IPC
Interprocess communication

conceitos de IPC, pipes

Taisy Weber

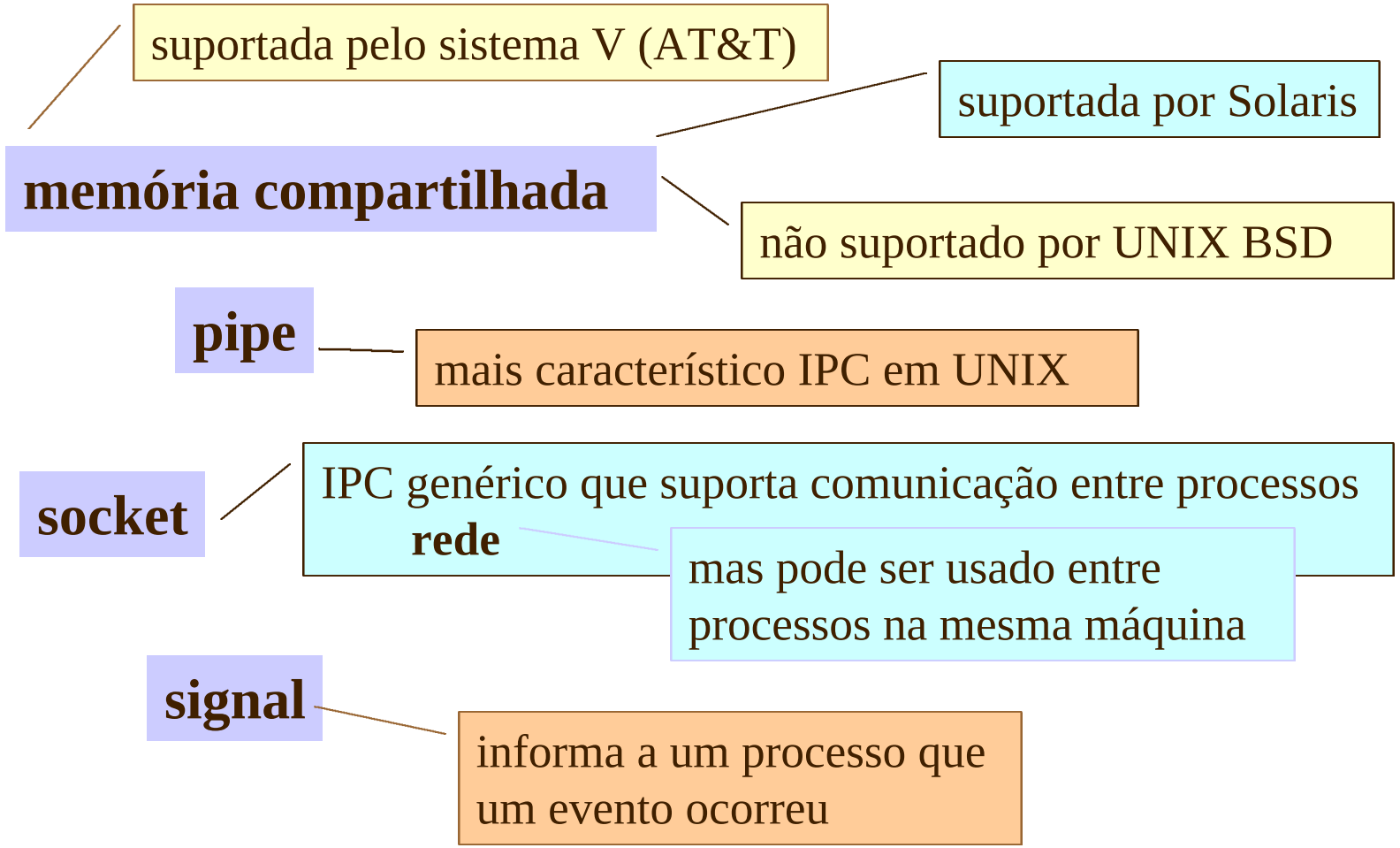
Comunicação entre processos

Mitchell, cap 5

Matthew & Stones, cap 12

- Pipes e FIFO
  - Definição, chamadas de sistemas, utilização de
- Message queues.
- Memória compartilhada.
- Sockets.
  - Definição, chamadas de sistemas, utilização de

IPC em UNIX


suportada pelo sistema V (AT&T)
suportada por Solaris
memória compartilhada
não suportado por UNIX BSD
pipe
mais característico IPC em UNIX
socket
IPC genérico que suporta comunicação entre processos
rede
mas pode ser usado entre processos na mesma máquina
signal
informa a um processo que um evento ocorreu

Linux

- ✓ suporta os mecanismos IPC padrões do

pipe

sockets

signals

- ✓ e também os mecanismos previstos pelo

semáforos

message queues

memória compartilhada

Exemplos de IPC

- pipes

mecanismo IPC em UNIX mais característico

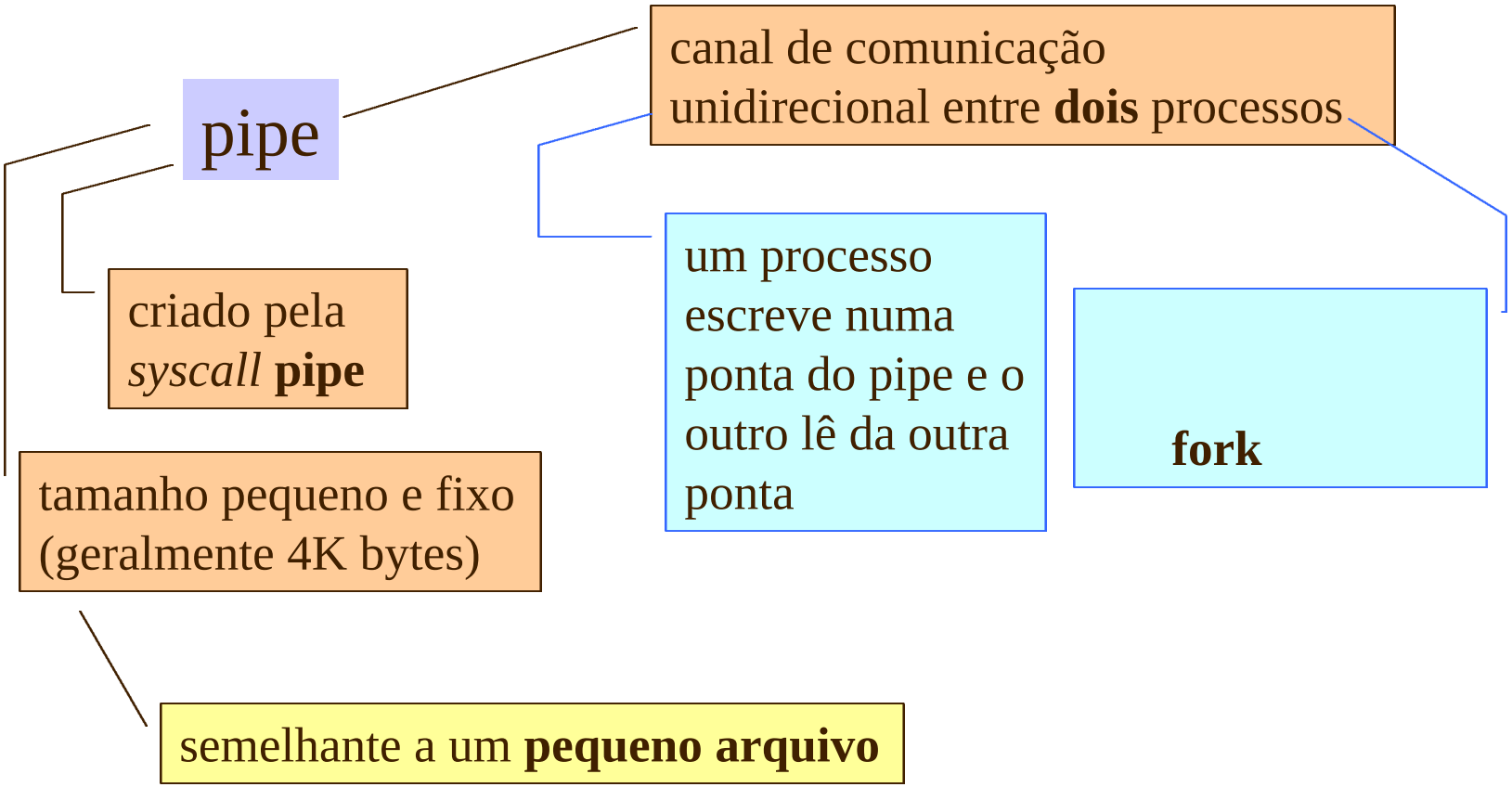
a mais antiga forma de IPC

usada inclusive na linguagem de comandos

- sockets

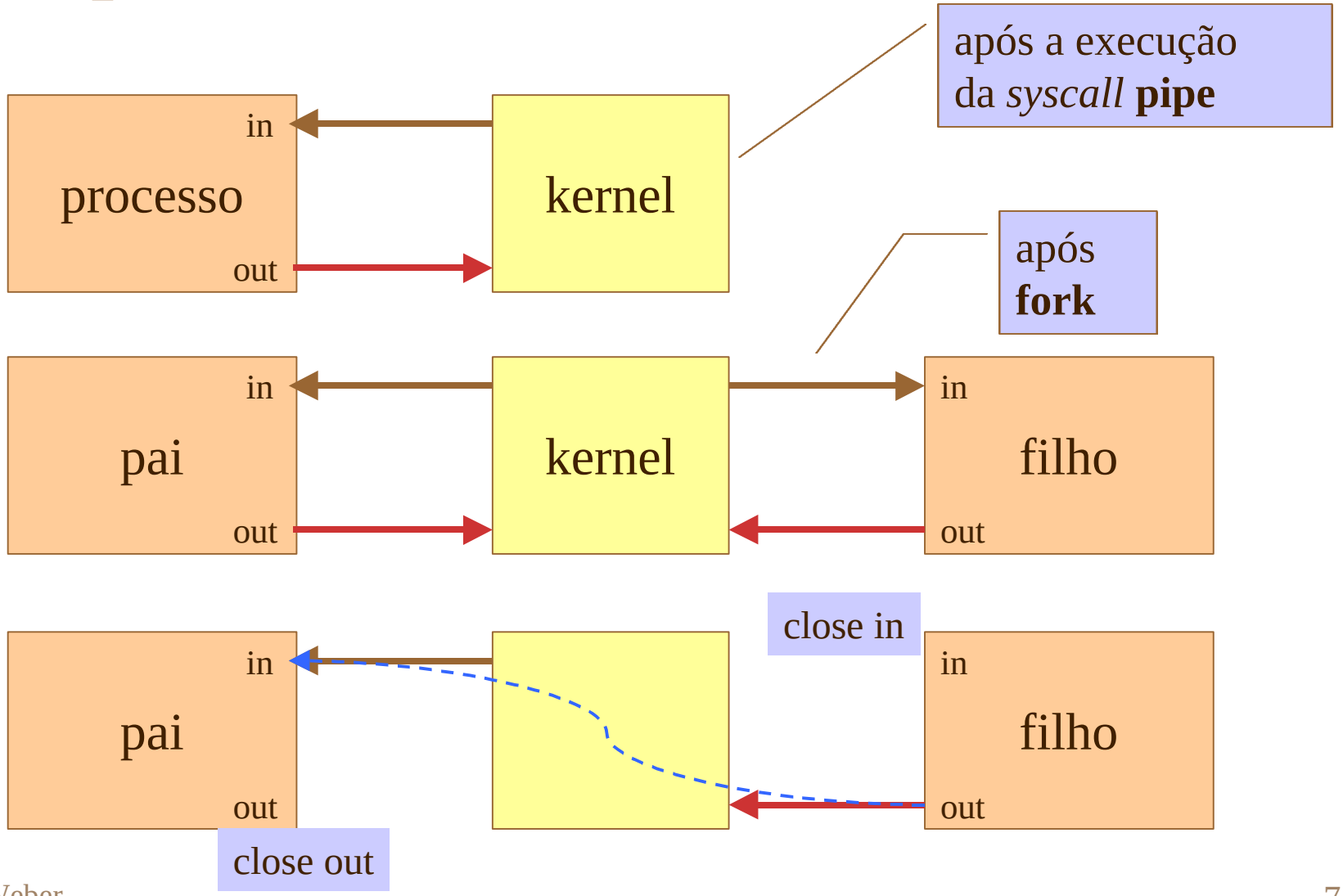
mecanismo genérico de IPC que suporta comunicação entre processos pela **rede,** mas que pode ser usado também para comunicação de processos na mesma

UNIX pipe


pipe
canal de comunicação unidirecional entre **dois** processos
criado pela *syscall* **pipe**
um processo escreve numa ponta do pipe e o outro lê da outra ponta
fork
tamanho pequeno e fixo (geralmente 4K bytes)
semelhante a um **pequeno arquivo**

Pipe

pipe conecta uma **saída** de um processo a uma **entrada** de outro processo



System call *pipe*

o pipe é unidirecional

system call: `pipe( );`

argumento: array de inteiros de 2 posições - fd

`fd[0]` é usado para **ler**

`fd[1]` é usado para **escrever**

descritores de arquivos

resultado: 0 (sucesso), -1 (em caso de erro)

se a chamada for bem sucedida, o descritores de arquivo

um descritor é usado para entrada no pipe (write),

o outro é usado para obter dados do pipe (read).

Função *pipe*

```
#include <unistd.h>

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

lembrar que são **descritores de arquivo** e não *streams*

`read` e `write` de baixo nível

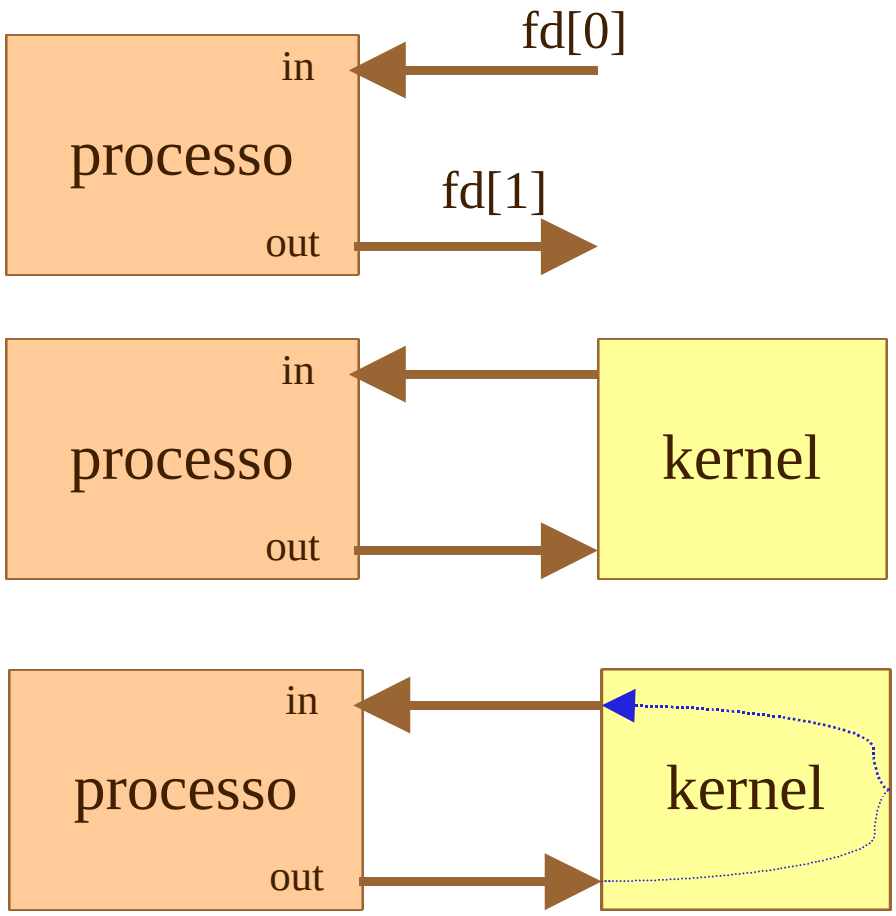
em caso de falha, retorna -1 e liga `errno`

escritas e leituras obedecem uma ordem FIFO

após a criação do filho, pai e filho precisam estabelecer o sentido da comunicação

de pai para filho ou de filho para pai

Pipe: criação



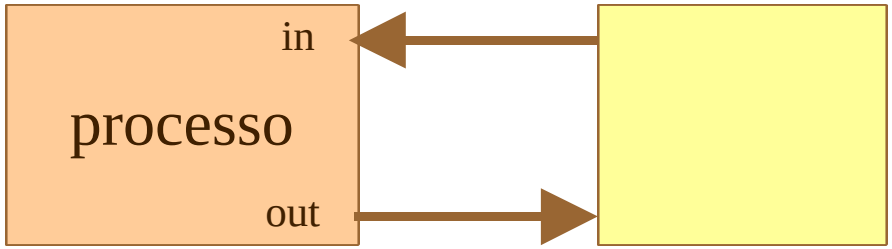
processo cria um pipe

pipes são representados internamente por um **inode**

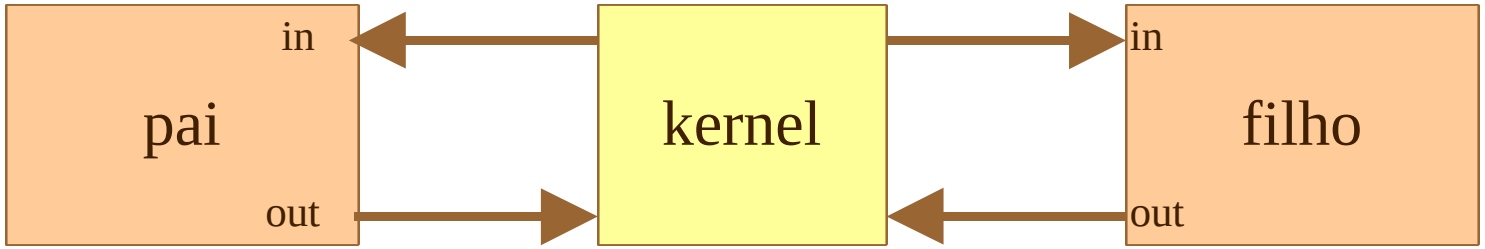
o kernel cria 2 *file descriptors*: um **fd** é para entrada no pipe (write), o outro **fd** é usado para obter dados do pipe (read).

até esse ponto o processo só pode se comunicar com ele mesmo

Pipe & fork: comunicação entre processos



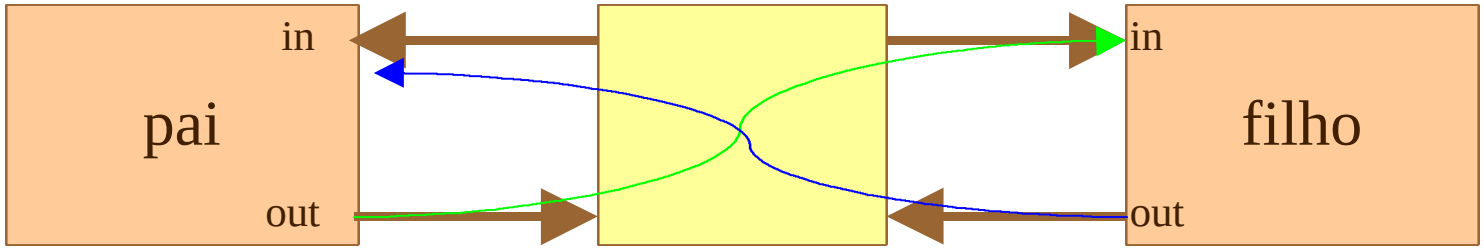
o processo bifurca (*forks)* um processo filho (o filho herda todos os descritores de arquivos abertos pelo pai)



comunicação multiprocesso entre pai e filho fica estabelecida

mas podem ocorrer conflitos de escrita (pai e filho escrevem em o canal ainda não está perfeitamente formado

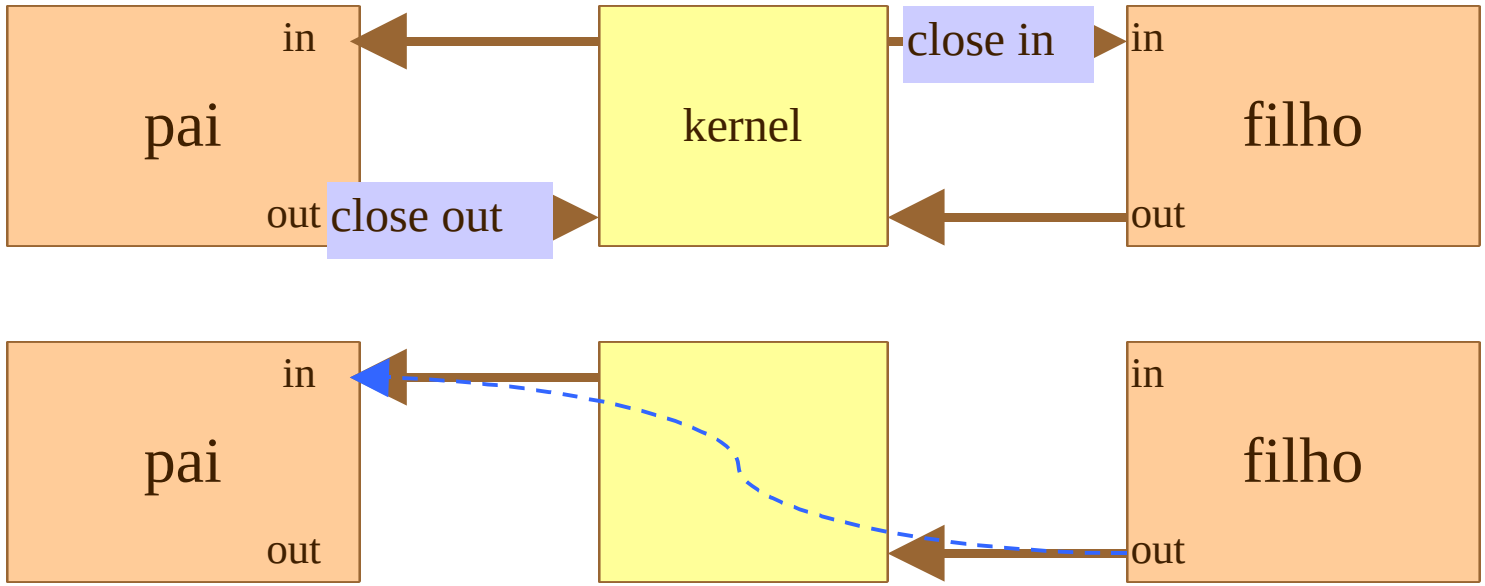
Direção da comunicação



- do pai para o filho ou vice e versa?
  - os 2 processos entram em acordo, e fecham (*close*) a ponta do pipe que cada um não vai usar

Pipe, fork e close

vamos supor que o filho se comunica com o pai



um canal (pipe) entre pai e filho foi agora criado

e pode ser usado para comunicação unidirecional entre os dois processos através do kernel

Como usar pipes?

- ✓ acesso a um pipe
  - mesmas funções usadas para *low-level file I/O*
    - pipes são representados internamente como um **inode**
  - write( ) para enviar dados pelo pipe
  - read( ) para obter dados do pipe.
    - certas funções, como lseek( ), não operam com descritores para pipes.

Exemplo 1: criação do pipe

Matthew & Stones, cap 12

fonte `pipe1.c`

após a criação o processo pode **pipe**, mas nenhum outro processo pode usá-lo

pipe

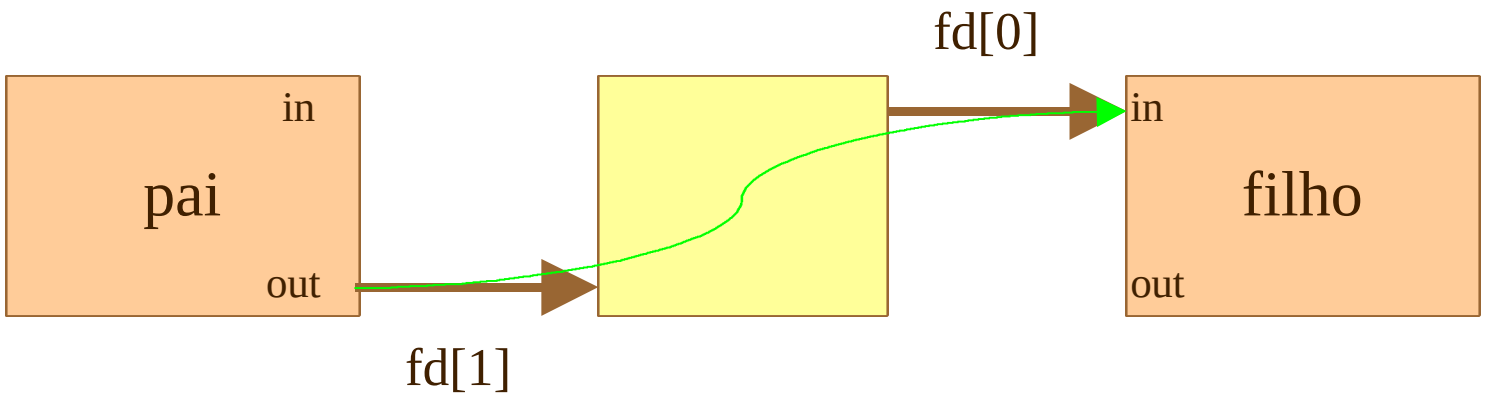
```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    if (pipe(file_pipes) == 0) {
        data_processed = write(file_pipes[1], some_data, strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

lab

Exemplo 2: pai e filho



pai escreve em fd[1]
filho lê em fd[0]

Exemplo 2: pipe & fork

fonte `pipe2.c`

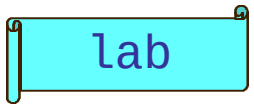
Matthew & Stones, cap 12

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```



Exemplo 2: pai e filho

```
// We've made sure the fork worked
        the child process

                                                    aqui o processo é filho

           data_                                  buffer, BUFSIZ);
           printf       %d bytes: %s\n", data_


             we must be the parent process

                                                    aqui o processo é pai

           data_
                                 strlen
           printf        %d bytes\n", data_
```

pai e filho executam o mesmo programa
próximo passo e fazer pai e filho executaram programas diferentes

Pipe & exec

- ✓ dificuldade
  - – processo novo (cuja imagem foi substituída) precisa saber qual o
- ✓ solução
  - – passar o fd como parâmetro ao novo programa
    - o fd é apenas um número
    - pode ser facilmente passado como parâmetro

Exemplo 3

parte 1 de 3

fonte `pipe3.c`

Matthew & Stones, cap 12

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

lab

Exemplo 3

parte 2 de 3

fonte `pipe3.c`

```
if (fork_result == 0) {
    sprintf(buffer, "%d", file_pipes[0]);
    (void)execl("pipe4", "pipe4", buffer, (char *)0);
    exit(EXIT_FAILURE);
}

else {
    data_processed = write(file_pipes[1], some_data,
                           strlen
    printf
```

aqui o processo é filho

aqui o processo é pai

sprintf - coloca o descritor no buffer

pipe4

ver parte 3

argumentos:
`argv[0]` - nome do programa
`argv[1]` - descritor do arquivo fd[0]
`(char *)0` termina os parâmetros

Exemplo 3

parte 3 de 3

fonte `pipe4.c`

```
// The 'consumer' program, pipe4.c, that reads the data is much simpler.

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

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

Pipes com standard input e output

- ✓ para tornar o processo filho mais simples
  - providenciar para que um dos descritores possua um valor conhecido
  - stdin = 0 e stdout=1
- – principal vantagem
  - invocar programas que não esperam receber um descritor como parâmetro
    - – por exemplo: programas-comandos da

Duplicando descritores de pipes

- forçando o uso de stdin: usar a função dup

```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```

dup - duplica um *file descriptor*
cria um novo descritor apontando para o mesmo arquivo
o menor disponível
pode ser igual a, ou o primeiro maior
que o segundo argumento

igual a zero, fechamos
dup o fd liberado (fd0), pois dup retorna sempre o menor valor disponível

Exemplo 4

fonte `pipe5.c`

Matthew & Stones, cap 12

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

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    pid_t fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
```

lab

Exemplo 4

fecha stdin e libera fd0

duplica fd[0] criando um novo fd com valor 0 que aponta para o pipe

processo é filho

fecha fd[0] e fd[1] por não serem necessários

exec comando od da shell

processo é pai

```
if (fork_result == (pid_t)0) {
    close(0);
    dup(file_pipes[0]);
    close(file_pipes[0]);
    close(file_pipes[1]);

    execlp("od", "od", "-c", (char *)0);
    exit(EXIT_FAILURE);
}
else {
    close(file_pipes[0]);
    data_processed = write(file_pipes[1], some_data,
                           strlen
    close
    printf
```

Fim

- de IPC pipe ...
- antes de terminar,
  - não esqueça de executar os exemplos e anotar as