

Sistemas Operacionais II

Comunicação entre Processos Parte 2



Sumário

- Pipes
- FIFOs
- Sockets

Pipes

- Um *pipe* é um dispositivo que permite comunicação unidirecional
 - Dados escritos no “lado de escrita” do *pipe* são lidos no “lado de leitura” do mesmo
 - São dispositivos seriais
 - Os dados são lidos na mesma ordem em que foram escritos
 - Tipicamente são usados na comunicação entre duas *threads* em um único processo ou entre processos pai e filho.

Pipes

- Em *shells*, o símbolo `|` cria um *pipe*.
 - Exemplo:
 - `ls | less`
 - Faz com que o *shell* crie dois processos filhos, um para **ls** e outro para **less**
 - O *shell* também cria um *pipe* conectando a saída padrão do subprocesso **ls** com a entrada padrão do subprocesso **less**
 - » Os nomes de arquivos gerados por **ls** são enviados para **less** na mesma ordem que seriam enviados diretamente ao terminal

Pipes

- A capacidade de armazenamento de dados dos *pipes* é limitada.
 - Se o processo escritor for mais rápido que o processo leitor, e se o *pipe* não puder armazenar mais dados, o processo escritor bloqueia até que mais capacidade se torne disponível.
 - Se o processo leitor tentar ler mas não houver dados disponíveis, ele bloqueia até que dados se tornem disponíveis.
 - Portanto, o *pipe* automaticamente sincroniza os dois processos.

Criando Pipes

- Para criar um *pipe*, chame o comando **pipe**
 - Forneça um array de inteiros de tamanho 2
 - A chamada à **pipe** armazena o descritor de arquivo de leitura na posição 0 e o descritor de arquivo de escrita na posição 1.
 - Exemplo:

```
int pipe_fds[2];  
int read_fd;  
int write_fd;  
  
pipe(pipe_fds);  
read_fd = pipe_fds[0];  
write_fd = pipe_fds[1];
```

- Dados escritos no descritor de arquivos **write_fd** podem ser lidos a partir de **read_fd**

Comunicação entre Processo Pai e Processo Filho

- Uma chamada para **pipe** cria os descritores de arquivos, que são válidos somente naquele processo e seus filhos.
 - Os descritores de arquivos de um processo não podem ser passados a processos não relacionados.
 - Porém, quando o processo chama **fork**, os descritores de arquivos são copiados para o novo processo filho.
 - Portanto, *pipes* podem conectar apenas processos relacionados.

Comunicação entre Processo Pai e Processo Filho

- No exemplo a seguir, um **fork** cria um novo processo filho.
- O filho herda os descritores de arquivos do *pipe*.
- O pai escreve uma *string* no *pipe*, e o filho lê.
- O programa converte os descritores de arquivos em *streams* **FILE*** usando **fdopen**
 - Isto permite usar *streams* em vez de descritores de arquivos, permitindo o uso de funções de alto nível da biblioteca padrão de entrada/saída do C, como **printf** e **fgets**.


```

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Escreve COUNT cópias de MESSAGE para STREAM, pausando por
um segundo entre cada. */

void writer (const char* message, int count, FILE* stream)
{
    for (; count > 0; --count) {
        /* Escreve a mensagem para o fluxo, e envia imediatamente. */
        fprintf (stream, "%s\n", message);
        fflush (stream);
        /* Cochila um pouco. */
        sleep (1);
    }
}

/* Lê strings aleatórias do fluxo por tanto tempo quando possível. */

void reader (FILE* stream)
{
    char buffer[1024];
    /* Ler até encontrar o fim do fluxo. fgets lê até encontrar
uma nova linha ou o fim-do-arquivo. */
    while (!feof (stream)
        && !ferror (stream)
        && fgets (buffer, sizeof (buffer), stream) != NULL)
        fputs (buffer, stdout);
}

```

pipe.c

Usando um Pipe
para Comunicação
com um Processo
Filho

```

int main ()
{
    int fds[2];
    pid_t pid;

    /* Cria um pipe. Descritores de arquivos para as duas pontas do pipe
são colocadas em fds. */
    pipe (fds);
    /* Fork cria um processo filho. */
    pid = fork ();
    if (pid == (pid_t) 0) {
        FILE* stream;
        /* Este é o processo filho. Feche nossa cópia da ponta de escrita
do descritor de arquivo. */
        close (fds[1]);
        /* Converte o descritor de arquivo de leitura em um objeto FILE
e lê a partir dele. */
        stream = fdopen (fds[0], "r");
        reader (stream);
        close (fds[0]);
    }
    else {
        /* Este é o processo pai. */
        FILE* stream;
        /* Feche nossa cópia da ponta de leitura do descritor de arquivos. */
        close (fds[0]);
        /* Converta o descritor de arquivos de escrita em um objeto FILE
e escreve nele. */
        stream = fdopen (fds[1], "w");
        writer ("Olá, mundo.", 5, stream);
        close (fds[1]);
    }

    return 0;
}

```

Redirecionando Fluxos de Entrada, Saída e Erro Padrão

- Para redirecionar os fluxos de entrada, saída e/ou erro padrão podemos usar a chamada **dup2**
 - Exemplo:
 - Para redirecionar a entrada padrão de um processo para um descritor de arquivos **fd**, use esta linha:
 - **dup2(fd, STDIN_FILENO);**
 - » A constante simbólica **STDIN_FILENO** representa o descritor de arquivos para a entrada padrão, que tem o valor 0.
 - » A chamada fecha a entrada padrão e então a reabre como uma duplicata de **fd**.
 - Ambos compartilham a mesma posição no arquivo e o mesmo conjunto de flags de status de arquivo.
 - Ou seja, caracteres lidos de **fd** não serão relidos da entrada padrão.

Redirecionando Fluxos de Entrada, Saída e Erro Padrão

- O exemplo a seguir usa **dup2** para enviar a saída de um *pipe* para o comando **sort**.
- Após criar um *pipe*, o programa bifurca (*forks*)
- O processo pai imprime algumas *strings* para o *pipe*.
- O processo filho acopla o descritor de arquivo de leitura do *pipe* à sua entrada padrão usando **dup2**. Então ele executa o programa **sort**.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
    int fds[2];
    pid_t pid;

    /* Cria um pipe. Descritores de arquivos para as duas pontas doFile
       pipe são colocados em fds. */
    pipe (fds);
    /* Cria um processo filho. */
    pid = fork ();
    if (pid == (pid_t) 0) {
        /* Este é o processo filho. Feche nossa cópia da ponta de saída
           do descritor de arquivos. */
        close (fds[1]);
        /* Conecte a ponta de saída à entrada padrão do pipe. */
        dup2 (fds[0], STDIN_FILENO);
        /* Substitua o processo filho com o programa "sort". */
        execlp ("sort", "sort", 0);
    }

```

```

else {
    /* Este é o processo pai. */
    FILE* stream;
    /* Feche nossa cópia da ponta de leitura do descritor de
       arquivos. */
    close (fds[0]);
    /* Converte o descritor de arquivo de escrita em um objeto
       FILE e escreve nele. */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "Este é um teste.\n");
    fprintf (stream, "Olá, mundo.\n");
    fprintf (stream, "Meu cachorro tem pulgas.\n");
    fprintf (stream, "Este programa é ótimo.\n");
    fprintf (stream, "Um peixe, dois peixes.\n");
    fflush (stream);
    close (fds[1]);
    /* Espere o processo filho terminar. */
    waitpid (pid, NULL, 0);
}

return 0;
}

```

dup2.c

Redirecionando a
Saída de um Pipe
com *dup2*

FIFOs

- FIFO (*first-in, first-out*) é um *pipe* que tem um nome no sistema de arquivos.
- Qualquer processo pode abrir ou fechar o FIFO.
- Os processos em ambas as pontas do *pipe* não precisam ser relacionados um com o outro.
- FIFOs também são chamados *pipe com nome* (*named pipes*)

FIFOs

- Você pode criar um FIFO usando o comando **mkfifo**
 - Especifique o caminho para o FIFO na linha de comando
 - Exemplo:
 - Para criar um FIFO em **/tmp/fifo** use:
 - » **mkfifo /tmp/fifo**
 - » **ls -l /tmp/fifo**

```
fabricio@fabricio-virtual-machine:~/so2/aula9$ mkfifo /tmp/fifo
fabricio@fabricio-virtual-machine:~/so2/aula9$ ls -l /tmp/fifo
prw-rw-r-- 1 fabricio fabricio 0 Mai  3 19:12 /tmp/fifo
```

- O primeiro caracter da saída de ls é um **p**, indicando que o arquivo é na verdade um FIFO

FIFOs

- Abra dois terminais.
- No primeiro terminal digite:
 - **cat < /tmp/fifo**
- No segundo terminal digite:
 - **cat > /tmp/fifo**
- Escreva no segundo terminal e veja o que acontece no primeiro cada vez que você pressiona ENTER.

Criando um FIFO

- Para criar um FIFO programaticamente utilize a função **mkfifo**.
 - Argumentos:
 - Caminho no qual o FIFO deve ser criado.
 - Permissões do dono do *pipe*, seu grupo e todos os demais.
 - As permissões precisam incluir leitura e escrita.
 - Se o *pipe* não puder ser criado (por exemplo, se um nome de arquivo já existir) **mkfifo** retorna -1.
 - Inclua **<sys/types.h>** e **<sys/stat.h>** se chamar **mkfifo**.

Acessando um FIFO

- O acesso a um FIFO é como o acesso a um arquivo comum.
- Para se comunicar através do FIFO, um programa precisa abri-lo para escrita e outro deve abri-lo para leitura.
- Tanto funções de I/O de baixo nível (**open**, **write**, **read**, **close**, etc.) quanto as funções de I/O da biblioteca C (**fopen**, **fprintf**, **fscanf**, **fclose**, etc.) podem ser usadas

Acessando um FIFO

- Exemplos:
 - Escrevendo um buffer de dados para um FIFO usando rotinas de I/O de baixo nível:
- Lendo uma *string* do FIFO usando funções de I/O da biblioteca C:

```
int fd = open(fifo_path, O_WRONLY);  
write (fd, data, data_length);  
close (fd);
```

```
FILE* fifo = fopen (fifo_path, "r");  
fscanf (fifo, "%s", buffer);  
fclose(fifo);
```

Acessando um FIFO

- Um FIFO pode ter múltiplos leitores ou múltiplos escritores.
- Bytes de cada escritor são escritos atomicamente até um máximo de **PIPE_BUF** (4KB no Linux).
- Pedacos de escritores simultâneos podem ser intercalados.
- Regras similares se aplicam a múltiplas leituras.

Sockets

- Um *socket* é um dispositivo que permite comunicação bidirecional.
- Pode ser usado por processos rodando na mesma máquina ou em máquinas diferentes.
- Programas de Internet, como Telnet, rlogin, FTP, talk, a World Wide Web, etc. usam sockets.

Sockets

- Exemplo:
 - Você pode obter uma página HTML de um servidor Web usando o programa Telnet, pois ambos usam sockets para comunicação em rede.
 - Abra uma conexão com **www.rc.unesp.br**
 - **telnet www.rc.unesp.br 80**
 - Digite o comando **GET /** , pressione ENTER e observe o que acontece.

Obs: as vezes é preciso
pressionar ENTER duas
vezes

Conceitos de Sockets

- Quando você cria um socket, precisa especificar três parâmetros:
 - Estilo de comunicação
 - Espaço de nomes
 - Protocolo

Estilos de Comunicação

- Controla como o socket trata dados transmitidos e especifica a quantidade de parceiros de comunicação.
- Quando dados são enviados através de um socket, eles são empacotados em porções chamadas pacotes (*packets*).
- O estilo de comunicação determina como estes pacotes são manipulados e como eles são endereçados do remetente ao destinatário.

Estilos de Comunicação

- *Connection* (orientado a conexões)
 - Estilo que garante a entrega de todos os pacotes na ordem em que foram enviados.
 - Se pacotes são perdidos ou reordenados por problemas na rede, o receptor automaticamente requisita retransmissões ao remetente.
- *Datagram* (orientado a datagramas)
 - Estilo que não garante a entrega ou ordem de chegada.
 - Pacotes podem ser perdidos ou reordenados em trânsito devido a erros de rede ou outras condições.
 - Cada pacote precisa estar rotulado com seu destino e não há garantia de entrega.
 - O sistema garante apenas que fará o “melhor esforço”, portanto pacotes podem desaparecer ou chegar em uma ordem diferente da que foram enviados.

Espaço de Nomes

- Especifica como *endereços de sockets* são escritos.
- Um endereço de socket identifica uma ponta de uma conexão de socket.
- Por exemplo:
 - Endereços de socket no “espaço de nomes local” são nomes de arquivos comuns.
 - Endereços de socket no “espaço de nomes da Internet” são compostos por:
 - Um endereço de Internet (endereço IP) de um hospedeiro conectado a rede.
 - Número de porta/porto.
 - Distingue entre vários sockets no mesmo host.

Protocolos

- Especifica como os dados são transmitidos.
- Exemplos:
 - TCP/IP
 - O protocolo da Internet.
 - IPX/SPX
 - Foi bastante usado nas redes Novell Netware.
 - AppleTalk
 - Protocolo da Apple, presente no primeiro Macintosh em 1985, removido do Mac OS X em 2009.
 - Protocolo de comunicação local do UNIX

Chamadas de Sistema

- Sockets são mais flexíveis que as outras técnicas de comunicação já discutidas.
- Estas são as chamadas de sistema envolvendo sockets:
 - **socket** – Cria um socket.
 - **close** – Destrói um socket.
 - **connect** – Cria uma conexão entre dois sockets.
 - **bind** – Rotula um socket de servidor com um endereço.
 - **listen** – Configura um socket para aceitar conexões.
 - **accept** – Aceita uma conexão e cria um novo socket para ela.
- Sockets são representados por descritores de arquivos.

Criando e Destruindo Sockets

- As funções **socket** e **close** criam e destroem sockets, respectivamente.
- Quando você cria um socket, especifica espaço de nomes, estilo de comunicação e protocolo.
 - Exemplos nos próximos slides

Criando e Destruindo Sockets

- Espaço de nomes
 - Use constantes iniciando com **PF_** (de “protocol families”)
 - Exemplos:
 - **PF_LOCAL** ou **PF_UNIX** especificam o espaço de nomes local.
 - **PF_INET** ou **PF_INET6** especifica o espaço de nomes da Internet em IPv4 ou IPv6, respectivamente
- Estilo de comunicação
 - Use constantes iniciando com **SOCK_**
 - Exemplos:
 - **SOCK_STREAM** para o estilo *connection*
 - **SOCK_DGRAM** para o estilo *datagram*

Criando e Destruindo Sockets

- Protocolo
 - Especifica o mecanismo de baixo nível para transmitir e receber dados.
 - Cada protocolo é válido para uma combinação particular de estilo e espaço de nomes.
 - Há um protocolo ideal para cada combinação, especifique **0** para escolhê-lo.
- Se **socket** for bem sucedido, retornará um descritor de arquivos para o *socket*.
 - Você pode ler ou escrever para o *socket* usando **read**, **write**, etc., como faria com outros descritores de arquivos.
- Quando terminar de usar um socket, chame **close** para removê-lo.

Chamando **connect**

- Para criar uma conexão entre dois sockets, o cliente chama **connect**, especificando o endereço do socket do servidor ao qual ele quer se conectar.
- O cliente chama **connect** para iniciar uma conexão do socket local ao socket do servidor especificado no segundo argumento.
- O terceiro argumento é o tamanho, em bytes, da estrutura de endereço apontada pelo segundo argumento.
 - Os formatos de endereços de sockets diferem dependendo do espaço de nomes do socket.

Enviando informação

- Qualquer técnica usada para escrever em um descritor de arquivo pode ser usada para escrever em um socket.
- A função **send**, específica de descritores de socket, fornece uma alternativa ao **write**, com algumas opções adicionais.
 - Veja a página do manual para mais informações.

Servidores

- O ciclo de vida de um servidor consiste de:
 - Criar um socket orientado a conexões.
 - Acoplar um endereço a ele.
 - Chamar **listen** para habilitar conexões ao socket.
 - Chamar **accept** para aceitar conexões entrantes.
 - Fechar o socket.
- Dados não são lidos e escritos diretamente via socket de servidor.
 - Em vez disso, cada vez que um programa aceita uma nova conexão, o Linux cria um socket separado para usar na transferência de dados sobre aquela conexão.

Servidores

- Um endereço precisa ser acoplado ao socket de servidor usando **bind** para que um cliente possa encontrá-lo.
 - Argumentos:
 1. Descritor de arquivo do socket.
 2. Ponteiro para uma estrutura de endereço de socket.
 - O formato depende da família de endereços do socket.
 3. Tamanho da estrutura de endereço, em bytes.

Servidores

- Quando um endereço é acoplado ao socket de servidor, ele deve invocar **listen** para indicar que é um servidor.
 - Argumentos:
 1. Descritor de arquivo do socket.
 2. Quantidade de conexões que podem ser enfileiradas.
 - Se a fila estiver cheia, conexões adicionais serão rejeitadas.
 - Não limita o número de conexões que o servidor pode lidar, mas apenas o número de clientes tentando conectar e que ainda não foram aceitos.

Servidores

- Um servidor aceita uma requisição de conexão de um cliente invocando **accept**.
 - Argumentos:
 1. Descritor de arquivo do socket.
 2. Ponteiro para uma estrutura de endereços de socket, preenchida com o endereço do socket do cliente.
 3. Tamanho, em bytes, da estrutura de endereços do socket.

Servidores

- O servidor pode usar o endereço do cliente para determinar se realmente quer se comunicar com ele.
- A chamada à **accept** cria um novo socket para a comunicação com o cliente, e retorna o descritor de arquivo correspondente.
- O socket de servidor original continua a aceitar novas conexões de clientes.

Servidores

- Para ler dados de um socket sem removê-los da fila de entrada, use **recv**.
 - Tem os mesmos argumentos de **read**, e um argumento adicional de FLAGS.
 - Uma flag MSG_PEEK faz com que dados sejam lidos, mas não removidos da fila de entrada.

Sockets Locais

- Sockets conectando processos no mesmo computador podem usar o espaço de nomes local representado por **PF_LOCAL** e **PF_UNIX**.
- Estes são chamados *sockets locais* ou *sockets do domínio UNIX*.
- Seus endereços de socket, especificados por nomes de arquivos, são usados somente quando criando conexões.

Sockets Locais

- O nome do socket é especificado na estrutura **sockaddr_un**.
- Você precisa configurar o campo **sun_family** para **AF_LOCAL**, indicando que este é o espaço de nomes local.
- O campo **sun_path** especifica o nome de arquivo a ser usado, e precisa ter no máximo 108 bytes.
- O tamanho real da estrutura **sockaddr_un** deve ser calculado com a macro **SUN_LEN**.
- Qualquer nome de arquivo pode ser usado, mas o processo precisa ter permissões de escrita no diretório.
- Para conectar a um socket, o processo precisa ter permissão de escrita para o arquivo.
- Apesar de diferentes computadores poderem compartilhar o mesmo sistema de arquivo, apenas processos executando no mesmo computador podem ser comunicar com sockets de espaço de nomes local.

Sockets Locais

- O único protocolo possível para o espaço de nomes local é 0.
- Como reside no sistema de arquivos, um socket local é listado como um arquivo.
 - O primeiro caractere das entradas de socket na saída produzida por ls é a letra s.
- Chame **unlink** para remover um socket local quando terminar de usá-lo.

Exemplo usando Sockets com Espaço de Nomes Local

- O uso de sockets com espaço de nomes local é ilustrado com dois programas:
 - **socket-server.c**
 - Cria um socket no espaço de nomes local e escuta conexões a ele.
 - Quando recebe uma conexão, lê mensagens de texto da mesma e as imprime até que a conexão se encerre.
 - Se uma das mensagens for “quit”, o programa servidor remove o socket e finaliza
 - Toma o primeiro argumento como o caminho para o socket.
 - **client-socket.c**
 - Conecta ao socket de espaço de nomes local e envia uma mensagem.
 - O caminho para o socket e a mensagem são passados como argumento.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
```

/ Lê texto do socket e imprime. Continua até que o socket feche.
Retorna não-zero se o cliente enviou um "quit" ou zero caso contrário. */*

```
int server (int client_socket)
{
    while (1) {
        int length;
        char* text;

        /* Primeiro, lê o tamanho de uma mensagem de texto do socket.  
Se a leitura retorna zero, o cliente fechou a conexão. */
        if (read (client_socket, &length, sizeof (length)) == 0)
            return 0;
        /* Aloca o buffer para armazenar o texto. */
        text = (char*) malloc (length);
        /* Lê o texto, e o imprime. */
        read (client_socket, text, length);
        printf ("%s\n", text);
        /* Se o cliente enviou a mensagem "quit", terminamos. */
        if (!strcmp (text, "quit"))
        {
            free (text); /* Libera o buffer. */
            return 1;
        }
        else free (text); /* Libera o buffer. */
    }
}
```

Obs: no código original há um erro. O buffer está sendo liberado antes de testar se o usuário digitou "quit"

socket-server.c

25/04/2016

Servidor com
Socket de Espaço
de Nomes Local

```
int main (int argc, char* const argv[])
{
    const char* const socket_name = argv[1];
    int socket_fd;
    struct sockaddr_un name;
    int client_sent_quit_message;

    /* Cria o socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
    /* Indica que este é um servidor. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    bind (socket_fd, (struct sockaddr *) &name, SUN_LEN (&name));
    /* Escuta por conexões. */
    listen (socket_fd, 5);

    /* Repetidamente aceita conexões, e cria um novo socket para lidar  
com cada cliente. Continua até um cliente enviar a mensagem "quit". */
    do {
        struct sockaddr_un client_name;
        socklen_t client_name_len;
        int client_socket_fd;

        /* Aceita uma conexão. */
        client_name_len = sizeof(client_name);
        client_socket_fd = accept (socket_fd, (struct sockaddr *) &client_name, &client_name_len);
        /* Manipula a conexão. */
        client_sent_quit_message = server (client_socket_fd);
        /* Fecha nossa ponta da conexão. */
        close (client_socket_fd);
    }
    while (!client_sent_quit_message);

    /* Remove o arquivo de socket. */
    close (socket_fd);
    unlink (socket_name);

    return 0;
}
```

Obs: no código original falta o cast de &name

Obs: no código original falta o cast de &client_name e a inicialização de client_name_len

```

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Escreve TEXT para o socket dado pelo descritor de arquivo SOCKET_FD. */

void write_text (int socket_fd, const char* text)
{
    /* Escreve o número de bytes na string, incluindo o terminador nulo */
    int length = strlen (text) + 1;
    write (socket_fd, &length, sizeof (length));
    /* Escreve a string. */
    write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
    const char* const socket_name = argv[1];
    const char* const message = argv[2];
    int socket_fd;
    struct sockaddr_un name;

    /* Cria um socket. */
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
    /* Guarda o nome do servidor no endereço do socket. */
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    /* Conecta o socket. */
    connect (socket_fd, (struct sockaddr *) &name, SUN_LEN (&name));
    /* Escreve o texto da linha de comando para o socket. */
    write_text (socket_fd, message);
    close (socket_fd);
    return 0;
}

```

Obs: no código original
falta o cast de &name

socket-client.c

Cliente com Socket
de Espaço de
Nomes Local

Exemplo usando Sockets com Espaço de Nomes Local

- Observações:
 - Antes de enviar o texto, o cliente envia o tamanho do texto.
 - Ele faz isso enviando os bytes da variável **length**.
 - Da mesma forma, o servidor lê primeiro o tamanho do texto para uma variável inteiro.
 - Isto permite que ele aloque um buffer de tamanho apropriado para armazenar a mensagem antes de lê-la do socket.

Sockets do domínio da Internet

- Sockets do domínio do UNIX só podem ser usados na comunicação entre dois processos no mesmo computador.
- Sockets do domínio da Internet, por outro lado, podem ser usados para conectar processos em diferentes máquinas conectadas por rede.

Sockets do domínio da Internet

- Sockets que conectam processos através da Internet utilizam o espaço de nomes da Internet representados por **PF_INET** (IPv4) ou **PF_INET6** (IPv6).
- O protocolo mais comum é o TCP/IP.
 - O IP (*Internet Protocol*) é um protocolo de baixo nível que move pacotes através da Internet, dividindo-os e juntando-os novamente se necessário.
 - Ele garante apenas o “melhor esforço” de entrega, então pacotes podem sumir ou ser reordenados durante o transporte.
 - Cada computador participante tem um endereço IP único.
 - O TCP (*Transmission Control Protocol*), na camada acima da camada do IP, fornece transporte confiável com pacotes ordenados.

Sockets do domínio da Internet

- Endereços de sockets da Internet tem duas partes: máquina e número de porta.
 - Esta informação é armazenada na variável **sockaddr_in**.
 - Ajuste o campo **sin_family** para **AF_INET** para indicar que este é um endereço do espaço de nomes da Internet.
 - O campo **sin_addr** armazena o endereço da Internet da máquina desejada como um número IP inteiro de 32 bits.
 - O número de porta distingue entre os diferentes sockets de uma mesma máquina.
 - Diferentes máquinas armazenam valores de mais de um byte de formas diferentes, portanto use **htons** para converter o número de porta para *ordem de bytes de rede*.

Sockets do domínio da Internet

- Para converter nomes de hosts, endereços IPs na notação padrão (Ex.: 10.0.0.1) ou nomes DNS (Ex.: www.rc.unesp.br) em números IPs de 32 bits, use **gethostbyname**.
 - Retorna um ponteiro para uma estrutura **struct hostent**
 - O campo **h_addr** contém o número IP do host.

Sockets do domínio da Internet

- Exemplo de uso de sockets do domínio da Internet:
 - **socket-inet.c**
 - O programa obtém a *home page* do servidor Web especificado como argumento na linha de comando

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

/ Imprime o conteúdo da home page do servidor ao qual o socket.
está conectado. Retorna uma indicação de sucesso. */*

```
void get_home_page (int socket_fd)
```

```
{
    char buffer[10000];
    ssize_t number_characters_read;
```

Obs: as vezes é necessário
usar dois `\n` após o
comando `GET` /

```
    /* Envia o comando HTTP GET para pegar a home page. */
    sprintf (buffer, "GET /\n");
    write (socket_fd, buffer, strlen (buffer));
    /* Lê do socket. read pode não retronar todos os dados de uma vez,
    então continue tentando até acabar. */
    while (1) {
        number_characters_read = read (socket_fd, buffer, 10000);
        if (number_characters_read == 0)
            return;
        /* Escreva os dados para a saída padrão. */
        fwrite (buffer, sizeof (char), number_characters_read,
stdout);
    }
}
```

```
int main (int argc, char* const argv[])
```

```
{
    int socket_fd;
    struct sockaddr_in name;
    struct hostent* hostinfo;
```

/ Cria o socket. */*

```
socket_fd = socket (PF_INET, SOCK_STREAM, 0);
/* Armazena o nome do servidor no endereço do socket. */
name.sin_family = AF_INET;
```

/ Converte de strings para números. */*
hostinfo = gethostbyname (argv[1]);
if (hostinfo == NULL)

```
    return 1;
```

```
else
```

```
    name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
/* Servidores Web usam a porta 80. */
name.sin_port = htons (80);
```

Obs: no código original
falta o cast de &name

/ Conecta ao servidor web */*

```
if (connect (socket_fd, (struct sockaddr *) &name, sizeof
(struct sockaddr_in)) == -1) {
    perror ("connect");
    return 1;
}
```

/ Recupera a home page do servidor. */*

```
get_home_page (socket_fd);
```

```
return 0;
```

```
}
```

socket-inet.c

Lê de um Servidor Web

25/04/2016

Pares de Sockets

- Como vimos anteriormente, a função **pipe** cria dois descritores de arquivos para o início e fim de um *pipe*.
- *Pipes* são limitados porque os descritores de arquivos precisam ser usados por processos relacionados e porque a comunicação é unidirecional.
- A função **socketpair** cria dois descritores de arquivos para dois sockets conectados no mesmo computador. Estes descritores de arquivos permitem comunicação em dois sentidos entre processos relacionados.
 - Os primeiros três parâmetros são os mesmos da chamada **socket**, especificam domínio, estilo de conexão e protocolo.
 - O último parâmetro é um array de dois inteiros, que são preenchidos com os descritores de arquivos dos dois sockets, de maneira similar ao **pipe**.
 - O domínio precisa ser necessariamente **PF_LOCAL**.

Exercício

- Modifique o exemplo de Sockets com Espaço de Nomes Local para funcionar na Internet.
- Execute servidor e cliente em máquinas diferentes.
- O servidor recebe como parâmetro a porta que será ouvida.
- O cliente recebe como parâmetro o nome de host do servidor (ou endereço IP), a porta do servidor, e a mensagem a ser enviada.

Referências Bibliográficas

1. [NEMETH, Evi.; SNYDER, Garth; HEIN, Trent R.; *Manual Completo do Linux: Guia do Administrador*. São Paulo: Pearson Prentice Hall, 2007. Cap. 4](#)
2. [DEITEL, H. M.; DEITEL, P. J.; CHOFFNES, D. R.; *Sistemas Operacionais: terceira edição*. São Paulo: Pearson Prentice Hall, 2005. Cap. 20](#)
3. [MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; *Advanced Linux Programming*. New Riders Publishing: 2001. Cap. 5](#)
4. [TANENBAUM, Andrew S.; *Sistemas Operacionais Modernos*. 3ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 10](#)

