NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 6

- Comunicação entre processos
  - pipe e fifo
- OSTEP: cap. 5
- Silberschatz, Operating Systems Concepts, 10th Ed. 3.7.4

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

2

2

Comunicação entre processos

InterProcess Communication (IPC)

- Mecanismos que permitem aos processos
  - Sincronizar as suas ações
  - Transferir informação
- Dois suportes:
  - Utilizando memória comum
    - Exemplo: processos que leem e escrevem variáveis partilhadas (será possível? como?)
  - Sem partilha de memória
    - Os processos têm memória privada distinta
    - Os processos podem estar em computadores diferentes
    - A informação tem de ser copiada entre processos
      - Como? Por quem? → SO

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

3

Comunicação entre processos

InterProcess Communication (IPC)

- Dois modelos de comunicação:
  - Troca de **mensagens**
    - Unidades indivisíveis de dados
    - Operações típicas: send / receive
  - ***Streams*** de bytes / canais de IO
    - Sequências de bytes (sem divisões)
    - Cada processo envia e recebe em blocos da dimensão que quiser (possivelmente diferentes)
    - Operações típicas: write / read

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

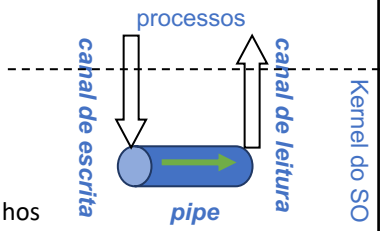
4

Comunicação usando canais de I/O

- Usando ficheiros não é fácil ter comunicação entre **processos concorrentes**
  - Este método é pouco eficiente: envolve escrita e leitura de discos...
  - Não é fácil sincronizar escritor com leitor
- Mas a abstração de canal e ler/escrever sequências ordenadas de bytes é simples e conveniente

Pipes e *fifos* no Unix

- Mecanismos de comunicação geridos pelo SO, oferecendo a noção de *stream*/canal entre processos
- Acedido por descritores, como nos ficheiros
  - operações read/write
  - a sincronização é garantida
- Os pipes podem ser anónimos:
  - Não têm um '*pathname*' associado
  - Só partilháveis entre processos pai e filhos

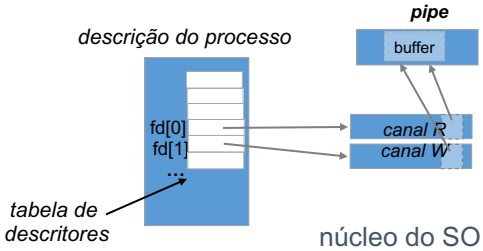


- Ou ter nome no sistema de ficheiros (chamados *fifos* ou *named pipes*)
  - acessíveis com open()

Criação de pipe anónimo

`int pipe( int fd[2] )`

- Cria um pipe e o SO devolve dois canais em **`fd`**
  - lê-se do de leitura: **`fd[0]`**
  - escreve-se no de escrita: **`fd[1]`**
- O SO mantém a ordem da sequência de bytes em trânsito
  - FIFO (*stream*)
- O SO implementa o pipe em memória finita
  - Existe um limite para a capacidade do pipe (min. 512 bytes, normalmente mais)

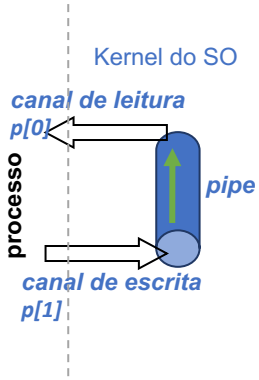


NOVA FACULDADE DE CIÊNCIAS E TECHNOLOGIA DEPARTAMENTO DE INFORMÁTICA

7

Exemplo (num só processo)

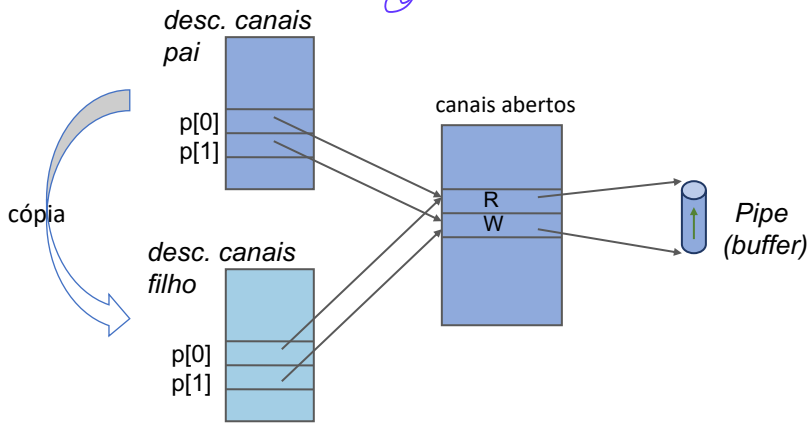
```
int p[2], n;
if (pipe(p)== -1){/*erro*/}
write(p[1], buf1, buf1sz);
write(p[1], buf2, buf2sz);
n=read(p[0], buf, bufsz);
while ( n>0 ) {
   // process buf
   n=read(p[0], buf, n);
}
```



NOVA FACULDADE DE CIÊNCIAS E TECHNOLOGIA DEPARTAMENTO DE INFORMÁTICA

8

Pipe e o fork



Comunicação filho/pai c/ *pipe*

- O pai cria o *pipe*; o filho partilha-o; o SO garante a comunicação sincronizada

```
if (pipe(p)==-1) abort();
switch(fork()){
  case -1: abort();
  case  0: // filho
           write(p[1],"ola",4);
           // ....
           exit(0);
  default: // pai
           read(p[0], buff, SZ);
           // ...
}
```

Funcionamento do pipe

- Comunicação num fluxo de bytes (FIFO e unidirecional em cada instante)
- A semântica dos read e write garante a **sincronização** desejada
  - Os read/write podem bloquear internamente enquanto decorre a comunicação, caso:
  - no write: se o pipe enche
  - no read: se o pipe vaza
- A comunicação só decorre enquanto existem:
  - leitores (quem possa ler/canal de leitura)
  - e escritores (quem possa escrever/canal de escrita)

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

11

Funcionamento do pipe (2)

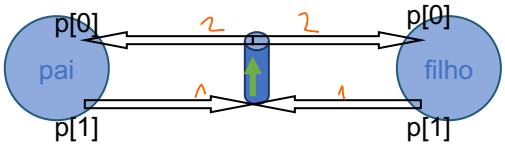
- Como um processo sabe que a comunicação terminou?
- ou como o SO determina que terminou a comunicação?
  - se todos os descritores de escrita forem fechados:
    - o canal de escrita é fechado
    - o read num pipe vazio sem canal para escrita devolve 0 (em vez de bloquear) como se "fim de ficheiro"
  - se todos os descritores de leitura forem fechados:
    - o canal de leitura é fechado
    - o write num pipe sem leitores dá erro (EPIPE)

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

12

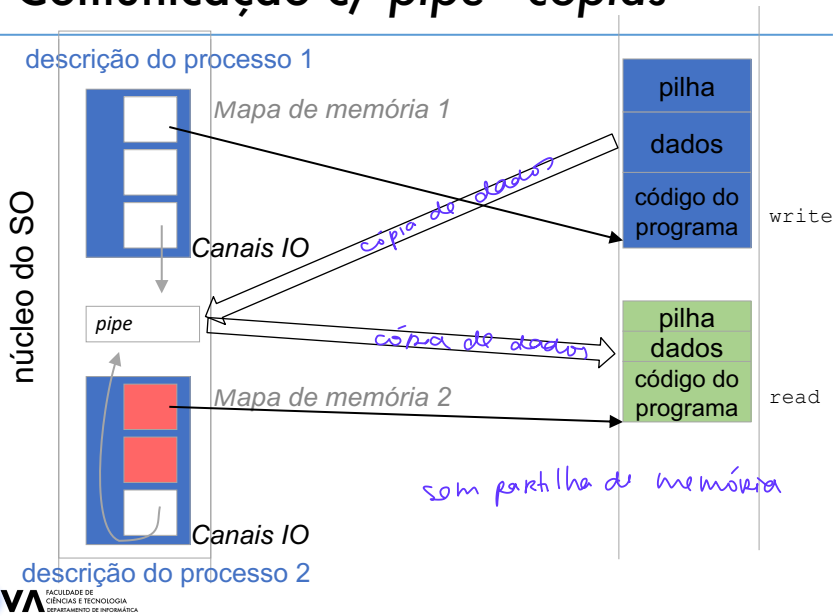
Comunicação pai/filho c/ pipe (2)

exemplo anterior: comunicação de filho para pai



- quem envia não usa o canal de leitura:
  - `close( p[0] )`
- quem recebe não usa o canal de escrita:
  - `close( p[1] )`

Comunicação c/ *pipe* - *copias*



Comportamento dos read e write

- No modelo de *stream* (fluxo de bytes)
  - Não existem "fronteiras" ou "separadores" que indiquem fim de um write
- read() devolve os bytes disponíveis, até ao número pedido
- write() internamente ao SO pode ter de escrever apenas os bytes que couberem até alguém ler para que possa continuar (fragmenta)
- Só escritas de menos bytes do que a capacidade do pipe são atómicas ou indivisíveis

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

15

Redirigir canais para *pipes*

- Lançar um *pipeline* de comandos. Equivalente no *shell* a:

```
ls -l | wc -l
```

- Objectivo: *standard ouput* do ls ligado ao *standard input* do wc → contar o num. de linhas do **`ls`**
  - quando `ls` fizer `write(1,…)`, deve escrever no lado de escrita do *pipe*
  - quando `wc` fizer `read(0,…)`, deve ler do lado de leitura do *pipe* os bytes escritos por `ls`
  - quando o `ls` termina → termina a comunicação

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

16

Redirigir canais para *pipes* (2)

- o *shell* cria um pipe e dois processos. → 2 forks()
  - no 1º proc. redireciona o STDOUT para canal de escrita no *pipe* antes do exec() do ls
  - no 2º proc. redireciona o STDIN para o canal de leitura do *pipe*, antes do exec() do wc
  - Todos os canais não usados têm de ser fechados antes dos exec() *(porquê?)*
- *Como redirigir para canais já abertos?*

Copiando descritores

- Obter descritores que são cópias de canais já abertos:
  - `int dup( int oldfile )`
    obtém novo descritor pela mesma ordem que *open* e *creat*
  - `int dup2(int oldfile, int newfile)`
    fecha *newfile* e depois duplica *oldfile* para *newfile*

duplicar

Duplicar descritores de canais

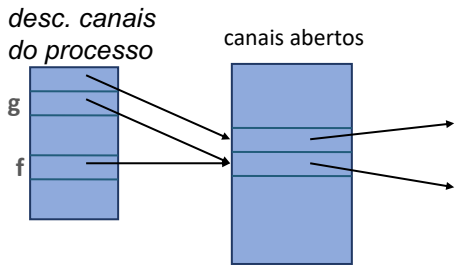
- Exemplos:

  `g = dup( f );`

  `g` é um novo descritor para o mesmo canal que `f`

  `g = dup2(f, 1);`

  fecha o `1` (STDOUT) e coloca nesse descritor uma cópia de `f` (ou seja, o canal em `f` passa a ser também o STDOUT)



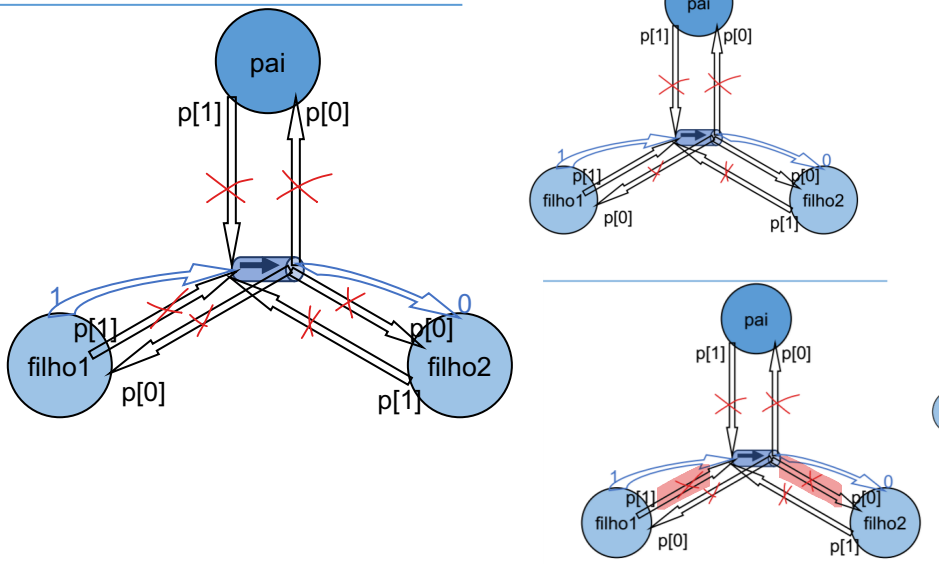
Programando: `ls -l | wc -l`

```
if (pipe(p)==-1) abort();
switch(fork())
{ case -1: abort();
  case  0: filho1(p);
           exit(1);
  default:
    switch(fork())
    {  case -1: abort();
    case  0: filho2(p);
             exit(1);
    default:
    }
    close(p[0]);
    close(p[1]);
    wait(NULL);
    wait(NULL);
}
```

```
void filho1( int p[] )
{
  dup2(p[1],1);
  close(p[0]);
  close(p[1]);
  execlp("ls","ls","-l",NULL);
}

void filho2( int p[] )
{
  dup2(p[0],0);
  close(p[1]);
  close(p[0]);
  execlp("wc","wc","-l",NULL);
}
```

filho1 | filho2



→ foram fechados pois já existem as suas cópias para o pipe

Pipe com nome (fifo)

- *FIFO* ou *named pipe*:
  - um *pipe* com um nome no sistema de ficheiros
  - criado com **`mkfifo(name, mode)`** (ou **`mknod()`**)
  - acedido com **`open()`**
- no *shell* temos os comandos `mkfifo/mknod`
- exemplo:

```
mkfifo /tmp/canal
wc -l /tmp/canal &
ls -l > /tmp/canal
```

- Diferenças do uso de um ficheiro temporário?