穷佐罗的Linux书
Poor Zorro's Linux Book

Linux的进程间通信 – 管道

Linux的进程间通信 – 管道

前言

管道是UNIX环境中历史最悠久的进程间通信方式。本文主要说明在Linux环境上如何使用管道。阅读本文可以帮你解决以下问题：

1. 什么是管道和为什么要有管道?
2. 管道怎么分类?
3. 管道的实现是什么样的?
4. 管道有多大?
5. 管道的大小是不是可以调整？如何调整?

如果觉得本文还不错，请扫码任意打赏捐助穷佐罗。

什么是管道?

管道，英文为pipe。这是一个我们在学习Linux命令行的时候就会引入的一个很重要的概念。它的发明人是道格拉斯.麦克罗伊，这位也是UNIX上早期shell的发明人。他在发明了shell之后，发现系统操作执行命令的时候，经常有需求要将一个程序的输出交给另一个程序进行处理，这种操作可以使用输入输出重定向加文件搞定，比如:

```
[zorro@zorro-pc pipe]$ ls  -l /etc/ > etc.txt
[zorro@zorro-pc pipe]$ wc -l etc.txt
183 etc.txt
```

但是这样未免显得太麻烦了。所以，管道的概念应运而生。目前在任何一个shell中，都可以使用“|”连接两个命令，shell会将前后两个进程的输入输出用一个管道相连，以便达到进程间通信的目的：

```
[zorro@zorro-pc pipe]$ ls -l /etc/ | wc -l
183
```

对比以上两种方法，我们也可以理解为，管道本质上就是一个文件，前面的进程以写方式打开文件，后面的进程以读方式打开。这样前面写完后面读，于是就实现了通信。实际上管道的设计也是遵循UNIX的“一切皆文件”设计原则的，它本质上就是一个文件。Linux系统直接把管道实现成了一种文件系统，借助VFS给应用程序提供操作接口。

虽然实现形态上是文件，但是管道本身并不占用磁盘或者其他外部存储的空间。在Linux的实现上，它占用的是内存空间。所以，Linux上的管道就是一个操作方式为文件的内存缓冲区。

管道的分类和使用

Linux上的管道分两种类型：

1. 匿名管道
2. 命名管道

这两种管道也叫做有名或无名管道。匿名管道最常见的形态就是我们在shell操作中最常用的”|”。它的特点是只能在父子进程中使用，父进程在产生子进程前必须打开一个管道文件，然后fork产生子进程，这样子进程通过拷贝父进程的进程地址空间获得同一个管道文件的描述符，以达到使用同一个管道通信的目的。此时除了父子进程外，没人知道这个管道文件的描述符，所以通过这个管道中的信息无法传递给其他进程。这保证了传输数据的安全性，当然也降低了管道了通用性，于是系统还提供了命名管道。

我们可以使用mkfifo或mknod命令来创建一个命名管道，这跟创建一个文件没有什么区别：

```
[zorro@zorro-pc pipe]$ mkfifo pipe
[zorro@zorro-pc pipe]$ ls -l pipe
prw-r--r-- 1 zorro zorro 0 Jul 14 10:44 pipe
```

可以看到创建出来的文件类型比较特殊，是p类型。表示这是一个管道文件。有了这个管道文件，系统中就有了对一个管道的全局名称，于是任何两个不相关的进程都可以通过这个管道文件进行通信了。比如我们现在让一个进程写这个管道文件：

```
[zorro@zorro-pc pipe]$ echo xxxxxxxxxxxxxx > pipe
```

此时这个写操作会阻塞，因为管道另一端没有人读。这是内核对管道文件定义的默认行为。此时如果有进程读这个管道，那么这个写操作的阻塞才会解除：

```
[zorro@zorro-pc pipe]$ cat pipe
xxxxxxxxxxxxxx
```

大家可以观察到，当我们cat完这个文件之后，另一端的echo命令也返回了。这就是命名管道。

Linux系统无论对于命名管道和匿名管道，底层都用的是同一种文件系统的操作行为，这种文件系统叫pipefs。大家可以在/etc/proc/filesystems文件中找到你的系统是不是支持这种文件系统：

```
[zorro@zorro-pc pipe]$ cat /proc/filesystems |grep pipefs
```

```
nodev   pipefs
```

观察完了如何在命令行中使用管道之后，我们再来看看如何在系统编程中使用管道。

PIPE

我们可以把匿名管道和命名管道分别叫做PIPE和FIFO。这主要因为在系统编程中，创建匿名管道的系统调用是pipe()，而创建命名管道的函数是mkfifo()。使用mknod()系统调用并指定文件类型为为S_IFIFO也可以创建一个FIFO。

使用pipe()系统调用可以创建一个匿名管道，这个系统调用的原型为：

```
#include <unistd.h>

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

这个方法将会创建出两个文件描述符，可以使用pipefd这个数组来引用这两个描述符进行文件操作。pipefd[0]是读方式打开，作为管道的读描述符。pipefd[1]是写方式打开，作为管道的写描述符。从管道写端写入的数据会被内核缓存直到有人从另一端读取为止。我们来看一下如何在一个进程中使用管道，虽然这个例子并没有什么意义：

```
[zorro@zorro-pc pipe]$ cat pipe.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define STRING "hello world!"

int main()
```

```
{
    int pipefd[2];
    char buf[BUFSIZ];

    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    if (write(pipefd[1], STRING, strlen(STRING)) < 0) {
        perror("write()");
        exit(1);
    }

    if (read(pipefd[0], buf, BUFSIZ) < 0) {
        perror("write()");
        exit(1);
    }

    printf("%s\n", buf);

    exit(0);
}
```

这个程序创建了一个管道，并且对管道写了一个字符串之后从管道读取，并打印在标准输出上。用一个图来说明这个程序的状态就是这样的：

一个进程自己给自己发送消息这当然不叫进程间通信，所以实际情况中我们不会在单个进程中使用管道。进程在pipe创建完管道之后，往往都要fork产生子进程，成为如下图表示的样子：

如图中描述，fork产生的子进程会继承父进程对应的文件描述符。利用这个特性，父进程

先pipe创建管道之后，子进程也会得到同一个管道的读写文件描述符。从而实现了父子两个进程使用一个管道可以完成半双工通信。此时，父进程可以通过fd[1]给子进程发消息，子进程通过fd[0]读。子进程也可以通过fd[1]给父进程发消息，父进程用fd[0]读。程序实例如下：

```
[zorro@zorro-pc pipe]$ cat pipe_parent_child.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STRING "hello world!"

int main()
{
    int pipefd[2];
    pid_t pid;
    char buf[BUFSIZ];

    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
        /* this is child. */
        printf("Child pid is: %d\n", getpid());
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }
```

```
        printf("%s\n", buf);

        bzero(buf, BUFSIZ);
        snprintf(buf, BUFSIZ, "Message from child: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
        }

    } else {
        /* this is parent */
        printf("Parent pid is: %d\n", getpid());

        snprintf(buf, BUFSIZ, "Message from parent: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
        }

        sleep(1);

        bzero(buf, BUFSIZ);
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }

        printf("%s\n", buf);

        wait(NULL);
    }


    exit(0);
}
```

父进程先给子进程发一个消息，子进程接收到之后打印消息，之后再给父进程发消息，父进程再打印从子进程接收到的消息。程序执行效果：

```
[zorro@zorro-pc pipe]$ ./pipe_parent_child
Parent pid is: 8309
Child pid is: 8310
Message from parent: My pid is: 8309
Message from child: My pid is: 8310
```

从这个程序中我们可以看到，管道实际上可以实现一个半双工通信的机制。使用同一个管道的父子进程可以分时给对方发送消息。我们也可以看到对管道读写的一些特点，即：

在管道中没有数据的情况下，对管道的读操作会阻塞，直到管道内有数据为止。当一次写的数据量不超过管道容量的时候，对管道的写操作一般不会阻塞，直接将要写的数据写入管道缓冲区即可。

如果觉得本文还不错，请扫码任意打赏捐助穷佐罗。

当然写操作也不会再所有情况下都不阻塞。这里我们要先来了解一下管道的内核实现。上文说过，管道实际上就是内核控制的一个内存缓冲区，既然是缓冲区，就有容量上限。我们把管道一次最多可以缓存的数据量大小叫做PIPESIZE。内核在处理管道数据的时候，底层也要调用类似read和write这样的方法进行数据拷贝，这种内核操作每次可以操作的数据量也是有限的，一般的操作长度为一个page，即默认为4k字节。我们把每次可以操作的数据量长度叫做PIPEBUF。POSIX标准中，对PIPEBUF有长度限制，要求其最小长度不得低于512字节。PIPEBUF的作用是，内核在处理管道的时候，如果每次读写操作的数据长度不大于PIPEBUF时，保证其操作是原子的。而PIPESIZE的影响是，大于其长度的写操作会被阻塞，直到当前管道中的数据被读取为止。

在Linux 2.6.11之前，PIPESIZE和PIPEBUF实际上是一样的。在这之后，Linux重新实现了一个管道缓存，并将它与写操作的PIPEBUF实现成了不同的概念，形成了一个默认长度为65536字节的PIPESIZE，而PIPEBUF只影响相关读写操作的原子性。从Linux 2.6.35

之后，在fcntl系统调用方法中实现了F_GETPIPE_SZ和F_SETPIPE_SZ操作，来分别查看当前管道容量和设置管道容量。管道容量容量上限可以在/proc/sys/fs/pipe-max-size进行设置。

```
 #define BUFSIZE 65536

......

ret = fcntl(pipefd[1], F_GETPIPE_SZ);
if (ret < 0) {
    perror("fcntl()");
    exit(1);
}

printf("PIPESIZE: %d\n", ret);

ret = fcntl(pipefd[1], F_SETPIPE_SZ, BUFSIZE);
if (ret < 0) {
    perror("fcntl()");
    exit(1);
}

......
```

PIPEBUF和PIPESIZE对管道操作的影响会因为管道描述符是否被设置为非阻塞方式而有行为变化，n为要写入的数据量时具体为：

O_NONBLOCK关闭，n <= PIPE_BUF：

n个字节的写入操作是原子操作，write系统调用可能会因为管道容量(PIPESIZE)没有足够的空间存放n字节长度而阻塞。

O_NONBLOCK打开，n <= PIPE_BUF：

如果有足够的空间存放n字节长度，write调用会立即返回成功，并且对数据进行写操作。

空间不够则立即报错返回，并且errno被设置为EAGAIN。

O_NONBLOCK关闭，n > PIPE_BUF:

对n字节的写入操作不保证是原子的，就是说这次写入操作的数据可能会跟其他进程写这个管道的数据进行交叉。当管道容量长度低于要写的数据长度的时候write操作会被阻塞。

O_NONBLOCK打开，n > PIPE_BUF:

如果管道空间已满。write调用报错返回并且errno被设置为EAGAIN。如果没满，则可能会写入从1到n个字节长度，这取决于当前管道的剩余空间长度，并且这些数据可能跟别的进程的数据有交叉。

以上是在使用半双工管道的时候要注意的事情，因为在这种情况下，管道的两端都可能有多个进程进行读写处理。如果再加上线程，则事情可能变得更复杂。实际上，我们在使用管道的时候，并不推荐这样来用。管道推荐的使用方法是其单工模式：即只有两个进程通信，一个进程只写管道，另一个进程只读管道。实现为:

```
[zorro@zorro-pc pipe]$ cat pipe_parent_child2.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STRING "hello world!"

int main()
{
    int pipefd[2];
    pid_t pid;
    char buf[BUFSIZ];
```

```
    if (pipe(pipefd) == -1) {
        perror("pipe()");
        exit(1);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork()");
        exit(1);
    }

    if (pid == 0) {
        /* this is child. */
        close(pipefd[1]);

        printf("Child pid is: %d\n", getpid());
        if (read(pipefd[0], buf, BUFSIZ) < 0) {
            perror("write()");
            exit(1);
        }

        printf("%s\n", buf);

    } else {
        /* this is parent */
        close(pipefd[0]);

        printf("Parent pid is: %d\n", getpid());

        snprintf(buf, BUFSIZ, "Message from parent: My pid is: %d", getpid());
        if (write(pipefd[1], buf, strlen(buf)) < 0) {
            perror("write()");
            exit(1);
        }

        wait(NULL);
    }


    exit(0);
}
```

这个程序实际上比上一个要简单，父进程关闭管道的读端，只写管道。子进程关闭管道的写端，只读管道。整个管道的打开效果最后成为下图所示：

此时两个进程就只用管道实现了一个单工通信，并且这种状态下不用考虑多个进程同时对管道写产生的数据交叉的问题，这是最经典的管道打开方式，也是我们推荐的管道使用方式。另外，作为一个程序员，即使我们了解了Linux管道的实现，我们的代码也不能依赖其特性，所以处理管道时该越界判断还是要判断，该错误检查还是要检查，这样代码才能更健壮。

FIFO

命名管道在底层的实现跟匿名管道完全一致，区别只是命名管道会有一个全局可见的文件名以供别人open打开使用。再程序中创建一个命名管道文件的方法有两种，一种是使用mkfifo函数。另一种是使用mknod系统调用，例子如下：

```
[zorro@zorro-pc pipe]$ cat mymkfifo.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

    if (argc != 2) {
        fprintf(stderr, "Argument error!\n");
        exit(1);
    }

/*
    if (mkfifo(argv[1], 0600) < 0) {
        perror("mkfifo()");
```

```
        exit(1);
    }
*/
    if (mknod(argv[1], 0600|S_IFIFO, 0) < 0) {
        perror("mknod()");
        exit(1);
    }

    exit(0);
}
```

我们使用第一个参数作为创建的文件路径。创建完之后，其他进程就可以使用open()、read()、write()标准文件操作等方法进行使用了。其余所有的操作跟匿名管道使用类似。需要注意的是，无论命名还是匿名管道，它的文件描述都没有偏移量的概念，所以不能用lseek进行偏移量调整。

关于管道的其它议题，比如popen、pclose的使用等话题，《UNIX环境高级编程》中的相关章节已经讲的很清楚了。如果想学习补充这些知识，请参见此书。

最后

希望这些内容对大家进一步深入了解管道有帮助。如果有相关问题，可以在我的微博、微信或者博客上联系我。

如果觉得本文还不错，请扫码任意打赏捐助穷佐罗。

大家好，我是Zorro！

如果你喜欢本文，欢迎在微博上搜索“**orroz**”关注我，地址是：http://weibo.com/orroz

大家也可以在微信上搜索：**Linux系统技术** 关注我的公众号。

我的所有文章都会沉淀在我的个人博客上，地址是：http://liwei.life。

欢迎使用以上各种方式一起探讨学习，共同进步。

公众号二维码:

共享此文章：

相关:

Linux进程间通信-共享内存

八月 8, 2016
类似文章

Linux的进程间通信-文件和文件锁

七月 31, 2016
类似文章

SHELL编程之内建命令

六月 13, 2016
在“shell编程”中

zorro / 七月 18, 2016 / Uncategorized

穷佐罗的Linux书 / 自豪地采用WordPress