

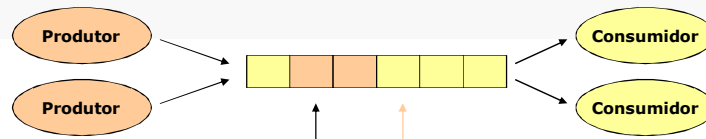
## Comunicação entre Processos

Canal de comunicação  
Arquitectura da comunicação  
Modelos de comunicação

## Necessidade da Comunicação

- A sincronização entre processos permitiu que diversas actividades possam cooperar na execução de um algoritmo.
- Contudo, logo se alertou para o facto de em muitas situações a cooperação implicar para além da sincronização [a transferência de informação](#)
- A comunicação entre processos ou IPC de InterProcess Communication é um dos aspectos do modelo computacional do sistema operativo que maior importância tem na programação de aplicações

## Exemplo de Cooperação entre Processos: Produtor - Consumidor



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco_p, trinco_c;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);
```

Optimização: Permite produzir e consumir ao mesmo tempo em partes diferentes do buffer

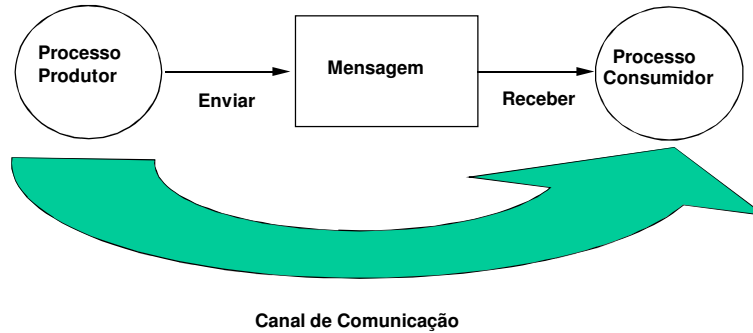
```
produtor()
{
    while(TRUE) {
        int item = produz();
        esperar(pode_prod);
        fechar(trinco_p);
        buf[prodptr] = item;
        prodptr = (prodptr+1) % N;
        abrir(trinco_p);
        assinalar(pode_cons);
    }
}
```

```
consumidor()
{
    while(TRUE) {
        int item;
        esperar(pode_cons);
        fechar(trinco_c);
        item = buf[consptr];
        consptr = (consptr+1) % N;
        abrir(trinco_c);
        assinalar(pode_prod);
        consome(item);
    }
}
```

## Comunicação entre Processos

- Para interagirem os processos necessitam de sincronizar-se e de trocar dados
  - Generalização do modelo de interação entre processos em que para além da sincronização existe transferência de informação
  - A transferência de informação é suportada por um canal de comunicação disponibilizado pelo sistema operativo
- Protocolo e estrutura das mensagens
  - Os processos que comunicam necessitam de estabelecer a estrutura das mensagens trocadas, bem como o protocolo que rege a troca das mesmas
  - Normalmente o sistema operativo considera as mensagens como simples seqüências de octetos
  - Numa visão OSI podemos ver estes mecanismo como correspondendo às camadas de Transporte e Sessão

## Comunicação entre Processos



- generalização do modelo de cooperação entre processos

## Exemplos

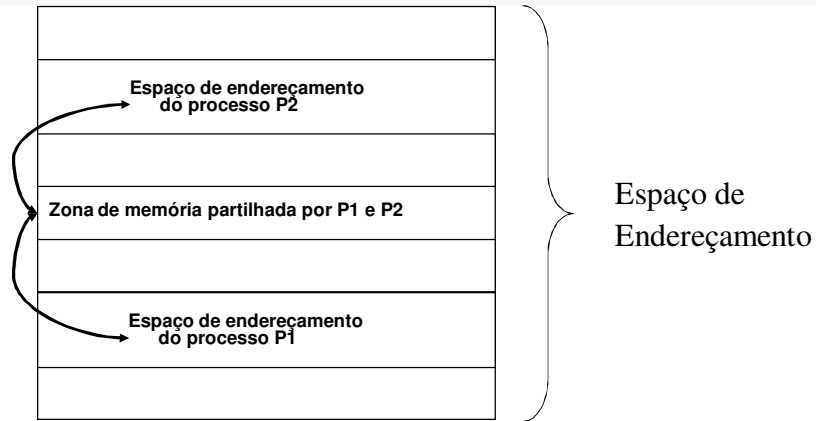
- A comunicação entre processos pode realizar-se no âmbito:
  - de uma única aplicação,
  - uma máquina
  - Entre máquinas interligadas por uma rede de dados
- Ex.. Outlook e exchange, servidores de base de dados, WWW, FTP, Telnet, SSH, MAIL, P2P

## Como implementar comunicação entre processos?

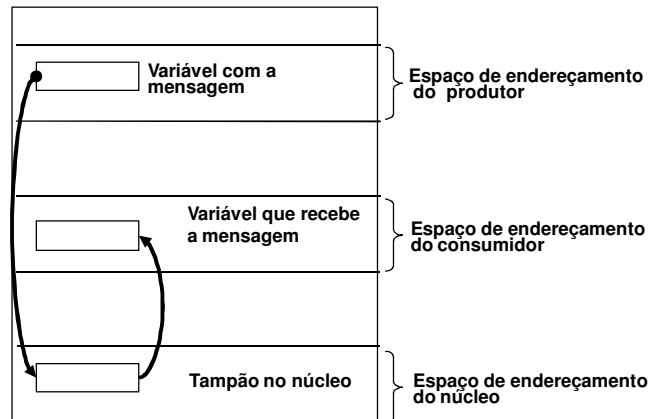
## Implementação do Canal de Comunicação

- Para implementar o canal de comunicação é fundamental definir como são transferidos os dados entre os espaços de endereçamento dos processos
- O canal de comunicação pode ser implementado com dois tipos de mecanismos:
  - **Memória partilhada:** os processos acedem a uma zona de memória que faz parte do espaço de endereçamento dos processos comunicantes
  - **Transferidos através do núcleo do sistema operativo;** os dados são sempre copiados para o núcleo antes de serem transferidos

## Arquitectura da Comunicação: memória partilhada



## Arquitectura da Comunicação: cópia através do núcleo



## Comparação

- A memória partilhada pode ser considerada como uma extensão à gestão da memória virtual, permitindo ultrapassar os mecanismos de protecção dos espaços de endereçamento
  - Num sistema paginado corresponde a acrescentar uma tabela de páginas que descreve a região de memória partilhada
- A comunicação no núcleo é semelhante a gestão de outros objectos do sistema em particular os ficheiros
  - Na execução da chamada sistema `Enviar` a mensagem é copiada para o núcleo e na chamada `Receber` é copiada do núcleo para o espaço de endereçamento do processo.

## Memória Partilhada

- `Apont = CriarRegião (Nome, Tamanho)`
- `Apont = AssociarRegião (Nome)`
- `EliminarRegião (Nome)`

### São necessários mecanismos de sincronização para:

- Garantir exclusão mútua sobre a zona partilhada
- Sincronizar a cooperação dos processos produtor e consumidor (ex. produtor-consumidor ou leitores-escretores)

## Objecto de Comunicação do Sistema

- `IdCanal = CriarCanal(Nome)`
- `IdCanal = AssociarCanal (Nome)`
- `EliminarCanal (IdCanal)`
- `Enviar (IdCanal, Mensagem, Tamanho)`
- `Receber (IdCanal, *Buffer, TamanhoMax)`

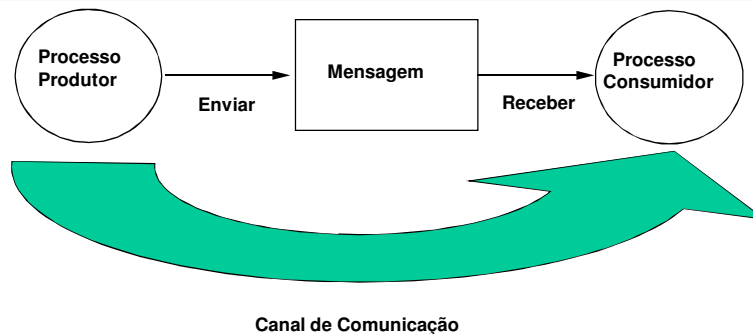
**Não são necessários mecanismos de sincronização adicionais porque são implementados pelo núcleo do sistema operativo**

## Comparação: memória partilhada vs. cópia através do núcleo

- **Memória partilhada:**
  - mecanismo mais eficiente
  - a sincronização tem de ser explicitamente programada
  - programação complexa
- **Objecto de Comunicação Sistema:**
  - velocidade de transferência limitada pelas duas cópias da informação e pelo uso das chamadas sistema para `Enviar` e `Receber`
  - sincronização implícita
  - fácil de utilizar

O resto do capítulo foca-se em **canais com cópia através do núcleo**

## Comunicação entre Processos



- generalização do modelo de cooperação entre processos



## Características do Canal

- Nomes dos objectos de comunicação
- Tipo de ligação entre o emissor e o receptor
- Estrutura das mensagens
- Capacidade de armazenamento
- Sincronização
  - no envio
  - na recepção
- Segurança – protecção envio/recepção
- Fiabilidade

## Ligação

- Antes de usar um canal de comunicação um processo tem de saber se existe e depois indicar ao sistema que se pretende associar
- Este problema decompõe-se em dois
  - Nomes dos canais de comunicação
  - Funções de associação e respectivo controlo de segurança

## Nomes dos objectos de comunicação

- Podemos ter duas soluções para os nomes
- Dar nomes explícitos aos canais
  - o espaço de nomes é gerido pelo sistema operativo e pode assumir diversas formas (cadeias de caracteres, números inteiros, endereços estruturados, endereços de transporte das redes)
    - Enviar ( IdCanal, mensagem )
    - Receber ( IdCanal, \*buffer )
  - É o mais frequente e muitas vezes baseia-se na gestão de nomes do sistema de ficheiros
- Os processos terem implicitamente associado um canal de comunicação
  - o canal é implicitamente identificado usando os identificadores dos processos
    - Enviar ( IdProcessoConsumidor, mensagem )
    - Receber ( IdProcessoProdutor, \*buffer )
  - Pouco frequente – ex.: enviar mensagens para janelas em Windows

## Ligação – função de associação

- Para usar um canal já existente um processo tem de se lhe associar
- Esta função é muito semelhante ao *open* de um ficheiro
- Tal como no *open* o sistema pode validar os direitos de utilização do processo, ou seja, se o processo pode enviar (escrever) ou receber (ler) mensagens

## Sincronização

- Sincronização (envio de mensagem):
  - **assíncrona** – o cliente envia o pedido e continua a execução
  - **síncrona** (rendez-vous) – o cliente fica bloqueado até que o processo servidor leia a mensagem
  - **cliente/servidor** – o cliente fica bloqueado até que o servidor envie uma mensagem de resposta
- Sincronização (recepção de mensagem):
  - bloqueante na ausência de mensagens, a mais frequente
  - Testa se há mensagens e retorna
- Capacidade de Armazenamento de Informação do canal
  - um canal pode ou não ter capacidade para memorizar várias mensagens
  - o armazenamento de mensagens num canal de comunicação permite desacoplar os ritmos de produção e consumo de informação, tornando mais flexível a sincronização

## Estrutura da informação trocada

- Fronteiras das mensagens
  - mensagens individualizadas
  - sequência de octetos (*byte stream*, vulgarmente usada nos sistemas de ficheiros e interfaces de E/S)
- Formato
  - Opacas para o sistema - simples sequência de octetos
  - Estruturada - formatação imposta pelo sistema
  - Formatada de acordo com o protocolo das aplicações

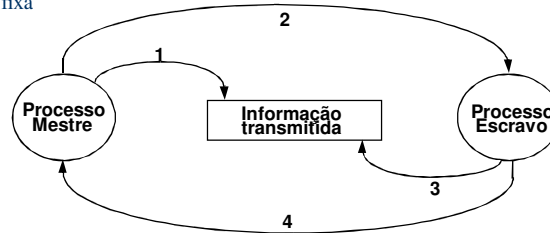
## Direccionalidade da comunicação

- A comunicação nos canais pode ser unidireccional ou bidireccional
  - Unidireccional o canal apenas permite enviar informação num sentido que fica definido na sua criação
    - Normalmente neste tipo de canais são criados dois para permitir a comunicação bidireccional. Ex.: *pipes*
  - Bidireccional o canal permite enviar mensagens nos dois sentidos
    - Ex.: *sockets*

## Modelos de Comunicação

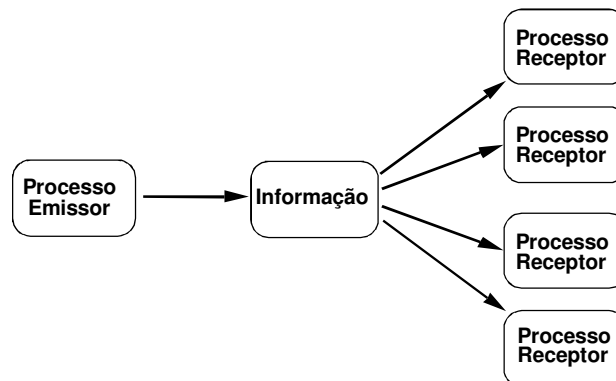
## Comunicação Mestre-Escravo

- o mestre não necessita de autorização para utilizar o escravo
- a actividade do processo escravo é controlada pelo processo mestre
- a ligação entre emissor e receptor é fixa



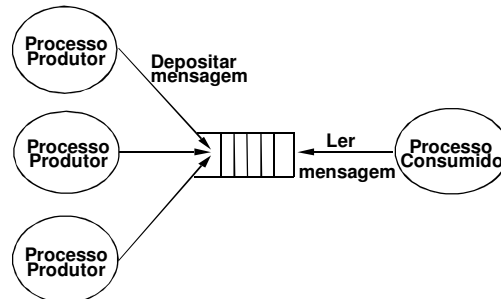
- Etapas:
  - 1 - informação para o processo escravo
  - 2 - assinalar ao escravo a existência de informação para tratar
  - 3 - leitura e eventualmente escrita de informação para o processo mestre
  - 4 - assinalar ao mestre o final da operação

## Difusão da Informação



## Correio (canal sem ligação)

- os processos emissores não controlam directamente a actividade do receptor ou receptores
- a ligação efectua-se indirectamente através das caixas de correio não existe uma ligação directa entre os processos
- a caixa de correio permite memorizar as mensagens quando estas são produzidas mais rapidamente do que consumidas



```
idCC = CriarCCorreio (Nome, parametros)
idCC = AssociarCCorreio (Nome, modo)
EliminarCCorreio (Nome)
```

## Programação da Caixa de Correio

**cliente**

```
#define NMax 10

typedef char TMensagem[NMax];
IdCC CCliente, CServidor;
TMensagem Mens;

/* Cria a mensagem de pedido do serviço o qual contém o identificador da caixa de correio de resposta */
void PreencheMensagem(TMensagem MS) {}

/* Processa a mensagem de resposta */
void ProcessaResposta(TMensagem MS) {}

void main() {
    CCliente=CriarCorreio("Cliente");
    CServidor=AssociarCorreio("Servidor");

    for (;;) {
        PreencheMensagem (Mens);
        Enviar (CServidor, Mens);
        Receber (CCliente, Mens);
        ProcessaResposta (Mens);
    }
}
```

**servidor**

```
#define NMAX 10
#define NCNome 64

typedef char TMensagem[NMAX];
typedef char Nome[NCNome];
IdCC CResposta, CServidor;
TMensagem Mens;
Nome NomeCliente;

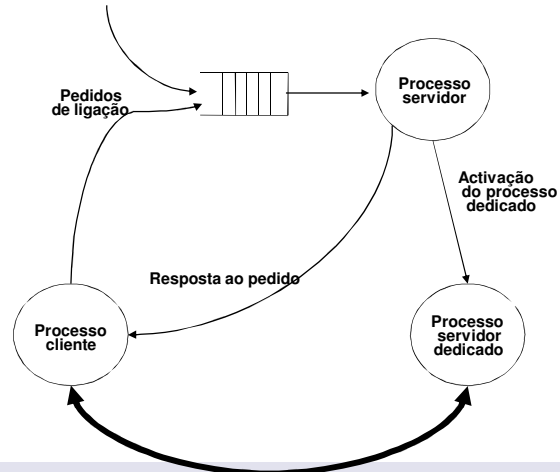
/* Trata a mensagem e devolve o nome da caixa de correio do cliente enviada na mensagem inicial */
void TrataMensagem (TMensagem Ms, Nome NomeCliente) {}

void main () {
    CServidor=CriarCorreio("Servidor");

    for (;;) {
        Receber (CServidor, Mens);
        TrataMensagem (Mens, NomeCliente);
        CResposta=AssociarCCorreio (NomeCliente);
        Enviar (CResposta, Mens);
        EliminarCC(CResposta);
    }
}
```

## Canal com ligação Modelo de Diálogo

- É estabelecido um canal de comunicação entre o processo cliente e o servidor
- O servidor pode gerir múltiplos clientes, mas dedica a cada um deles uma actividade independente
- O servidor pode ter uma política própria para atender os clientes



## Diálogo

### Servidor

- Primitiva para Criação de Canal  
`IdCanal = CriarCanal (Nome);`
- Primitivas para Aceitar/Desligar/Eliminar Ligações  
`IdCanal= AceitarLigacao (IdCanServidor);`  
`Desligar (IdCanal);`  
`Eliminar (Nome);`

### Cliente

- Primitivas par Associar/Desligar ao Canal  
`IdCanal:= PedirLigacao (Nome);`  
`Desligar (IdCanal);`

## Modelo de Diálogo - Canal com ligação

### Cliente

```
IdCanal Canal;
int Ligado;

void main() {
  while (TRUE) {
    Canal=PedirLigacao("Servidor");
    Ligado = TRUE;

    while (Ligado) {
      ProduzInformacao(Mens);
      Enviar(Canal, Mens);
      Receber(Canal, Mens);
      TratarInformacao(Mens);
    }
    TerminarLigacao(Canal);
  }
  exit(0);
}
```

### Servidor

```
IdCanal CanalServidor, CanalDialogo;

void main() {
  CanalPedido=CriarCanal("Servidor");

  for (;;) {
    CanalDialogo=AceitarLigacao(CanalPedido);
    CriarProcesso(TrataServico, CanalDialogo);
  }
}
```

## Muitos-para-muitos

- Transferência assíncrona de informação (mensagens) de vários processos produtores para um canal de comunicação associado a múltiplos processos consumidor



WIKIPEDIA  
*The Free Encyclopedia*



## Unix– Modelo Computacional - IPC

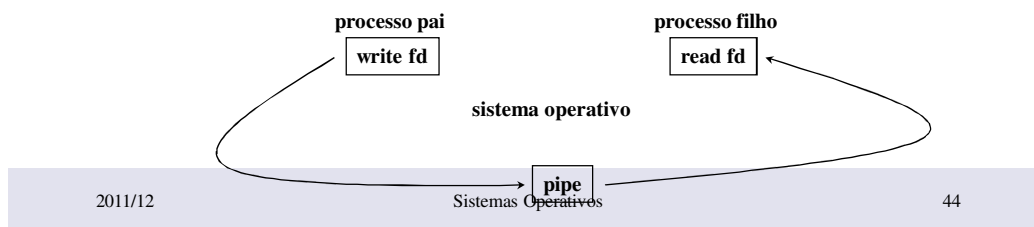
pipes  
sockets  
IPC sistema V

## IPC no UNIX

- Mecanismo inicial:
  - pipes
- Extensão dos pipes:
  - pipes com nome
- Evolução do Unix BSD 4.2:
  - sockets
- Unix sistema V:
  - regiões de memória partilhada
  - semáforos
  - caixas de correio

## Pipes

- Mecanismo original do Unix para comunicação entre processos.
- Têm uma interface idêntica à dos ficheiros
- Constitui um dos conceitos unificadores na estrutura do interpretador de comandos
- Canal (*byte stream*) ligando dois processos
- Permite um fluxo de informação unidireccional, um processo escreve num pipe e o correspondente lê na outra extremidade – modelo um para um
- Não tem nome lógico associado
- As mensagens são sequências de bytes de qualquer dimensão



## Pipes (2)

```
int pipe (int *fds);
```

```
fds[0] - descritor aberto para leitura
fds[1] - descritor aberto para escrita
```

- Os descritores de um pipe são análogos ao dos ficheiros
- As operações de read e write sobre ficheiros são válidas para os pipes
- Os descritores são locais a um processo podem ser transmitidos para os processos filhos através do mecanismo de herança
- O processo fica bloqueado quando escreve num pipe cheio
- O processo fica bloqueado quando lê de um pipe vazio

## DUP – System Call

### NAME

dup - duplicate an open file descriptor

### SYNOPSIS

```
#include <unistd.h>
int dup(int fildes);
```

### DESCRIPTION

The dup() function returns a new file descriptor having the following in common with the original open file descriptor fildes:

- same open file (or pipe)
- same file pointer (that is, both file descriptors share one file pointer)
- same access mode (read, write or read/write)

The new file descriptor is set to remain open across exec functions (see fcntl(2)).

The file descriptor returned is the lowest one available.

The dup(fildes) function call is equivalent to: fcntl(fildes, F\_DUPFD, 0)

## Redireccionamento de Entradas/Saídas

```
#include <stdio.h>
#include <fcntl.h>

#define TAMMSG 100
char msg[] = "mensagem de teste";
char tmp[TAMMSG];

main() {
    int fds[2], pid_filho;

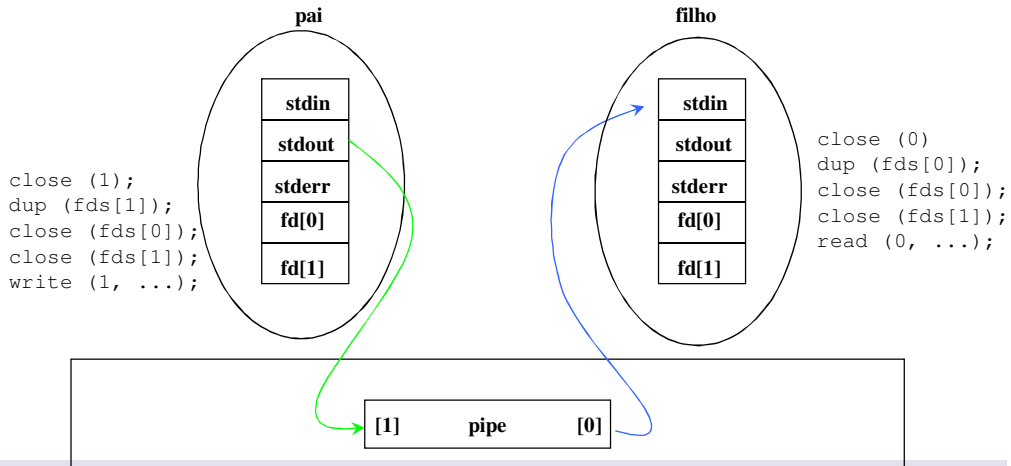
    if (pipe (fds) < 0) exit(-1);
    if (fork () == 0) {
        /* processo filho */
        /* liberta o stdin (posição zero) */
        close (0);

        /* redirecciona o stdin para o pipe de
        leitura */
        dup (fds[0]);
```

```
/* fecha os descritores não usados pelo
filho */
    close (fds[0]);
    close (fds[1]);

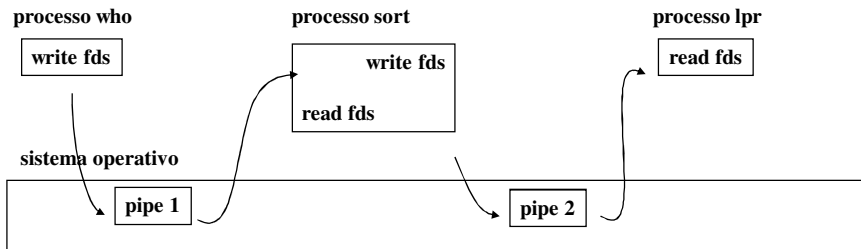
    /* lê do pipe */
    read (0, tmp, sizeof (msg));
    printf ("%s\n", tmp);
    exit (0);
}
else {
    /* processo pai */
    /* escreve no pipe */
    write (fds[1], msg, sizeof (msg));
    pid_filho = wait();
}
}
```

## Redirecionamento de Entradas/Saídas (2)



## Redirecionamento de Entradas/Saídas no Shell

**exemplo:**  
**who | sort | lpr**



## popen

### NAME

**popen, pclose** - initiate a pipe to or from a process

### SYNOPSIS

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *mode);
```

```
int pclose(FILE *stream);
```

### DESCRIPTION

The `popen()` function creates a pipe between the calling program and the command to be executed. The arguments to `popen()` are pointers to null-terminated strings. The `command` argument consists of a shell command line.

The `mode` argument is an I/O mode, either `r` for reading or `w` for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is `w`, by writing to the file stream (see `intro(3)`); and one can read from the standard output of the command, if the I/O mode is `r`, by reading from the file stream. Because open files are shared, a type `r` command may be used as an input filter and a type `w` as an output filter.

The environment of the executed command will be as if a child process were created within the `popen()` call using `fork()`. The child is invoked with the call: `execl("/usr/bin/ksh", "ksh", "-c", command, (char *)0)`; otherwise, the child is invoked with the call:

A stream opened by `popen()` should be closed by `pclose()`, which closes the pipe, and waits for the associated process to terminate and returns the termination status of the process running the command language interpreter.

## popen (1)

### EXAMPLE

The following program will print on the standard output (see `stdio(3S)`) the names of files in the current directory with a `.c` suffix.

```
#include <stdio.h>
#include <stdlib.h>
main() {
    char *cmd = "/usr/bin/ls *.c";
    char buf[BUFSIZ];
    FILE *ptr;

    if ((ptr = popen(cmd, "r")) != NULL)
        while (fgets(buf, BUFSIZ, ptr) != NULL)
            printf("%s", buf);
    return 0;
}
```



## Named Pipes

- um named pipe é um canal :
  - unidireccional
  - interface sequência de caracteres (byte stream)
  - um processo associa-se com a função open
  - é eliminado com a função unlink
  - o envio de informação é efectuado com a função write
  - a leitura da informação é efectuada com a função read
- A função mknod ou mkfifo permite criar ficheiros com características especiais e serve para criação dos named pipes.
 

```
int mknod (char *pathname, int mode)
```

2011/12

Sistemas Operativos

59

<pre> /* Cliente */ #include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt; #include &lt;fcntl.h&gt; #define TAMMSG 1000  void produzMsg (char *buf) {     strcpy (buf, "Mensagem de teste"); }  void trataMsg (buf) {     printf ("Recebeu: %s\n", buf); }  main() {     int fcli, fserv;     char buf[TAMMSG];      if ((fserv = open ("/tmp/servidor", O_WRONLY)) &lt; 0) exit (-1);     if ((fcli = open ("/tmp/cliente", O_RDONLY)) &lt; 0) exit (-1);      produzMsg (buf);     write (fserv, buf, TAMMSG);     read (fcli, buf, TAMMSG);     trataMsg (buf);      close (fserv);     close (fcli); } </pre>	<pre> /* Servidor */ #include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt; #include &lt;fcntl.h&gt;  #define TAMMSG 1000  main () {     int fcli, fserv, n;     char buf[TAMMSG];      unlink ("/tmp/servidor");     unlink ("/tmp/cliente");      if (mkfifo ("/tmp/servidor", 0777) &lt; 0) exit (-1);     if (mkfifo ("/tmp/cliente", 0777) &lt; 0) exit (-1);      if ((fserv = open ("/tmp/servidor", O_RDONLY)) &lt; 0) exit (-1);     if ((fcli = open ("/tmp/cliente", O_WRONLY)) &lt; 0) exit (-1);      for (;;) {         n = read (fserv, buf, TAMMSG);         if (n &lt;= 0) break;         trataPedido (buf);         n = write (fcli, buf, TAMMSG);     }     close (fserv);     close (fcli);     unlink ("/tmp/servidor");     unlink ("/tmp/cliente"); } </pre>
---	---

## Sockets

- Interface de programação para comunicação entre processos introduzida no Unix 4.2 BSD
- Objectivos:
  - independente dos protocolos
  - transparente em relação à localização dos processos
  - compatível com o modelo de E/S do Unix
  - eficiente

## Domínio e Tipo de Sockets

- Domínio do socket - define a família de protocolos associada a um socket:
  - Internet: família de protocolos Internet
  - Unix: comunicação entre processos da mesma máquina
  - outros...
- Tipo do socket - define as características do canal de comunicação:
  - stream: canal com ligação, bidireccional, fiável, interface tipo sequência de octetos
  - datagram: canal sem ligação, bidireccional, não fiável, interface tipo mensagem
  - raw: permite o acesso directo aos níveis inferiores dos protocolos (ex: IP na família Internet)



## Domínio e Tipo de Sockets (2)

- Relação entre domínio, tipo de socket e protocolo:

tipo \ domínio	AF_UNIX	AF_INET	AF_NS
SOCK_STREAM	SIM	TCP	SPP
SOCK_DGRAM	SIM	UDP	IDP
SOCK_RAW	-	IP	SIM
SOCK_SEQPACKET	-	-	SPP

## Interface Sockets: definição dos endereços

```

/* ficheiro <sys/socket.h> */
struct sockaddr {
    u_short family; /* definição do domínio (AF_XX) */
    char sa_data[14]; /* endereço específico do domínio*/
};

/* ficheiro <sys/un.h> */
struct sockaddr_un {
    u_short family; /* definição do domínio (AF_UNIX) */
    char sun_path[108]; /* nome */
};

```

```

/* ficheiro <netinet/in.h> */
struct in_addr {
    u_long addr; /* Netid+Hostid */
};

struct sockaddr_in {
    u_short sin_family; /* AF_INET */
    u_short sin_port; /* número do porto -
    16 bits*/
    struct in_addr sin_addr; /*
    Netid+Hostid */
    char sin_zero[8]; /* não utilizado*/
};

```

**struct sockaddr\_un**

<b>family</b>
<b>pathname</b> (up to 108 bytes)

**struct sockaddr\_in**

<b>family</b>
<b>2-byte port</b>
<b>4-byte net ID, host ID</b> (unused)

## Interface Sockets: criação de um socket e associação de um nome

- Criação de um socket:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int dominio, int tipo, int protocolo);
```

- domínio: AF\_UNIX, AF\_INET
- tipo: SOCK\_STREAM, SOCK\_DGRAM
- protocolo: normalmente escolhido por omissão
- resultado: identificador do socket (sockfd)

- Um socket é criado sem nome

- A associação de um nome (endereço de comunicação) a um socket já criado é feito com a chamada bind:

```
int bind(int sockfd, struct sockaddr *nome, int dim)
```

## Sockets com e sem Ligação

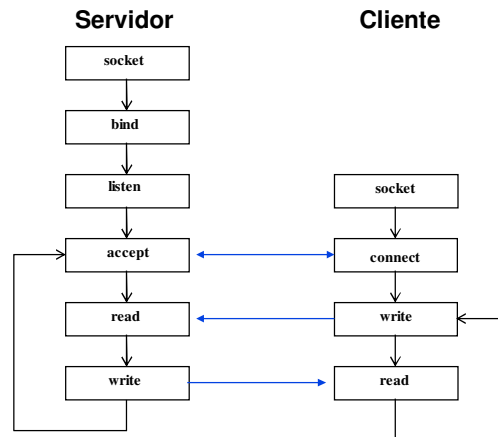
- Sockets com ligação:

- Modelo de comunicação tipo diálogo
- Canal com ligação, bidireccional, fiável, interface tipo sequência de octetos

- Sockets sem ligação:

- Modelo de comunicação tipo correio
- Canal sem ligação, bidireccional, não fiável, interface tipo mensagem

## Sockets com Ligação



## Sockets com Ligação

- **listen** - indica que se vão receber ligações neste socket:
  - `int listen (int sockfd, int maxpendentes)`
- **accept** - aceita uma ligação:
  - espera pelo pedido de ligação
  - cria um novo socket
  - devolve:
    - identificador do novo socket
    - endereço do interlocutor
  - `int accept(int sockfd, struct sockaddr *nome, int *dim)`
- **connect** - estabelece uma ligação com o interlocutor cujo endereço é nome:
  - `int connect (int sockfd, struct sockaddr *nome, int dim)`

## unix.h e inet.h

### unix.h

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define UNIXSTR_PATH
"/tmp/s.unixstr"
#define UNIXDG_PATH
"/tmp/s.unixdgx"
#define UNIXDG_TMP
"/tmp/dgXXXXXXXX"
```

### inet.h

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#define SERV_UDP_PORT 6600
#define SERV_TCP_PORT 6601
#define SERV_HOST_ADDR "193.136.128.20"
/* endereço do servidor */
#define SERV_HOSTNAME "mega"
/* nome do servidor */
```

## Exemplo

- Servidor de eco
- Sockets no domínio Unix
- Sockets com ligação

## Cliente STREAM AF\_UNIX

```
/* Cliente do tipo socket stream.
#include "unix.h"
main(void) {
    int sockfd, servlen;
    struct sockaddr_un serv_addr;

/* Cria socket stream */
    if ((sockfd= socket(AF_UNIX, SOCK_STREAM, 0) ) < 0)
        err_dump("client: can't open stream socket");

/* Primeiro uma limpeza preventiva */
    bzero((char *) &serv_addr, sizeof(serv_addr));
/* Dados para o socket stream: tipo + nome que
identifica o servidor */
    serv_addr.sun_family = AF_UNIX;
    strcpy(serv_addr.sun_path, UNIXSTR_PATH);
    servlen = strlen(serv_addr.sun_path) +
        sizeof(serv_addr.sun_family);
```

## Cliente STREAM AF\_UNIX(2)

```
/* Estabelece uma ligação. Só funciona se o socket
tiver sido criado e o nome associado*/

    if(connect(sockfd, (struct sockaddr *) &serv_addr,
        servlen) < 0)
        err_dump("client: can't connect to server");

/* Envia as linhas lidas do teclado para o socket
*/
    str_cli(stdin, sockfd);

/* Fecha o socket e termina */
    close(sockfd);
    exit(0);
}
```

## Cliente STREAM AF\_UNIX (3)

```
#include <stdio.h>
#define MAXLINE 512

/*Lê string de fp e envia para
sockfd. Lê string de sockfd e envia
para stdout*/
```

```
str_cli(fp, sockfd)
FILE *fp;
int sockfd;
{
    int n;
    char sendline[MAXLINE],
    recvline[MAXLINE+1];

    while (fgets(sendline, MAXLINE, fp)
           != NULL) {
```

```
/* Envia string para sockfd.
Note-se que o \0 não é enviado */
n = strlen(sendline);
if (writen(sockfd, sendline, n) != n)
    err_dump("str_cli:writen error on socket");
```

```
/* Tenta ler string de sockfd.
Note-se que tem de terminar a string com \0 */
n = readline(sockfd, recvline, MAXLINE);
if (n<0) err_dump("str_cli:readline error");
recvline[n] = 0;
```

```
/* Envia a string para stdout */
fputs(recvline, stdout);
}
if (ferror(fp))
    err_dump("str_cli: error reading file");
}
```

## Servidor STREAM AF\_UNIX

```
/* Recebe linhas do cliente e reenvia-as para o cliente */
#include "unix.h"
```

```
main(void) {
    int sockfd, newsockfd, cliilen, childpid, servlen;
    struct sockaddr_un cli_addr, serv_addr;
```

```
/* Cria socket stream */
if ((sockfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
    err_dump("server: can't open stream socket");
```

```
/* Elimina o nome, para o caso de já existir.
/* O nome serve para que os clientes possam identificar o servidor */
bzero((char *)&serv_addr, sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, UNIXSTR_PATH);
servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);
if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
    err_dump("server, can't bind local address");
```

```
listen(sockfd, 5);
```

## Servidor STREAM AF\_UNIX (2)

```
for (;;) {
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
    if (newsockfd < 0) err_dump("server: accept error");

    /* Lança processo filho para tratar do cliente */
    if ((childpid = fork()) < 0) err_dump("server: fork error");
    else if (childpid == 0) {
        /* Processo filho.
        Fecha sockfd já que não é utilizado pelo processo filho
        Os dados recebidos do cliente são reenviados para o cliente
        */
        close(sockfd);
        str_echo(newsockfd);
        exit(0);
    }
    /* Processo pai. Fecha newsockfd que não utiliza */
    close(newsockfd);
}
2011/12                                     Sistemas Operativos                                     75
}
```

## Servidor STREAM AF\_UNIX (3)

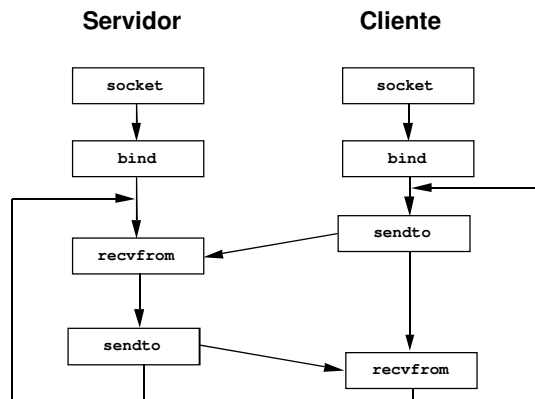
```
#define MAXLINE 512
/* Servidor do tipo socket stream. Reenvia as linhas recebidas para o cliente*/

str_echo(int sockfd)
{
    int n;
    char line[MAXLINE];

    for (;;) {
        /* Lê uma linha do socket */
        n = readline(sockfd, line, MAXLINE);
        if (n == 0) return;
        else if (n < 0) err_dump("str_echo: readline error");

        /* Reenvia a linha para o socket. n conta com o \0 da string,
        caso contrário perdia-se sempre um caracter! */
        if (writen(sockfd, line, n) != n)
            err_dump("str_echo: writen error");
    }
}
2011/12                                     Sistemas Operativos                                     76
}
```

## Sockets sem Ligação



## Sockets sem Ligação

- `sendto`: Envia uma mensagem para o endereço especificado

```
int sendto(int sockfd, char *mens, int dmens,
           int flag, struct sockaddr *dest, int *dim)
```

- `recvfrom`: Recebe uma mensagem e devolve o endereço do emissor

```
int recvfrom(int sockfd, char *mens, int dmens,
             int flag, struct sockaddr *orig, int *dim)
```



## Cliente DGRAM AF\_UNIX

```
#include "unix.h"
main(void) {
    int sockfd, cliilen, servlen;
    char *mktemp();
    struct sockaddr_un cli_addr, serv_addr;

    /* Cria socket datagram */
    if(( sockfd = socket(AF_UNIX, SOCK_DGRAM, 0) ) < 0)
        err_dump("client: can't open datagram socket");
    /* O nome temporário serve para ter um socket para resposta do
    servidor */
    bzero((char *) &cli_addr, sizeof(cli_addr));
    cli_addr.sun_family = AF_UNIX;
    mktemp(cli_addr.sun_path);
    cliilen = sizeof(cli_addr.sun_family) + strlen(cli_addr.sun_path);

    /* Associa o socket ao nome temporário */
    if (bind(sockfd, (struct sockaddr *) &cli_addr, cliilen) < 0)
        err_dump("client: can't bind local address");
}
```

## Cliente DGRAM AF\_UNIX(2)

```
/* Primeiro uma limpeza preventiva!
bzero((char *) &serv_addr, sizeof(serv_addr));

serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, UNIXDG_PATH);
servlen=sizeof(serv_addr.sun_family) +
        strlen(serv_addr.sun_path);

/* Lê linha do stdin e envia para o servidor. Recebe a linha do
servido e envia-a para stdout */
dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, servlen);

close(sockfd);
unlink(cli_addr.sun_path);
exit(0);
}
```

## Cliente DGRAM AF\_UNIX (3)

```
#include <stdio.h>
#define MAXLINE 512

/* Cliente do tipo socket datagram.
   Lê string de fp e envia para sockfd.
   Lê string de sockfd e envia para stdout */

#include <sys/types.h>
#include <sys/socket.h>
```

## Cliente DGRAM AF\_UNIX (4)

```
while (fgets(sendline, MAXLINE, fp) != NULL) {
    n = strlen(sendline);

    /* Envia string para sockfd. Note-se que o \0 não é enviado */
    if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n)
        err_dump("dg_cli: sendto error on socket");

    /* Tenta ler string de sockfd. Note-se que tem de
       terminar a string com \0 */
    n = recvfrom(sockfd, recvline, MAXLINE, 0,
                 (struct sockaddr *) 0, (int *) 0);
    if (n < 0) err_dump("dg_cli: recvfrom error");
    recvline[n] = 0;

    /* Envia a string para stdout */
    fputs(recvline, stdout);
}
if (ferror(fp)) err_dump("dg_cli: error reading file");
}
```

## Servidor DGRAM AF\_UNIX

```

/* Servidor do tipo socket datagram. Recebe linhas do cliente e devolve-as para o
cliente */
#include "unix.h"
main (void) {
    int sockfd, servlen;
    struct sockaddr_un serv_addr, cli_addr;

    /* Cria socket datagram */
    if ((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        err_dump("server: can't open datagram socket");

    unlink(UNIXDG_PATH);
    /* Limpeza preventiva*/
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sun_family = AF_UNIX;
    strcpy(serv_addr.sun_path, UNIXDG_PATH);
    servlen = sizeof(serv_addr.sun_family) + strlen(serv_addr.sun_path);
    /* Associa o socket ao nome */
    if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
        err_dump("server: can't bind local address");

    /* Fica à espera de mensagens do client e reenvia-as para o cliente */
    dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
}
2011/12                                     Sistemas Operativos                                     83

```

## Servidor DGRAM AF\_UNIX (3)

<pre> #define MAXLINE 512  /* Servidor do tipo socket datagram. Manda linhas recebidas de volta para o cliente */  #include &lt;sys/types.h&gt; #include &lt;sys/socket.h&gt; #define MAXMSG 2048  /* pcli_addr especifica o cliente */ dg_echo(sockfd, pcli_addr, maxclilen) int sockfd; struct sockaddr *pcli_addr; int maxclilen; { </pre>	<pre> int n, clilen; char mesg[MAXMSG];  for (;;) {     clilen = maxclilen;      /* Lê uma linha do socket */     n = recvfrom(sockfd, mesg, MAXMSG, 0, pcli_addr, &amp;clilen);     if (n &lt; 0)         err_dump("dg_echo: recvfrom error");      /*Manda linha de volta para o socket */     if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)         err_dump("dg_echo: sendto error"); } } </pre>
---	--

## Espera Múltipla com Select

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfd, fd_set* leitura, fd_set*
           escrita, fd_set* excepcao, struct timeval*
           alarme)
```

**select:**

- espera por um evento
- bloqueia o processo até que um descritor tenha um evento associado ou expire o alarme
- especifica um conjunto de descritores onde espera:
  - receber mensagens
  - receber notificações de mensagens enviadas (envios assíncronos)
  - receber notificações de acontecimentos excepcionais

## Select

- **exemplos de quando o select retorna:**
  - Os descritores (1,4,5) estão prontos para leitura
  - Os descritores (2,7) estão prontos para escrita
  - Os descritores (1,4) têm uma condição excepcional pendente
  - Já passaram 10 segundos

## Espera Múltipla com Select (2)

```
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
}
```

- esperar para sempre → parâmetro efectivo é null pointer)
- esperar um intervalo de tempo fixo → parâmetro com o tempo respectivo
- não esperar → parâmetro com o valor zero nos segundos e microssegundos
  
- as condições de excepção actualmente suportadas são:
  - chegada de dados out-of-band
  - informação de controlo associada a pseudo-terminais

## Manipulação do fd\_set

- Definir no select quais os descritores que se pretende testar
  - void FD\_ZERO (fd\_set\* fdset) - clear all bits in fdset
  - void FD\_SET (int fd, fd\_set\* fd\_set) - turn on the bit for fd in fdset
  - void FD\_CLR (int fd, fd\_set\* fd\_set) - turn off the bit for fd in fdset
  - int FD\_ISSET (int fd, fd\_set\* fd\_set) - is the bit for fd on in fdset?
- Para indicar quais os descritores que estão prontos, a função select modifica:
  - fd\_set\* leitura
  - fd\_set\* escrita
  - fd\_set\* excepcao

## Servidor com Select

<pre> /* Servidor que utiliza sockets stream e datagram em simultâneo. O servidor recebe caracteres e envia-os para stdout */  #include &lt;stdio.h&gt; #include &lt;sys/types.h&gt; #include &lt;sys/time.h&gt; #include &lt;sys/socket.h&gt; #include &lt;sys/un.h&gt; #include &lt;errno.h&gt;  #define MAXLINE 80 #define MAXSOCKS 32  #define ERRORMSG1 "server: cannot open stream socket" #define ERRORMSG2 "server: cannot bind stream socket" #define ERRORMSG3 "server: cannot open datagram socket" #define ERRORMSG4 "server: cannot bind datagram socket" #include "names.h" </pre>	<pre> int main(void) {     int strmfd, dgrmfd, newfd;     struct sockaddr_un         servstrmaddr, servdgrmaddr, clientaddr;     int len, clientlen;     fd_set testmask, mask;      /* Cria socket stream */     if ((strmfd=socket (AF_UNIX, SOCK_STREAM, 0))&lt;0) {         perror(ERRORMSG1);         exit(1);     }     bzero((char*)&amp;servstrmaddr,         sizeof(servstrmaddr));     servstrmaddr.sun_family = AF_UNIX;     strcpy(servstrmaddr.sun_path, UNIXSTR_PATH);     len = sizeof(servstrmaddr.sun_family)         +strlen(servstrmaddr.sun_path);     unlink(UNIXSTR_PATH);     if (bind(strmfd, (struct sockaddr *)&amp;servstrmaddr,         len)&lt;0)     {         perror(ERRORMSG2);         exit(1);     } } </pre>
<p>2011/12</p>	<p>Sist</p>

## Servidor com Select (2)

<pre> /*Servidor aceita 5 clientes no socket stream*/ listen(strmfd, 5); /* Cria socket datagram */ if ((dgrmfd = socket (AF_UNIX, SOCK_DGRAM, 0)) &lt; 0) {     perror(ERRORMSG3);     exit(1); } /*Inicializa socket datagram: tipo + nome */ bzero((char     *)&amp;servdgrmaddr, sizeof(servdgrmaddr)); servdgrmaddr.sun_family = AF_UNIX; strcpy(servdgrmaddr.sun_path, UNIXDG_PATH); len=sizeof(servdgrmaddr.sun_family)+     strlen(servdgrmaddr.sun_path); unlink(UNIXDG_PATH); if (bind(dgrmfd, (struct     *)&amp;servdgrmaddr, len)&lt;0) {     perror(ERRORMSG4);     exit(1); } </pre>	<pre> /* - Limpa-se a máscara - Marca-se os 2 sockets - stream e datagram. - A mascara é limpa pelo sistema de cada vez que existe um evento no socket. - Por isso é necessário utilizar uma mascara auxiliar */ FD_ZERO(&amp;testmask); FD_SET(strmfd, &amp;testmask); FD_SET(dgrmfd, &amp;testmask); </pre>
<p>2011/12</p>	<p>Sistemas Operativos 90</p>

## Servidor com Select (3)

```
for(;;) {
    mask = testmask;

    /* Bloqueia servidor até que se dê um evento. */
    select (MAXSOCKS, &mask, 0, 0, 0);

    /* Verificar se chegaram clientes para o socket stream */
    if (FD_ISSET (strmfd, &mask)) {
        /* Aceitar o cliente e associa-lo a newfd. */
        clientlen = sizeof (clientaddr);
        newfd = accept (strmfd, (struct sockaddr*)&clientaddr, &clientlen);
        echo (newfd);
        close (newfd);
    }

    /* Verificar se chegaram dados ao socket datagram. Ler dados */
    if (FD_ISSET (dgrmfd, &mask))
        echo (dgrmfd);

    /*Voltar ao ciclo mas não esquecer da mascara! */
}
}
```