



Comunicação por Troca de Mensagem entre Processos

Introdução

Pipes

Named pipes

Sistemas Operativos

Dois paradigmas para programação concorrente

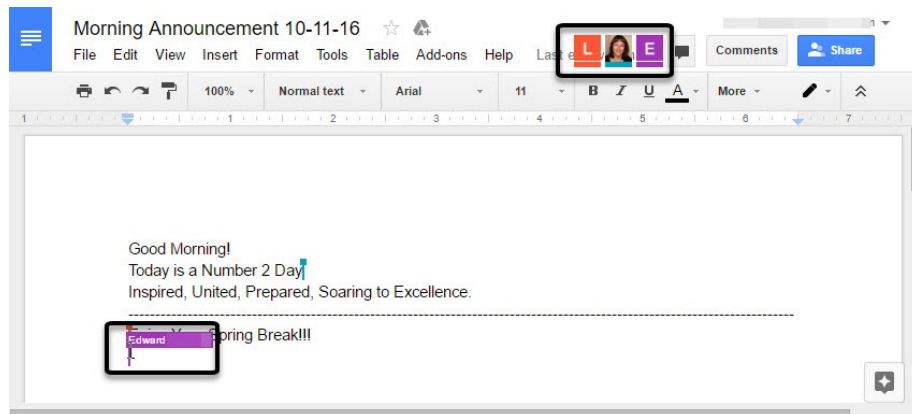
- Por memória partilhada
 - Tarefas partilham dados (no *heap*/amontoadado)
 - Troca de dados é feita escrevendo e lendo da memória partilhada
 - Sincronização recorre a mecanismos adicionais (p.e., trincos, semaforos,...).
- Por troca de mensagens
 - Cada tarefa trabalha exclusivamente sobre dados privados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

Analogia:

Edição concorrente de um documento

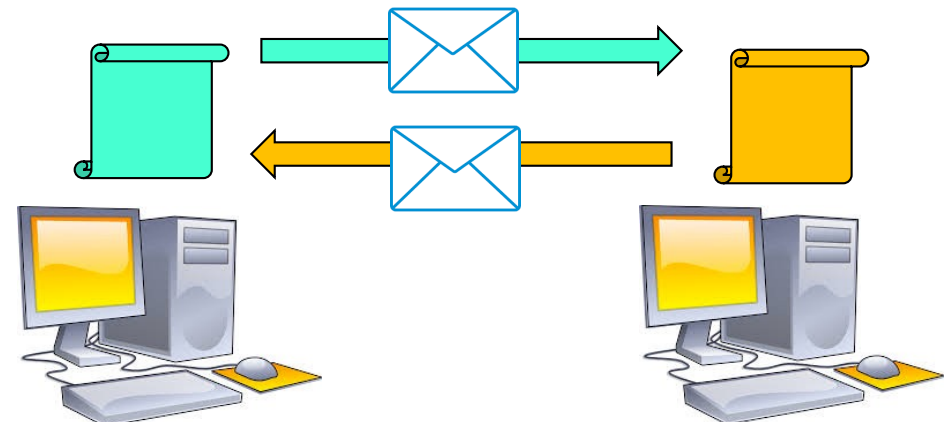
Memória partilhada

- 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 mensagens

- 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

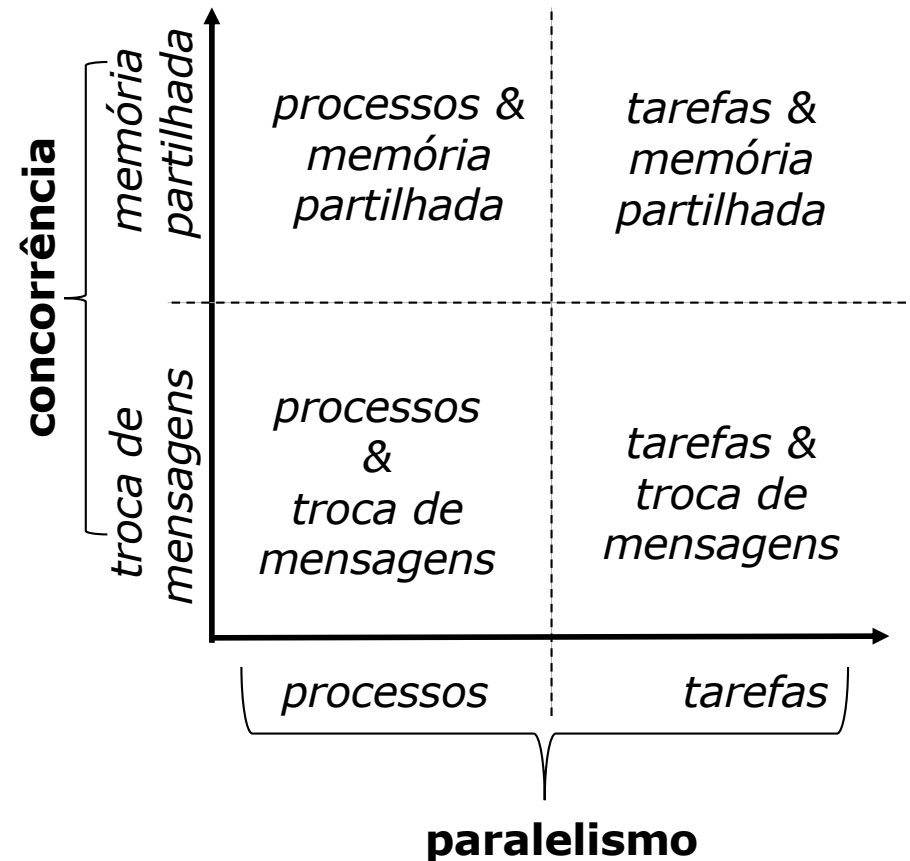


Porquê diferentes paradigmas?

- 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

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!
 - resultado: 4 alternativas



O que construímos até agora...

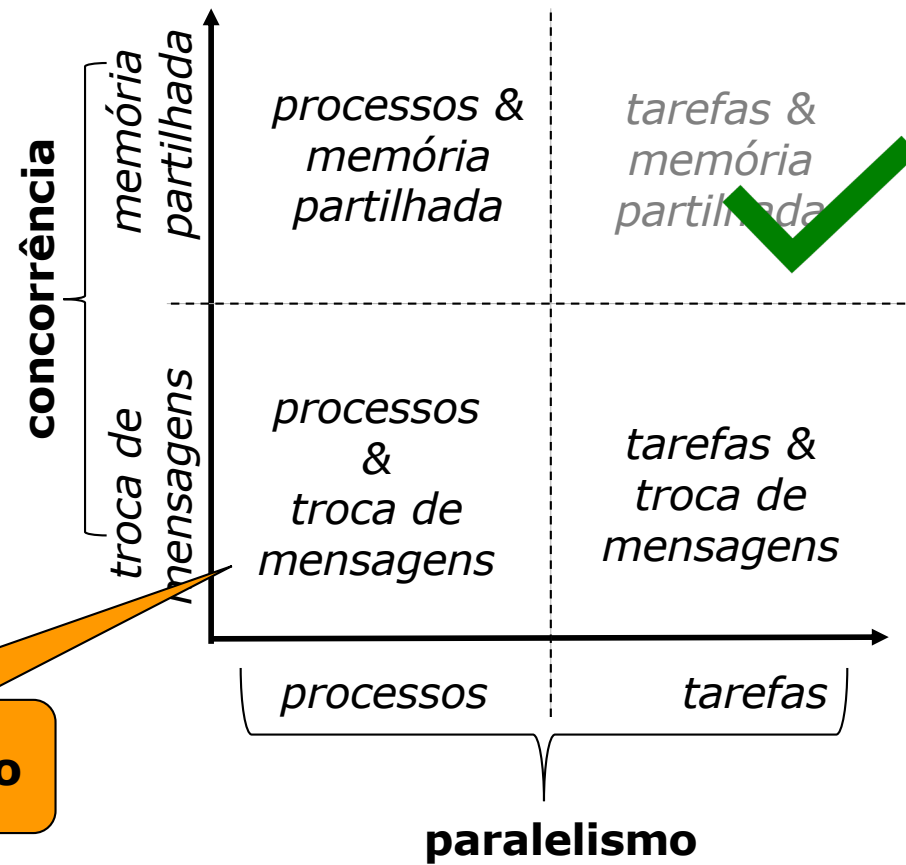


A abstração de processo

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

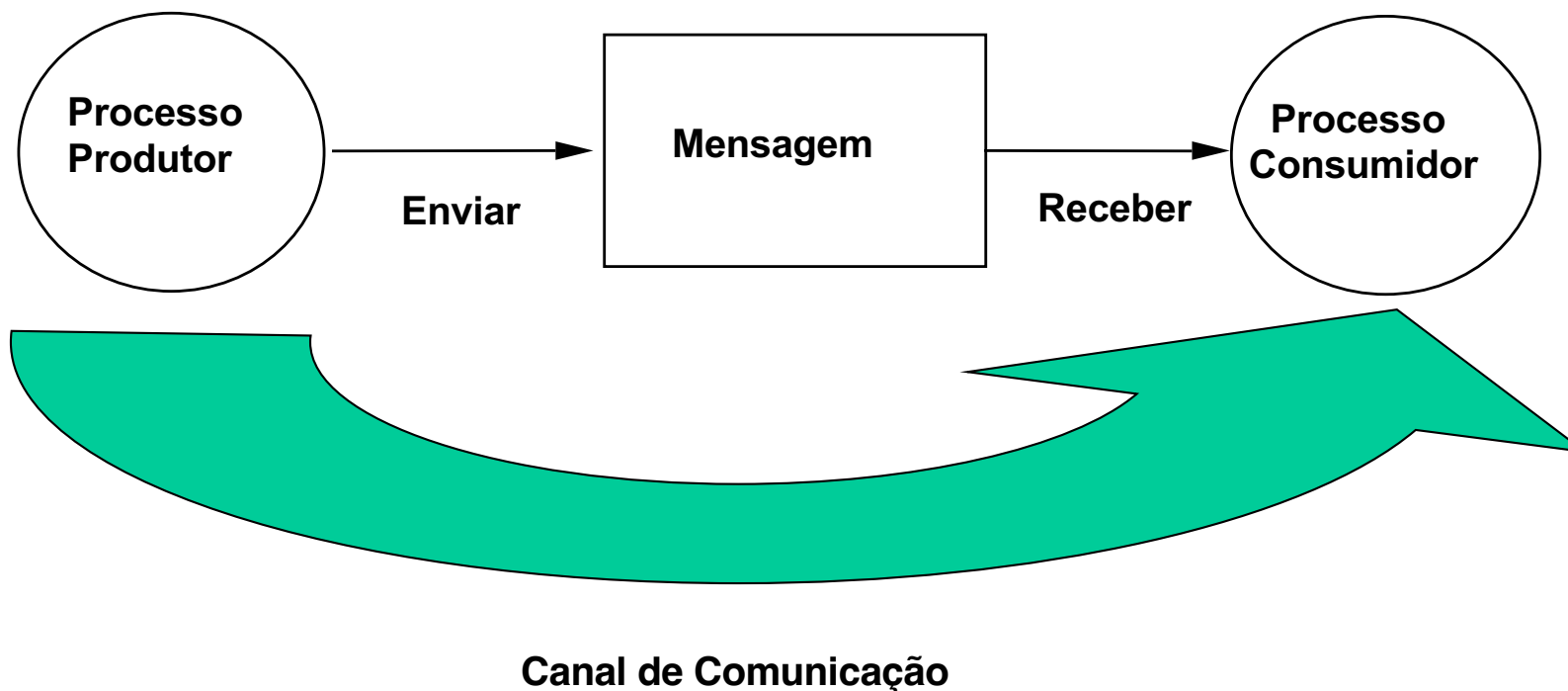


Combinações de modelos de paralelismo e coordenação



Próximo capítulo

Comunicação por Troca de Mensagem entre Processos





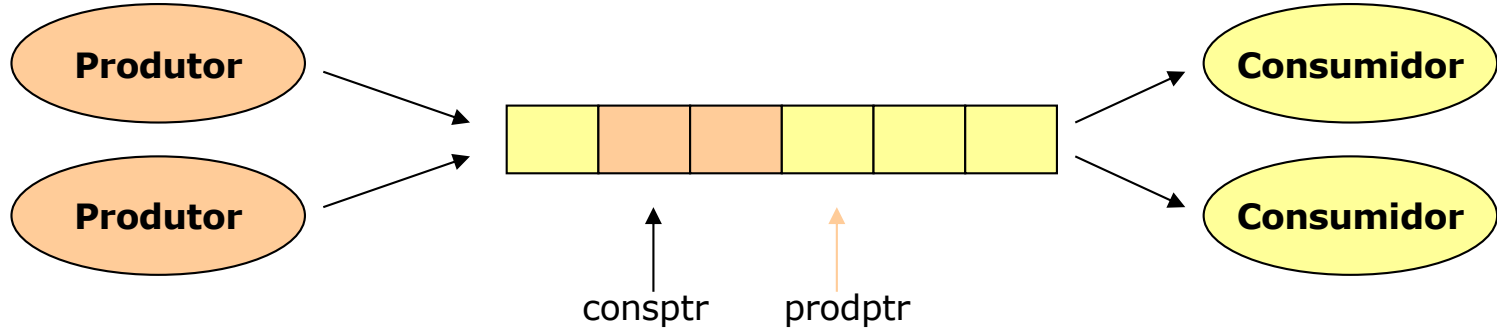
Exemplos

- 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)

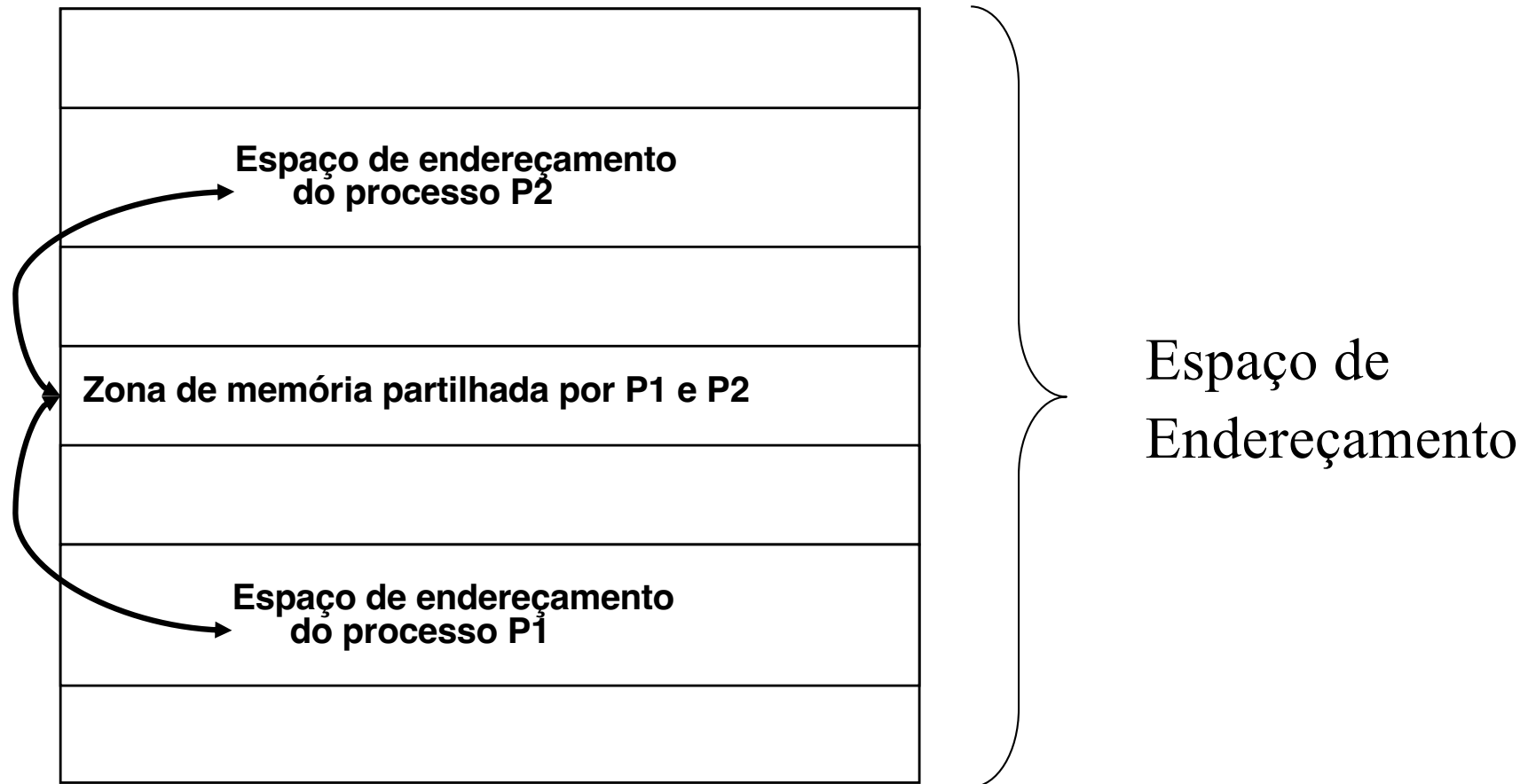




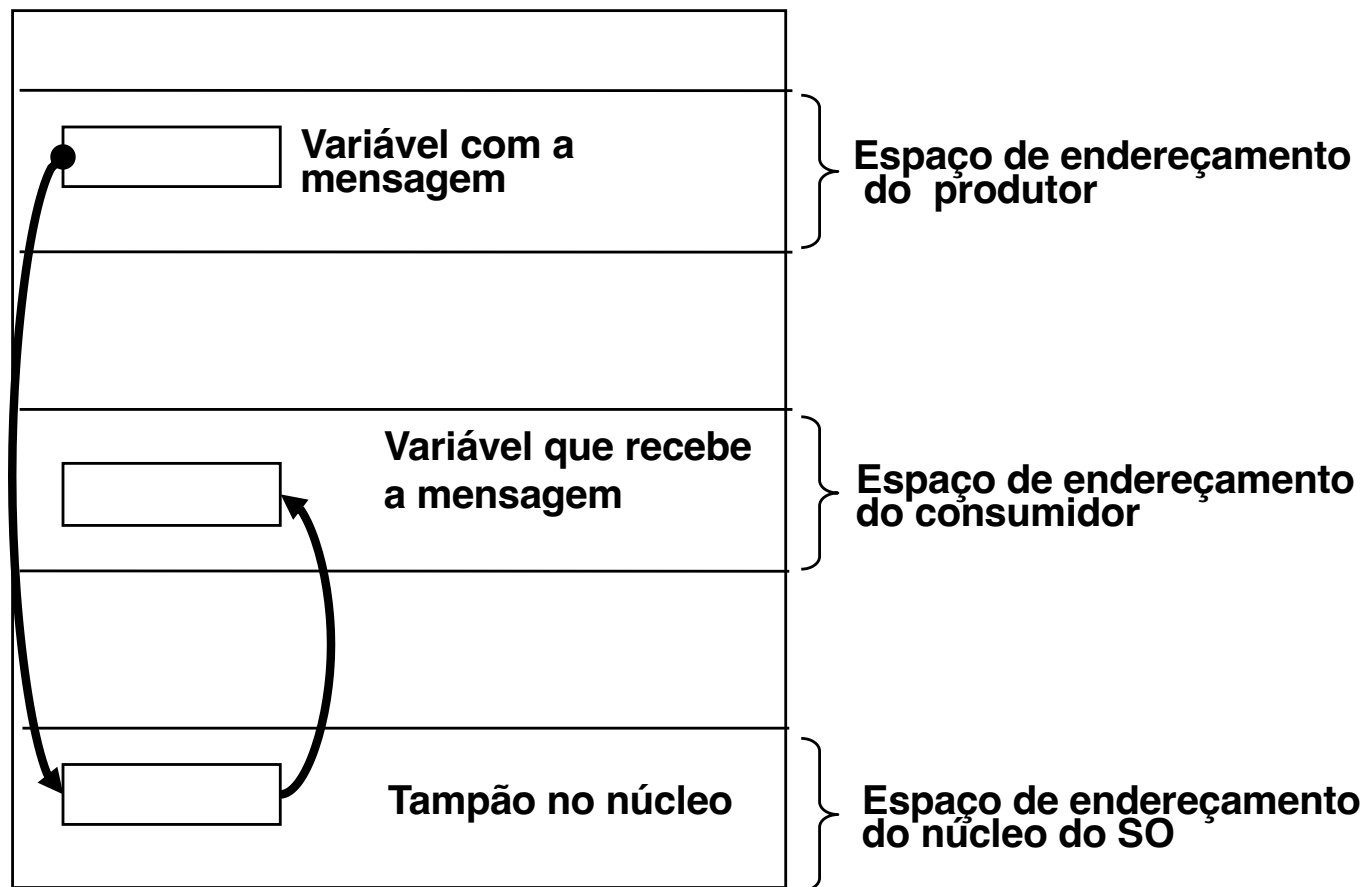
Implementação do Canal de Comunicação

- 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

Arquitetura da Comunicação: por memória partilhada



Arquitetura da Comunicação: cópia através do núcleo





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



Unix– Modelo Computacional - IPC

pipes
signals

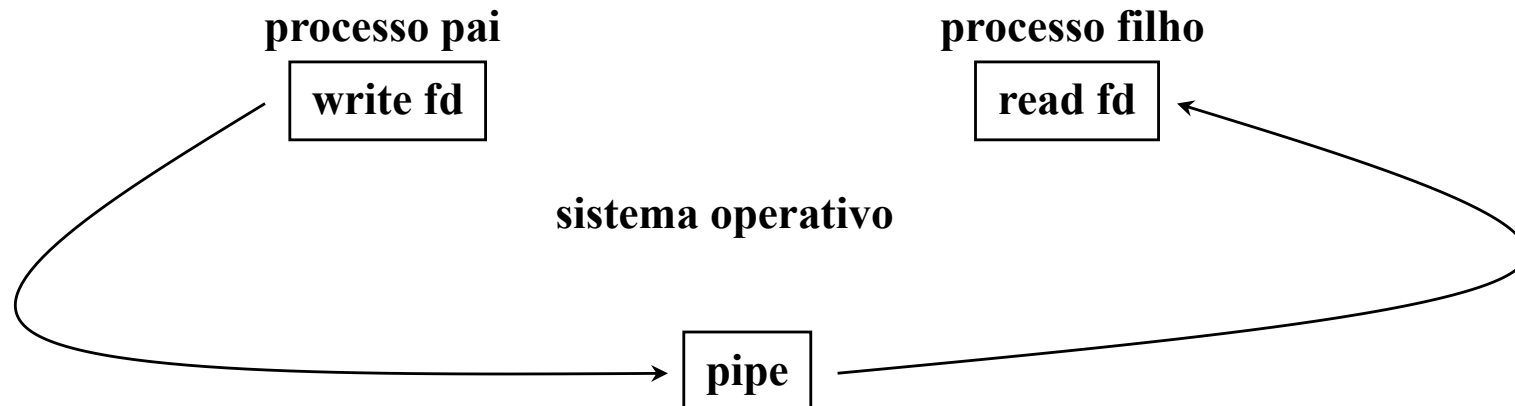


IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- 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)

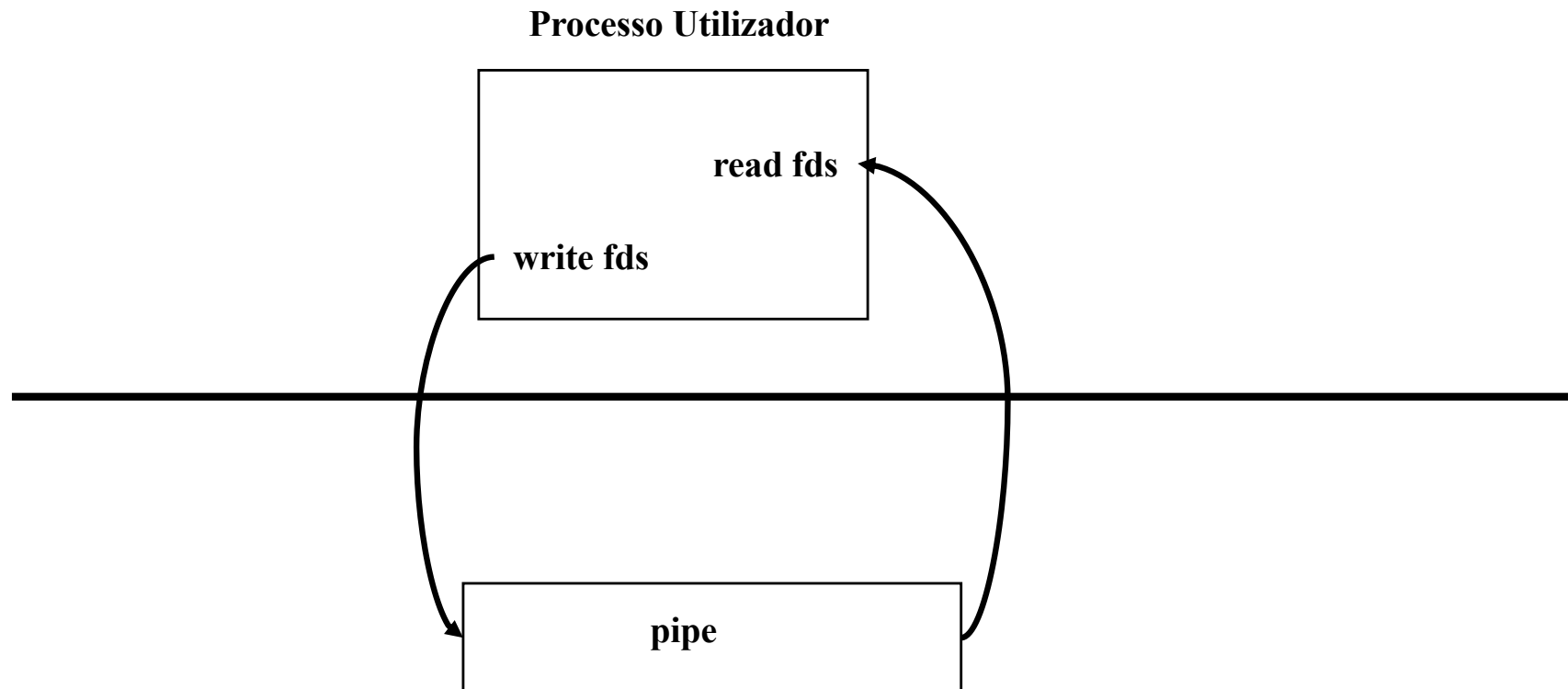
```
char msg[] = "utilizacao de pipes";

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

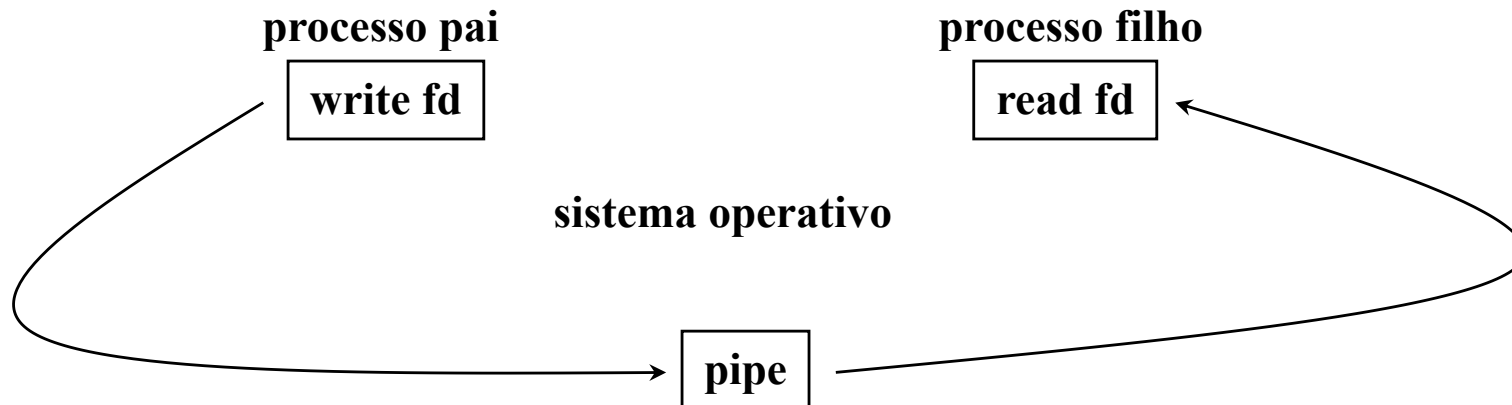
    pipe(fds);

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

Criar e usar pipe (exemplo inútil!)



Como partir do exemplo (inútil) anterior para conseguir comunicação de processo pai para filho?



Como veremos mais adiante, no *fork* o processo filho herda o contexto núcleo do pai pelo que aí se encontram todos os ficheiros abertos e consequentemente os *pipes* que o pai criou



Criar e usar pipe

(Exemplo útil: comunicação pai-filho)

```
#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*/
        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();
}
}
```




Questões relacionadas com os *file descriptors*

- Os *file descriptors* do processo pai e do filho permitiriam ambos ler e escrever, mas esta não é a utilização habitual porque seria impossível estabelecer uma sincronização entre as leituras e escritas
- O canal torna-se unidireccional fechando as extremidades não utilizadas
- É boa pratica fechá-las sempre porque o numero de *file descriptors* de um processo é limitado
- Fechar a extremidade de escrita de um pipe permite também desbloquear processos bloqueados na extremidade de leitura:
 - Neste caso é gerado um *End-Of-File*(EOF) que permite sair do *read*
 - Relembrar que a *read* nos pipes é bloqueante!



Redirecção de entradas/saídas com pipes



Relembrar ficheiros e Entradas/Saídas

- No Unix os ficheiros normais são meros vectores de *bytes*, portanto idênticos às interfaces dos periféricos habituais como o teclado ou o ecrã
- 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`), normalmente idêntico ao `stdout`
- Desta forma, sem efectuar qualquer operação, o processo pode receber *inputs* e efectuar *outputs* como estamos todos habituados na programação



Tabela de Ficheiros Abertos

- Como sabemos, as funções sistema sobre os ficheiros usam geralmente para designar o ficheiro um *file descriptor* cujo valor é obtido tipicamente no *open*
- Os *file descriptors* são inteiros que referenciam a tabela de ficheiros abertos do processo
- A tabela de ficheiros abertos é um simples vector (finito) que é gerido pelo núcleo, preenchendo e libertando registos à medida que os ficheiro são abertos ou fechados



A chamada sistema *dup*

- A função `int dup(int fd)` cria um novo *file descriptor* para um ficheiro aberto já existente.
- O novo *file descriptor* é um duplicado totalmente idêntico ao original:
 - o mesmo *file pointer* (posição de leitura/escrita)
 - o mesmo modo de acesso (*read, write or read/write*)

o novo *file descriptor* é o que o núcleo encontra livre com o **número mais baixo** na tabela de ficheiros abertos



File descriptors e dup

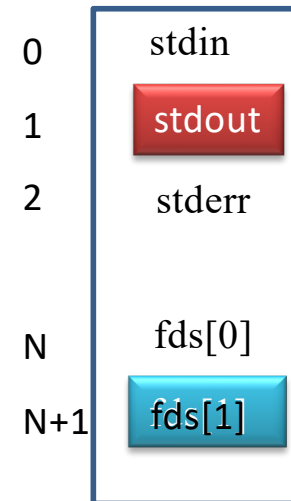
- Quando executamos `pipe()`, a chamada devolve 2 *file descriptors* um para cada extremidade
- Se, por exemplo, fecharmos o `stdout` e executarmos `dup()` da extremidade de escrita do *pipe* esta irá ocupar a posição 1 da tabela que entretanto tinha ficado vazia

```
int fds[2];

pipe(fds);

/* outras operações como criação de processo filho */

close(STDOUT_FILENO);
dup (fds[1])
```





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]);
```

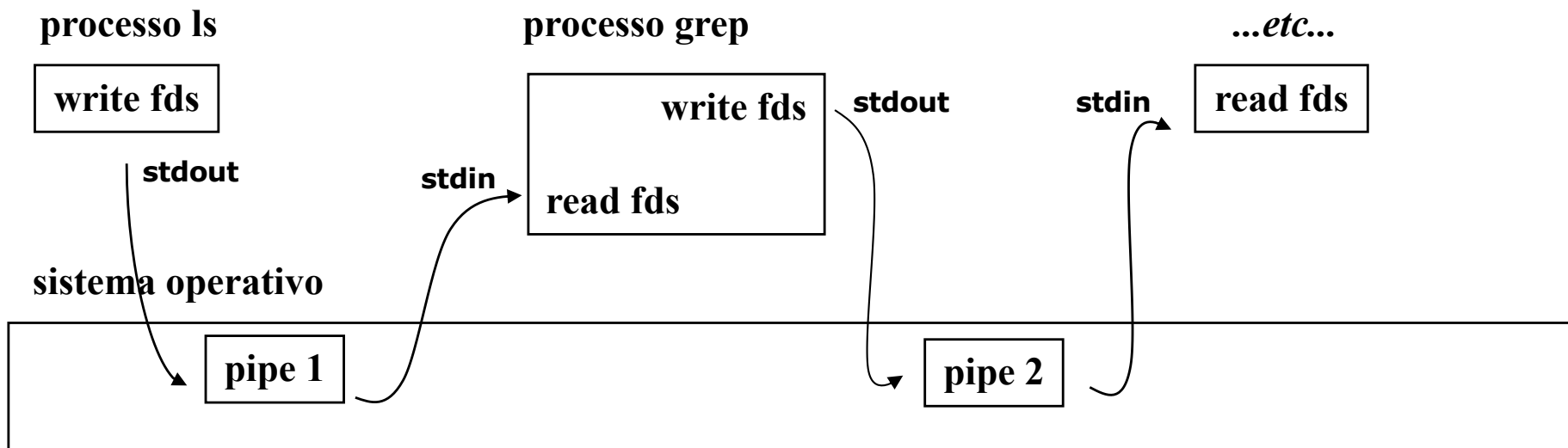
```
/* fecha os descritores não usados pelo
filho */
        close (fds[0]);
        close (fds[1]);

    /* 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();
    }
}
```

Redirecionamento de Entradas/Saídas no Shell

exemplo:

`ls -la | grep xpto | ...etc...`



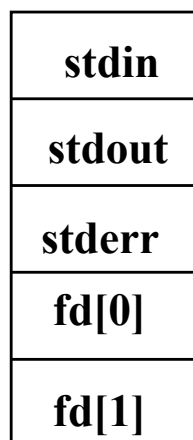
Como implementar a situação acima?

Redirecionamento de Entradas/Saídas (2)

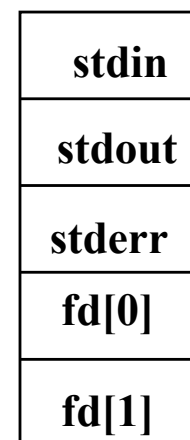
```
pipe(fds[0], fds[1]);
p = fork();
if (p>0) {
    close (1);
    dup (fds[1]);
    close (fds[0]);
    close (fds[1]);
    execv(...);
}
```

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

pai



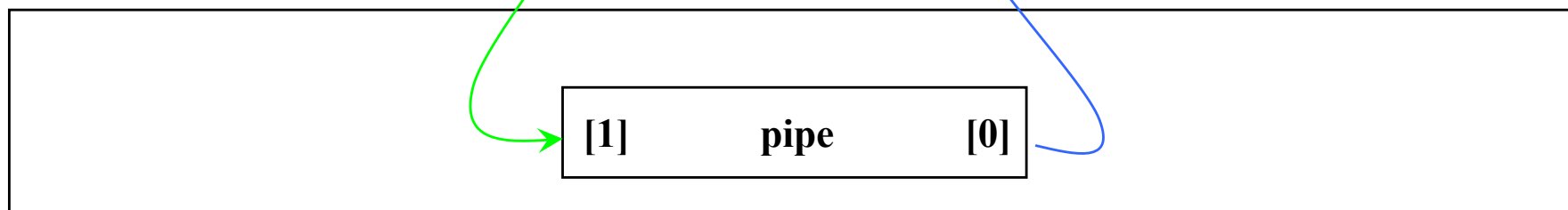
filho



```
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

[1] pipe [0]





Reutilização

- A possibilidade de reutilização de programas já existentes, é muito interessante
- É importante perceber que nada foi modificado nos programas dos comandos que foram executados deste modo. **O código é absolutamente o mesmo** (nem poderia deixar de ser uma vez que nada compilamos)
- Reutilizar código existente é um dos objectivos mais difíceis em Engenharia de Software, o Unix encontrou uma forma muito eficaz de o fazer



IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- 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

- Criar 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];

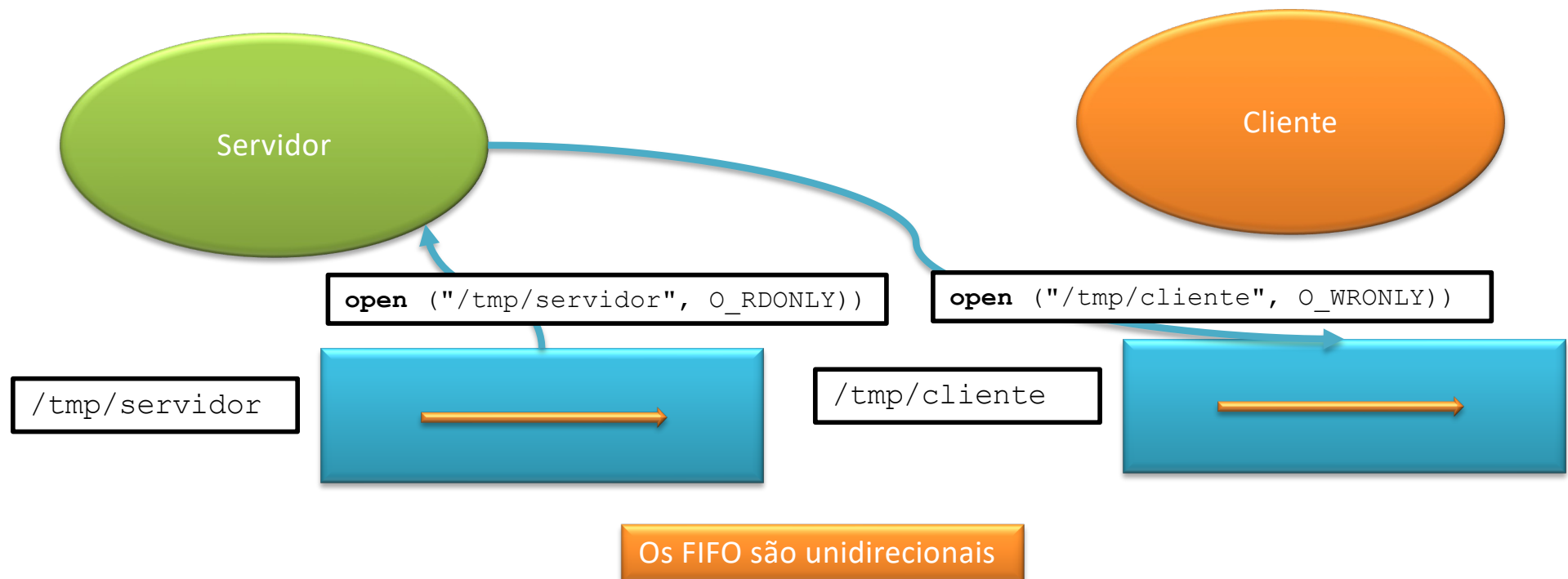
    unlink("/tmp/servidor");
    unlink("/tmp/cliente");

    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");
}
```

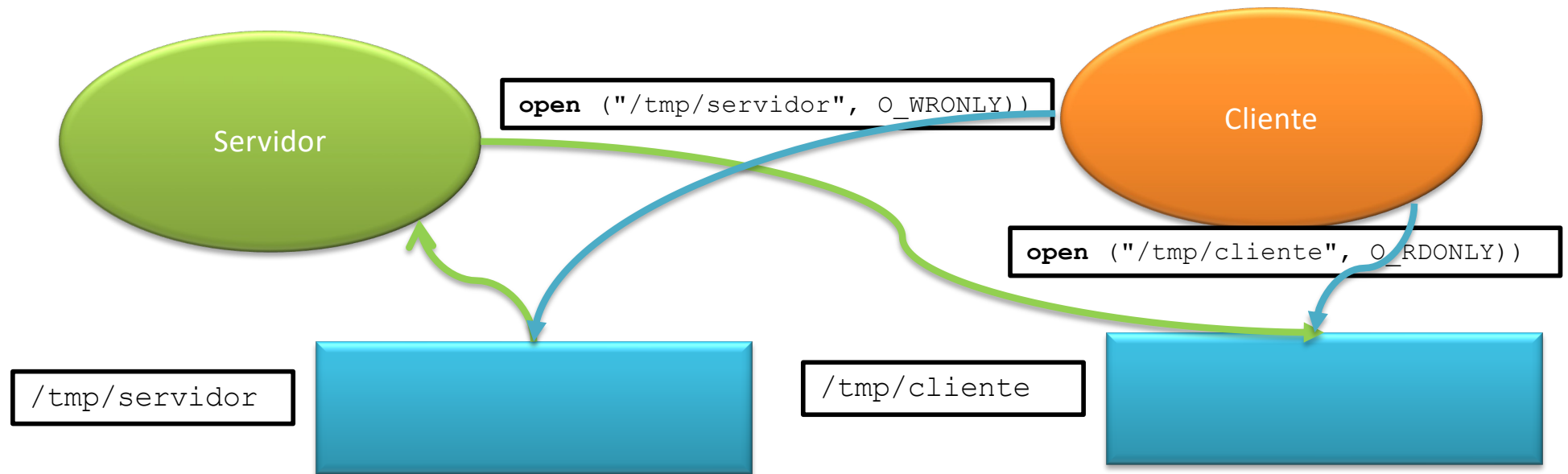
Servidor cria o Canal de IPC



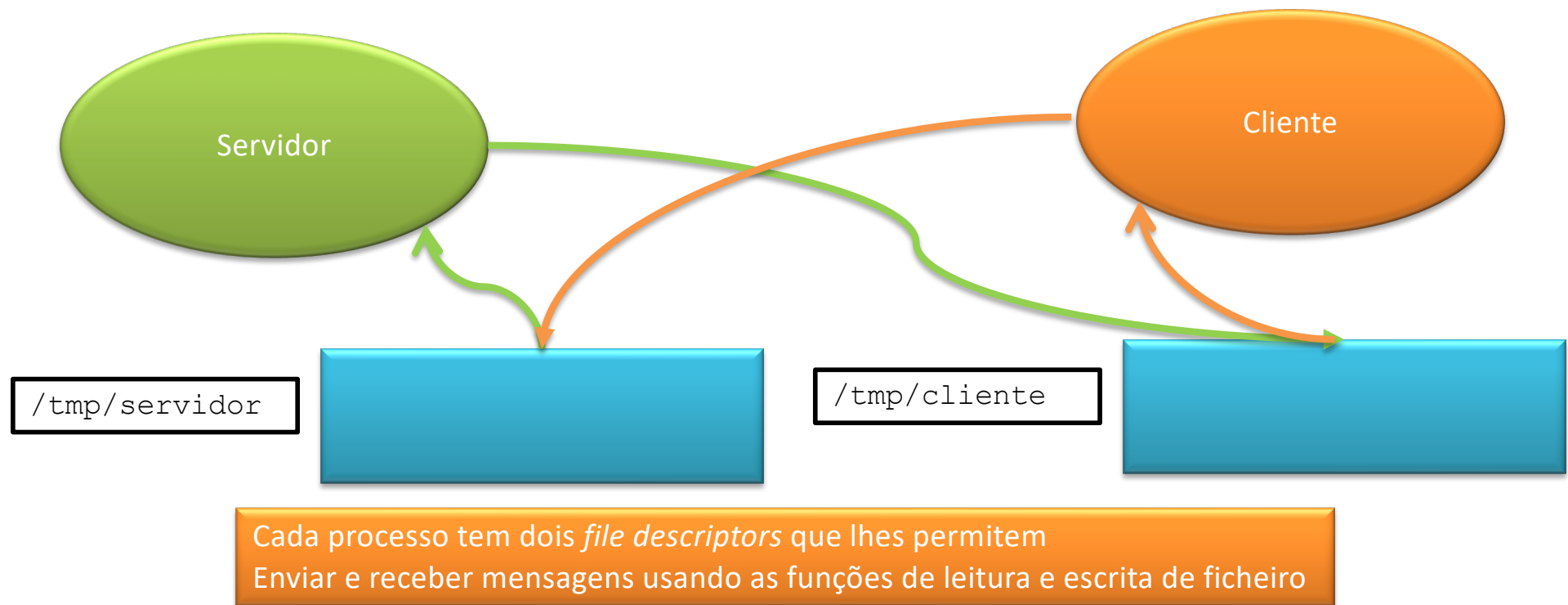
Semântica da abertura do FIFO

- Sincronização na abertura é diferente dos ficheiros. Devido a semântica de que um *pipe* é para dois processo comunicarem, se apenas um processo faz `open` do *pipe* (leitor ou escritor) ficará bloqueado até o outro abrir (há a possibilidade de não bloquear com uma *flag* específica)
- Se já houver outros processos a usar o *pipe* o `open` retorna de imediato

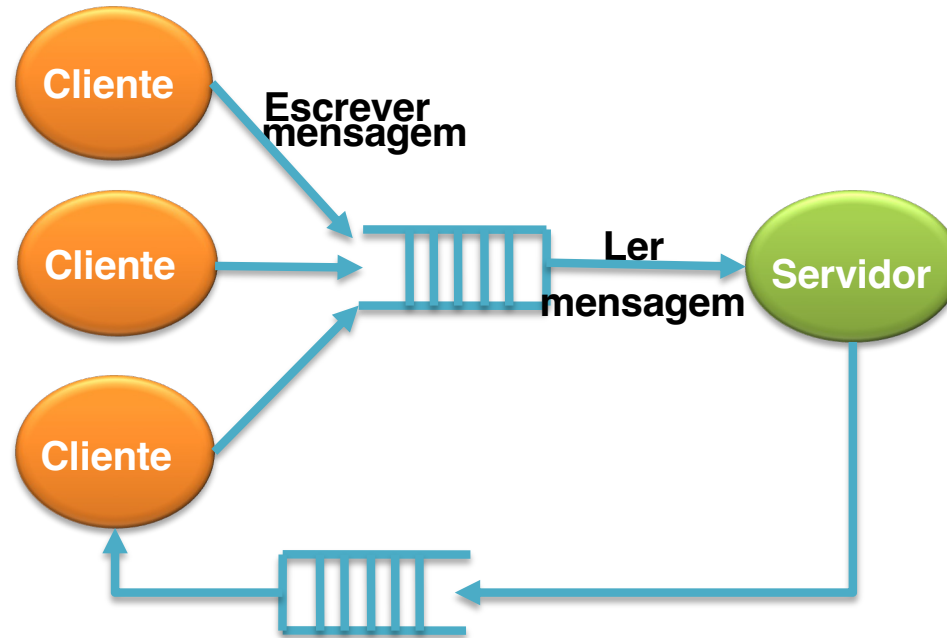
Cliente associa-se ao canal



Canal estabelecido

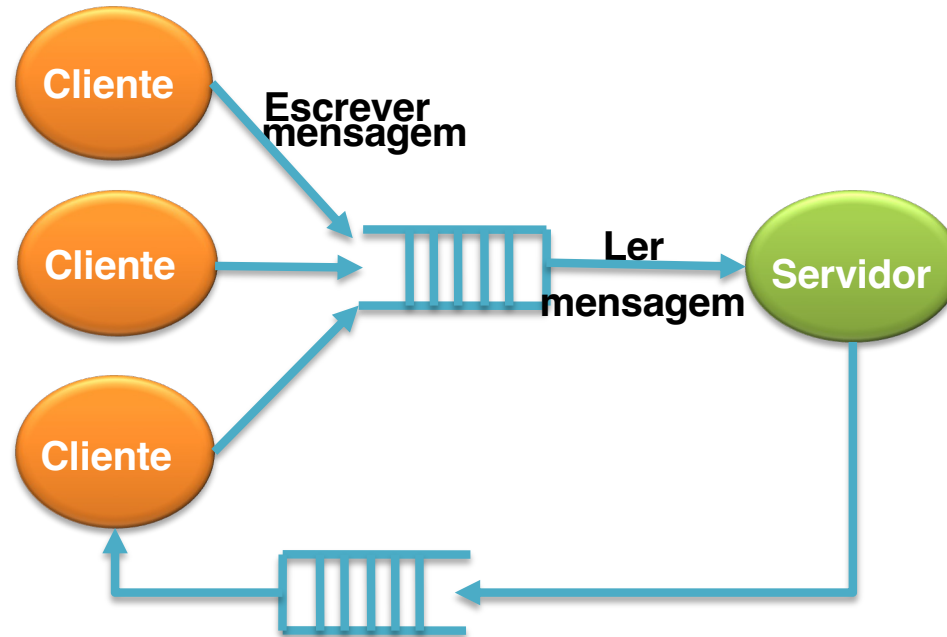


Como criar um cliente-servidor com FIFOs



- O servidor cria um FIFO
- Cada cliente envia uma mensagem para o servidor
- Cada cliente terá um FIFO para receber as mensagens de resposta

Problemas!



1. Como é que os clientes sabem o nome do FIFO do servidor?
2. Como distinguir as mensagens uma vez que o *pipe* é *bytestream*
3. O servidor tem de conhecer os nomes dos FIFOs dos clientes.
 - Conceptualmente, o servidor poderia responder para um FIFO único,...
 - mas a concorrência na leitura dos diferentes clientes tornaria a comunicação aleatória!

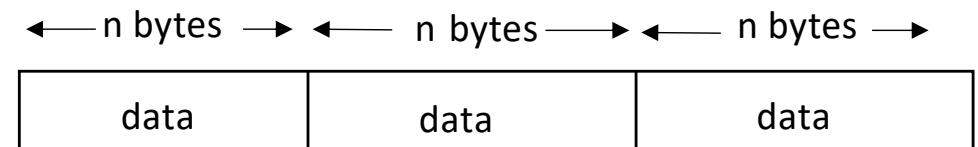
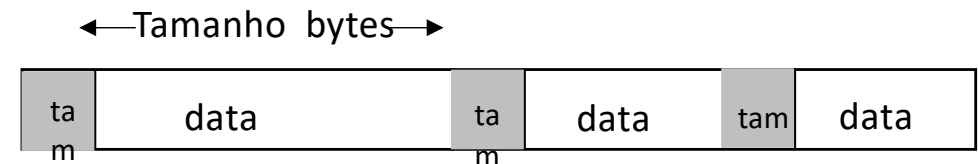
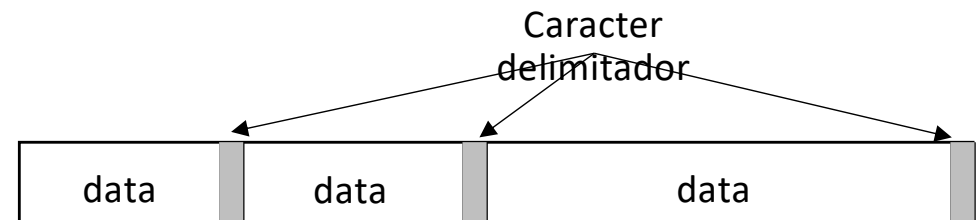


Gestão de Nomes

- Problema clássico nos sistemas cliente servidor
- Uma solução típica é o FIFO do servidor ter um nome pré-estabelecido (*well known name*). É a solução adotada nos exemplos, colocando na diretoria `/tmp` por portabilidade (em termos de segurança não é recomendável em aplicações reais, colocar acesso a servidor em diretorias publicas deste tipo).
- Pode haver um gestor de nomes, solução mais complexas e que repõe o problema anterior no endereço do servidor de nomes...
- Existe também um problema de gestão de nomes nos clientes que têm de enviar ao servidor o nome do FIFO de resposta. Podem fazê-lo enviando na mensagem ou existindo uma convenção de nomes em que o PID do processo entra na geração do nome

Formatação das mensagens

- Como o canal é *bytestream* podemos:
 - Colocar um caracter como delimitador ex.: *newline*. Implica que a mensagem tem de ser lida caracter a caracter para detetar o delimitador
 - Mensagens com cabeçalho que indique o tamanho da mensagem. O servidor lê o cabeçalho e depois pode efetuar uma leitura da restante informação. Se houver qualquer problema com o tamanho nos cabeçalhos o sistema fica dessincronizado
 - Mensagens de tamanho fixo. Geralmente desperdiça espaço das mensagens e limita mensagens grandes





IPC no UNIX

- Mecanismo inicial:
 - pipes
- Extensão dos pipes:
 - pipes com nome
- 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

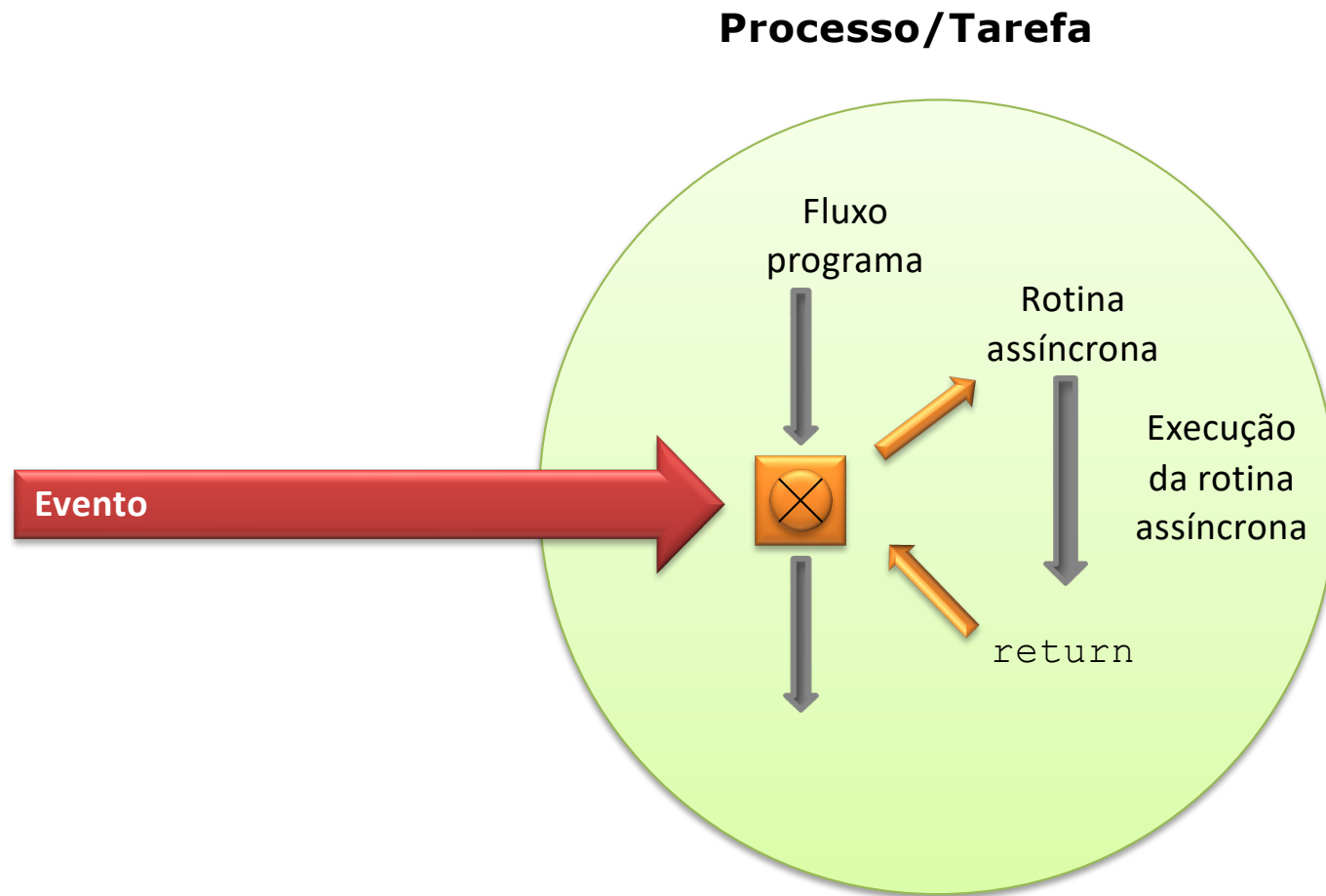
- Rotinas Assíncronas para Tratamento de acontecimentos assíncronos e exceções



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



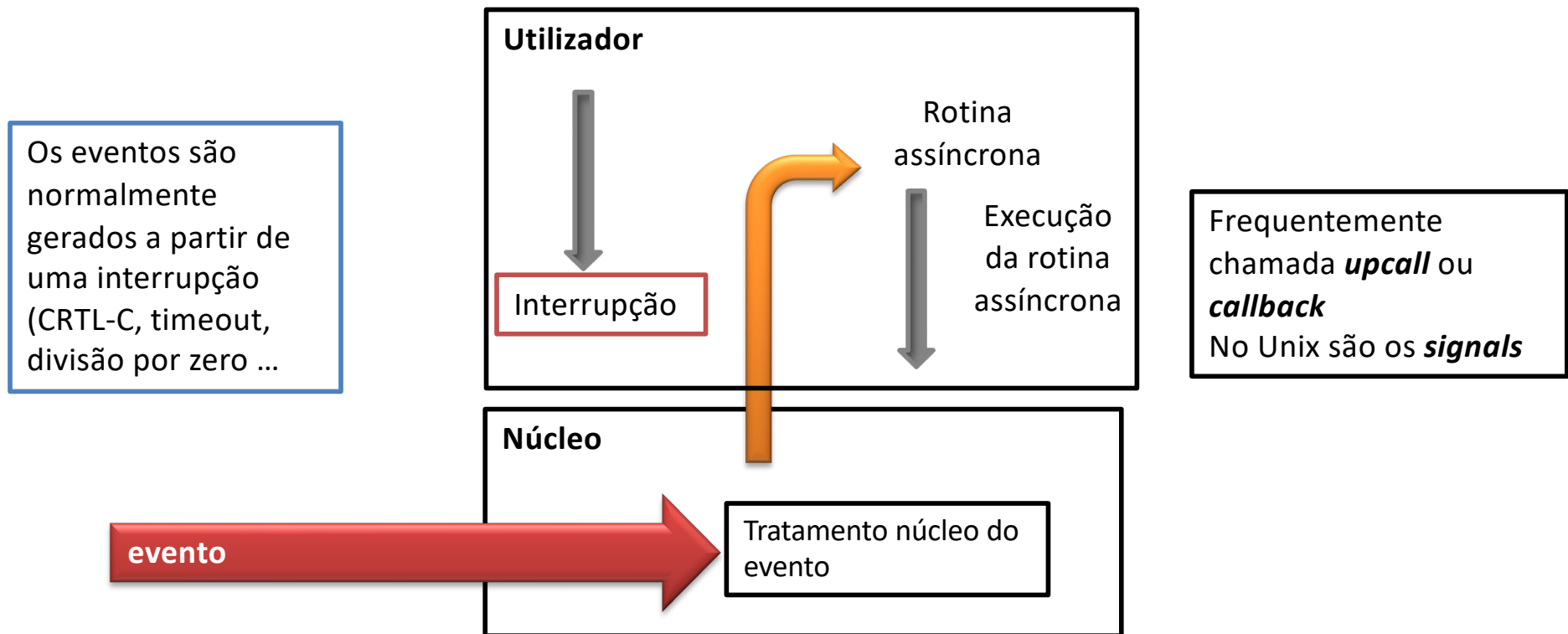
Parecido com outro conceito ?



Semelhante a interrupções

- O tratamento de eventos assíncronos é semelhante às interrupções, muitas vezes são designadas ***software interrupts***
- Mas, já foi frisado que as interrupções são executadas em modo núcleo e não no código utilizador
- Se fossem rotinas de interrupção não poderiam ser rotinas normais incluídas nos programas

Relação dos eventos com as interrupções



Que problemas temos de resolver

- O que faz o núcleo se o evento não for tratado?
- Como sabe o núcleo qual rotina invocar na *upcall*?
- Como colocar a rotina em execução?
- Durante a execução de uma *upcall* podem ocorrer outros eventos, permitir *upcalls* sucessivas?
- Se o processo receber múltiplos *signals* são todos tratados?

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);
}
```

Invocar printf/getchar num
sig_handler não é seguro,
como veremos mais a frente!

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



Quando é que o processo trata

- O *signal* não é tratado imediatamente apenas quando o processo for novamente escolhido para se executar, na passagem de modo núcleo a modo utilizador o despacho testa se há *signals* pendentes para o processo
- Enquanto não for tratado, o *signal* é memorizado no contexto do processo e está no estado pendente
- O núcleo não efetua uma contagem dos *signals*, apenas regista que enquanto não for tratado há um pendente, mesmo que tenha recebidos vários *signals* idênticos. Portanto, não se pode programar assumindo que o processo recebe todos os *signals*



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**



Exemplo de funções não reentrantes: malloc

- Durante uma chamada a malloc/free, as listas de áreas já alocadas são alteradas
- Um *signal* pode ser recebido durante uma chamada à malloc, i.e., enquanto estas listas estão a ser manipuladas
- Neste caso, se o *signal handler* invoca também malloc, o resultado é imprevisível:
 - chamadas à malloc pelo sig_handler podem observar estados inconsistentes!



Outros exemplos de funções não reentrantes

- Funções do `stdio.h`, como `printf`, `scanf`, `getc`:
 - num *sig handler* é recomendado usar `write` e `read`, que são reentrantes
- Funções da `pthread`, como `pthread_mutex_lock`:
 - antes de aceder a áreas de memória partilhada com o `sig_handler`, o processo deve bloquear a receção de signals
 - Trata-se de uma solução cara e complexa...
 - ...portanto, é boa prática minimizar a partilha de memória entre o processo principal e `sig_handlers`



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);
}
```


Mascarar os *signals*

- É possível programaticamente mascarar os *signals* durante a execução de um segmento de código por exemplo para garantir que não há concorrência entre o programa e o handler

```
int sigprocmask(int how, const sigset_t *restrict set,  
                sigset_t *restrict oldset);
```

Define *Block, Unblock, Setmask*

Máscara a aplicar

Máscara anterior

Pipes e signals

- Quando se escreve para um *pipe* ou FIFO que não tem leitor (ex.: o processo cliente pode entretanto ter terminado) é enviado o *signal* `sigpipe` que se não tiver tratamento, termina o processo por omissão.
- No servidor é recomendável ignorar o *signal* para que o processo não termine quando tenta escrever para o FIFO de um cliente que por qualquer razão já não existe
- A exceção pode ser tratada porque o `write`, nesta situação, dá um erro de `EPIPE` que pode permitir ao servidor descartar esse cliente



Signals em processos multi-tarefa

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

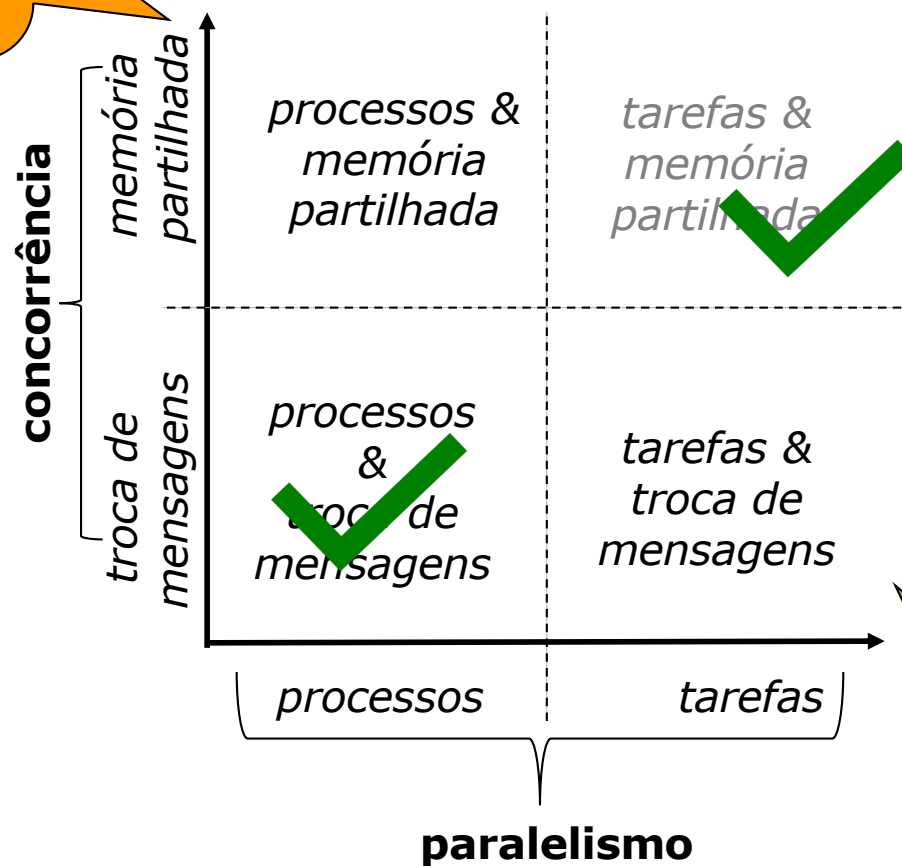
Qual das tarefas é interrompida para executar a função?

- 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

Restantes slides
dão pistas sobre
esta possibilidade

Mas está fora
da matéria



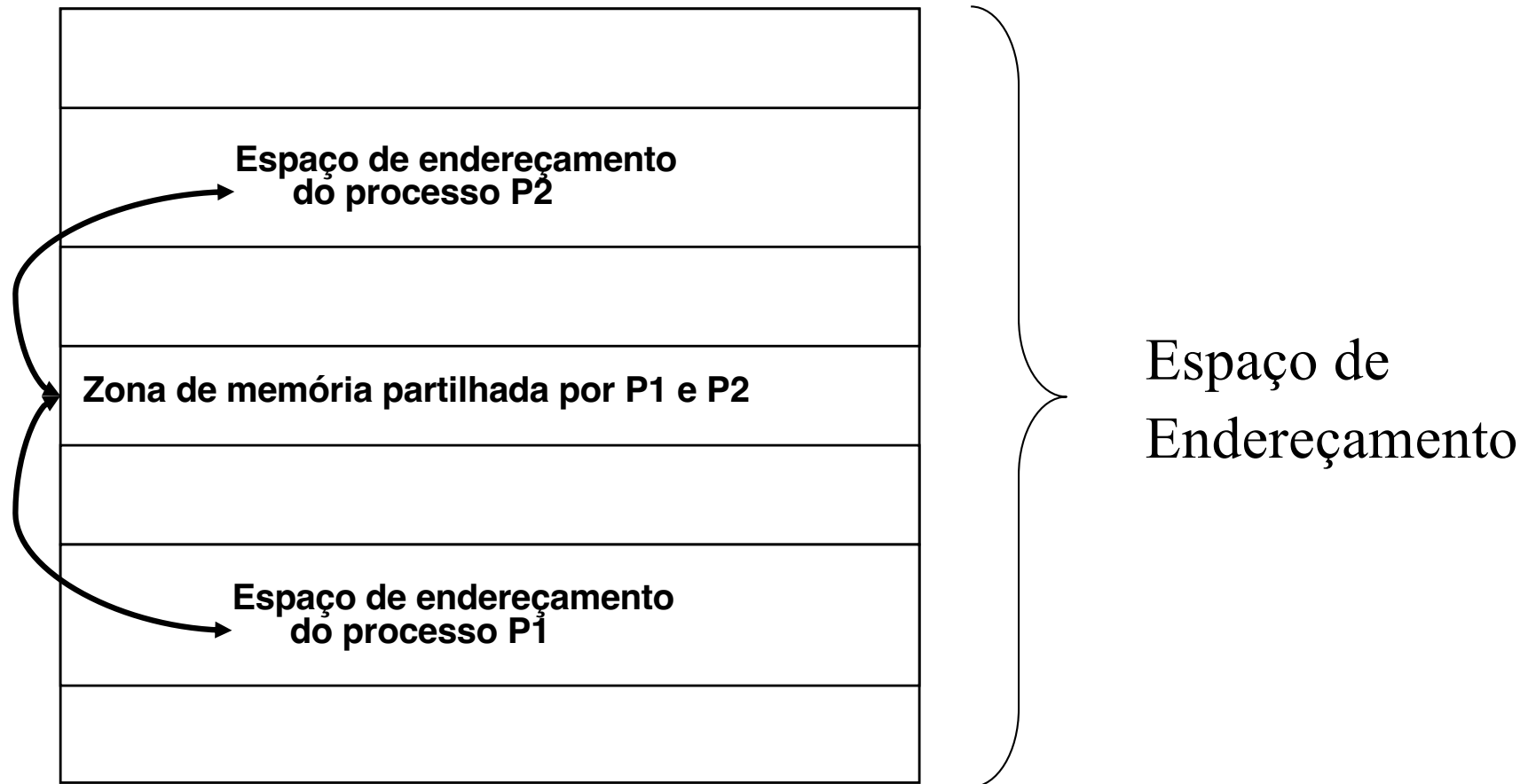
Existem
várias bibliotecas
(em *user level*)
que suportam
este caso



[Final da matéria avaliada deste capítulo]

Restantes slides apenas para referência
opcional (fora da matéria)

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-escretores)**



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)
```

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

Devolve o endereço
base em que o segmento
foi mapeado

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 regioao 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

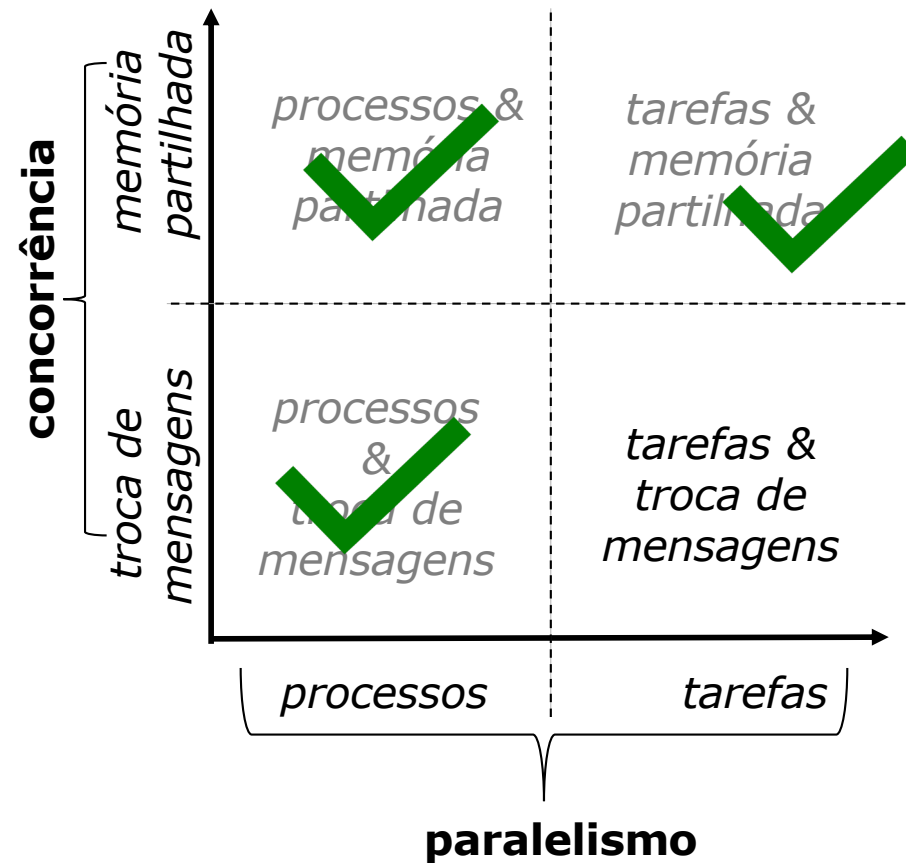


Então o que é melhor para comunicar entre processos?

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



Sockets



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



O que é um socket?

- Um socket é uma extremidade de um canal de comunicação
- Num exemplo com dois processos, A e B, a comunicar, existirão (pelo menos) **dois sockets**
 - Um socket local a A
 - Um socket local a B
- Cada socket pode ter um nome (*socket address*) associado
 - Para permitir que outros processos o referenciem
- Contraste: como era com pipes?



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)



Nome de um socket (*socket address*)

```
/* 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)
```



Antes de avançarmos, o objetivo para hoje...

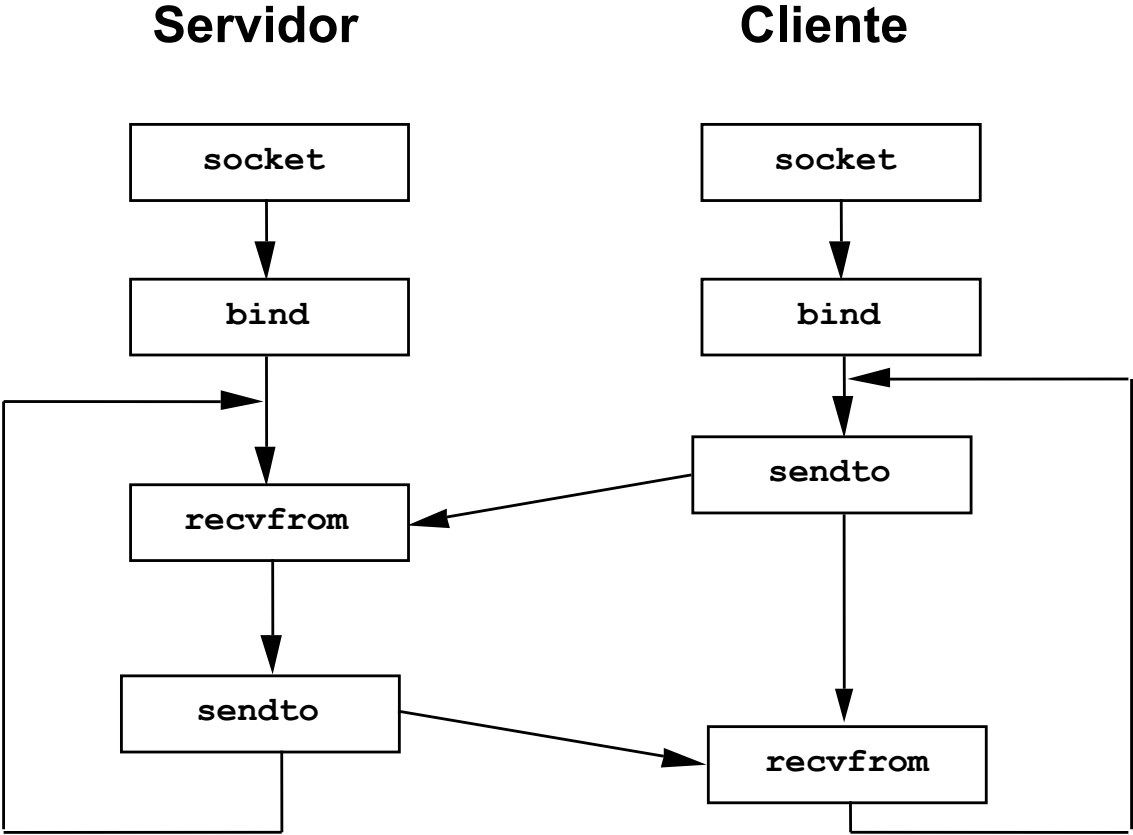
- Aplicação cliente-servidor
 - Usando sockets UNIX (cliente e servidor na mesma máquina)
 - Experimentando com ambos os tipos de socket
- Comportamento:
 - Cliente string com seu nome ao servidor
 - Servidor responde com frase simpática



Relembrando: tipos de socket

- Sockets sem ligação (*datagram*):
 - Modelo de comunicação tipo correio
 - Canal sem ligação, bidireccional, não fiável, interface tipo mensagem
- Sockets com ligação (*stream*):
 - Modelo de comunicação tipo diálogo
 - Canal com ligação, bidireccional, fiável, interface tipo sequência de octetos

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)
```

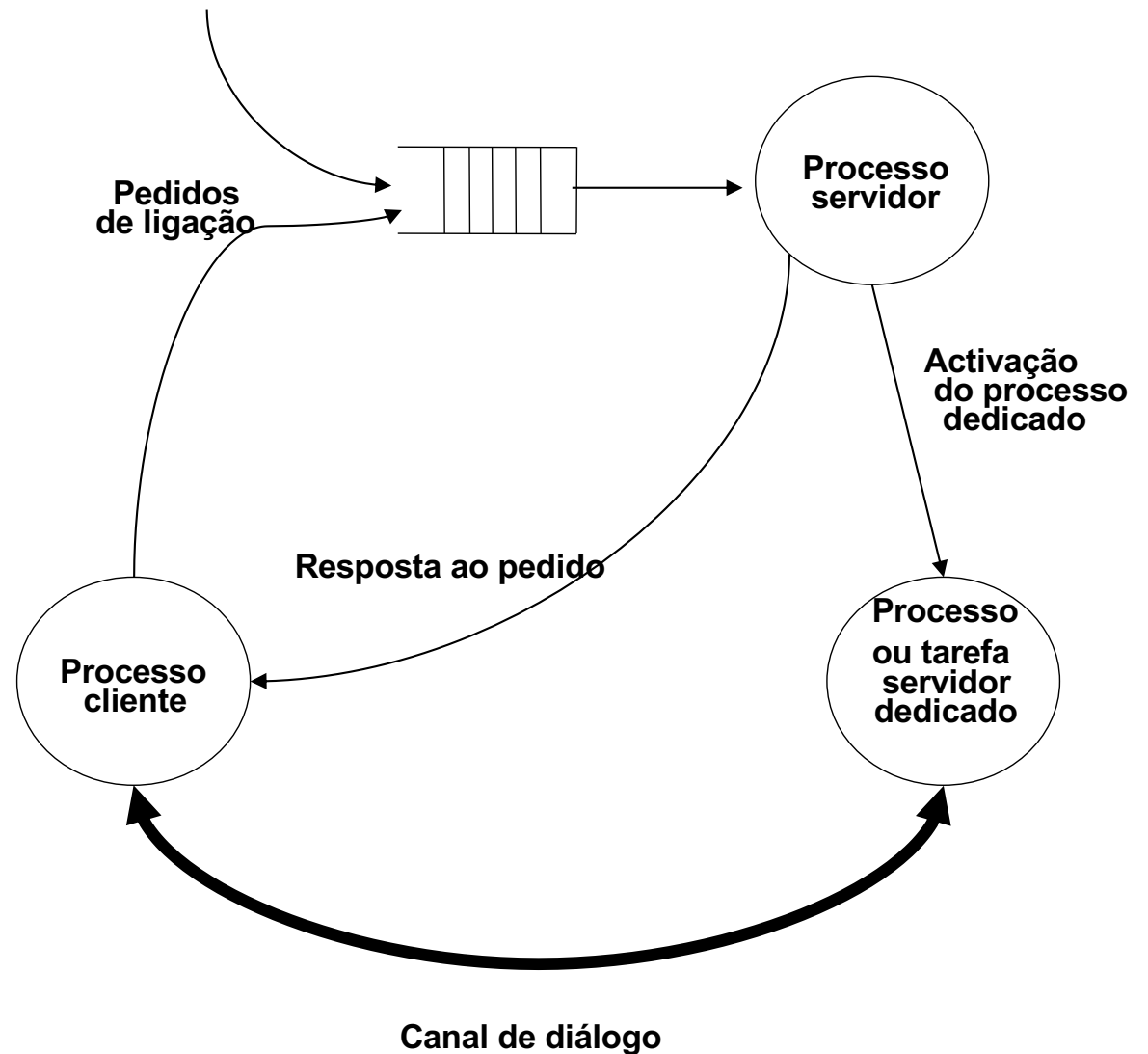


Sockets com e sem Ligação

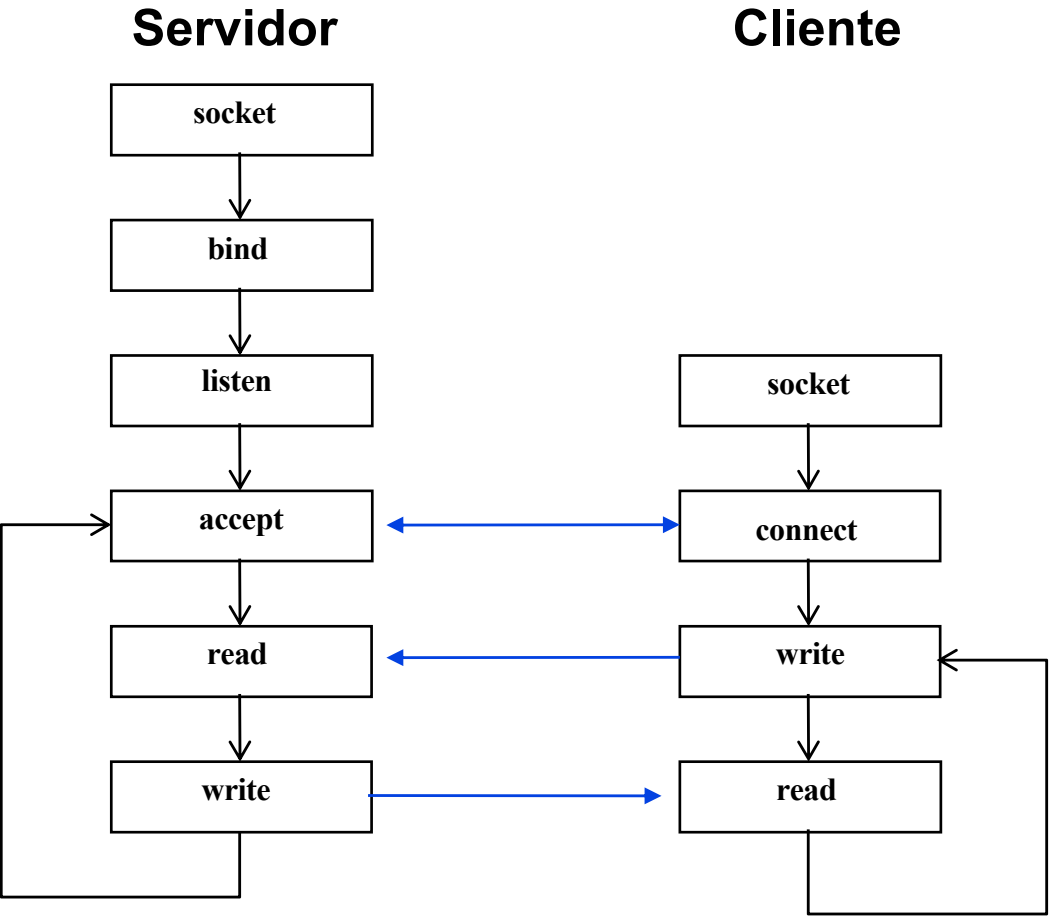
- Sockets com ligação:
 - Modelo de comunicação tipo diálogo
 - Canal com ligação, bidireccional, fiável, interface tipo sequência de octetos
- Sockets sem ligação:
 - Modelo de comunicação tipo correio
 - Canal sem ligação, bidireccional, não fiável, interface tipo mensagem

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



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)`



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)

```
/*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);
}
```

```
/*
- 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);
```



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! */  
}  
}
```