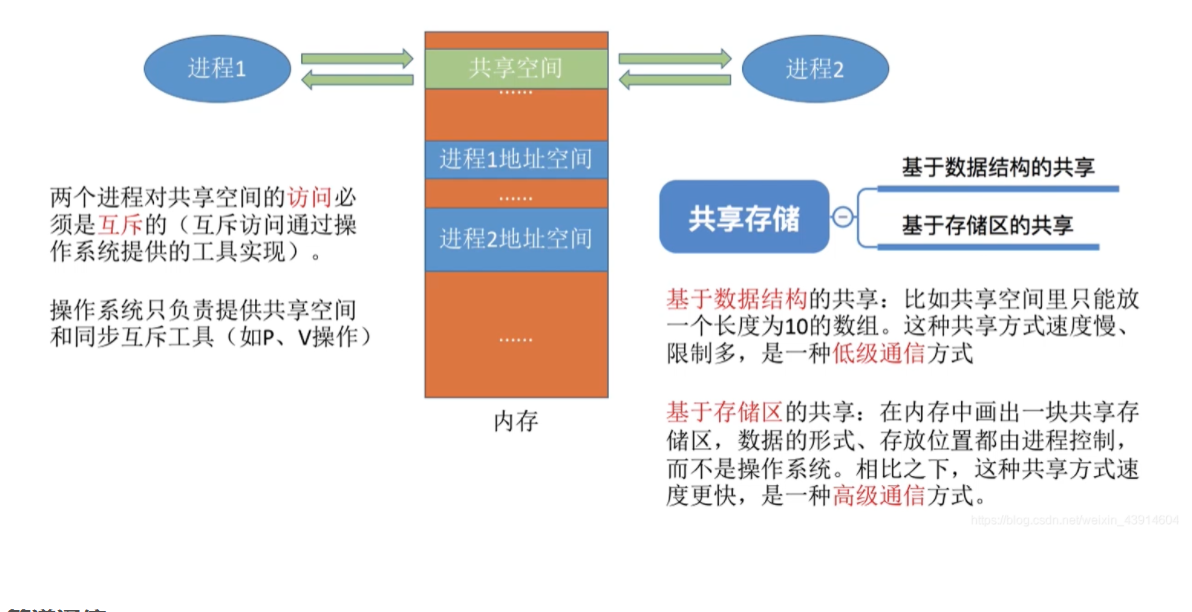
进程定义

PCB (进程控制块) + 程序段 + 数据段



### 共享内存 - 互斥 shared memary

共享内存主攻 + 信号量控制互斥性



**特别提醒**：共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取，所以我们通常需要用其他的机制来同步对共享内存的访问，例如信号量

**2.进程互斥（同一时间只有一个人可以操作资源）**

互斥，亦称间接制约关系。进程互斥指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源。

在这里需复习一下临界资源的概念。

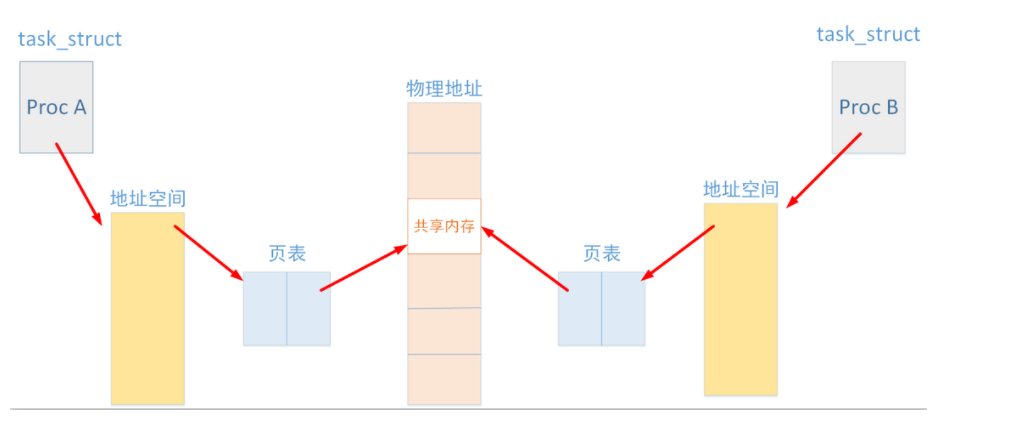
我们把一个时间段内只允许一个进程使用的资源称为临界资源。许多物理设备(比如摄像头、打印机)都属于临界资源。此外还有许多变量、数据、内存缓冲区等都属于临界资源。

对临界资源的访问，必须互斥地进行

**共享内存的通信原理**

在Linux中，每个进程都有属于自己的进程控制块（PCB）和地址空间（Addr Space），并且都有一个与之对应的页表，负责将进程的虚拟地址与物理地址进行映射，通过内存管理单元（MMU）进行管理。两个不同的虚拟地址通过页表映射到物理空间的同一区域，它们所指向的这块区域即共享内存。

共享内存的通信原理示意图：



对于上图我的理解是：当两个进程通过页表将虚拟地址映射到物理地址时，在物理地址中有一块共同的内存区，即共享内存，这块内存可以被两个进程同时看到。这样当一个进程进行写操作，另一个进程读操作就可以实现进程间通信。但是，我们要确保一个进程在写的时候不能被读，因此我们使用信号量来实现同步与互斥。

对于一个共享内存，实现采用的是**引用计数**的原理，当进程脱离共享存储区后，计数器减一，挂架成功时，计数器加一，只有当计数器变为零时，才能被删除。当进程终止时，它所附加的共享存储区都会自动脱离。

# **为什么共享内存速度最快？**

借助上图说明：Proc A 进程给内存中写数据， Proc B 进程从内存中读取数据，在此期间一共发生了两次复制

（1）Proc A 到共享内存       （2）共享内存到 Proc B

因为直接在内存上操作，所以共享内存的速度也就提高了。

Linux 查看共享内存：

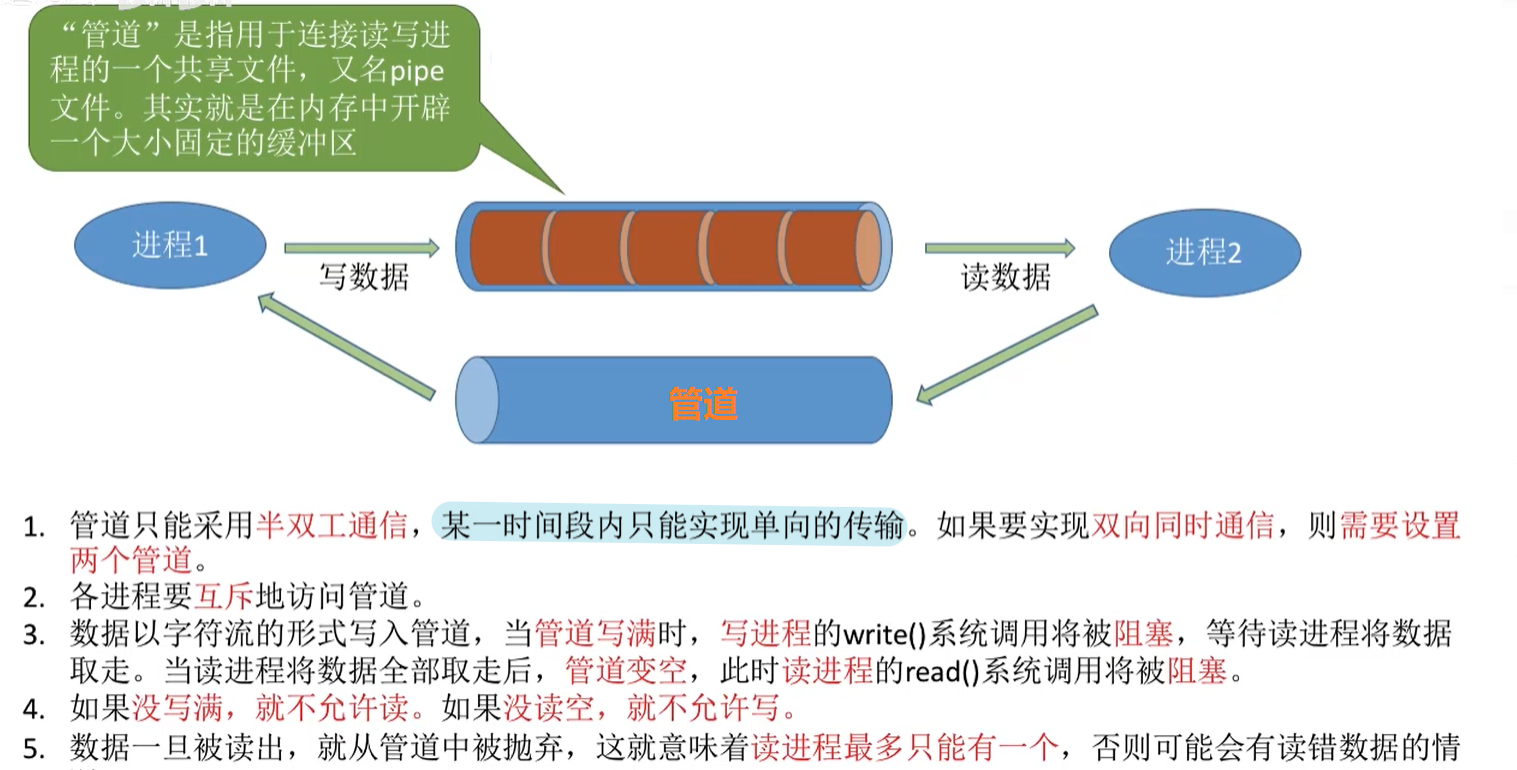
Ipcs -m

创建共享内存

shmget ( )：

### 管道 命名管道 - 互斥

****管道就是操作系统在内核中开辟的一段缓冲区，进程1可以将需要交互的数据拷贝到这段缓冲区，进程2就可以读取了，类似于下面这张图。****



管道分为两种

1. 匿名管道
2. 命名管道

**匿名管道**

匿名管道只支持有血缘关系的进程通信 - 父子 兄弟 有共同祖先的进程

Linux创建命令：

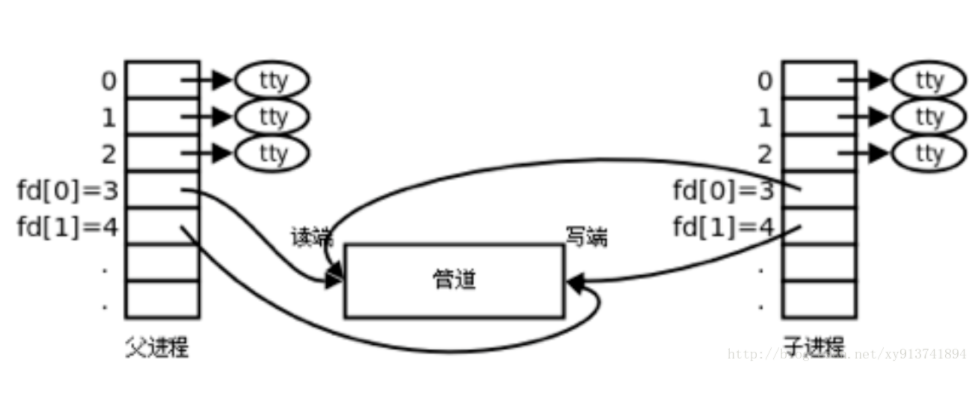
int pipe(int filedes[2]);

功能： 创建无名管道。

参数： filedes为 int 型数组的首地址，其存放了管道的文件描述符 filedes[0]、filedes[1]。

当一个管道建立时，它会创建两个文件描述符 fd[0] 和 fd[1]。其中 fd[0] 固定用于读管道，而 fd[1] 固定用于写管道。

返回值：成功：0 ; 失败：-1



由于管道是单向通信，如果是子进程写，父进程读取的话，需要关闭子进程的读端和父进程的写端。数据从写端流入从读端流出,这样就实现了进程间通信

**匿名管道大小 - pipe\_buff有64k**

**命名管道**

****命名管道也被称为FIFO文件，它是一种特殊类型的文件****，它在文件系统中以文件名的形式存在，但是它的行为却和之前所讲的没有名字的管道（匿名管道）类似。

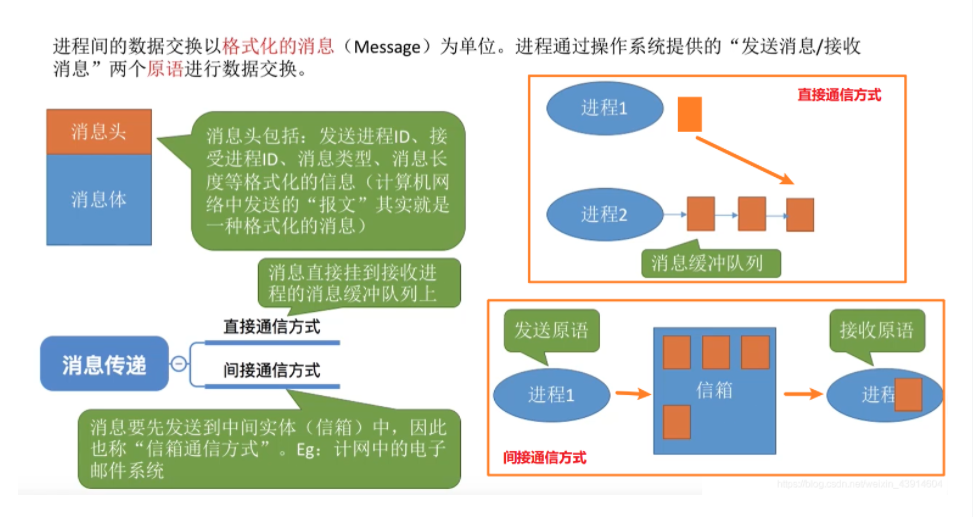
命名管道允许 非血缘关系进程通信

**Linux创建命令：**

**mkfifo**

****使用命名管道时，必须先调用open()将其打开。因为命名管道是一个存在于硬盘上的文件，而管道是存在于内存中的特殊文件。****

### 消息队列



### 信号

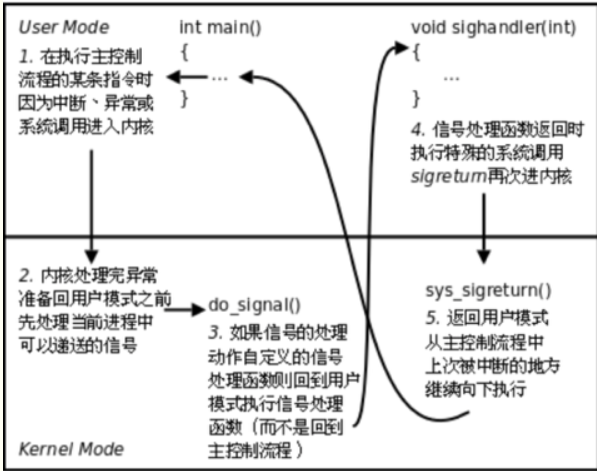
我感觉这么写下去越来越不像进程间的通信了，更像是进程间的打招呼。。。信号就是这样的，某某（这个某某有很多可能性）给进程一个信号，**进程就会在适当的情况下处理这个信号**（**这说明进程可能不会立即处理信号**），什么是适当的时候呢？**比如说中断返回的时候，或者内核态返回用户态的时候（这个情况出现的比较多）**等等（推荐本书《Linux内核设计与实现》，里面讲过）。。。这个也不是本篇的主题就不多述了。

首先来说**信号是怎么产生的**！

* **由硬件产生，别入从键盘敲入组合键发送一个信号，常用的Ctrl+C就可以给前台进程发送。**
* **由进程发送（或者说是由软件产生），比如我们可以在shell进程下输入kill命令给一个进程发送信号（命令：kill -信号标号 PID）**
* **异常，当异常的发生的时候肯定是会发送信号的**

然后说信号的处理方式，**谁来处理信号？肯定是操作系统来**，难不成还是程序员么。文章一开头就说了，**信号不一定会被立即处理，操作系统不会为了处理一个信号而把当前正在运行的进程挂起（切换进程）或者杀掉（肯定不会杀掉啊，难道看见一个信号就杀害一个无辜群众么），挂起（进程切换）的话消耗太大了，如果不是紧急信号，可能是不会立即处理的。操作系统多选择在内核态切换回用户态的时候处理信号，这样就利用两者的切换来处理了（不用单独进行进程切换以免浪费时间）。**

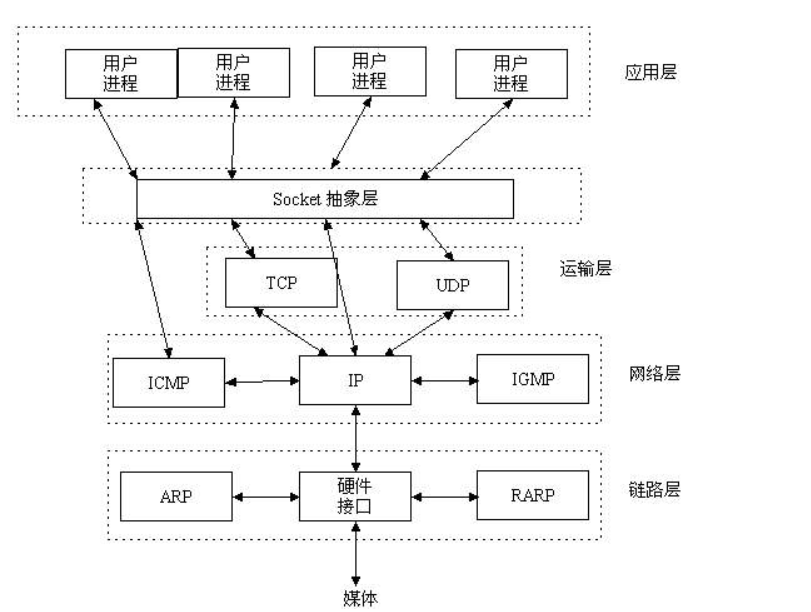
总归是不能避免的，因为**很有可能在睡眠的进程就接收到信号，操作系统肯定不愿意切换当前正在happy地跑着的进程，于是就得把信号储存啊，因为是进程收到的信号，所以把信号储存在进程唯一的PCB（就是task\_struct）当中。**struct sigpending pending;字段就是存放信号的信号表，之后会解释pending。



### Socket

我们知道两个进程如果需要进行通讯最基本的一个前提能能够唯一的标示一个进程，在本地进程通讯中我们可以使用PID来唯一标示一个进程，但PID只在本地唯一，网络中的两个进程PID冲突几率很大，这时候我们需要另辟它径了，我们知道IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。

能够唯一标示网络中的进程后，它们就可以利用socket进行通信了，什么是socket呢？我们经常把socket翻译为套接字，socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。



socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。

**socket通信过程**

