



Comunicação por Troca de Mensagem entre Processos

Canal de comunicação
Arquitetura da comunicação
Modelos de comunicação

Sistemas Operativos

2019 - 2020

Dois paradigmas para programação concorrente

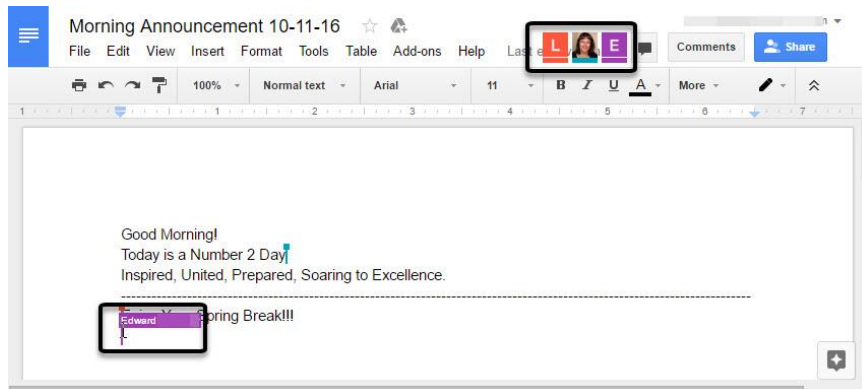
- Por memória partilhada
 - Tarefas partilham dados (no *heap*/amontoadado)
 - Troca de dados é feita escrevendo e lendo da memória partilhada
 - Sincronização recorre a mecanismos adicionais (p.e., trincos, semaforos,...).
- Por troca de mensagens
 - Cada tarefa trabalha exclusivamente sobre dados privados
 - Tarefas transmitem dados trocando mensagens
 - Mensagens também servem para sincronizar tarefas

Analogia:

Edição concorrente de um documento

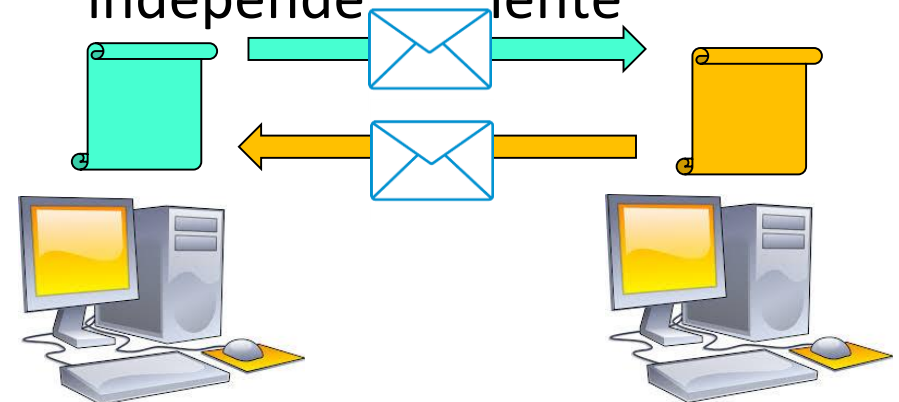
Memória partilhada

- Google docs
 - Única cópia online do documento
 - As alterações de um editor são imediatamente aplicadas ao documento partilhado e visíveis logo aos outros editores



Troca de mensagem

- Cada editor mantém uma cópia privada do documento no seu computador
- Alterações enviadas por email e aplicadas independentemente



Porquê diferentes paradigmas?

- Historicamente:
 - Algumas arquiteturas só permitiam que programas a correr em CPUs distintos trocassem mensagens
 - Cada CPU com a sua memória privada, interligados por alguma rede
 - Outras suportavam memória partilhada
 - E.g. CPUs podiam aceder à mesma memória RAM através de protocolo de coerência de cache

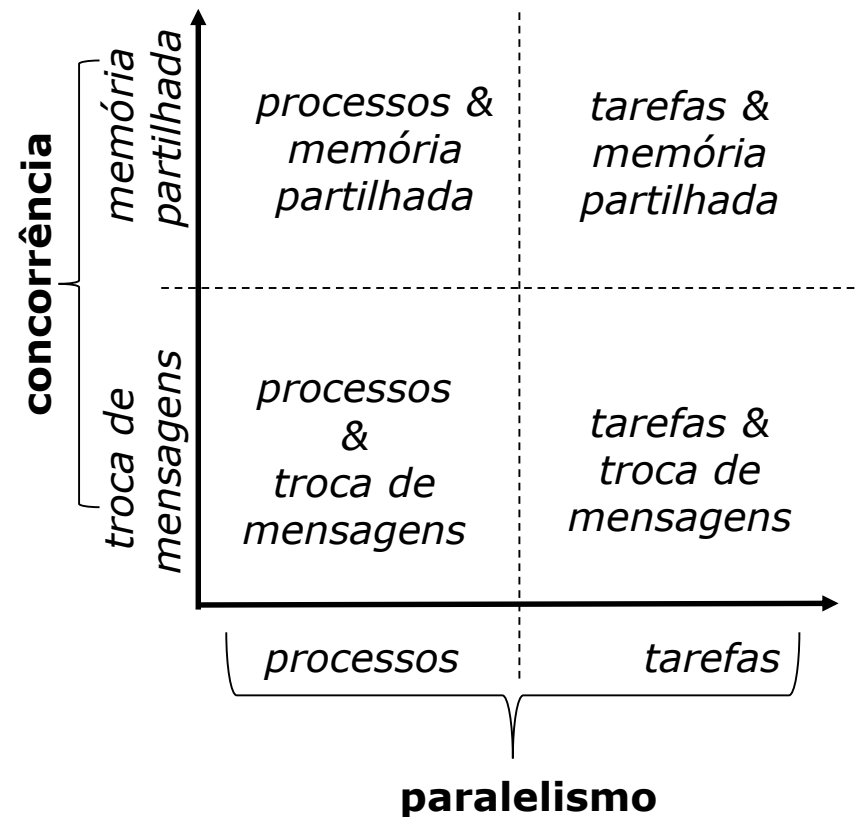


Porquê diferentes paradigmas?

- Estilos diferentes de programação, com virtudes e defeitos:
 - Diferentes ambientes de programação mais apropriados para cada paradigma
 - Preferências de cada programador
 - Alguns problemas mais fáceis de resolver eficientemente num paradigma que noutro
- Voltaremos a esta discussão daqui a umas semanas

Combinações de modelos de paralelismo e coordenação

- Dois modelos de paralelismo:
 1. por tarefa
 2. por processo
- Dois modelos de concorrência:
 1. por troca de mensagens
 2. por memória partilhada
- Os modelos de paralelismo e concorrência podem ser combinados!



O que construímos até agora...

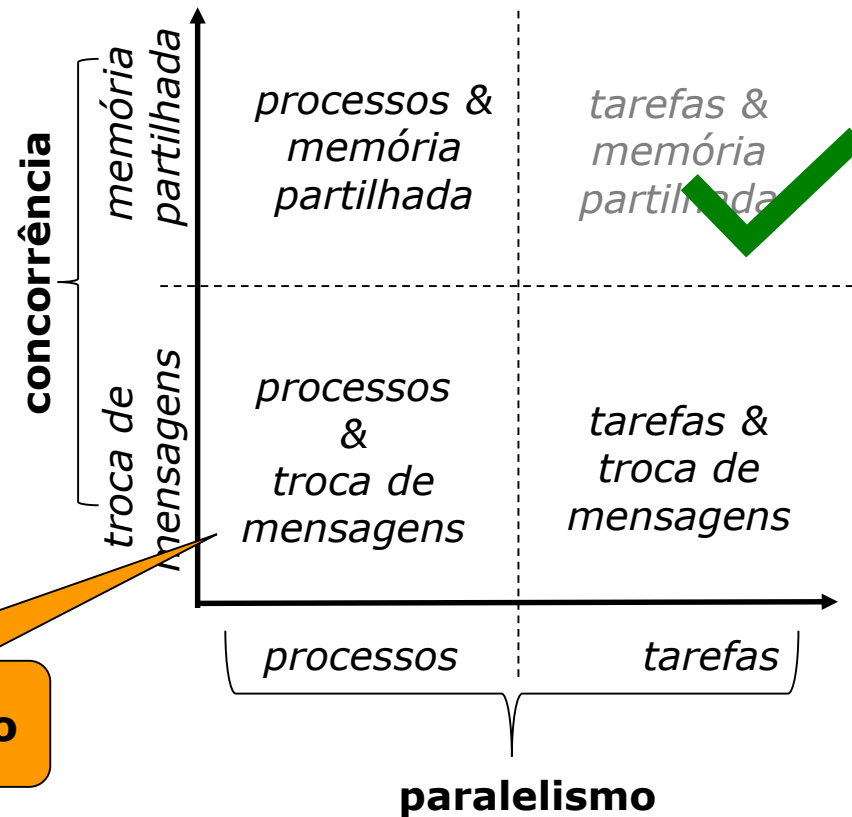


A abstração de processo

A possibilidade de ter
paralelismo e **partilha de**
7 **dados** dentro do processo

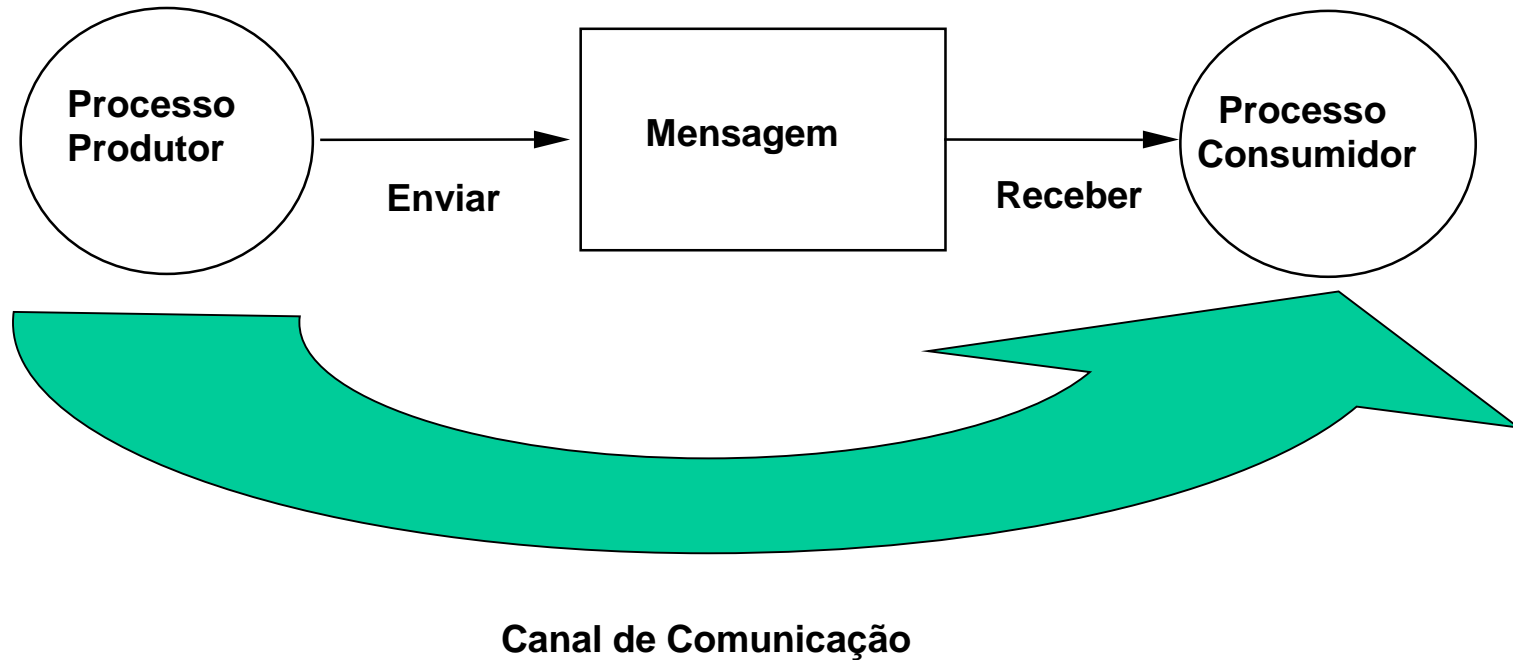


Combinações de modelos de paralelismo e coordenação



Próximo capítulo

Comunicação por Troca de Mensagem entre Processos



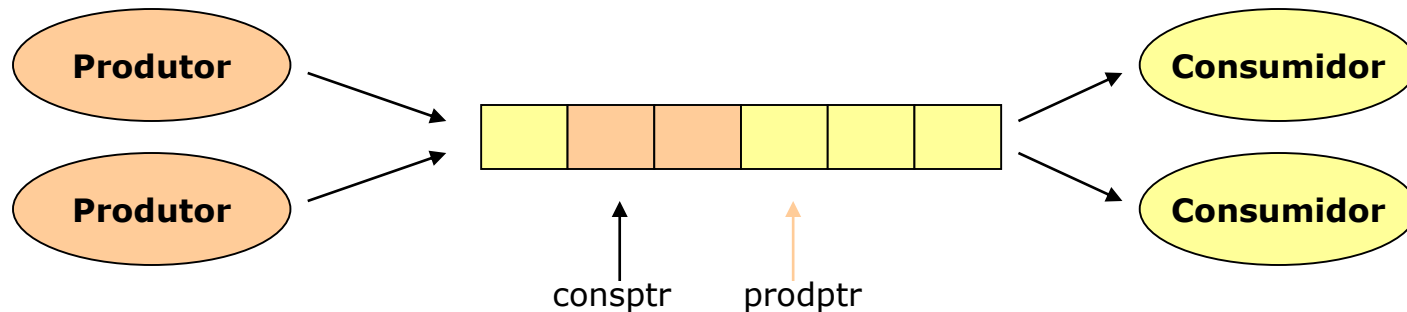
Exemplos

- A comunicação entre processos pode realizar-se no âmbito:
 - de uma única aplicação,
 - entre aplicações numa mesma máquina
 - entre máquinas interligadas por uma rede de dados
- Exemplos:
 - servidores de base de dados,
 - browser e servidor WWW,
 - cliente e servidor SSH,
 - cliente e servidor de e-mail,
 - nós BitTorrent



Como implementar comunicação entre processos?

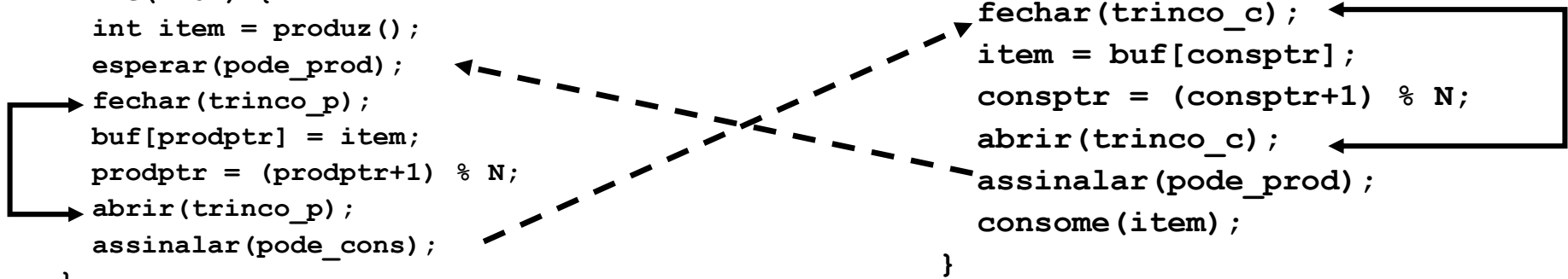
Exemplo de implementação de um canal de comunicação (solução para o problema do Produtor – Consumidor)



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

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

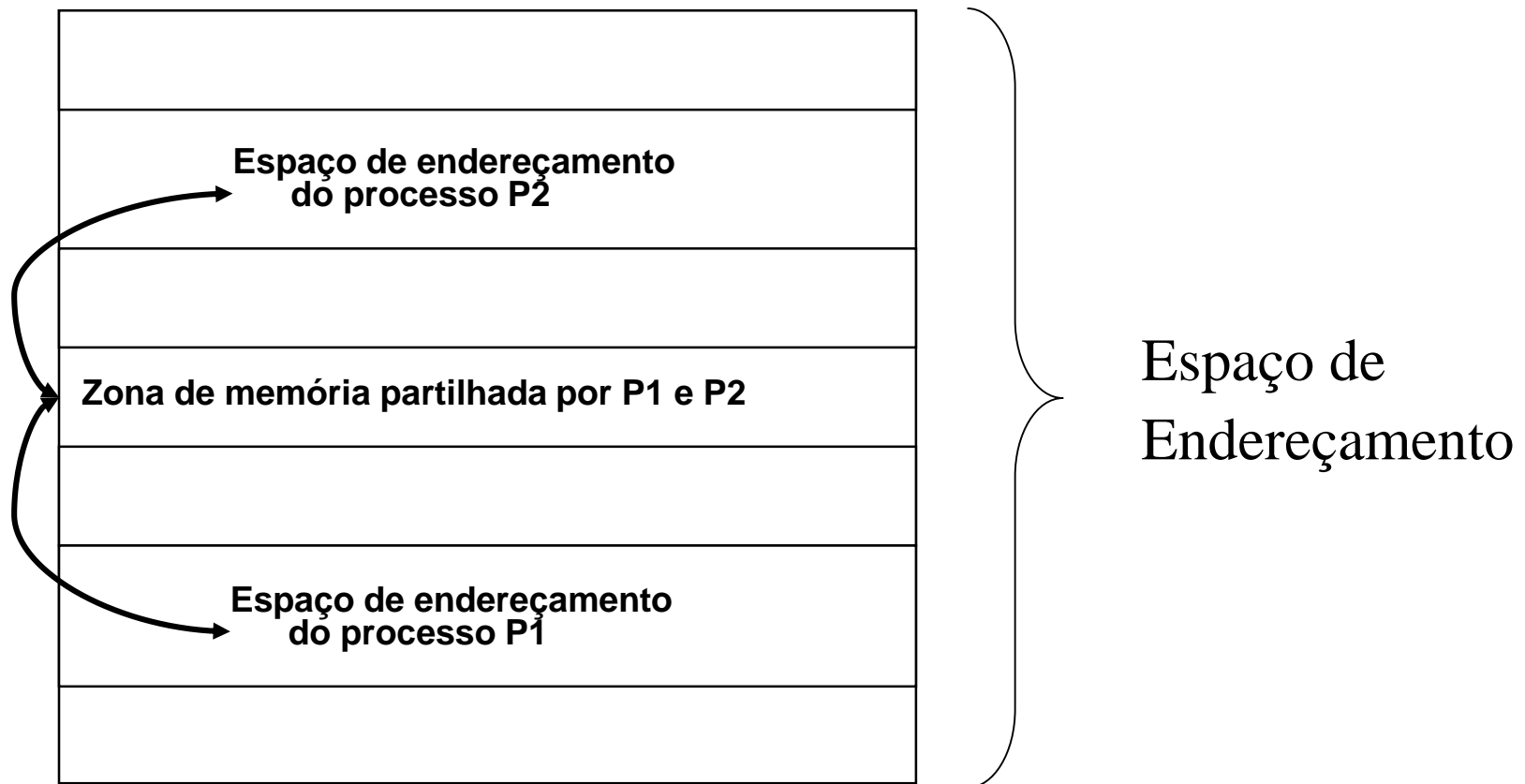
```
consumidor()
{
    while(TRUE) {
        int item;
        esperar(podede_cons);
        fechar(trinco_c);
        item = buf[consptr];
        consptr = (consptr+1) % N;
        abrir(trinco_c);
        assinalar(podede_prod);
        consome(item);
    }
}
```



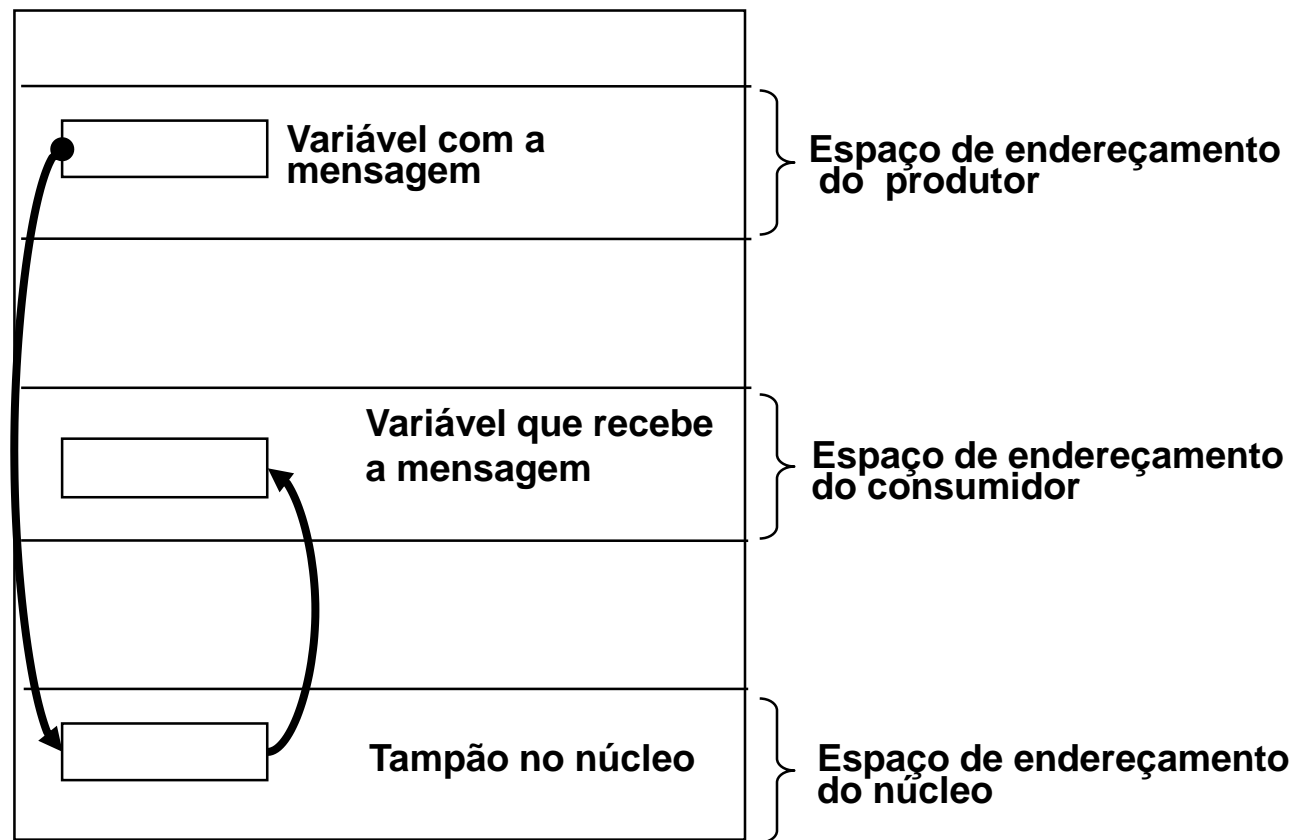
Implementação do Canal de Comunicação

- O canal de comunicação pode ser implementado a dois níveis:
 - **No núcleo do sistema operativo:** os dados são enviados/recebidos por chamadas sistema
 - **No *user level*:** os processos acedem a uma zona de **memória partilhada** entre ambos os processos comunicantes
 - Veremos mais à frente como isto é possível

Arquitetura da Comunicação: por memória partilhada



Arquitetura da Comunicação: cópia através do núcleo





Inicialmente, consideraremos apenas canais de comunicação implementados pelo núcleo do SO

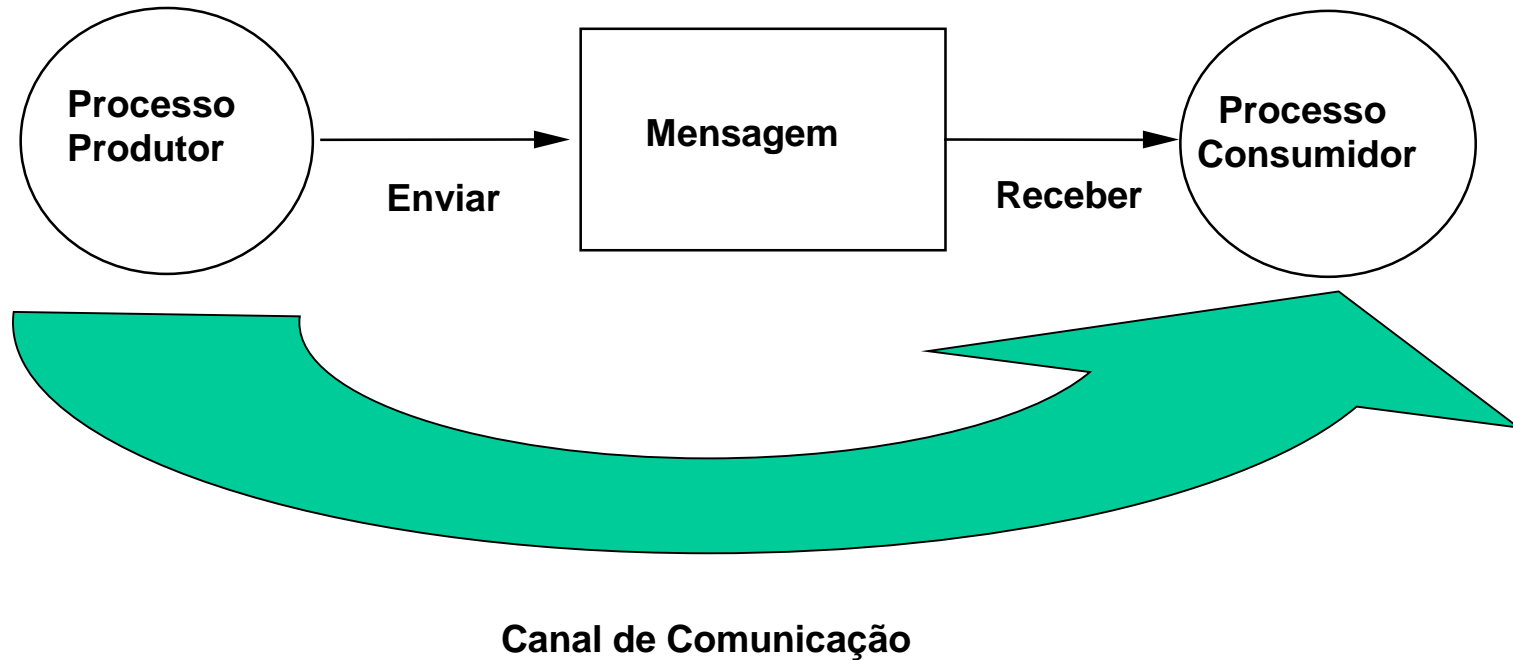


Objecto de Comunicação do Sistema

- `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

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: duas alternativas

- Dar nomes explícitos aos canais (*o mais frequente*)
 - O espaço de nomes é gerido pelo sistema operativo
 - Muitas vezes baseia-se na gestão de nomes do sistema de ficheiros
 - Pode assumir diversas formas: cadeias de caracteres, números inteiros, endereços estruturados, endereços de transporte das redes
 - Enviar (IdCanal, mensagem)
 - Receber (IdCanal, *buffer)
- Os processos terem implicitamente associado um canal de comunicação
 - Canal implicitamente identificado pelos 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: 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

- Assíncrona: testa se há mensagens e retorna
- Síncrona: bloqueante na ausência de mensagens (a mais frequente)



Sincronização:

Capacidade de armazenamento de informação do canal

- Um canal pode ou não ter capacidade para memorizar várias mensagens
 - Maior capacidade 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

Sequência de octetos vs. mensagens individuais

- Interface mensagens individuais
 - *receber* devolve uma mensagem que foi enviada numa chamada à função *enviar*
 - Se houver N chamadas à função *enviar*, para enviar N mensagens, é necessário N chamadas à função *receber*
 - Ou seja, as fronteiras de cada mensagem são preservadas
- Interface sequência de octetos:
 - Bytes das mensagens enviadas são acumulados no canal, suas fronteiras são esquecidas
 - Chamada à função *receber* pode devolver conteúdo vindo de múltiplas chamadas a *enviar*
 - Ou seja, fronteiras entre mensagens são perdidas no canal

Direccionalidade da comunicação

- 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
- Bidireccional - o canal permite enviar mensagens nos dois sentidos



Resumo do Modelo Computacional

- IDCanal = **CriarCanal** (Nome, Dimensão)
- IDCanal = **AssociarCanal** (Nome, Modo)
- **EliminarCanal** (IDCanal)
- **Enviar** (IDCanal, Mensagem, Tamanho)
- **Receber** (IDCanal, buffer, TamanhoMax)



Modelos de Comunicação

Modelos de Comunicação

- Com as funções do modelo computacional poderíamos criar qualquer tipo de estrutura de comunicação entre os processos.
- Contudo existem algumas que, por serem mais frequentes, correspondem a padrões que os programadores utilizam ou que o sistema operativo oferece directamente como canais nativos

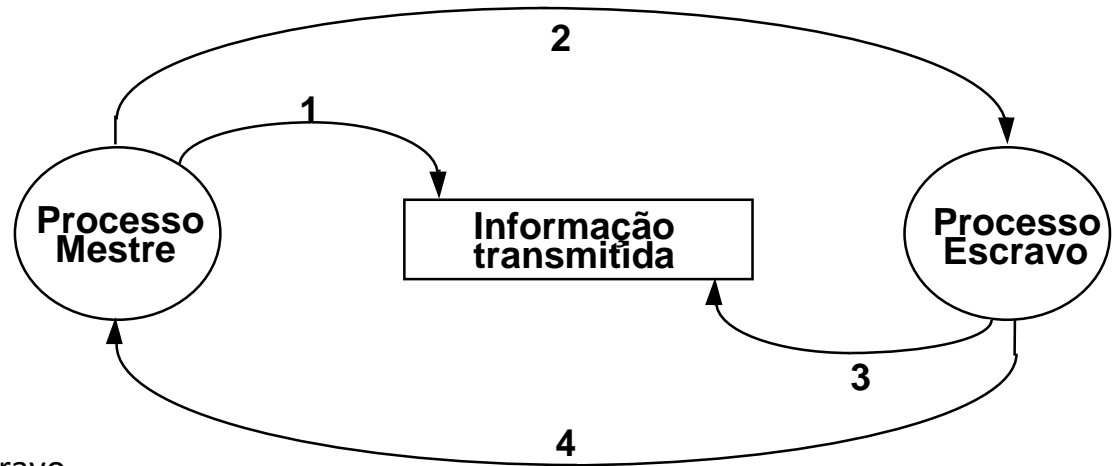


Modelos de Comunicação

- Um-para-Um (fixo)- Mestre/escravo:
 - O processo consumidor (escravo) tem a sua acção totalmente controlada por um processo produtor (mestre)
 - A ligação entre produtor consumidor é fixa
- Um-para-Muitos - Difusão:
 - Envio da mesma informação a um conjunto de processos consumidores
- Muitos-para-Um (caixa de correio, canal sem ligação):
 - Transferência assíncrona de informação (mensagens), de vários processos produtores, para um canal de comunicação associado a um processo consumidor
 - Os produtores não têm qualquer controlo sobre os consumidores/receptores
- Um-para-Um de vários (diálogo, canal com ligação):
 - Um processo pretende interactuar com outro, negociam o estabelecimento de um canal dedicado, mas temporário, de comunicação entre ambos.
 - Situação típica de cliente servidor
- Muitos-para-Muitos
 - Transferência assíncrona de informação (mensagens) de vários processos produtores para um canal de comunicação associado a múltiplos processos consumidor

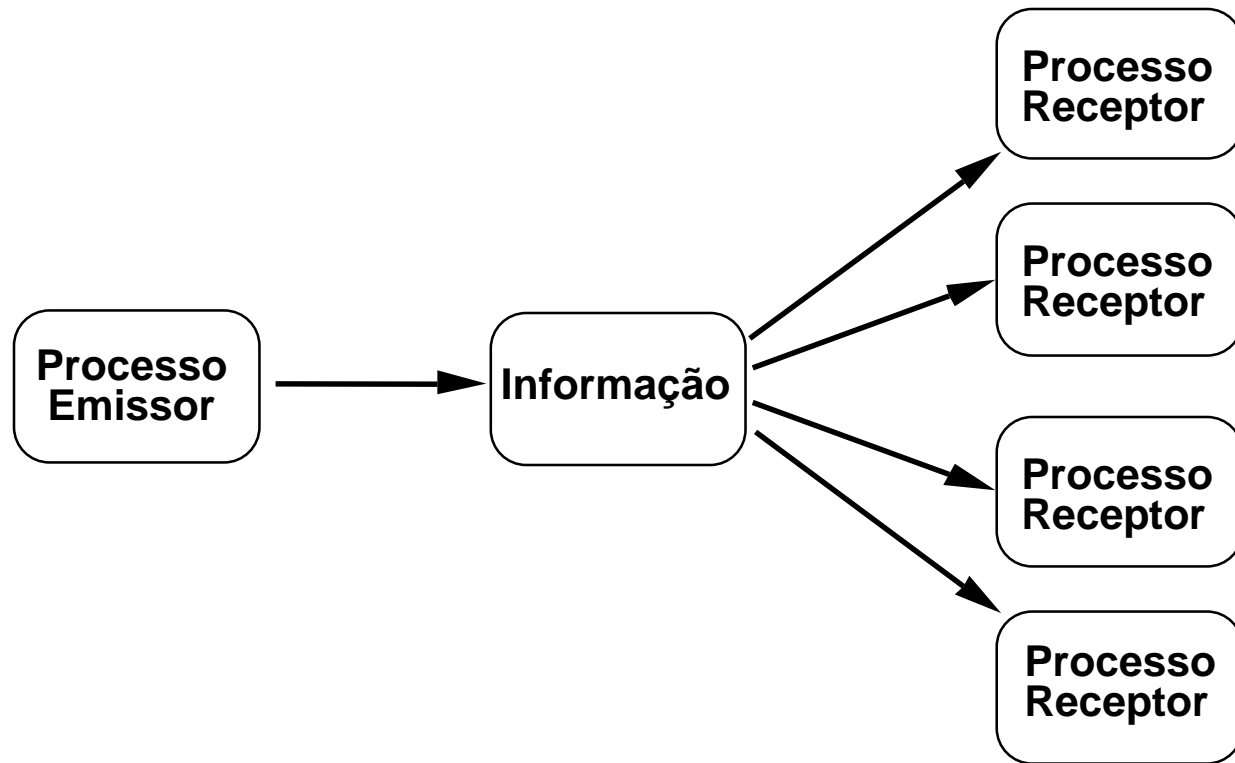
Comunicação Mestre-Escravo

- 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



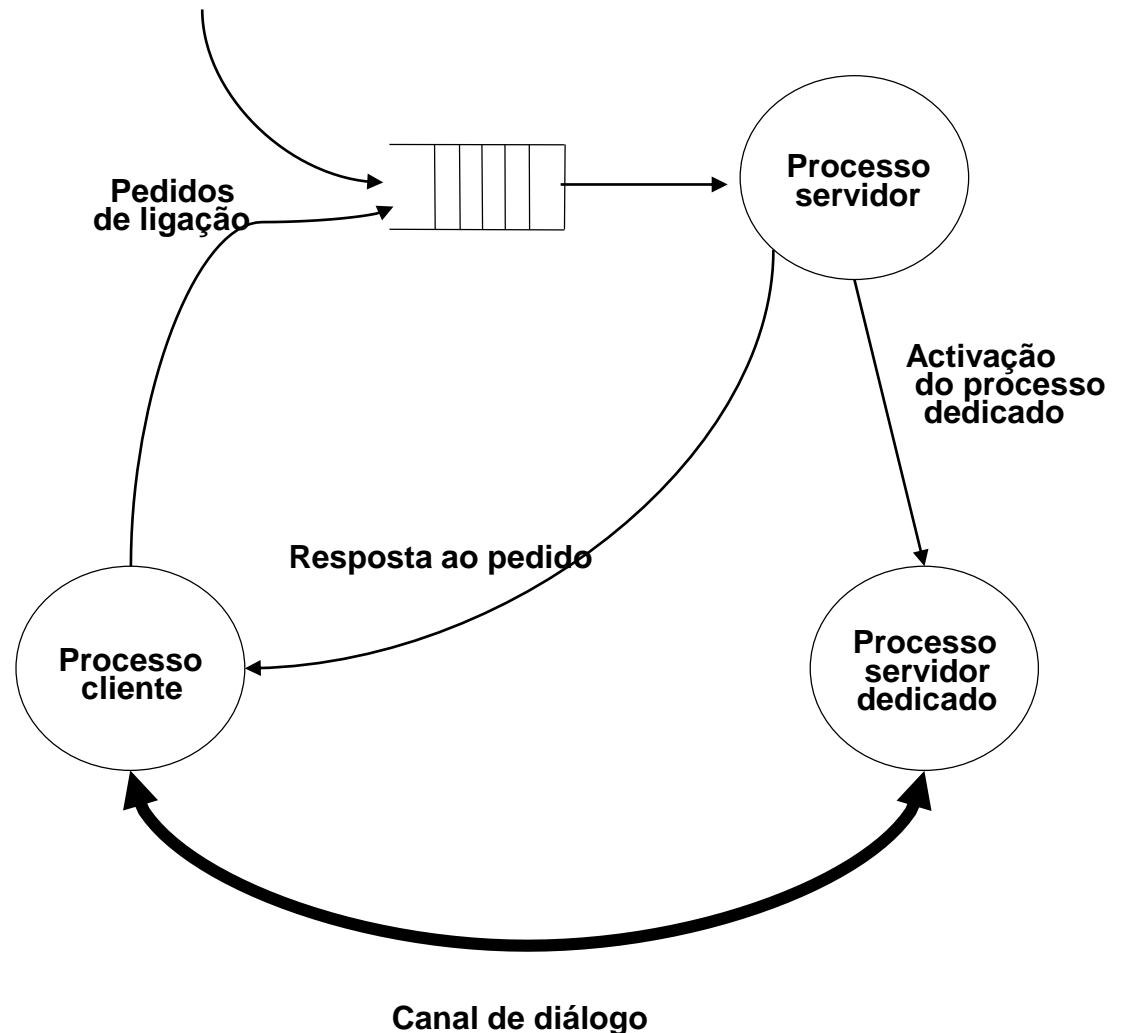
- Etapas:
 - 1 - informação para o processo escravo
 - 2 - assinalar ao escravo a existência de informação para tratar
 - 3 - leitura e eventualmente escrita de informação para o processo mestre
 - 4 - assinalar ao mestre o final da operação

Um-para-Muitos ou Difusão da Informação



Um-para-Um de vários ou 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





Muitos-para-muitos

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



Unix— Modelo Computacional - IPC

pipes
sockets
signals



Mecanismos de Comunicação em Unix

- No Unix houve uma tentativa de uniformização da interface de comunicação entre processos com a interface dos sistemas de ficheiros.
- Para perceber os mecanismos de comunicação é fundamental conhecer bem a interface com o sistema de ficheiros.



Sistema de Ficheiros

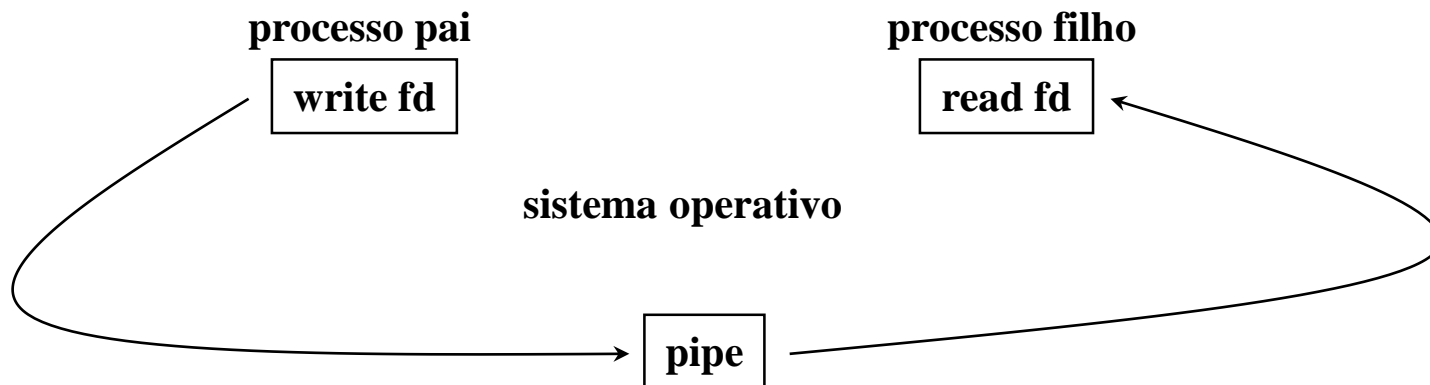
- Sistema de ficheiros hierarquizado
- Tipos de ficheiros:
 - Normais – sequência de octetos (bytes) sem uma organização em registos (records)
 - Ficheiros especiais – periféricos de E/S, pipes, sockets
 - Ficheiros directório
- Quando um processo se começa a executar o sistema abre três ficheiros especiais
 - `stdin` – input para o processo (fd – 0)
 - `stdout` – output para o processo (fd – 1)
 - `stderr` – periférico para assinalar os erros (fd – 2)
- Um file descriptor é um inteiro usado para identificar um ficheiro aberto (os valores variam de zero até máximo dependente do sistema)

IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- Evolução do Unix BSD 4.2:
 - sockets
- Gestão de eventos assíncronos:
 - signals (System V e BSD)

Pipes

- Mecanismo original do Unix para comunicação entre processos.
- Canal *byte stream* ligando dois processos, unidirecional
- Não tem nome externo
 - Os descritores são internos a um processo
 - Podem ser transmitidos para os processos filhos através do mecanismo de herança
- Os descritores de um pipe são análogos ao dos ficheiros
 - As operações de read e write sobre ficheiros são válidas para os pipes
 - O processo fica bloqueado quando escreve num pipe cheio
 - O processo fica bloqueado quando lê de um pipe vazio





Criação de um pipe

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

fds[0] - descritor aberto para leitura

fds[1] - descritor aberto para escrita



Criar e usar pipe (Exemplo inútil)

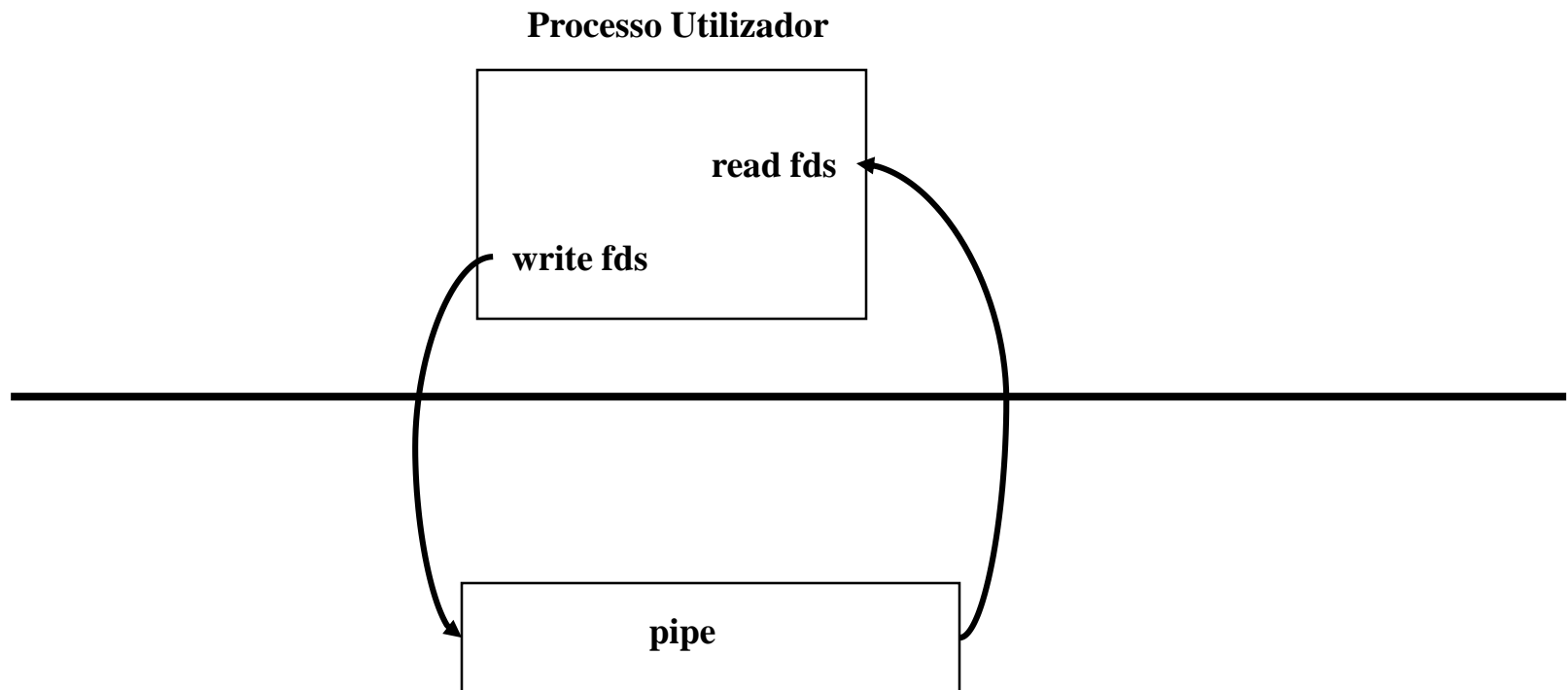
```
char msg[] = "utilizacao de pipes";

main() {
    char tampao[1024];
    int fds[2];

    pipe(fds);

    for (;;) {
        write (fds[1], msg, sizeof (msg));
        read (fds[0], tampao, sizeof (tampao));
    }
}
```

Criar e usar pipe (Exemplo inútil)





Criar e usar pipe

(Exemplo útil: comunicação pai-filho)

```
#include <stdio.h>
#include <fnctl.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*/
        close(fds[1]);
        /* lê do pipe */
        read (fds[0], tmp, sizeof (msg));
        printf ("%s\n", tmp);
        exit (0);
    }
```

```
else {
    /* processo pai */
    close(fds[0]);
    /* escreve no pipe */
    write (fds[1], msg, sizeof (msg));
    pid_filho = wait();
}
```



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 <fnctl.h>

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

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]);
```

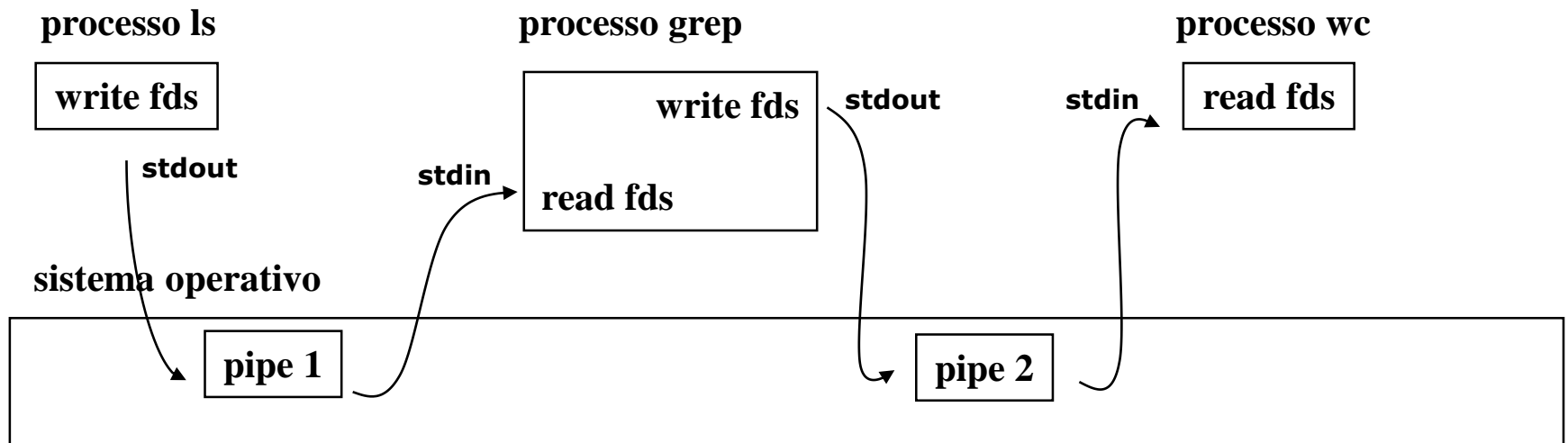
```
/* fecha os descritores não usados pelo
filho */
        close (fds[0]);
        close (fds[1]);

    /* 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();
    }
}
```

Redireccionamento de Entradas/Saídas no Shell

exemplo:

ls -la | grep xpto | wc



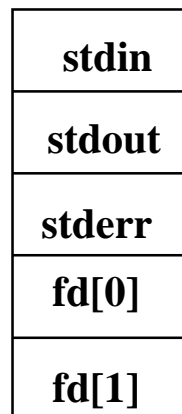
Redireccionamento de Entradas/Saídas (2)

```

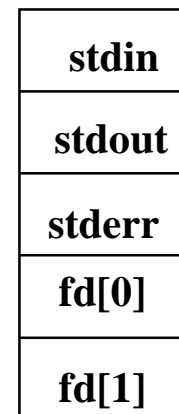
pipe(fds[0], fds[1]);
p = fork();
if (p>0) {
    close (1);
    dup (fds[1]);
    close (fds[0]);
    close (fds[1]);
    execv(...);
}

```

pai



filho



```

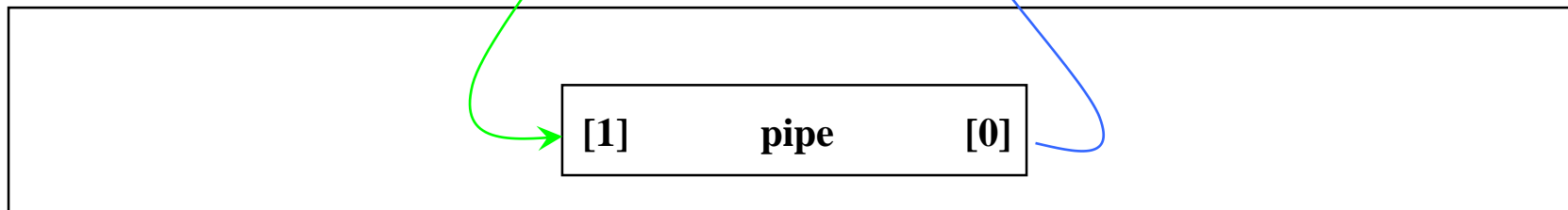
else if (p==0) {
    close (0);
    dup (fds[0]);
    close (fds[0]);
    close (fds[1]);
    execv(...)
}

```

Sem saber, o programa executado lerá (stdin) do pipe

Sem saber, o programa executado escreverá (stdout) para o pipe

[1] pipe [0]





IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- Evolução do Unix BSD 4.2:
 - sockets
- Gestão de eventos assíncronos:
 - signals (System V e BSD)



Named Pipes ou FIFO

- Para dois processos (que não sejam pai e filho) comunicarem é preciso que o pipe seja identificado por um nome
- Atribui-se um nome lógico ao pipe, usando o **espaço de nomes do sistema de ficheiros**
 - Um **named pipe** comporta-se externamente como um ficheiro, existindo uma entrada na directoria correspondente
- Um **named pipe** pode ser aberto por processos que não têm qualquer relação hierárquica
 - Tal como um ficheiro tem um dono e permissões de acesso



Named Pipes

- Um named pipe é um canal :
 - Unidireccional
 - Interface sequência de caracteres (*byte stream*)
 - Identificado por um nome de ficheiro
 - Entre os restantes ficheiros do sistema de ficheiros
 - Ao contrário dos restantes ficheiros, named pipe **não é persistente**



Named Pipes: como usar

- Cria um named pipe no sistema de ficheiros
 - Usando função mkfifo
- Um processo associa-se com a função open
 - Processo que abra uma extremidade do canal bloqueia até que pelo menos 1 processo tenha aberto a outra extremidade
- Eliminado com a função unlink
- Leitura e envio de informação feitos com API habitual do sistema de ficheiros (read, write, etc)



```
/* Cliente */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#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)) < 0) exit(1);
    if ((fcli = open ("/tmp/cliente",
O_RDONLY)) < 0) exit(1);

    produzMsg (buf);
    write (fserv, buf, TAMMSG);
    read (fcli, buf, TAMMSG);
    trataMsg (buf);

    close (fserv);
    close (fcli);
}
```

```
/* Servidor */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define TAMMSG 1000

main () {
    int fcli, fserv, n;
    char buf[TAMMSG];

    unlink("/tmp/servidor");
    unlink("/tmp/cliente");

    if (mkfifo ("/tmp/servidor", 0777) < 0)
        exit (1);
    if (mkfifo ("/tmp/cliente", 0777) < 0)
        exit (1);

    if ((fserv = open ("/tmp/servidor",
O_RDONLY)) < 0) exit(1);
    if ((fcli = open ("/tmp/cliente",
O_WRONLY)) < 0) exit(1);

    for (;;) {
        n = read (fserv, buf, TAMMSG);
        if (n <= 0) break;
        trataPedido (buf);
        n = write (fcli, buf, TAMMSG);
    }
    close (fserv);
    close (fcli);
    unlink("/tmp/servidor");
    unlink("/tmp/cliente");
}
```

IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- Evolução do Unix BSD 4.2:
 - sockets
- Gestão de eventos assíncronos:
 - signals (System V e BSD)



Sockets

- Interface de programação para comunicação entre processos introduzida no Unix 4.2 BSD (1983)
- 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 {
    /* definição do domínio (AF_XX) */
    u_short family;

    /* endereço específico do domínio*/
    char sa_data[14];
};
```

```
/* ficheiro <sys/un.h> */
struct sockaddr_un {
    /* definição do domínio (AF_UNIX) */
    u_short family;

    /* nome */
    char sun_path[108];
};
```

struct sockaddr_un

family
pathname (up to 108 bytes)

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

struct sockaddr_in {
    u_short sin_family; /* AF_INET */

    /* número do porto - 16 bits */
    u_short sin_port;

    struct in_addr sin_addr; /* Netid+Hostid */

    /* não utilizado*/
    char sin_zero[8];
};
```

struct sockaddr_in

family
2-byte port
4-byte net ID, host ID
(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:

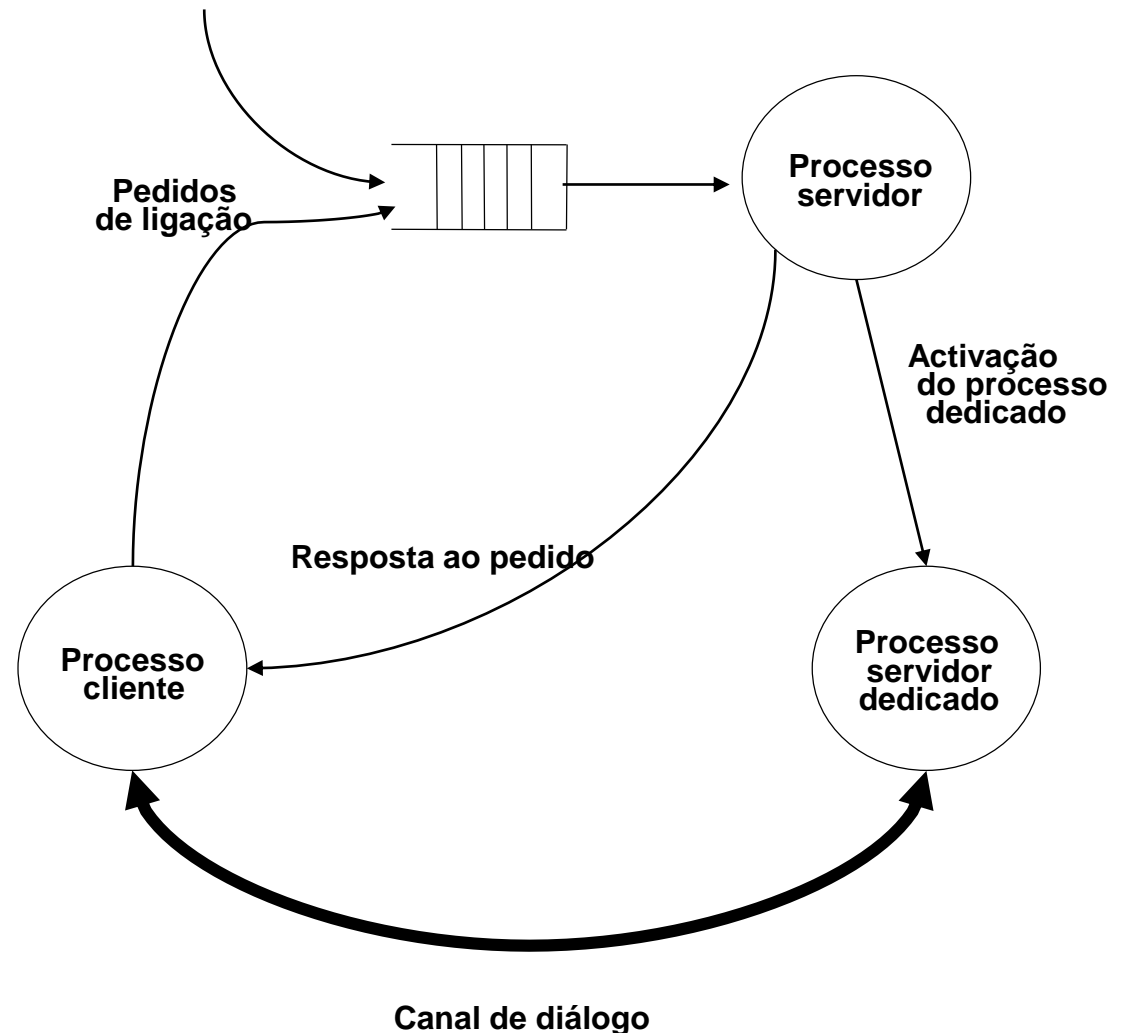
```
int bind(int sockfd, struct sockaddr *nome, int dim)
```

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

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

```
IdCanServidor = 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);
}
```

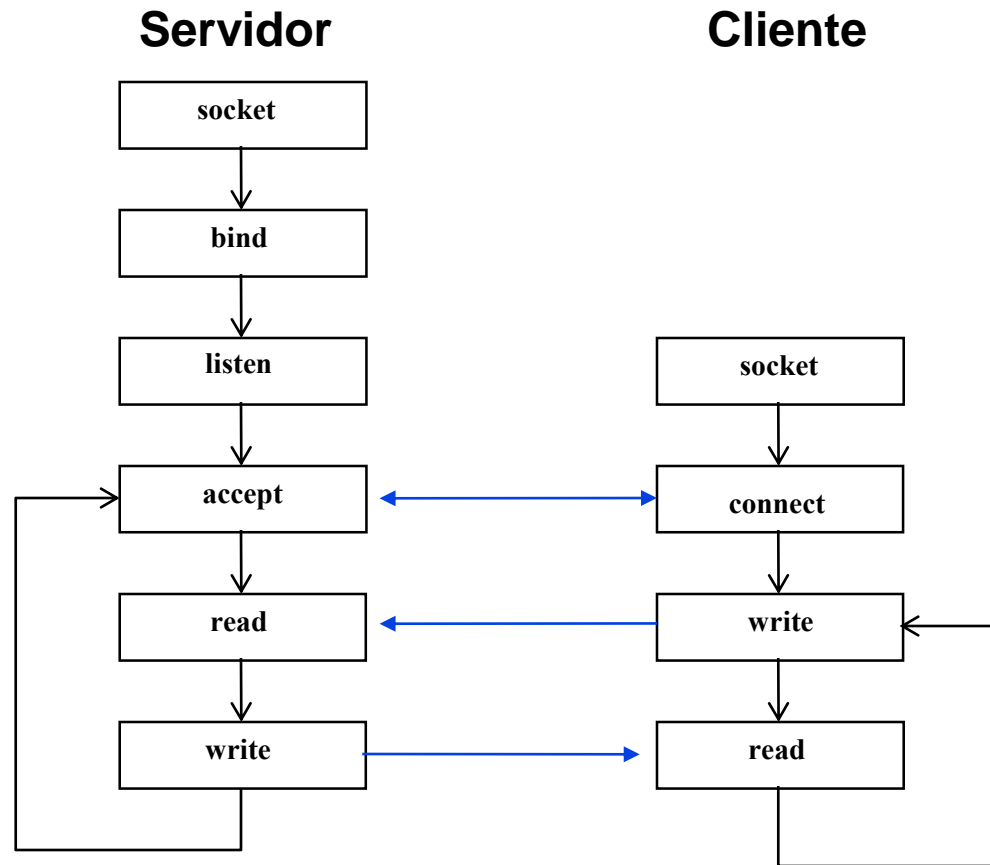
Servidor

```
IdCanal CanalServidor, CanalDialogo;

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

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


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/dgXXXXXXX"
```

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

/* endereço do servidor */
#define SERV_HOST_ADDR "193.136.128.20"

/* nome do servidor */
#define SERV_HOSTNAME  "mega"
```



Exemplo

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



Servidor STREAM AF_UNIX

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

```
main(void) {  
    int sockfd, newsockfd, clilen, 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.
```

```
    unlink(UNIXSTR_PATH);
```

```
    /* 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);  
}  
}
```



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 (write(sockfd, line, n) != n)
            err_dump("str_echo: write error");
    }
}
```



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 (write(sockfd, sendline, n) != n)
    err_dump("str_cli:write 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;
```

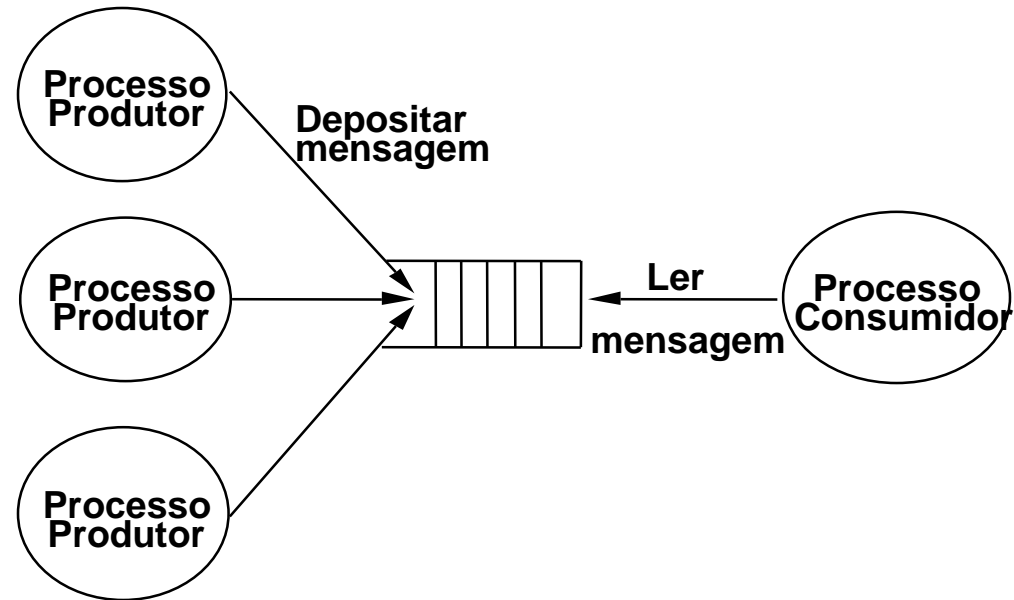
```
/* Envia a string para stdout */
fputs(recvline, stdout);
}
if (ferror(fp))
    err_dump("str_cli: error reading file");
}
```

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

Modelo de comunicação tipo correio (canal sem ligação)

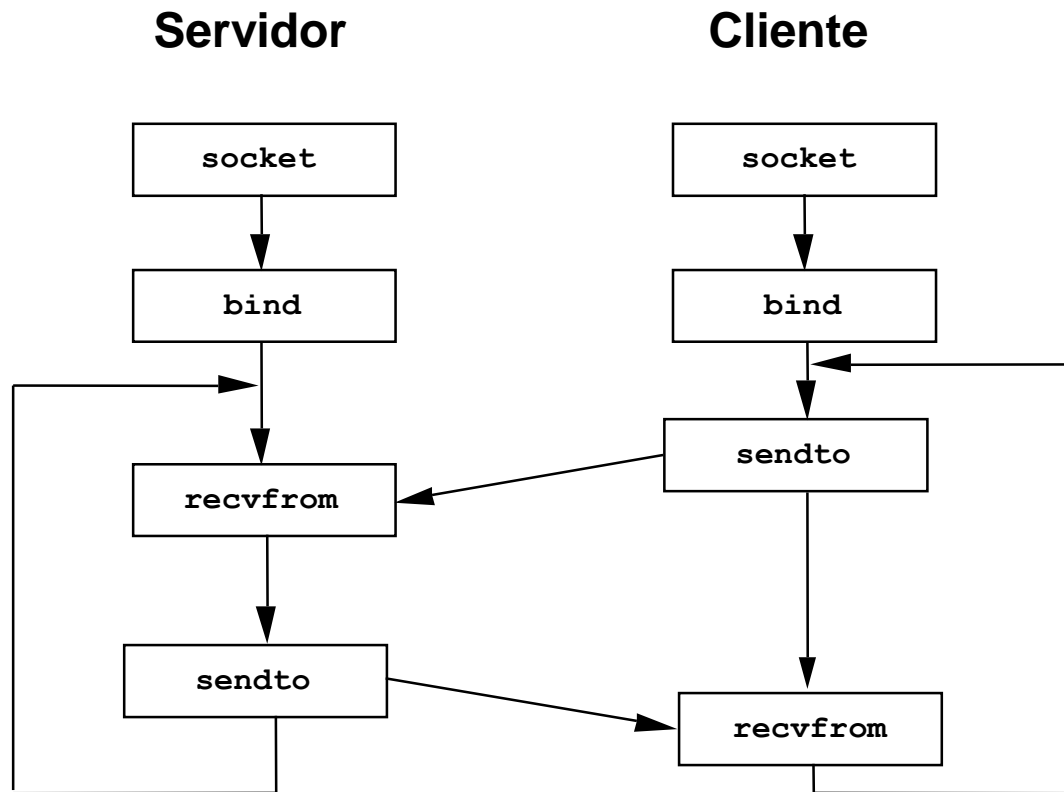
- os processos emissores não controlam directamente a actividade do receptor ou receptores
- a ligação efectua-se indirectamente através das caixas de correio não existe uma ligação directa entre os processos
- a caixa de correio permite memorizar as mensagens quando estas são produzidas mais rapidamente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



```

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

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, clilen, 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);
    clilen = sizeof(cli_addr.sun_family) + strlen(cli_addr.sun_path);

    /* Associa o socket ao nome temporário */
    if (bind(sockfd, (struct sockaddr *) &cli_addr, clilen) < 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>

dg_cli(fp, sockfd, pserv_addr, servlen)
FILE *fp;
int sockfd;
struct sockaddr *pserv_addr;
int servlen;
{
    int n;
    static char sendline[MAXLINE], recvline[MAXLINE+1];
    struct sockaddr x;
    int xx = servlen;
```



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));
}
```



Servidor DGRAM AF_UNIX (3)

```
#define MAXLINE 512

/* Servidor do tipo socket datagram.
   Manda linhas recebidas de volta
   para o cliente */

#include <sys/types.h>
#include <sys/socket.h>
#define MAXMSG 2048

/* pcli_addr especifica o cliente */

dg_echo(sockfd, pcli_addr, maxclilen)
int sockfd;
struct sockaddr *pcli_addr;
int maxclilen;
{
```

```
int n, clilen;
char msg[MAXMSG];

for (;;) {
    clilen = maxclilen;

    /* Lê uma linha do socket */
    n = recvfrom(sockfd, msg, MAXMSG,
                  0, pcli_addr, &clilen);

    if (n < 0)
        err_dump("dg_echo:recvfrom error");

    /*Manda linha de volta para o socket */
    if (sendto(sockfd, msg, n, 0,
               pcli_addr, clilen) != n)
        err_dump("dg_echo: sendto error");
}
```



Servidor TCP AF_INET

(e.g. see http://www.linuxhowtos.org/C_C++/socket.htm)

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg) {
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno;
    socklen_t clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;

    if (argc < 2) {
        fprintf(stderr, "ERROR, no port provided\n");
        exit(1);
    }
}
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) error("ERROR opening socket");

bzero((char *) &serv_addr, sizeof(serv_addr));
portno = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

**binds to any
local interface
(i.e. IP address)**

```
if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0) error("ERROR on binding");
```

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

```
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
if (newsockfd < 0) error("ERROR on accept");
```

```
bzero(buffer,256);
n = read(newsockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
printf("Here is the message: %s\n",buffer);
```

```
n = write(newsockfd,"I got your message",18);
if (n < 0) error("ERROR writing to socket");
close(newsockfd);
close(sockfd);
```

```
return 0;
```

```
}
```



ClienteTCP AF_INET

(e.g. see http://www.linuxhowtos.org/C_C++/socket.htm)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg) {
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[]) {
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];

    if (argc < 3) {
        fprintf(stderr,"usage %s hostname
        port\n", argv[0]);
        exit(0);
    }

    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
```

```
if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
}
```

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr,
      (char *)&serv_addr.sin_addr.s_addr,
      server->h_length);
serv_addr.sin_port = htons(portno);
```

```
if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
printf("Please enter the message: ");
```

```
bzero(buffer,256);
fgets(buffer,255,stdin);
n = write(sockfd,buffer,strlen(buffer));
```

```
if (n < 0) error("ERROR writing to socket");
bzero(buffer,256);
n = read(sockfd,buffer,255);
if (n < 0) error("ERROR reading from socket");
```

```
printf("%s\n",buffer);
close(sockfd);
return 0;
```

```
}
```

Autenticação de processos comunicantes

- Em sockets UNIX, é possível a um processo saber com qual outro processo/utilizador está comunicar
 - Informação disponível: PID, UID, GID
 - Esta informação é fornecida pelo SO, logo não forjável por processos maliciosos
- Este suporte não existe com sockets Internet
 - Porquê?
 - Na Internet precisamos de outros mecanismos para autenticação (será estudado na disciplina Sistemas Distribuídos)



Autenticação de processos comunicantes: com sockets stream

```
#define _GNU_SOURCE
#include <sys/socket.h>
#include <sys/un.h>
```

Atenção a este detalhe,
necessário para usar a struct ucred
*Nota: esta solução é específica de
Linux, pode não funcionar noutros SO*

```
struct ucred ucred;
```

```
...
```

```
newsockfd = accept(sockfd, NULL, NULL);
if (newsockfd == -1) {...}
```

Após ligação estabelecida,
podemos pedir as credenciais do
processo ligado no outro socket

O mesmo pode ser feito no lado
do cliente, após connect(...)

```
len = sizeof(struct ucred);
if (getsockopt(newsockfd, SOL_SOCKET, SO_PEERCRED, &ucred, &len) == -1) {...}
```

```
printf("Credenciais do processo no outro lado da ligação:");
printf("pid=%ld, euid=%ld, egid=%ld\n",
      (long) ucred.pid, (long) ucred.uid, (long) ucred.gid);
```

Credenciais fornecidas
incluem PID, EUID, EGID

Autenticação de processos comunicantes: com sockets datagram

- Neste caso, o recetor de mensagem pode autenticar o processo que enviou a mensagem
- Mais complicado de programar:
 - Necessário a mensagem ser enviada com cabeçalho que transporta as credenciais
 - Funções sendto/recvfrom não servem
 - Precisamos usar variantes sendmsg/recvmsg, que permitem maior controlo
 - Esse cabeçalho é preenchido pelo SO



Espera Múltipla com Select

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfd, fd_set* leitura, fd_set*
            escrita, fd_set* excepcão, 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 microsegundos
- 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* excecao



Servidor com Select

```
/* Servidor que utiliza sockets stream e
   datagram em simultâneo.
   O servidor recebe caracteres e envia-os
   para stdout */

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

#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"
```

```
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))<0) {
        perror(ERRORMSG1);
        exit(1);
    }
    bzero((char*)&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 *)&servstrmaddr,
           len)<0)
    {
        perror(ERRORMSG2);
        exit(1);
    }
}
```



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)) < 0) { perror(ERRORMSG3); exit(1); } /*Inicializa socket datagram: tipo + nome */ bzero((char *)&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 sockaddr*)&servdgrmaddr, len)<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(&testmask); FD_SET(strmfd,&testmask); FD_SET(dgrmfd,&testmask);</pre>
---	---



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! */  
}  
}
```




IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- Evolução do Unix BSD 4.2:
 - sockets
- Gestão de eventos assíncronos:
 - signals (System V e BSD)



Signals

Signals

- Dois propósitos distintos:
 - Mecanismo usado pelo núcleo do SO para notificar um processo de que ocorreu um dado evento relevante
 - Exemplos: CTRL-C, *timeout*, acesso inválido a memória, etc.
 - Mecanismo limitado de comunicação entre processos
 - Permite a um processo notificar outro que ocorreu um dado evento
 - Exemplo: processo servidor notifica outros processos para que iniciem procedimento de terminação
- Em ambos os casos, o **evento** é tratado de forma **assíncrona** pelo processo



Eventos Assíncronos

- Rotinas Assíncronas para Tratamento de acontecimentos assíncronos e exceções

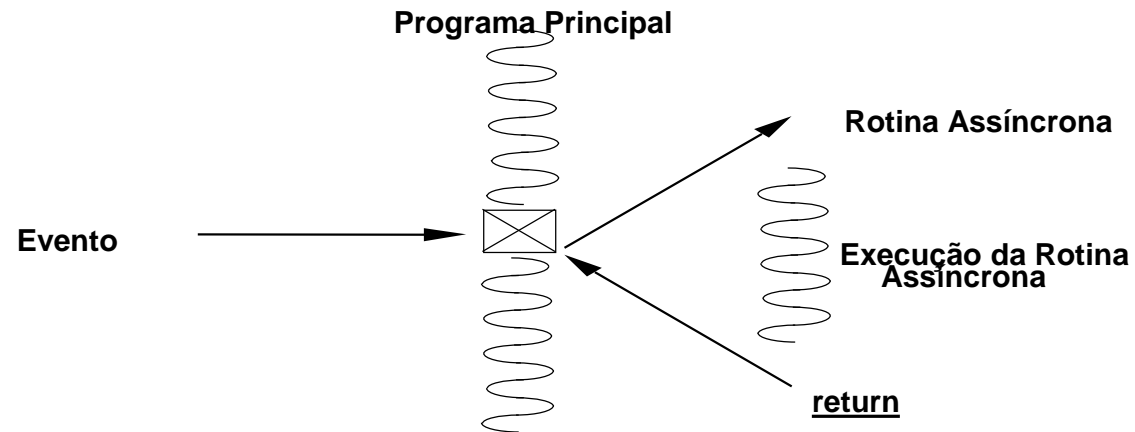


Rotinas Assíncronas

- Certos acontecimentos devem ser tratados pelas aplicações, embora não seja possível prever a sua ocorrência
 - Ex: Ctrl-C
 - Ex: Acção desencadeada por um timeout
- Como tratá-los na programação sequencial?

Modelo de Eventos

- Semelhante a outro conceito...



Rotinas Assíncronas

RotinaAssincrona (Evento, Procedimento)

**Tem de existir
uma tabela com
os eventos que o
sistema pode
tratar**

**Identificação do
procedimento a
executar
assincronamente
quando se
manifesta o evento.**

Signals

Acontecimentos Assíncronos em Unix

Signal	Causa
SIGALRM	O relógio expirou
SIGCHLD	Um dos processos filhos alterou o seu estado (terminou, suspendeu ou retomou a execução)
SIGFPE	Divisão por zero
SIGINT	O utilizador carregou na tecla para interromper o processo (normalmente o CNTL-C)
SIGQUIT	O utilizador quer terminar o processo e provocar um core dump
SIGKILL	Signal para terminar o processo. Não pode ser tratado
SIGPIPE	O processo escreveu para um pipe que não tem receptores
SIGSEGV	Acesso a uma posição de memória inválida
SIGTERM	O utilizador pretende terminar ordeiramente o processo
SIGUSR1	Definido pelo utilizador
SIGUSR2	Definido pelo utilizador

Exceção

Interação com o terminal

Desencadeado por interrupção HW

- Há mais, definidos em signal.h

Explicitamente desencadeado por outro processo para notificar de algum acontecimento relacionado com a aplicação (forma limitada de IPC)

Tratamento por omissão

- Cada signal tem um tratamento por omissão, que pode ser:
 - Terminar o processo
 - Terminar o processo e criar ficheiro “core”
 - Ignorar signal
 - Suspende o processo
 - Continuar o processo suspenso



Redefinir o tratamento de um Signal

- Função `signal` permite mudar o tratamento de um signal:
 - Mudar para outro tratamento pré-definido (slide anterior)
 - Associar uma rotina do programa para tratar o signal
- O signal `SIGKILL` não pode ser redefinido. Porquê?

Redefinir o tratamento de um Signal: Chamada Sistema “Signal”

```
void (*signal (int sig, void (*func)(int))) (int);
```

**A função
retorna um
ponteiro para
função
anteriormente
associada ao
signal**

**Identificador
do signal
para o qual
se pretende
definir um
handler**

**Ponteiro para a
função
ou macro
especificando:
•SIG_DFL – acção
por omissão
•SIG_IGN –
ignorar o signal**

**Parâmetro
para a
função de
tratamento**



Exemplo do tratamento de um Signal

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

void apanhaCTRLC (int s) {
    char ch;
    printf("Quer de facto terminar a execucao?\n");
    ch = getchar();
    if (ch == 's') exit(0);
    else {
        printf ("Entao vamos continuar\n");
        signal (SIGINT, apanhaCTRLC);
    }
}

int main () {
    signal (SIGINT, apanhaCTRLC);
    printf("Associou uma rotina ao signal SIGINT\n");
    for (;;)
        sleep (10);
}
```

Chamada Sistema Kill

- Envia um signal ao processo
- Nome enganador. Porquê?

```
kill (pid, sig);
```

Identificador do processo

Se o pid for zero é enviado a todos os processos do grupo

Está restrito ao superuser o envio de *signals* para processos de outro user

Identificador do signal

Outras funções associadas aos signals

- `unsigned alarm (unsigned int segundos);`
 - o *signal* SIGALRM é enviado para o processo depois de decorrerem o número de segundos especificados. Se o argumento for zero, o envio é cancelado.
- `pause();`
 - aguarda a chegada de um *signal*
- `unsigned sleep (unsigned int segundos);`
 - A função chama *alarm* e bloqueia-se à espera do *signal*
- `int raise(int sig)`
 - o signal especificado em input é enviado para o próprio processo

Diferentes semânticas dos signals: Unix System V e Unix BSD

- System V:
 - A associação de uma rotina a um *signal* é apenas efetiva para uma ativação
 - Depois de receber o *signal*, o tratamento passa a ser novamente o por omissão (necessário associar de novo)
 - Entre o lançamento de rotina de tratamento e a nova associação → tratamento por omissão
 - Preciso restabelecer a associação na primeira linha da rotina de tratamento
 - Problema se houver receção sucessiva de signals
- BSD (e nas versões de Linux mais recentes, desde glibc2):
 - Associação *signal*-rotina **não** é desfeita após ativação
 - A receção de um novo signal é inibida durante a execução da rotina de tratamento

Diferentes semânticas dos signals: Como conseguir código portátil?

- Não associar *signals* a rotinas
 - Associar apenas a SIG_DFL ou SIG_IGN

ou

- Usar função *sigaction*
 - Ver detalhes nas *man pages*

ou

- Em plataformas Linux, para ter a certeza de obter semântica BSD, usar `bsd_signal`



Dificuldades com programação usando *signals*: funções não reentrantes

- Um signal pode interromper o processo em qualquer altura!
 - inclusive em alturas “críticas” em que o estado do processo se encontra parcialmente atualizado
- Portanto o *signal handler* deve executar exclusivamente funções cuja correção é garantida independentemente do estado em que se encontra processo:
 - são chamadas **funções reentrantes**

Funções “async-signal-safe”

- A lista das funções que podem ser chamadas a partir dum signal pode ser obtida na página de manual do [signal\(7\)](#)
- Estas funções são também chamadas “***async-signal-safe***” e incluem:
 - funções reentrantes
 - funções cuja execução não pode ser interrompidas por *signals* (pois os bloqueiam durante a própria execução)



Exemplo do tratamento de um Signal usando apenas funções async-signal-safe

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

void apanhaCTRLC (int s) {
    char ch;
    char* str1="Quer de facto terminar a execucao?\n";
    char* str2="Entao vamos continuar\n";
    write (1,str1, strlen(str1));
    read(0,&ch,1);
    if (ch == 's') exit(0);
    else {
        write (2,str2, strlen(str2));
        signal (SIGINT, apanhaCTRLC);
    }
}

int main () {
    signal (SIGINT, apanhaCTRLC);
    printf("Associou uma rotina ao signal SIGINT\n");
    for (;;)
        sleep (10);
}
```



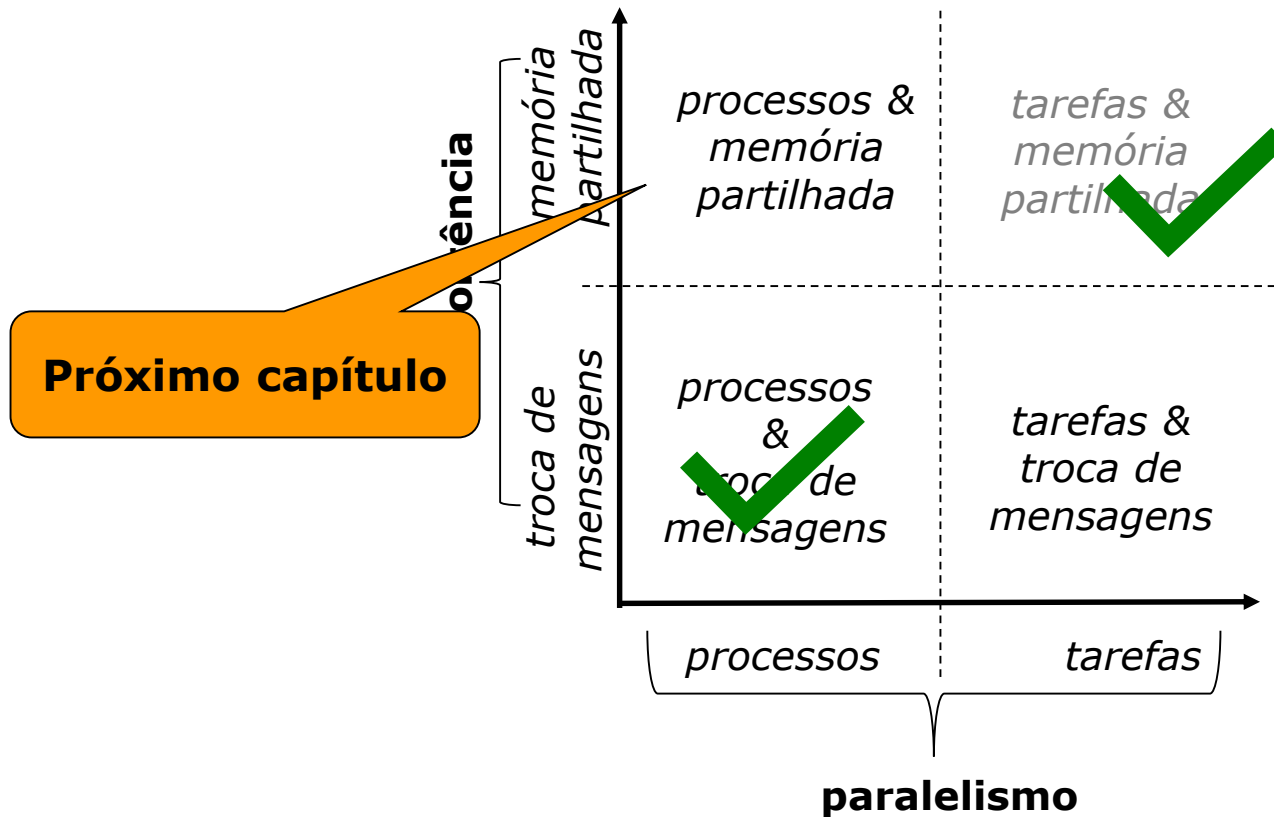
Signals em processos multi-tarefa

Processo com múltiplas tarefas recebe um signal associado a uma função de tratamento.

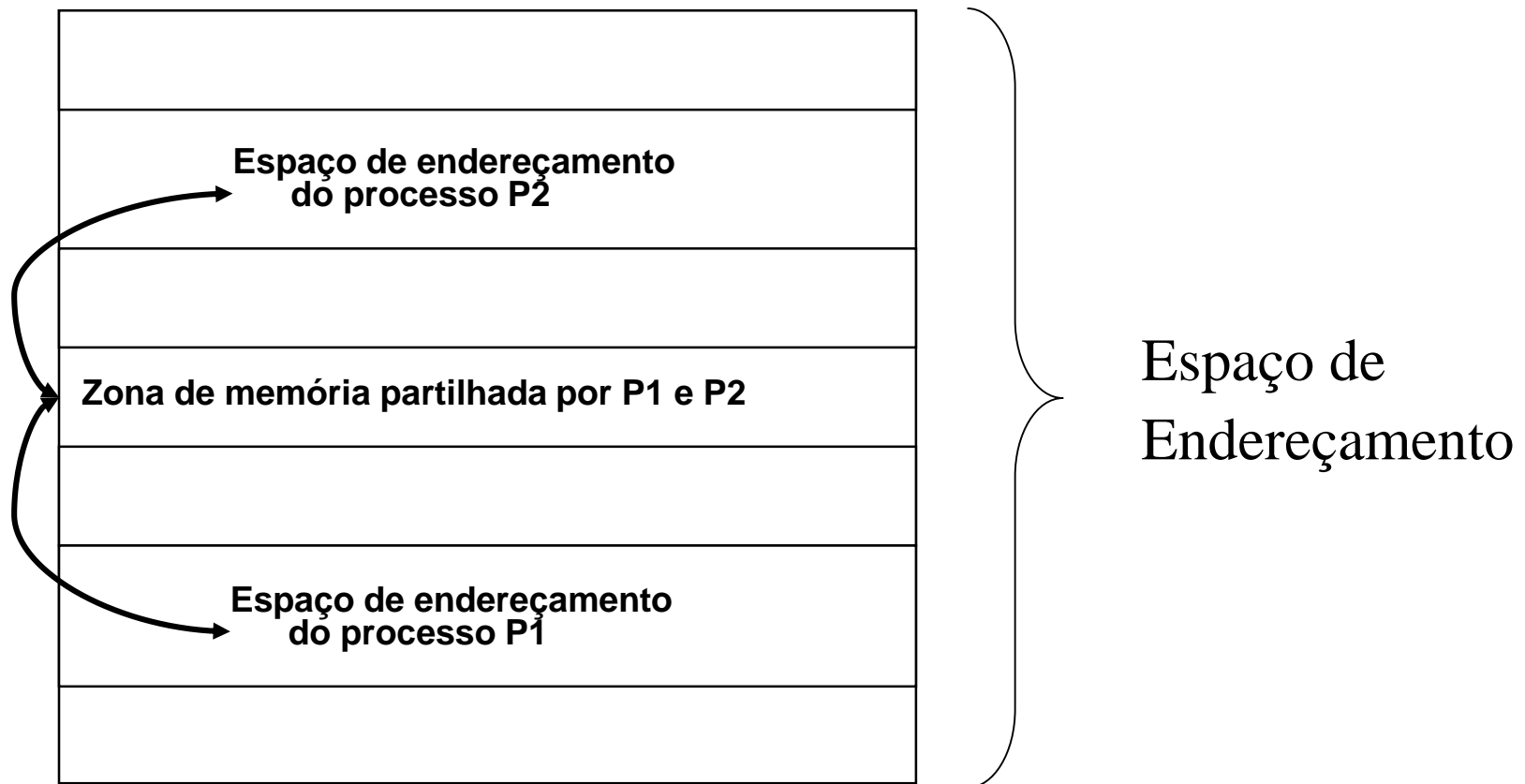
Qual das tarefas é interrompida para executar a função?

- Por omissão, o OS escolhe uma qualquer tarefa do processo
- Podemos usar função `pthread_sigmask` para impor que determinadas tarefas não tratem aquele signal
 - Basta cada tarefa chamar `pthread_sigmask` para bloquear o signal

Combinações de modelos de paralelismo e coordenação



Arquitetura da Comunicação: por memória partilhada





Memória Partilhada: em teoria

- `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-escritores)**



Memória Partilhada: na prática em sistemas Unix/Linux

- Duas principais implementações:
 - Segmentos de memória partilhada do *System V*
 - Historicamente mais populares
 - Mapeamento de ficheiros do *BSD*
 - Adoptado nas interfaces standard POSIX



Segmentos de Memória Partilhada (*System V*)

Modelo de programação no *System V*

- cada objeto é identificado por uma key
- o espaço de nomes é separado do sistema de ficheiros
- os nomes são locais a uma máquina
- as permissões de acesso são idênticas às de um ficheiro (r/w para *user/group/other*)
- os processos filho herdam os objetos abertos

Segmentos de memória partilhada (System V)

- criar/abrir um segmento:

```
int shmget (key_t key, int size, int shmflg)
```

Identificador

Tamanho (em bytes)

Opções, por exemplo
se é para criar caso
ainda não exista

- associar um segmento ao espaço de endereçamento do processo:

```
char* shmat (int shmid, char *shmaddr, int shmflg)
```

Devolve o endereço
base em que o segmento
foi mapeado

Solicita um determinado endereço base.
Se for zero, o endereço é escolhido livremente pelo SO.

Se SHM_RDONLY o acesso
fica restrito a leitura



Segmentos de memória partilhada (System V), cont.

- eliminação da associação:

```
int shmdt (char *shmaddr) ;
```



Exemplo: Memória Partilhada

```
/* produtor */  
  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/shm.h>  
  
#define CHAVEMEM 10  
int IdRegPart;  
int *Apint;  
int i;
```

```
main () {  
    IdRegPart = shmget (CHAVEMEM, 1024, 0777| IPC_CREAT);  
    if (IdRegPart<0) perror(" shmget:");  
  
    printf (" criou uma regiao de identificador %d \n",  
            IdRegPart);  
  
    Apint = (int *)shmat (IdRegPart, (char *) 0, 0);  
    if (Apint == (int *) -1) perror("shmat:");  
  
    for (i = 0; i<256; i++) *Apint++ = i;  
}
```



Exemplo: Memória Partilhada

```
/* consumidor*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CHAVEMEM 10

int IdRegPart;
int *Apint;
int i;
```

```
main() {
    IdRegPart = shmget (CHAVEMEM, 1024, 0777);
    if (IdRegPart < 0)
        perror("shmget:");

    Apint=(int*)shmat (IdRegPart, (char *)0, 0);
    if(Apint == (int *) -1)
        perror("shmat:");

    printf(" mensagem na regioao de memoria partilhada \n");
    for (i = 0; i<256; i++)
        printf ("%d ", *Apint++);

    printf (" \n liberta a regioao partilhada \n");
    shmctl (IdRegPart, 0, IPC_RMID,0);
}
```



Mapeamento de ficheiros (BSD, POSIX)

Alternativa para partilha de memória entre
processos



Diferenças importantes em relação a segmentos partilhados em System V

- Identificador de segmento partilhado entre processos passa a ser um **nome de ficheiro**
 - Em vez de uma chave numérica
- Mapeamento com sistema de ficheiros permite programar com estruturas de dados em ficheiros sem usar read/write/etc.
 - Basta mapear ficheiro em memória, ler e alterar diretamente
 - Alterações são propagadas para o ficheiro automaticamente
- Além de outras diferenças... (ver man pages)

Mapeamento de ficheiros (BSD, POSIX)

- Mapear ficheiro em memória

Endereço base desejado

Tamanho do segmento

Acesso pretendido
(PROT_READ, etc)

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

Opções várias, incluindo:
Segmento poder ser partilhado com
outros processos

offset dentro do ficheiro

(Opcional)
Ficheiro previamente aberto (com open) cujo
conteúdo é mapeado no segmento.
Permite a outros processos partilharem este
segmento de memória, com open+mmap do mesmo
ficheiro.

- Remover mapeamento: **munmap**



Recapitulando...

Dificuldades com programação em memória partilhada (I)

- *malloc, free* não funcionam sobre os segmentos partilhados
 - Tipicamente, o programador tem de gerir manualmente a memória
- Uso de ponteiros dentro da região partilhada é delicado
 - Exemplo: numa lista mantida em memória partilhada, o que acontece se diferentes processos seguem o ponteiro para o primeiro elemento da lista?
 - Só funciona corretamente se todos os processos mapearem o segmento partilhado no mesmo endereço base!

Dificuldades com programação em memória partilhada (II)

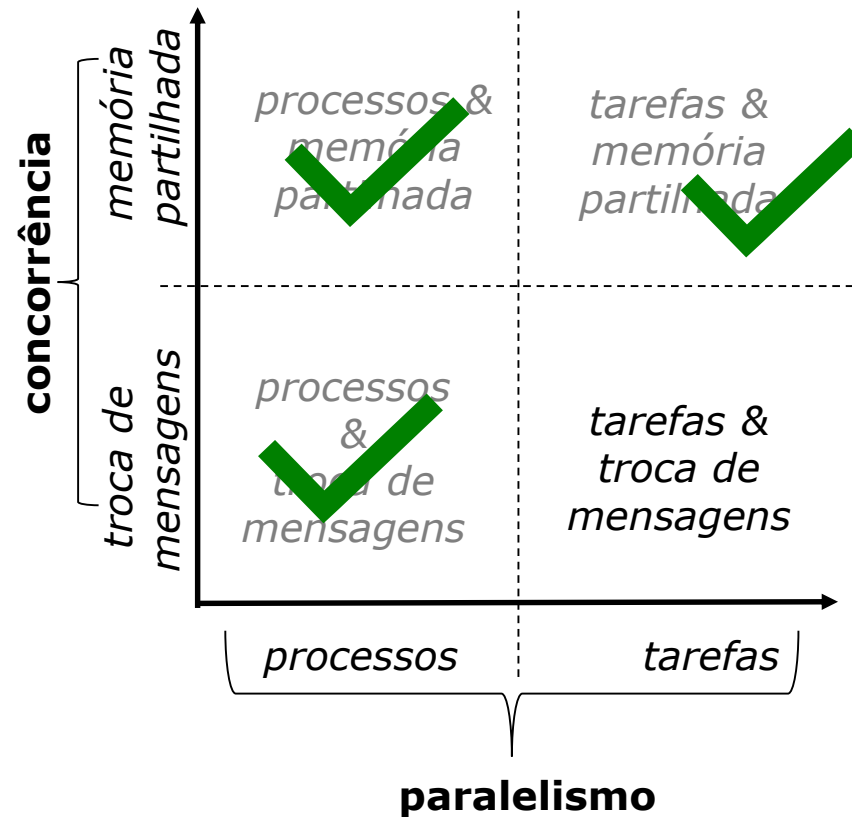
- Dados no segmento partilhado podem ser acedidos concorrentemente, logo precisamos de **sincronização entre processos**
 - Mecanismos de sincronização que estudámos podem ser inicializados com opção multi-processo
 - Exemplo: opção `_POSIX_THREAD_PROCESS_SHARED` de `pthread_mutex_t`
 - Outros mecanismos foram propostos para este caso:
 - Exemplo: semáforos System V
- Sincronização entre processos traz desafios não triviais que não existiam entre tarefas
 - Exemplo: processo adquiriu mutex sobre variável partilhada mas *crasha* sem libertar o mutex

Então o que é melhor para comunicar entre processos?

Memória partilhada vs. canal de comunicação do SO

- Memória partilhada:
 - programação complexa
 - a sincronização tem de ser explicitamente programada
 - mecanismo mais eficiente (menos cópias)
- Canal de comunicação do SO:
 - fácil de utilizar (?)
 - sincronização implícita
 - velocidade de transferência limitada pelas duas cópias da informação e pelo uso das chamadas sistema para Enviar e Receber

Combinações de modelos de paralelismo e coordenação



Tarefas e trocas de mensagens

- Modelo computacional apropriado em caso de:
 - problemas cuja paralelização é mais simples usando troca de mensagens que memória partilhada
 - evita-se intencionalmente partilha de memória para evitar problemas de sincronização
 - implementações portáteis em que com o mesmo código pretendemos executar na mesma máquina ou máquinas distribuídas
- Várias alternativas possíveis pela implementação do mecanismo de troca de mensagens:
 - nível SO:
 - p.e., (Unix) sockets, anonymous pipes
 - nível aplicação, p.e. :
 - “queue” implementada em memória partilhada e sincronizada usando mutexes/semáforos