FSO
03 de Dezembro 2018

Processos cooperantes

- Sincronização
- Comunicação

  Exemplo: *Pipes* no UNIX

Bibliografia: OSTEP 5.4 (pipes)

Processos cooperantes

- **Os processos são independentes** o que se passa num processo não afecta nada o que se passa noutra
- **Processos independentes podem cooperar** Um programa constituído por vários processos
- Vantagens da cooperação entre processos
  - Partilha da informação
  - Aceleração das computações
  - Facilidade no desenvolvimento (Modularidade)
  - Natureza do problema a resolver

Os processos cooperantes têm de

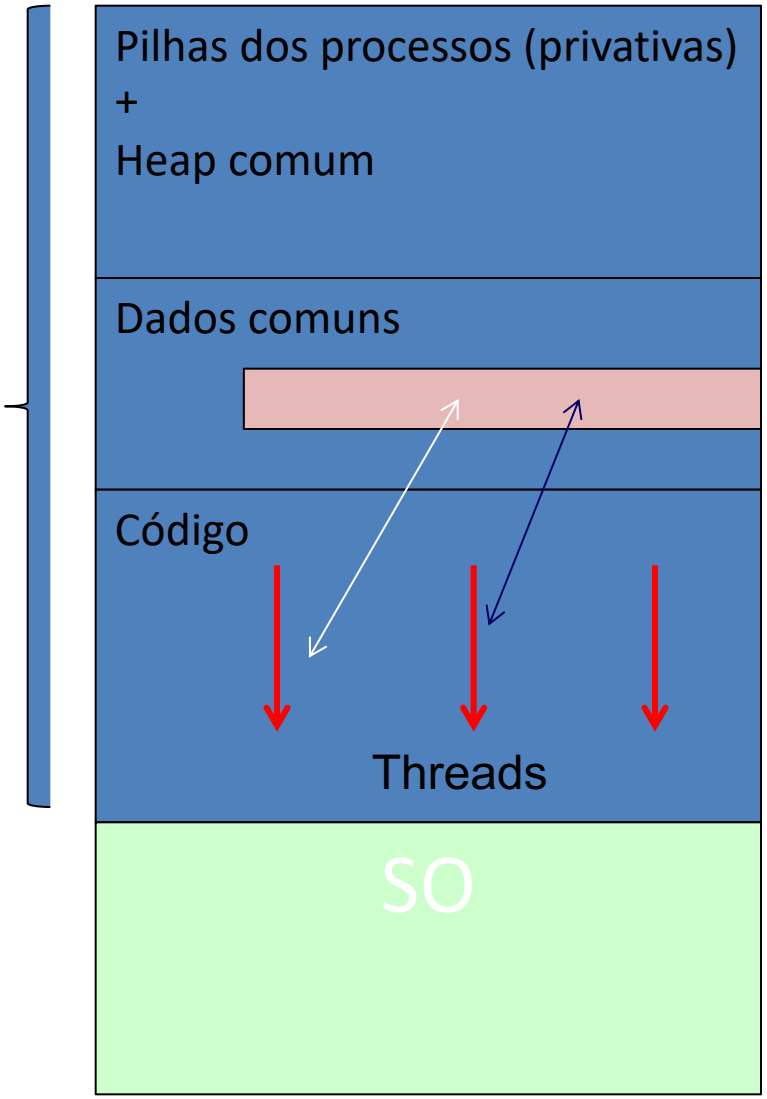
- **Transferir informação entre si:** transferir bytes entre processos. Exemplos: browser / servidor web
- **Sincronizar acções:** Garantir que a acção B no processo P2 só acontece depois da acção A no processo P1. Exemplo: garantir actulização correcta de variáveis comuns (mutexes)

Comunicação entre processos *Interprocess Communication* (IPC)

- Mecanimos que permitem aos processos
  - Sincronizar as suas acções
  - Transferir informação
  - Utilizando memória comum Exemplo: Nos Pthreads, processos que lêm e escrevem variáveis partilhadas
  - Sem partilha de memória
    - Exemplo 1: um processo UNIX criado por fork() não partilha memória com outros processos no sistema
    - Exemplo 2: um processo na máquina H1 e outro que executa na máquina H2

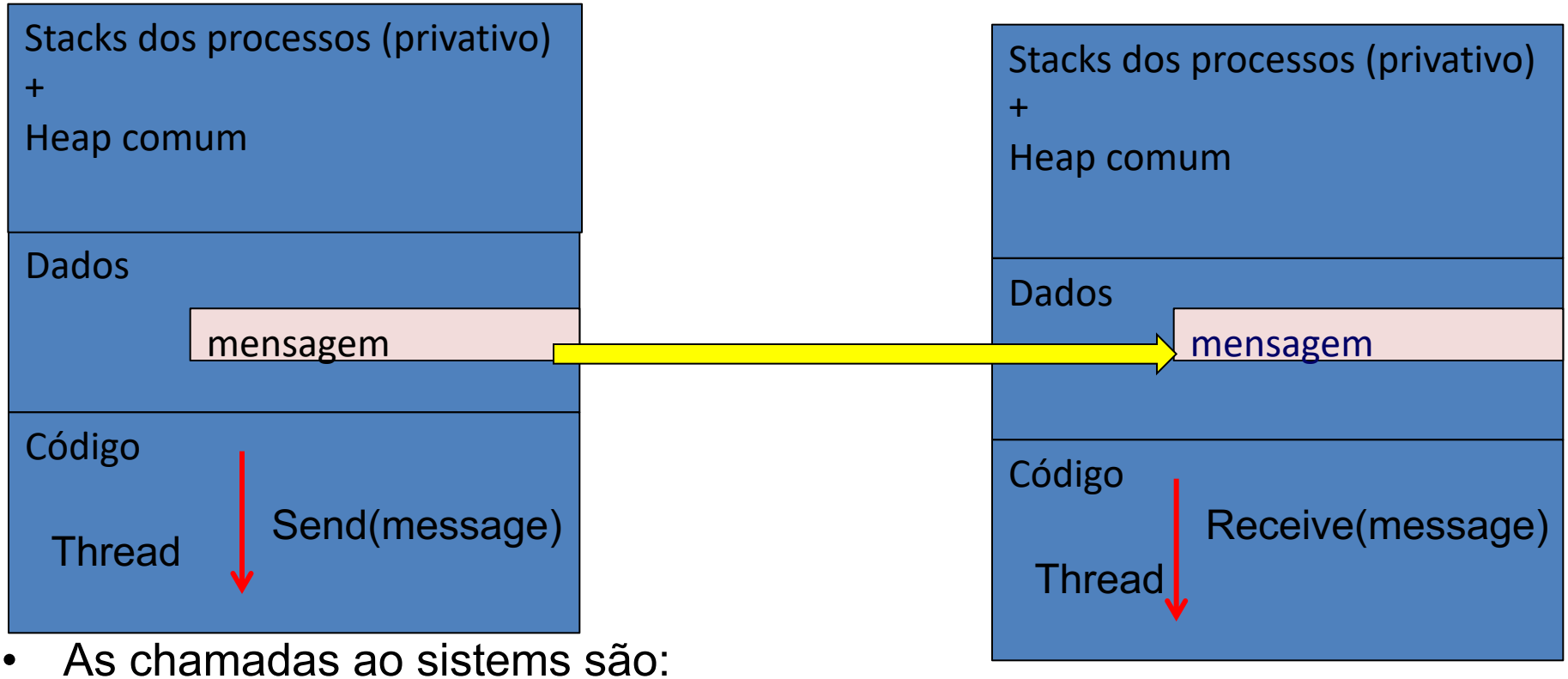
IPC com memória partilhada

Imagem do processo em RAM



- Threads comunicando através da leitura e escrita em variáveis
- Não é precisa a intervenção do SO (rápido)
- Ler e escrever em variáveis comuns tem de ser feita com regras (perigoso)

IPC sem memória partilhada: envio e recepção de mensagens

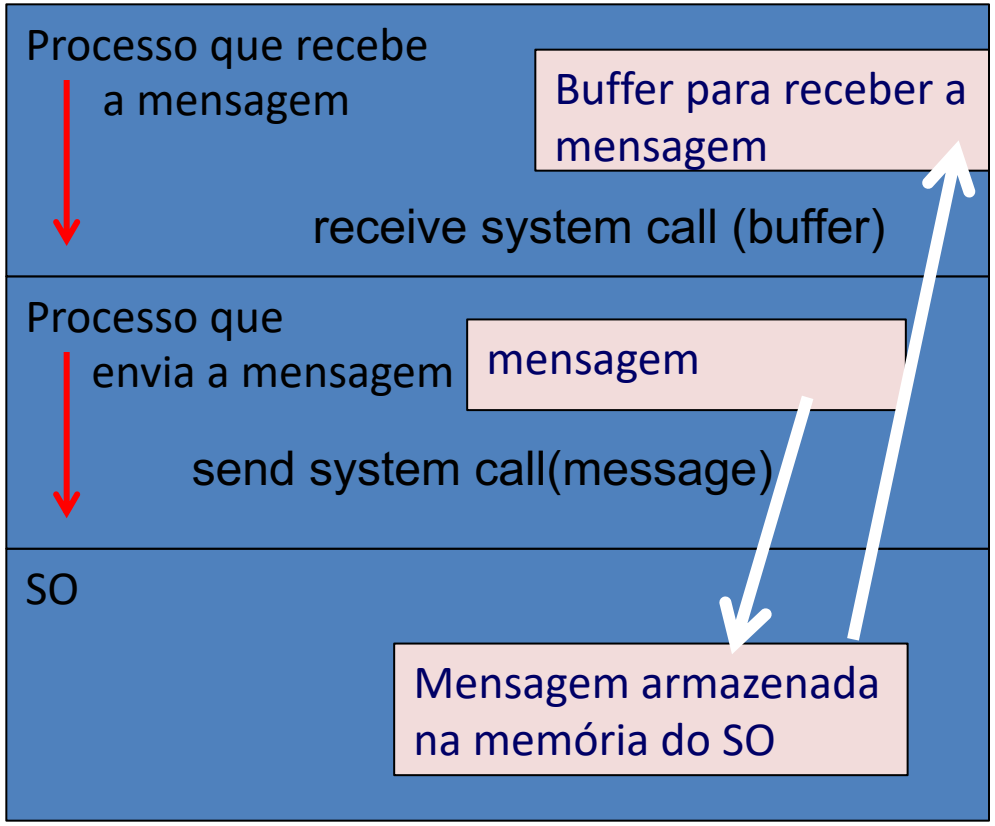


- As chamadas ao sistems são:
  - **send**(destino, *sequência de bytes*) – um conjunto de bytes que estão na memória do processo emissor é enviado para um destino
  - **receive**(*espaço em memória*) –bytes enviados são copiados para um espaço previamente reservado na memória do processo receptor.

Como especificar o destinatário

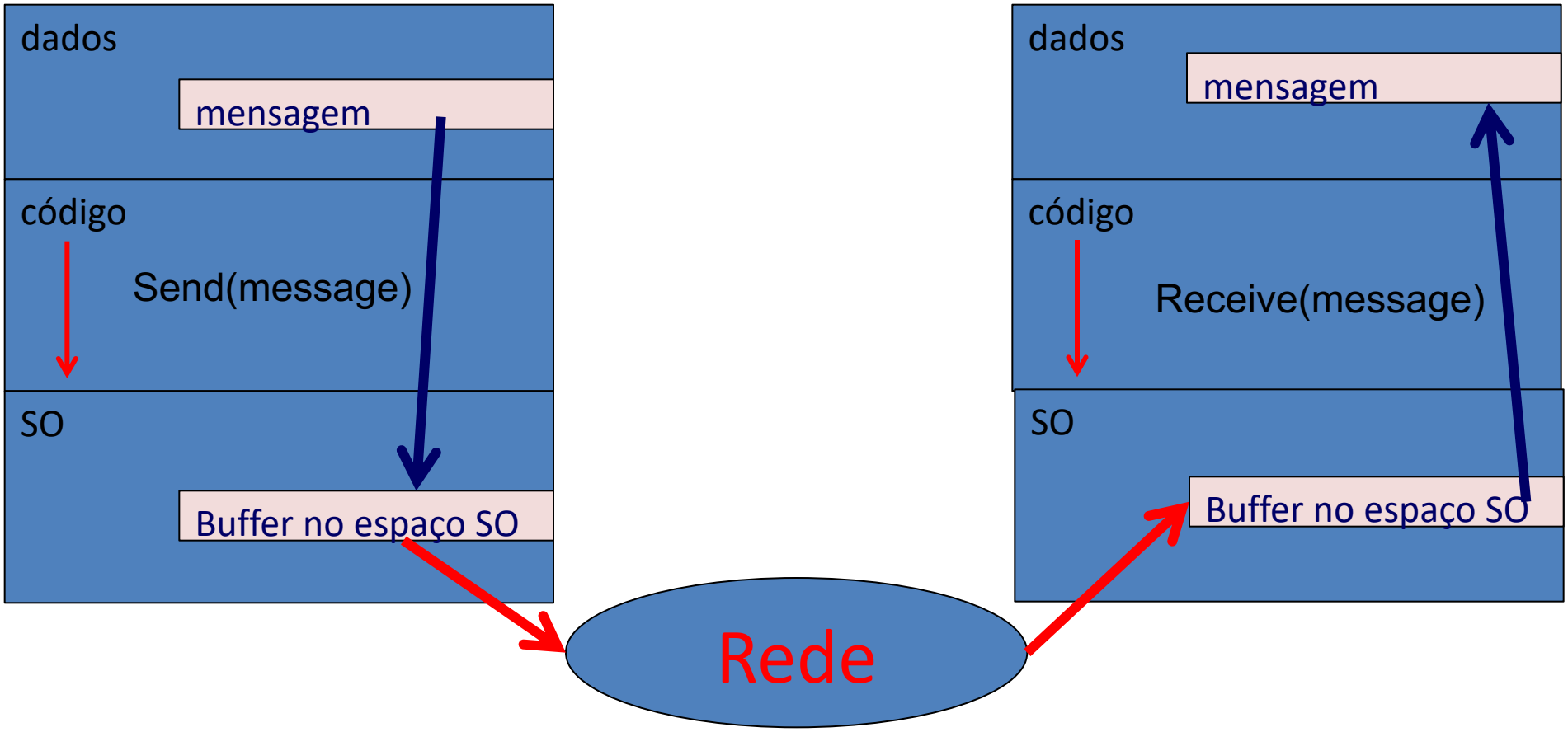
- Comunicação directa – o receptor da mensagem é especificado pelo identificador do processo pid
  - Os processos designam-se explicitamente:
    - **send** (*P, message*) – enviar uma mensagem ao processo P
    - **receive**(*Q, message*) – receber uma mensagem do processo Q
- Comunicação indirecta – As mensagens são enviadas para entidades externas aos processos(mailboxes or ports)
  - Cada mailbox / port tem um identificador distinto
    - **send**(*A, message*) – enviar mensagem para a mailbox /port A
    - **receive**(*B, message*) – receber mensagem da mailbox / port B

IPC sem memória partilhada precisa de suporte do SO



- Na mesma máquina, o SO tem de transferir os bytes, uma vez que um processo não pode ler / escrever na memória de outro processo

Comunicação entre processos em máquinas distintas



- O SO pode enviar e receber bytes através da rede
- O emissor tem de especificar: o endereço da máquina onde os bytes são entregues, e uma port / mailbox na máquina remota

Sincronização associada ao envio / recepção de mensagens

- **Recepção de mensagens** é usualmente **bloqueante**
  - O receptor espera até que haja uma mensagem disponível. O SO coloca o processo no estado Bloqueado (Waiting) até que a mensagem chegue
- **Envio de mensagens** é usualmente **não bloqueante**
  - O emissor copia os bytes a enviar para a memória do SO e continua

Pipes no UNIX (OSTEP Ch4, Sec 5)

- Redirecção de Input / Output

  wc –l xpto.c > count.txt redirige o output de *wc* para o ficheiro *count.txt*

- **Combinação de comandos**
  - **ls –al | sort** o output do 1º comando é o input of segundo comando

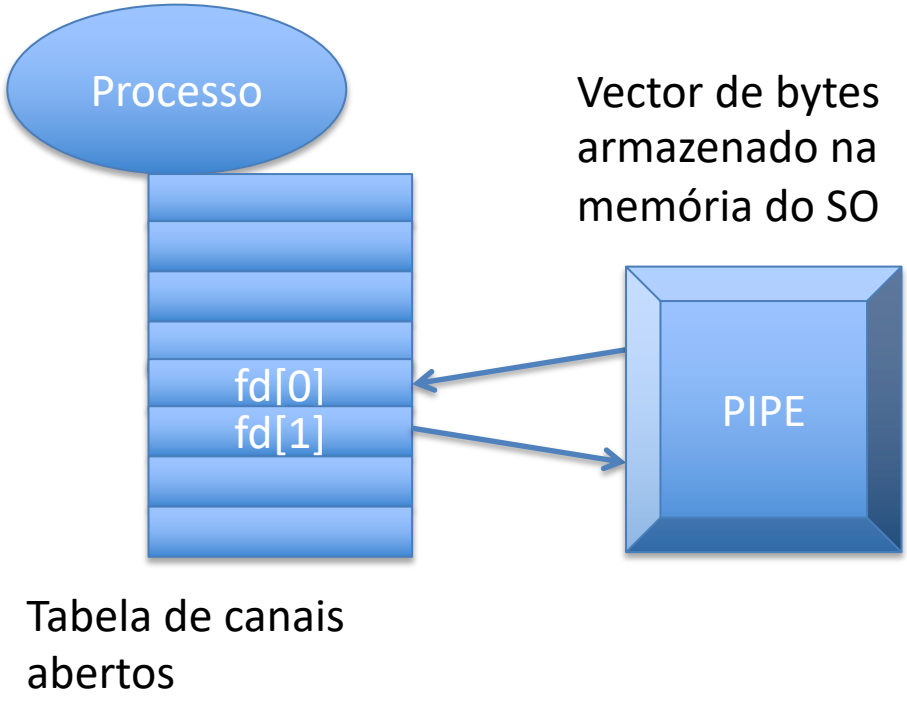
Chamada ao sistema *pipe*

int pipe(int fd[2]) cria um par de canais

- fd[0] for reading
- fd[1] for writing

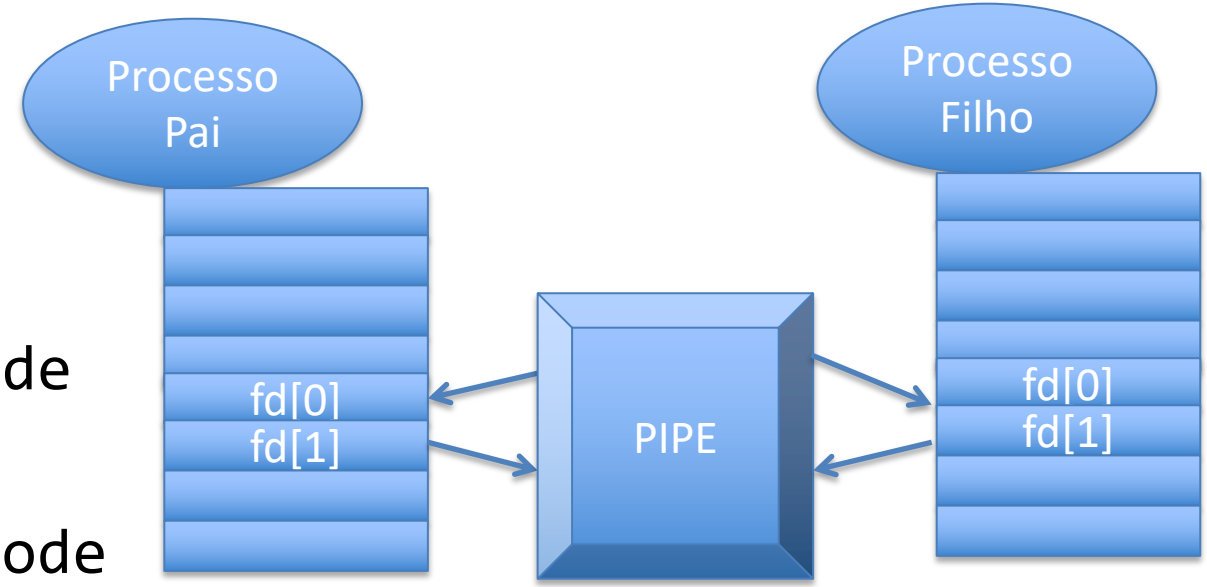
returns 0 on success, -1 on error

```
#include <stdio.h>
#include <unistd.h>
#include<sys/types.h>
main(){
   int fd[2];
   pipe(fd);
      . . .
```



Pipes e fork( )

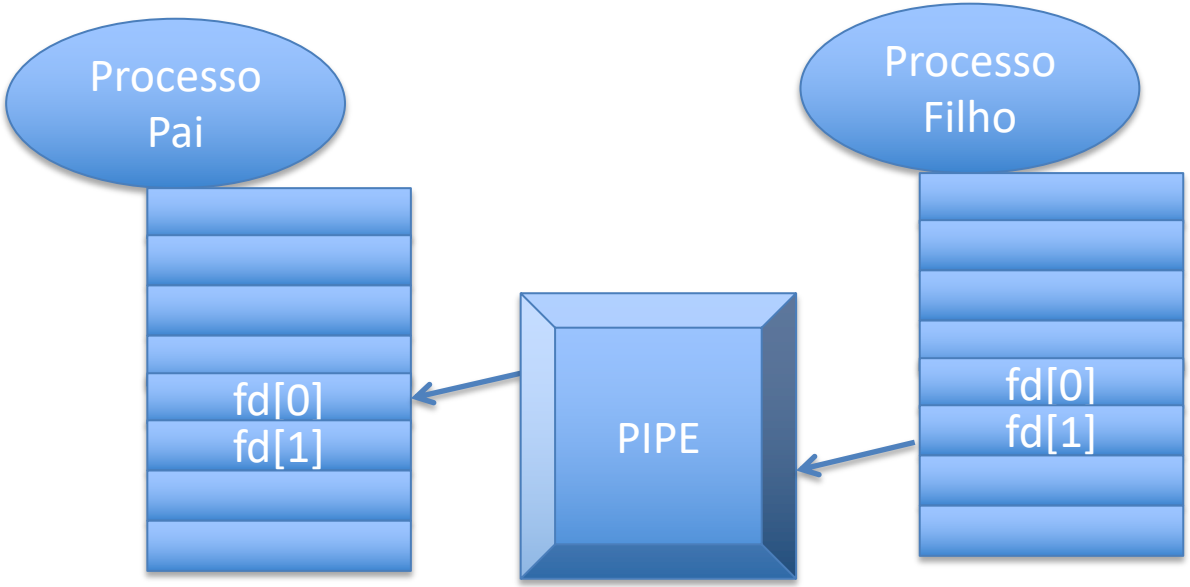
```
#include <stdio.h>
#include <unistd.h>
#include<sys/types.h>
main(){
   int fd[2];
   pipe(fd);
      p = fork();
      if( p==0){
         // child code
      } else{
         // parent code
      }
```



Pai e filho comunicando através de um pipe [1]

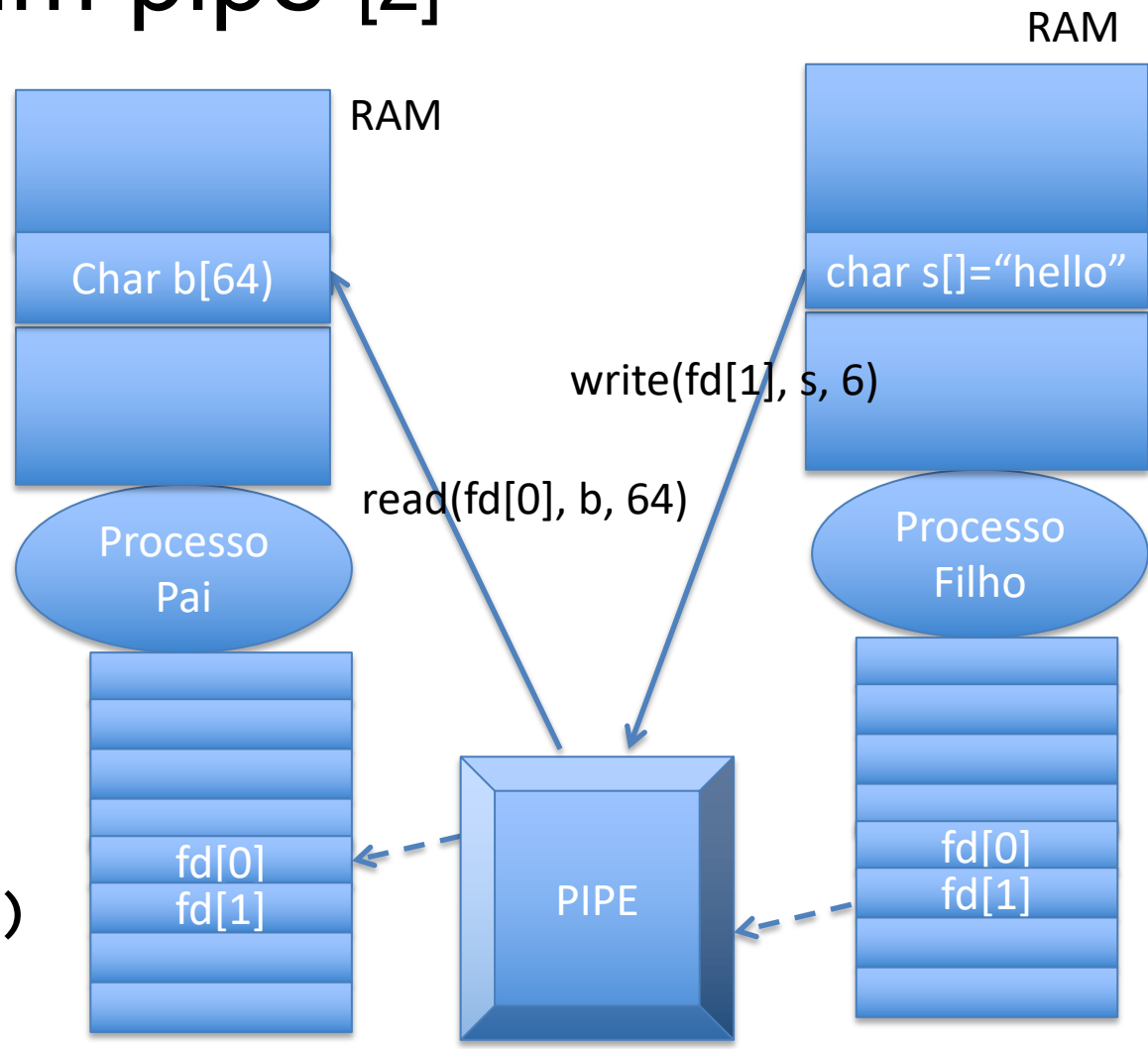
```
.. .
main(){
    int fd[2];
    pipe(fd);
      p = fork();
      if( p==0){
         // child code
         close(fd[0]);
        // child closes
        //input channel
         ...
      } else{
         // parent code
         close(fd[1]);
       // parent closes
       //output channel

      }
```



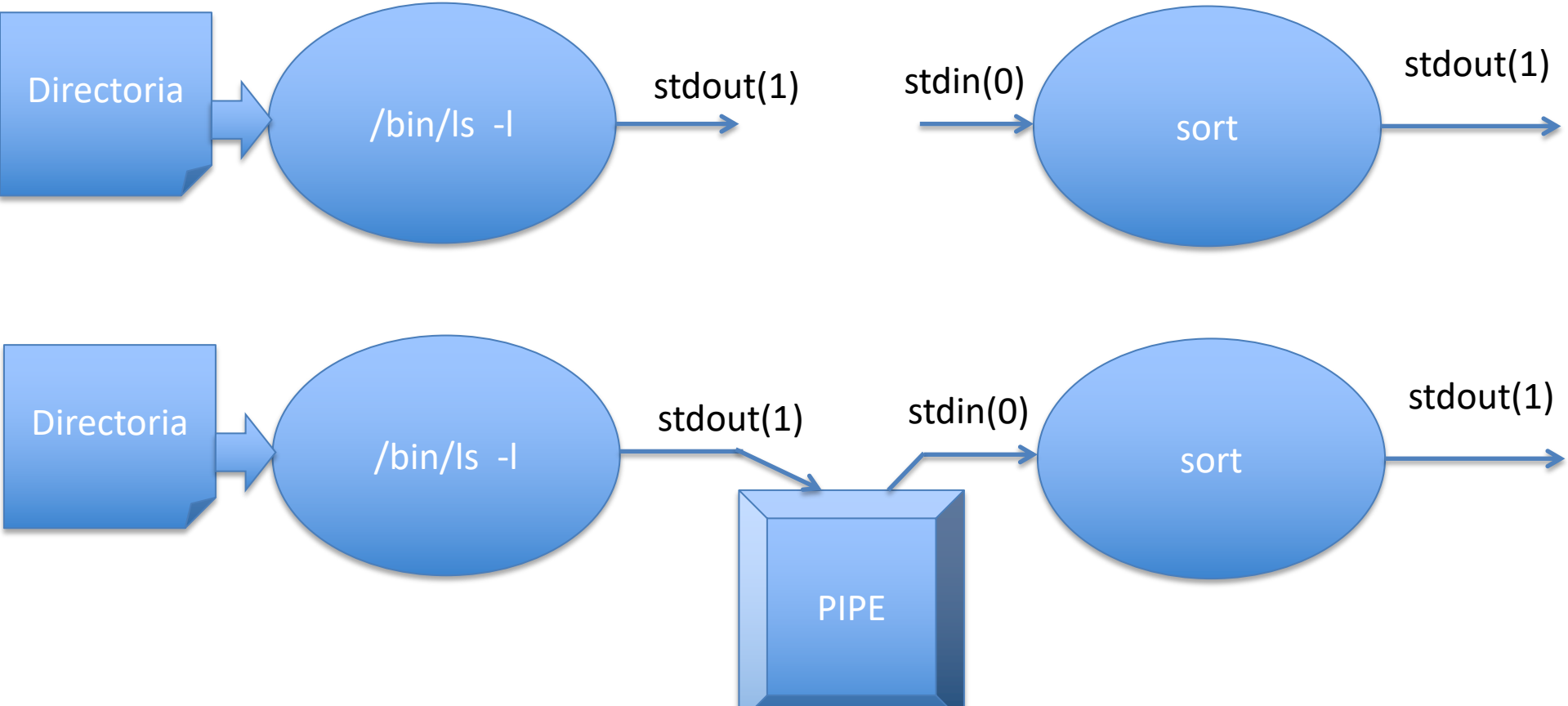
Pai e filho comunicando através de um pipe [2]

```
main(){
      char s[]="Hello\n";
      char b[64];
   . . .
     if( p==0){
        // child code
        close(fd[0]);
        write( fd[1], s,
          strlen(s)+1);
     } else{
        // parent code
        close(fd[1]);
        read( fd[0],b, 64)
     }
```



Uso de *Pipes* para combinar processos

- **Combinação de comandos**
  - **ls –al | sort** o output do 1º comando é o input of segundo comando



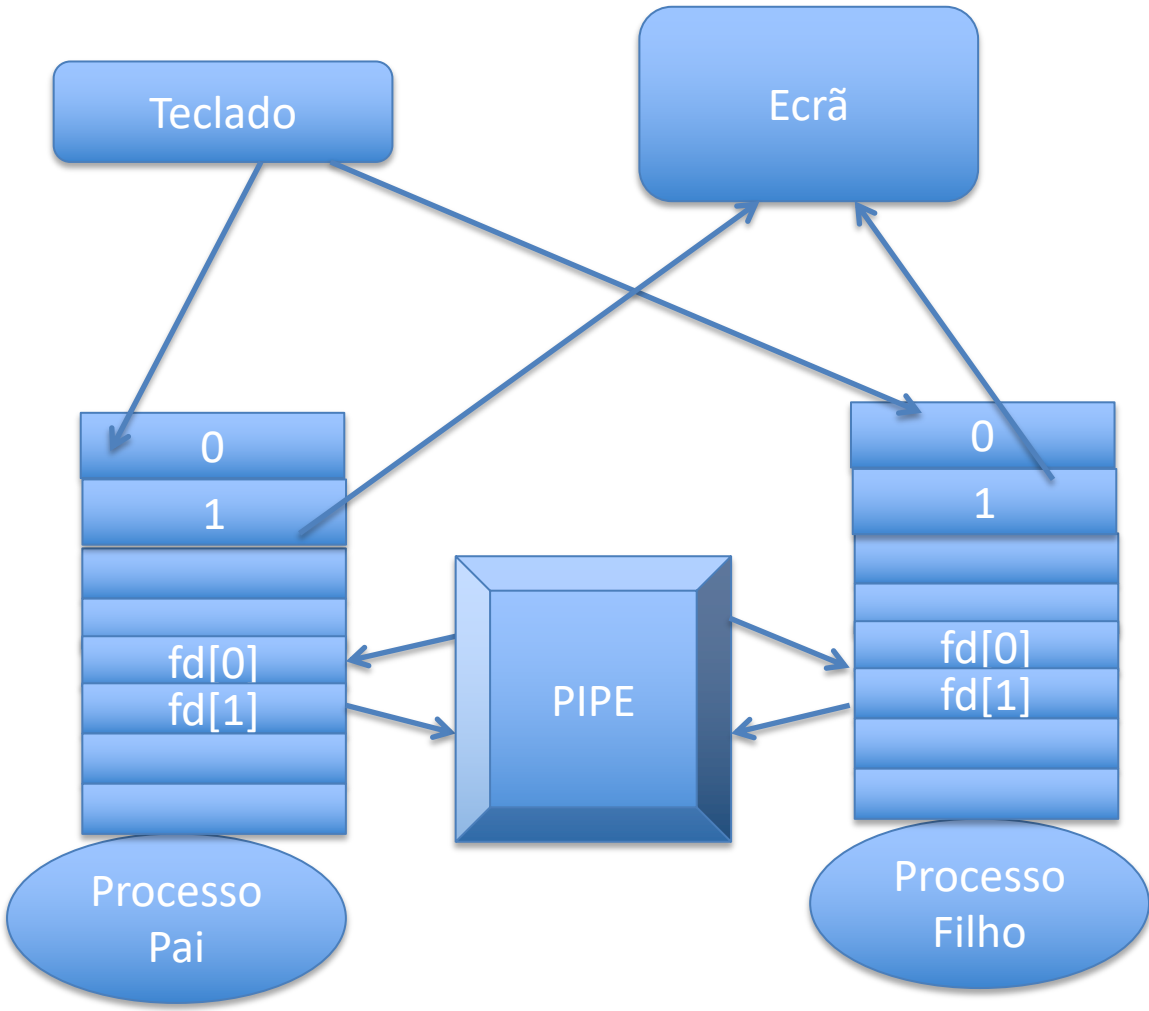
Chamada ao sistema Dup

int dup(int existingFd) duplica o canal *existingFd* retornando o nº do novo canal; retorna -1 se há erro. O novo canal tem a menor entrada livre na tabela de canais

```
int fd[2];
pipe(fd);
p = fork(); // o pipe existe no pai e no filho
if(p = 0)
  close(0); // fecha stdin
  dup(fd[0]); // parte de input do pipe
  execvp( ... ); // herda os canais abertos
```

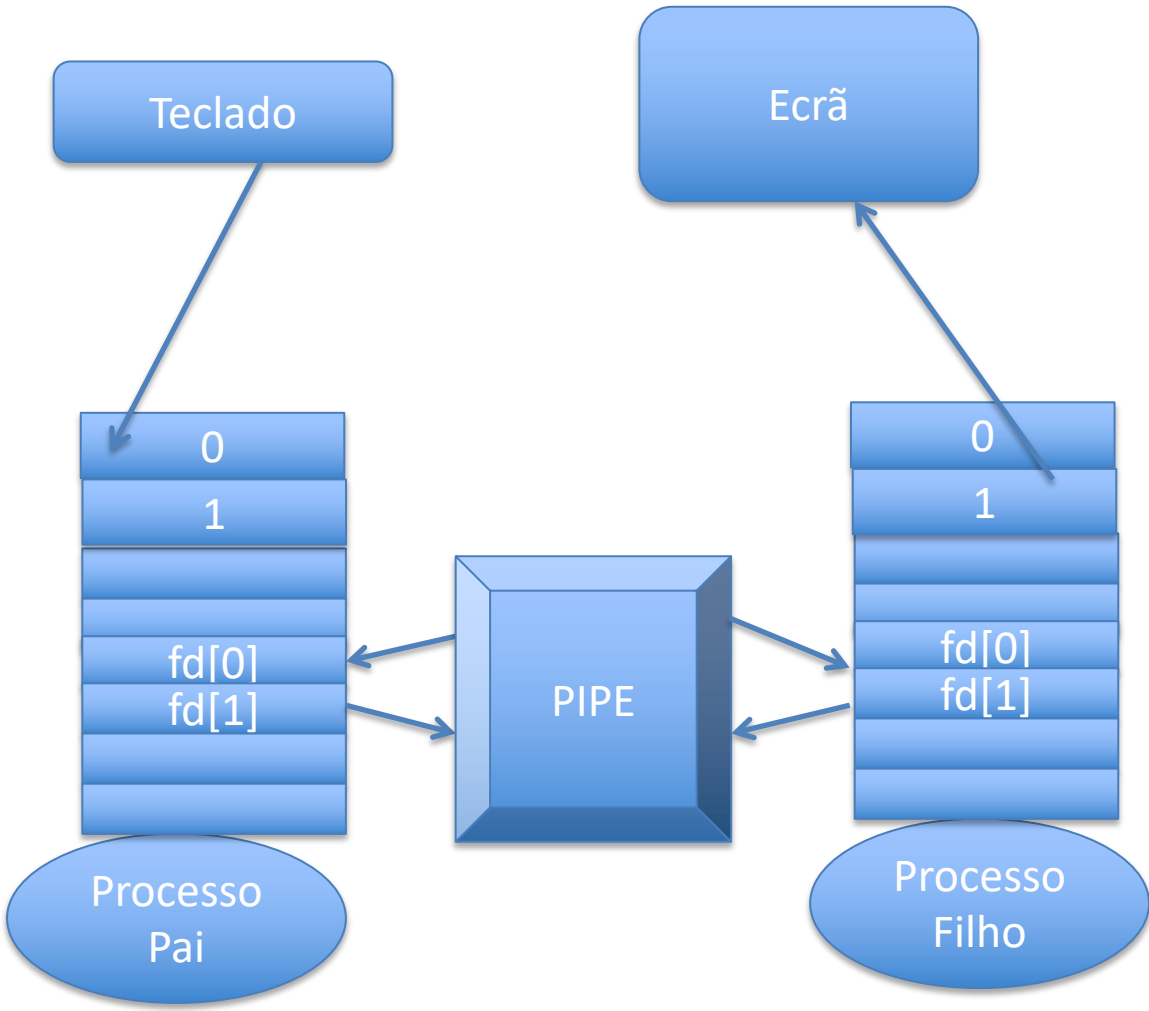
Dup system call [2]

```
int fd[2];
pipe(fd);
p = fork();
if(p != 0){
          . . .
}else{
          . . .
```



Dup system call [2]

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



Dup system call [2]

```
int fd[2];
pipe(fd);
p = fork();
if(p != 0){
    close(1);
    dup(fd[1]);
          . . .
}else{
     close(0);
     dup(fd[0]);
          . . .
```