## 第6课 讲稿（进程间通信-1）

1. **Linux 下进程间通信概述**

进程是一段程序的执行过程。这里所说的进程一般是指运行在**用户态（对应于内核态）**的进程，而由于处于用户态的不同进程之间是彼此隔离的，就像处于不同城市的人们，它们必须通过某种方式来进行通信，例如人们使用手机等方式相互通信。

* 用户态：user mode，非特权状态，每个进程都在各自的用户空间中运行，而不允许存取其他程序的用户空间。
* 用户空间：Linux系统将自身划分为两部分，一部分为核心软件，即kernel，也称作内核空间；另一部分为普通应用程序，称为用户空间。

就像人们有多种通信方式一样，进程之间的通信也有多种形式，目前在 Linux 中使用较多的进程间通信方式主要有以下几种：

1. 管道（pipe）和有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信；有名管道，除具有管道所具有的功能外，还允许无亲缘关系进程间的通信。

* 管道（pipe）：在进程中产生，进程结束也随之结束，如“ls | grep abc”；
* 有名管道（named pipe）：通过mkfifo或mknod创建的管道文件。

1. 信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。
2. 消息队列（Messge Queue）：消息队列是消息的链接表，它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。
3. 共享内存（Shared memory）：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以即时看到对方进程中对共享内存中数据的更新。这种通信方式需要某种同步机制，如互斥锁和信号量（信号灯）等。
4. 信号量（Semaphore）：又称为信号灯，主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。
5. 套接字（Socket）：这是一种更为一般的进程间通信机制，可用于网络中不同机器之间的进程间通信，应用非常广泛。
6. **管道和有名管道**
7. 管道概述

管道是Linux中一种很重要的通信方式，它是把一个程序的输出直接连接到另一个程序的输入。之前第3课讲稿中已经用使用过有名管道，这里主要介绍**无名管道**，它具有如下特点：

* 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）；
* 它具有固定的读端和写端；
* 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的read()和write()等函数。但是它不是普通的文件，不属于任何文件系统，只存在于内核的内存空间中；
* 进程退出，管道也随之关闭。

1. 管道系统调用
2. 管道创建与关闭说明

管道是基于文件描述符的通信方式，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，如下图所示，这样就构成了一个通道。



管道关闭时只需将这两个文件描述符关闭即可，可使用普通的close()函数逐个关闭各个文件描述符。

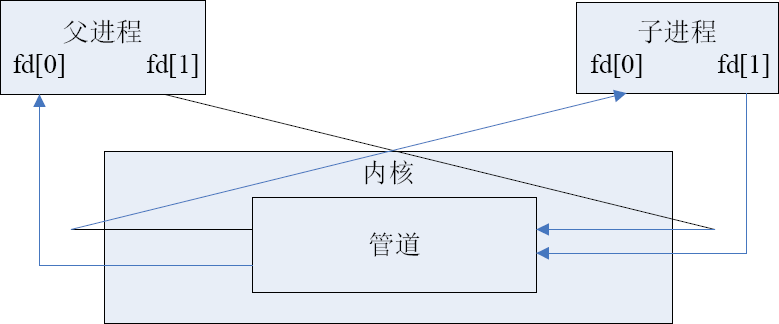
1. 管道创建函数

创建管道可以通过调用 pipe()来实现，表1列出了pipe()函数的语法要点。

|  |  |
| --- | --- |
| 表1 pipe()函数语法要点 | |
| 所需头文件 | #include <unistd.h> |
| 函数原型 | int pipe(int fd[2]) |
| 函数传入值 | fd[2]：管道的两个文件描述符，pipe之后就可以直接操作这两个文件描述符 |
| 返回值 | 成功：0 |
| 出错：-1 |

1. 管道读写说明

用pipe()函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是创建一个管道，再通过fork()函数创建一子进程，该子进程会继承父进程所创建的管道，这时，父子进程管道的文件描述符对应关系如下图所示：



此时的关系看似非常复杂，实际上却已经给不同进程之间的读写创造了很好的条件。父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。例如在下图中将父进程的写端 fd[1]和子进程的读端 fd[0]关闭。此时，父子进程之间就建立起了一条“子进程写入父进程读取”的通道。



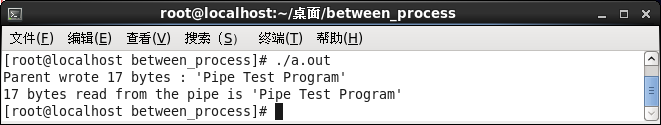
同样，也可以关闭父进程的 fd[0]和子进程的 fd[1]，这样就可以建立一条“父进程写入子进程读取”的通道。另外，父进程还可以创建多个子进程，各个子进程都继承了相应的 fd[0]和 fd[1]，这时，只需要关闭相应端口就可以建立其各子进程之间的通道。

1. 管道使用实例

本例主要建立一条父进程写、子进程读的管道。首先创建管道，然后父进程使用fork()函数创建子进程，再关闭父进程的读描述符和子进程的写描述符，建立起它们之间的管道通信。

|  |
| --- |
| /\* pipe.c \*/  #include <unistd.h>  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_DATA\_LEN 256  #define DELAY\_TIME 1  int main()  {  pid\_t pid;  int pipe\_fd[2];  char buf[MAX\_DATA\_LEN];  const char data[] = "Pipe Test Program";  int real\_read, real\_write;  memset((void\*)buf, 0, sizeof(buf));  /\* 创建管道 \*/  if (pipe(pipe\_fd) < 0)  {  printf("pipe create error\n");  exit(1);  }  /\* 创建一子进程 \*/  if ((pid = fork()) == 0)  {  /\* 子进程关闭写描述符，并使子进程暂停1s等待父进程已关闭相应的读描述符 \*/  close(pipe\_fd[1]);  sleep(DELAY\_TIME);  /\* 子进程读取管道内容 \*/  if ((real\_read = read(pipe\_fd[0], buf, MAX\_DATA\_LEN)) > 0)  {  printf("%d bytes read from the pipe is '%s'\n", real\_read, buf);  }  /\* 关闭子进程读描述符 \*/  close(pipe\_fd[0]);  exit(0);  }  else if (pid > 0)  {  /\* 父进程关闭读描述符，写入相应信息 \*/  close(pipe\_fd[0]);  if((real\_write = write(pipe\_fd[1], data, strlen(data))) != -1)  {  printf("Parent wrote %d bytes : '%s'\n", real\_write, data);  }  /\*关闭父进程写描述符\*/  close(pipe\_fd[1]);  /\*收集子进程退出信息\*/  waitpid(pid, NULL, 0);  exit(0);  }  } |

该程序的运行结果如下所示：



1. 管道读写注意点

* 只有在管道的读端存在时，向管道写入数据才有意义。
* 向管道写入数据时，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。
* 父子进程在运行时，它们的先后次序并不能保证，因此，在这里为了保证父子进程已经关闭了相应的文件描述符，可在两个进程中调用sleep()函数，当然这种调用不是很好的解决方法，在后面会学习进程之间的同步与互斥机制。

1. 标准流管道
2. 标准流管道函数说明

与Linux的文件操作中有基于文件流的标准I/O操作一样，管道的操作也支持基于文件流的模式。**这种基于文件流的管道主要是用来创建一个连接到另一个进程的管道，这里的“另一个进程”也就是一个可以进行一定操作的可执行文件**，例如，执行“ ls -l”或者编写的程序“./pipe”等。由于这一类操作很常用，因此标准流管道就将一系列的创建过程合并到一个函数popen()中完成。popen()所完成的工作有以下几步：

* 创建一个管道；
* fork()一个子进程；
* 在父子进程中关闭不需要的文件描述符；
* 执行exec函数族调用；
* 执行函数中所指定的命令。

这个函数的使用可以大大减少代码的编写量，但同时也有一些不利之处，例如，它不如前面管道创建的函数那样灵活多样，并且用popen()创建的管道必须使用**标准I/O函数**进行操作，而不能使用前面的read()、write()一类不带缓冲的I/O函数。

与之相对应，关闭用popen()创建的流管道必须使用函数pclose()，该函数关闭标准I/O流，并等待命令执行结束。

1. 函数格式

popen()和pclose()函数格式如表2和表3所示。

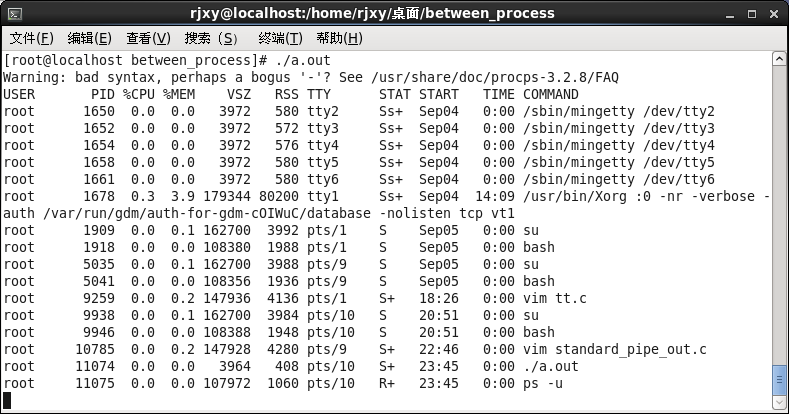
|  |  |  |
| --- | --- | --- |
| 表2 popen()函数语法要点 | | |
| 所需头文件 | #include | |
| 函数原型 | FILE \*popen(const char \*command, const char \*type) | |
| 函数传入值 | command：指向的是一个以null 结束符结尾的字符串，这个字符串包含一个shell命令，并由shell来执行 | |
| type | “r”：文件指针连接到command的标准输出，即该命令的结果产生输出  “w”：文件指针连接到command的标准输入，即该命令的结果产生输入 |
| 返回值 | 成功：文件流指针  当type=“r”时，可以读取命令的执行结果；当type=‘w’时，可以给命令输入参数 | |
| 出错：NULL | |

|  |  |
| --- | --- |
| 表3 pclose()函数语法要点 | |
| 所需头文件 | #include <stdio.h> |
| 函数原型 | int pclose(FILE \*stream) |
| 函数传入值 | stream：要关闭的文件流 |
| 返回值 | 成功：0 |
| 出错：-1 |

1. 函数使用实例
2. 在该实例中，测试type=‘r’的情况，使用popen()来执行“ps -u”命令。可以看出，popen()函数的使用能够使程序变得短小精悍。

|  |
| --- |
| /\* standard\_pipe\_r.c \*/  #include <stdio.h>  #include <stdlib.h>  #define BUFSIZE 1024  int main()  {  FILE \*fp;  char \*cmd = "ps -u";  char buf[BUFSIZE];  /\*调用popen()函数执行相应的命令\*/  if ((fp = **popen(cmd, "r")**) == NULL)  {  printf("Popen error\n");  exit(1);  }  while ((fgets(buf, BUFSIZE, fp)) != NULL)  {  printf("%s",buf);  }  pclose(fp);  exit(0);  } |

下面是该程序的执行结果。



1. 在该实例中，测试type=‘w’的情况，使用popen()来执行“cat > aaa”命令，并用fwrite()函数给命令输入参数。

|  |
| --- |
| /\* standard\_pipe.c \*/  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <fcntl.h>  #define BUFSIZE 1024  int main()  {  FILE \*fp;  char \*cmd = "cat > aaa";  char buf[BUFSIZE]="how do you do\n";  /\*调用popen()函数执行相应的命令\*/  if ((fp = popen(cmd, "w")) == NULL)  {  printf("Popen error\n");  exit(1);  }  fwrite(buf,BUFSIZE,1,fp);  pclose(fp);  exit(0);  } |

1. FIFO（有名管道）
2. 有名管道说明

前面介绍的管道是无名管道，它只能用于具有亲缘关系的进程之间，这就大大地限制了管道的使用。

有名管道的出现突破了这种限制，它可以使互不相关的两个进程实现彼此通信。该管道可以通过路径名来指出，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便。不过值得注意的是， **FIFO是严格地遵循先进先出规则的，对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾，它们不支持如 lseek()等文件定位操作**。

有名管道的创建可以使用函数mkfifo()，该函数类似文件中的open()操作，可以指定管道的路径和打开的模式。

1. 对于读进程。

* 若该管道是阻塞打开，且当前FIFO内没有数据，则对读进程而言将一直阻塞到有数据写入。
* 若该管道是非阻塞打开，则不论FIFO内是否有数据，读进程都会立即执行读操作。如果FIFO内没有数据，则读函数将立刻返回0。

1. 对于写进程。

* 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。
* 若该管道是非阻塞打开而不能写入全部数据，则写操作进行部分写入或者调用失败。

1. mkfifo()函数格式

表4列出了 mkfifo()函数的语法要点。

|  |  |
| --- | --- |
| 表4 mkfifo()函数语法要点 | |
| 所需头文件 | #include <sys/types.h>  #include <sys/state.h> |
| 函数原型 | int mkfifo(const char \*filename, mode\_t mode) |
| 函数传入值 | filename：要创建的管道 |
| mode：读写权限 |
| 返回值 | 成功：0 |
| 出错：-1 |

表5对FIFO 相关的出错信息做了归纳，方便查错：

|  |  |
| --- | --- |
| 表5 mkfifo相关的出错信息 | |
| EACCESS | 参数filename所指定的目录路径无可执行的权限 |
| EEXIST | 参数filename所指定的文件已存在 |
| ENAMETOOLONG | 参数filename的路径名称太长 |
| ENOENT | 参数filename包含的目录不存在 |
| ENOSPC | 文件系统的剩余空间不足 |
| ENOTDIR | 参数filename路径中的目录存在但却非真正的目录 |
| EROFS | 参数filename指定的文件存在于只读文件系统内 |

1. 使用实例

下面的实例包含了两个程序，一个用于读管道，另一个用于写管道。其中在读管道的程序里创建管道，并且作为main()函数里的参数由用户输入要写入的内容。读管道的程序会读出用户写入到管道的内容，这两个程序采用的是阻塞式读写管道模式。

以下是写管道的程序：

|  |
| --- |
| /\* fifo\_write.c \*/  #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define MYFIFO "myfifo" /\* 有名管道文件名\*/  #define MAX\_BUFFER\_SIZE PIPE\_BUF /\*PIPI\_BUF为管道的缓冲区大小，在limits.h中定义\*/  int main(int argc, char \* argv[]) /\*参数为即将写入的字符串\*/  {  int fd;  char buff[MAX\_BUFFER\_SIZE];  int nwrite;  if(argc <= 1)  {  printf("Usage: ./fifo\_write string\n");  exit(1);  }  sprintf(buff,"%s",argv[1]);  /\* 以只写阻塞方式打开FIFO 管道\*/  fd = open(MYFIFO, O\_WRONLY);  if (fd == -1)  {  printf("Open fifo file error\n");  exit(1);  }  /\*向管道中写入字符串\*/  if ((nwrite = write(fd, buff, MAX\_BUFFER\_SIZE)) > 0)  {  printf("Write '%s' to FIFO\n", buff);  }  close(fd);  exit(0);  } |

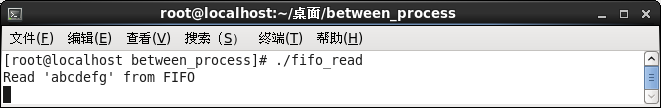
以下是读管道程序：

|  |
| --- |
| /\*fifo\_read.c\*/  #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #include <string.h>  #define MYFIFO "myfifo" /\* 有名管道文件名\*/  #define MAX\_BUFFER\_SIZE PIPE\_BUF /\*定义在于limits.h 中\*/  int main()  {  char buff[MAX\_BUFFER\_SIZE];  int fd;  int nread;  /\* 判断有名管道是否已存在，若尚未创建，则以相应的权限创建\*/  if ((mkfifo(MYFIFO, 0666) < 0) && (errno != EEXIST))  {  printf("Cannot create fifo file\n");  exit(1);  }  /\* 以只读阻塞方式打开有名管道\*/  fd = open(MYFIFO, O\_RDONLY);  if (fd == -1)  {  printf("Open fifo file error\n");  exit(1);  }  while (1)  {  memset(buff, 0, sizeof(buff));  if ((nread = read(fd, buff, MAX\_BUFFER\_SIZE)) > 0)  {  printf("Read '%s' from FIFO\n", buff);  }  }  close(fd);  exit(0);  } |

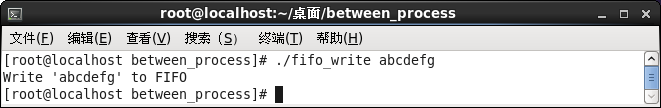
为了能够较好地观察运行结果，需要把这两个程序分别在两个终端里运行，在这里首先启动读管道程序。

读管道进程在建立管道之后就开始循环地从管道里读出内容，如果没有数据可读，则一直阻塞到写管道进程向管道写入数据。在启动了写管道程序后，读进程能够从管道里读出用户的输入内容，程序运行结果如下所示。

终端一：



终端二：



【注意1】由于读管道程序和写管道程序都是以阻塞方式打开的FIFO文件，所以写管道进程在读管道进程还未准备好时会阻塞等待，读管道进程在写管道进程没有准备好时也会阻塞等待。

如下所示，不开启读管道程序，直接运行写管道程序，那么进程就会阻塞等待：

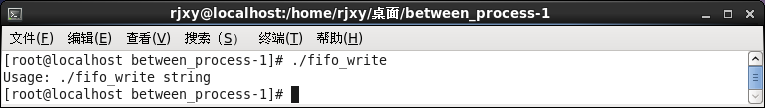


当读管道程序运行后，写管道进程阻塞解除，写入数据后退出：





【注意2】如果写管道程序以非阻塞方式打开FIFO文件，即把fifo\_write.c中的“open(MYFIFO, O\_WRONLY)”改为“open(MYFIFO, O\_WRONLY|O\_NONBLOCK)”，那么不开启读管道程序，直接运行写管道程序时，会遇到下面问题：



这种情况原因是写管道以只写方式（WRONLY）打开FIFO文件，对于管道来说，如果读端不开启，写操作没有意义。所以这里可以将写管道的方式改为读写方式（O\_RDWR），这样先运行写程序就不会出现问题，但是后续打开的读程序无法读到以前写入管道的信息。