Laboratório de Pesquisa em Redes e Multimídia

Inter-process Communication (IPC)

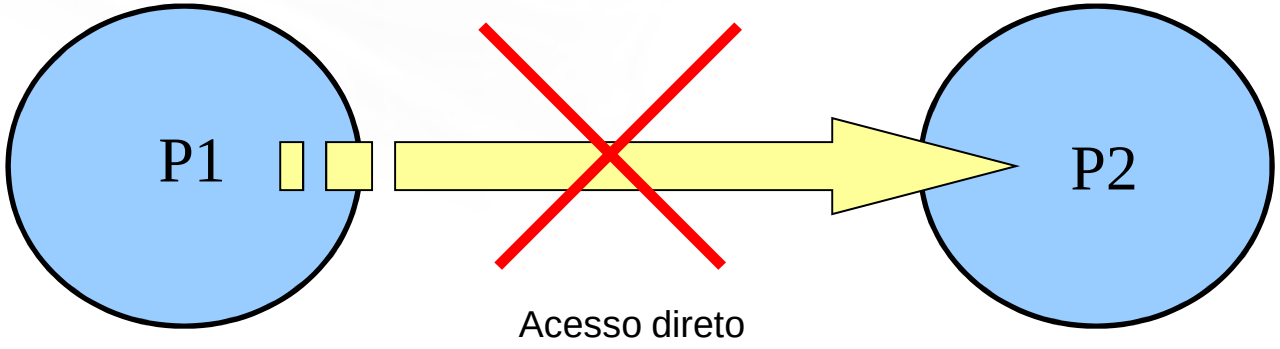
Comunicação entre processos (1)

Introdução
Tubos (*Pipes*)

Universidade Federal do Espírito Santo
Departamento de Informática

Comunicação entre Processos (1)

- Os sistemas operacionais implementam mecanismos que asseguram a independência entre processos.
  - Processos executam em cápsulas autônomas
    - A execução de um processo não afeta os outros.
  - Hardware oferece proteção de memória.
    - Um processo não acessa o espaço de endereçamento do outro.



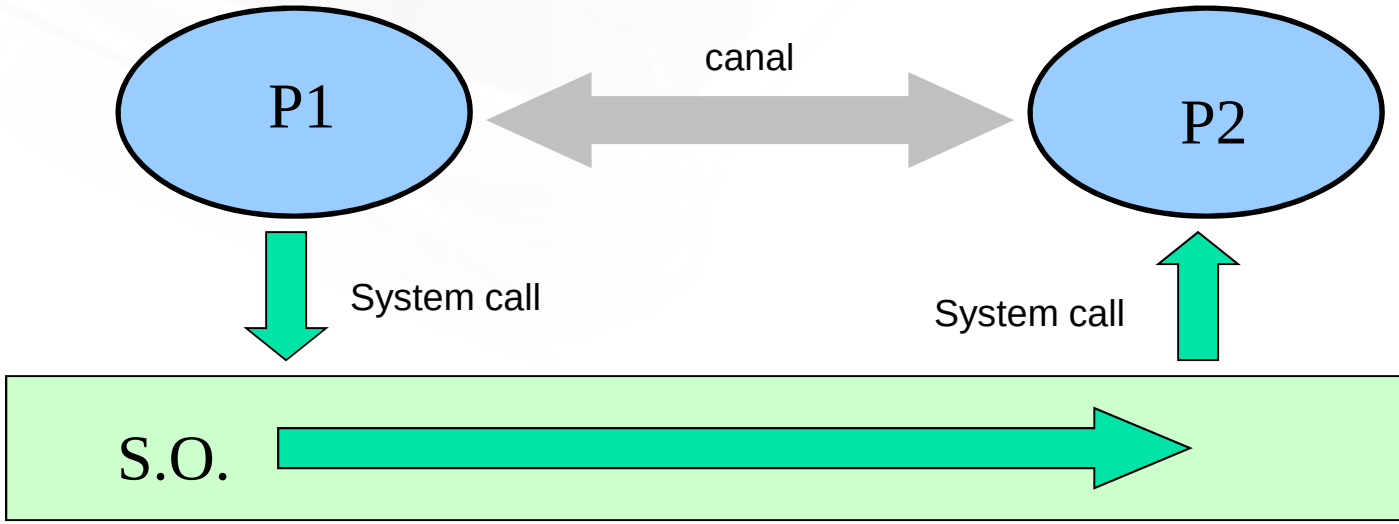
Acesso direto

Comunicação entre Processos (2)

- Processos, entretanto, interagem e cooperam na execução de tarefas. Em muitos casos, processos precisam trocar informação de forma controlada para
  - dividir tarefas e aumentar a velocidade de computação;
  - aumentar da capacidade de processamento (rede);
  - atender a requisições simultâneas.
- Solução: S.O. fornece mecanismos que permitem aos processos comunicarem-se uns com os outros (IPC).
- *IPC - Inter-Process Communication*
  - conjunto de mecanismos de troca de informação entre múltiplas threads de um ou mais processos.
  - Necessidade de coordenar o uso de recursos (sincronização).

Comunicação entre Processos (3)

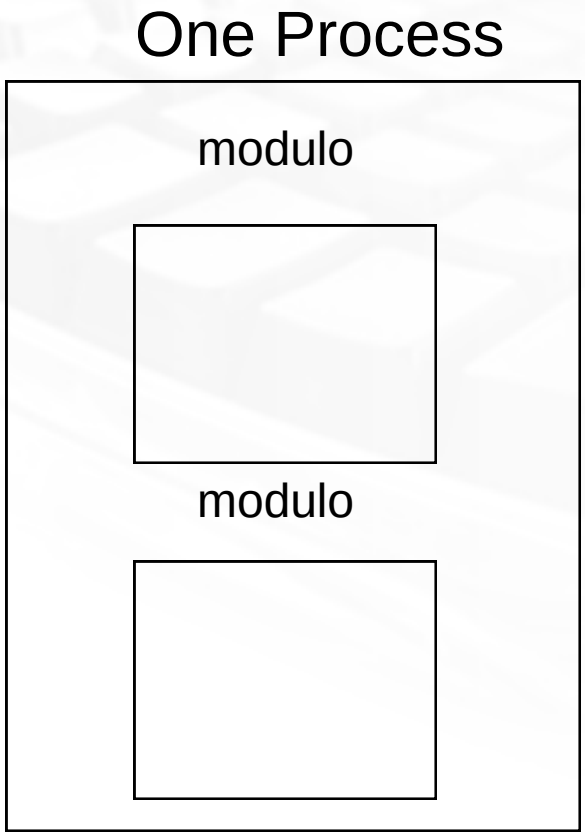
- Ao fornecer mecanismos de IPC, o S.O implementa "canais" de comunicação (implícitos ou explícitos) entre processos.



Comunicação entre Processos (4)

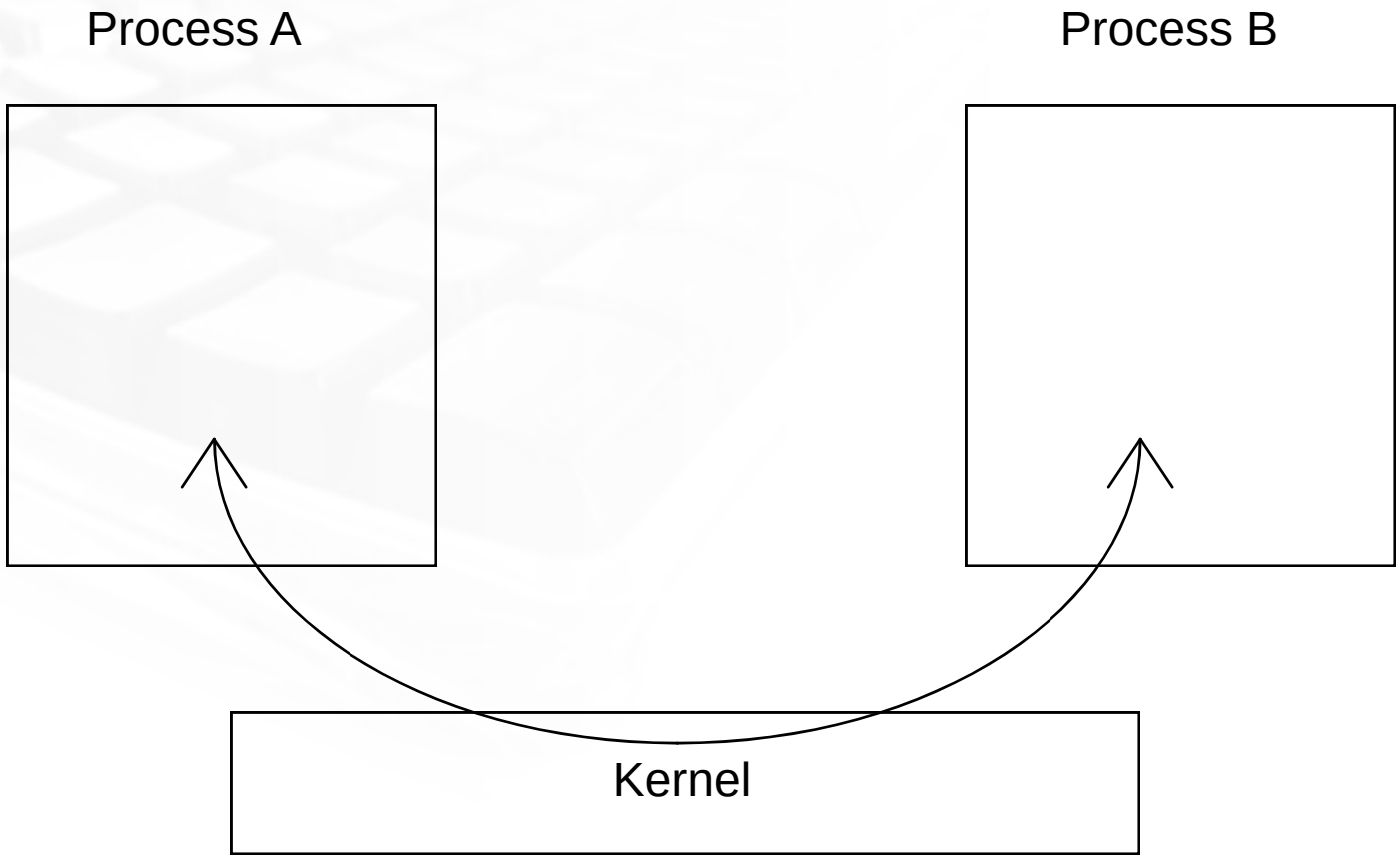
- Características desejáveis para IPC
  - Rápida
  - Simples de ser utilizada e implementada
  - Possuir um modelo de sincronização bem definido
  - Versátil
  - Funcione igualmente em ambientes distribuídos
- Sincronização é uma das maiores preocupações em IPC
  - Permitir que o *sender* indique quando um dado foi transmitido.
  - Permitir que um *receiver* saiba quando um dado está disponível .
  - Permitir que ambos saibam o momento em que podem realizar uma nova IPC.

IPC x Comunicação Interna

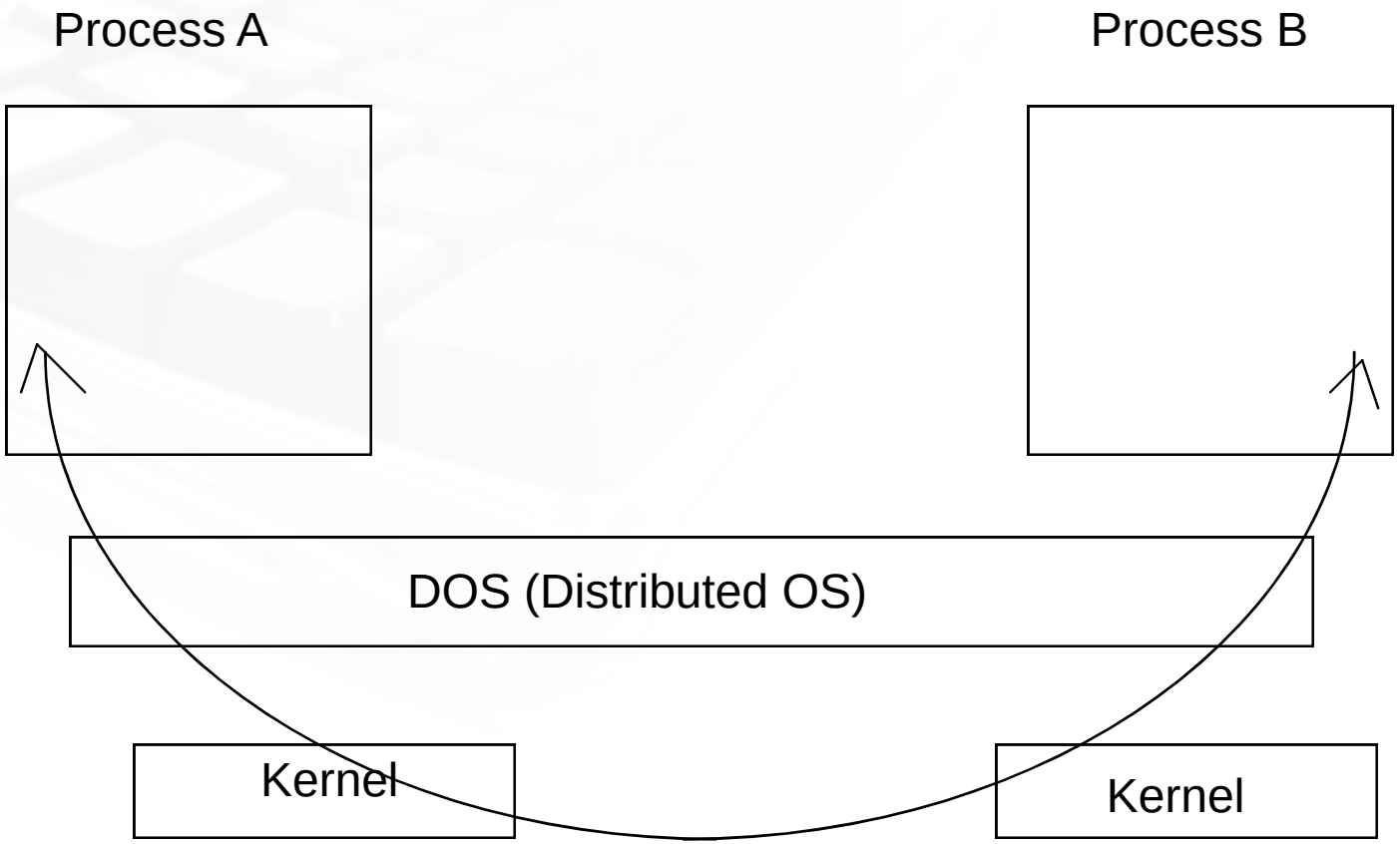


- Mecanismos de Comunicação Internos:
  - Variáveis globais
  - Chamadas de função
    - Parâmetros
    - resultados

IPC – Um Computador



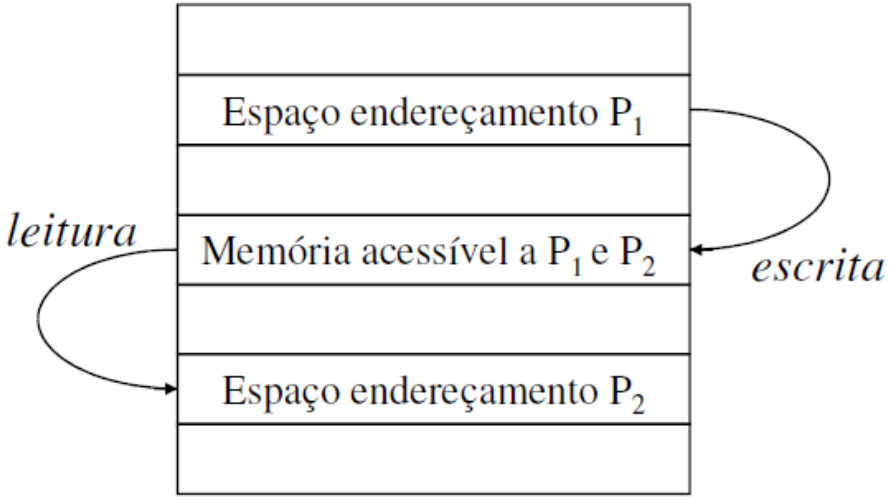
IPC – Dois Computadores



Mecanismos de IPC

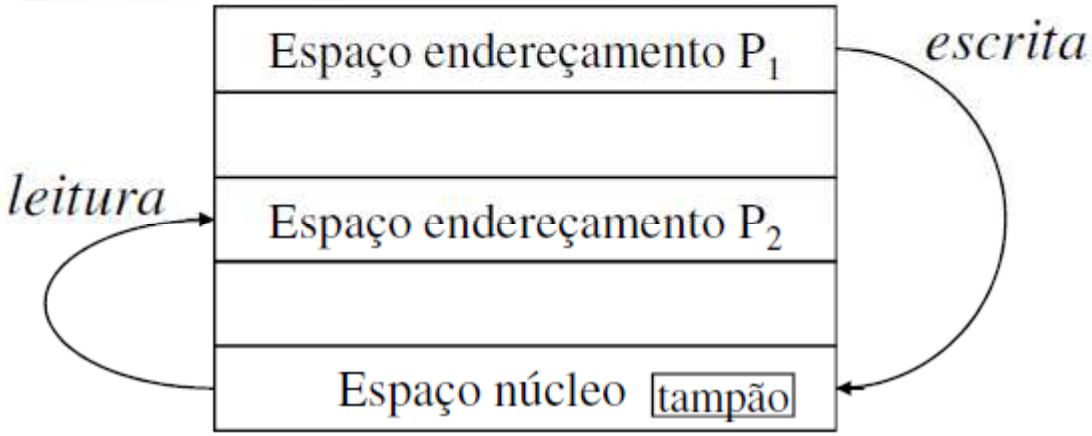
- Fundamentalmente, existem duas abordagens:
  - Suportar alguma forma de espaço de endereçamento compartilhado.
    - Shared memory (memória compartilhada)
  - Utilizar comunicação via núcleo do S.O., que ficaria então responsável por transportar os dados de um processo a outro. São exemplos:
    - 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

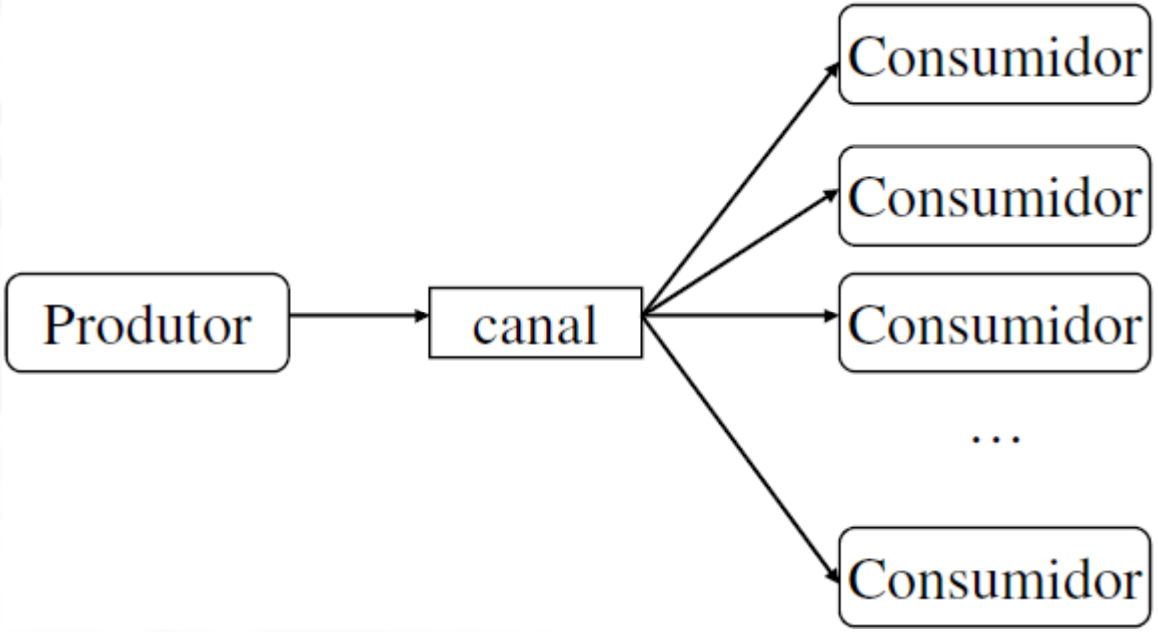


- Vantagens:
  - Pode ser realizada em sistemas com várias CPUs.
  - Sincronização implícita.
- Inconveniente:
  - Mais complexa e demorada (uso de recursos adicionais do núcleo).

Modelos de Comunicação

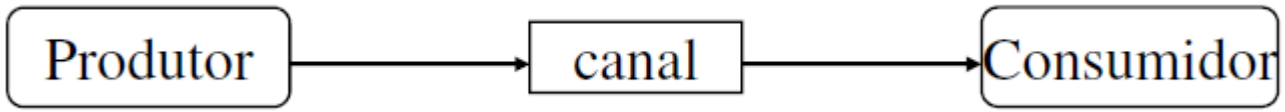
- Difusão ("*broadcast*"):
  - o emissor envia a mesma mensagem a todos os receptores.
- Produtor-consumidor:
  - comunicação uni-direcional.
- Cliente-servidor:
  - cliente controla totalmente o servidor.
- Caixa de correio (mailbox):
  - mensagens lidas por um processo receptor sem que o emissor (um entre vários) possa controlar quem recolhe a mensagem.
- Diálogo:
  - dois processos recebem um canal temporário para troca de mensagens durante uma sessão.

Modelos de Comunicação: Difusão



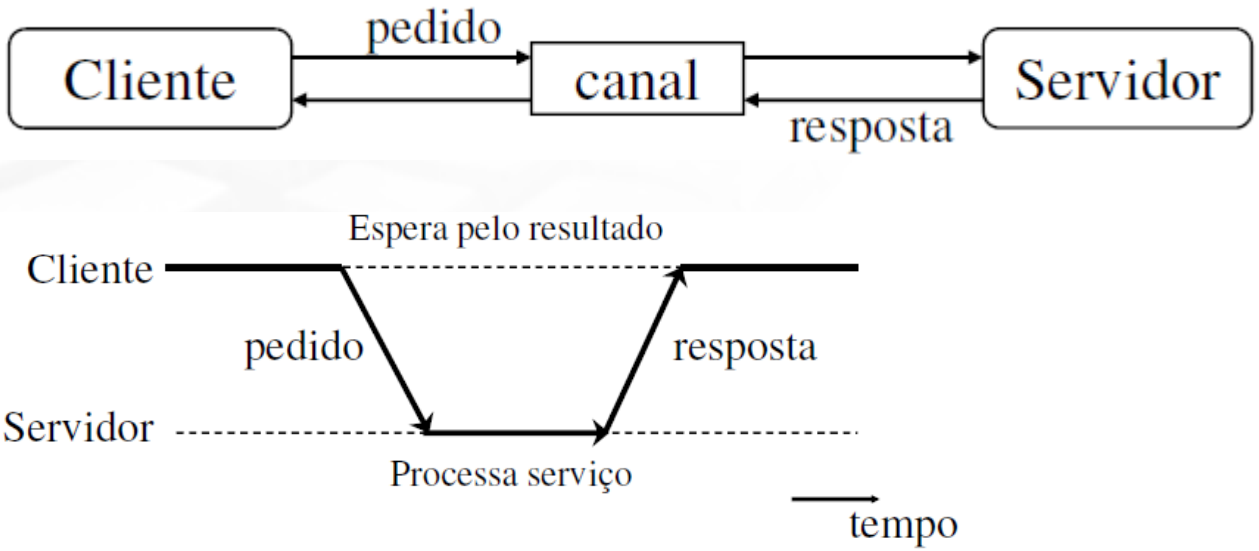
- O produtor envia mensagem a todos os consumidores, sem saber quem e quantos são.
- Comunicação "*broadcast*".

Modelos de Comunicação: Produtor-Consumidor



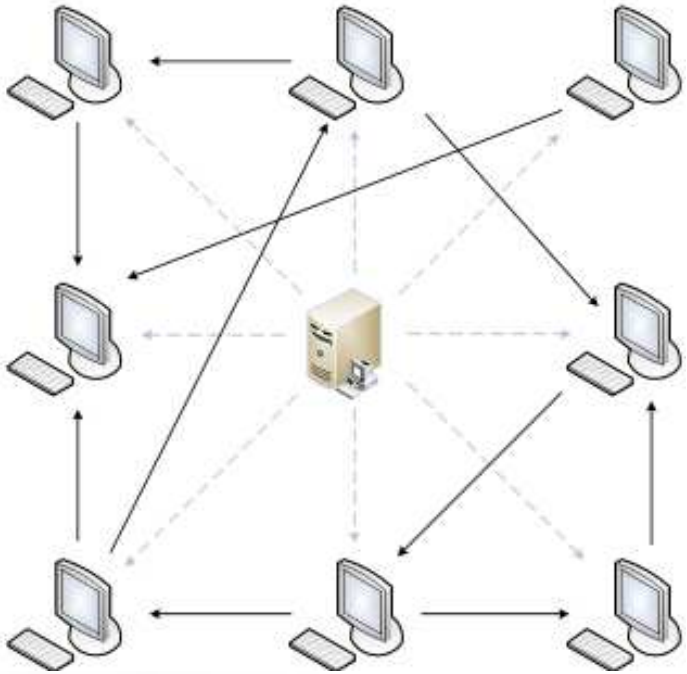
- Conexão unidirecional fixa, do produtor para o consumidor.
- Comunicação "*Unicast*".

Modelos de Comunicação: Cliente-Servidor



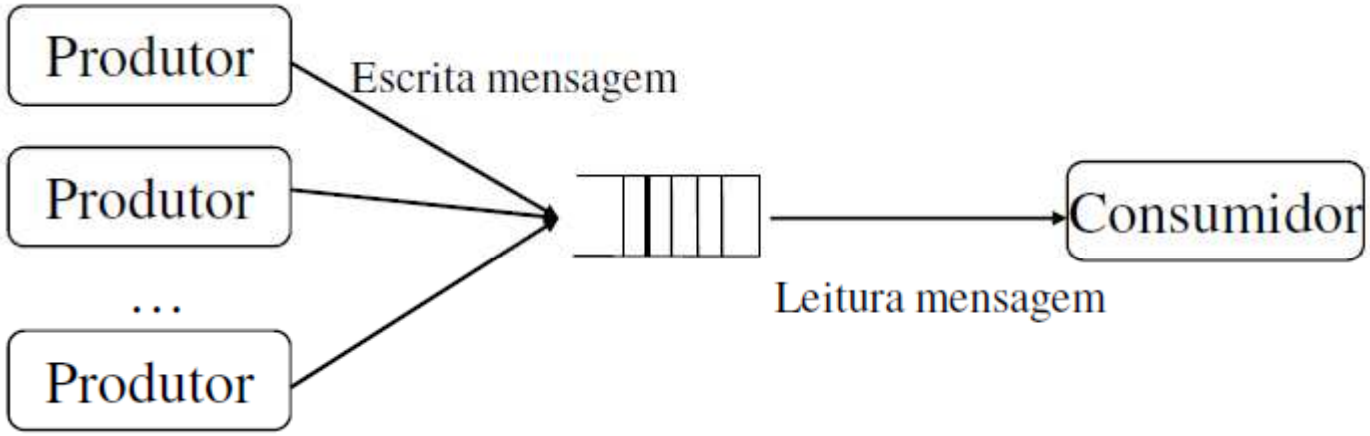
- Conexão bi-direcional fixa, entre o cliente (computador, programa ou processo que requer a informação) e o servidor (computador, programa ou processo que disponibiliza determinado serviço ao cliente).

Modelo de Comunicação: Peer-to-Peer (P2P)



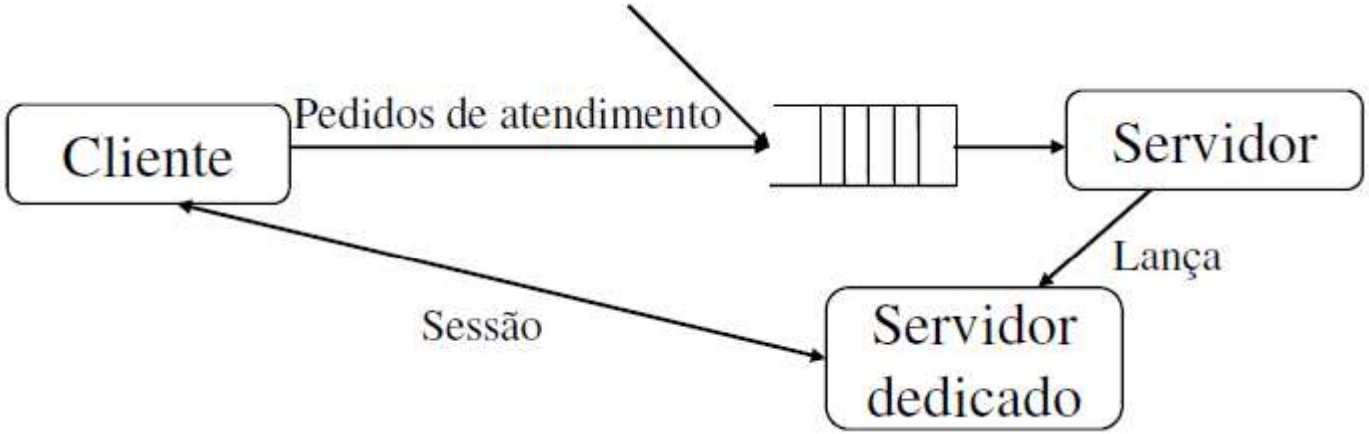
- No modelo P2P cada nó realiza, simultaneamente, tanto funções de servidor quanto de cliente.

Modelo de Comunicação: Mailbox



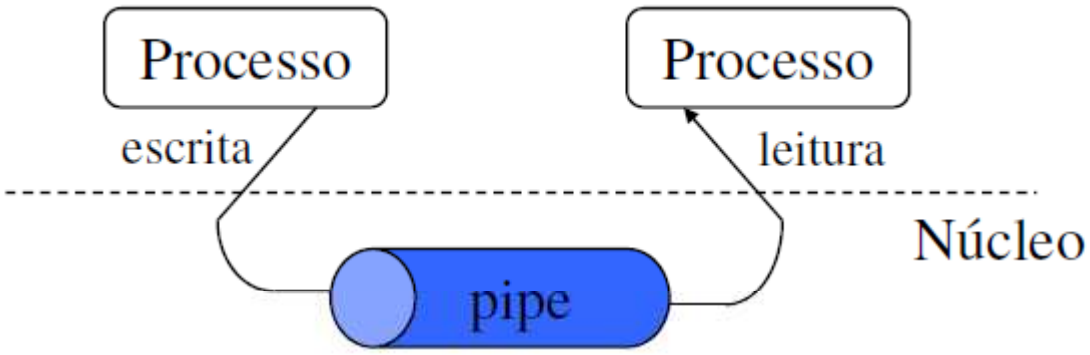
- A ligação entre produtor e consumidor é indireta, via caixa do correio (mailbox).
  - O consumidor não escolhe o produtor que escreveu a mensagem.
  - Tal como no modelo produtor-consumidor, a escrita não é bloqueante (admitindo uma caixa de correio de capacidade ilimitada) e a leitura é bloqueante quando a caixa se encontra vazia.

Modelo de Comunicação: "Diálogo"



- Criado um servidor dedicado para cada cliente, com ligação por canal dedicado.
- O canal é desligado quando a sessão termina.

Tubos (Pipes) (1)



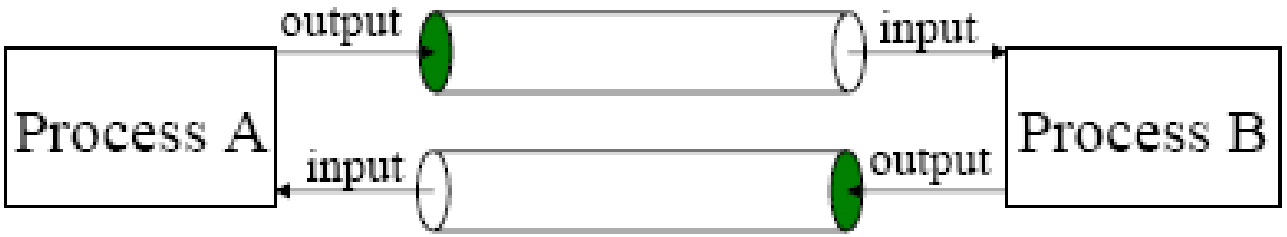
- No UNIX, os *pipes* constituem o mecanismo original de comunicação unidirecional entre processos.
- São um mecanismo de I/O com duas extremidades, correspondendo, na verdade, a filas de caractereres tipo FIFO.
- As extremidades são implementadas via descritores de arquivos (vide adiante).

Tubos (Pipes) (2)

- Um *pipe* tradicional caracteriza-se por ser:
  - Anônimo (não tem nome).
  - Temporário: dura somente o tempo de execução do processo que o criou.
- Vários processos podem fazer leitura e escrita sobre um mesmo *pipe*, mas nenhum mecanismo permite diferenciar as informações na saída do *pipe.*
- 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, consequentemente, abra espaço no *pipe*).

Tubos (Pipes) (3)

- É impossível fazer qualquer movimentação no interior de um *pipe*.
- Com a finalidade de estabelecer um diálogo entre dois processos usando *pipes*, é necessário a abertura de um *pipe* em cada direção.

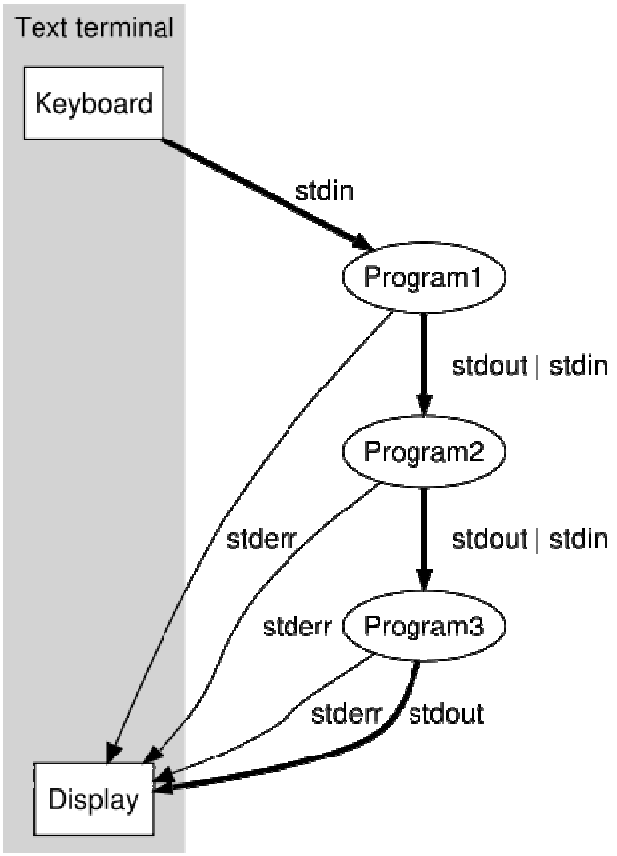


Uso de Pipes

- who | sort | lpr
  - output of *who* is input to *sort*
  - output of *sort* is input to */pr*

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" | \
sed 's/[^a-zA-Z ]/ /g' | \
tr 'A-Z ' 'a-z\n' | \
grep '[a-z]' | \
sort -u | \
comm -23 - /usr/dict/words
```

(1) **curl** obtains the HTML contents of a web page. (2) **sed** removes all characters which are not spaces or letters from the web page's content, replacing them with spaces. (3) **tr** changes all of the uppercase letters into lowercase and converts the spaces in the lines of text to newlines (each 'word' is now on a separate line). (4) **grep** removes lines of whitespace. (5) **sort** sorts the list of 'words' into alphabetical order, and removes duplicates. (6) Finally, **comm** finds which of the words in the list are not in the given dictionary file (in this case, /usr/dict/words).

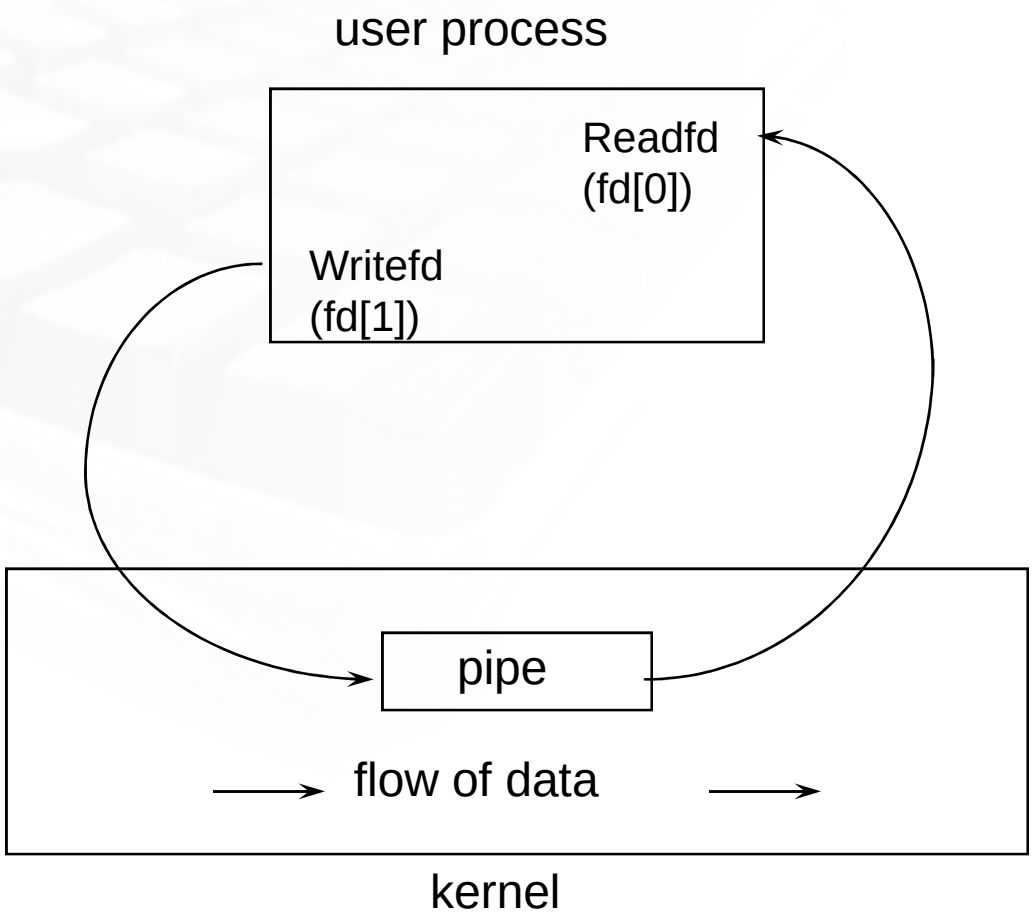


Criação de Pipes (1)

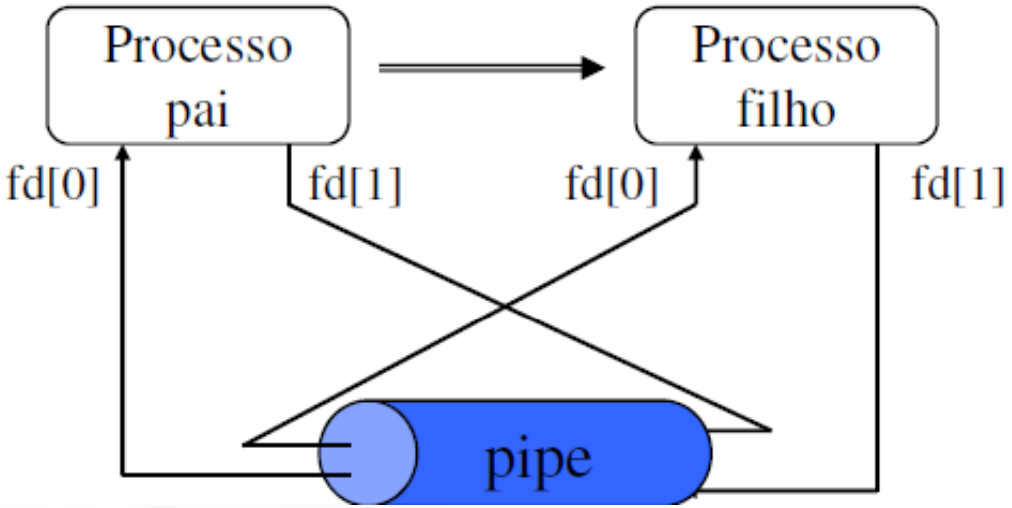
- *Pipes* constituem um canal de comunicação entre processos pai-filho.
  - Os *pipes* são definidos antes da criação dos processos descendentes.
  - Os *pipes* podem ligar apenas processos com antepassado comum.
- Um *pipe* é criado pela chamada de sistema:

  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.
- 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*.

Fechamento de Pipes (1)

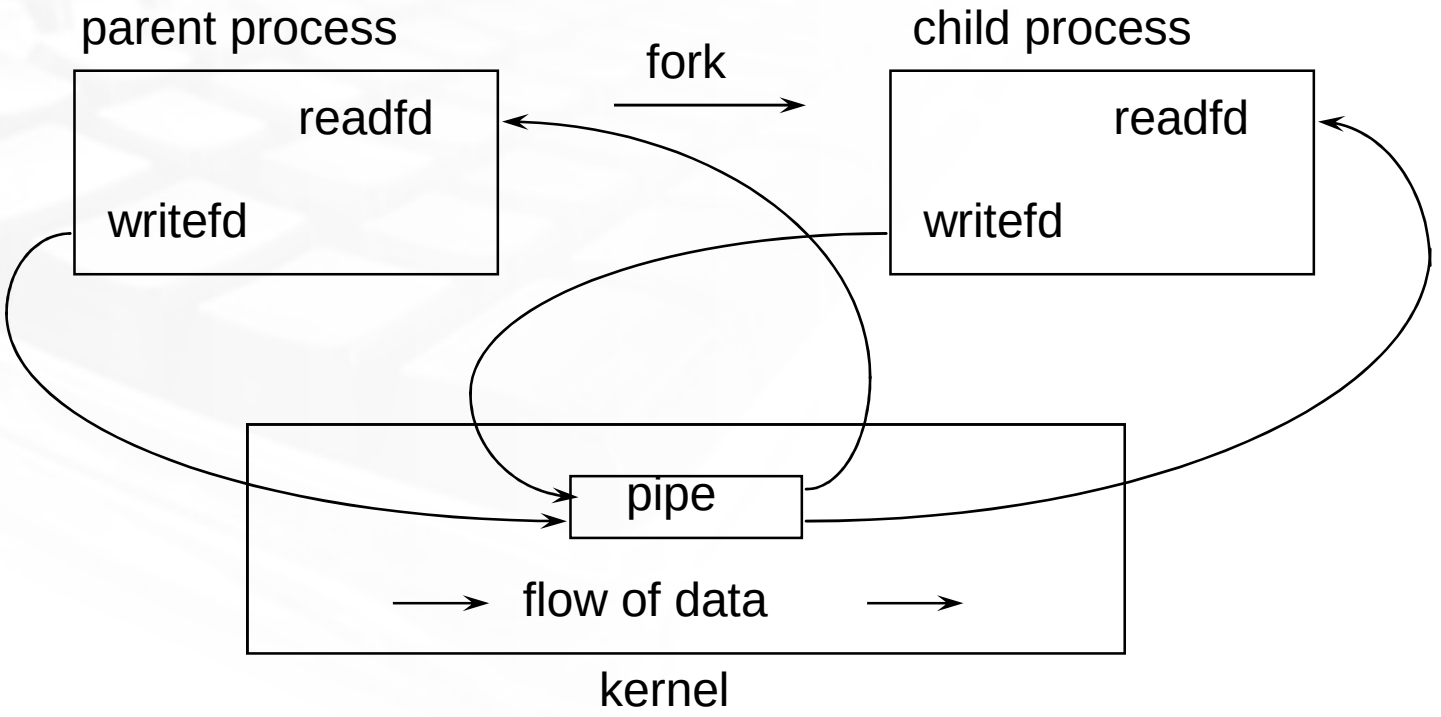
- Depois de usados, ambos os descritores devem ser fechados pela chamada do sistema:

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

- 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]);
}
```

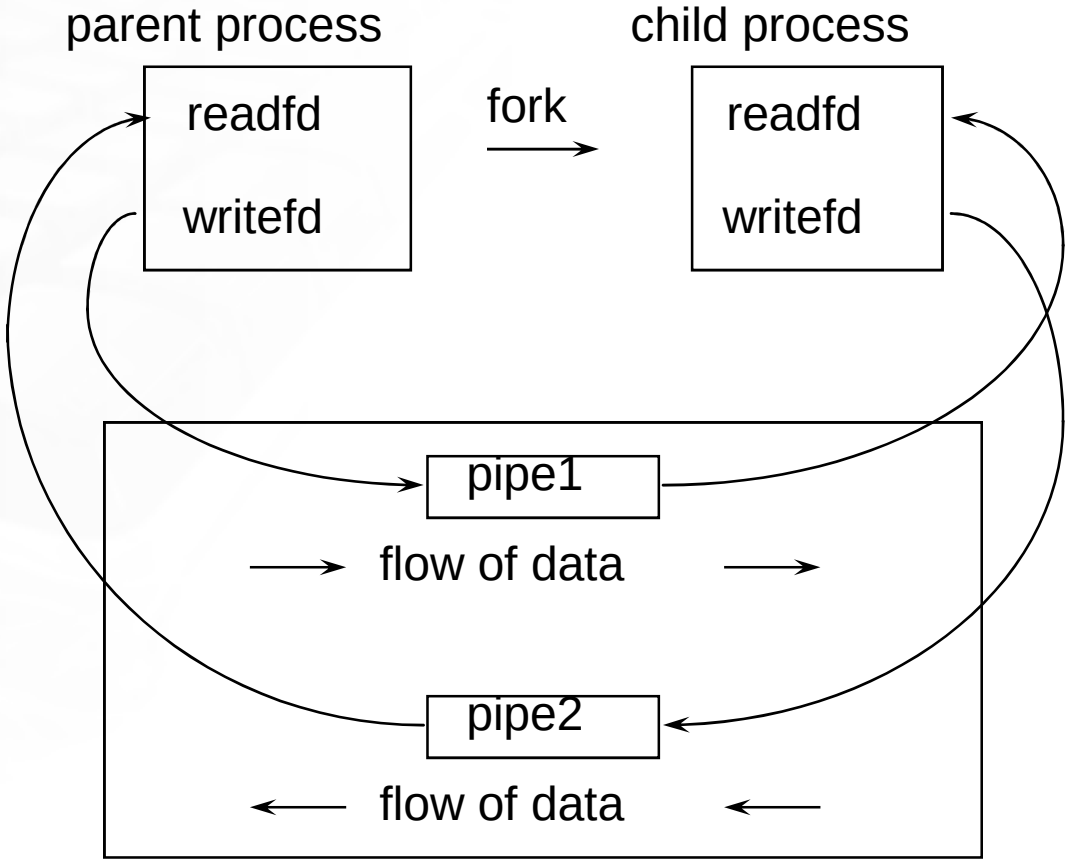
Comunicação Pai-Filho Unidirecional



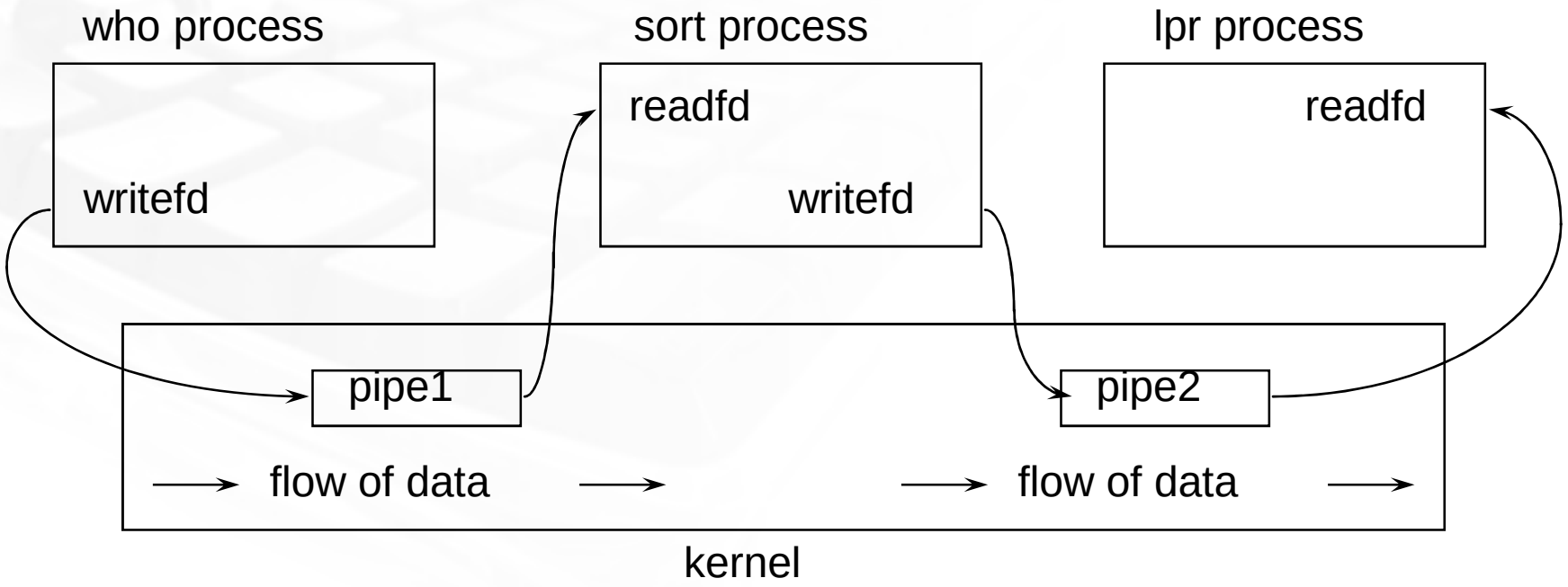
- Processo pai cria o *pipe*.
- Processo pai faz o *fork()*.
- Os descritores são herdados pelo processo filho.

Comunicação Pai-Filho Bi-Direcional

- 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.



who | sort | lpr



- Processo *who* escreve no *pipe1*.
- Processo *sort* lê do *pipe1* e grava no *pipe2*.
- Processo *lpr* lê do *pipe2*.

Escrita e Leitura em Pipes (1)

- 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.

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.

Exemplo1

- Processo pai envia dados para processo filho via pipe.

```
#include “ourhdr.h”
int main(void)
{
  int    n, fd[2];
  pid_t  pid;
  char   line[MAXLINE];

  if (pipe(fd) < 0) {
    err_sys("pipe error");}

  if (pipe(fd) = fork() < 0) {
    err_sys(“fork error");}

  else if (pid > 0) {                    /* processo pai */
    close(fd[0]);
    write(fd[1], "hello world\n", 12);
    error ("write error");

  }
  else {                                 /* processo filho */
    close(fd[1]);
    n = read(fd[0], line, MAXLINE);
    write(STDOUT_FILENO, line, n);
  }
  exit (0);
}
```

Exemplo 2

```
#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.

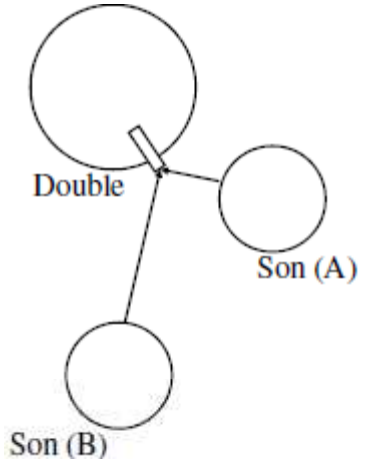
Exemplo 2 (cont.)

```
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(); }
}
```

Exemplo 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); }
```

- Dois processos filhos enviam mensagens para o processo pai.



Exemplo 3 (cont.)

```
#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("/home/ec-ps/public_html/Exemplos/Comunicacao/Pipe/Son",
              "Son", channel, "1", NULL); }
   pidA = pid;
```

Exemplo 3 (cont.)

```
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("/home/ec-ps/public_html/Exemplos/Comunicacao/Pipe/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 4

```
int count=0;
main()
{
  char c='x';
  if (pipe(p) < 0)
    error("pipe call");
  signal(SIGALRM,alarm_action);
  for(;;) {
    alarm(20);
    write(p[1],&c,1);
    alarm(0);
    if((++count%1024)==0)
      printf("%d chars in pipe\n, count");
  }
}
alarm_action()
{
   printf("write blocked after %d chars \n", count);
exit(0)
}
```

- O que faz este programa?