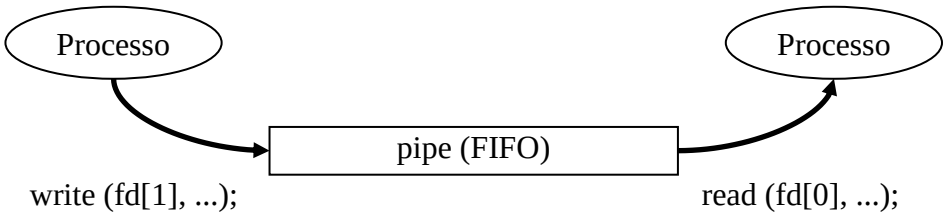
1. *Pipes* unidireccionais

Um *pipe* disponibiliza um mecanismo de comunicação e sincronização entre processos hierarquicamente relacionados. É constituído por um buffer de capacidade limitada sobre o qual se pode escrever e ler seguindo a ordem FIFO (*Fisrt In Fisrt Out*). É utilizado para o estabelecimento de um canal de comunicação unidireccional entre dois processos. Um dos processos utiliza o *pipe* para a escrita de dados que, por sua vez, serão lidos pelo outro processo.
A criação de um *pipe* é realizada pela chamada ao SO 'int pipe(int fd[2]);' (+ info em man 2 pipe). 'fd' é uma tabela de dois descritores para acesso ao pipe. 'fd[0]' representa o descritor para leitura e 'fd[1]' representa o descritor para escrita. A figura seguinte ilustra o mecanismo de comunicação de dados entre dois processos.



2. Criação, escrita e leitura

Considere o programa 'pipe1.c':

```
#include <unistd.h>
#include <stdio.h>
main()
{ int fd[2], i, j;
  pipe(fd); /* Criação do pipe */
  if(!fork()){ /* filho envia, pai recebe*/
    close(fd[0]); /* este processo não usa lado de leitura */
    for(i=0; i<10; i++)
      write(fd[1], &i, sizeof(int));
    close(fd[1]);
   } else {
    close(fd[1]); /* este processo não usa lado de escrita */
    for(i=0; i<10; i++)
     {read(fd[0], &j, sizeof(int));
      printf("%d\n", j);}
    close(fd[0]);
   }
}
```

O processo filho escreve uma sequência de números inteiros no *pipe*. Esta sequência de números é lida pelo processo pai. Teste o programa.
Devido às características da função 'fork()', ambos os processos têm acesso ao descritor para leitura e ao descritor para escrita. Sendo um mecanismo de comunicação unidireccional, cada um dos processos deve apenas usar um dos descritores (leitura ou escrita).
Na fase de terminação de um processo, todos os descritores são encerrados. Neste exemplo, todas as chamadas às funções 'close()' podem ser eliminadas sem alterar o comportamento do programa. Valide esta afirmação.
Considere o programa 'pipe2.c'

```
#include <stdio.h>
#include <unistd.h>
main()
{
 int fd[2], i, n;
 pipe(fd); /* Criação do pipe */
```

```
 if(!fork()){ /* filho envia, pai recebe */
   printf("n?");
   scanf("%d",&n);
   for(i=0; i<n; i++)
     write(fd[1], &i, sizeof(int));
   } else {
   close(fd[1]); /*ESSENCIAL!!!*/
   while ( read(fd[0], &i, sizeof(int)) >0 ) /* + dados? */
     printf("%d\n", i);
   }
}
```

Neste exemplo, o número de inteiros enviados através do *pipe* não é previamente conhecido. Deve, por isso, ser testado o valor devolvido pela função 'read()'. Enquanto o valor for superior a zero, novos dados poderão estar disponíveis no *pipe*. Quando é que o valor zero é devolvido? Qual a razão para ser obrigatória a chamada à função 'close(fd[1]);'? Elimine esta função e teste o programa. O que acontece? Qual a razão para o sucedido? Retire as devidas conclusões.

3. Capacidade de um pipe

O programa 'cpipe.c', apresentado em seguida, permite testar a capacidade do *pipe*. O processo vai escrevendo dados no *pipe* até que fica bloqueado à espera que outro processo leia os dados (o que neste exemplo não acontece).

Programa 'cpipe.c':

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#define exit_on_error(s,m) if (s < 0) { perror(m); exit(1); }
main()
{ int fd[2], i,n;
  char c = 'A';
  pipe(fd);
  /* testar a capacidade, escrevendo no pipe
  até que fica bloqueado*/
  i=1;
  while (1) {
   n=write(fd[1], &c, sizeof(char));
   exit_on_error(n,"Erro:");
   printf("%d bytes escrito no pipe.\n", i++);
   }
}
```

Execute o programa para determinar a capacidade de um *pipe*.

4. Comunicação bidireccional

Considere o programa 'pdobro.c':

```
#include <stdio.h>
#include <unistd.h>
main()
{int pedido[2], resposta[2], n;
 pipe(pedido);
 pipe(resposta);
 if (!fork()){ /* filho */
   n=-1;
   while(n!=0) {
     printf("N (0 para terminar)?");
     scanf("%d", &n);
     if (n!=0){
```

```
        printf("O dobro de %d é ", n);
        write(pedido[1], &n, sizeof(int));
        read(resposta[0], &n, sizeof(int));
        printf("%d\n", n);}
       }
  } else { /* pai */
  close (pedido[1]); /*Essencial!!*/
  while(read(pedido[0], &n, sizeof(int))>0)
     {n*=2;
     write(resposta[1], &n, sizeof(int));
     }
  printf("Já não haverá mais pedidos, vou terminar\n");
  }
}
```

Este exemplo procura demonstrar uma comunicação bidireccional entre dois processos, recorrendo a dois *pipes*. O processo pai fica bloqueado na função 'read()' até que exista informação no *pipe* 'pedido'. O processo filho interage com o utilizador, solicitando um número para o qual se pretende determinar o dobro. O valor digitado é enviado pelo processo filho ao processo pai através do *pipe* 'pedido'. O processo pai determina o dobro do valor e envia-o através do *pipe* 'resposta'.
A sincronização entre processos pai e filho é realizada de forma implícita através da função 'read()' dado a seu comportamento bloqueante. O processo filho termina quando é introduzido o valor 0. Ao terminar, são encerrados todos os descritores para ambos os *pipes*. Como já não existem processos (pai e filho) com descritores para escrita no *pipe* 'pedido', a função 'read()' devolve o valor 0, terminando, em consequência, o processo pai.
A funcionalidade deste programa pode ser implementada através de um único *pipe*, sendo, no entanto, necessário recorrer a um mecanismo de sincronização entre os processos pai e filho. Reescreva o programa que utilize apenas um *pipe* e que recorra aos sinais para a sincronização entre processos.

5. Redireccionamento do stdout/stdin

A *shell* bash disponibiliza o operador '|' que permite o redireccionamento do stdout de um comando para o stdin de um segundo comando (ex. ls -la | wc). O exemplo seguinte ('redirp.c') demonstra a utilização de um *pipe* unidireccional para implementar este redireccionamento.
'redirp.c':

```
#include <unistd.h>
#include <stdio.h>
/* Implementa o equivalente a 'ls -la | wc' */
main()
{ int p[2];
  pipe(p);
  if(!fork()) { /*filho (por exemplo)*/
    close(p[0]);/* fecha descritor de leitura do pipe*/
    close(1);   /* fecha stdout */
    dup(p[1]);  /* stdout <==> p[1] */
    close(p[1]);/* já não vai ser utilizado*/
    execlp("ls", "ls", "-la", NULL);
   } else {
    close(p[1]);/* fecha descritor de escrita do pipe*/
    close(0);   /* fecha stdin */
    dup(p[0]);  /* stdin <==> p[0] */
    close(p[0]);/* já não vai ser utilizado*/
    execlp("wc", "wc", NULL);
   }
}
```

Altere o programa de modo a utilizar a função 'dup2()'. Utilize, para o efeito, a informação da respectiva *manpage*.

Escreva um programa que produza o equivalente ao comando:

```
$ cat < dados.txt | wc -l > result.txt
```

6. Exercício de aplicação

Escreva um programa de acordo com as seguintes especificações:

- o programa é invocado através da linha de comando:
  $ psomar <fich1> <fich2> <fich3> ..... <fichN>
- cada <fichi> é um ficheiro binário que armazena uma sequência de valores inteiros;
- por cada <fichi> (i=[1,N]) é lançado um processo filho que vai processar o ficheiro <fichi>;
- cada processo filho deve:
  - indicar o seu índice (1..N), o seu PID e o ficheiro que irá processar;
  - abrir o respectivo ficheiro, somar os valores inteiros e escrever a soma num *pipe* unidireccional.
- o processo pai procede à leitura das somas colocadas no *pipe* e imprime o valor total.

Utilize o seguinte programa 'fint.c'para criar ficheiros binários de valores inteiros.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
main(int argc, char *argv[])
{
  int i,j,fd_w;
  if (argc<=2) {
    printf("Usage: %s <fich> <i1> <i2> ... <in>\n",argv[0]);
    exit(1);}
  if ( (fd_w = open(argv[1], O_WRONLY|O_TRUNC|O_CREAT, 0644)) == -1) {
    perror("Erro na abertura/criação do ficheiro de destino");
    exit(1);}
  for(i=2;i<argc;i++){
    j=atoi(argv[i]);
    write(fd_w, &j, sizeof(int));}
  close(fd_w);
}
```