



Laboratório de Pesquisa em Redes e Multimídia

# Sistemas Operacionais

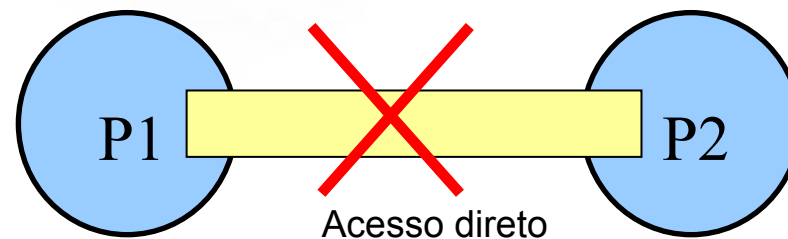
Inter Process Communication: Pipes



Universidade Federal do Espírito Santo  
Departamento de Informática

# Comunicação entre Processos (1)

- Processos executam no S.O. em cápsulas autônomas.
  - Um processo não acessa o espaço de endereçamento do outro.
  - A execução de um processo não afeta os outros.
  - Hardware oferece proteção de memória.
- Processos, entretanto, interagem e cooperam na execução de tarefas.
  - Em muitos casos, eles precisam trocar informação de forma controlada.
- S.O. fornece mecanismos que permitem aos processos comunicarem-se uns com os outros. Coletivamente estes mecanismos são conhecidos como **IPC - Inter-Process Communication**



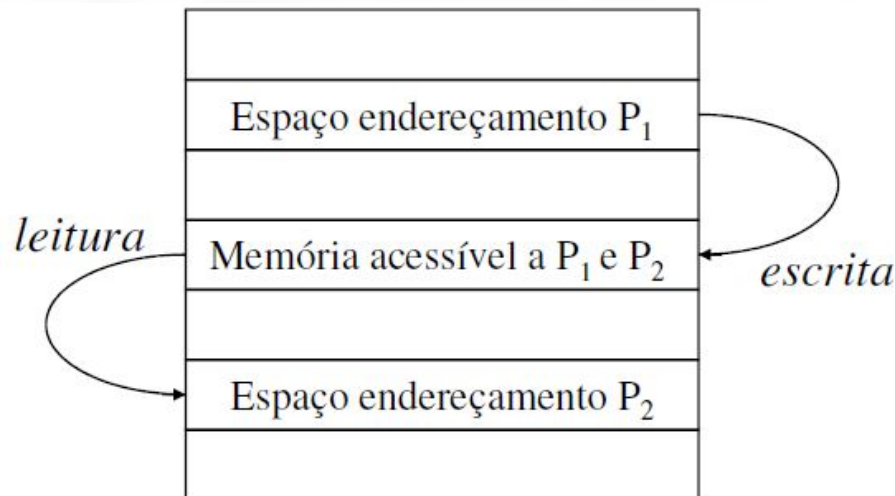
# Mecanismos de IPC

- Fundamentalmente, existem duas abordagens:
  - Suportar alguma forma de **espaço de endereçamento compartilhado**
    - **Shared memory** (memória compartilhada)

ou

- Utilizar **comunicação via núcleo do S.O.**, que ficaria então responsável por transportar os dados de um processo a outro. São exemplos:
  - **Pipes e Sinais** (ambiente centralizado)
  - **Troca de Mensagens** (ambiente distribuído)
  - **RPC – Remote Procedure Call** (ambiente distribuído)

# Comunicação via Memória Compartilhada



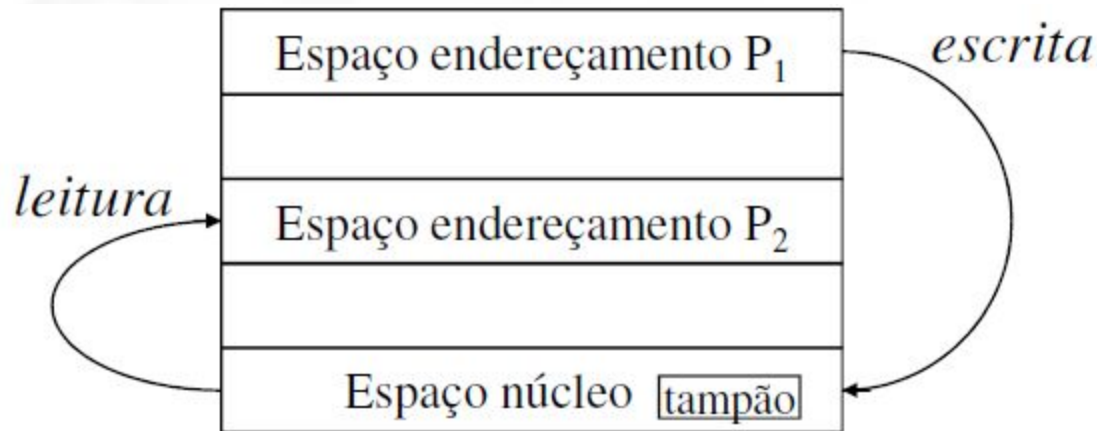
## ■ Vantagens:

- Mais eficiente (rápida), já que não exige a cópia de dados para alguma estrutura do núcleo.

## ■ Inconveniente:

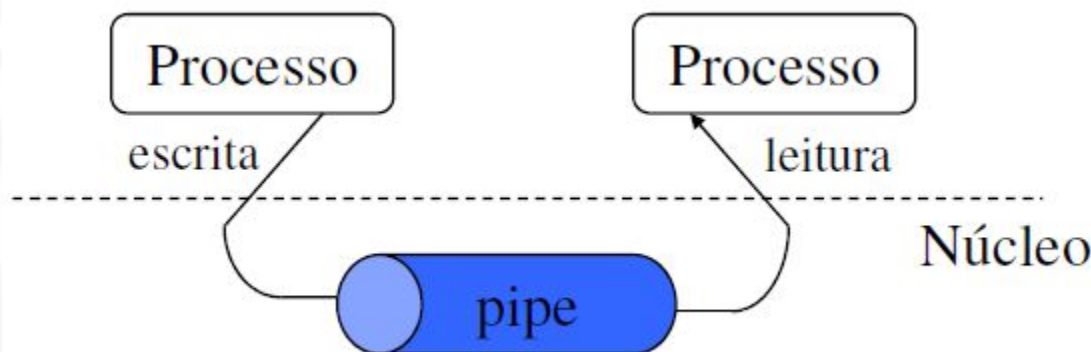
- Problemas de sincronização.

# Comunicação via Núcleo



- Vantagens:
  - Sincronização implícita.
- Inconveniente:
  - Mais complexa e demorada (uso de recursos adicionais do núcleo).

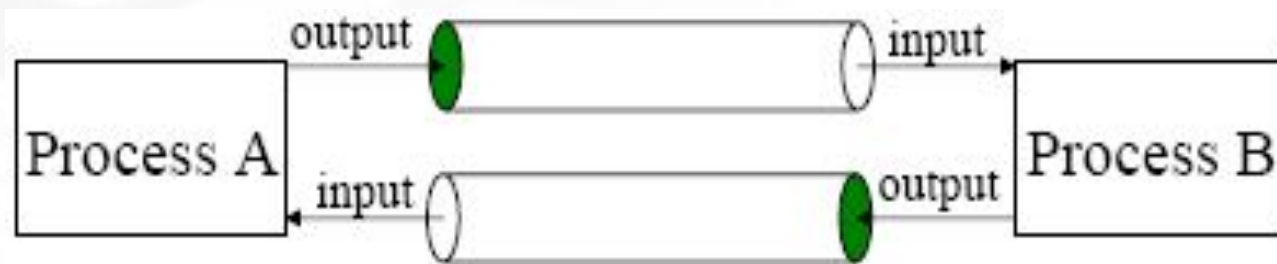
## Tubos (Pipes) (1)



- *Pipes* são o mecanismo original de comunicação unidirecional entre processos no Unix.
- É uma estrutura que possui duas extremidades, que são implementadas como arquivos. Como tal, o processo pode escrever e ler destas extremidades.
- Um pipe forma, na verdade, uma fila de caracteres do tipo FIFO.

## Tubos (Pipes) (3)

- Para estabelecer um diálogo bidirecional entre dois processos usando *pipes* é necessária a abertura de dois *pipes*, um para cada sentido da comunicação.



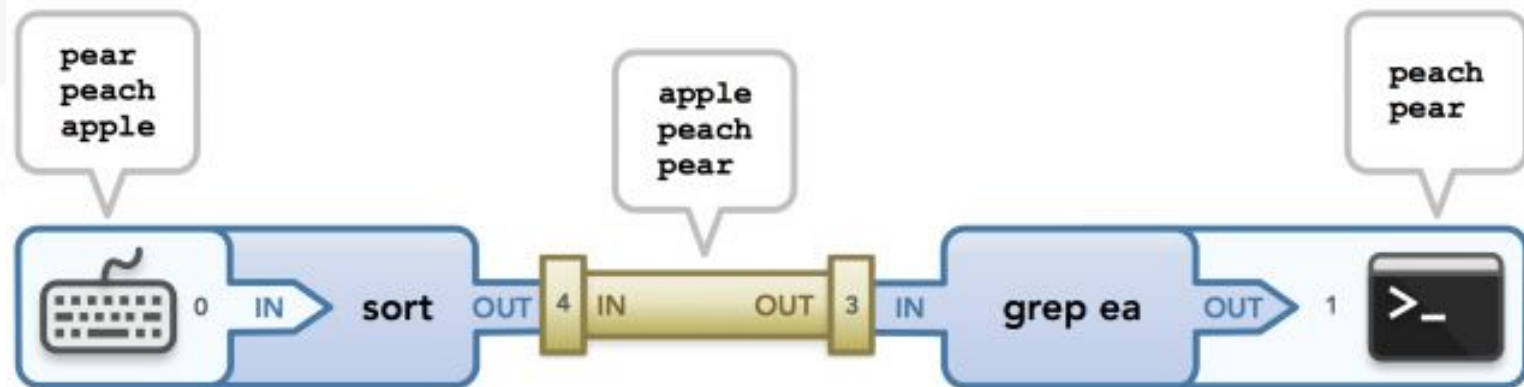
## Tubos (Pipes) (2)

- Um *pipe* é caracteriza-se por ser:
  - **Anônimo** (não tem nome).
  - **Temporário**: dura somente o tempo de execução do processo que o criou (não permanece no sistema de arquivos).
- A capacidade do *pipe* é limitada
  - Se uma escrita é feita e existe espaço no pipe, o dado é colocado no *pipe* e a chamada retorna imediatamente.
  - Se a escrita sobre um *pipe* continua mesmo depois dele estar cheio, ocorre uma situação de bloqueio (que permanece até que algum outro processo leia e, conseqüentemente, abra espaço no *pipe*).
- Não existe nenhum mecanismo que permita diferenciar as informações na saída do *pipe*.
- É **impossível** fazer qualquer movimentação no interior de um pipe.



## Uso de Pipes (1)

- `$ sort | grep ea`

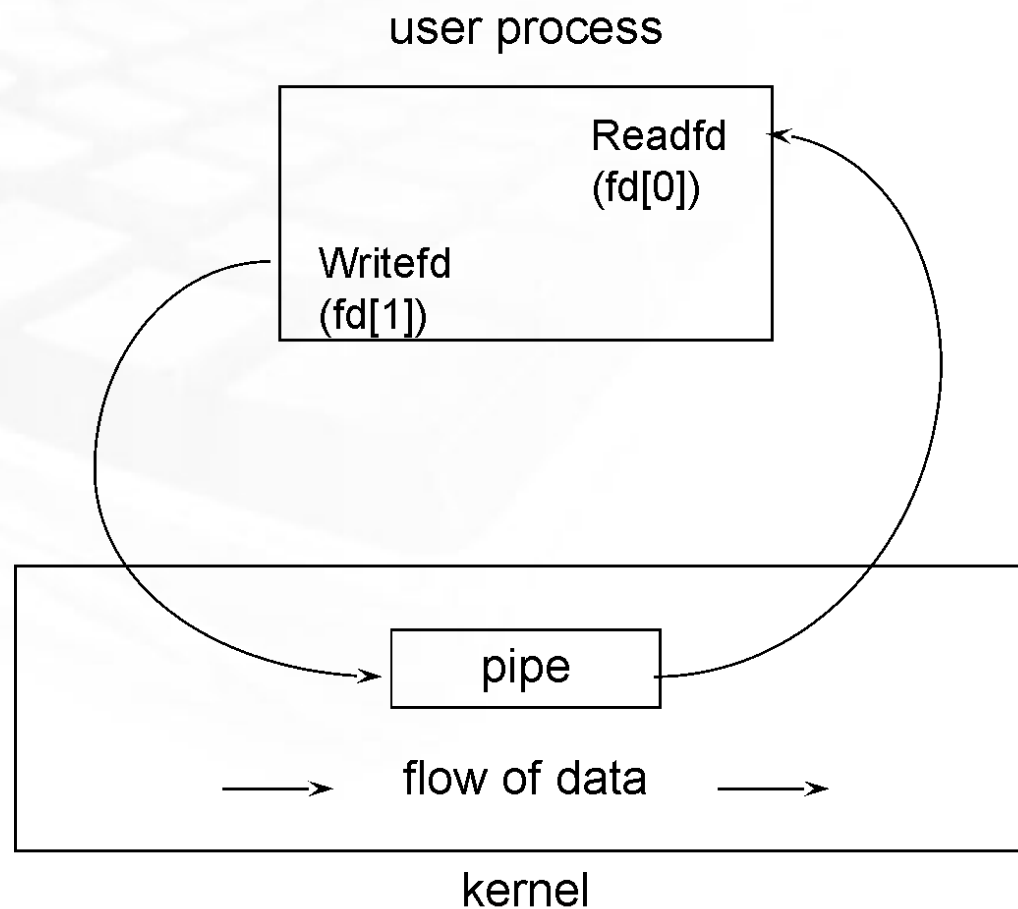


*Obs: use Ctrl-d to stop the “sort” input*

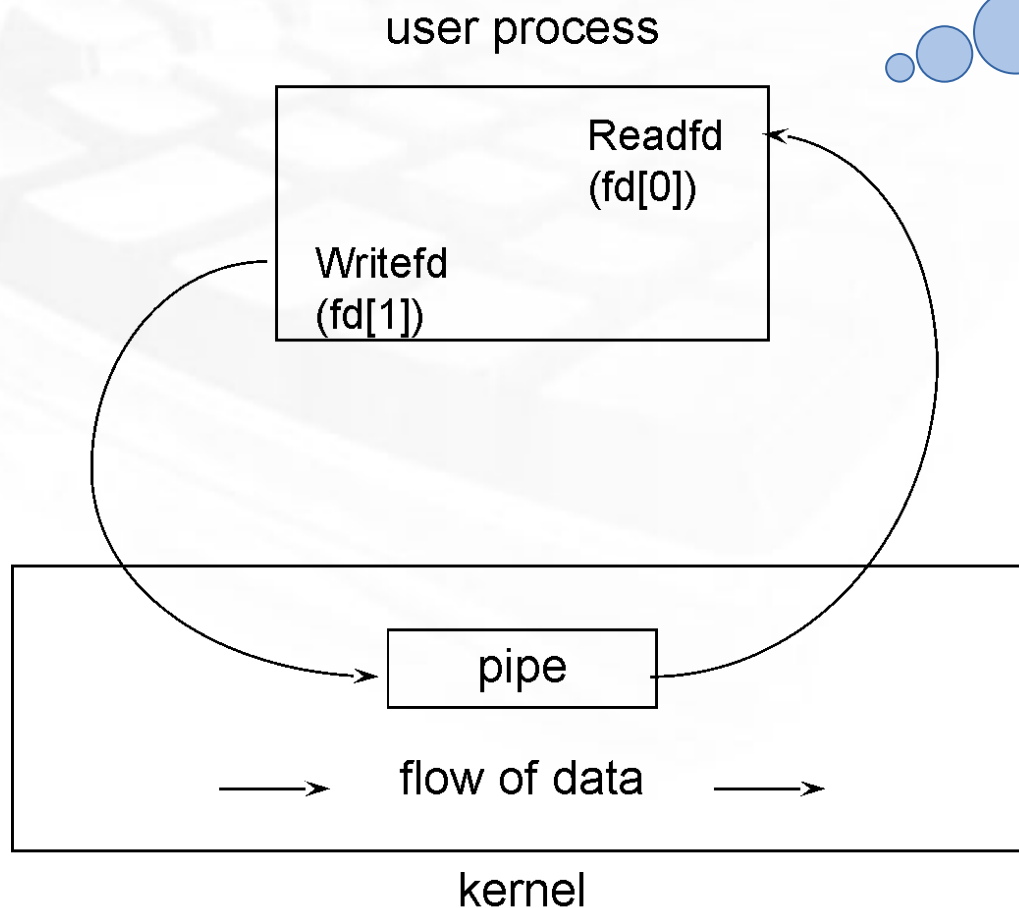
## Criação de Pipes (1)

- *Pipes* constituem um canal de comunicação entre processos pai-filho.
  - Os *pipes* são definidos antes da criação dos processos descendentes.
  - Os *pipes* podem ligar apenas processos com antepassado comum.
- Um *pipe* é criado pela chamada de sistema:
  - **POSIX:** `#include <unistd.h>`  
`int pipe(int fd[2])`
- São retornados dois descritores:
  - Descritor `fd[0]` - aberto para leitura
  - Descritor `fd[1]` - aberto para escrita.

## Criação de Pipes (2)

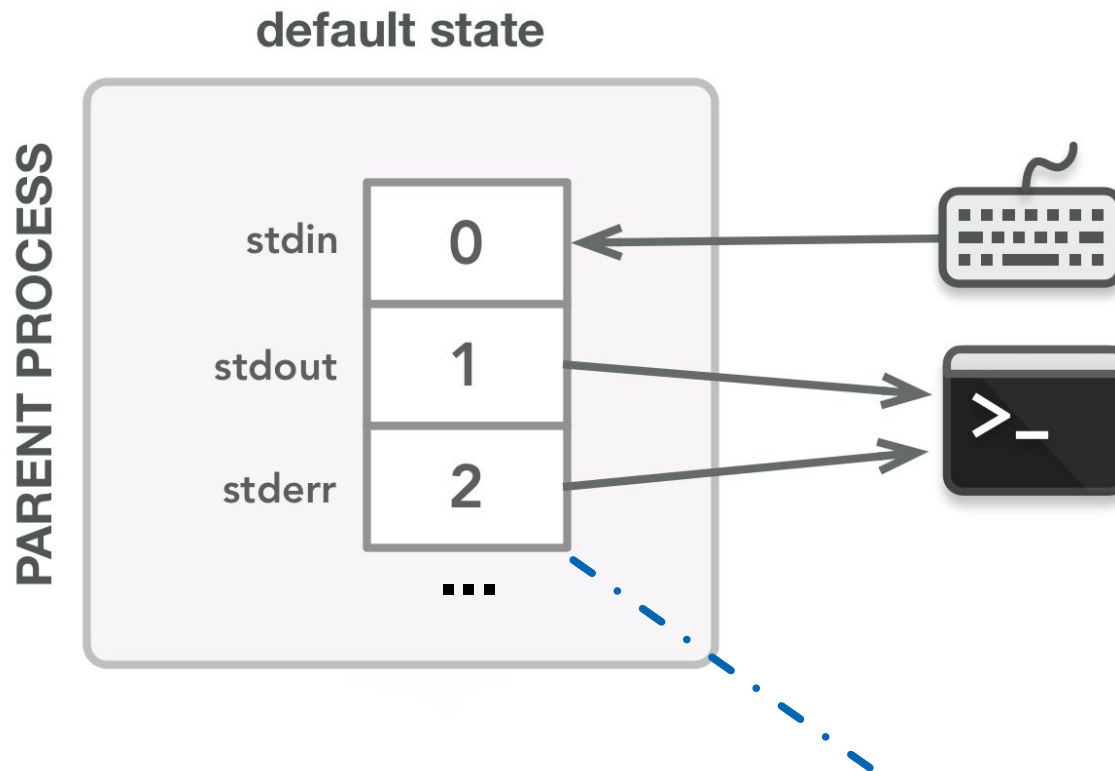


## Criação de Pipes (3)



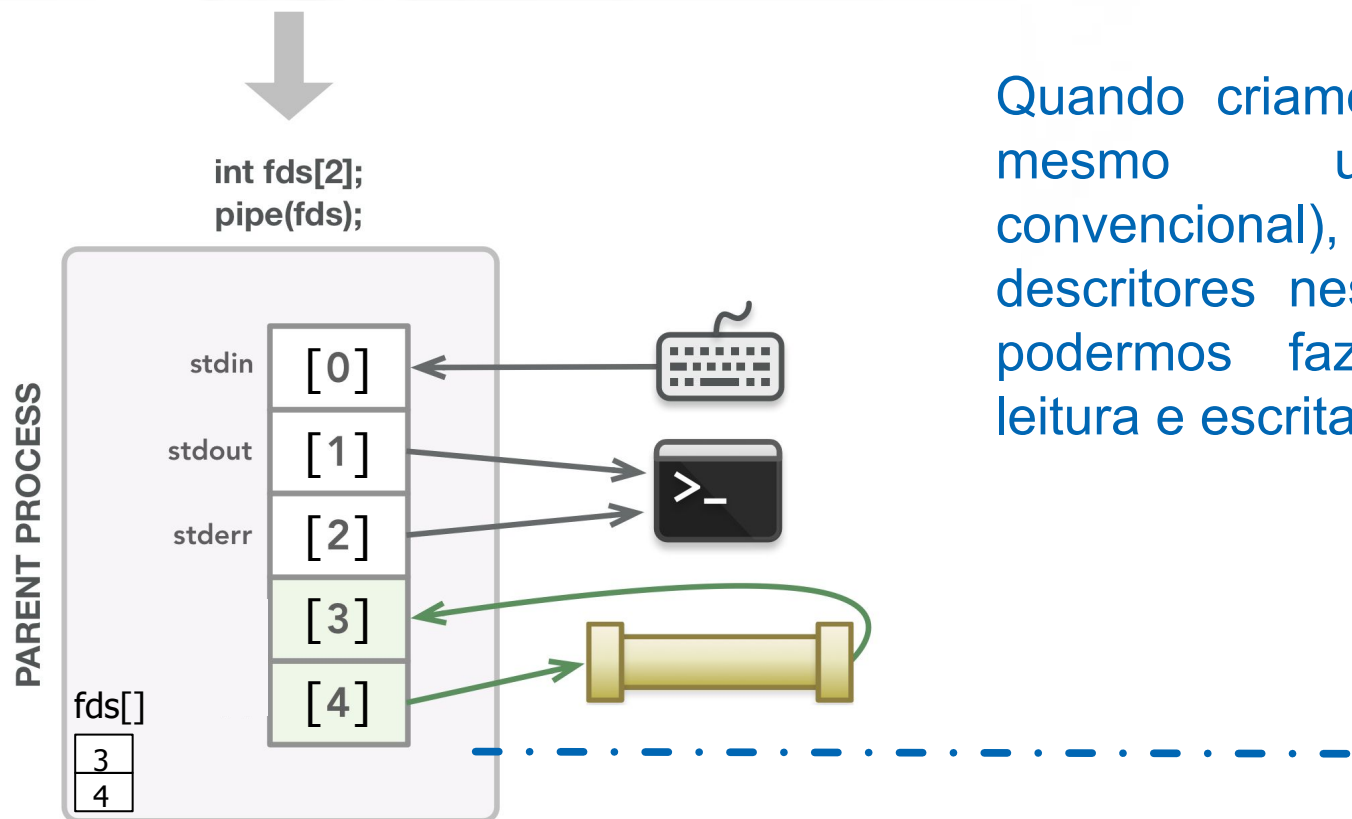
Um pipe criado em um único processo é quase sem utilidade. Normalmente, depois do pipe, o processo chama **fork()**, criando um canal e comunicação entre pai e filho.

# Criação de Pipes (Descritores de Arquivo)



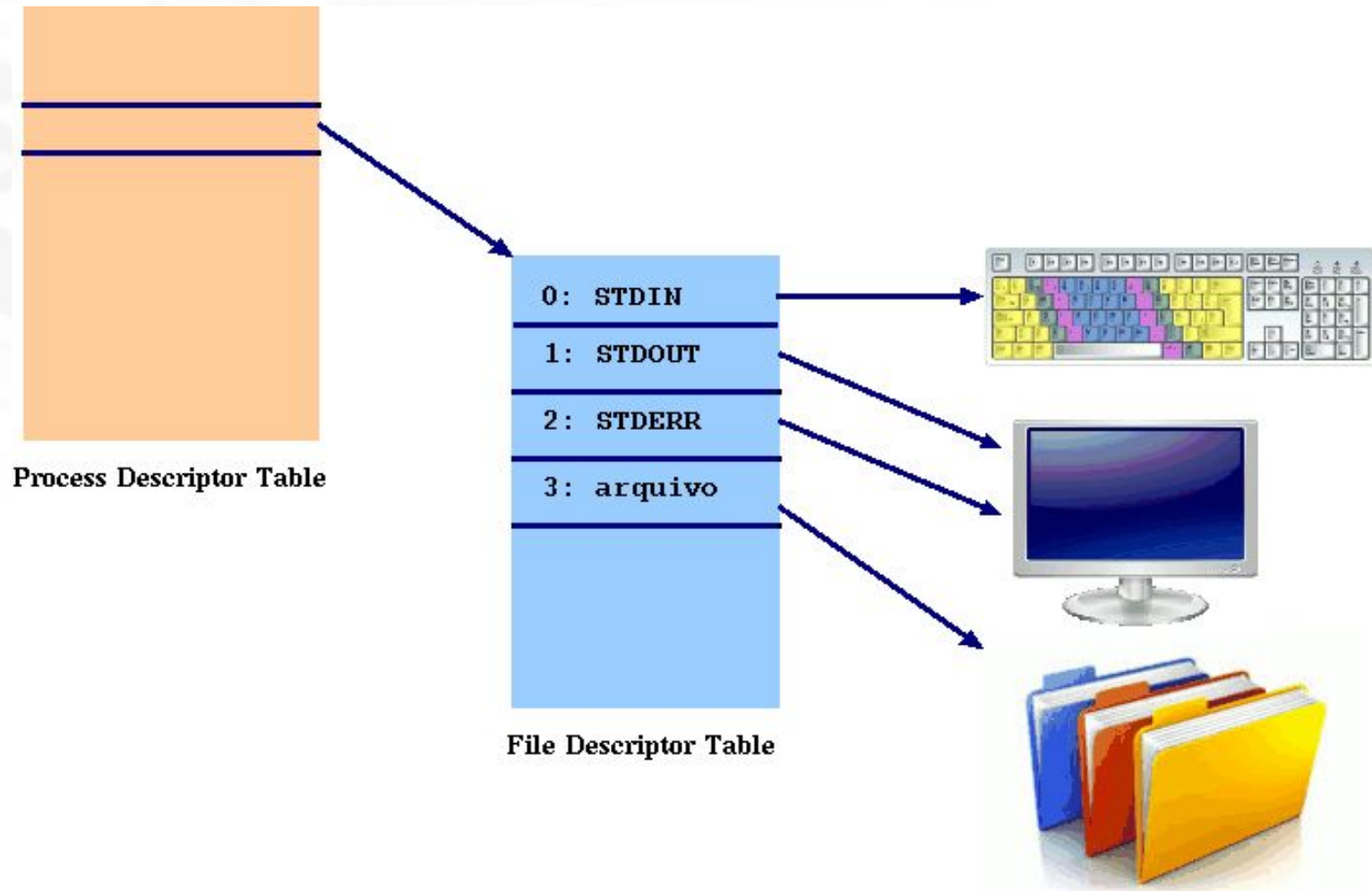
Cada processo tem uma *File Descriptor Table*

# Criação de Pipes (obs. Descritor de Arquivo)

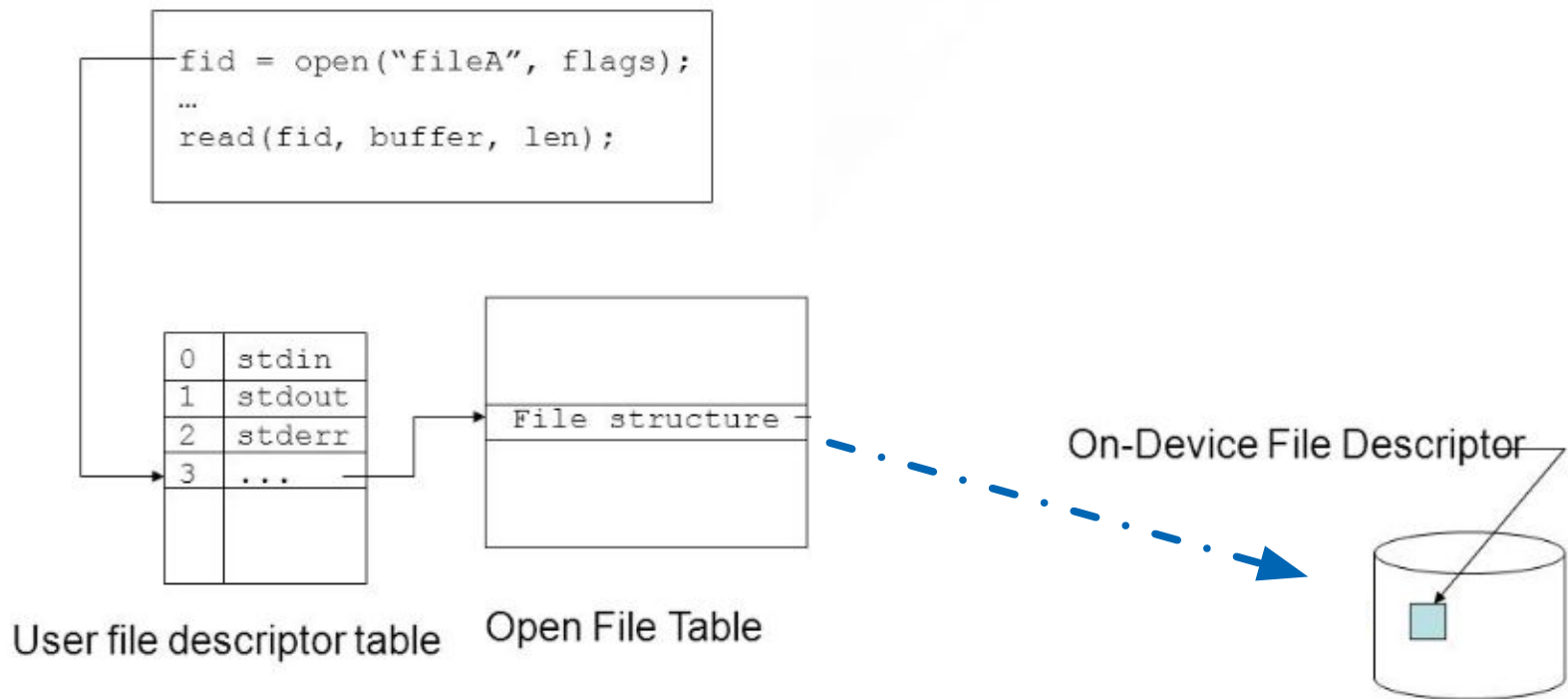


Quando criamos um pipe (ou mesmo um arquivo convencional), são alocados descritores nessa tabela para podermos fazer acesso de leitura e escrita ao pipe.

# Criação de Pipes (Descritor de Arquivo)

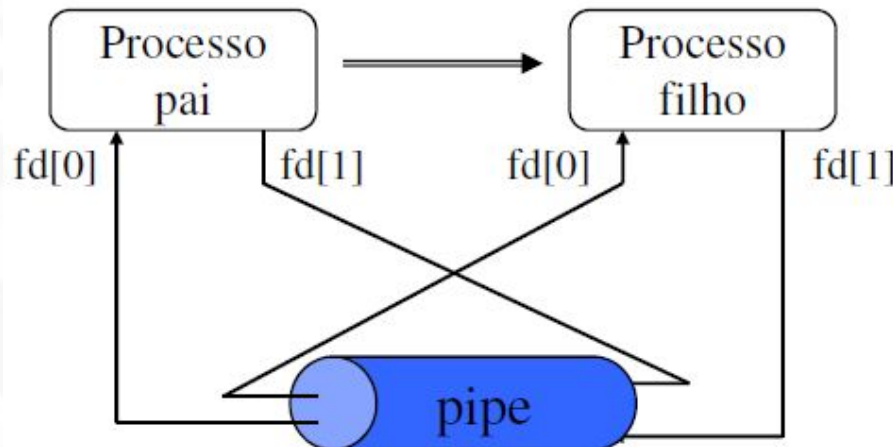


# Criação de Pipes (Descritor de Arquivo)





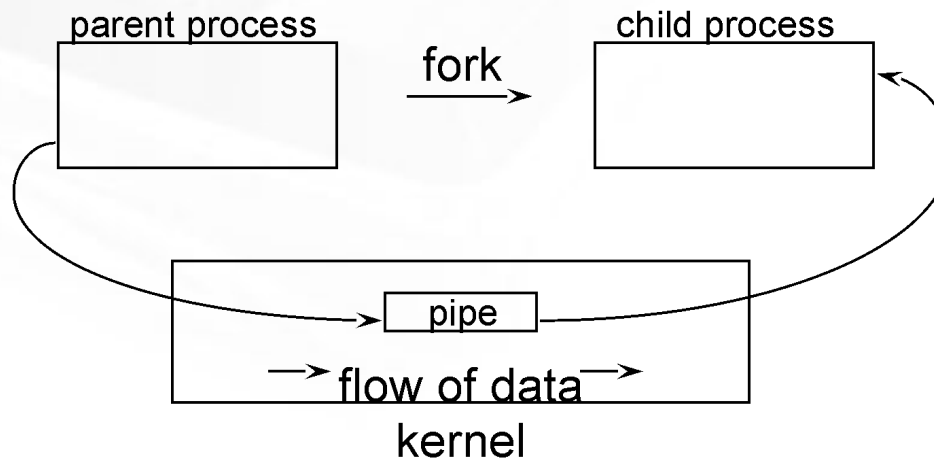
## Criação de Pipes (3)

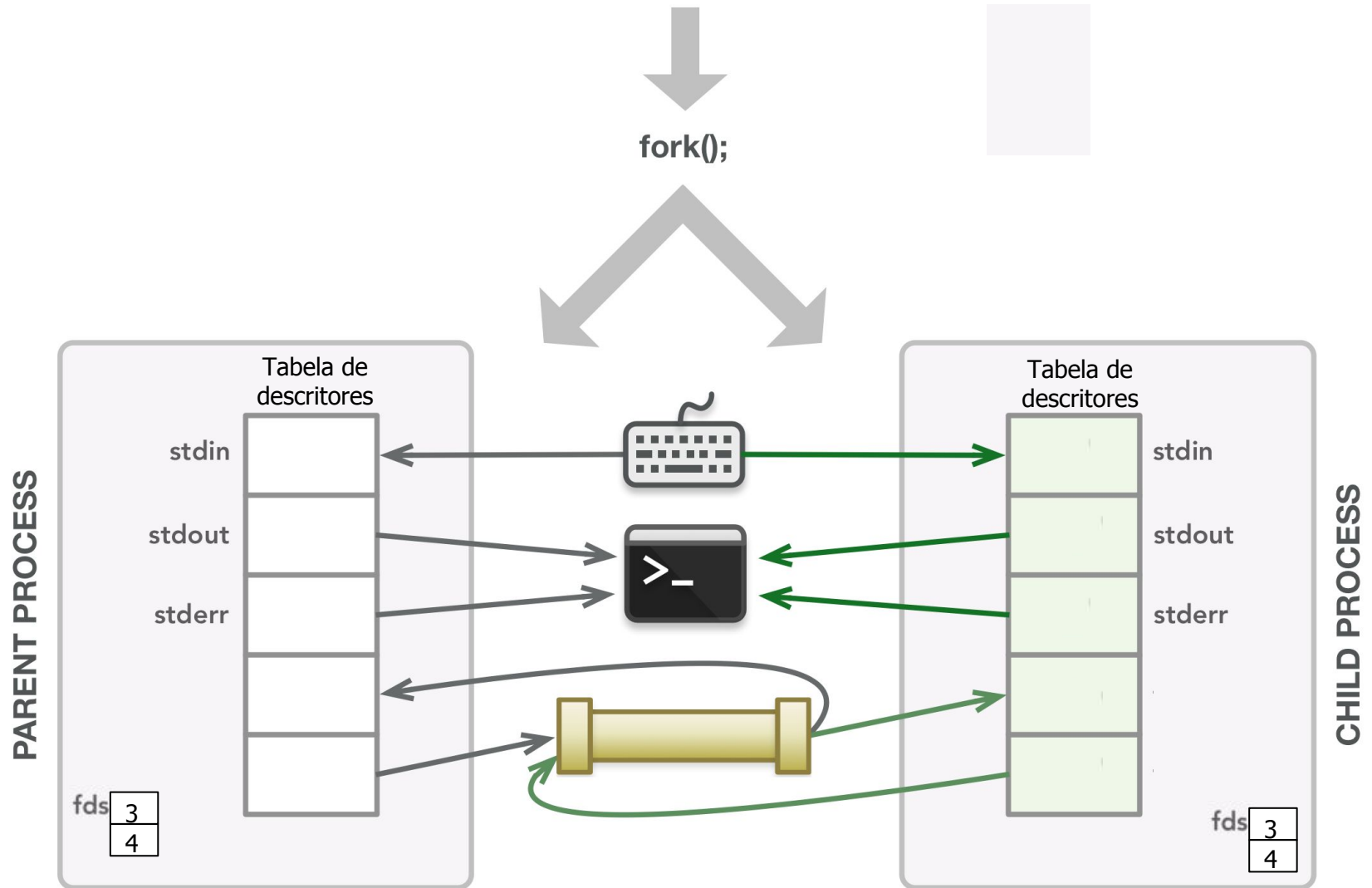


- Quando um processo faz um ***fork()*** depois de criado o *pipe*, o processo filho recebe os mesmos descritores de leitura e escrita do pai. Cada um dos processos deve fechar a extremidade não aproveitada do *pipe*. Ex:
  - se o pai vai escrever, ele fecha o descritor de leitura
  - se o filho vai ler, ele fecha o descritor de escrita

## Criação de Pipes (4)

- Um *pipe* criado em um único processo é quase sem utilidade. Normalmente, depois do *pipe*, o processo chama `fork()`, criando um canal e comunicação entre pai e filho.





## Fechamento de Pipes

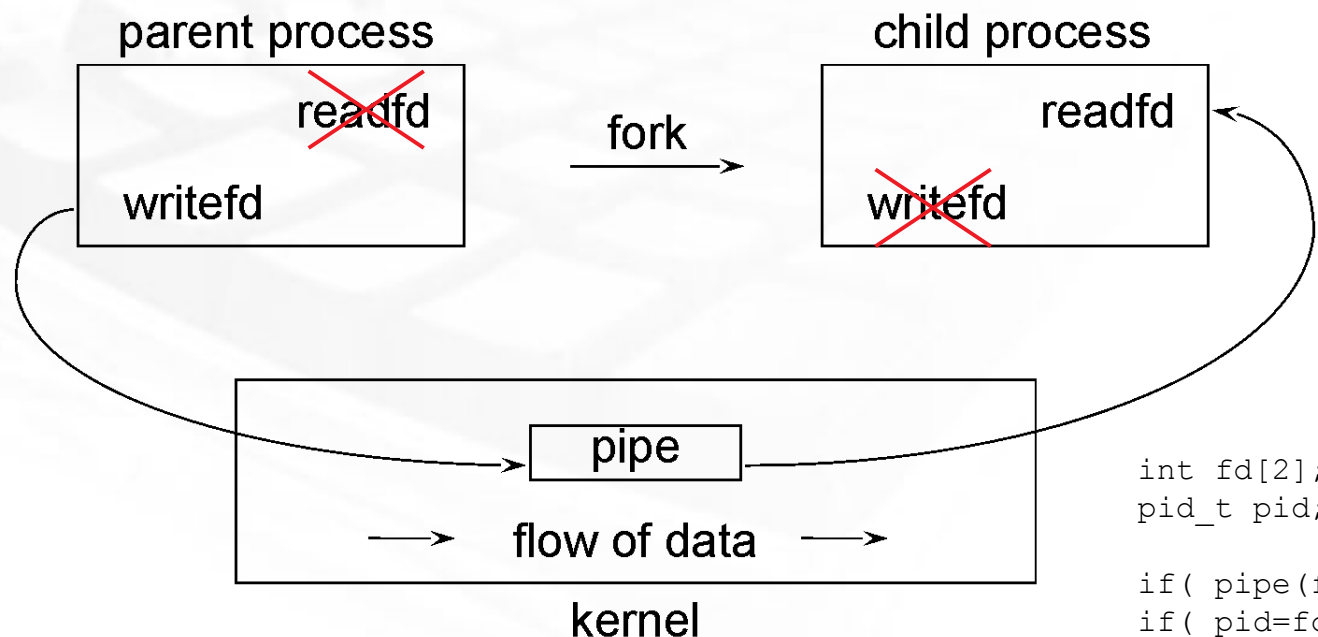
- Depois de usados, ambos os descritores devem ser fechados pela chamada do sistema:

```
POSIX:#include <unistd.h>
      int close (int);
```

- Quando todos os descritores associados a um *pipe* são fechados, todos os dados residentes no *pipe* são perdidos.
- Em caso de sucesso retorna 0. Em caso de erro retorna -1, com causa de erro indicada na variável de ambiente `int errno`.
- Exemplo:

```
int fd[2];
if (pipe(fd)==0) {
    ...
    close(fd[0]); close(fd[1]);
}
```

# Comunicação Pai-Filho Unidirecional



- Processo pai cria o *pipe*.
- Processo pai faz o *fork()*.
- Os descritores são herdados pelo processo filho.
- Pai fecha *fd[0]* // ele vai escrever no pipe
- Filho fecha *fd[1]* // ele vai ler do pipe

```
int fd[2];  
pid_t pid;
```

```
if( pipe(fd)<0 ) exit(1);  
if( pid=fork()<0 ) exit(1);
```

```
if ( pid==0 ) { /* processo filho */  
    close( fd[1] );  
    ...  
}  
if ( pid>0 ) { /* processo pai */  
    close( fd[0] );  
    ...  
}
```

## Escrita e Leitura em Pipes <sup>(1)</sup>

- A comunicação de dados em um *pipe* (leitura e escrita) é feita pelas seguintes chamadas de sistema:

```
POSIX: #include <unistd.h>
```

```
    ssize_t read(int, char *, int);
```

```
    ssize_t write(int, char *, int);
```

- 1º parâmetro: descritor de arquivo.
  - 2º parâmetro: endereço dos dados.
  - 3º parâmetro: número de bytes a comunicar.
- A função retorna o número de bytes efetivamente comunicados.

## Exemplo 1

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

```
#define READ 0
#define WRITE 1
#define STDOUT 1
```

```
int main() {
    int n, fd[2];
    pid_t pid;
```

```
if ( pipe(fd)<0 ) { fprintf(stderr,"Erro no tubo\n");_exit(1); }
```

```
if ( (pid=fork())<0 ) { fprintf(stderr,"Erro no fork\n");_exit(1);
}
```

- Processo filho envia dados para o processo pai.

## Exemplo 1 (cont.)

```
if ( pid>0 ) { /* processo pai */

#define MAX 128
    char line[MAX];
    close(fd[WRITE]);
    n = read(fd[READ],line,MAX);
    write(STDOUT, &line[0], n);
    close(fd[READ]);
    kill(pid,SIGKILL); /* elimina processo descendente */
    _exit(0); }

if ( pid==0 ) { /* processo filho */

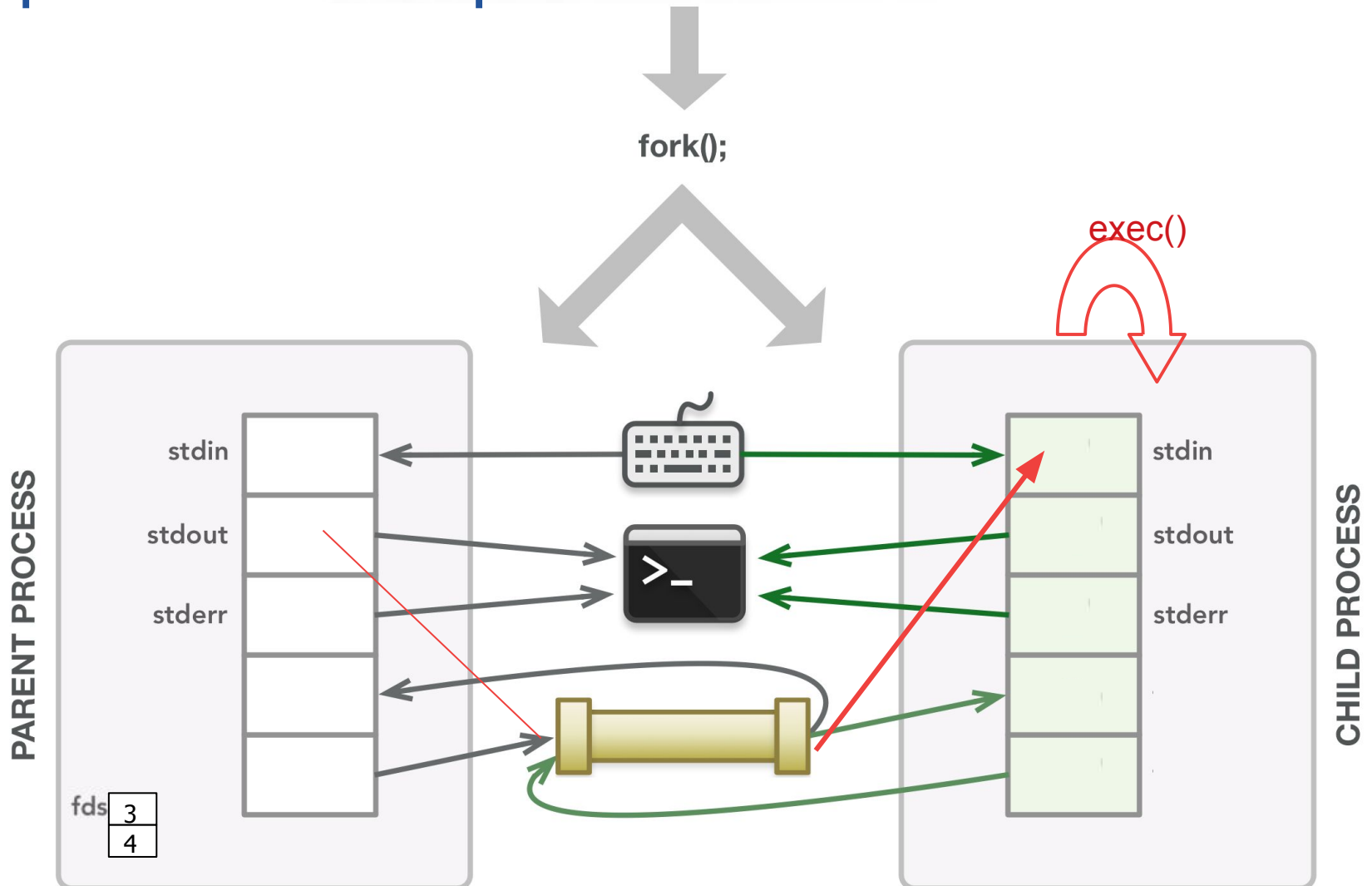
#define LEN 8
    char msg[LEN]={'B','o','m',' ','d','i','a','\n'};
    close( fd[READ] );
    write( fd[WRITE], &msg[0], LEN);
    close( fd[WRITE] );
    pause(); }
```



## Escrita e Leitura em Pipes (2)

- Regras aplicadas aos processos escritores:
  - Escrita para descritor fechado resulta na geração do sinal SIGPIPE
  - Escrita de dimensão inferior a `_POSIX_PIPE_BUF` é **atômica** (i.e., os dados não são entrelaçados).
  - No caso do pedido de escrita ser superior a `_POSIX_PIPE_BUF`, os dados podem ser entrelaçados com pedidos de escrita vindos de outros processos.
    - O número de bytes que podem ser temporariamente armazenados por um *pipe* é indicado por `_POSIX_PIPE_BUF` (512B, definido em `<limits.h>`).
- Regras aplicadas aos processos leitores:
  - Leitura para descritor fechado retorna valor 0.
  - Processo que pretende ler de um *pipe* vazio fica bloqueado até que um processo escreva os dados.

# O que acontece após um `exec()`?

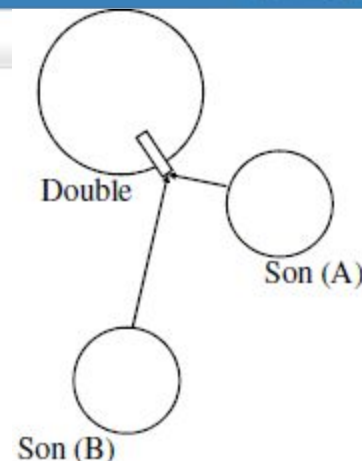


## Exemplo 2 (1)

```
#include <stdio.h> /* double.c */
#include <unistd.h>
#include <sys/types.h>
#include "defs.h"
int main() {
    int fd[2];          /* tubo de leitura do processo principal */
    pid_t pid, pidA, pidB;
    char buf[LEN];
    int i, n, cstat;
    if ( pipe(fd)<0 ) { fprintf(stderr,"Erro no tubo\n");_exit(1); }
    if ( (pid=fork())<0 ) { fprintf(stderr,"Erro no fork\n");_exit(1); }

    if ( pid==0 ) { /* primeiro processo descendente */
        char channel[20];
        close( fd[0] );
        sprintf( channel,"%d",fd[1] );
        execl("./son", "son", channel, "1", NULL); }
    pidA = pid;
```

- Dois processos filhos enviam mensagens para o processo pai.



## Exemplo 2 (2)

```
if ( (pid=fork())<0 ) {fprintf(stderr,"Erro no fork\n");_exit(1);}
if ( pid==0 ) { /* segundo processo descendente */
    char channel[20];
    close( fd[0] );
    sprintf( channel,"%d",fd[1] );
    execl("./son",
          "son", channel, "2", NULL); }
pidB = pid;
close( fd[1] );
n = read( fd[0],buf,LEN );
for( i=0;i<LEN;i++) printf("%c",buf[i]); printf( "\n" );
n = read( fd[0],buf,LEN );
for( i=0;i<LEN;i++) printf("%c",buf[i]); printf( "\n" );
waitpid( pidA,&cstat,0 ); waitpid( pidB,&cstat,0 );
_exit(0); }
```

## Exemplo 2 (3)

```
#define LEN 11      /* defs.h */

#include <unistd.h> /* son.c */
#include <stdlib.h>
#include "defs.h"
int
main(int argc, char *argv[]) {
    /* argv[1] - descritor de escrita; argv[2] - posicao do filho */
    char texto[LEN] = {' ',':',' ', 'B','o','m',' ', 'd','i','a','!'};
    texto[0] = 'A'+atoi(argv[2])-1;
    write( atoi(argv[1]), texto, LEN );
    _exit(0); }
```

# dup2(fd1,fd2)

**Descriptor table  
before dup2 (4 , 1)**

|      |   |
|------|---|
| fd 0 |   |
| fd 1 | a |
| fd 2 |   |
| fd 3 |   |
| fd 4 | b |



**Descriptor table  
after dup2 (4 , 1)**

|      |   |
|------|---|
| fd 0 |   |
| fd 1 | b |
| fd 2 |   |
| fd 3 |   |
| fd 4 | b |

## Exemplo 3

- O que faz esse programa?

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

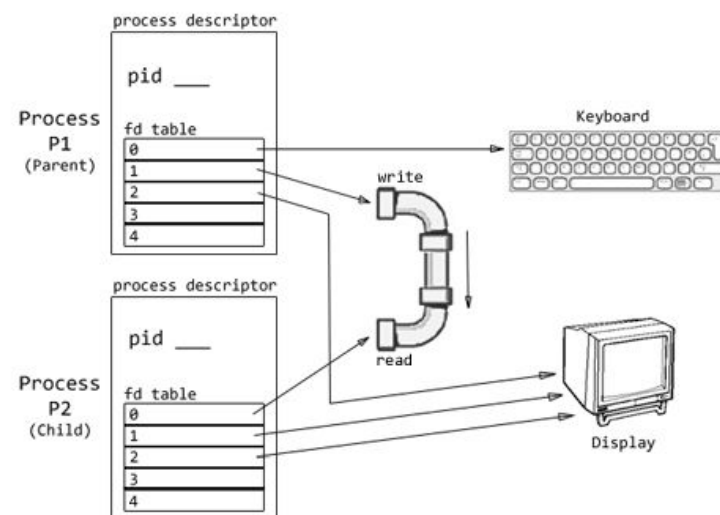
int main(void)
{
    int pfd[2];
    pipe(pfd);
    if (fork() == 0) {
        close(pfd[1]);
        dup2(pfd[0], 0);
        close(pfd[0]);
        execlp("wc", "wc", (char *) 0);
    } else {
        close(pfd[0]);
        dup2(pfd[1], 1);
        close(pfd[1]);
        execlp("ls", "ls", (char *) 0);
    }
    exit(0);
}
```

## Exemplo 3

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

int main(void)
{
    int pfd[2];
    pipe(pfd);
    if (fork() == 0) {
        close(pfd[1]);
        dup2(pfd[0], 0);
        close(pfd[0]);
        execlp("wc", "wc", (char *) 0);
    } else {
        close(pfd[0]);
        dup2(pfd[1], 1);
        close(pfd[1]);
        execlp("ls", "ls", (char *) 0);
    }
    exit(0);
}
```

- Usando-se a técnica de IPC *pipes*, pode-se implementar o comando `"ls | wc"`. Resumidamente: (i) cria-se um pipe; (ii) executa-se um `fork`; (iii) o processo pai chama `exec` para executar `"ls"`; (iv) o processo filho chama `exec` para executar `"wc"`.
- O problema é que normalmente o comando `"ls"` escreve na saída padrão 1 e `"wc"` lê da entrada padrão 0. Como então associar a saída padrão com a saída de um *pipe* e a entrada padrão com a entrada de um *pipe*? Isso pode ser conseguido através da chamada de sistema `int dup2(int oldfd, int newfd)`.
- Essa chamada cria uma cópia de um descritor de arquivo existente (`oldfd`) e fornece um novo descritor (`newfd`) tendo exatamente as mesmas características que aquele passado como argumento na chamada. A chamada `dup2` fecha antes `newfd` se ele já estiver aberto.





## Exemplo 4

- O que faz este programa?

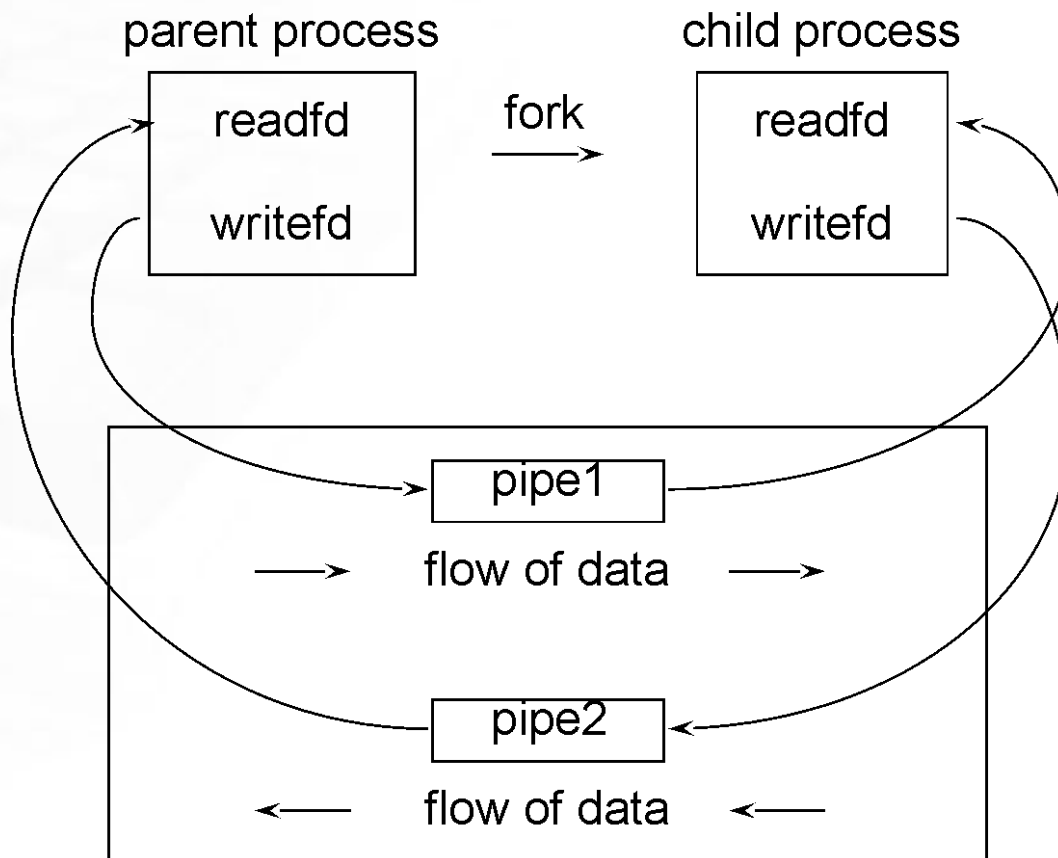
```
int count=0;

void alarm_action(int par){
    printf("write blocked after %d chars \n", count);
    exit(0);
}

int main(){
    int p[2];
    char c='x';
    if (pipe(p) < 0)
        perror("pipe call");
    signal(SIGALRM,alarm_action);
    for(;;) {
        alarm(20); //Seria diferente se fosse fora do "for"?
        write(p[1],&c,1);
        if(++count%1024)==0)
            printf("%d chars in pipe\n", count);
    }
}
```

# Comunicação Pai-Filho Bi-Direcional (Idealmente!!)

- Ex: pai envia *filename* para o filho. Filho abre e lê o arquivo, e retorna o conteúdo para o pai.
  - Pai cria pipe1 e pipe2.
  - Pai fecha descritor de leitura de pipe1.
  - Pai fecha descritor de escrita de pipe2.
  - Filho fecha descritor de escrita de pipe1.
  - Filho fecha descritor de leitura de pipe2.



## Fila (FIFO, Named Pipe)

- Trata-se de uma extensão do conceito de *pipe*.
  - *Pipes* só podem ser usados por processos que tenham um ancestral comum.
  - Filas (FIFOs – First In First Out), também designados de “tubos nomeados” (“*named pipes*”), permitem a comunicação entre processos não relacionados.
- As Filas:
  - são referenciadas por um identificador dentro do sistema de arquivos
  - persistem além da vida do processo
  - são mantidas no sistema de arquivos até serem apagadas (ou seja, precisam ser eliminadas quando não tiverem mais uso).
- Normalmente são implementadas por meio de arquivos especiais (tipo: pipe).
  - Um processo abre a Fila para escrita, outro para leitura.

## Criação de Filas <sup>(1)</sup>

- Uma fila é criada pela chamada de sistema:

```
POSIX: #include <sys/stat.h>
        int mkfifo(char *, mode_t);
```

- 1º parâmetro: nome do arquivo.
- 2º parâmetro: identifica as permissões de acesso, iguais a qualquer arquivo, determinados por OU de grupos de bits.
- As permissões de acesso também podem ser indicados por 3 dígitos octais, cada um representando os valores binários de rwx (Read, Write, eXecute).
- Exemplo: modo 644 indica permissões de acesso:  
Dono: 6 = 110 (leitura e escrita)  
Grupo e Outros: 4 = 100 (leitura)

## Criação de Filas (2)

- Uma fila também pode ser criada, via shell, por meio do comando:

```
#mkfifo [-m modo] fichID
```

- Exemplo 1:

```
[rgc@asterix]$ mkfifo -m 644 tubo
```

```
[rgc@asterix]$ ls -l tubo
```

```
prw-r--r-- 1 rgc docentes 0 2008-10-11 15:56 tubo
```

```
[rgc@asterix]$
```

OBS: **p** indica que "tubo" é um arquivo do tipo named pipe

- Exemplo 2:

```
#mkfifo teste
```

```
#cat < teste /* o pipe fica esperando até obter algum dado */
```

Em outra tela execute:

```
# ls > teste /* a saída do comando ls será redirecionada para o  
pipe nomeado "teste" */
```

## Eliminação de Filas

- Uma fila é eliminada pela seguinte chamada ao sistema:

```
POSIX:#include <unistd.h>
      int unlink(char *);
```

- 1º parâmetro: nome do arquivo.
- Uma fila também é eliminada via shell, usando o comando:

```
#rm fichID
```

## Abertura de Filas <sup>(1)</sup>

- Antes de ser usada, a fila tem de ser aberta pela chamada de sistema:

```
POSIX: #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>
        int open(char *,int);
```

- 1º parâmetro: nome do arquivo.
- 2º parâmetro : formado por bits que indicam:
  - Modos de acesso: `O_RDONLY` (leitura apenas) ou `O_WRONLY` (escrita apenas)
  - Opções de abertura: `O_CREAT` (criado se não existir)
  - `O_NONBLOCK` (operação de E/S não são bloqueadas)
- O valor de retorno é o descritor da fila (positivo) ou erro (-1).

## Abertura de Filas (2)

- Regras aplicadas na abertura de filas:
  - Se um processo tentar abrir uma fila em modo de leitura, e nesse instante não houver um processo que tenha aberto a fila em modo de acesso de escrita, o processo fica bloqueado, exceto se:
    - a opção `O_NONBLOCK` tiver sido indicada no `open()` (nesse caso, é devolvido o valor `-1` e `errno` fica com valor `ENXIO`).
  - Se um processo tentar abrir uma fila em modo de escrita, e nesse instante não houver um processo que tenha aberto a fila em modo de acesso de leitura, o processo fica bloqueado, exceto se:
    - a opção `O_NONBLOCK` tiver sido indicada no `open()` (nesse caso, é devolvido o valor `-1` e `errno` fica com valor `ENXIO`).



## Leitura e Escrita em Filas (1)

- A comunicação em uma fila é feita pelas mesmas chamadas de sistema dos *pipes*:

```
POSIX: #include <unistd.h>
        ssize_t read(int, char *,int);
        ssize_t write(int, char *,int);
```

- Regras aplicadas aos processos escritores:
  - Escrita para uma fila que ainda não foi aberta para leitura gera o sinal `SIGPIPE` (termina o processo). Se ignorado `read` retorna -1 com `errno` igual a `EPIPE`.
  - Após o último processo escritor tiver encerrado a fila, os processos leitores recebem `EOF`.

## Exemplo

- Dois processos *writer* enviam mensagens para o processo *reader* através de uma fila.
  - O identificador da fila e o comprimento da memória tampão (buffer) é definida no arquivo à parte.

```
#define LEN 100  
#define FNAME "testFIFO"
```

## Exemplo (cont.)

### **writer.c**

```
#include <stdio.h>
#include <string.h>
#include <sys/file.h>

#include "defs.h"

main() {
    int fd, i;
    char msg[LEN];
    do {
        fd=open(FNAME,O_WRONLY);
        if (fd==-1) sleep(1); }
    while (fd==-1);
    for( i=1;i<=3;i++ ) {
        sprintf(msg,"Hello no %d from process %d\n",i,getpid());
        write( fd,msg,strlen(msg)+1 );
        sleep(3); }
    close(fd);
}
```

## Exemplo (cont.)

### **reader.c**

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include "defs.h"

int readChar(int fd, char *buf) {
    int n;
    do n=read(fd,buf,1);
    while (n>0 && *buf++!='\0');
    return n>0; }

main() {
    int fd;
    char str[LEN];
    mkfifo(FNAME,0660);
    fd=open(FNAME,O_RDONLY);
    if (fd<0) { printf("Erro na abertura da fila\n"); exit(1); }
    while (readChar(fd,str)) printf("%s",str);
    close(fd); }
```

## Exemplo (cont.)

```
[rgc@asterix FIFO]$ reader & writer & writer &
```

```
[1] 7528
```

```
[2] 7529      ⇔      PIDs dos processos lançados
```

```
[3] 7530
```

← Lançados 1 leitor e  
2 escritores

```
[rgc@asterix FIFO]$ Hello no 1 from process 7530
```

```
Hello no 1 from process 7529
```

```
Hello no 2 from process 7530
```

```
Hello no 2 from process 7529
```

```
Hello no 3 from process 7530
```

```
Hello no 3 from process 7529
```

```
[1] Done          reader
```

```
[2]- Done          writer
```

```
[3]+ Done          writer
```

```
[rgc@asterix FIFO]$
```

## Exemplo (cont.)

```
[rgc@asterix FIFO]$ ls -l
total 48
-rw-r----- 1 rgc ec-ps      42   2007-05-17 15:17 defs.h
-rwxr----- 1 rgc ec-ps    5420   2007-05-17 15:45 reader
-rw-r--r-- 1 rgc ec-ps     442   2007-05-17 15:45 reader.c
prw-r----- 1 rgc docentes    0   2008-10-11 16:01 testFIFO
-rwxr----- 1 rgc ec-ps    5456   2007-05-17 15:23 writer
-rw-r--r-- 1 rgc ec-ps     371   2007-05-17 15:23 writer.c
```

```
[rgc@asterix FIFO]$ rm testFIFO
rm: remove fifo `testFIFO'? y
[rgc@asterix FIFO]$
```

Observe que a fila não havia sido eliminada pelos programas (arquivo testFIFO tem tipo **p**, de **named pipe**).