linux管道-应用编程-专题讲座

written by 王保明

# linux管道基本概念

## 1管道基本概念及操作

|  |
| --- |
| 管道基本概念   * 管道是Unix中最古老的进程间通信的形式。 * 我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”   比如：eg: ls | wc -l ps –u wbm01|grep “aa” |
| 管道的本质====》固定大小的内核缓冲区 |
| 管道限制   * 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道 * 只能用于具有共同祖先的进程（具有亲缘关系的进程）之间进行通信；通常，一个管道由一个进程创建，然后该进程调用fork，此后父、子进程之间就可应用该管道**。** |
| 匿名管道pipe   * 包含头文件<unistd.h> * 功能:创建一无名管道 * 原型   + int pipe(int fd[2]); * 参数   + fd：文件描述符数组,其中fd[0]表示读端, fd[1]表示写端 * 返回值:成功返回0，失败返回错误代码 |
| 管道创建后示意图 |
| 管道基本编程示例 |

## 2管道和文件描述符在一起

|  |
| --- |
| 管道命令的编程实现：  ls | wc –w |
| 文件描述符的深入理解 |
| #include <unistd.h>  #include <sys/stat.h>  #include <sys/wait.h>  #include <sys/types.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #include <errno.h>  #include <string.h>  #include <signal.h>  #define ERR\_EXIT(m) \  do \  { \  perror(m); \  exit(EXIT\_FAILURE); \  } while(0)  int main21(void )  {  int pipefd[2];  pid\_t pid;  if (pipe(pipefd) == -1 )  {  printf("pipe() err..\n");  return -1;  }  pid = fork();  if (pid == -1)  {  printf("fork err..\n");  return -1;  }  if (pid == 0)  {  close(pipefd[0]);  //复制文件描述符pipefd[1]，给标准输出，言外之意：execlp的ls命令输出到管道中  dup2(pipefd[1], STDOUT\_FILENO);  close(pipefd[1]);    execlp("ls", "ls", NULL);  //如果替换新的进程印象失败，则会执行下面一句话  sprintf(stderr, "execute the cmd ls err..\n");  exit(0);      }  else if (pid > 0 )  {  int len = 0;  char buf[100] = {0};  close(pipefd[1]);  //复制文件描述符pipefd[0]，给标准输入，言外之意：execlp的wc命令从管道中读  dup2(pipefd[0], STDIN\_FILENO);  close(pipefd[0]);  //len = read(pipefd[0], buf, 100);  execlp("wc", "wc", "-w", NULL);  printf("len:%d, buf:%s \n", len , buf);  //close(pipefd[0]);  }  wait(NULL);  printf("parent ..quit\n");  return 0;  }  int main(int argc, char \*argv[])  {  close(0); //关闭表示输入  open("makefile", O\_RDONLY); //makefile文件变成标准输入  close(1);//关闭标准输出  open("makefile2", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); //maifle2变成标准输出  execlp("cat", "cat", NULL); //替换进程印象后，执行cat命令    //cat命名从标准输入中按行读，紧接着写到标准输出  return 0;  } |
| O\_TRUNC：打开文件并清空 |

# 2管道的读写规则

|  |
| --- |
| 管道读写规则研究（重点）   * 当没有数据可读时   + O\_NONBLOCK disable：read调用阻塞，即进程暂停执行，一直等到有数据来到为止。   + O\_NONBLOCK enable：read调用返回-1，errno值为EAGAIN。 * 当管道满的时候   + O\_NONBLOCK disable： write调用阻塞，直到有进程读走数据   + O\_NONBLOCK enable：调用返回-1，errno值为EAGAIN * 当管道不停的被写，写满的时候   + O\_NONBLOCK disable： write调用阻塞   + O\_NONBLOCK enable：调用返回-1，errno值为EAGAIN * 如果所有管道写端对应的文件描述符被关闭，则read返回0 * 如果所有管道读端对应的文件描述符被关闭，则write操作会产生信号SIGPIPE * 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。和系统页有关，一般为64k，ulimit -a man 7 pipe * 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。因为阻塞或者非租塞返回导致的cpu调度 |
| 测试管道读(阻塞、非阻塞)  测试管道两端读、写均关闭  测试管道容量 |
| 编程实践  man fctl  File descriptor flags  The following commands manipulate the flags associated with a file descriptor. Currently, only one such  flag is defined: FD\_CLOEXEC, the close-on-exec flag. If the FD\_CLOEXEC bit is 0, the file descriptor will  remain open across an execve(2), otherwise it will be closed.  F\_GETFD  Read the file descriptor flags.  F\_SETFD  Set the file descriptor flags to the value specified by arg.  File status flags  Each open file description has certain associated status flags, initialized by open(2) and possibly modified  by fcntl(2). Duplicated file descriptors (made with dup(), fcntl(F\_DUPFD), fork(), etc.) refer to the same  open file description, and thus share the same file status flags.  The file status flags and their semantics are described in open(2).  F\_GETFL  Read the file status flags.  F\_SETFL  Set the file status flags to the value specified by arg. File access mode (O\_RDONLY, O\_WRONLY,  O\_RDWR) and file creation flags (i.e., O\_CREAT, O\_EXCL, O\_NOCTTY, O\_TRUNC) in arg are ignored. On  Linux this command can only change the O\_APPEND, O\_ASYNC, O\_DIRECT, O\_NOATIME, and O\_NONBLOCK flags. |
| 容量查询  man -6 pipe  Pipe Capacity  A pipe has a limited capacity. If the pipe is full, then a write(2) will block or fail, depending on  whether the O\_NONBLOCK flag is set (see below). Different implementations have different limits for the  pipe capacity. Applications should not rely on a particular capacity: an application should be designed so  that a reading process consumes data as soon as it is available, so that a writing process does not remain  blocked.  In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096  bytes on x86). Since Linux 2.6.11, the pipe capacity is 65536 bytes. |
| 读写规则  man -6 pipe  If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read(2)  from the pipe will see end-of-file (read(2) will return 0). If all file descriptors referring to the read  end of a pipe have been closed, then a write(2) will cause a SIGPIPE signal to be generated for the calling  process. If the calling process is ignoring this signal, then write(2) fails with the error EPIPE. An  application that uses pipe(2) and fork(2) should use suitable close(2) calls to close unnecessary duplicate  file descriptors; this ensures that end-of-file and SIGPIPE/EPIPE are delivered when appropriate.  It is not possible to apply lseek(2) to a pipe. |

|  |
| --- |
| #include <linux/limits.h>  PIPE\_BUF |
| [root@localhost ~]# find / -name "limits.h"  /usr/include/linux/limits.h  [root@localhost ~]# find / -name "limits.h" | cat  [root@localhost ~]# find / -name "limits.h" | sed -n "1p"  [root@localhost ~]# cat /usr/include/linux/limits.h  #ifndef \_LINUX\_LIMITS\_H  #define \_LINUX\_LIMITS\_H  #define NR\_OPEN 1024  #define NGROUPS\_MAX 65536 /\* supplemental group IDs are available \*/  #define ARG\_MAX 131072 /\* # bytes of args + environ for exec() \*/  #define CHILD\_MAX 999 /\* no limit :-) \*/  #define OPEN\_MAX 256 /\* # open files a process may have \*/  #define LINK\_MAX 127 /\* # links a file may have \*/  #define MAX\_CANON 255 /\* size of the canonical input queue \*/  #define MAX\_INPUT 255 /\* size of the type-ahead buffer \*/  #define NAME\_MAX 255 /\* # chars in a file name \*/  #define PATH\_MAX 4096 /\* # chars in a path name including nul \*/  #define PIPE\_BUF 4096 /\* # bytes in atomic write to a pipe \*/  #define XATTR\_NAME\_MAX 255 /\* # chars in an extended attribute name \*/  #define XATTR\_SIZE\_MAX 65536 /\* size of an extended attribute value (64k) \*/  #define XATTR\_LIST\_MAX 65536 /\* size of extended attribute namelist (64k) \*/  #define RTSIG\_MAX 32  #endif  [root@localhost ~]# |
| man 7 pipe  Pipe Capacity  A pipe has a limited capacity. If the pipe is full, then a write(2) will block or fail, depending on whether the O\_NONBLOCK flag is set (see below). Different implementations have different limits for the pipe capacity. Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.  In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on x86). Since Linux 2.6.11, the pipe capacity is 65536 bytes. |

# 3 FIFO管道

|  |
| --- |
| 命名管道   * 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。 * 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。 * 命名管道是一种特殊类型的文件 |
| 命名管道和匿名管道区别   * 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。 * 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。 * 命名管道是一种特殊类型的文件 |
| 创建一个命名管道   * 命名管道可以从命令行上创建，命令行方法是使用下面这个命令：   $ mkfifo filename   * 命名管道也可以从程序里创建，相关函数有：   int mkfifo(const char \*filename,mode\_t );   * man 3 mkfifo |
| 匿名管道与命名管道PK   * 匿名管道由pipe函数创建并打开。 * 命名管道由mkfifo函数创建，打开用open * FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，一但这些工作完成之后，它们具有相同的语义。 |
| 命名管道的打开规则   * 如果当前打开操作是为读而打开FIFO时   + O\_NONBLOCK disable：阻塞直到有相应进程为写而打开该FIFO   + O\_NONBLOCK enable：立刻返回成功 * 如果当前打开操作是为写而打开FIFO时   + O\_NONBLOCK disable：阻塞直到有相应进程为读而打开该FIFO   + O\_NONBLOCK enable：立刻返回失败，错误码为ENXIO   命名管道的读写规则   * 同匿名管道 |
| mknod  管道文件不能使用vi打开。。就是vi不能打开看管道内容 |
|  |

# 4FIFO管道编程实践

|  |
| --- |
| 利用管道，两个进程间进行文件复制。  fifow  读文件1.txt  写入管道  fifor  读管道  写入2.txt文件 |
| FIFO(7) Linux Programmer's Manual FIFO(7)  NAME  fifo - first-in first-out special file, named pipe |
| man 3 mkfifo  mkfifo - make a FIFO special file (a named pipe)  SYNOPSIS  #include <sys/types.h>  #include <sys/stat.h>  int mkfifo(const char \*pathname, mode\_t mode);  DESCRIPTION  mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions. It is modi-  fied by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).  A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an  anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo().  Once you have created a FIFO special file in this way, any process can open it for reading or writing, in  the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can  proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some  other process opens the same FIFO for writing, and vice versa. See fifo(7) for non-blocking handling of FIFO  special files.  RETURN VALUE  On success mkfifo() returns 0. In the case of an error, -1 is returned (in which case, errno is set appro-  priately). |
|  |

ps -u 用户

发信号

kill -l

man 7 signal