C8 进程间通信

Linux系统中进程通信的机制继承自Unix，后经贝尔实验室与BSD对进程间通讯手段的改进与扩充，以及POSIX标准对Unix标准的统一，发展出如今Linux系统中使用的进程通信（IPC）机制，即包含管道通信、信号量、消息队列、共享内存以及socket通信等的诸多通讯机制。

管道

管道是一种最基本的进程通信机制，其实质是由内核管理的一个缓冲区，可以形象地认为管道的两端连接着两个需要进行通信的进程，其中一个进程进行信息输出，将数据写入管道；另一个进程进行信息输入，从管道中读取信息。管道的逻辑结构如图所示。

管道分为匿名管道（pipe）和命名管道（named pipe）。

在进程中创建的管道是匿名管道，进程退出后管道会被销毁，匿名管道只能用于有亲缘关系的进程间通信。

命名管道被具象化为一个文件，在进程中可使用操作文件的方式向内存中写入或从内存中读出数据，命名管道与进程的联系较弱，相当于一个读写内存的接口，进程退出后，命名管道依然存在。

匿名管道

pipe()

匿名管道利用fork机制建立联系，刚创建出的管道，读写两端都连接在同一个进程上，当进程中调用fork()创建子进程后，父子进程共享文件描述符，因此子进程拥有与父进程相同的管道。pipe()创建管道后读端对应的文件描述符为fd[0]，写端对应的文件描述符为fd[1]，fork后父子进程中文件描述符与

（1）管道采用半双工通信方式，只能进行单向数据传递，为严谨起见，应使用close()函数关闭除通信端口之外的端口。

（2）管道只能进行半双工通信，若要实现同时双向通信，需要为通信的进程创建两个管道。

（3）有指向管道读端的文件描述符打开时，向管道中写入数据才有意义。

（4）若所有指向管道写端的文件描述符都被关闭后仍有进程从管道的读端读取数据，那么管道中剩余的数据都被读取后，再次read会返回0。

（5）若有指向管道写端的文件描述符未关闭，而管道写端的进程也没有向管道中写入数据，那么当进程从管道中读取数据，且管道中剩余的数据都被读取时，再次read会阻塞，直到写端向管道写入数据，阻塞才会解除。

（6）若有指向管道读端的文件描述符没关闭，但读端进程没有从管道中读取数据，写端进程持续向管道中写入数据，那么管道缓存区写满时再次write会阻塞，直到读端将数据读出，阻塞才会解除。

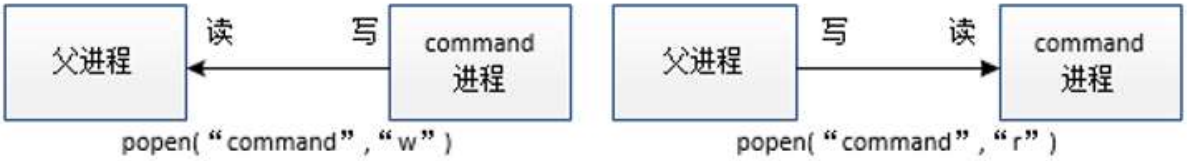
（7）管道中的数据以字节流的形式传输，这要求管道两端的进程事先约定好数据的格式。

popen()/pclose()

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);

popen()函数的功能是：调用pipe()函数创建管道，调用fork()函数创建子进程，之后在子进程中通过execve()函数调用shell命令执行相应功能，若整个流程都成功执行，则返回一个I/O文件指针；若pipe()或fork()函数调用失败，或因无法分配内存等原因造成popen()函数调用失败，该函数将会返回NULL。



pclose()函数的功能是关闭由popen()打开的I/O流，并通过调用wait()函数等待子进程命令执行结束，返回shell的终止状态，防止产生僵尸进程。与文件操作函数fopen()类似， popen()调用之后务必要使用pclose()函数关闭打开的文件I/O指针，若pclose()函数调用失败，则返回-1。

命名管道

命名管道又名FIFO（first in first out），它与匿名管道的不同之处在于：命名管道与系统中的一个路径名关联，以文件的形式存在于文件系统中，如此，系统中的不同进程可以通过FIFO的路径名访问FIFO文件，实现彼此间的通信。

mkfifo()

mkfifo [选项] 参数

mkfifo命令的参数一般为文件名，其常用参数为-m，用于指定所创建文件的权限。

在程序中创建FIFO文件的函数与mkfifo同名，mkfifo()的头文件为sys/type.h与sys/stat.h，其函数声明如下：

int mkfifo(const char *pathname, mode_t mode);

消息队列

消息队列的实质是一个存放消息的链表，该链表由内核维护；消息队列中的每个消息可以视为一条记录，消息包括一个长整型的类型字段和需要传递的数据。消息队列由消息队列标识符（queue ID）标识，对消息队列有读权限的进程可以从队列中读取消息，对消息队列有写权限的进程可以按照规则，向其中添加消息。

与管道相比，消息队列的通信方式更为灵活：

提供有格式的字节流，无需通信双方额外约定数据传输格式

将消息设定为不同类型，并分配了不同的优先级

新添加的消息总是在队尾，但接收消息的进程可以读取队列中间的数据

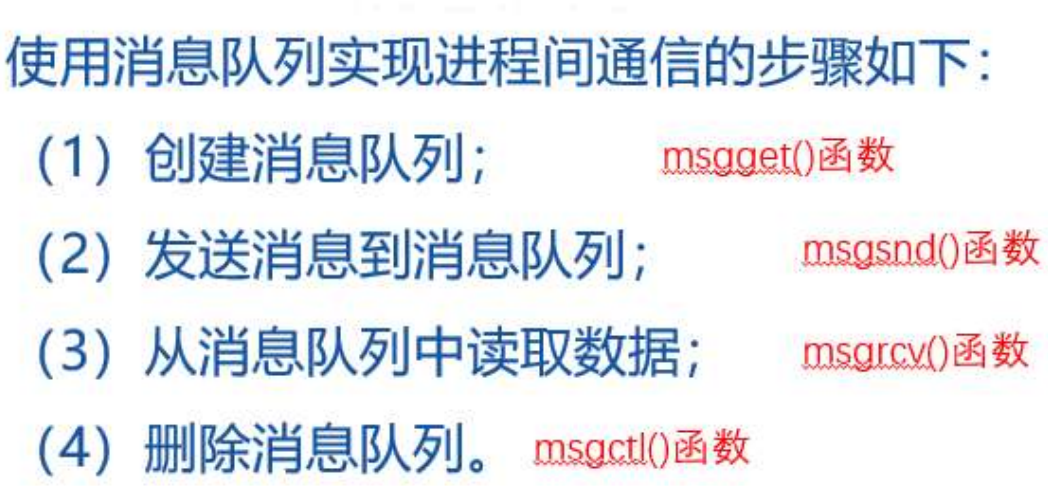
降低了读写进程间的耦合强度

与FIFO类似，消息队列可以实现无亲缘关系进程间的通信，且独立于通信双方的进程之外，若没有删除内核中的消息队列，即便所有使用消息队列的进程都已终止，消息队列仍存在于内核中，直到内核重新启动、管理命令被执行或调用系统接口删除消息队列时，消息队列才会真正被销毁。

注意：

（1）系统中的最大消息队列数与系统中最大消息数都有一定限制，分别由宏MSGMNI和宏MSGTOL定义；

（2）消息队列的每个消息中所含数据块的长度以及队列中所含数据块的总长度也有限制，分别由宏MSGMAX和MSGMNB定义。



键值与标识符

对多个进程来说，要通过消息队列机制实现进程间通信，必须能与相同消息队列进行关联，键值（key）就是实现进程与消息队列关联的关键。当在进程中调用msgget()函数创建消息队列时，传入的key值会被保存到内核中，与msgget()函数创建的消息队列一一对应；若进程中调用msgget()函数获取已存在的消息队列，只需向msgget()函数中传入键值，就能获取到内核中与键值对应的消息队列。也就是说，键值是消息队列在内存级别的唯一标识。

对单个进程来说，可能需要实现与多个进程间的通信，因此会与多个消息队列关联，当多次调用msgget()函数与多个消息队列进行关联时，每个msgget()函数都会返回一个非负整数，这个非负整数就是进程对消息队列的标识，标识符是消息队列在进程级别的唯一标识。

信号量

Linux系统采用多道程序设计技术，允许多个进程同时在内核中运行，但同一个系统中的多个进程之间，可能因为进程合作或资源共享，产生制约关系。

制约关系分为直接相互制约关系和间接相互制约关系：

（1）需要进程间协调合作导致的制约关系，称为直接相互制约关系。

（2）因资源共享导致的制约关系，称为间接相互制约关系。

直接相互制约的进程间有同步关系，间接相互制约的进程间有互斥关系，同步与互斥存在的根源是系统中存在临界资源（Critical Resouce）。

计算机中的硬件资源（如内存、打印机、磁盘）以及软件资源（如共享代码段、变量等）都是临界资源，为了避免多进程的并发执行造成的不一致性，临界资源在同一时刻只允许有限个进程对其进行访问或修改。

计算机中的多个进程必须互斥地访问系统中的临界资源，用于访问临界资源的代码称为临界区（Critical Section），临界区也属于临界资源，若能保证进程间互斥地进入自己的临界区，就能实现进程对临界资源的互斥访问。

信号量（Semaphore）是专门用于解决进程同步与互斥问题的一种通信机制，它与信号无关，也不同于管道、FIFO以及消息队列，一般不用来传输数据，信号量包括一个被称为信号量的表示资源数量的非负整型变量、修改信号量的原子操作P和V，以及该信号量下等待资源的进程队列。

在Linux系统中，不同的进程通过获取同一个信号量键值进行通信，实现进程间对资源的互斥访问。使用信号量进行通信时，通常需要以下步骤：

semget() （1）创建信号量/信号量集，或获取系统中已有的信号量/信号量集；

semctl() （2）初始化信号量。早期信号量通常被初始为1，但有些进程一次需要多个同类的临界资源，或多个不同类且不唯一的临界资源，因此可能需要初始化的不是信号量，而是一个信号量集；

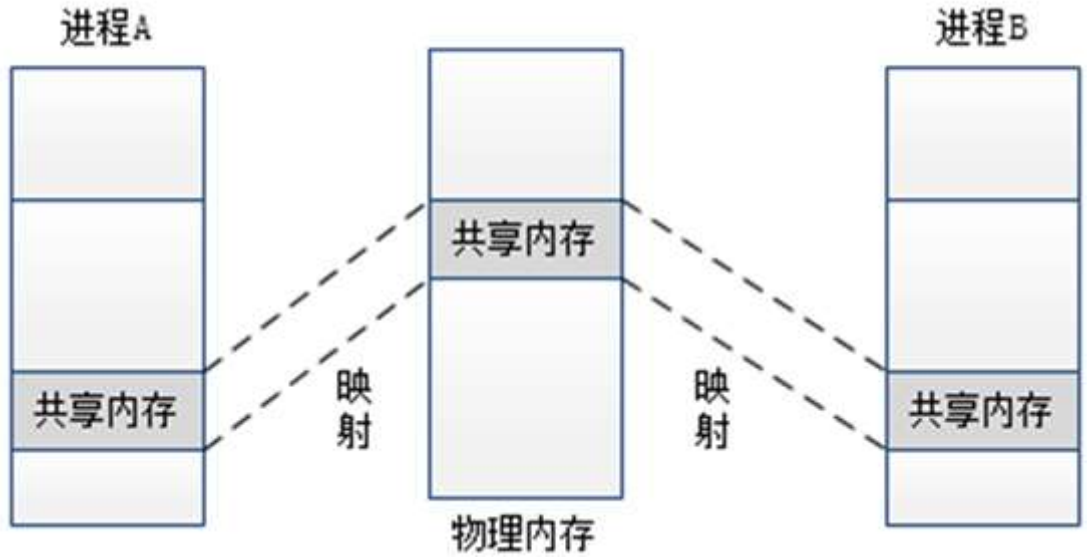
semop() （3）信号量的P、V操作，根据进程请求，修改信号量的数量。执行P操作会使信号量-1，执行V操作会使信号量+1；

semctl() （4）从系统中删除不需要的信号量。

共享内存

共享内存允许两个或多个进程访问给定的同一块存储区域。已知当一个进程被启动时，系统会为其创建一个0~4G的虚拟内存空间，根据虚拟地址与物理地址之间的映射关系，进程可以通过操作虚拟地址，实现对物理页面的操作。

一般情况下，每个进程的虚拟地址空间会与不同的物理地址进行映射，但是当使用共享内存进行通信时，系统会将同一段物理内存映射给不同的进程。两个进程的虚拟地址空间与共享内存之间的映射关系如图8-7所示。



shmget()创建一块新的共享内存，或打开一块已经存在的共享内存

shmat()进行地址映射，将共享内存映射到进程虚拟地址空间中

shmdt()解除物理内存与进程虚拟地址空间的映射关系

shmctl()对已存在的共享内存进行操作