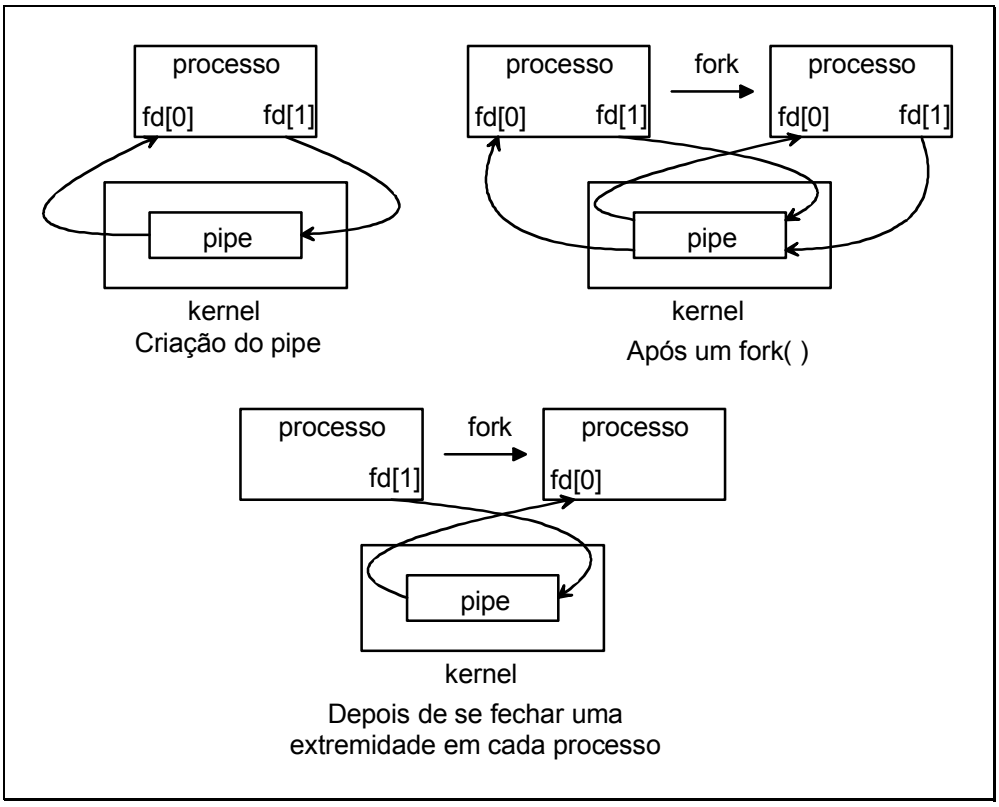
6. Comunicação entre processos - *Pipes*

6.1 O que são *pipes*

Os *pipes* em UNIX constituem um canal de comunicação unidirecional entre processos com um ascendente comum (entre um pai e um seu descendente). Uma vez estabelecido o *pipe* entre os processos, um deles pode enviar "mensagens" (qualquer sequência de bytes) para o outro. O envio e recebimento destas "mensagens" é feito com os serviços normais de leitura e escrita em ficheiros - `read()` e `write()`. Os *pipes* possuem descritores semelhantes aos dos ficheiros.

Quando se cria um *pipe* o sistema retorna, para o processo que chamou o serviço de criação, dois descritores que representam o lado de escrita no *pipe* e o lado de leitura no *pipe*. Inicialmente esses descritores pertencem ambos a um processo. Quando esse processo lança posteriormente um filho este herdará esses descritores (herda todos os ficheiros abertos) ficando assim pronto o canal de comunicação entre os dois processos. Consoante o sentido de comunicação pretendido deverá fechar-se, em cada processo, um dos lados do *pipe* (ver a figura seguinte).



6.2 Criação e funcionamento de *pipes*

Assim, para criar um *pipe* num processo deve invocar-se o serviço:

```
#include <unistd.h>

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

Retorna 0 no caso de sucesso e -1 no caso de erro.

Deve passar-se ao serviço `pipe()` um array de 2 inteiros (**`filedes`**) que será preenchido pelo serviço com os valores dos descritores que representam os 2 lados do *pipe*. O descritor contido em `filedes[0]` está aberto para leitura - é o lado receptor do *pipe*. O descritor contido em `filedes[1]` está aberto para escrita - é o lado emissor do *pipe*.

Quando um dos lados do `pipe` está fechado e se tenta uma operação de leitura ou escrita do outro lado, aplicam-se as seguintes regras:

1. Se se tentar ler de um *pipe* cujo lado emissor tenha sido fechado, após se ter lido tudo o que porventura já tenha sido escrito, o serviço `read()` retornará o valor `0`, indicador de fim de ficheiro.

2. Se se tentar escrever num *pipe* cujo lado receptor já tenha sido fechado, gera-se o sinal `SIGPIPE` e o serviço `write()` retorna um erro (se `SIGPIPE` não terminar o processo, que é a sua acção por defeito).

Exemplo - estabelecimento de um *pipe* entre pai e filho

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
      int   n, fd[2];
      pid_t pid;
      char  line[MAXLINE];

      if (pipe(fd) < 0) {
            fprintf(stderr, "pipe error\n");
            exit(1);
      }
      if ( (pid = fork()) < 0) {
            fprintf(stderr, "fork error\n");
            exit(2);
      }
      else if (pid > 0) {           /* pai */
            close(fd[0]);           /* fecha lado receptor do pipe */
            write(fd[1], "hello world\n", 12);
      } else {                      /* filho */
            close(fd[1]);           /* fecha lado emissor do pipe */
            n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
      }
      exit(0);
}
```

Outro exemplo - Usar um programa externo, p. ex. um pager (formatador de texto), para lhe enviar, através de um pipe redireccionado para a sua entrada standard, o texto a paginar.

```
#include    <unistd.h>
#include    <sys/wait.h>
#include    <stdio.h>

#define     PAGER "/usr/bin/more"                       /* programa pager */
```

```
int main(int argc, char *argv[])
{
      int   n, fd[2];
      pid_t pid;
      char  line[MAXLINE], *pager, *argv0;
      FILE  *fp;

      if (argc != 2) {
            printf("usage: prog filename\n");
            exit(0);
      }

      /* abertura do ficheiro de texto usando bibl. standard do C */

      if ( (fp = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "can't open %s\n", argv[1]);
            exit(1);
      }
      if (pipe(fd) < 0) {
            fprintf(stderr, "pipe error\n");
            exit(2);
      }
      if ( (pid = fork()) < 0) {
            fprintf(stderr, "fork error\n");
            exit(3);
      }
      else if (pid > 0) {                              /* pai */
            close(fd[0]);                        /* fecha o lado receptor */

            /* copia argv[1] para o pipe */

            while (fgets(line, MAXLINE, fp) != NULL) {
                  n = strlen(line);
                  if (write(fd[1], line, n) != n) {
                        fprintf(stderr, "write error to pipe\n");
                        exit(4);
                  }
            }
            if (ferror(fp)) {
                  fprintf(stderr, "fgets error");
                  exit(5);
            }
            close(fd[1]);                  /* fecha lado emissor do pipe */

            /* espera pelo fim do filho */

            if (waitpid(pid, NULL, 0) < 0) {
                  fprintf(stderr, "waitpid error\n");
                  exit(6);
            }
            exit(0);                                 /* termina normalmente */
      } else {                     /* filho */
            close(fd[1]);                             /* fecha lado emissor */
            if (fd[0] != STDIN_FILENO) {        /* programação defensiva */

                  /* redirecciona fd[0] para entrada standard */

                  if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO) {
                        fprintf(stderr, "dup2 error to stdin\n");
                        exit(7);
```

```
                }
                close(fd[0]);          /* não é preciso depois do dup2()*/
        }
        pager = PAGER;                  /* obtém argumentos para execl()*/
        if ( (argv0 = strrchr(pager, '/')) != NULL)
                argv0++;                            /* após último slash */
        else
                argv0 = pager;                /* não há slashes em pager */

        /* executa pager com a entrada redireccionada */

        if (execl(pager, argv0, NULL) < 0) {
                fprintf(stderr, "execl error for %s\n", pager);
                exit(8);
        }
    }
}
```

Constitui uma operação comum, quando se utilizam *pipes*, criar um *pipe* ligado à entrada ou saída standard de um outro processo que se lança na altura através de `fork()` - `exec()`. A fim de facilitar esse trabalho a biblioteca standard do C contém as seguintes funções.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Retorna apontar válido, ou `NULL` no caso de erro.

```
int pclose(FILE *fp);
```

Retorna código de terminação de cmdstring ou -1 no caso de erro.

O parâmetro **`cmdstring`** indica o processo a lançar (juntamente com os seus argumentos) ao qual se vai ligar o *pipe*; o parâmetro **`type`** pode ser `"r"` ou `"w"`; se for `"r"` o *pipe* transmite do novo processo para aquele que chamou `popen()` (ou seja quem chamou `popen()` pode ler (`read`) o *pipe*); se for `"w"` a transmissão faz-se em sentido contrário. A função `popen()` retorna um `FILE *` que representa o lado visível do pipe no processo que chama `popen()` como se fosse um ficheiro da biblioteca standard do C (leituras e escritas com `fread()` e `fwrite()` respectivamente).

Para fechar o *pipe* usa-se `pclose()` que também retorna o código de terminação do processo lançado por `popen()`.

Exemplo - O exemplo anterior escrito com `popen()` (sem verificação de erros)

```
#include    <stdio.h>

#define     PAGER       "/usr/bin/more"         /* programa pager */

int main(int argc, char *argv[])
{
      char  line[MAXLINE];
      FILE  *fpin, *fpout;

      fpin = fopen(argv[1], "r"))
```

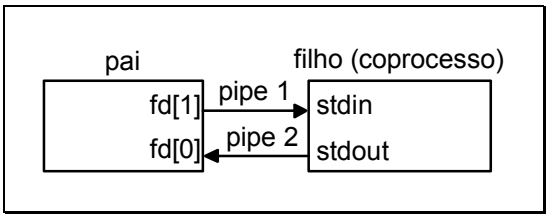
```
      fpout = popen(PAGER, "w"))
      while (fgets(line, MAXLINE, fpin) != NULL)
            fputs(line, fpout);
      pclose(fpout);
      return 0;
}
```

6.3 Coprocessos

Um filtro é um programa que lê informação da sua entrada standard, a modifica, e escreve essa informação modificada na sua saída standard.

Quando um outro programa envia informação para a entrada standard de um filtro e depois recolhe a resposta na sua saída standard, o filtro passa a chamar-se um coprocesso desse programa.

A forma usual de utilizar um coprocesso é lançá-lo como filho de um processo inicial e estabelecer ligações com as suas entrada e saída stantard através de dois *pipes*, como se mostra na figura seguinte.



O coprocesso nunca se apercebe da utilização dos pipes. Simplesmente lê a entrada standard e escreve a sua resposta na saída standard.
No exemplo seguinte temos o código de um coprocesso que lê dois inteiros da entrada standard e escreve a sua soma na saída standard.

```
#include <stdio.h>

int main(void)
{
      int   n, int1, int2;
      char  line[MAXLINE];

      while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
            line[n] = 0;                    /* terminar linha lida com 0 */
            if (sscanf(line, "%d %d", &int1, &int2) == 2) {
                  sprintf(line, "%d\n", int1 + int2);
                  n = strlen(line);
                  if (write(STDOUT_FILENO, line, n) != n) {
                        fprintf(stderr, "write error"\n);
                        return 1;
                  }
            } else {
                  if (write(STDOUT_FILENO, "invalid args\n", 13) != 13) {
                        fprintf(stderr, "write error\n");
                        return 1;
                  }
            }
      }
      return 0;
}
```

Um programa que usa o anterior como coprocesso pode ser:

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

static void sig_pipe(int);                          /* our signal handler */

int main(void)
{
      int   n, fd1[2], fd2[2];
      pid_t pid;
      char  line[MAXLINE];

      signal(SIGPIPE, sig_pipe);
      pipe(fd1);
      pipe(fd2);
      pid = fork();
      if (pid > 0) {                                              /* parent */
            close(fd1[0]);
            close(fd2[1]);
            while (fgets(line, MAXLINE, stdin) != NULL) {
                  n = strlen(line);
                  write(fd1[1], line, n);
                  n = read(fd2[0], line, MAXLINE);
                  if (n == 0) {
                        printf("child closed pipe\n");
                        break;
                  }
                  line[n] = 0;
                  fputs(line, stdout);
            }
            return 0;
      } else {                                                     /* child */
            close(fd1[1]);
            close(fd2[0]);
            if (fd1[0] != STDIN_FILENO) {
                  dup2(fd1[0], STDIN_FILENO);
                  close(fd1[0]);
            }
            if (fd2[1] != STDOUT_FILENO) {
                  dup2(fd2[1], STDOUT_FILENO);
                  close(fd2[1]);
            }
            execl("./add2", "add2", NULL);
      }
}

static void sig_pipe(int signo)
{
      printf("SIGPIPE caught\n");
      exit (1);
}
```

6.4 *Pipes* com nome ou FIFOs

Os FIFOs são por vezes chamados *pipes* com nome e podem ser utilizados para estabelecer canais de comunicação entre processos não relacionados ao contrário dos

pipes, que exigem sempre um ascendente comum entre os processos que ligam.

Quando se cria um FIFO o seu nome aparece no directório especificado no sistema de ficheiros.

A criação de FIFOs faz-se com o seguinte serviço:

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

int mkfifo(const char *pathname, mode_t mode);
```

Retorna 0 se houver sucesso e -1 no caso contrário.

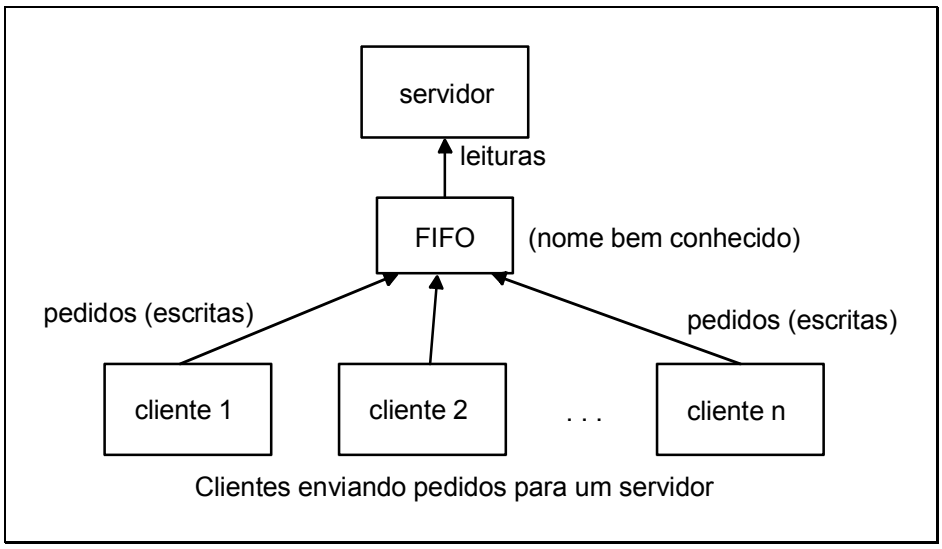
O parâmetro **pathname** indica o directório e o nome do FIFO a criar, enquanto que o argumento **mode** indica as permissões de acesso ao FIFO (ver serviço `open()` no capítulo 2).
Uma vez criado o FIFO é necessário abri-lo para leitura ou escrita, como se fosse um ficheiro, com o serviço `open()`. Um FIFO suporta múltiplos escritores, mas apenas um leitor.

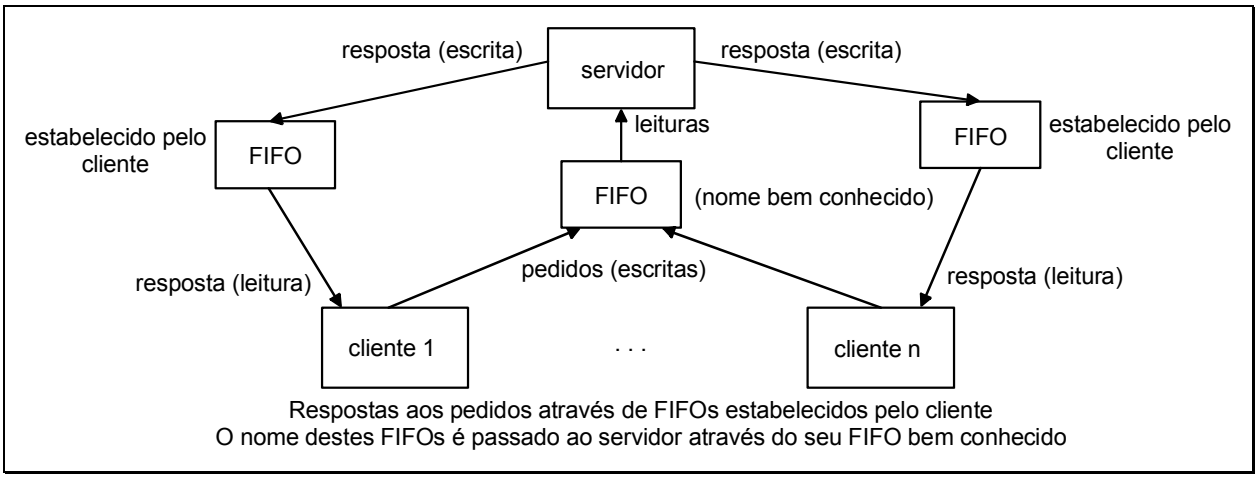
As leituras, escritas, fecho e eliminação de FIFOs fazem-se com os serviços correspondentes para ficheiros (`read()`, `write()`, `close()` e `unlink()`).

Quando se abre um FIFO a *flag* `O_NONBLOCK` de `open()` afecta a sua operação. Se a *flag* não for especificada a chamada a `open()` para leitura ou para escrita bloqueia até que um outro processo faça uma chamada a `open()` complementar (escrita ou leitura respectivamente). Quando a *flag* é especificada, uma chamada a `open()` para leitura retorna imediatamente com sucesso, enquanto que uma abertura para escrita retorna um erro se o FIFO não estiver já aberto para leitura por um outro processo.

O comportamento das leituras e escritas nos FIFOs (com `read()` e `write()`) é semelhante ao dos *pipes*.

As duas figuras seguintes ilustram algumas utilizações de FIFOs no funcionamento de sistemas de processos clientes-servidor.




resposta (escrita)
servidor
resposta (escrita)
estabelecido pelo cliente
FIFO
leituras
FIFO
estabelecido pelo cliente
FIFO
(nome bem conhecido)
resposta (leitura)
pedidos (escritas)
resposta (leitura)
cliente 1
. . .
cliente n
Respostas aos pedidos através de FIFOs estabelecidos pelo cliente
O nome destes FIFOs é passado ao servidor através do seu FIFO bem conhecido