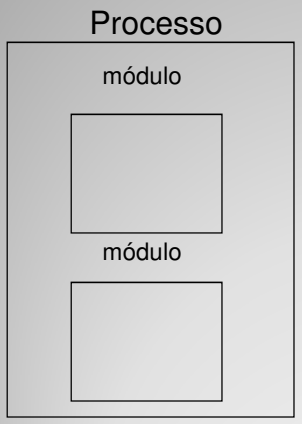
IPC no UNIX

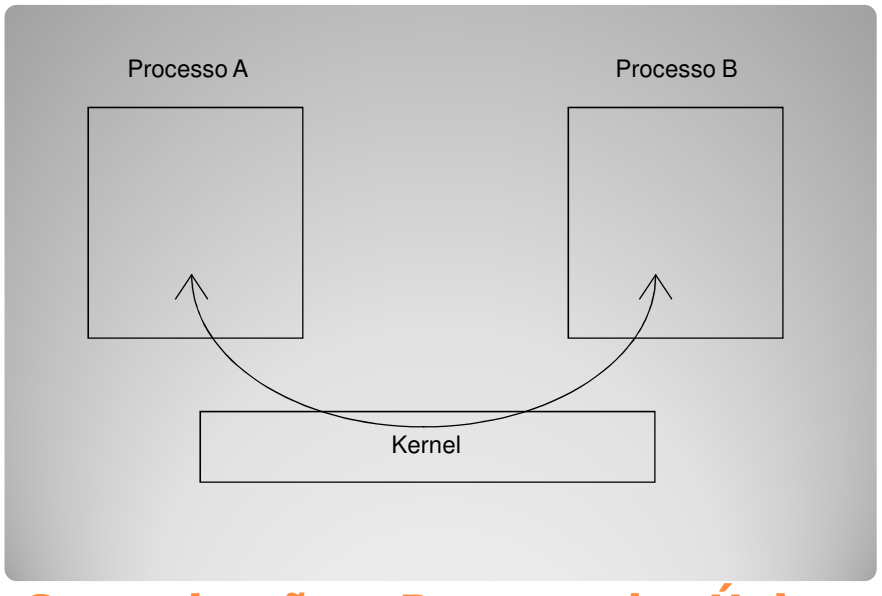
Onofre Trindade Jr

Comunicação – Processo Único

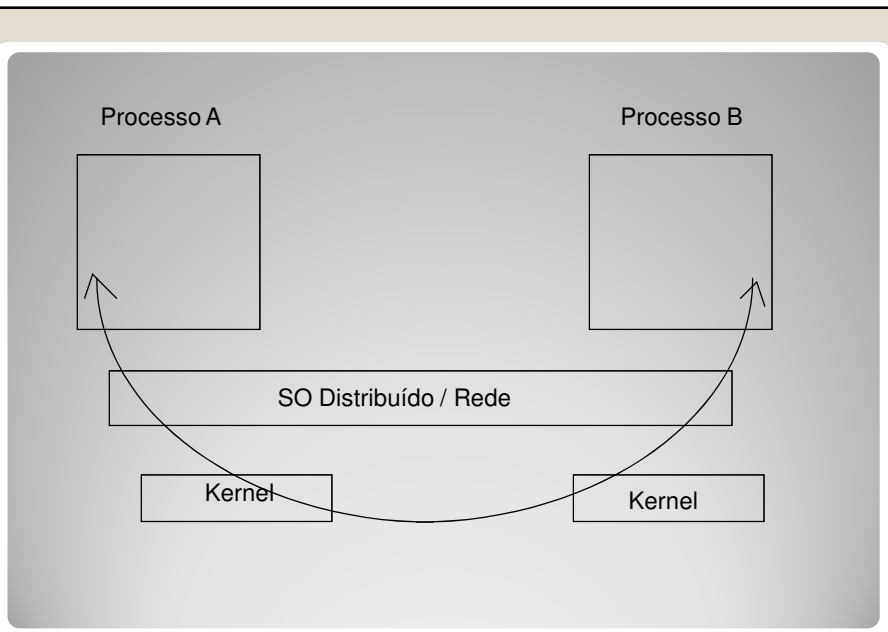


Comunicação entre módulos

- Variáveis globais
- Chamada de funções
  - parâmetros
  - resultados



Comunicação – Processador Único



Comunicação – Dois Processadores

Sinais (Interrupções de Software)
Pipes e FIFOs
Filas de Mensagens
Memória Compartilhada
Semáforos
Sockets
RPC (Remote Procedure Calls)
Mutex e Variáveis Condicionais

IPC no UNIX

- Interrupções de Software
- Uma notificação para um processo da ocorrência de um evento
- Sinais podem ser enviados por:
  - De um processo para outro processo
  - Do kernel para um processo

Sinais

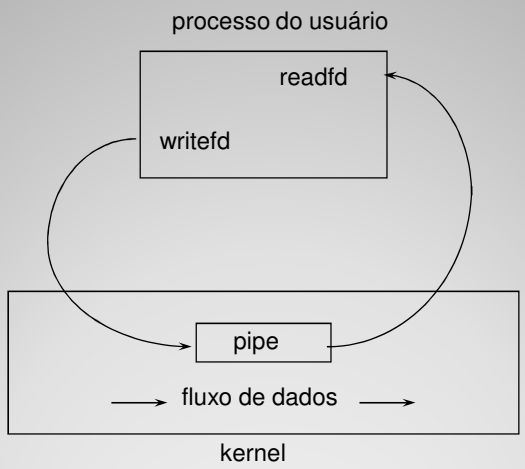
Pipes e Fifos

- Um pipe provê um fluxo unidirecional de dados
  - exemplo: `who | sort| lpr`
- Pipe é uma estrutura de dados circular do Kernel
- Somente um processo pode acessar um pipe de cada vez
- Um pipe é criado pela system call

```
int pipe(int* filedes);
```

- Dois descritores de arquivos são retornados
  - filedes[0], aberto para leitura
  - filedes[1], aberto para escrita
- Processos que compatilham um pipe devem ter um parente em comum
- Fifos têm um nome associado com elas permitindo a comunicação entre processos não relacionados

Pipes



- Dois tipos de sockets no UNIX:
  - Internet - dois processos em máquinas diferentes
  - UNIX – dois processos na mesma máquina
- Pipes são implementados utilizando-se sockets UNIX

Implementação de Pipes no Unix

```
main()
{
  int pipefd[2], n;
  char buff[100];
  if (pipe(pipefd) < 0) {
    error ("pipe error");
  }
  printf ("readfd = %d, writefd = %d\n",
           pipefd[0], pipefd[1]);
  if (write(pipefd[1], "hello world\n", 12) != 12) {
    error ("write error");
  }
  if ((n=read(pipefd[0], buff, sizeof[buff])) < 0) {
    error ("read error");
  }
  write (1, buff, n);
  close(pipefd[0]);
  close(pipefd[1]);
  exit (0);
}
```

Exemplo 1

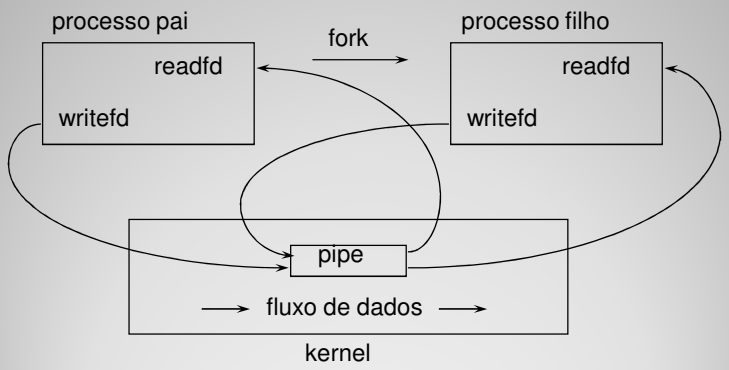
- O tamanho de um pipe é definido, isto é, somente uma quantidade definida de bytes pode permanecer no pipe sem ser lida
- Se uma escrita é feita em um pipe e existe espaço suficiente, a chamada retorna imediatamente
- Se uma escrita é feita e não existe espaço suficiente, a execução do processo é suspensa até que um outro processo leia dados do pipe e libere o espaço necessário
- Tamanho típico de 512 bytes (Mínimo definido pelo POSIX)

Tamanho dos Pipes

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

Tamanho dos Pipes - Exemplo

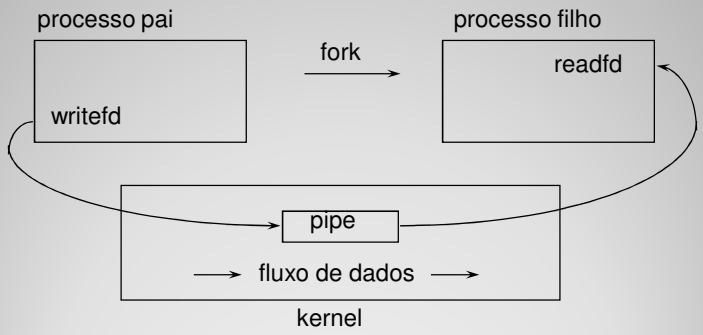
- O processo cria um pipe e depois utiliza fork para criar uma cópia



Criação de Pipes

Pai abre, filho lê

- pai fecha o lado de leitura
- filho fecha o lado de escrita

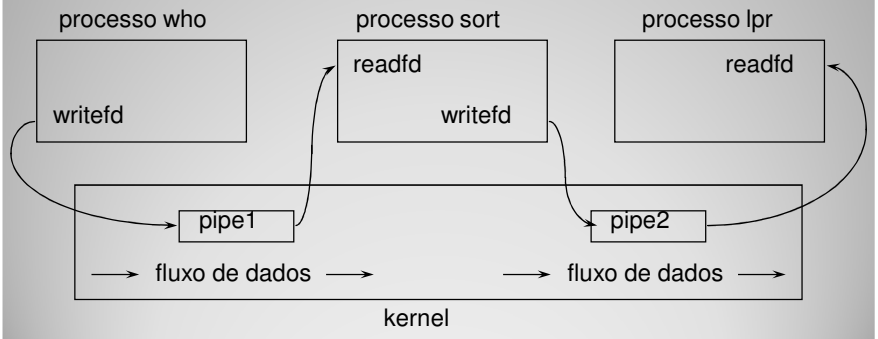


Criação de Pipes

Uso de Pipes

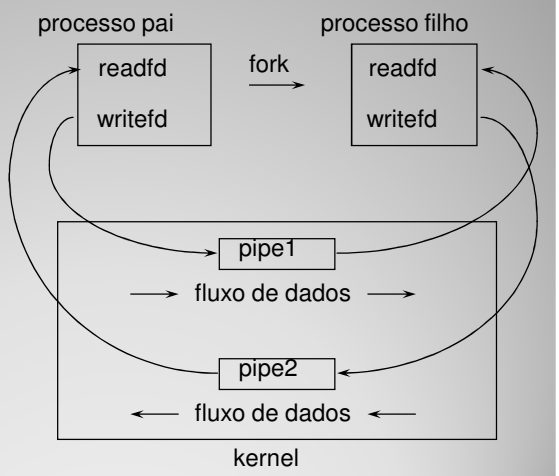
`who | sort | lpr`

who escreve no pipe 1
sort lê do pipe 1 e escreve no pipe 2
lpr lê do pipe 2

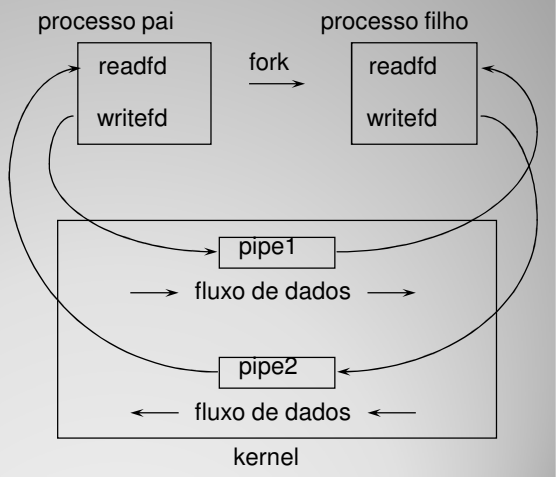


Uso de Pipes

pai envia nome de arquivo para filho (pipe 1) que lê o arquivo e devolve o conteúdo para o pai (pipe 2).



- pai cria pipe 1 e pipe 2
- fork
- pai fecha lado de leitura do pipe 1
- pai fecha lado de escrita do pipe 2
- filho fecha lado de escrita do pipe 1
- filho fecha lado de leitura do pipe 2



Uso de Pipes

- Processos podem ler e escrever em listas de mensagens (tal como mailboxes)
- System calls:
  - *int msgget (key_t key, int flag) :* Cria ou permite acesso à uma fila de mensagens, retorna o identificador da fila
  - *int msgsnd(int msqid, const void *msgp, size_t msgsz, int flag) :* Escreve uma mensagem na fila
  - *int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg)* : Recebe uma mensagem e a armazena em *msgp*
  - *msgtype*: As mensagens podem ter tipo e cada tipo define um canal de comunicação
  - *int msgctl(int msqid, int cmd, struct msqid_ds *buf)* : Provê operações de controle na fila de mensagens (e.x. remoção)
- O processo é bloqueado se:
  - tenta ler de uma fila vazia
  - tenta escrever em uma fila cheia

Mensagens

```
/* Send and receive messages within a process */

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

#defineBUFFSIZE      128
#definePERMS         0666

#define KEY ((key_t) 7777)

main()
{
  int  i, msqid;
  struct {
    long m_type;
    char m_text[BUFFSIZE];
  } msgbuffs, msgbuffr;

  if ( (msqid = msgget(KEY, PERMS | IPC_CREAT)) < 0)
     perror("msgget error");
  msgbuffs.m_type = 1L;
  strcpy(msgbuffs.m_text,"a REALLY boring message");
```

Mensagens - Exemplo

```
  if (msgsnd(msqid, &msgbuffs, BUFFSIZE, 0) < 0)
     perror("msgsnd error");

  printf("the message sent is: %s \n", msgbuffs.m_text);

  if (msgrcv(msqid, &msgbuffr, BUFFSIZE, 0L, 0) !=
  BUFFSIZE)
    perror("msgrcv error");

  printf("the message received is: %s \n",
  msgbuffr.m_text);

  // remove msg
  if (msgctl(msqid, IPC_RMID, (struct msqid_ds *) 0) < 0)
       perror("IPC_RMID error");

  exit(0);
}
```

Mensagens - Exemplo

saída:
the message sent is: a REALLY boring message
the message received is: a REALLY boring message

Mensagens - Exemplo

- Um semáforo é um contador inteiro positivo normalmente usado para coordenar o acesso a recursos compatilhados
- System calls:
  - *int sema_init(sema_t *sp, unsigned int count, int type, void * arg):* Inicia o semáforo apontado por sp em count. *type* pode ser utilizado para atribuir diferentes tipos de comportamento ao semáforo
  - *int sema_destroy(sema_t *sp);* destrói qualquer estado relacionado ao semáforo apontado por sp. O espaço de memória alocado ao semáforo não é automaticamente liberado.
  - *int sema_wait(sema_t *sp):* bloqueia o thread até que o contador do semáforo apontado por sp seja maior que zero, e neste caso, decrementa o contador.

Semáforos

- *int sema_trywait(sema_t *sp):* decrementa atomicamente o contador do semáforo apontado por sp, se o contador é maior que zero. Caso contrário, retorna um erro
- *int sema_post(sema_t *sp);* incrementa atomicamente o o contador do semáforo apontado por sp. Se existirem threads bloqueados pelo semáforo, um será desbloqueado.

- Exemplo: O cliente aguardando na fila de um banco é análogo à sincronização provida pelas funções sema_wait() e sema_trywait():

Semáforos

```
#include <errno.h>
#define TELLERS 10
sema_t tellers; /* semaphore */
int banking_hours(), deposit_withdrawal;
void *customer(), do_business(), skip_banking_today();
  ...

sema_init(&tellers, TELLERS, USYNC_THREAD, NULL);
/* 10 tellers available */

while(banking_hours())
pthread_create(NULL, NULL, customer, deposit_withdrawal);
  ...

void * customer(int deposit_withdrawal)
{
  int this_customer, in_a_hurry = 50;
  this_customer = rand() % 100;
```

Semáforos - Exemplo

```
if (this_customer == in_a_hurry) {
   if (sema_trywait(&tellers) != 0)
      if (errno == EAGAIN) { /* no teller available */ {
         skip_banking_today(this_customer);
         return;
      } /* else go immediately to available teller and
        decrement tellers */
 ...

else {
 /* wait for next teller, then proceed, and decrement tellers */
    sema_wait(&tellers);
    do_business(deposit_withdrawal);
/* increment tellers; this_customer's teller is now available */
    sema_post(&tellers);
  }
```

Semáforos - Exemplo

- Processos podem compartilhar o mesmo segmento de memória quando ele é mapeado no espaço de endereçamento dos dois processos
- Comunicação mais rápida
- System calls:
  - *int shmget(key_t key, size_t size, int shmflg) :* cria uma nova região de memória compartilhada ou retorna uma existente
  - *void *shmat(int shmid, const void *shmaddr, int shmflg) :* agrupa uma região de memória compartilhada ao espaço de endereçamento do processo
  - *int shmdt(char *shmaddr):* desagrupa uma região de memória compartilhada
- As regiões de memória compartilhada devem ser acessadas por um processo de cada vez (exclusão mútua)

Memória Compartilhada

```
// IPC communication between a child and a parent process using
// shared memory : the parent puts messages into the shared memory
// the child reads the shared memory and prints the messages

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

#define SHMSIZE 15   // maximum number of bytes in a message
#define MESGNUM 2    //number of messages

void cleanup (int shm_id, char *addr); // cleanup procedure
int shm_id; // ID of shared memory

int main (int argc, char *argv[])
{
 char message [SHMSIZE]; // the message to send/receive
 int i;
 int number_of_messages; // number to be sent
 int nbytes;            // number of bytes in a message
```

Memória Compartilhada - Exemplo

```
int number_of_messages; // number to be sent
int nbytes;            // number of bytes in a message
int status;
char *addr = (char *) malloc (SHMSIZE * sizeof (char));

number_of_messages = MESGNUM;
nbytes = SHMSIZE;

// set up shared memory segment using PID as the key
if ((shm_id = shmget ((key_t) getpid(), SHMSIZE, 0666 | IPC_CREAT)) == -1 )
  {
    printf ("Error in shared memory region setup.\n");
    exit (2);
} // if shared memory get failed

// attach the shared memory segment
addr = (char *) shmat (shm_id, (char *) 0, 0);
```

Memória Compartilhada - Exemplo

```
if (fork ()) { // true if in parent process

   // create message of required length
   for (i=0; i < nbytes; i++)
     message [i] = i % 26 + 'a';
     message [nbytes] = '\0';

     // send message using the shared memory segment
     for (i = 0; i < number_of_messages; i++) {
       if (memcpy (addr, message, nbytes+1) == NULL) {
          puts ("Error in memory copy");
          cleanup (shm_id, addr);
          exit (3);
       } // end if error in memory copy
     } // end for as many messages as requested

 wait (&status); // wait for child to return

 // get the message sent by the child
 strcpy (message, addr);
 printf ("Parent - message from child: \n %s\n", message);
```

Memória Compartilhada - Exemplo

```
 strcpy (message, addr);
 printf ("Parent - message from child: \n %s\n", message);
 cleanup (shm_id, addr);
 exit(0);
} // end parent process

  // in child process
  puts ("Child - messages from parent:");

  for (i = 0; i < number_of_messages; i++) {
     if (memcpy (message, addr, nbytes+1) == NULL) {
         puts ("Error in memcpy");
         cleanup (shm_id, addr);
         exit (5);
     } // end if error in shared memory get
     else
        puts (message);
  } // end for each message sent

   strcpy (addr, "I have received your messages!");
```

Memória Compartilhada - Exemplo

```
 exit (0);
} // end main program

// remove shared memory segment
void cleanup (int shm_id, char *addr)
{
  shmdt (addr);
  shmctl (shm_id, IPC_RMID, 0);
} // end cleanup function
```

Memória Compartilhada - Exemplo

- **Saída:**
- Child - messages from parent:
- abcdefghijklmno
- abcdefghijklmno
- Parent - message from child:
- I have received your messages!

Memória Compartilhada - Exemplo

Sockets - Conceitos Básicos

- Sockets são uma forma de IPC (InterProcess Communication) definida no 4.3 BSD que fornecem comunicação entre processos residentes em sistema único ou processos residentes em sistemas remotos.
- Sockets criados por diferentes programas usam nomes para se referenciarem.
- Esses nomes geralmente devem ser traduzidos em endereços.

Tipos de Sockets

- STREAM SOCKET - Provê sequenciamento e fluxo bidirecional.
- No domínio UNIX, o SOCKET_STREAM trabalha igual a um pipe, no domínio INTERNET este tipo de socket é implementado sobre TCP/IP.

Tipos de Sockets

- SOCK_DGRAM - Suporta fluxo de dados bidirecional mas não oferece um serviço confiável como STREAM_SOCKET.
- Mensagens duplicadas e em ordem diferente são problemas que podem aparecer neste tipo de socket.

Tipos de Sockets

- RAW_SOCKET - permite o acesso a interface de protocolos de rede. Disponível para usuários avançados e que possuam autoridade de usuário root
- permite que uma aplicação acesse diretamente protocolos de comunicação de baixo nível
- permite a construção de novos protocolos sobre os protocolos de baixo nível já existentes

Associação

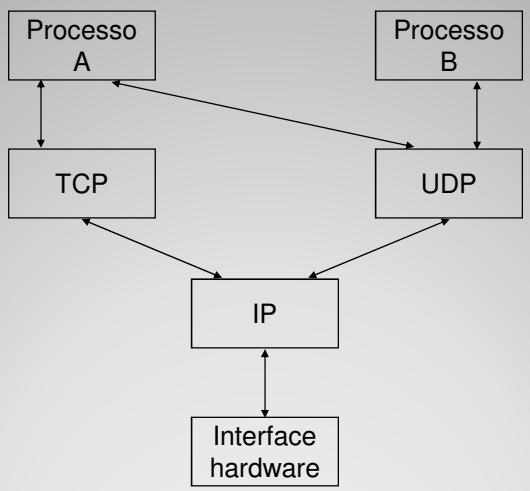
- O socket é criado sem nome
- É necessário um nome para para a sua utilização
- Os processos são ligados por uma associação
- Associação: <protocolo, end. máquina local, porta local, end. máquina remota, porta remota>

Domínios e Protocolos

- O espaço no qual o endereço é especificado é chamado de domínio
- Domínios básicos:
  - INTERNET - AF_INET - os endereços consistem do end. de rede da máquina e da identificação do no. da porta, o que permite a comunicação entre processos de sistemas diferentes
  - Unix: AF_UNIX - os processos se comunicam referenciando um pathname, dentro do espaço de nomes do sistema de arquivos

- Domínio Internet
  - Implementação Unix do protocolo TCP/UDP/IP
  - Consiste de:
    - end. de rede da máquina
    - identificação do no. da porta
  - Permite a comunicação entre máquinas diferentes
  - Conexões sob a forma de sockets do tipo stream e do tipo datagramas

Domínios e Protocolos

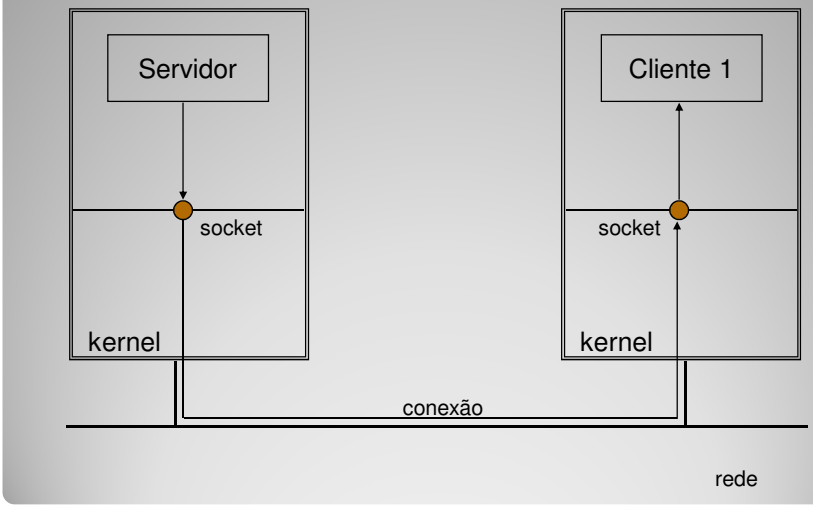


Protocolos TCP/IP

Definida por:

•Um protocolo: TCP ou UDP
• Endereço IP local
• Porta local
• Endereço IP remoto
• Porta distante

Associação



Comunicação por *sockets*

Cria um socket com
- Família (ou domínio): UNIX, Internet, XNS
- Tipo: stream, datagrama, raw
- Protocolo: TCP, UDP

Servidor

socket ( )

sockfd = (int) socket (int family, int type, int protocol)

Comunicação TCP

Servidor

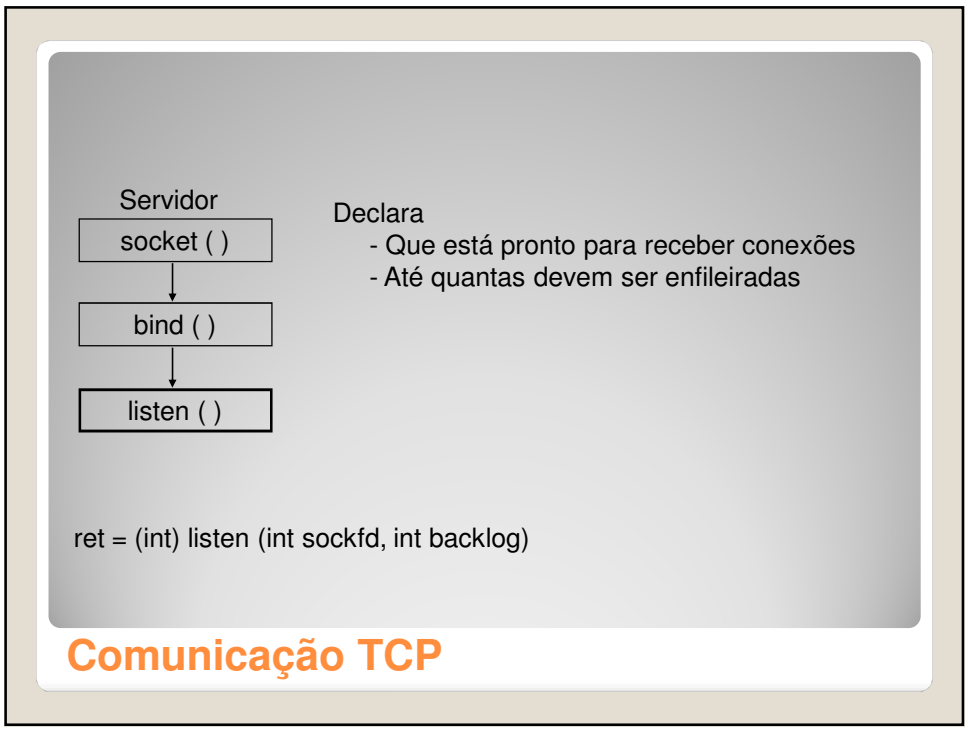
socket ( )

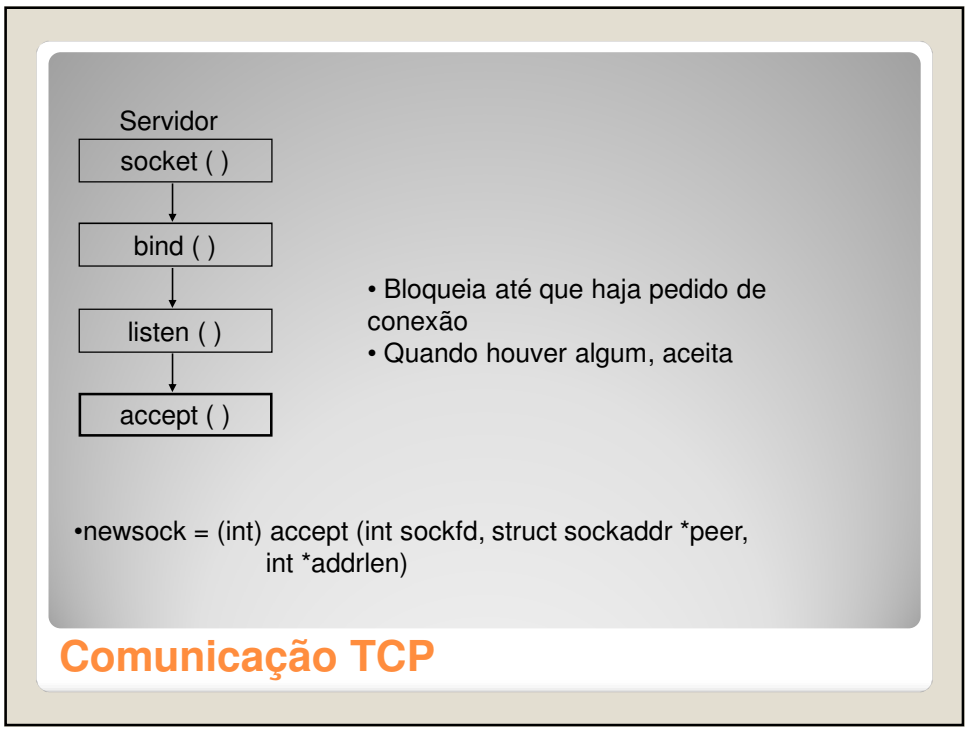
bind ( )

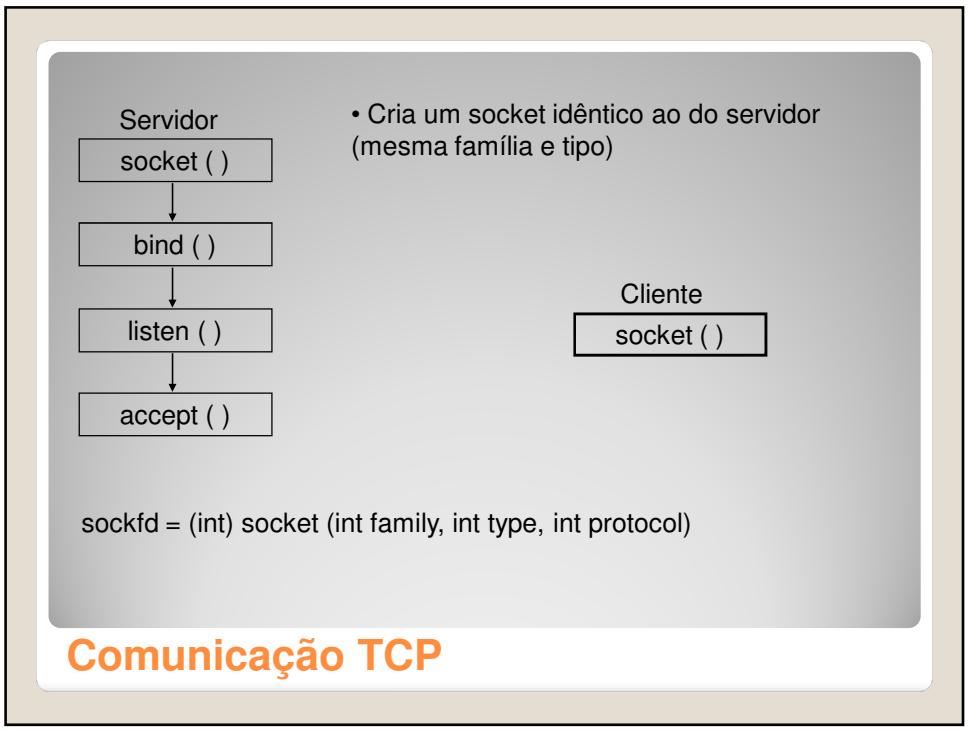
Atribui ao socket
- Endereço Internet (pode ser “any”)
- Porta de comunicação

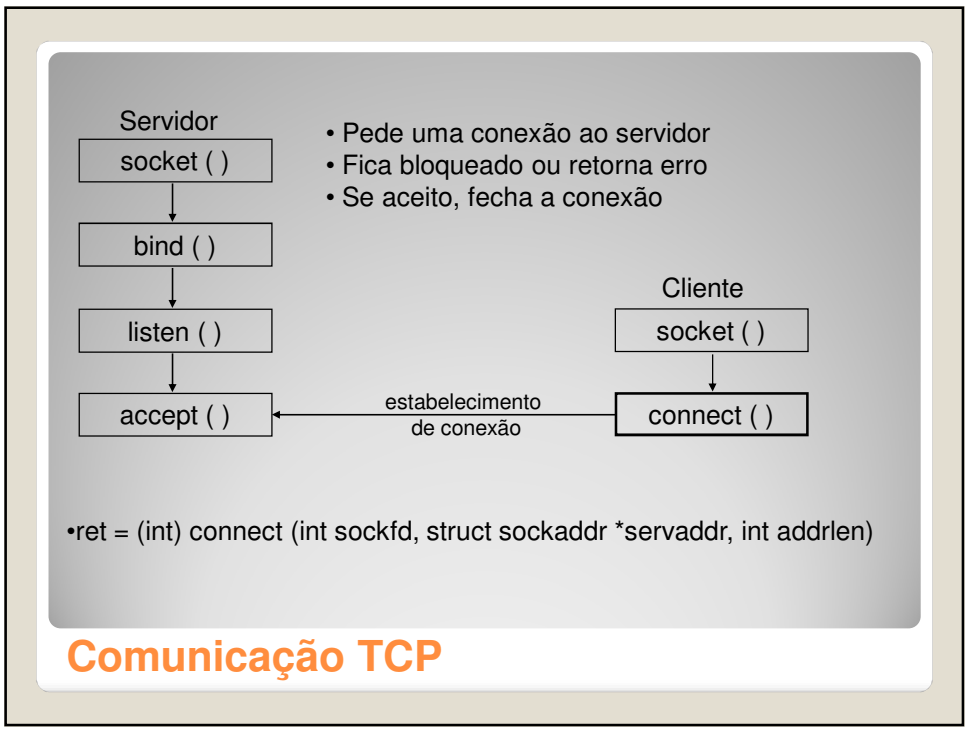
ret = (int) bind (int sockfd, struct sockaddr *myaddr, int addrlen)

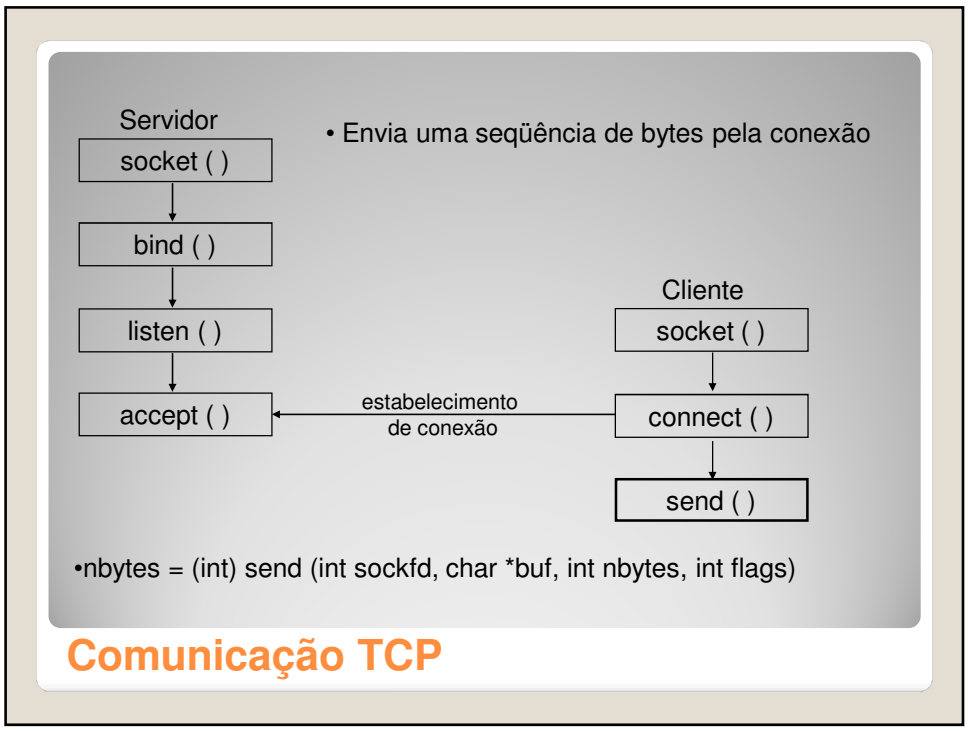
Comunicação TCP

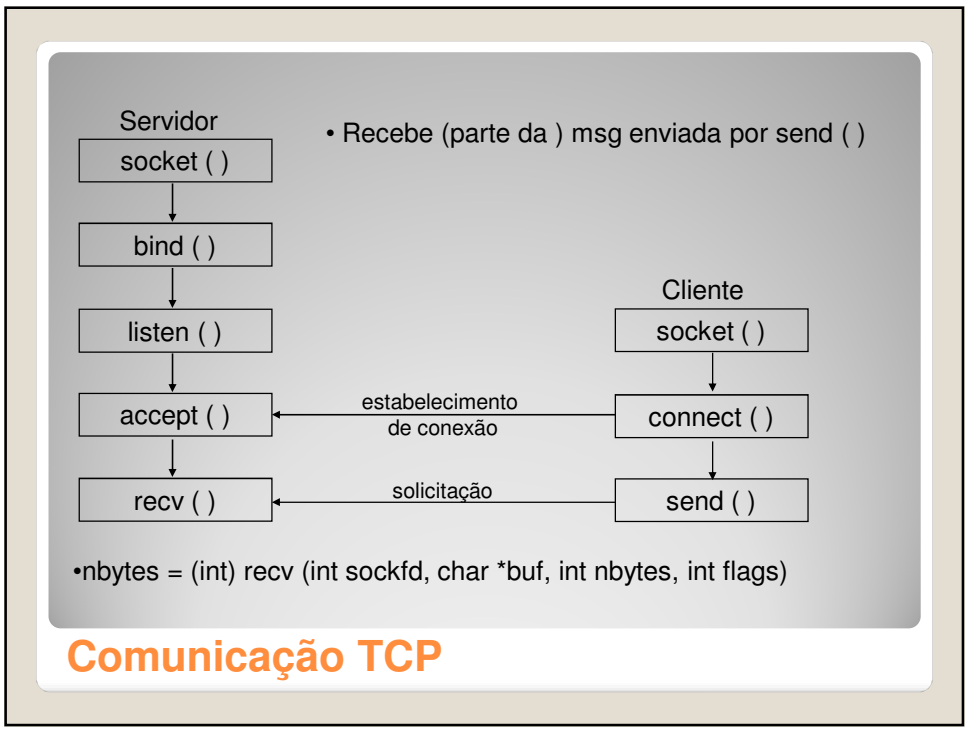

Servidor
socket ( )
bind ( )
listen ( )
Declara
- Que está pronto para receber conexões
- Até quantas devem ser enfileiradas
ret = (int) listen (int sockfd, int backlog)
Comunicação TCP

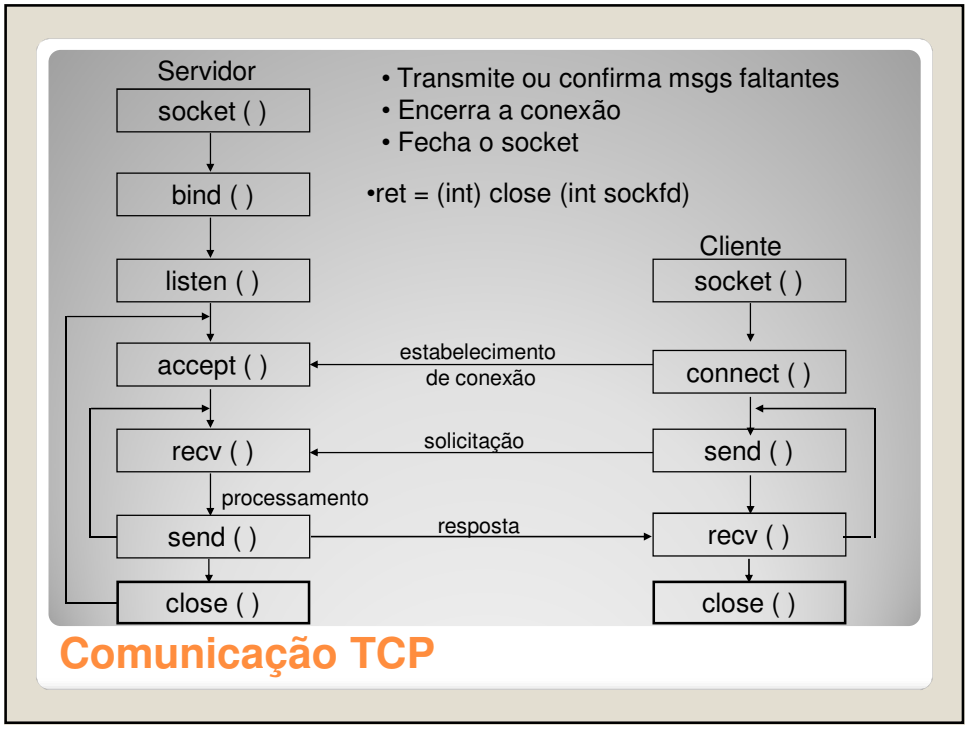

Servidor
socket ( )
bind ( )
listen ( )
accept ( )
• Bloqueia até que haja pedido de conexão
• Quando houver algum, aceita
•newsock = (int) accept (int sockfd, struct sockaddr *peer, int *addrlen)
Comunicação TCP


Servidor
socket ( )
bind ( )
listen ( )
accept ( )
• Cria um socket idêntico ao do servidor
(mesma família e tipo)
Cliente
socket ( )
sockfd = (int) socket (int family, int type, int protocol)
Comunicação TCP

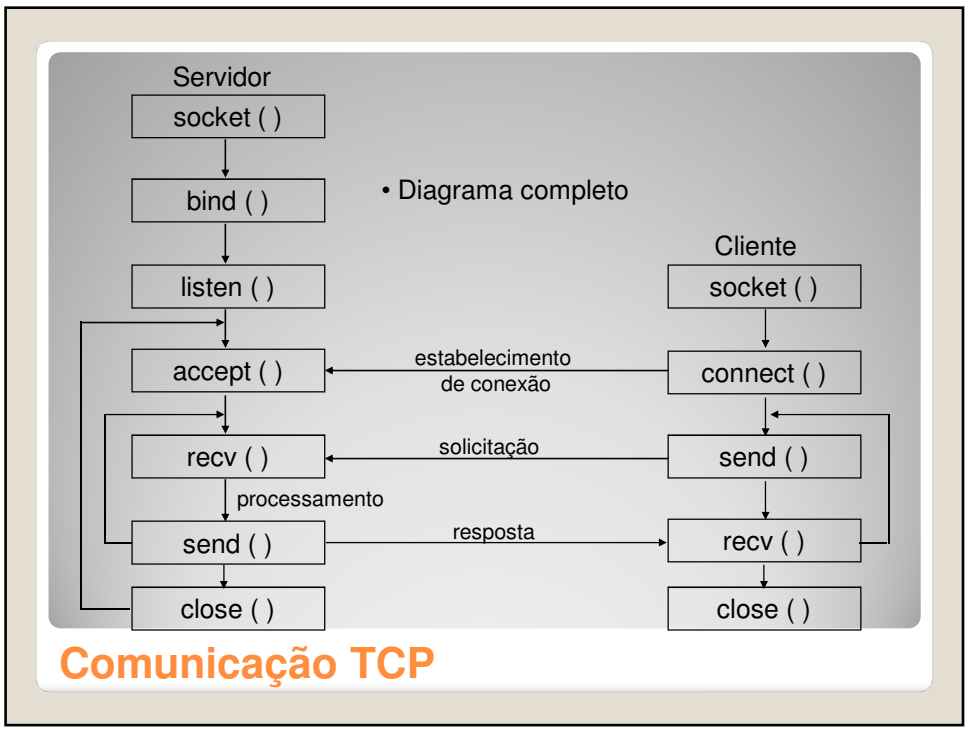

Servidor
socket ( )
bind ( )
listen ( )
accept ( )
• Pede uma conexão ao servidor
• Fica bloqueado ou retorna erro
• Se aceito, fecha a conexão
Cliente
socket ( )
connect ( )
estabelecimento
de conexão
•ret = (int) connect (int sockfd, struct sockaddr *servaddr, int addrlen)
Comunicação TCP


Servidor
socket ( )
bind ( )
listen ( )
accept ( )
• Envia uma seqüência de bytes pela conexão
Cliente
socket ( )
connect ( )
send ( )
estabelecimento
de conexão
•nbytes = (int) send (int sockfd, char *buf, int nbytes, int flags)
Comunicação TCP


Servidor
socket ( )
bind ( )
listen ( )
accept ( )
recv ( )
• Recebe (parte da) msg enviada por send ( )
Cliente
socket ( )
connect ( )
send ( )
estabelecimento
de conexão
solicitação
•nbytes = (int) recv (int sockfd, char *buf, int nbytes, int flags)
Comunicação TCP



Comunicação TCP



Comunicação TCP

Comunicação UDP

- Cliente e servidor criam seus sockets
- Família = Internet, tipo = datagrama

Comunicação UDP

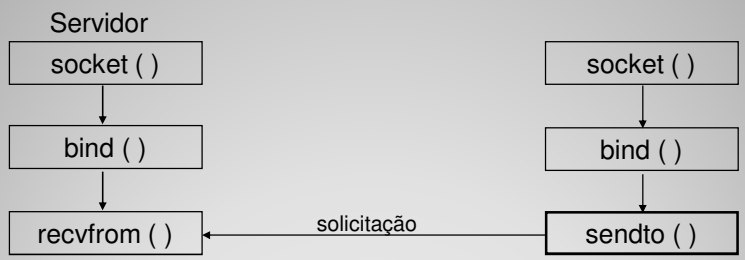
- Cliente e servidor definem endereços

• Recebe pacote enviado do endereço informado
• Se não houver nada, bloqueia

•nbytes = (int) recvfrom (int sockfd, char *buf, int nbytes, int flags, struct sockaddr *from, int *addrlen)

Comunicação UDP

• Envia pacote para o endereço informado

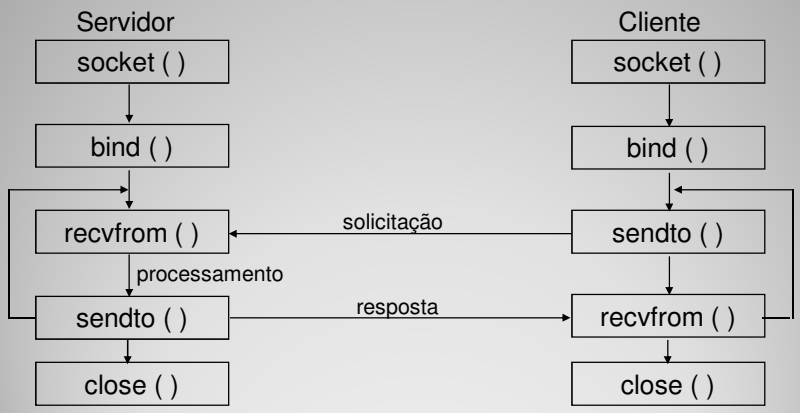


•nbytes = (int) recvfrom (int sockfd, char *buf, int nbytes, int flags, struct sockaddr *from, int *addrlen)

Comunicação UDP

Comunicação UDP

• Diagrama completo



Multiplexação

Problema:

•Servidor esperando conexão em vários sockets
• Como aceitar a 1ª que chegar?

Soluções

•Definir sockets não bloqueantes e fazer *polling*
=> busy wait
• Criar um filho para cada socket
• Usar I/O assíncrono (evento gera SIGIO)
=> programação não trivial
• Função select ( )

Multiplexação

ret = (int) select (int maxfd, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)

•maxfd indica o maior descritor a ser pesquisado
• readfds é um vetor de bits, cada bit correspondendo a um descritor onde se espera uma entrada
• writefds idem, se espera uma saída
• exceptfds idem, se espera uma exceção
• timeout define por quanto tempo o select espera

Função select ( )

```
#include <sys/time.h>
#include <sys/types.h>
...
fd_set readfds;
struct timeval wait;
...
for (;;) {
        wait.tv_sec = 1;        /* ajusta timeout para 1 seg */
        wait.tv_usec = 0;
        FD_ZERO (&readfds);  /* zera vetor de bits */
        FD_SET (sd1, &readfds);  /* liga bit correspondente ao socket 1 */
        FD_SET (sd2, &readfds);  /* liga bit correspondente ao socket 2 */
        nb = select (FD_SETSIZE, &readfds, (fd_set *) 0, (fd_set *) 0, &wait);
        if  (nb <= 0)  {
           /*  ocoreu erro ou expirou o timeout  */
        }
        if  (FD_ISSET (s1, &readfds))  {
                        /*  socket 1 está pronto para ser lido */
        }
        ...
```

Função select ( )

Exemplo Sockets

RPC – Remote Procedure Call