Tubos ou Pipes de Comunicação

1. Introdução

Os tubos (ou pipes) constituem um mecanismo fundamental de comunicação unidirecional entre processos. Eles são um mecanismo de I/O com duas extremidades, ou socketc, correspondendo na verdade a filas de caractereres do tipo FIFO (First In First Out): as informações são introduzidas numa das extremidades (num socket) e retiradas em outra (outro socket). Por exemplo, quando o usuário executa o comando shell prog1|prog2, está se fazendo o chamado "piping" entre a saída de um programa prog1 para a entrada de outro prog2.

Os tubos são implementados como arquivos (eles possuem um i-node na tabela de indexação), mesmo se eles não têm um nome definido no sistema. Assim, o programa especifica sua entrada e sua saída somente como um descritor na tabela de índices, o qual aparece como uma variável ou constante, podendo a fonte (entrada) e o destino (saída) serem alteradas sem que para isso o texto do programa tenha que ser alterado. No caso do exemplo, na execução de prog1 a saída padrão (stdout) é substituída pela entrada do tubo. No prog2, de maneira similar, a entrada padrão (stdin) é substituída pela saída de prog1.

A técnica dos tubos é freqüentemente utilizada nos shells para redirecionar a saída padrão de um comando para a entrada de um outro.

2. Particularidades dos tubos

Uma vez que eles não têm nomes, os tubos de comunicação são temporários, existindo apenas em tempo de execução do processo que os criou;

A criação dos tubos é feita através de uma primitiva especial: pipe();

Vários processos podem fazer leitura e escrita sobre um mesmo tubo, mas nenhum mecanismo permite de diferenciar as informações na saída do tubo;

A capacidade é limitada (em geral a 4096 bytes). Se a escrita sobre um tubo continua mesmo depois do tudo estar completamente cheio, ocorre uma situaçào de bloqueio (dead-lock);

Os processos comunicando-se através dos tubos devem ter uma ligação de parentesco, e os tubos religando processos devem ter sidos abertos antes da criação dos filhos (veja a passagem de descritores de arquivos abertos durante a execução do fork() na seção 2.2.1);

É impossível fazer qualquer movimentação no interior de um tubo.

Com a finalidade de estabelecer um diálogo entre dois processos usando tubos, é necessário a abertura de um tubo em cada direção.

3. A Criação de um Tubo: A Primitiva pipe()

```
#include <unistd.h>

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

Valor de retorno: 0 se a criação tiver sucesso, e -1 em caso de falha.

A primitiva pipe() cria um par de descritores, apontando para um i-node, e coloca-os num vetor apontado por desc:

desc[0] contém o número do descritor pelo qual pode-se ler no tubo

desc[1] contém o número do descritor pelo qual pode-se escrever no tubo

Assim, a escrita sobre desc[1] introduz dados no tubo, e a leitura em desc[0] extrai dados do tubo.

4. Segurança do sistema

No caso em que todos os descritores associados aos processos susceptíveis de ler num tubo estiverem fechados, um processo que tenta escrever neste tubo deve receber um sinal SIGPIPE, sendo então interrompido se ele não possuir uma rotina de tratamento deste sinal.

Se um tubo está vazio, ou se todos os descritores susceptíveis de escrever sobre ele estiverem fechados, a primitiva read() retornará o valor 0 (fim de arquivo lido).

Exemplo 1: emissão de um sinal SIGPIPE

```
/* arquivo test_pipe_sig.c */
/* teste de escrita num tubo fechado a leitura */

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void it_sigpipe()
{
     printf("Sinal SIGPIPE recebido \n") ;
}
int main()
{
     int p_desc[2] ;
     signal(SIGPIPE,it_sigpipe) ;
     pipe(p_desc) ;
     close(p_desc[0]) ;  /* fechamento do tubo em leitura */
     if (write(p_desc[1],"0",1) == -1)
          perror("Error write") ;
     exit(0);
}
```

Resultado da execução:

test_pipe_sig

Sinal SIGPIPE recebido
Error write: Broken pipe

Neste exemplo, tenta-se escrever num tubo sendo que ele acaba de ser fechado em leitura; o sinal SIGPIPE é emitido e o programa é desviado para a rotina de tratamento deste sinal. No retorno, a primitiva write() retorna -1 e perror imprime na tela a mensagem de erro.

Exemplo 2: Leitura num tubo fechado em escrita.

```
/* arquivo test_pipe_read.c */
/* teste de leitura num tubo fechado em escrita */

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  int i, p_desc[2] ;
  char c ;
        pipe(p_desc) ; /* criacao do tubo */
        write(p_desc[1],"AB",2) ; /* escrita de duas letras no tubo */
        close(p_desc[1]) ;        /* fechamento do tubo em escrita */

        /* tentativa de leitura no tuno */
        for (i=1; i<=3; i++) {
          if ( (read(p_desc[0],&c,1) == 0) )
                        printf("Tubo vazio\n") ;
                else
                        printf("Valor lido: %c\n",c) ;
        }
        exit(0);
}
```

Resultado da execução:

test_pipe_read
Valor lido: A
Valor lido: B
Tubo vazio

Este exemplo mostra que a leitura num tubo é possível, mesmo se este tiver sido fechado para a escrita. Obviamento, quando o tubo estiver vazio, read() vai retornar o valor 0.

5. Aplicações das primitivas de entrada e saída

É possível utilizar as funções da biblioteca padrão sobre um tubo já aberto, associando a esse tubo - por meio da função fopen() - um ponteiro apontando sobre uma strutura do tipo FILE:

write() : os dados são escritos no tubo na ordem em que eles chegam. Quando o tubo está cheio, write() se bloqueia esperando que uma posição seja liberada. Pode-se evitar este bloqueio utilizando-se o flag O_NDELAY.

read() : os dados são lidos no tubo na ordem de suas chegadas. Uma vez retirados do tubo, os dados não poderão mais serem relidos ou restituídos ao tubo.

close() : esta função é mais importante no caso de um tubo que no caso de um arquivo. Não somente ela libera o descritor de arquivo, mas quando o descritor de arquivo de escritura está fechado, ela funciona como um fim de arquivo para a leitura.

dup() : esta primitiva combinada com pipe() permite a implementaçãão dos comandos religados por tubos, redirecionando a saída padrão de um comando para a entrada padrão de um outro.

5.1. Implementação de um comando com tubos

Este exemplo permite observar como as primitivas pipe() e dup() podem ser combinadas com o objetivo de produzir comandos shell do tipo ls|wc|wc. Note que é necessário fechar os descritores não utilizados pelos processos que executam a rotina.

```
/* arquivo test_pipe.c */
/* este programa é equivalente ao comando shell ls|wc|wc */

#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int p_desc1[2] ;
int p_desc2[2] ;

void faire_ls()
{
     /* saida padrao redirecionada para o 1o. tubo */
     close (1) ;
     dup(p_desc1[1]) ;
     close(p_desc1[1]) ;

     /* fechamento dos descritores nao-utilizados */
     close(p_desc1[0]) ;
     close(p_desc2[1]) ;
     close(p_desc2[0]) ;

     /* executa o comando */
     execlp("ls","ls",0) ;
     perror("impossivel executar ls ") ;
}

void faire_wc1()
{
     /* redirecionamento da entrada padrao para o 1o. tubo*/
     close(0) ;
     dup(p_desc1[0]) ;
     close(p_desc1[0]) ;
     close(p_desc1[1]) ;

     /* redirecionamento da saida padrao para o 2o. tubo*/
     close(1) ;
     dup(p_desc2[1]) ;
     close(p_desc2[1]) ;
```

```
     close(p_desc2[0]) ;

     /* executa o comando */
     execlp("wc","wc",0) ;
     perror("impossivel executar o 1o. wc") ;

}

void faire_wc2()
{
     /* redirecionamento da entrada padrao para o 2o. tubo*/
     close (0) ;
     dup(p_desc2[0]) ;
     close(p_desc2[0]) ;

     /* fechamento dos descritores nao-utilizados */
     close(p_desc2[1]) ;
     close(p_desc1[1]) ;
     close(p_desc1[0]) ;

     /* executa o comando */
     execlp("wc","wc",0) ;
     perror("impossivel executar o 2o. wc") ;
}

int main()
{
     /* criacao do primeiro tubo*/
     if (pipe(p_desc1) == -1)
          perror("Impossivel criar o 1o. tubo") ;

     /* criacao do segundo tubo */
     if (pipe(p_desc2) == -1)
          perror("impossivel criar o 1o. tubo") ;

     /* lancamento dos filhos */
     if (fork() == 0) faire_ls() ;
     if (fork() == 0) faire_wc1() ;
     if (fork() == 0) faire_wc2() ;
     exit(0);
}
```

Resultado da execução:

```
# ls|wc|wc
#    1    3    24
# test_pipe
#    1    3    24
```

5.2. Comunicação entre pais e filhos usando um tubo

Exemplo 1: Envio de uma mensagem ao usuário

Este programa permite que processos troquem mensagens com ajuda do sistema de correio eletrônico.

```
/* arquivo test_pipe_mail.c */
```

```
/* teste do envio de um mail usando tubos */

#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>

int main()
{
    FILE *fp;
    int pid, pipefds[2];
    char *username, *getlogin();
    /* da o nome do usuario */
    if ((username = getlogin()) == NULL) {
        fprintf(stderr, "quem e voce?\n");
        exit(1);    }
     /* Cria um tubo. Isto deve ser feito antes do fork para que
      * o filho possa herdar esse tubo
      */
    if (pipe(pipefds) < 0) { perror("Error pipe");
        exit(1);    }
    if ((pid = fork()) < 0) {    perror("Error fork");
        exit(1); }
     /* Codigo do filho:
      * executa o comando mail e entao envia ao username
      * a mensagem contida no tubo */
    if (pid == 0) {
  /* redirige a stdout para o tubo; o comando executado em seguida tera
   como entrada (uma mensagem) a leitura do tubo */
        close(0);
        dup(pipefds[0]);
        close(pipefds[0]);
  /* fecha o lado de escrita do tubo, para poder ver a saida na tela */
        close(pipefds[1]);
        /* executa o comando mail */
        execl("/bin/mail", "mail", username, 0);
        perror("Error execl");
        exit(1);
    }
    /* Codigo do pai:
     * escreve uma mensagem no tubo */
    close(pipefds[0]);
    fp = fdopen(pipefds[1], "w");
    fprintf(fp, "Hello from your program.\n");
    fclose(fp);
    /* Espera da morte do processo filho */
    while (wait((int *) 0) != pid) ;
    exit(0);
}
```

Resultado da execução: O usuário que executa o programa vai enviar a si mesmo um mensagem por correio eletrônico. A mensagem deve ser exatamente igual à mostrada a seguir:

Date: Fri, 13 Oct 2000 10:28:34 -0200
From: Celso Alberto Saibel Santos <saibel@leca.ufrn.br>
To: saibel@leca.ufrn.br

Hello from your program.

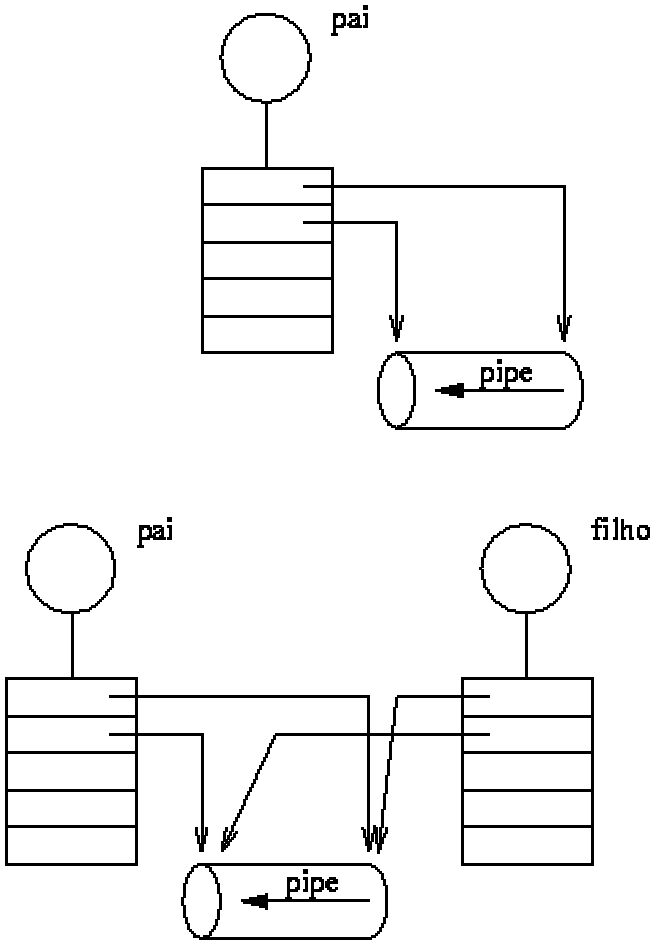

Figura 4.1: Compartilhamento de um tubo entre pai e filho.

Exemplo 2: Enfoca mais uma vez a herança dos descritores através de um fork(). O programa cria um tubo para a comunicação entre um processo pai e seu filho.

```
/* arquivo test_pipe_fork.c */
/* Testa heranca dos descritores na chamada do fork().
 * O programa cria um tubo e depois faz um fork. O filho
 * vai se comunicar com o pai atraves desse tubo
*/

#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#define DATA "Testando envio de mensagem usando pipes"

int main()
{
   int sockets[2], child;
   char buf[1024];
   /* criacao de um tubo */
   if ( pipe(sockets) == -1 ) {
          perror("Error opening stream socket pair") ;
          exit(10);
```

```
   }
   /* criacao de um filho */
   if ( (child = fork()) == -1)
     perror ("Error fork") ;
   else if (child) {
     /* Esta ainda e a execução do pai. Ele lê a mensagem do filho */
     if ( close(sockets[1]) == -1) /* fecha o descritor nao utilizado */
          perror("Error close") ;
     if (read(sockets[0], buf, 1024) < 0 )
       perror("Error: reading message");
     printf("-->%s\n", buf);
     close(sockets[0]);
   } else {
     /* Esse e o filho. Ele escreve a mensagem para seu pai */
      if ( close(sockets[0]) == -1) /* fecha o descritor nao utilizado */
        perror("Error close") ;
      if (write(sockets[1], DATA, sizeof(DATA)) < 0 )
        perror("Error: writing message");
      close(sockets[1]);
   }
   sleep(1);
   exit(0);
}
```

Resultado da execução:

test_pipe_fork
-->Testando envio de mensagem usando pipes

Um tubo é criado pelo processo pai, o qual logo após faz um fork. Quando um processo faz um fork, a tabela de descritores do pai é automaticamente copiada para o processo filho.

No programa, o pai faz um chamada de sistema pipe() para criar um tubo. Esta rotina cria um tubo e inclui na tabela de descritores do processos os descritores para os sockets associados às duas extremidades do tubo. Note que as extremidades do tubo não são equivalentes: o sockets com índice 0 está sendo aberto para leitura, enquanto que o de índice 1 está sendo aberto somente para a escrita. Isto corresponde ao fato de que a entrada padrão na tabela de descritores é associada ao primeiro descritor, enquanto a saída padrão é associada ao segundo.

Após ser criado, o tubo será compartilhado entre pai e filho após a chamada fork.

A tabela de descritores do processo pai aponta para ambas as extremidades do tubo. Após o fork, ambas as tabelas do pai e do filho estarão apontando para o mesmo tubo (herança de descritores). O filho então usa o tubo para enviar a mensagem para o pai.

5.3. Utilização dos tubos

É possível que um processo utilize um tubo tanto para a escrita quanto para a leitura de dados. Este tubo não tem mais a função específica de fazer a comunicação entre processos, tornando-se muito mais uma implementação da estrutura de um arquivo. Isto permite, em certas máquinas, de ultrapassar o limite de tamanho da zona de dados. O mecanismo de comunicação por tubos apresenta um certo número de incovenientes como o não

armazenamento da informação no sistema e a limitação da classe de processos podendo trocar informações via tubos.

6. As FIFOS ou tubos com nome

Uma FIFO combina as propriedades dos arquivos e dos tubos:

Como um arquivo, ela tem um nome e todo processo que tiver as autorizações apropriadas pode abri-lo em leitura ou escrita (mesmo se ele não tiver ligação de parentesco com o criador do tubo). Assim, um tubo com nome, se ele não estiver destruído, persistirá no sistema, mesmo após a terminação do processo que o criou.

Uma vez aberto, uma FIFO se comporta muito mais como um tubo do que como um arquivo: os dados escritos são lidos na ordem "First In First Out", seu tamanho é limitado, e além disso, é impossível de se movimentar no interior do tubo.

6.1. Criação de um Tubo com Nome: Primitiva mknod()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Valor de retorno: 0 se a criação do tubo tem sucesso, e -1 caso contrário.

A primitiva mknod() permite a criação de um nó (arquivo normal ou especial, ou ainda um tubo) cujo nome é apontado por pathname, especificado por mode e dev. O argumento mode especifica os direitos de acesso e o tipo de nó a ser criado. O argumento dev não é usado na criação de tubos com nome, devendo ter seu valor igual a 0 neste caso.

A criação de um tubo com nome é o único caso onde o usuário normal tem o direito de utilizar esta primitiva, reservada habitualmente ao super-usuário. Afim de que a chamada mknod() tenha sucesso, é indispensável que o flag S_IFIFO esteja "setado" e nos parâmetros de mode os direitos de acesso ao arquivo estejam indicados: isto vai indicar ao sistema que uma FIFO vai ser criada e ainda que o usuário pode utilizar mknod() mesmo sem ser root.

Dentro de um programa, um tubo com nome pode ser eliminado através da primitiva unlink(const char *pathname), onde pathname indica o nome do tubo a ser destruído.

Exemplo:

```
/* arquivo test_fifo.c */
/* este programa mostra a criacao e destruicao de tubos
 * com nome
 */

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
     printf("Vou criar um tubo de nome 'fifo1'\n") ;
     printf("Vou criar um tubo de nome 'fifo2'\n") ;
     if (mknod("fifo1",S_IFIFO | O_RDWR, 0) == -1) {
          perror("Criacao de fifo1 impossivel") ;
          exit(1) ;
     }
     if (mknod("fifo2",S_IFIFO | O_RDWR, 0) == -1) {
          perror("Criacao de fifo2 impossivel") ;
          exit(1) ;
     }
     sleep(10) ;
     printf("Vou apagar o tubo de nome 'fifo1'\n") ;
     unlink("fifo1") ;
     exit(0);
}
```

Resultado da execução:

O programa é lançado em background e pode-se verificar (através do comando shell ls -l fifo*) que os tubos denominados fifo1 e fifo2 foram criados e depois, que o tubo fifo1 foi destruído.

```
# test_fifo &
[2] 812
Vou criar um tubo de nome 'fifo1'
Vou criar um tubo de nome 'fifo2'
# ls -l fifo*
p---------   1 saibel   prof            0 Sep 27 10:07 fifo1|
p---------   1 saibel   prof            0 Sep 27 10:07 fifo2|
# Vou apagar o tubo de nome 'fifo1'

[2]    Done                          test_fifo
euler:~/> ls -l fifo*
p---------   1 saibel   prof            0 Sep 27 10:07 fifo2|
```

Observações:

Note que a presença do bit p indica que fifo1 e fifo2 são tubos (pipes) com nome;

Pode-se notar ainda que o tubo denominado fifo2 permanece no sistema, mesmo após a morte do processo que o criou.

A eliminação de um tubo com nome pode ser feita a partir do shell, como no caso de um arquivo comum, usando-se o comando rm

6.2. Manipulação das FIFOs

As instruções read() e write() são bloqueantes:

Na tentativa de leitura de uma FIFO vazia, o processo ficará em espera até que haja um preenchimento suficiente de dados dentro da FIFO;

Na tentativa de escrita de uma FIFO cheia, o processo irá esperar que a FIFO seja sucessivamente esvaziada para começar a preenchê-la com seus dados.

Neste caso ainda, a utilização do flag O_NDELAY permite de manipular o problema de bloqueio, uma vez que nesse caso as funções read() e write() vão retornar um valor nulo.

Bibliografia

SANTOS, Celso Alberto Saibel. **Programação em Tempo Real**: Módulo II - Comunicação InterProcessos. UFRN, Laboratório de Engenharia de Computação e Automação. http://www.dca.ufrn.br/~adelardo/cursos/DCA409/all.html. Acessado em:16/03/2005.