Pipes e *FIFOS*

Objectivos

No final desta aula, os estudantes deverão ser capazes de:

- **Explicar a utilidade de *Pipes* e *FIFOS***
- **Explicar as diferenças entre uns e outros**
- **Utilizar *Pipes* e *FIFOS* para comunicação entre dois ou mais processos**
- **Identificar alguns dos problemas que podem surgir na utilização destes mecanismos de comunicação e tomar providências para evitá-los**

Comunicação entre Processos
Pipes e FIFOS

***Pipes* são:**

- **um mecanismo de comunicação que permite que dois ou mais processos a correr no mesmo computador enviem dados uns aos outros.**

Tipos de *pipes:*

- ***pipes* sem nome (*unnamed pipes* ou apenas *pipes*)**
  - » **São *half-duplex* ou unidireccionais. Os dados só podem fluir num sentido.**
  - » **Só podem ser usados entre processos que tenham um antecessor comum.**
- ***pipes* com nome (*named pipes* ou *FIFOS*)**
  - » **São *half-duplex* ou unidireccionais.**
  - » **Podem ser usados por processos não relacionados entre si.**
  - » **Têm um nome que os identifica, existente no sistema de ficheiros.**

Pipes

- **Um *pipe* pode ser visto como um canal ligando 2 processos, permitindo um fluxo de informação unidireccional.**
- **Esse canal tem uma certa capacidade de *bufferização* especificada pela constante `PIPE_BUF` (ou outra com nome semelhante, em `<limits.h>`).**
- **Cada extremidade de um *pipe* tem associado um descritor de ficheiro.**
- **Um *pipe* é criado usando a chamada de sistema *pipe( )* a qual devolve dois descritores, um representando a extremidade de escrita e outro a de leitura.**
- **Para o programador, os *pipes* têm uma interface idêntica à dos ficheiros. Um processo escreve numa extremidade do *pipe* como para um ficheiro e o outro processo lê na outra extremidade.**
- **Um *pipe* pode ser utilizado como um ficheiro *ou* em substituição do periférico de entrada ou de saída de um programa.**

Pipes

- **Protótipo da função *pipe*:**

```
# include <unistd.h>

int pipe (int filedes[2]);

Retorna: 0 se OK, -1 se houve erro
```

- **A função retorna 2 descritores de ficheiros:**
  - **`filedes[0]` – está aberto para leitura**
  - **`filedes[1]` – está aberto para escrita**
- **As primitivas de leitura e escrita são *read* e *write*:**

```
# include <unistd.h>
ssize_t read (int fd, char * buf, int count);
ssize_t write (int fd, char * buf, int count);

Retornam: nºde bytes lidos/escritos, 0 se EOF (só read),
          -1 se houve erro
```


fdopen()

```
# include <stdio.h>

FILE * fdopen(int fildes, const char *mode)

Retorna: FILE * se bem sucedida
         ou NULL se houve erro (e errno é actualizada)
```

- **`fdopen()`**
  associa uma *stream*
  a um descritor de ficheiro, `filedes`, já existente
- **Desta forma é possível aceder a um *pipe***
  usando as funções de leitura/escrita da biblioteca standard de C:
  - **fscanf(), fprintf(), fread(), fwrite(), …**
- **O modo da *stream***
  (um dos valores `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, `"a+"`)
  deve ser compatível com o modo do descritor do ficheiro

Pipes

Representação simplificada
de um *pipe* num único processo

Processo do utilizador

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

fd[0] fd[1]

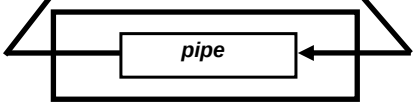
- Um *pipe* envolvendo um único processo é praticamente inútil.

Convém não esquecer que, de facto, os dados circulam através do *kernel*

Processo do utilizador

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

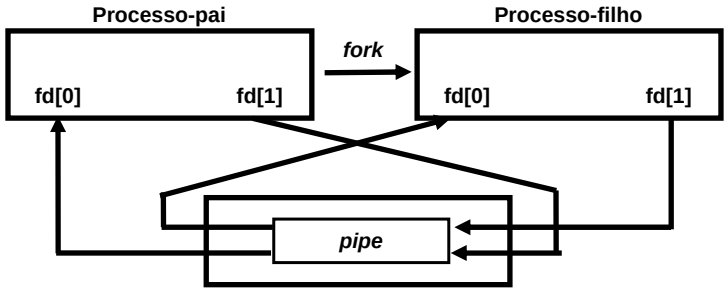
fd[0] fd[1]



Kernel

Pipes

- Normalmente, o processo que cria o *pipe*, invoca *fork* a seguir, criando assim um canal de comunicação entre pai e filho ou vice-versa.

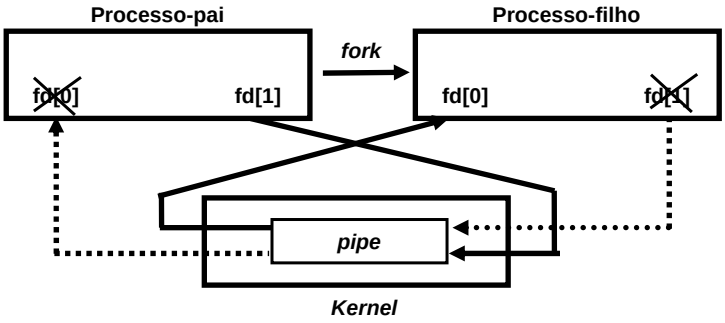


- Só o processo que cria o *pipe* e os seus descendentes podem usar o *pipe*.

Pipes

- O que se faz depois da chamada *fork*
  depende do sentido em que se pretende o fluxo de dados.

- Exemplo: fluxo no sentido do pai p/ o filho
  - o pai fecha a extremidade de leitura - fd[0]
  - o filho fecha a extremidade de escrita - fd[1]



Pipes

- **Sequência típica de operações para a comunicação unidireccional entre o processo-pai e o processo-filho:**
  - **O processo-pai cria o *pipe*, usando a chamada *pipe( )*.**
  - **O processo-pai invoca *fork( )*.**
  - **O processo-escritor fecha a sua extremidade de leitura do *pipe* e o processo-leitor fecha a sua extremidade de escrita do *pipe*.**
  - **Os processos comunicam usando chamadas *write( )* e *read( )*.**
    - » ***write*** **acrescenta dados numa extremidade do *pipe* (extremidade de escrita)**
    - » ***read*** **lê dados da outra extremidade do *pipe* (extremidade de leitura)**
  - **Cada processo fecha o seu descritor activo do *pipe* quando tiver terminado a sua utilização.**
- **A comunicação bidireccional é possível usando 2 *pipes*.**

Exemplo

Envio de dados do pai p/ o filho usando um *pipe:*

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

#define MAXLINE  4096
#define READ     0
#define WRITE    1

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

        if (pipe(fd) < 0) {fprintf(stderr,"Pipe error"); exit(1);}

        if ( (pid = fork()) < 0) {fprintf(stderr,"Fork error"); exit(2);}

        else if (pid > 0) {       /* parent, writer */
                close(fd[READ]);
                write(fd[WRITE], "Hello world\n", 12);

             } else {             /* child, reader */
                close(fd[WRITE]);
                n = read(fd[READ], line, MAXLINE);
                write(STDOUT_FILENO, line, n);  // <--Why not printf ...?
        }
        exit(0);
}
```


Pipes

Regras aplicáveis aos processos-leitores:

- **Se um processo executar *read* de um *pipe* cuja extremidade de escrita foi fechada, depois de todos os dados terem sido lidos, *read* retorna 0, indicando fim de ficheiro.**
  - » **NOTA:**
    - **Frequentemente existe um único leitor e um único escritor de/para um *pipe* .**
    - **No entanto, é possível ter, por exemplo, vários escritores e um único leitor.**
    - **Neste último caso, o fim de ficheiro só é retornado quando todos os escritores tiverem fechado o terminal de escrita do *pipe*.**
- **Se um processo executar *read* de um *pipe* vazio cuja extremidade de escrita ainda estiver aberta fica bloqueado até haver dados disponíveis. (*)**
- **Se um processo tentar ler mais *bytes* do que os disponíveis são lidos os *bytes* disponíveis e a chamada *read* retorna o número de *bytes* lidos.**

(*) - ver notas finais acerca da activação da *flag* O_NONBLOCK)

Pipes

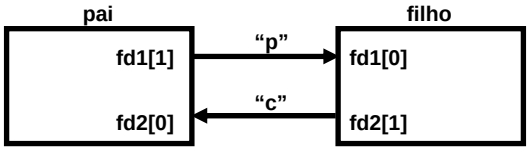
Regras aplicáveis aos processos-escritores:

- **Se um processo executar *write* para um *pipe* cuja extremidade de leitura foi fechada a escrita falha e ao escritor é enviado o sinal `SIGPIPE`. A acção por omissão deste sinal é terminar o receptor do sinal.**
- **Se um processo escrever `PIPE_BUF` *bytes* ou menos é garantido que a escrita é feita atomicamente, isto é, não é interlaçada com escritas de outros processos que escrevam para o mesmo *pipe*.**
- **Se um processo escrever mais do que `PIPE_BUF` *bytes* não são dadas garantias de atomicidade da escrita, isto é, os dados dos diversos escritores podem surgir interlaçados.**

Utilização dos *pipes*

1. **Enviar dados de um processo p/ outro (exemplo anterior)**
2. **Sincronização entre processos**

   ⇒ **usar 2 *pipes***



3. **Ligar a *standard output* de um processo à *standard input* de outro**

   ⇒ **"duplicar" os descritores de um *pipe* para a *standard input* de um dos processos e para a *standard output* do outro**

Duplicação de um descritor

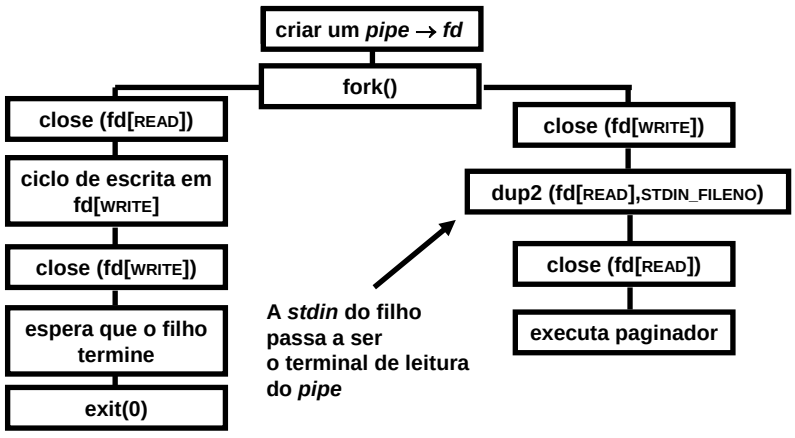
Pode ser feita c/ as funções *dup* ou *dup2*.

```
# include <unistd.h>

int dup (int filedes);
int dup2 (int filedes, int filedes2);

Retornam: novo descritor se OK, -1 se houve erro
```

- **dup**
  - procura o descritor livre c/ o número mais baixo e põe-no a apontar p/ o mesmo ficheiro que `filedes` .
- **dup2**
  - fecha `filedes2` se ele estiver actualmente aberto e põe `filedes2` a apontar p/ o mesmo ficheiro que `filedes`;
  - se `filedes=fildes2`, retorna `filedes2` sem fechá-lo .
  - exemplo:
    `dup2(fd,STDIN_FILENO)`
    redirecciona a entrada standard (teclado) para o ficheiro cujo descritor é `fd`.

Exemplo (utilização 3)

Programa que mostra a sua saída, uma página de cada vez, usando o paginador do UNIX (programa 14.2-Stevens)



As funções *popen* e *pclose*

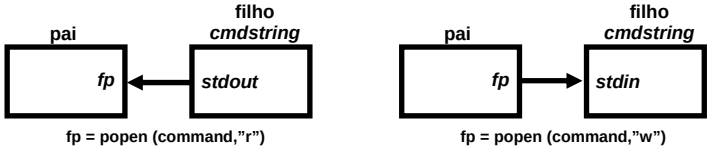
- ***popen***
  - **Cria um *pipe* entre o processo que a invocou e um programa a executar; este programa tanto pode receber como fornecer dados ao processo.**
  - **Faz parte do trabalho do exemplo anterior:**
    - criar um *pipe;*
    - executar *fork;*
    - executar um programa
      invocando uma *subshell* (sh) à qual o programa é passado como comando a executar.
      Vantagem: a *subshell* faz a expansão dos argumentos (por exemplo `*.c`)
      o que permite executar com popen comandos que seria mais complicado executar com exec.
      Desvantagem: é criado um processo adicional (a *subshell*) para executar o programa.
  - **Retorna um apontador para um ficheiro (`FILE *`) que será o ficheiro de entrada ou de saída do programa, consoante um parâmetro de *popen*.**
- ***pclose***
  - **Fecha o ficheiro.**
  - **Espera que o programa termine (mais concretamente, a *shell*).**
  - **Retorna o *termination status* da *subshell* usada para executar o programa.**

As funções *popen* e *pclose*

```
#include <stdio.h>                    /* FUNÇÕES DA BIBLIOTECA DE C */

FILE *popen(const char *cmdstring, const char *type);
Retorna: file pointer se OK; NULL se houve erro

int pclose(FILE *fp);
Retorna: termination status de cmdstring se OK; -1 se houve erro
```

cmdstring
- **programa a executar**

type
- ***"r"* – o *file pointer* retornado está ligado à *standard output* de *cmdstring***
- ***"w"* – o *file pointer* retornado está ligado à *standard input* de *cmdstring***



fp = popen (command,"r") **fp = popen (command,"w")**

Exemplo

Programa que mostra um ficheiro, página a página, usando o paginador do UNIX.

```
#include  <stdio.h>
#include  <stdlib.h>

#define   MAXLINE   1000
#define   PAGER     "/bin/more"

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

 if (argc != 2) { printf("usage: %s filename\n“, argv[0]); exit(1); }
 if ((fpin = fopen(argv[1], "r")) == NULL) {fprintf(stderr,"can't open %s", argv[1]); exit(1);}
 if ((fpout = popen(PAGER, "w")) == NULL) {fprintf(stderr,"popen error"); exit(1);}
 /* copy filename contents to pager - file=argv[1] */
 while (fgets(line, MAXLINE, fpin) != NULL)
  { if (fputs(line, fpout) == EOF) { printf("fputs error to pipe"); exit(1); }
  }
 if (ferror(fpin)) { fprintf(stderr,"fgets error"); exit(1); }
 if (pclose(fpout) == -1) { fprintf(stderr,"pclose error"); exit(1); }
 exit(0);
}
```


Filtros

Filtro

- **um programa que lê da *standard input* e escreve p*/* a *standard output* normalmente ligado a outros processos, constituindo um *pipeline***

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h> /*char handling */

int main(void)
{
 int c;

 while ((c=getchar()) != EOF)
  {if (isupper(c)) c=tolower(c);
   if (putchar(c)==EOF)
    {printf("output error"); exit(1);}
   if (c=='\n') fflush(stdout);
  }
 exit(0);
}
```

Filtro que converte maiúsculas em minúsculas (executável → up_to_low)

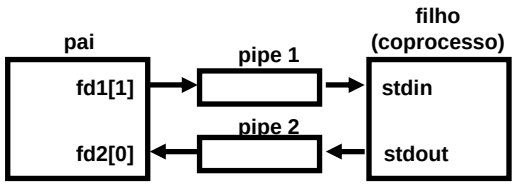
```
#include ...

#define MAXLINE 1000

int main(void)
{
 char line[MAXLINE];
 FILE *fpin;

 if ((fpin=popen(“./up_to_low","r")) == NULL)
  {printf("popen error"); exit(1);}
 for ( ; ; )
  { fputs("prompt > ",stdout); fflush(stdout);
    if (fgets(line,MAXLINE,fpin) == NULL) break;
    if (fputs(line,stdout) == EOF)
     { fprintf(stderr,"fputs error"); exit(1);}
  }
 if (pclose(fpin)==-1)
  {fprintf(stderr,"pclose error"); exit(1);}
 putchar('\n');
 exit(0);
}
```

Programa que usa o filtro (terminar com CTRL-D ≡ *end of input*)

Coprocessos

Coprocesso

- **é um filtro especial**
  cuja *standard input* e *standard output* estão ligadas a um outro processo, através de *pipes*



Exemplo (coprocesso)

```
// Programa "somador"
#include <stdio.h>
#include <unistd.h>

#define   MAXLINE   100

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

 while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0)
  { line[n] = 0;             /* null terminate */
    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"); exit(1);}
     }
    else
     { if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
        { fprintf(stderr,"write error"); exit(1);}
     }
  }
 exit(0);
}
```

Coprocesso

- **Lê 2 números da *standard input***
- **calcula a sua soma**
- **e escreve o resultado na *standard output***

O executável deverá chamar-se *somador*

Exemplo (coprocesso - cont.)

Programa que invoca o coprocesso

```
#include ...

#define MAXLINE 1000
#define READ    0
#define WRITE   1

void sig_pipe(int signo);
void err_sys(char *msg);
void err_msg(char *msg);

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

 if (signal(SIGPIPE, sig_pipe)
  err_sys("signal error");
 if (pipe(fd1)<0 || pipe(fd2)<0)
  err_sys("pipe error");
 if ((pid=fork())<0) err_sys("fork error");
 else
```

```
 if (pid>0) /* PARENT */
   {
    close(fd1[READ]); close(fd2[WRITE]);
    while (fgets(line, MAXLINE,stdin) != NULL)
     {
      n=strlen(line);
      if (write(fd1[WRITE],line,n) != n)
       err_sys("write error to pipe");
      if ((n=read(fd2[READ],line,MAXLINE)) < 0)
       err_sys("read error from pipe");
      if (n==0) {err_msg("child closed pipe"); break;}
      line[n]=0;
      if (fputs(line,stdout)==EOF) err_sys("fputs error");
     }
    if (ferror(stdin)) err_sys("fgets error on stdin");
    exit(0);
   }
```


Exemplo (coprocesso - cont.)

Programa que invoca o coprocesso (cont.)

```
 else       /* CHILD */
   {
    close(fd1[WRITE]); close(fd2[READ]);
    if (fd1[READ] != STDIN_FILENO)
     {
      if (dup2(fd1[READ],STDIN_FILENO) != STDIN_FILENO)
       err_sys("dup2 error to stdin");
      close(fd1[READ]);
     }
    if (fd2[WRITE] != STDOUT_FILENO)
     {
      if (dup2(fd2[WRITE],STDOUT_FILENO) != STDOUT_FILENO)
       err_sys("dup2 error to stdout");
      close(fd2[WRITE]);
     }
    if (execlp("somador","somador",(char *) 0) < 0)
     err_sys("execlp error");
   }
}
```

Ver justificação para este teste na pág. 433 do livro de W. Stevens (Advanced Programming in the UNIX Environment)

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

void err_sys(char *msg)
 {
  fprintf(stderr,"%s\n",msg);
  exit(1);
 }

void err_msg(char *msg)
 {
  printf("%s\n",msg); return;
 }
```

FIFOS / Named Pipes

- ***pipes* (*unnamed*)**
  - **troca de dados entre processos c/um antecessor comum.**
- ***FIFOS (named)***
  - **troca de dados entre processos não relacionados entre si a correr no mesmo *host* (ver adiante) .**

- **Um *FIFO* é um tipo de ficheiro. Tem um nome que existe no sistema de ficheiros.**
- **Podemos testar se um ficheiro é um *FIFO* c/ a macro `S_ISFIFO`.**
- **Um *FIFO* pode ser criado usando**
  - ***mkfifo* (invoca *mknod*) (função e utilitário)**
- **Um *FIFO* tem existência até ser explicitamente destruído.**
  - ***unlink* (função)**
  - ***rm* (utilitário)**

FIFOS

Função `mkfifo`

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

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

Retorna: 0 se OK, -1 se houve erro
```

pathname
- **nome do *FIFO* a criar**

mode
- **permissões de acesso (read, write, execute) p/ *owner*, *group* e *other***

	owner	group	other
	rwx	rwx	rwx
	111	101	000
mode →	7	5	0

Nota:
a permissão de acesso final é afectada pelo valor da *file creation mask* (default = 022 permissão de escrita só para o *owner*)

(v. função *umask*)

FIFOS

Utilização de um *FIFO*:

- **criar usando *mkfifo* ou *mknod***
- **abrir usando *open* ou *fopen***
  - » **include files - sys/types.h, sys/stat.h, fcntl.h**
  - » `int open (const char *filename, int mode [, int permissions]);`
    - **mode - OR *bit* a *bit* de (O_RDONLY ou O_WRONLY) e O_NONBLOCK**
      (um *FIFO* é *half-duplex*, não deve ser aberto em modo *read-write* (O_RDWR))
- **escrever / ler usando *write* / *read***
  - » **include files - unistd.h**
  - » `ssize_t read (int fd, char * buf, int count);`
  - » `ssize_t write (int fd, char * buf, int count);`
- **fechar usando *close***
  - » **include files - unistd.h**
  - » `int close (int fd);`
- **destruir usando *unlink***
  - » **include files - unistd.h**
  - » `int unlink (const char *pathname);`

FIFOS

Regras aplicáveis aos processos que usam *FIFOS:*

Abertura

- **Se um processo tentar abrir um *FIFO* em modo *read only***
  e nenhum processo tiver o *FIFO* actualmente aberto p/ escrita
  o leitor esperará que um processo abra o *FIFO* p/ escrita
  a menos que a *flag* O_NONBLOCK esteja activada
  (a activação pode ser feita ao fazer *open* ou com a função *fcntl*),
  caso em que *open* retornará imediatamente.
- **Se um processo tentar abrir um *FIFO* em modo *write only***
  e nenhum processo tiver o *FIFO* actualmente aberto p/ leitura
  o escritor esperará que um processo abra o *FIFO* p/ leitura
  a menos que a *flag* O_NONBLOCK esteja activada,
  caso em que *open* falha imediatamente (retorna -1).

FIFOS

Regras aplicáveis aos processos que usam *FIFOS:*

Leitura / Escrita

- **Escrita p/ um *FIFO* que nenhum processo tem aberto p/ leitura ⇒ o sinal `SIGPIPE` é enviado ao processo-escritor.**
  - » **Se este sinal não for tratado conduz à terminação do processo.**
  - » **Se o sinal for ignorado ou se for tratado e o *handler* retornar, então `write` retorna o erro `EPIPE`**
- **Após o último escritor ter fechado um *FIFO*, um EOF é gerado em resposta às leituras seguintes, após o *FIFO* ficar vazio.**
- **Se houver vários processos-escritores, só há garantia de escritas atómicas quando se escreve no máximo `PIPE_BUF` *bytes.***

Exemplo

```
/* PROGRAMA reader */

#include ...

int readline(int fd, char *str);

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

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

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

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

```
/* PROGRAMA writer */

#include ...


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

 do
  {
   fd=open("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);
}
```

Exemplo (cont.)

Exemplo de execução:

```
/usr/users1/silva> reader & writer & writer &   <-- lança 1 reader e 2 writers
[1] 29161                                       <-- processo reader
[2] 29162                                       <-- 1º writer
[3] 29163                                       <-- 2º writer
/usr/users1/silva> Hello no. 1 from process 29162
Hello no. 1 from process 29163
Hello no. 2 from process 29162
Hello no. 2 from process 29163
Hello no. 3 from process 29163
Hello no. 3 from process 29162

[2]    Done                 writer

[3]    Done                 writer

[1]    Done                 reader

/usr/users1/silva> ls -la myfifo            <-- o fifo não foi destruído
prw-r-----  1 silva          0 Nov 19 14:44 myfifo   pelos programas
/usr/users1/silva> rm myfifo                <-- destroi o fifo
```


FIFOS

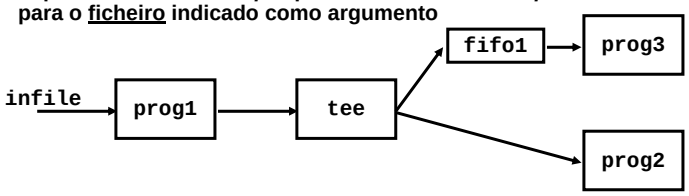
Utilitário *mkfifo*

Exemplo: utilização de *FIFO*s p/ duplicar *output streams* numa sequência de comandos da *shell*

```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
```

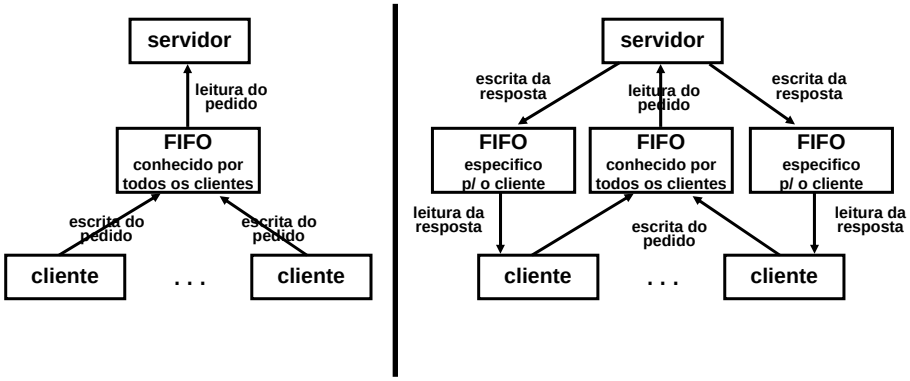
tee

- **copia a sua *standard input* para a sua *standard output* e para o ficheiro indicado como argumento**



FIFOS

Exemplo: utilização de *FIFO*s p/ comunicação cliente-servidor



Propriedades adicionais de *Pipes* e *FIFOS*

- `mkfifo()` tem implícito o modo `O_CREAT|O_EXCL`, isto é,
  cria um novo *FIFO*
  ou retorna o erro `EEXIST` se já existir um *FIFO* com o nome especificado
- Ao abrir um *FIFO*, pode-se activar a *flag* `O_NONBLOCK` :

```
fd=open(FIFO1,O_WRONLY|O_NONBLOCK);
```

- Esta *flag* influencia o comportamento de `open()`, `read()` e `write()`.
- Se um descritor já estiver aberto
  pode usar-se `fcntl()` para activar a *`flag`* `O_NONBLOCK`.
  - » Com *pipes*
    esta é a única possibilidade
    de activar esta *flag*,
    dado que não se usa `open()`.

```
...
int flags;
...
flags = fcntl(fd,F_GETFL,0);
flags = flags | O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
...
```

```
open(FIFO1,O_RDONLY);
```

- **open() bloqueia até que um processo abra o *FIFO* para escrita.**

```
open(FIFO1,O_RDONLY|O_NONBLOCK);
```

- **open() é bem sucedida e retorna imediatamente mesmo que o *FIFO* ainda não tenha sido aberto para escrita por nenhum processo.**

```
open(FIFO1,O_WRONLY);
```

- **open() bloqueia até que um processo abra o *FIFO* para leitura.**

```
open(FIFO1,O_WRONLY|O_NONBLOCK);
```

- **open() retorna imediatamente;**
  se algum processo tiver o *FIFO* aberto para leitura, retorna um descritor do *FIFO*
  se não retorna -1 (erro ENXIO) e o *FIFO* não será aberto.

- **`read()` de mais dados do que os disponíveis no *Pipe/FIFO***
  - » **retorna os dados disponíveis**
- **`read()` de um *Pipe/FIFO* vazio, não aberto para escrita**
  - » **retorna `0` (*end of file*), independentemente de `O_NONBLOCK`**
- **`read()` de um *Pipe/FIFO* vazio, já aberto para escrita**
  - » **se `O_NONBLOCK` não estiver activado**
    - **bloqueia até que sejam escritos dados no *Pipe/FIFO***
      ou até que o *Pipe/FIFO* deixe de estar aberto para escrita
  - » **se `O_NONBLOCK` estiver activado**
    - **retorna um erro, `EAGAIN`**
- **`write()` num *Pipe/FIFO*, não aberto para leitura**
  - » **`SIGPIPE` é enviado ao escritor, independentemente de `O_NONBLOCK`**
- **`write()` num *Pipe/FIFO*, já aberto para leitura** (→ a seguir)

- `write()` num *Pipe/FIFO*, já aberto para leitura
  - » Se `O_NONBLOCK` não estiver activado
    - Se nº de *bytes* a escrever <= `PIPE_BUF`
      - Se há espaço no *Pipe/FIFO* para o nº de *bytes* pretendido, todos os *bytes* são escritos.
      - Se não bloqueia até haver espaço no *Pipe/FIFO* para escrever os dados.
    - Se nº de *bytes* a escrever > `PIPE_BUF`
      - escreve parte dos dados, retornando o nº de *bytes* efectivamente escritos (pode ser zero).
  - » Se `O_NONBLOCK` estiver activado
    o valor de retorno de `write()` depende do nº de *bytes* a escrever e do espaço disponível nesse momento, no *Pipe/FIFO*:
    - Se nº de *bytes* a escrever <= `PIPE_BUF`
      - Se há espaço no *Pipe/FIFO* para o nº de *bytes* pretendido, todos os *bytes* são escritos.
      - Se não há espaço, `write()` retorna imediatamente com o erro `EAGAIN`.
    - Se nº de *bytes* a escrever > `PIPE_BUF`
      - Se houver espaço no *Pipe/FIFO* para pelo menos 1 *byte* o *kernel* transfere para lá o nº de *bytes* que lá couberem e `write()` retorna o nº de *bytes* escritos.
      - Se o *Pipe/FIFO* estiver cheio, `write()` retorna imediatamente com o erro `EAGAIN`.

FIFOS e NFS

- Os *FIFOS são* um mecanismo de *IPC* que pode ser usado num único *host*.
- Apesar de terem nomes no sistema de ficheiros, só podem ser usados em sistemas de ficheiros locais, e não em sistemas de ficheiros montados através de *NFS*.
- Alguns sistemas, permitem criar *FIFOS* num sistema de ficheiros montado em *NFS,* no entanto, não permitem a passagem de dados entre 2 sistemas, através desses *FIFOS*.

  Neste caso o *FIFO* só terá utilidade como mecanismo de *rendez-vous* entre 2 processos. Um processo num *host* não pode enviar dados a outro processo, noutro *host*, através do *FIFO*, apesar de ambos poderem abrir o *FIFO*, que está acessível a ambos através de *NFS*.