# Comunicação baseada em canais de I/O e *Pipelines* no sistema Unix

**José C. Cunha, DI-FCT/UNL**

# Mecanismos de comunicação…

- **Ficheiros em disco (geridos pelo SO)**
- ***´´Pipes´´ (buffers geridos pelo SO)***

- **Mensagens (filas geridas pelo SO)**
- **Memória partilhada (acesso pelo *bus*)**

- **Mecanismos de troca de sinais (sincronização)**

# Redirecção dos canais de I/O

- **Utilizar os canais standard 0 e 1**
- **permite que os programas não tenham de ser alterados quando se redirigem os canais para outros ficheiros**

**Utilizar os canais standard 0 e 1**

**permite que os programas não tenham de ser alterados quando se redirigem os canais para outros ficheiros:**

**Se os programas lerem e escreverem com base em:**

**read(0, …) -- ler do STDIN**

**e**

**write(1,…) -- escrever no STDOUT**

# Redirecção de canais de I/O

- **Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**
  - **Ligando a ficheiros, a periféricos**

# Redirecção de canais de I/O

**Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**

- **Ligando a ficheiros, a periféricos**
- **ou a outros processos… através de *pipes***

# Redirecção de canais de I/O

**Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**

**Ligando a ficheiros, a periféricos**

**ou a outros processos… através de *pipes***

**a) Para ficheiros: por exemplo**

**→ Ligar o stdout (1) a um ficheiro ´´f´´ significa:**

# Redirecção de canais de I/O

**Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**

**Ligando a ficheiros, a periféricos**

**ou a outros processos… através de *pipes***

**a) Para ficheiros: por exemplo**

**→ Ligar o stdout (1) a um ficheiro ´´f´´ significa:**

**close: fechar o canal corrente 1**

# Redirecção de canais de I/O

**Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**

**Ligando a ficheiros, a periféricos**

**ou a outros processos… através de *pipes***

**a) Para ficheiros: por exemplo**

**→ Ligar o stdout (1) a um ficheiro ´´f´´ significa:**

**close: fechar o canal corrente 1**

**open: abrir um canal para o ficheiro ´´f´´**

# Redirecção de canais de I/O

**Ligar os canais standard para configurar as ligações de entrada e saída de cada processo:**

**Ligando a ficheiros, a periféricos**

**ou a outros processos… através de *pipes***

**a) Para ficheiros: por exemplo**

**→ Ligar o stdout (1) a um ficheiro ´´f´´ significa:**

**close: fechar o canal corrente 1**

**open: abrir um canal para o ficheiro ´´f´´**

**(se o canal 0 se mantiver aberto)**

# Semântica do ´´open´´ no Unix

- **Procura a primeira entrada livre na Tab. Canais do Processo e reserva-a para guardar o ´´file descriptor´´ do canal a abrir**

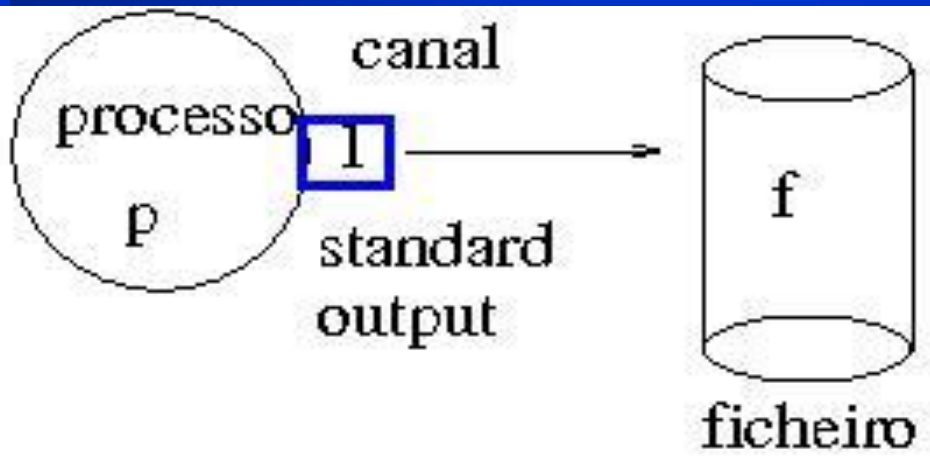
# Semântica do ´´open´´ no Unix

- **Procura a primeira entrada livre na Tab. Canais do Processo e reserva-a para guardar o ´´file descriptor´´ do canal a abrir**
- **Devolve assim o menor número de entrada livre da tabela (*file descriptor*) encontrado.**

# Semântica do ´´open´´ no Unix

- **Procura a primeira entrada livre na Tab. Canais do Processo e reserva-a para guardar o ´´file descriptor´´ do canal a abrir**
- **Devolve assim o menor número de entrada livre da tabela (*file descriptor*) encontrado.**
- **As 3 primeiras entradas da tabela (0, 1 e 2) representam os canais *standard* (INPUT, OUTPUT, ERROR)**

# Redirecção do STDOUT




linha de comando
$ p > f

ao ser criado, p herda
o canal 1 aberto do pai

pode fechar :

```
close(1);
```

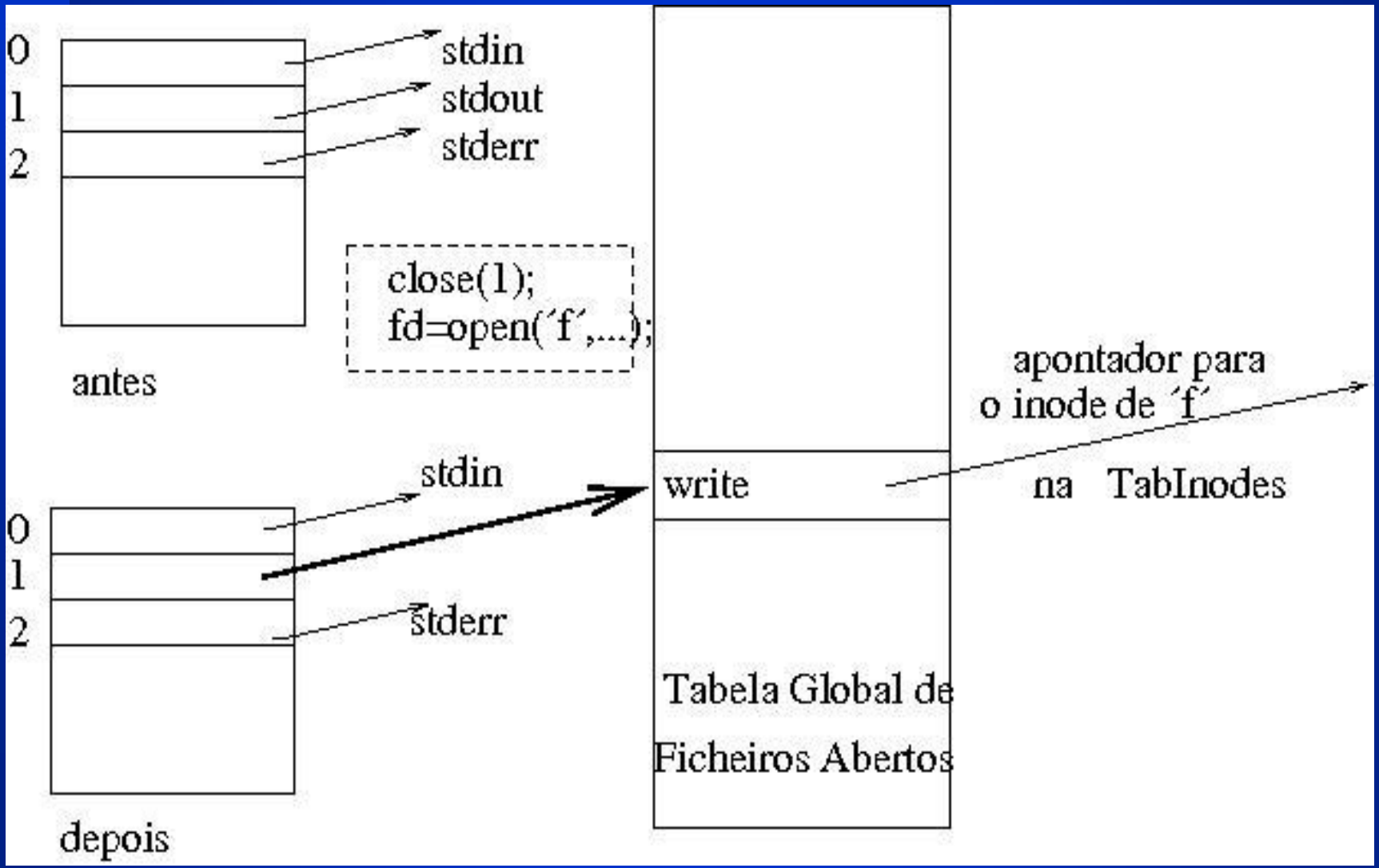
e abrir um canal para f

```
fd = open('f',O_WRONLY);
```

... o valor devolvido de fd = 1

se o canal 0 nao tiver sido fechado.

# Redirecção do STDOUT



# b) Comunicação entre processos?

```
p=fork();
if (p==0) -- processo filho lê
{  fd=open("/tmp/f",O_RDONLY);
   read(fd,buf,count);
}
else  -- processo pai escreve
{  fd=open("/tmp/f",O_WRONLY);
   write(fd,"ola",3);
}
```

- **Será que o filho lê o que o pai escreve?**

# b) Comunicação entre processos?

```
p=fork();
if (p==0) -- processo filho lê
{  fd=open("/tmp/f",O_RDONLY);
   read(fd,buf,count);
}
else  -- processo pai escreve
{  fd=open("/tmp/f",O_WRONLY);
   write(fd,"ola",3);
}
```

- **Será que o filho lê o que o pai escreve?**
- **Só se garantir que o filho só lê depois do pai escrever! → sincronização**

# Comunicando por ficheiro: trocando a ordem... ???

```
p=fork();
if (p==0)   -- processo filho escreve
{  fd=open("/tmp/f",O_WRONLY);
   write(fd,"ola",3);
   exit(0);
}
else    -- processo pai lê, no fim...
{  wait(&status); //ponto de sincronização
   fd=open("/tmp/f",O_RDONLY);
   read(fd,buf, BUFSZ);
}
```

# Sincronização

- **Sincronização: AGUARDAR/ASSINALAR**
  - **um processo aguarda que outro processo atinja um certo ponto na sua execução (para enviar dados ou mensagens)**
  - **um processo assinala a ocorrência de algum evento a um outro processo**

# Sincronização

- **Sincronização: AGUARDAR/ASSINALAR**
  - **um processo aguarda que outro processo atinja um certo ponto na sua execução (para enviar dados ou mensagens)**
  - **um processo assinala a ocorrência de algum evento a um outro processo**
- **Comunicação: TRANSFERIR INFORMAÇÃO**
  - **múltiplos processos concorrentes comunicam através de *buffers* em memória**
  - **processos comunicam por mensagens**

Na comunicação por ficheiros, não se consegue ter garantia da coordenação dos acessos concorrentes:

… um processo que vá escrevendo e outro que vá lendo,

Na comunicação por ficheiros, não se consegue ter garantia da coordenação dos acessos concorrentes:

… um processo que vá escrevendo e outro que vá lendo,

não se tem controlo sobre:

Na comunicação por ficheiros, não se consegue ter garantia da coordenação dos acessos concorrentes:

… um processo que vá escrevendo e outro que vá lendo,

não se tem controlo sobre:

-- a ordem de execução dos processos e a ordem relativa das operações de leitura e escrita nos canais de I/O

Na comunicação por ficheiros, não se consegue ter garantia da coordenação dos acessos concorrentes:

… um processo que vá escrevendo e outro que vá lendo,

não se tem controlo sobre:

-- a ordem de execução dos processos e a ordem relativa das operações de leitura e escrita nos canais de I/O

-- o modo como o SO vai actualizando os dados dos buffers de memória para disco…

Na comunicação por ficheiros, não se consegue ter garantia da coordenação dos acessos concorrentes:

… um processo que vá escrevendo e outro que vá lendo,

não se tem controlo sobre:

-- a ordem de execução dos processos e a ordem relativa das operações de leitura e escrita nos canais de I/O

-- o modo como o SO vai actualizando os dados dos buffers de memória para disco…

e portanto não se sabe exactamente o que cada processo vai ler…

# *Pipes* no Unix

- **Mecanismo de comunicação gerido pelo SO**

# *Pipes* no Unix

- **Mecanismo de comunicação gerido pelo SO**
- **Realizados por *buffers* em regiões de memória geridas pelo SO**

# *Pipes* no Unix

- **Mecanismo de comunicação gerido pelo SO**
- **Realizados por *buffers* em regiões de memória geridas pelo SO**
- **Um *pipe* suporta a passagem de fluxos de bytes consecutivos: *streams***
  - **acessíveis por canais e operações concorrentes read/write**

# *Pipes* no Unix

- **Mecanismo de comunicação gerido pelo SO**
- **Realizados por *buffers* em regiões de memória geridas pelo SO**
- **Um *pipe* suporta a passagem de fluxos de bytes consecutivos: *streams***
  - **acessíveis por canais e operações concorrentes read/write**
- **Na versão original, são anónimos:**
  - **Não têm '*pathname*' (não surgem na directoria)**

# *Pipes* no Unix

- **Mecanismo de comunicação gerido pelo SO**
- **Realizados por *buffers* em regiões de memória geridas pelo SO**
- **Um *pipe* suporta a passagem de fluxos de bytes consecutivos: *streams***
  - **acessíveis por canais e operações concorrentes read/write**
- **Na versão original, são anónimos:**
  - **Não têm '*pathname*' (não surgem na directoria)**
  - **Só são acessíveis a processos pais e filhos**

# Comunicação por pipes

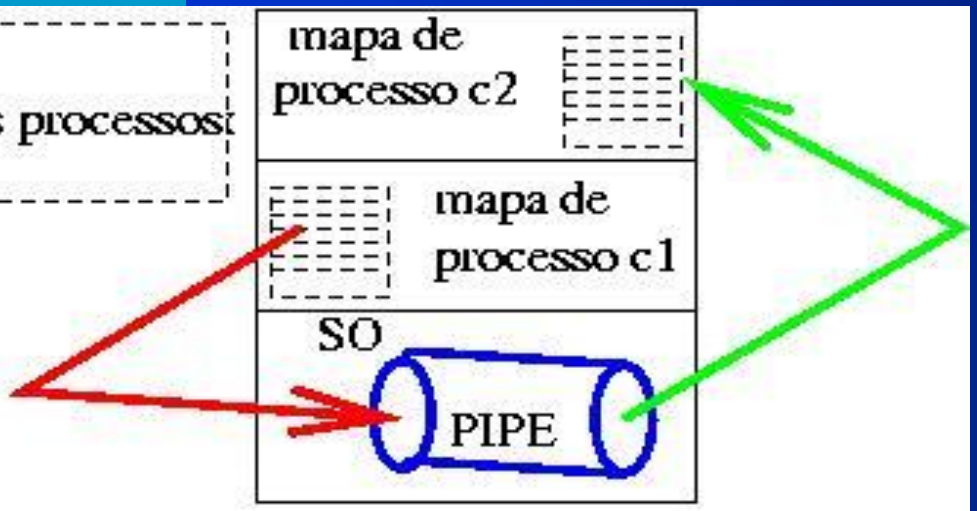
- **Como os pipes não têm um *pathname* que apareça na listagem de ficheiros de uma directoria, não são conhecidos por todos os processos…**

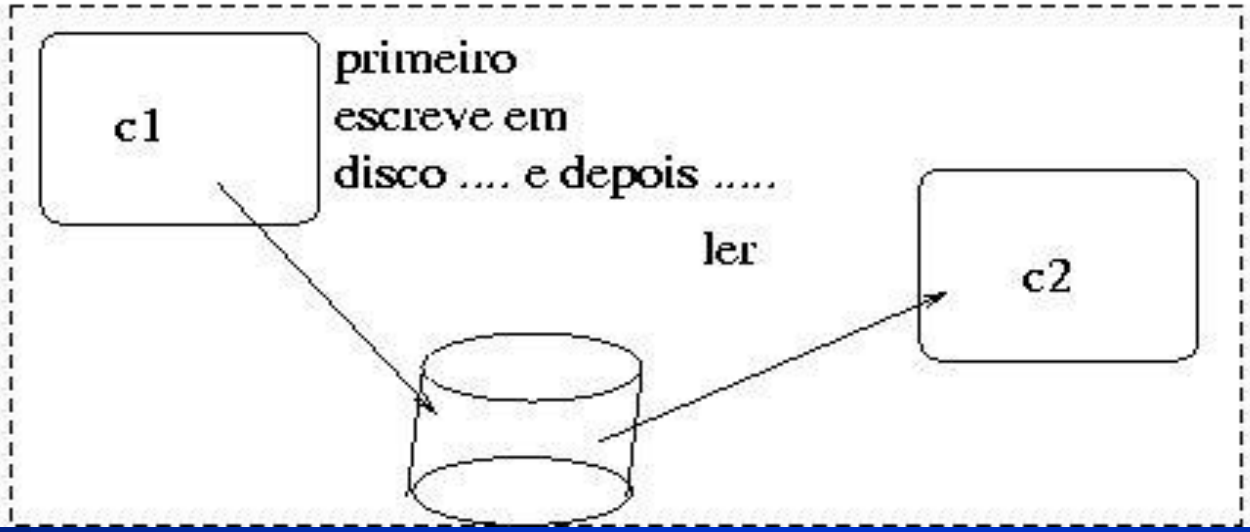
- **Os pipes só podem ser acedidos pelos canais de I/O, que sejam herdados na criação de processos filhos pela chamada fork()**

na linha de comandos do shell:
$ c1 | c2  um pipeline entre dois processos

diferente de :

$ c1 > tempfile
$ c2 < tempfile



int pipe(int fd[2]); chamada ao sistema

cria um
PIPE: canal unidireccional

fluxo de bytes (stream)

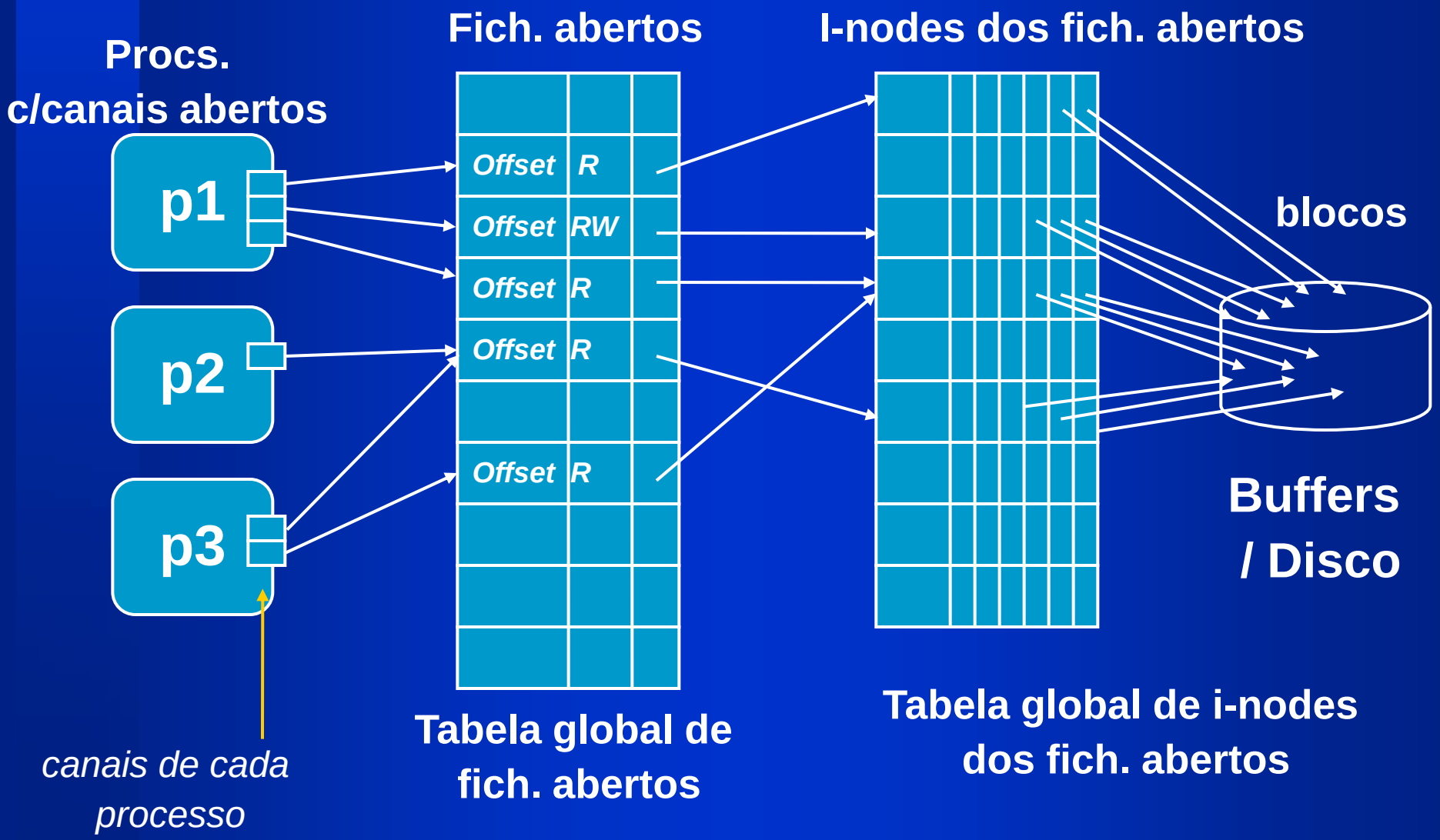
ordem FIFO (first–in first–out)

devolve dois canais abertos para o pipe:

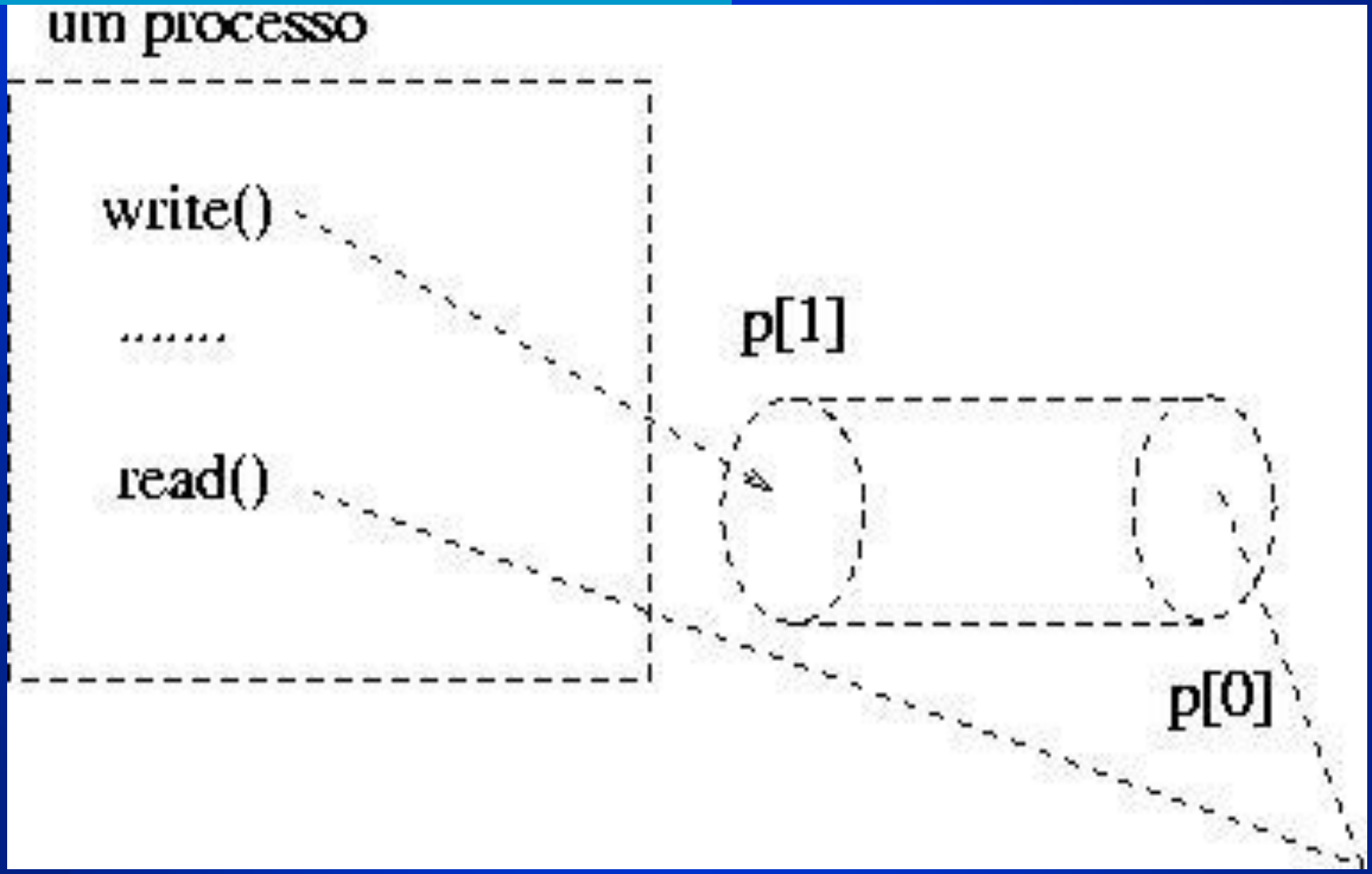
fd[0] para leitura

fd[1] para escrita

# Tabelas em memória



# Pode-se fazer isto, mas…



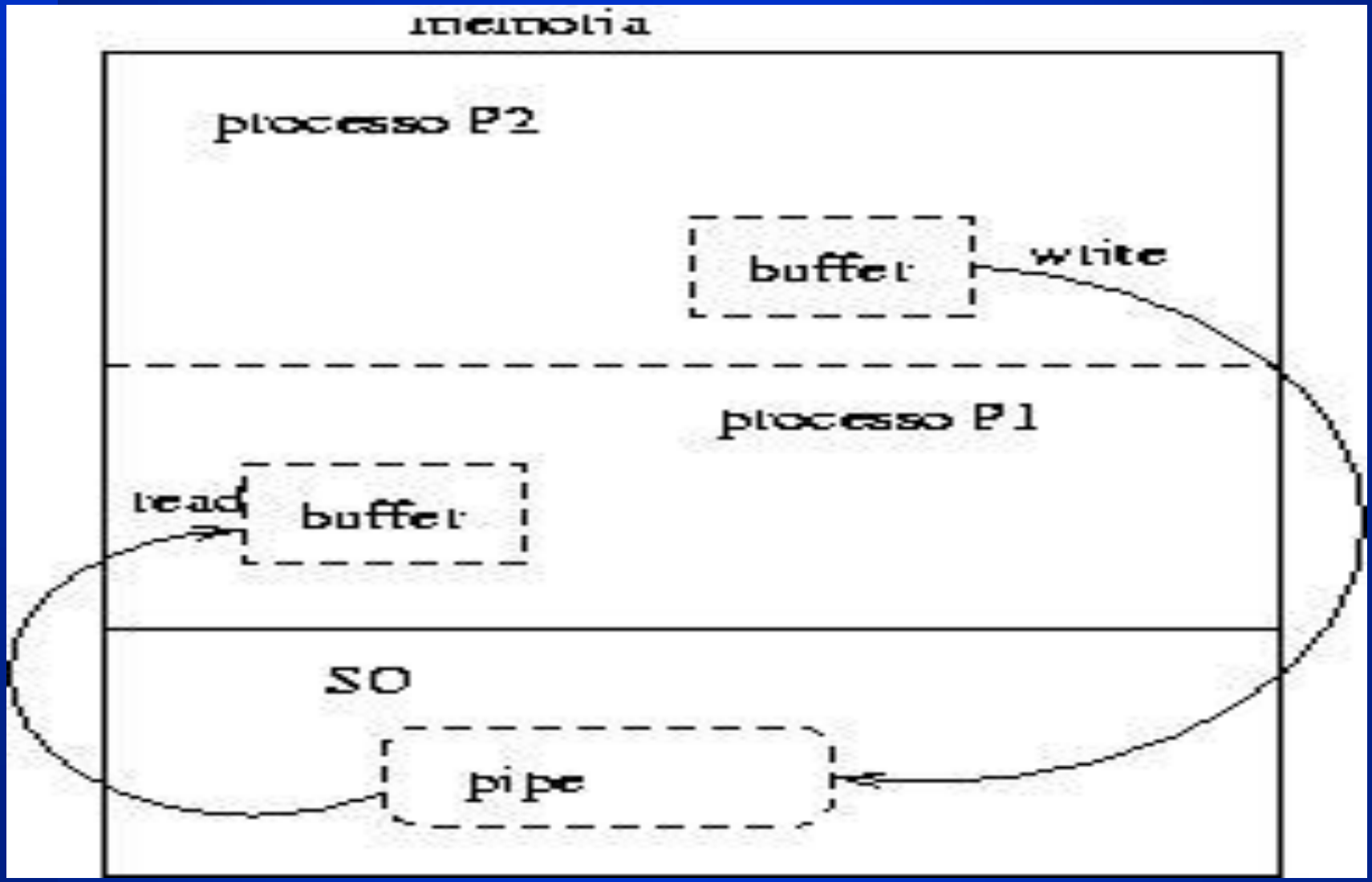
```
int p[2];
 if (pipe(p) == -1) { …erro…}
. . .
```

```
int p[2];
 if (pipe(p) == -1) { …erro…}
 write(p[1], buf1, buf1size);
. . .
```

```
int p[2];
 if (pipe(p) == -1) { …erro…}
 write(p[1], buf1, buf1size);
 write(p[1], buf2, buf2size);
. . .
```

```
int p[2];
 if (pipe(p) == -1) { …erro…}
 write(p[1], buf1, buf1size);
 write(p[1], buf2, buf2size);
. . .
 read(p[0], buf, n);
. . .
```

# Dois processos comunicantes



# Comunicação por pipes

- **Como os pipes não têm um *pathname* que apareça na listagem de ficheiros de uma directoria, não são conhecidos por todos os processos…**

- **Os pipes só podem ser acedidos pelos canais de I/O, que sejam herdados na criação de processos filhos pela chamada fork()**

# Comunicação pai/filho via *pipe*

- **O pai cria o *pipe*; o filho partilha-o; o SO garante a sincronização:**

```
if (pipe(p)==-1) abort();
```

# Comunicação pai/filho via *pipe*

- **O pai cria o *pipe*; o filho partilha-o; o SO garante a sincronização:**

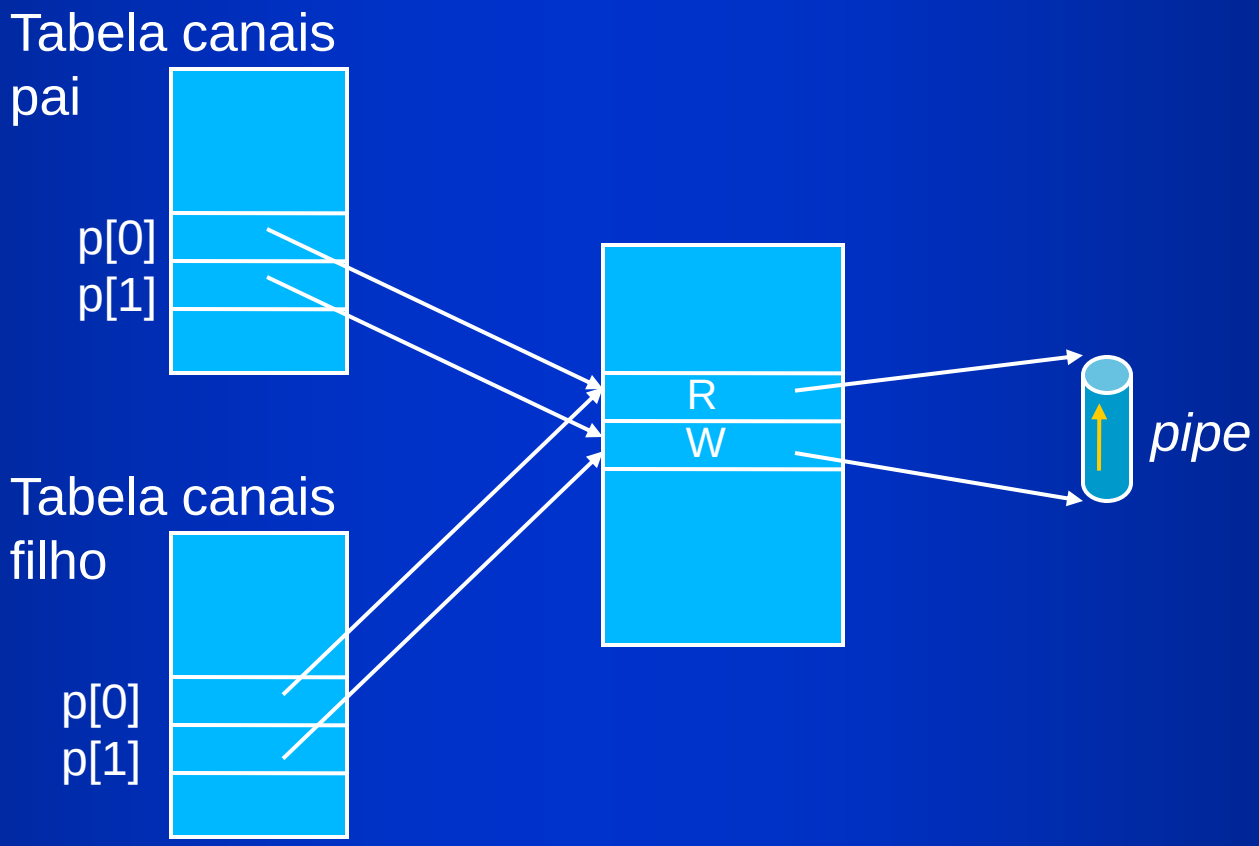
```
if (pipe(p)==-1) abort();
switch(fork())
{ case -1: abort();
  case  0: // filho
           read(p[0], bufin, n);
           ...
  default:
}
```

# Comunicação pai/filho via *pipe*

- **O pai cria o *pipe*; o filho partilha-o; o SO garante a sincronização:**

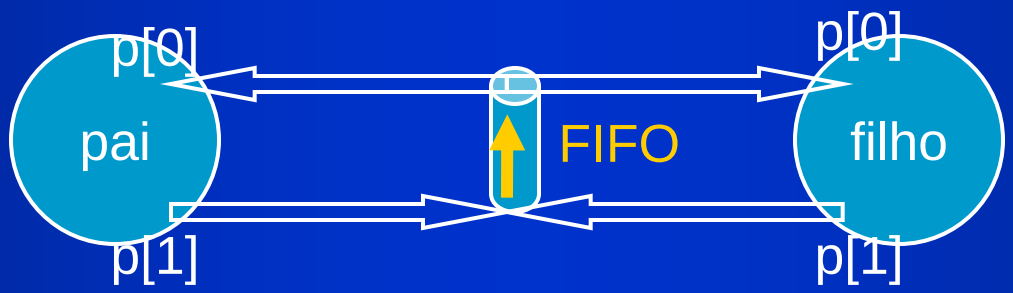
```
if (pipe(p)==-1) abort();
switch(fork())
{ case -1: abort();
  case  0: // filho
           read(p[0], bufin, n);
           ...
  default: // pai
           write(p[1],"ola",3); ...
}
```

# Comunicação pai/filho via *pipe*



# Comunicação pai/filho c/ pipe

exemplo anterior: pai envia para filho




- **escritor não usa o canal de leitura:**
  - **close( p[0] ) no pai**
- **leitor não usa o canal de escrita:**
  - **close( p[1] ) no filho**

# Exemplo

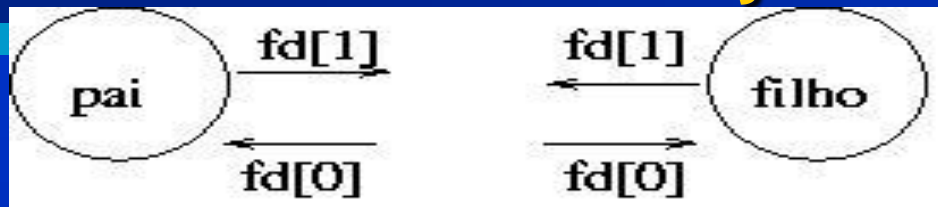
- **Escritor envia:**

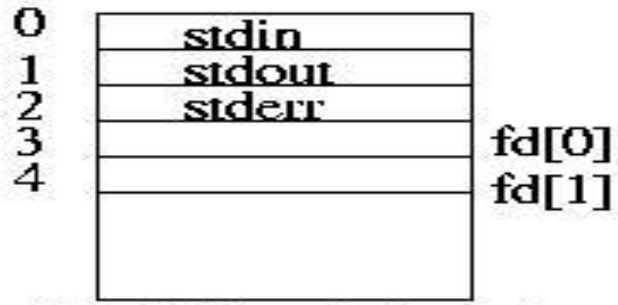
```
close(p[0]);
      …
write(p[1],…);
```

- **Leitor recebe:**

```
close(p[1]);
read(p[0],…);
     …;
```

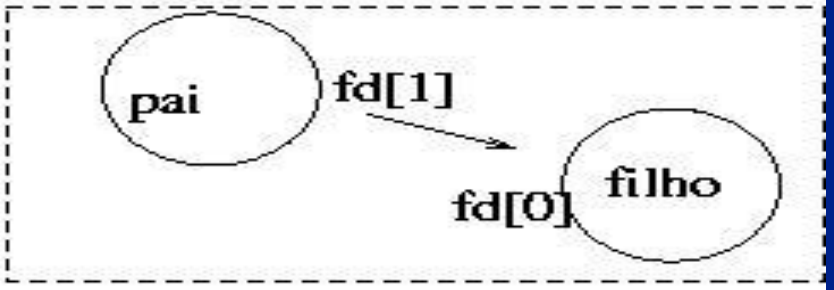
# Comunicação Pai -- Filho

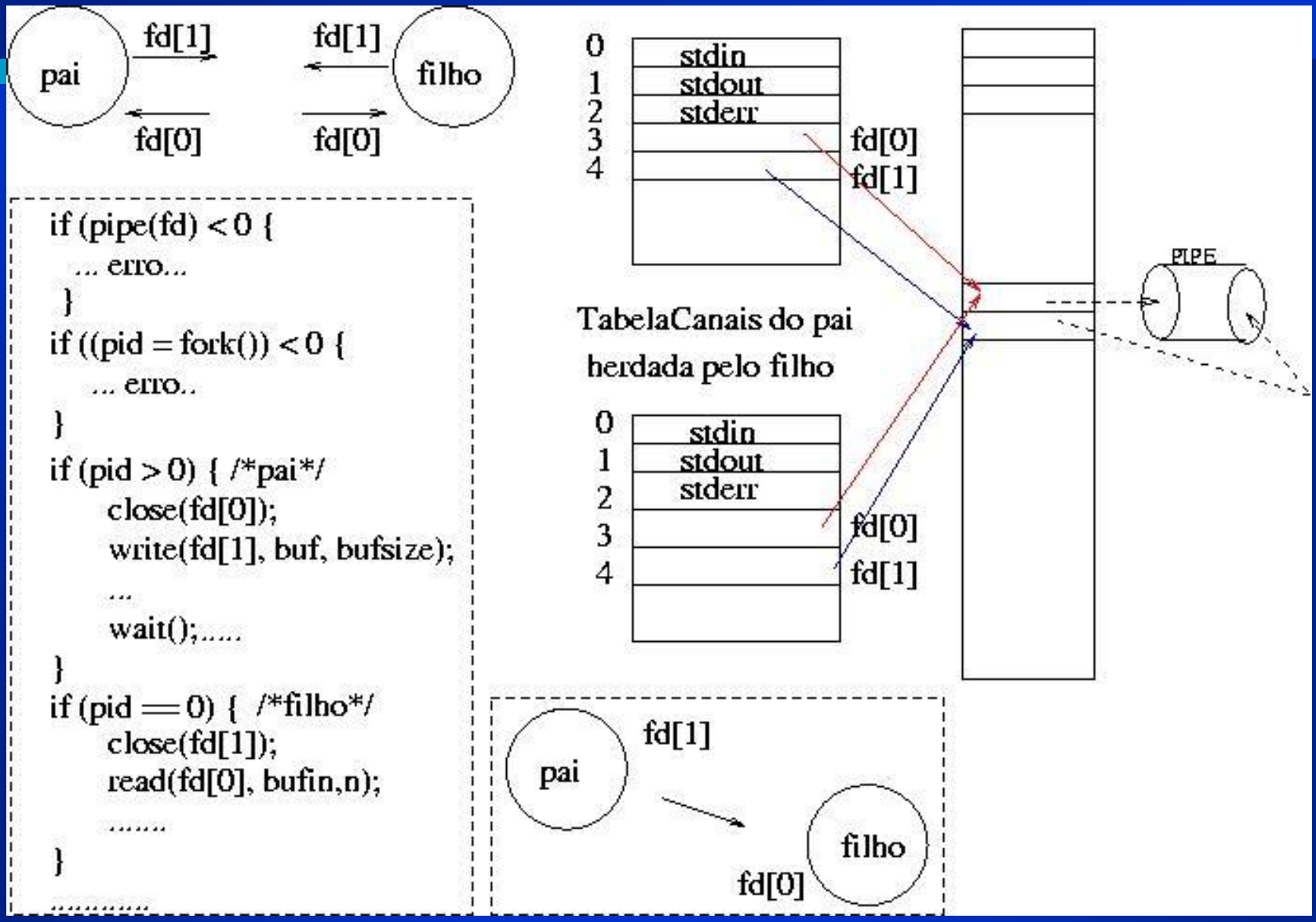



TabelaCanais do pai herdada pelo filho

```
if (pipe(fd) < 0 {
   ... erro...
 }
if ((pid = fork()) < 0 {
    ... erro..
}
if (pid > 0) { /*pai*/
      close(fd[0]);
      write(fd[1], buf, bufsize);
      ...
      wait();.....
}
if (pid == 0) {  /*filho*/
      close(fd[1]);
      read(fd[0], bufin,n);
      .......
}
.....
```




pai
filho
fd[1]
fd[1]
fd[0]
fd[0]
if (pipe(fd) < 0 {
... erro...
}
if ((pid = fork()) < 0 {
... erro..
}
if (pid > 0) { /*pai*/
close(fd[0]);
write(fd[1], buf, bufsize);
...
wait();.....
}
if (pid == 0) { /*filho*/
close(fd[1]);
read(fd[0], bufin,n);
.....
}
.....
0
1
2
3
4
stdin
stdout
stderr
fd[0]
fd[1]
TabelaCanais do pai
herdada pelo filho
0
1
2
3
4
stdin
stdout
stderr
fd[0]
fd[1]
PIPE
pai
fd[1]
filho
fd[0]

# Funcionamento do *pipe*

- **Um fluxo de bytes (FIFO) unidireccional**

# Funcionamento do *pipe*

- **Um fluxo de bytes (FIFO) unidireccional**
- **Os read / write podem bloquear enquanto decorre a comunicação:**
  - **write: se o pipe está cheio**
  - **read: se o pipe está vazio**

# Funcionamento do *pipe*

- **Um fluxo de bytes (FIFO) unidireccional**
- **Os read / write podem bloquear enquanto decorre a comunicação:**
  - **write: se o pipe está cheio**
  - **read: se o pipe está vazio**
- **Mas a comunicação só decorre enquanto existam:**

# Funcionamento do *pipe*

- **Um fluxo de bytes (FIFO) unidireccional**
- **Os read / write podem bloquear enquanto decorre a comunicação:**
  - **write: se o pipe está cheio**
  - **read: se o pipe está vazio**
- **Mas a comunicação só decorre enquanto existam:**

  **Leitor, que possa ler: canal de leitura**

# Funcionamento do *pipe*

- **Um fluxo de bytes (FIFO) unidireccional**
- **Os read / write podem bloquear enquanto decorre a comunicação:**
  - **write: se o pipe está cheio**
  - **read: se o pipe está vazio**
- **Mas a comunicação só decorre enquanto existam:**

  **Leitor, que possa ler: canal de leitura**

  **Escritor, que possa escrever: canal de escrita**

# Ler de um *pipe*

**read(fd[0], buf, count)**

- **bloqueia o processo que a invoca (leitor), se o buffer do pipe está vazio**

# Ler de um *pipe*

**read(fd[0], buf, count)**

- **bloqueia o processo que a invoca (leitor), se o buffer do pipe está vazio**
- **devolve os bytes que puder ler, mesmo se forem menos dos que os pedidos:**

**n = read(fd[0], buf, count)**

# Escrever num *pipe*

- **Tamanho do *buffer* usado pelo SO para o pipe: parâmetro do SO (min. 512 bytes)**

**write(fd[1],buf,bufsize)**

**Se bufsize excede o nº de bytes livres no buffer do pipe: -> bloqueia o escritor, até haver espaço livre**

# Escrever num *pipe*

- **Tamanho do *buffer* usado pelo SO para o pipe: parâmetro do SO (min. 512 bytes)**

**write(fd[1],buf,bufsize)**

**Se bufsize excede o nº de bytes livres no buffer do pipe: -> bloqueia o escritor, até haver espaço livre**

**Se bufsize excede a capacidade do buffer do pipe:**

# Escrever num *pipe*

- **Tamanho do *buffer* usado pelo SO para o pipe: parâmetro do SO (min. 512 bytes)**

**write(fd[1],buf,bufsize)**

**Se bufsize excede o nº de bytes livres no buffer do pipe: -> bloqueia o escritor, até haver espaço livre**

**Se bufsize excede a capacidade do buffer do pipe:**

- **escreve a parte dos bytes de buf que couberem**

# Escrever num *pipe*

- **Tamanho do *buffer* usado pelo SO para o pipe: parâmetro do SO (min. 512 bytes)**

**write(fd[1],buf,bufsize)**

**Se bufsize excede o nº de bytes livres no buffer do pipe: -> bloqueia o escritor, até haver espaço livre**

**Se bufsize excede a capacidade do buffer do pipe:**

- **escreve a parte dos bytes de buf que couberem**
- **bloqueia o escritor, até que possa escrever os restantes**

# Escrever num *pipe*

- **Tamanho do *buffer* usado pelo SO para o pipe: parâmetro do SO (min. 512 bytes)**

**write(fd[1],buf,bufsize)**

**Se bufsize excede o nº de bytes livres no buffer do pipe: -> bloqueia o escritor, até haver espaço livre**

**Se bufsize excede a capacidade do buffer do pipe:**

- **escreve a parte dos bytes de buf que couberem**
- **bloqueia o escritor, até que possa escrever os restantes**
- **a escrita deixa de ser ATÓMICA ou INDIVISÍVEL!**

# Funcionamento do *pipe*

- **Como sabe um processo que a comunicação terminou?**
- **Ou como termina um processo a comunicação?**

# Funcionamento do *pipe*

- **Como sabe um processo que a comunicação terminou?**
- **Ou como termina um processo a comunicação?**
- **se todos os canais de escrita -> fechados:**

  **- read de pipe vazio devolve 0 (em vez de bloquear)**

# Funcionamento do *pipe*

- **Como sabe um processo que a comunicação terminou?**
- **Ou como termina um processo a comunicação?**
- **se todos os canais de escrita -> fechados:**

  **- read de pipe vazio devolve 0 (em vez de bloquear)**
- **se todos os canais de leitura -> fechados:**

  **- write em pipe sem leitores dá erro EPIPE**

# Fechar canais abertos para pipe

**close(fd[1]): fecha canal de escrita**

**Se há outros canais abertos de escrita:**

- **Limita-se a fechar a entrada da TabCanais do processo**

# Fechar canais abertos para pipe

**close(fd[1]): fecha canal de escrita**

**Se há outros canais abertos de escrita:**

- **Limita-se a fechar a entrada da TabCanais do processo**

**Se não há mais canais abertos de escrita:**

- **e se o pipe está vazio:**
  - **quaisquer read() posteriores → devolvem 0**

# Fechar canais abertos para pipe

**close(fd[1]): fecha canal de escrita**

**Se há outros canais abertos de escrita:**

**Limita-se a fechar a entrada da TabCanais do processo**

**Se não há mais canais abertos de escrita:**

**e se o pipe está vazio:**

- **quaisquer read() posteriores → devolvem 0**
- **quaisquer read() que estejam bloqueados:**

**→ são reactivados e devolvem 0**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

**Se não há mais canais abertos de leitura:**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

**Se não há mais canais abertos de leitura:**

- **quaisquer write() bloqueados, são reactivados com um sinal SIGPIPE (interrupção software):**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

**Se não há mais canais abertos de leitura:**

- **quaisquer write() bloqueados, são reactivados com um sinal SIGPIPE (interrupção software):**

  **- se os processos não o previram → são abortados**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

**Se não há mais canais abertos de leitura:**

- **quaisquer write() bloqueados, são reactivados com um sinal SIGPIPE (interrupção software):**
  - **se os processos não o previram → são abortados**
  - **se o previram → tratam o sinal e depois a função write() retorna –1 e errno == EPIPE**

# close(fd[0]): fecha canal de leitura

**Se há outros canais abertos de leitura:**

**- apenas fecha essa entrada na TabCanais**

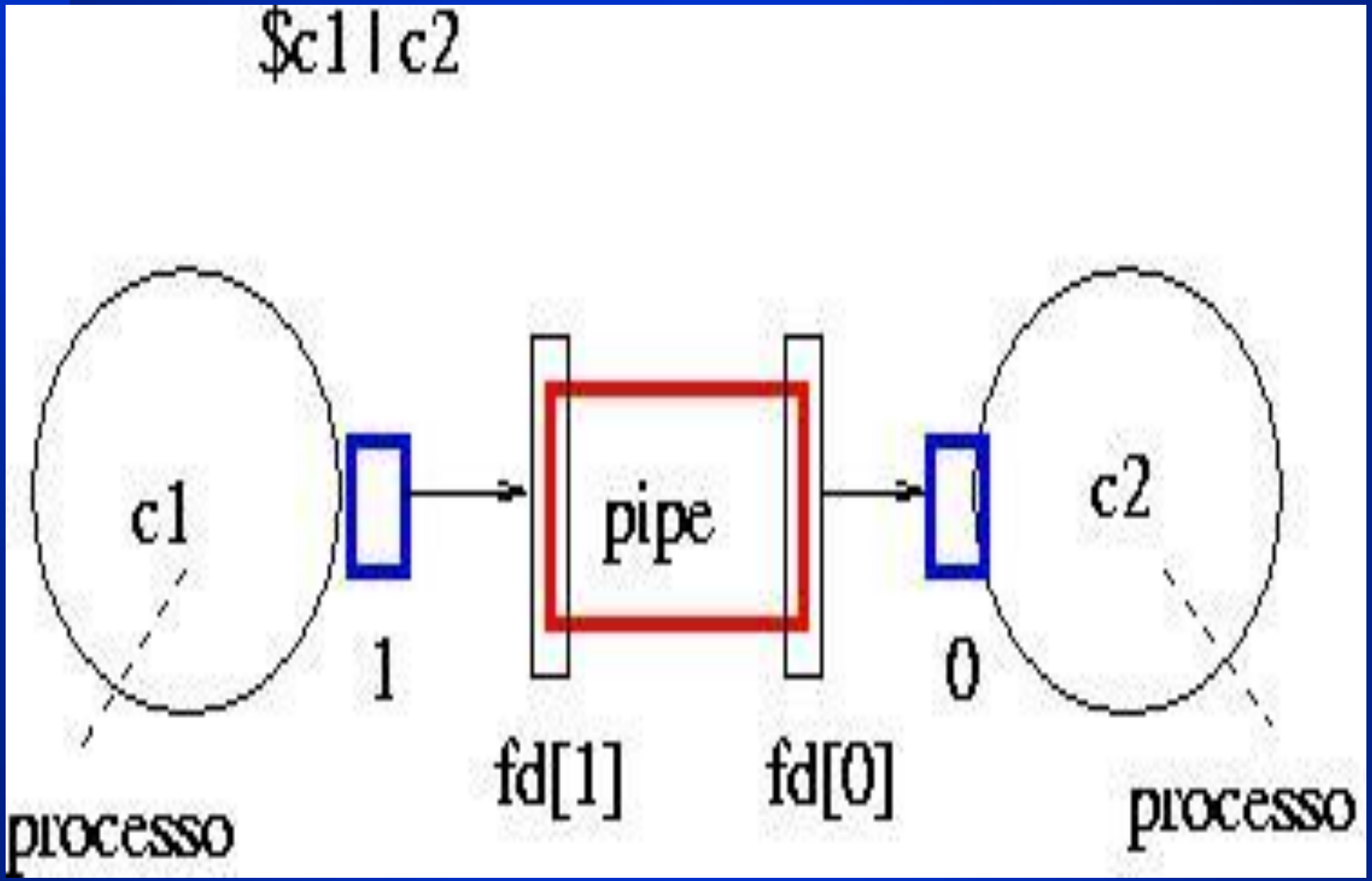
**Se não há mais canais abertos de leitura:**

- **quaisquer write() bloqueados, são reactivados com um sinal SIGPIPE (interrupção software):**
  - **se os processos não o previram → são abortados**
  - **se o previram → tratam o sinal e depois a função write() retorna –1 e errno == EPIPE**
- **quaisquer write() posteriores:**

  **também recebem o sinal.**

# Redirecção dos canais de I/O

# Um pipeline de processos



# Redirigir canais para *pipes*

- **No *shell* (*pipeline* de comandos):**

```
ls /bin | wc -l
```

**`wc` conta o num. de linhas geradas por `ls`**

# Redirigir canais para *pipes*

- **No *shell* (*pipeline* de comandos):**

```
ls /bin | wc -l
```

**`wc` conta o num. de linhas geradas por `ls`**

- **quando `ls` fizer `write(1,…)`, deve escrever no lado de escrita do *pipe***

# Redirigir canais para *pipes*

- **No *shell* (*pipeline* de comandos):**

```
ls /bin | wc -l
```

**`wc` conta o num. de linhas geradas por `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`**

# Redirigir canais para *pipes*

- No *shell* (*pipeline* de comandos):

```
ls /bin | wc -l
```

`wc` conta o num. de linhas geradas por `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, fecha o canal de escrita do pipe → termina a comunicação

Ao fazer fork(), o filho herda os canais abertos pelo pai:

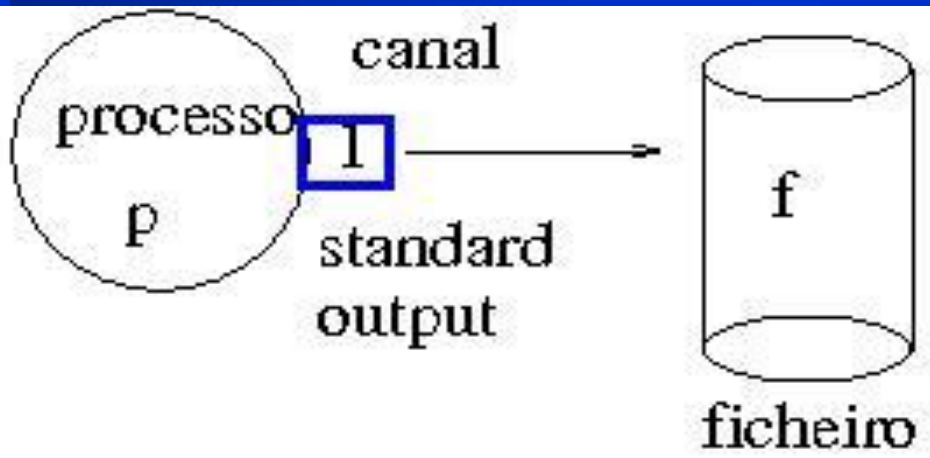
se um processo pai criar um pipe, antes do fork(), todos os seus filhos herdam os canais abertos para o pipe…

# Ligar um canal a um lado de um pipe

**Se um ´´pipe´´ fosse um ficheiro em disco… →**

- **Ligar o stdout (1) a um ficheiro ´´f´´ significa:**
  - **close: fechar o canal corrente 1**
  - **open: abrir um canal para o ficheiro ´´f´´**
  - **garantir que o canal aberto recebe o nº 1**

# Ex: Redirigir STDOUT para ficheiro




linha de comando
$ p > f

ao ser criado, p herda
o canal 1 aberto do pai

pode fechar :

```
close(1);
```

e abrir um canal para f

```
fd = open('f',O_WRONLY);
```

... o valor devolvido de fd = 1

se o canal 0 nao tiver sido fechado.

# Ligar um canal a um lado de um pipe

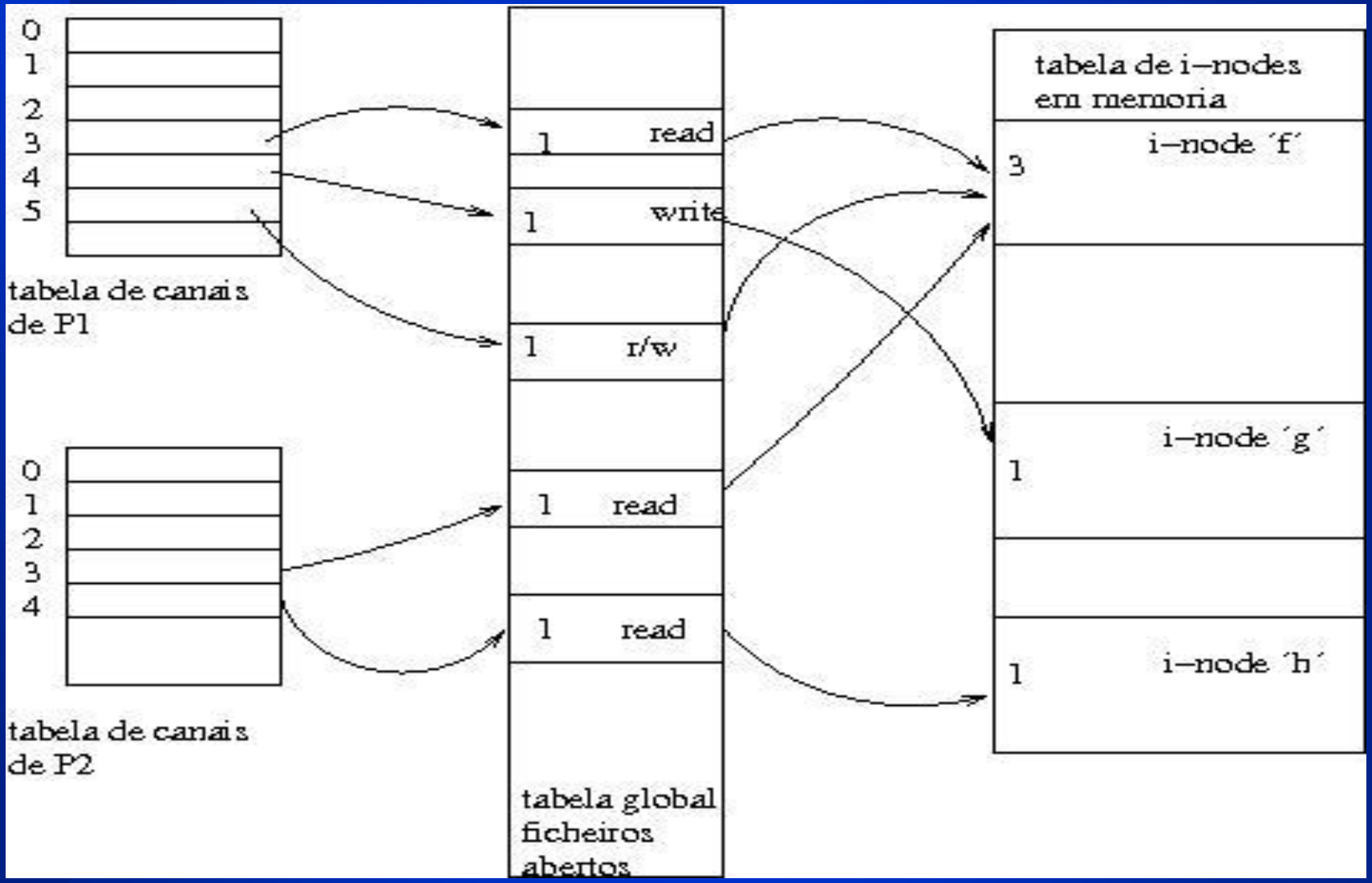
**?**

**Um *pipe* não tem nome…**

**Então não posso fazer**

**fd = open(´´nome´,…)**

**para abrir um canal para um *pipe***


0
1
2
3
4
5
tabela de canais de P1
0
1
2
3
4
tabela de canais de P2
1 read
1 write
1 r/w
1 read
1 read
tabela global ficheiros abertos
tabela de i-nodes em memoria
3
i-node ´f´
1
i-node ´g´
1
i-node ´h´

# ligar um canal standard a um pipe

-- **criar um *pipe*: abre duas entradas novas na Tab. Canais do processo, apontando os lados de leitura e de escrita do *pipe***

# ligar um canal standard a um pipe

**-- criar um *pipe*: abre duas entradas novas na Tab. Canais do processo, apontando os lados de leitura e de escrita do *pipe***

**Exemplo:**

```
int p[2]; -- array com posições p[0] e p[1]
pipe(p); -- por exemplo, p[0]= 3 e p[1]=4
```

**(se as entradas 0 e 1 estiverem ocupadas)**

**E agora?**

# Uma nova Chamada ao SO: Duplicar um ´´file descriptor´´

```
int dup( int old-fd )
```

- Na Tabela de Canais do Processo, cria uma nova entrada, na qual põe uma cópia da entrada old-fp

# Uma nova Chamada ao SO: Duplicar um ´´file descriptor´´

```
int dup( int old-fd )
```

- Na Tabela de Canais do Processo, cria uma nova entrada, na qual põe uma cópia da entrada old-fp

  e devolve o n° de canal da nova entrada.

# Uma nova Chamada ao SO: Duplicar um ´´file descriptor´´

```
int dup( int old-fd )
```

- Na Tabela de Canais do Processo, cria uma nova entrada, na qual põe uma cópia da entrada `old-fp`

  e devolve o n° de canal da nova entrada.

- A nova entrada é a 1ª que encontrou livre!

# Uma nova Chamada ao SO: Duplicar um ´´file descriptor´´

```
int dup( int old-fd )
```

- Na Tabela de Canais do Processo, cria uma nova entrada, na qual põe uma cópia da entrada old-fp

  e devolve o n° de canal da nova entrada.

- A nova entrada é a 1ª que encontrou livre!
- Depois de dup, há duas entradas com o mesmo ´´file descriptor´´

# Exemplo

TabCanais do processo

A

```
newdf = dup(3);      /*copia entrada 3 para ?*/
                     /*newfd == ?*/
```

B

```
close(1);            /*liberta entrada 1*/
newdf = dup(3);      /*copia entrada 3 para ?*/
                     /*newfd == ?*/
```

# Copiando descritores de canais

- **Obter cópias de canais já abertos:**

```
int dup2( int old-fd, int new-fd )
```

# Copiando descritores de canais

- **Obter cópias de canais já abertos:**

```
int dup2( int old-fd, int new-fd )
```

**1º fecha new-fd, se estiver aberto**

# Copiando descritores de canais

- **Obter cópias de canais já abertos:**

```
int dup2( int old-fd, int new-fd )
```

**1º fecha new-fd, se estiver aberto**

**2º na entrada new-fd da Tabela de Canais do processo, põe uma cópia da entrada old-fd**

# Diferença entre dup e dup2

```
fd = dup( old-fd );
```

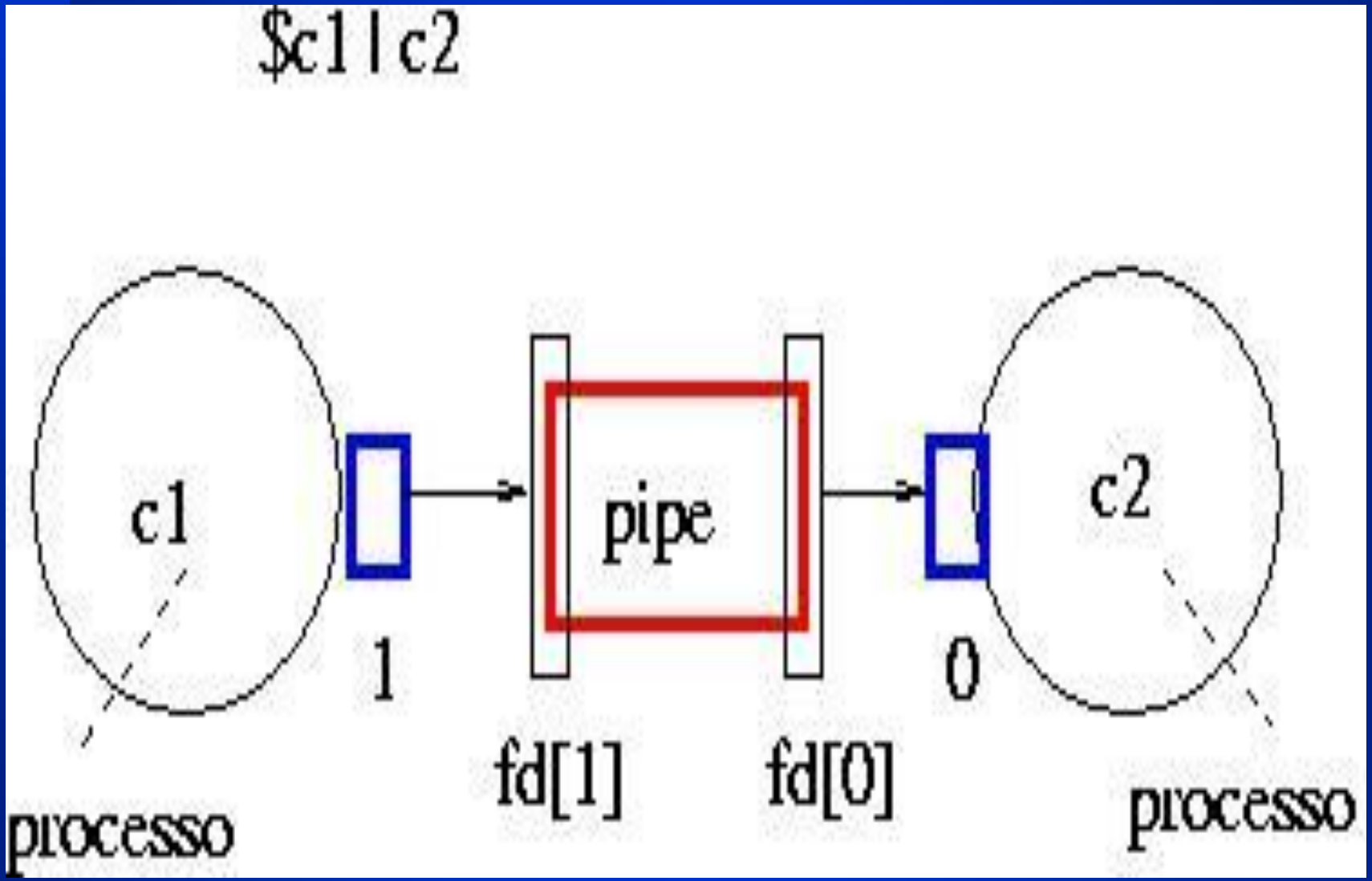
fd é um novo descritor para o mesmo canal que old-fd

(se quiser fd=1, tem de, primeiro, fazer close(1) para libertar essa entrada)

```
fd = dup2( old-fd, 1 );
```

fecha a entrada 1 (STDOUT) e coloca nessa entrada uma cópia da entrada old-fd: as entradas 1 e old-fd passam a ter dois canais abertos para o mesmo ´´ficheiro´´

# Um pipeline de processos



# Redirigir canais para *pipes*

- No *shell* (*pipeline* de comandos):

```
ls /bin | wc -l
```

- Contar o num. de linhas geradas por *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

# Programando: `ls /bin | wc -l`

- o *shell* cria dois processos e um pipe:
  - Ambos os filhos herdam os canais de leitura e escrita do pipe

# Programando: `ls /bin | wc -l`

- o *shell* cria dois processos e um pipe:
  - Ambos os filhos herdam os canais de leitura e escrita do pipe
  - No 1º processo, redirecciona-se o STDOUT (1) para o canal de escrita no *pipe*, antes de fazer *exec* do programa *ls*

# Programando: `ls /bin | wc -l`

- o *shell* cria dois processos e um pipe:
  - Ambos os filhos herdam os canais de leitura e escrita do pipe
  - No 1º processo, redirecciona-se o STDOUT (1) para o canal de escrita no *pipe*, antes de fazer *exec* do programa *ls*
  - No 2º processo, redirecciona-se o STDIN (0) para o canal de leitura do pipe, antes de fazer *exec* do programa *wc*

# Programando: `ls /bin | wc -l`

- o *shell* cria dois processos e um pipe:
  - Ambos os filhos herdam os canais de leitura e escrita do pipe
  - No 1º processo, redirecciona-se o STDOUT (1) para o canal de escrita no *pipe*, antes de fazer *exec* do programa *ls*
  - No 2º processo, redirecciona-se o STDIN (0) para o canal de leitura do pipe, antes de fazer *exec* do programa *wc*
  - Todos os canais não usados devem ser fechados, antes de invocar *exec*

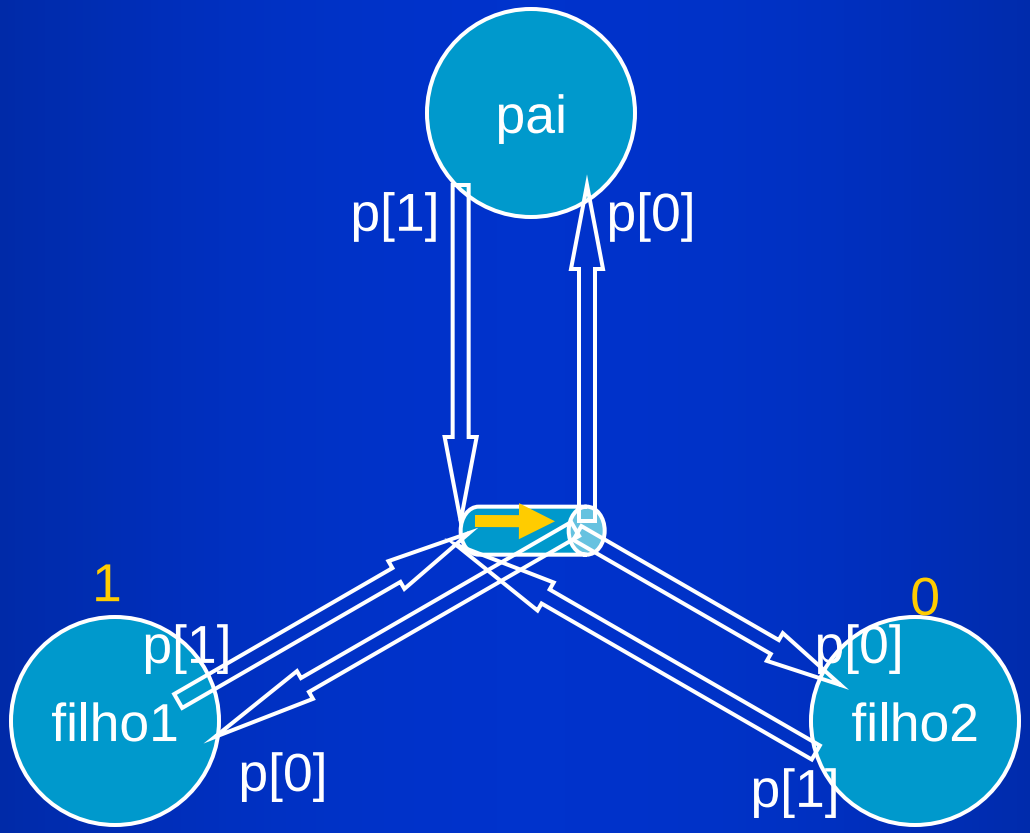
# Programando: ls /bin | wc -l

```
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);
}
```

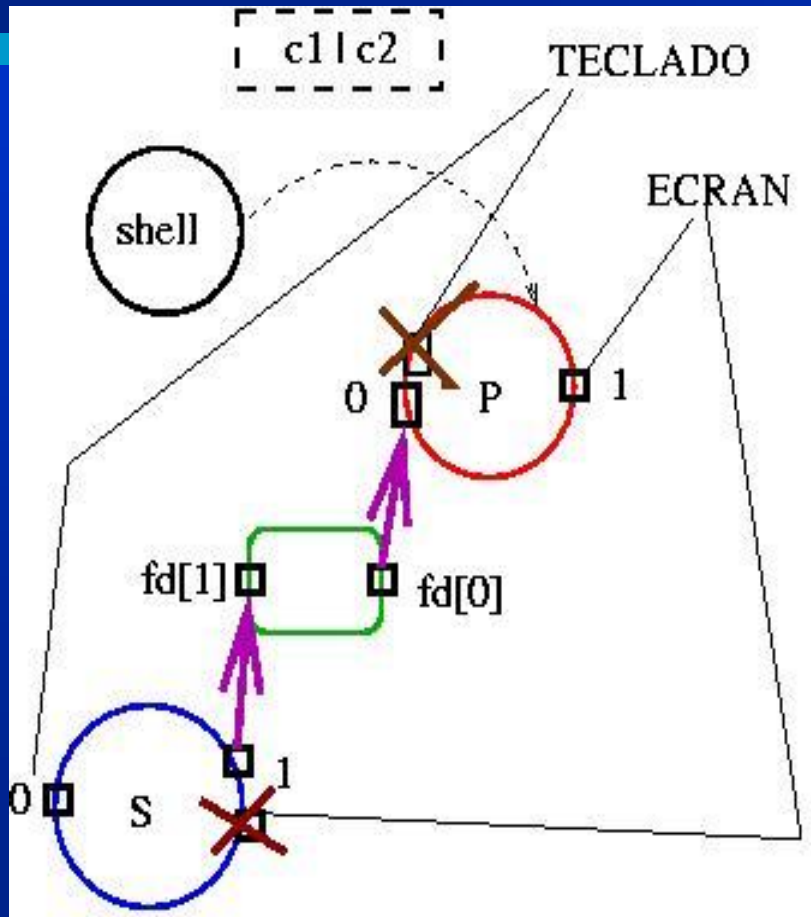
```
void filho1( int p[] )
{
  dup2(p[1],1);
  close(p[0]);
  close(p[1]);
  execlp("ls","ls,"/bin",0);
}


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

# filho1 | filho2



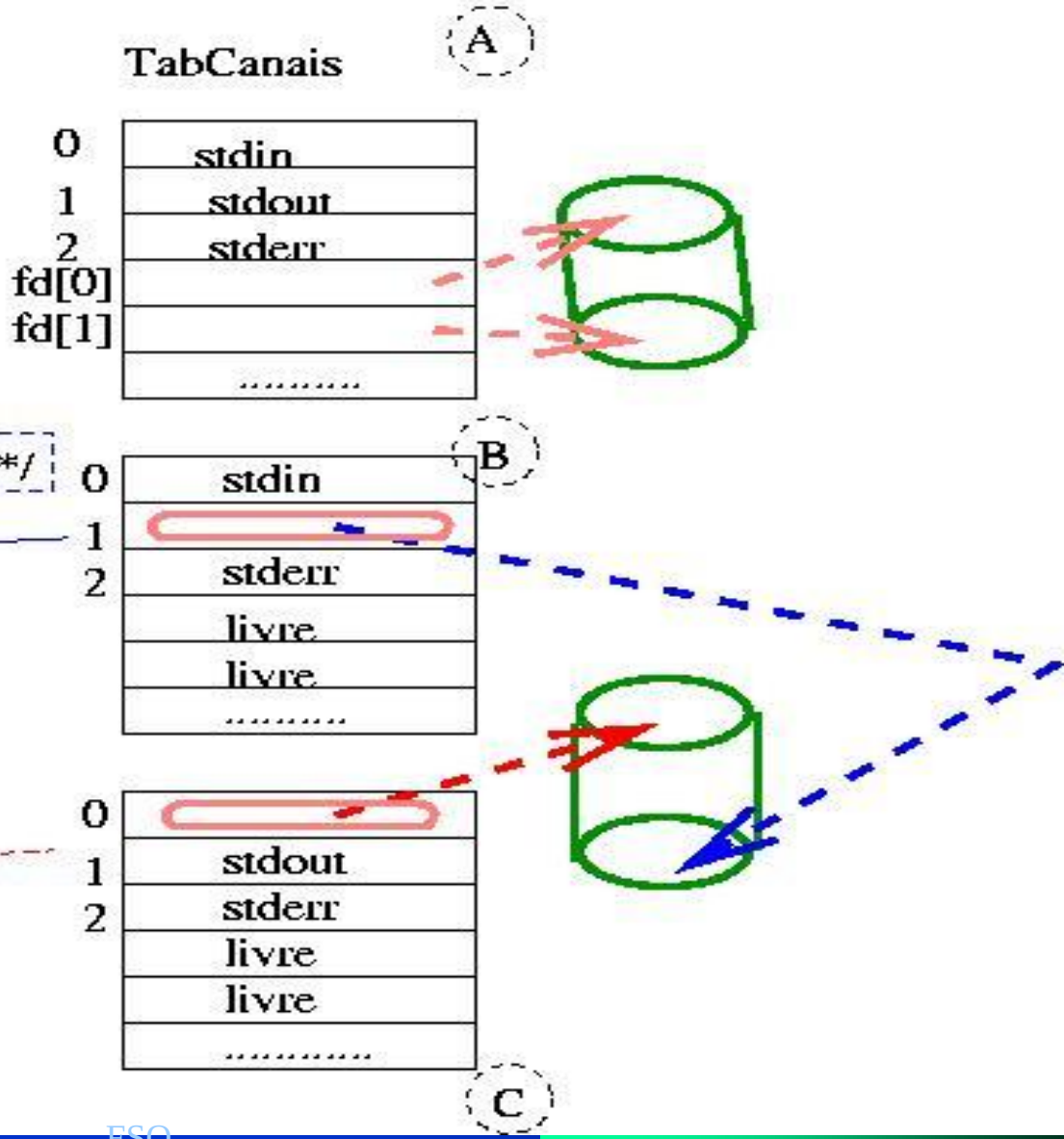
# Outra possível solução




Shell cria um processo filho
e depois faz uma chamada wait()

— um filho P:
  — cria um pipe
  — cria um processo filho S:
  — fecha o canal stdin (0)
  — liga o canal 0 ao lado leitura do pipe
  — fecha os canais inuteis
  — faz exec() para executar c2

— o filho S:
  — fecha o canal stdout
  — liga o canal 1 ao lado escrita do pip
  — fecha os canais inuteis
  — faz exec() para executar c1

```
int join(char *c1[], char*c2[])
{
  int fd[2], status, ret1, ret2;
 if (ret1 = fork()) > 0 { /*pai*/
          wait(&status);
          return(status);
 }
/* filho P*/
if (pipe(fd) < 0 {
        /*erro*/
}
/*filho P cria processo S */
if (ret2 = fork()) == 0 {/*processo S*/
      close(1);
      dup(fd[1]);
      close(fd[0]);
      close(fd[1]);
      execvp(c1[0],c1);
}
/*filho P  */
      close(0);
      dup(fd[0]);
      close(fd[0]);
      close(fd[1]);
      execvp(c2[0],c2);
}
```

c1 | c2