FOLHA DE PROBLEMAS Nº 5

Pipes e *FIFOS*

---

1. – Comunicação unidireccional usando um *pipe*. Envio de dados de diferentes tipos.

a) Escreva um programa com a funcionalidade que se descreve a seguir, em que dois processos, pai e filho, comunicam entre si através de um *pipe*:

- o processo-pai lê, do teclado, dois números inteiros (guardando-os em variáveis do tipo inteiro) e envia-os ao processo-filho, através de um *pipe*;
- o processo-filho calcula a soma, a diferença, o produto e o quociente dos dois números e apresenta o resultado no écrã (note que o quociente de dois números inteiros pode não ser inteiro e que a operação de divisão por zero é inválida).

b) Altere o programa 1.a de modo a que os números sejam guardados nos campos de uma *struct* a enviar ao processo-filho. Note que esta *struct* pode ser enviada numa única chamada `write()`.

c) Altere o programa 1.a de modo a que os números sejam lidos do teclado para variáveis do tipo *string* e como tal sejam enviados ao processo-filho. Admita que o utilizador introduz *strings* representando números válidos.

2. – Comunicação bidireccional usando *pipes*.

Altere o programa 1.a por forma a que o processo-filho envie o resultado ao processo-pai através de um outro *pipe*, sendo os resultados apresentados no écrã apenas pelo processo-pai. Conceba e implemente um protocolo de comunicação, através deste *pipe*, que permita que o processo-filho informe o processo-pai sobre qual o tipo de resultados: inteiro, *float* ou inválido (no caso da divisão por zero).

3. – Redireccionamento da entrada/saída para um *pipe*.

Escreva um programa que tenha como argumento da linha de comando o nome de um ficheiro de texto contendo nomes de pessoas (um nome por linha de texto) e que mostre os nomes ordenados alfabeticamente. Use o utilitário `sort` para fazer a ordenação, ligando a sua entrada *standard* à saída de um *pipe* para cuja entrada deve ser enviado o conteúdo do ficheiro. Implemente duas versões do programa:

a) Uma versão em que a abertura do ficheiro, a sua leitura e a escrita no *pipe* são feitas no código do programa.

b) Uma versão em que o conteúdo do ficheiro é escrito no *pipe* usando o utilitário `cat`, ao qual deve ser passado o nome do ficheiro como argumento, ligando a saída *standard* deste utilitário à entrada do *pipe*.

4. – Execução de múltiplos comandos em *pipeline*.

Escreva um programa que execute os seguintes três comandos, ligando a saída de cada um à entrada do seguinte através de um *pipe*: `ls` *`dir`* `-laR`, `grep` *`arg`*, `sort`. Os argumentos *`dir`*, de `ls`, e *`arg`*, de `grep`, devem ser passados como argumentos do programa (ex: sendo `p04` o nome do executável deste programa, o comando `p04 ei00001 jpg`, deve dar origem à execução do seguinte *pipeline* de comandos: `ls ei00001 -laR | grep jpg | sort`. Os *pipes* devem ser criados pelo programa.

5. – Execução de múltiplos comandos em *pipeline*.

Escreva um programa que leia uma linha de comando constituída por um número arbitrário de comandos (ex: `ls ei00001 -laR | grep jpg | sort | more`) e os execute em *pipeline*, isto é, ligando a saída *standard* de cada comando à entrada *standard* do comando seguinte. Sugestão: use o código do problema 10.c da folha de problemas nº 1.

6. – Comunicação unidireccional usando um *FIFO*.

Considere os seguintes dois programas:

```
// PROGRAMA p06_reader.c
#include ...  //a completar

int readline(int fd, char *str);

int main(void)
{
 int   fd;
 char  str[100];

 mkfifo("/tmp/myfifo",0660);
 fd=open("/tmp/myfifo",O_RDONLY);
 while(readline(fd,str)) printf("%s",str);
 close(fd);
 return 0;
}

int readline(int fd, char *str)
{
 int n;

 do {
   n = read(fd,str,1);
 } while (n>0 && *str++ != '\0');

 return (n>0);
}

----------------------------------------

// PROGRAMA p06_writer.c
#include ...  //a completar

int main(void)
{
 int   fd, messagelen, i;
 char  message[100];

 do {
   fd=open("/tmp/myfifo",O_WRONLY);
   if (fd==-1) sleep(1);
 } while (fd==-1);

 for (i=1; i<=3; i++) {
   sprintf(message,"Hello no. %d from process %d\n", i, getpid());
   messagelen=strlen(message)+1;
   write(fd,message,messagelen);
   sleep(3);
 }
 close(fd);
 return 0;
}
```

a) Analise e interprete o código. Execute os programas, lançando-os em execução a partir de diferentes janelas de terminal. Experimente lançá-los em execução por ordem diferente.

b) Como pôde verificar o processo-leitor termina assim que o processo-escritor terminar. Explique por que é que isso acontece (sugestão: o que é que a chamada `read()` devolve quando o terminal de escrita do *FIFO* for fechado pelo processo-escritor ?). Altere o programa-leitor por forma a abrir o *FIFO* em modo de leitura e escrita. Volte a executar os programas e interprete o que acontecer. Qual a dificuldade que surgiu? Na prática, haverá alguma diferença entre fazer uma única chamada `open()` para abrir o FIFO em modo de leitura e escrita ou fazer duas chamadas, uma para leitura e outra para escrita ?

7. – Comunicação bidireccional usando *FIFOS.* Arquitectura cliente-servidor (1).

Resolva o problema 2 usando dois *FIFOS* com os nomes `fifo_req` e `fifo_ans` para a comunicação; `fifo_req` é o *FIFO* através do qual os pedidos devem ser enviados ao processo-calculador e `fifo_ans` é o *FIFO* através do qual este processo devolve os resultados. Neste caso, os dois processos, o que lê os números e apresenta os resultados dos cálculos (cliente) e o que faz estes cálculos (servidor) serão processos independentes, cujos executáveis poderão chamar-se, por exemplo, `p07_client` e `p07_server`. O processo-servidor deve manter-se em funcionamento até que os números a processar sejam ambos iguais a zero, situação em que não deve efectuar qualquer cálculo, mas apenas deve destruir os dois *FIFOS*. Desta forma torna-se possível que múltiplos clientes solicitem a execução de cálculos. Os *FIFOS* devem ser criados no directório `/tmp` pelo servidor.

8. – Comunicação unidireccional usando um *FIFO.*

Num sistema de computação existe um utilizador (utilizador do tipo S) a quem outros utilizadores (utilizadores do tipo C) devem enviar, através de um *FIFO*, uma mensagem, logo que cada um destes utilizadores iniciar o seu trabalho. O conteúdo da mensagem é apenas o nome do utilizador do tipo C que a enviou. Os utilizadores do tipo C enviam as mensagens ao utilizador do tipo S usando o programa `p08_chg`, ao qual deve ser passado como argumento o nome do utilizador (ex: o comando `p08_chg Xyz`, indica que o utilizador `Xyz` iniciou o seu trabalho). O utilizador do tipo S deve lançar em execução um processo, cujo executável se chama `p08_trl_chg`, que fique permanentemente, durante alguns minutos, a aguardar as mensagens enviadas pelos utilizadores do tipo C, apresentando-as no écran, sob a forma "`CHEGOU xxx`", em que `xxx` representa o conteúdo da mensagem, isto é, o nome do utilizador do tipo C que a enviou. Ao fim do tempo estabelecido, este processo deverá apresentar no écran o número total de mensagens que recebeu e terminar. Admita que o *FIFO*, cujo *pathname* é `/tmp/fifo_chg`, foi criado previamente, com as permissões de acesso adequadas. Note que a mensagem contendo o nome do utilizador, a enviar pelo programa `chg`, tem um comprimento que pode ser variável.
Escreva os programas `ctrl_chg` e `chg`.

9. – Comunicação bidireccional usando um *FIFO.* Arquitectura cliente-servidor (2).

Pretende-se implementar uma arquitectura cliente-servidor, escrevendo para isso dois programas diferentes - o servidor e o cliente. O servidor deverá criar um *FIFO* bem conhecido (por exemplo, `/tmp/fifo.s`) e ficar à espera que os clientes lhe enviem comandos. Os comandos são *strings* representando programas executáveis que podem ter ou não ter parâmetros na linha de comando. Quando o servidor lê um destes comandos do *FIFO* e executa-o (criando um novo processo), direccionando a sua saída *standard* para um ficheiro auxiliar. Após a execução envia o conteúdo do ficheiro auxiliar produzido (ou uma mensagem de erro, se não conseguiu executar o comando) para o cliente através de outro *FIFO* criado pelo cliente (por exemplo, `/tmp/fifo.`*`pid`*, em que *`pid`* é o identificador do cliente, também enviado para o servidor). O cliente simplesmente cria o *FIFO* para receber a resposta, envia o seu *pid* para o servidor e o comando para este executar, espera pela resposta e, quando esta chegar, mostra-a no écran e termina.