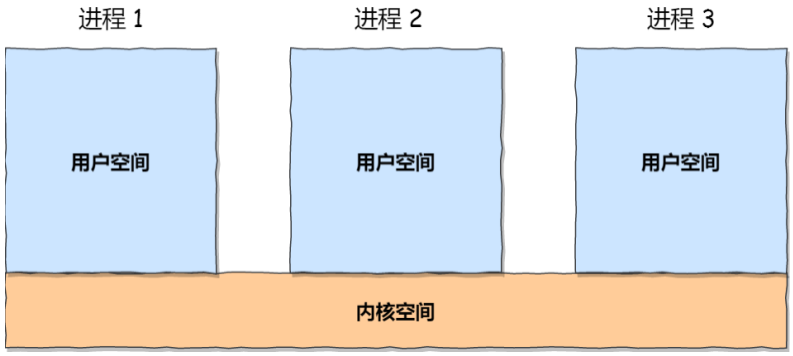
进程间通信

没时间看正文的话，可以直接看文后的总结，也是挺方便记忆的。

什么是进程间通信？？

每个进程各自有不同的用户地址空间,任何一个进程的全局变量在另一个进程中都看不到，**所以进程之间要交换数据必须通过内核,在内核中开辟一块缓冲区**,进程A把数据从用户空间拷到内核缓冲区,进程B再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信。



管道

> linux命令中 "|"就是一个管道

```
$ ps -ef | grep xxxxx
```

功能是将前面一个命令的输出，作为后面一个命令的输入，可以从描述看出，管道传输数据是单向的（半双工），如果想要互相通信，就得创建两个管道才行。这个管道也叫**匿名管道**，用完就销毁

还有一种管道叫**命名管道**，FIFO，因为数据是先进先出的传输方式，在使用命名管道前，先需要通过mkfifo命令来创建

`$ mkfifo pipename`

在linux一切皆文件，所以管道也是以文件形式存在

```
$ echo "hello" > pipename// 将数据写进管道
                          // 停住了 ...
```

当我们往管道里面写东西时，进程会卡住，因为只有管道内的数据被读取完，命令才可以正常退出。

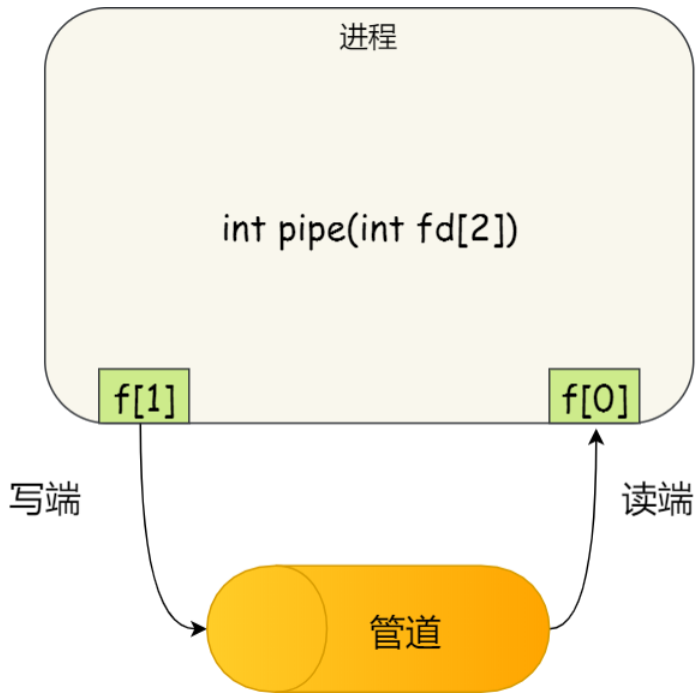
```
$ cat < pipename  //读取
hello
```

所以**管道这种方式效率低，不适合进程间频繁交换数据，好处是容易知道管道内数据已经被另一个进程读取了**

C 复制代码

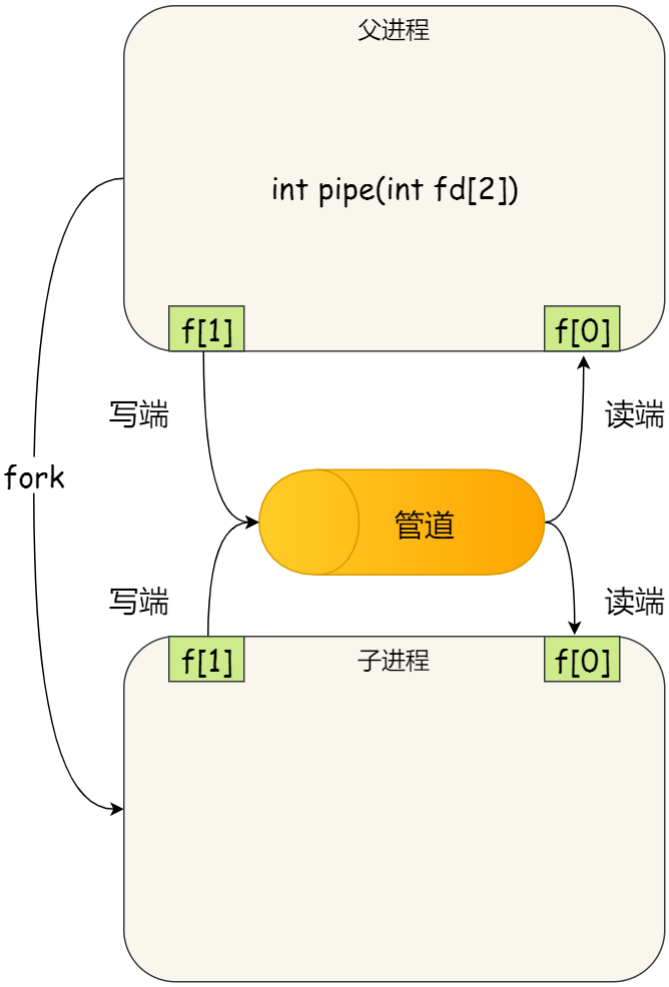
```c
// 需要的头文件
#include <unistd.h>

// 通过pipe()函数来创建匿名管道
// 返回值：成功返回0，失败返回-1
// fd参数返回两个文件描述符
// fd[0]指向管道的读端，fd[1]指向管道的写端
// fd[1]的输出是fd[0]的输入。
int pipe (int fd[2]);
```



所谓的管道也就是内核里面额一串缓存，从管道的一端写入的数据，实际是缓存在内核中的，另一端读取，也就是从内核中读取这段数据。另外，管道传输的数据是无格式的流且大小受限。这里是在一个进程中通信，并没有跨进程，那么如何跨进程进行通信呢？？？

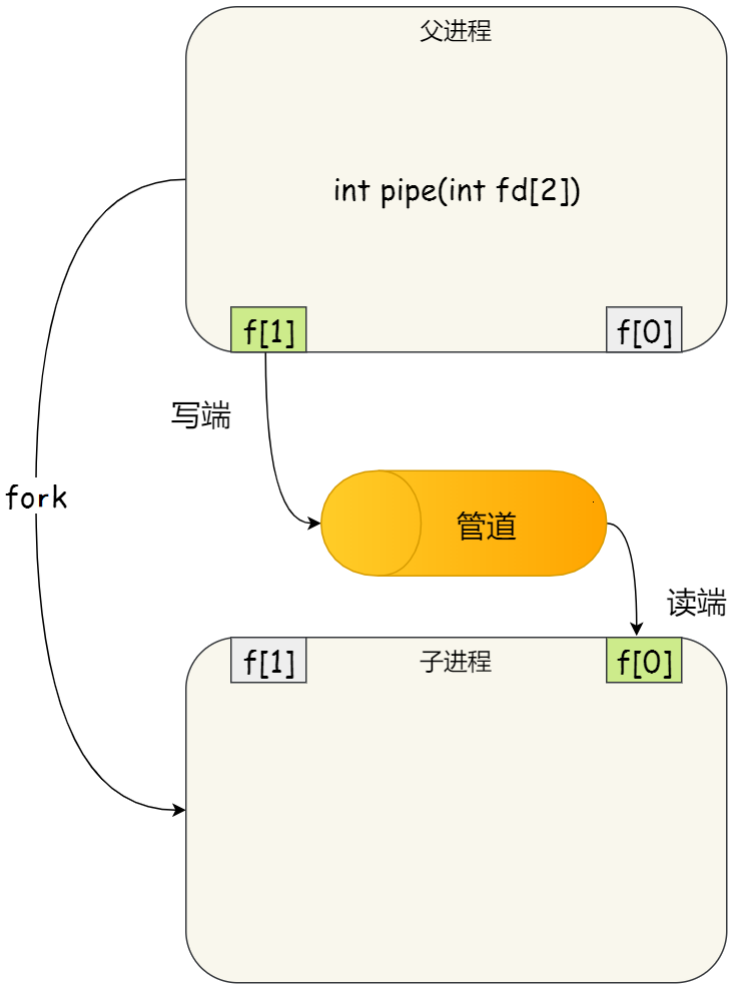
我们可以fork来创建子进程，创建的子进程会复制父进程的文件描述符，这样两个进程各有两个fd[0]与fd[1]，两个进程就可以通过各自的fd写入和读取同一个管道文件实现跨进程通信了。



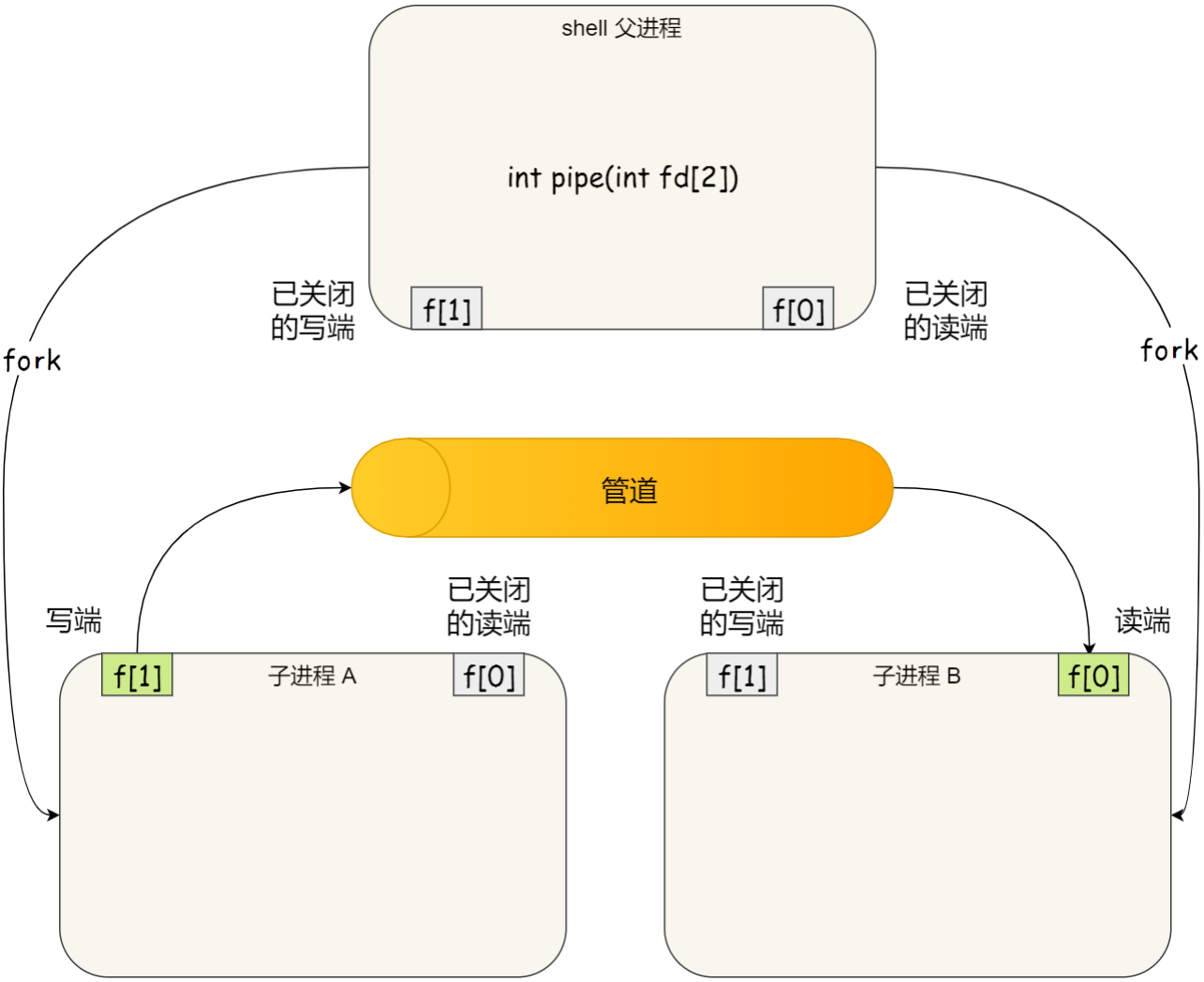
因为父进程和子进程都可以同时写入，也都可以读出。那么，为了避免这种情况，通常的做法是：

1. 父进程关闭读取的 fd[0]，只保留写入的 fd[1]；
2. 子进程关闭写入的 fd[1]，只保留读取的 fd[0]；

变成如下情况：



所以如果需要双向通信，就应该创建两个管道



到这里，我们仅仅解析了使用管道进行父进程与子进程之间的通信，但是在我们shell里面并不是这样的。在shell里面执行 A | B 命令的时候， A进程和 B 进程都是shell创建出来的子进程， A 和 B 之间不存在父子关系，它俩的父进程都是 shell。

所以说，在 shell 里通过"|"匿名管道将多个命令连接在一起，实际上也就是创建了多个子进程，那么在我们编写 shell 脚本时，能使用一个管道搞定的事情，就不要多用一个管道，这样可以减少创建子进程的系统开销。

我们可以得知， 对于匿名管道，它的通信范围是存在父子关系的进程。因为管道没有实体，也就是没有管道文件，只能通过 fork 来复制父进程 fd 文件描述符，来达到通信的目的。
另外， 对于命名管道，它可以在不相关的进程间也能相互通信。因为命令管道，提前创建了一个类型为管道的设备文件，在进程里只要使用这个设备文件，就可以相互通信。

不管是匿名管道还是命名管道，进程写入的数据都是缓存在内核中，另一个进程读取数据时候自然也是从内核中获取，同时通信数据都遵循先进先出原则，不支持 lseek 之类的文件定位操作。 效率低，不适

合进程之间频繁地交换数据。

消息队列

对于管道效率低不适合**进程中之间频繁交换数据**这个问题，消息队列可以解决：
A 进程要给 B 进程发送消息， A 进程把数据放在对应的消息队列后就可以正常返回了， B 进程需要的时候再去读取数据就可以了。同理， B 进程要给 A 进程发送消息也是如此。

再来， 消息队列是**保存在内核中的消息链表**，在发送数据时，会分成一个一个独立的数据单元，也就是消息体（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，所以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。如果进程从消息队列中读取了消息体，内核就会把这个消息体删除。（其实类比kafka之类的是可以的）

但是这种通信方式也是有问题的：

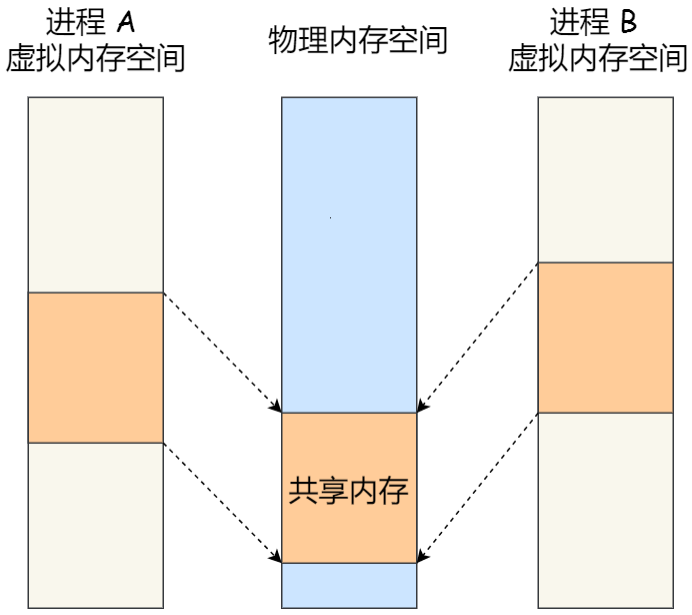
1. **消息队列不适合比较大数据的传输**：因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。
2. **消息队列通信过程中，存在用户态到内核态的数据拷贝开销**：因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

共享内存

消息队列存在的用户态到内核态数据拷贝开销问题，共享内存就可以会很好解决

> 现代操作系统，对于内存管理，采用的是虚拟内存技术，也就是每个进程都有自己独立的虚拟内存空间，不同进程的虚拟内存映射到不同的物理内存中。所以，即使进程 A 和 进程 B 的虚拟地址是一样的，其实访问的是不同的物理内存地址，对于数据的增删查改互不影响。

共享内存机制，就是拿出一块虚拟地址空间，映射到相同的物理内存中，这样一个进程写入的东西，另一个进程就马上能看到，不需要进行拷贝，大大提高了进程间通信的速度。



不过这种方式又带来新的问题，那就是如果多个进程同时修改一个共享内存，很有可能就冲突了。假如两个进程都同时写一个地址，那么先写的进程内容就会被覆盖了

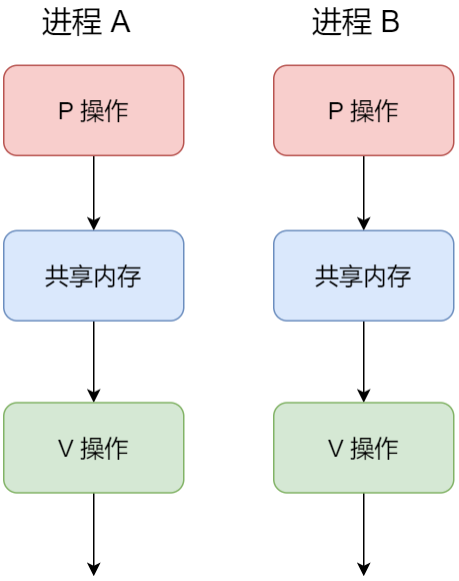
信号量

为了解决共享内存中多进程竞争资源，造成数据混乱，所以引入信号量，来实现保护机制，使得共享资源在任意时刻只能被一个进程访问

信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。

> 信号量表示资源的数量，控制信号量的方式有两种原子操作：
> 一个是 P 操作，这个操作会把信号量减去 1，相减后如果信号量 < 0，则表明资源已被占用，进程需阻塞等待；相减后如果信号量 >= 0，则表明还有资源可使用，进程可正常继续执行。
> 另一个是 V 操作，这个操作会把信号量加上 1，相加后如果信号量 <= 0，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0，则表明当前没有阻塞中的进程；
> **P 操作是用在进入共享资源之前， V 操作是用在离开共享资源之后，这两个操作是必须成对出现的。**

如果要使得两个进程互斥访问共享内存，我们可以初始化信号量为 1 。



具体的过程如下：

进程A在访问共享内存前，先执行了P操作，由于信号量的初始值为1，故在进程A执行P操作后信号量变为0，表示共享资源可用，于是进程A就可以访问共享内存。

若此时，进程B也想访问共享内存，执行了P操作，结果信号量变为了-1，这就意味着临界资源已被占用，因此进程B被阻塞。

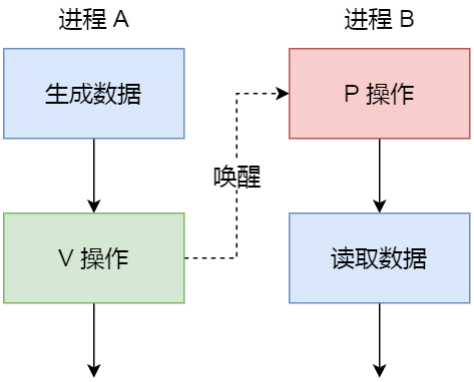
直到进程A访问完共享内存，才会执行V操作，使得信号量恢复为0，接着就会唤醒阻塞中的线程B，使得进程B可以访问共享内存，最后完成共享内存的访问后，执行V操作，使信号量恢复到初始值1。

可以发现，信号初始化为1，就代表着是互斥信号量，它可以保证共享内存在任何时刻只有一个进程在访问，这就很好的保护了共享内存。

另外，在多进程里，每个进程并不一定是顺序执行的，它们基本是以各自独立的、不可预知的速度向前推进，但有时候我们又希望多个进程能密切合作，以实现一个共同的任务。

例如，进程A是负责生产数据，而进程B是负责读取数据，这两个进程是相互合作、相互依赖的，进程A必须先生产了数据，进程B才能读取到数据，所以执行是有前后顺序的。

那么这时候，就可以用信号量来实现多进程同步的方式，我们可以初始化信号量为0。



具体过程：

如果进程B比进程A先执行了，那么执行到P操作时，由于信号量初始值为0，故信号量会变为-1，表示进程A还没生产数据，于是进程B就阻塞等待；
接着，当进程A生产完数据后，执行了V操作，就会使得信号量变为0，于是就会唤醒阻塞在P操作的进程B；
最后，进程B被唤醒后，意味着进程A已经生产了数据，于是进程B就可以正常读取数据了。
可以发现，信号初始化为0，就代表着是同步信号量，它可以保证进程A应在进程B之前执行。

信号

以上的几个进程间通信，都是常规状态下的工作模式，对于异常情况下的工作模式，需要**信号**
在Linux操作系统中，为了响应各种各样的事件，提供了几十种信号，分别代表不同的意义。我们可以通过kill -l命令，查看所有的信号：

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

运行在 shell 终端的进程，我们可以通过键盘输入某些组合键的时候，给进程发送信号。例如
Ctrl+C 产生 SIGINT 信号，表示终止该进程；
Ctrl+Z 产生 SIGTSTP 信号，表示停止该进程，但还未结束；
如果进程在后台运行，可以通过 kill 命令的方式给进程发送信号，但前提需要知道运行中的进程 PID 号，例如：
kill -9 1050 ，表示给 PID 为 1050 的进程发送 SIGKILL 信号，用来立即结束该进程；
所以，**信号事件的来源主要有硬件来源（如键盘 Cltr+C ）和软件来源（如 kill 命令）。**
信号是进程间通信机制中唯一的异步通信机制，因为可以在任何时候发送信号给某一进程，一旦有信号产生，我们就有下面这几种，用户进程对信号的处理方式。
1.**执行默认操作**。 Linux 对每种信号都规定了默认操作，例如，上面列表中的 SIGTERM 信号，就是终止进程的意思。
2.**捕捉信号**。我们可以为信号定义一个信号处理函数。当信号发生时，我们就执行相应的信号处理函数。
3.**忽略信号**。当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即 SIGKILL 和 SEGSTOP ，它们用于在任何时候中断或结束某一进程。

Socket

前面的几种通信方式都是在同一个主机上进行进程通信，如果需要跨网络进行不同网络不同主机间进行通信，就需要socket通信了。

实际上， Socket 通信不仅可以跨网络与不同主机的进程间通信，还可以在同主机上进程间通信。

我们来看看创建 socket 的系统调用：

int socket(int domain, int type, int protocal)

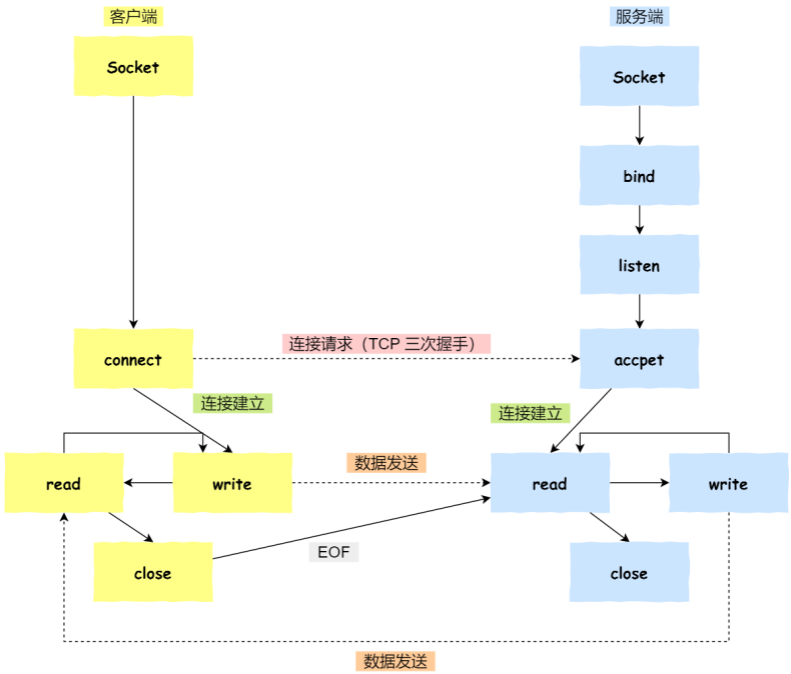
三个参数分别代表：

- domain 参数用来指定协议族，比如 AF_INET 用于 IPV4、 AF_INET6 用于 IPV6、AF_LOCAL/AF_UNIX 用于本机；
- type 参数用来指定通信特性，比如 SOCK_STREAM 表示的是字节流，对应 TCP、SOCK_DGRAM表示的是数据报，对应 UDP、 SOCK_RAW 表示的是原始套接字；
- protocal 参数原本是用来指定通信协议的，但现在基本废弃。因为协议已经通过前面两个参数指定完成， protocol 目前一般写成 0 即可；

根据创建 socket 类型的不同，通信的方式也就不同：

- 实现 TCP 字节流通信： socket 类型是 AF_INET 和 SOCK_STREAM；
- 实现 UDP 数据报通信： socket 类型是 AF_INET 和 SOCK_DGRAM；
- 实现本地进程间通信： 「本地字节流 socket 」类型是 AF_LOCAL 和 SOCK_STREAM，「本地数据报 socket 」类型是 AF_LOCAL 和 SOCK_DGRAM。另外， AF_UNIX 和 AF_LOCAL 是等价的，所以AF_UNIX 也属于本地 socket；

> 针对 TCP 协议通信的 socket 编程模型



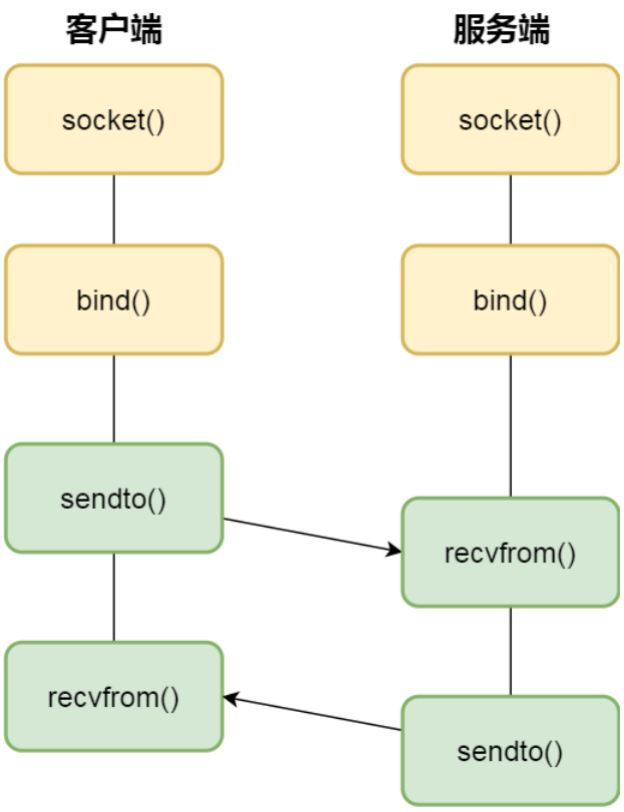
- 服务端和客户端初始化 `socket`，得到文件描述符；
- 服务端调用 `bind`，将绑定在 IP 地址和端口;
- 服务端调用 `listen`，进行监听；
- 服务端调用 `accept`，等待客户端连接；
- 客户端调用 `connect`，向服务器端的地址和端口发起连接请求；
- 服务端 `accept` 返回用于传输的 `socket` 的文件描述符；
- 客户端调用 `write` 写入数据；服务端调用 `read` 读取数据；
- 客户端断开连接时，会调用 `close`，那么服务端 `read` 读取数据的时候，就会读取到了 `EOF`，待处理完数据后，服务端调用 `close`，表示连接关闭。

这里需要注意的是，服务端调用 `accept` 时，连接成功了会返回一个已完成连接的 socket，后续用来传输数据。

所以，监听的 socket 和真正用来传送数据的 socket，是「两个」 socket，一个叫作**监听 socket**，一个叫作**已完成连接 socket**。

成功连接建立之后，双方开始通过 read 和 write 函数来读写数据，就像往一个文件流里面写东西一样。

> 针对 UDP 协议通信的 socket 编程模型



UDP 是没有连接的，所以不需要三次握手，也就不需要像 TCP 调用 listen 和 connect，但是 UDP 的交互仍然需要 IP 地址和端口号，因此也需要 bind。

对于 UDP 来说，不需要要维护连接，那么也就没有所谓的发送方和接收方，甚至都不存在客户端和服务端的概念，只要有一个 socket 多台机器就可以任意通信，因此每一个 UDP 的 socket 都需要 bind。

另外，每次通信时，调用 sendto 和 recvfrom，都要传入目标主机的 IP 地址和端口。

> 针对本地进程间通信的 socket 编程模型

本地 socket 被用于在同一台主机上进程间通信的场景：

- 本地 socket 的编程接口和 IPv4 、IPv6 套接字编程接口是一致的，可以支持「字节流」和「数据报」两种协议；
- 本地 socket 的实现效率大大高于 IPv4 和 IPv6 的字节流、数据报 socket 实现；

对于本地字节流 socket，其 socket 类型是 AF_LOCAL 和 SOCK_STREAM。

对于本地数据报 socket，其 socket 类型是 AF_LOCAL 和 SOCK_DGRAM。

本地字节流 socket 和 本地数据报 socket 在 bind 的时候，不像 TCP 和 UDP 要绑定 IP 地址和端口，而是绑定一个本地文件，这也就是它们之间的最大区别。

总结

由于每个进程的用户空间相互独立，不同相互访问，所以需要借助内核空间来进行进程间通信，原因很简单，进程共享一个内核空间。

Linux提供了不少进程间通信方式：

1. 最简单的就是**管道**，分为匿名管道和命名管道。匿名管道只能用于父子进程之间的通信，生命周期随着进程创建而创建，终止而消失；命名管道突破了只能在父子关系进程间的通信。管道都是单向通信，通信数据是无格式且大小受限。
2. **消息队列**解决了管道通信的数据是无格式字节流的问题，消息队列的消息体是可以用户自定义的数据类型，但是其通信速度是最不及时的，因为每次数据的写入和读取都需要经过用户态和内核态的拷贝。
3. **共享内存**可以解决消息队列通信中用户态和内核态数据拷贝带来的开销，它直接分配一个共享空间，每个进程都可以直接访问，不需要用户态和内核态的切换，大大提高了通信速度，是最快的进程间通信方式。但是由于多个进程可能竞争同一个共享资源造成问题
4. **信号量**就是来保护共享资源的，确保任何时刻只有一个进程可以访问共享资源，也就是互斥访问
5. 上面几个都是正常的进程间通信，**信号**是进程间通信中唯一一个异步通信机制，信号可以在应用和内核之间直接交互，也可以利用信号来通知用户空间的进程发生了哪些系统事件，信号事件的来源主要有硬件来源（如键盘 Cltr+C ）和软件来源（如 kill 命令）
6. 前面的几种进程间通信方式都是工作在同一个主机，如果需要在不同主机之间进行进程间通信，就需要Socket通信了。当然Socket也是可以用于本地主机进程间的通信。根据Socket类型不同，分为三种：一种基于TCP，一种基于UDP，一种是本地进程间通信。

以上，就是进程间通信的主要机制了。你可能会问了，那线程通信间的方式呢?

同个进程下的线程之间都是共享进程的资源，只要是共享变量都可以做到线程间通信，比如全局变量，**所以对于线程间关注的不是通信方式，而是关注多线程竞争共享资源的问题**，信号量也同样可以在线程间实现**互斥与同步**：