O Fantástico Mundo da Linguagem C

Alexandre Fiori

Prefácio

Após anos ministrando treinamentos de sistema operacional e redes, percebi que poucas pessoas entendem realmente como o mundo funciona. O mundo da tecnologia, dos sistemas, das linguagens.

Normalmente os especialistas estão focados em determinado nicho da tecnologia e não conseguem enxergar além do monitor. Boa parte dos administradores de sistema e redes são meros usuários de aplicações um pouco mais avançadas que editores de texto.

É raro quando um especialista em redes sabe o que realmente acontece quando os programas trocam pacotes, ou mesmo o que é uma conexão lógica através de um ambiente físico. As portas dos protocolos TCP e UDP são algo de outro mundo, que permitem entrar no computador - é o que muito acreditam.

No meio acadêmico então, isso é quase uma lástima. A coisa mais comum que existe é ouvir alguém dizer: tive C na faculdade mas não sei programar.

Tanto os professores quanto os alunos passam a ser usuários de linguagens de alto nível programando com *clicks* do mouse pois essas linguagens são fáceis e possuem uma série de recursos prontos.

Isso não pode acontecer. Em outros países há gente desenvolvendo tecnologia de ponta com linguagens de baixo nível, enquanto muitos de nós, Brasileiros, simplesmente compramos tecnologia ao invés de desenvolver.

Por mais que as empresas desenvolvam sistemas complexos e robustos em linguagens de alto nível, ainda terão que comprar a própria linguagem - como é o caso do *Delphi*, *.NET*, etc. Além disso, essas linguagens estão sujeitas a modificações periódicas, pois são meros produtos das empresas extrangeiras.

O conteúdo que será apresentado neste livro visa clearear o modo de pensar dos aprendizes de programação na Linguagem C.

A idéia principal é desmistificar assuntos que assombram muitos programadores, como os ponteiros, o acesso a recursos do sistema, a manipulação de arquivos, imagens e principalmente *sockets*.

Baseando-se no sistema operacional Linux e ferramentas providas pela GNU, seguimos um caminho tranquilo e repleto de exemplos práticos, com aplicações funcionais para o dia a dia.

Grande parte das funções utilizadas nos exemplos possuem documentação oficial distribuída com o sistema na forma de manuais, os quais não serão reproduzidos aqui.

É importante ter os manuais *(manpages)* do sistema em mãos, pois é com eles que você irá conviver depois deste livro.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Do básico mas não tão básico. Para ler este livro o leitor precisa ter um conhecimento prévio da linguagem C. Detalhes como o *hello world* não serão apresentados aqui.

Existem diversos livros que explicam todos esses menores detalhes sobre como criar os primeiros programas, como dar os primeiros passos. São simples, porém não muito claros, e existe uma grande barreira entre sair do básico e conhecer o que realmente é a linguagem C.

Este é o propósito do livro. Mostrar ao leitor a grande quantidade de funções disponíveis para criar aplicações sólidas e profissionais, e tudo que você precisa conhecer é o básico: como criar um programa, como compilar, o que é o pré-processador, o que são e como funcionam os *loops*, vetores, etc.

1.1 O que é a Linguagem C?

No menor dos resumos, é uma maneira de manipular memória. A manipulação dá-se pelo uso de variáveis e funções, sendo que ambas possuem tamanho fixo determinado pelo tipo. Portanto, o tipo da variável ou da função é que determina o tamanho da área em memória.

Como são variáveis se são fixos? O conteúdo é variável, mas o tamanho é fixo.

Por essas e outras, a linguagem C parece complicada quando na verdade não é. A grande barreira está no sistema operacional, e se você entende como o ele funciona tudo se torna mais fácil.

Todo o conteúdo que será apresentado baseia-se no sistema operacional Linux[1], com um conjunto de ferramentas providas pela GNU[2].

1.2 Memória

Quando falamos de memória em programação, estamos falando de *address space*. Os *address spaces* são áreas de memória RAM providas pelo sistema operacional para que os programas funcionem.

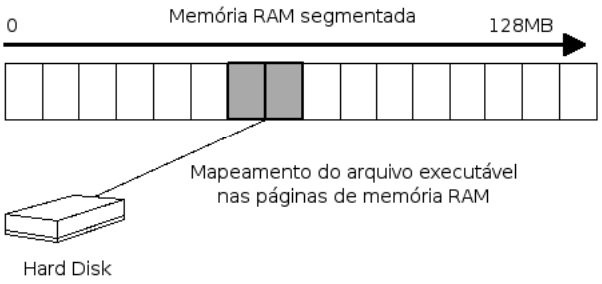
Quando o computador é iniciado o sistema operacional mapeia toda a memória RAM e cria segmentos de tamanho fixo para que seja mais fácil manipular seu conteúdo. Esses segmentos são conhecidos como páginas de memória. Um procedimento semelhante é utilizado nos discos, pois quando são formatados o sistema de arquivos é baseado em blocos.

Os programas executáveis, por sua vez, são arquivos binários provenientes de um código fonte que foi compilado para determinada plataforma e possui um formato, o formato executável do sistema operacional.

[1] http://www.linux.org
[2] http://www.gnu.org

Quando um programa é executado, o sistema operacional pega o código binário que está no disco e mapeia na memória, utilizando uma ou mais páginas para alocar seu conteúdo - essas páginas de memória podem ter as permissões de leitura, gravação ou execução.

Figura 1.1: Mapeamento de arquivos na memória



Na figura 1.1 a área marcada em cinza representa a as páginas de memória alocadas para o arquivo executável no sistema operacional. Ali é o *address space* do programa.

Todas as variáveis e funções serão criadas naquela área, exceto a memória alocada dinamicamente que será vista posteriormente.

Durante o desenvolvimento de aplicações o programador pode cometer erros e acessar áreas de memória fora de seu *address space*. Quando isso ocorre o sistema operacional finaliza o programa emitindo um sinal chamado SIGSEGV[3], ou Falha de Segmentação *(Segmentation Fault)*.

O programa a seguir será utilizado para detalhar o *address space* e as áreas internas dentro dele.

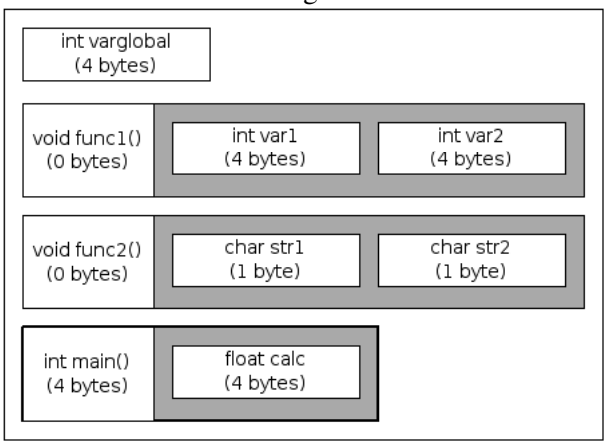
addrspace.c

```
/*
 * addrspace.c: demonstração de address space
 *
 * Para compilar:
 *  cc -Wall addrspace.c -o addrspace
 *
 * Alexandre Fiori
 */

/* variável global */
int varglobal;

void func1(void)
{
    /* variáveis de func1() */
    int var1, var2;
}

void func2(void)
{
    /* variáveis de func2() */
    char str1, str2;
}

int main()
{
    /* variável de main() */
    floar calc;

    return 0;
}
```

[3]SIGNAL(7) - Linux Programmer's Manual

Ali, algumas variáveis foram criadas e estão em espaços de memória segmentados dentro do *address space*. As variáveis globais estão disponíveis em todo o *address space*, enquanto as variáveis declaradas dentro das funções só estão disponíveis naquele escopo, veja:

Figura 1.2: Dentro do *address space* do programa



A figura 1.2 mostra claramente que a variável *varglobal* está no mesmo nível das funções, portanto é global. As variáveis *var1* e *var2* só existem no escopo de *func1()* e as variáveis *str1* e *str2* só existem no escopo de *func2()*. Na função *main()* há apenas uma variável, *calc,* que só existe naquele escopo.

A variável global pode ser acessada por qualquer uma das funções mencionadas, porém as demais só podem ser acessadas dentro de seu próprio escopo.

Note que as variáveis possuem tamanhos diferentes, relativas ao tipo. Como já foi mencionado, o tipo da variável *int*, *float* ou *char* serve apenas para delimitar o tamanho que ela ocupa na memória.

As funções também têm tipo, utilizado para o comando *return* que coloca naquela área de memória o valor a ser retornado. O tipo *void* é considerado nulo, utilizado para funções que não retornam valores.

1.3 Tipos de variáveis

A linguagem C provê alguns tipos de variáveis e modificadores, que serão apresentados aqui.

Embora muitos acreditem que o tipo *char* só deve ser utilizado para caracteres e o tipo *int* para números, isso não é verdade.

Como eles são apenas delimitadores de tamanho, foram convencionados a ter o uso por determinado conteúdo, mas podemos fazer contas utilizando *char* e colocar caracteres em variáveis do tipo *int* sem nenhum problema.

Já os tipos *float* e *double* são utilizados para armazenar números fracionários - chamados de ponto flutuante.

1.3.1 Tipos básicos

Seguem os tipos básicos:

Tabela 1.1: Tipos básicos de variáveis e funções

Tipo	**Tamanho em *bits***
int	32
char	8
float	32
double	64

O tipo *int* pode variar de acordo com a plataforma. No DOS, por exemplo, o *int* tem apenas 16 *bits* - pois o sistema operacional é 16 *bits*.

Já o Linux e outros sistemas 32 *bits* operam com o *int* de 32 bits.

1.3.2 Modificadores de tamanho

Os modificadores de tamanho permitem alterar a capacidade de armazenamento de apenas dois tipos: *int* e *double*, veja:

Tabela 1.2: Modificadores de tamanho

Tipo	**Tamanho em *bits***
short int	16
long int	32
long double	96

Nos sistemas 32 *bits* o *int* comum é sempre um *long int*, de maneira implícita. Caso o programador deseje manter isso explícito pode utilizar a palavra *long*.

Os outros tipos básicos não podem ser alterados.

1.3.3 Modificadores de sinal

As variáveis podem ser preenchidas com valores numéricos positivos ou negativos. Os modificadores de sinal são apenas dois: *signed* e *unsigned*.

Variáveis do tipo *unsigned int* nunca terão valores numéricos negativos, assim como *unsigned char*.

Por padrão todas elas são do tipo *signed* implicitamente.

A modificação do tipo para *signed* ou *unsigned* reflete diretamente no conteúdo a ser armazenado na variável e faz com que os *bits* lá presentes sejam tratados de forma diferente, veja:

Tabela 1.3: Modificadores de sinal

Tipo	***Bit7***	***Bit6***	***Bit5***	***Bit4***	***Bit3***	***Bit2***	***Bit1***	***Bit0***
unsigned	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
signed	$-2^7 = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

A tabela acima exibe a maneira como são representados os números positivos e negativos de acordo com o modificador em uma variável do tipo *char*, de 8 *bits*.

Para números binários como 00001010, seu valor decimal será 10 independente do modificador existir ou não. Mas para valores binários que utilizam o primeiro *bit* mais à esquerda, o valor decimal será completamente diferente.

Quando a variável é do tipo *signed* o *bit* mais à esquerda é o *bit* do sinal. Se ele for 1, o número sempre será negativo, veja:

Tabela 1.4: Diferenças entre *unsigned* e *signed*

binário	***unsigned***	***signed***
00100001	33	33
01010001	81	81
10001000	136	-120
10101010	170	-86

Quando convertemos o binário para decimal, somamos o valor dos *bits*. No caso do número 10001000 *unsigned* temos apenas o *bit7* e o *bit3* ligados, portanto 128+8 resulta em 136.

Porém, quando somamos o *bit7* e o *bit3* do número 10001000 *signed*, fazemos -128+8, resultando em -120.

1.3.4 Variáveis *const* e *static*

O tipo *const* faz com que a variável seja apenas para leitura, portanto é necessário inicializá-la durante a programação, veja:

```
const float pi = 3.141516;
```

Durante a execução do programa esta variável nunca poderá ser alterada. Ela é uma constante e só pode ser lida.

O tipo *static* é um pouco diferente. Quando o compilador gera o código binário ele deixa pré-definido que ao entrar no escopo de uma função as variáveis de lá devem ser criadas. Portanto, quando uma função é chamada sua área de memória é criada com as variáveis e quando ela encerra aquela área deixa de existir. Isso faz com que o *address space* tenha tamanho variável.

As variáveis do tipo *static* sempre estarão presentes, mesmo que declaradas dentro uma função que não está em uso. Isso permite armazenar valores dentro de funções para uso em chamadas subsequentes, veja:

```
int count(void)
{
    static int current = 0;
    return current++;
}
```

Ao iniciar o programa, a variável *current* terá o valor *0*. Cada vez que a função *count()* for chamada, este valor irá incrementar um. Se a variável não fosse do tipo *static* ela seria criada no memento da chamada de *func()*, que sempre retornaria *0*.

As variáveis *static* fazem com que o programa consuma mais memória pelo fato de sempre existirem lá.

1.3.5 Os enumeradores, *enum*

Os enumeradores são utilizados para criar variáveis com nomes diferentes e valores numéricos sequenciais. Essas variáveis são sempre do tipo *const int*.

Exemplo:

```
enum {
    jan = 1,
    fev,
    mar,
    abr,
    ...
};
```

A partir desta declaração, as variáveis *jan*, *fev*, *mar* e assim por diante serão constantes com os valores *1*, *2*, *3* e assim consecutivamente.

Caso o valor inicial não seja definido, será sempre *0*.

1.3.6 As estruturas, *struct*

As estruturas são utilizadas para criar novos tipos de dados à partir dos tipos básicos. São de extrema utilidade na programação.

Exemplo:

```
struct pessoa {
    int idade;
    double telefone;
};
```

Agora o programa conta com um novo tipo de dados, o tipo *struct pessoa*. É possível criar variáveis com este tipo, porém há uma pequena regra para acessar seu conteúdo, veja:

```
struct pessoa p1, p2;

p1.idade = 33;
p1.telefone = 50723340;

p2.idade = 21;
p2.telefone = 82821794;
```

As variáveis *p1* e *p2* são do tipo *struct pessoa*. Para acessar os membros dessas variáveis é necessário utilizar o . e depois o nome do membro.

Na memória, essas variáveis são organizadas exatamente como são definidas, sem nenhum *bit* a mais nem menos. No caso do tipo *struct pessoa*, é uma variável com 32+64 *bits*, resultando em 96 *bits* ou simplesmente 12 *bytes*.

1.3.7 As uniões, *union*

Como o próprio nome já diz, são utilizadas para unir uma mesma área de memória que pode ser acessada de diferentes maneiras.

Segue a declaração de uma *union* de exemplo:

```
union valores {
    unsigned int vi;
    float vf;
};

union valores temp;

temp.vi = 10;
```

Assim como nas estruturas, as uniões premitem a criação de tipos. O tipo *union valores* é utilizado para criar uma variável com 32 *bits* que pode ser acessada como *int* ou *float*. Quando a variável *temp* foi criada, ao invés de ter o tamanho de *unsigned int* + *float*, passou a ter o tamanho do maior. Como ambos são do mesmo tamanho, o *temp* passou a ter 32 *bits*.

A vantagem é que podemos atribuir qualquer número à *temp.vi* e depois trabalhar com ele em *temp.vf* ou vice-versa.

1.3.8 O *typedef*

O *typedef* permite criar um nome alternativo para um tipo de variável. É muito utilizado com *enum*, *struct* e *union*.

Exemplo:

```
typedef enum {
     jan = 1,
     fev,
     mar,
} mes;
```

Nesse caso o tipo *mes* é válido e só pode receber os valores definidos em *enum*. Podemos criar variáveis assim: *mes atual = fev;*

Onde *mes* é o tipo e *atual* é a variável, sendo *fev* uma constante representando o número 2.

Outro exemplo:

```
typedef struct {
     int idade;
     double telefone;
} pessoa;
```

Agora o tipo *pessoa* é válido para criar qualquer variável. Podemos escrever no código *pessoa p1, p2;* e essas variáveis serão do tipo *pessoa*.

O mesmo acontece com *union* quando utilizado com *typedef*.

1.4 Utilizando a área de memória

Este programa mostra que é extremamente possível utilizar a área de memória de uma variável do tipo *char* para realizar cálculo. Porém, este tipo tem apenas 1 *byte* o que significa que seu valor decimal *unsigned* pode ser no máximo 255.

somachar.c

```
/*
 * somachar.c: soma utilizando tipo char
 *
 * Para compilar:
 *  cc -Wall somachar.c -o somachar
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main()
{
     char x = 10, y = 20, z = x + y;

     fprintf(stdout, "%d\n", (int) z);

     return 0;
}
```

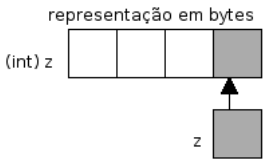
Executando:

```
$ ./somachar
30
```

Viu? Não tem segredo. O *char* é uma área de memória exatamente como a de um *int*, porém com menor quantidade de *bits*.

O *type casting* utilizado ali, *(int) z*, faz com que a variável *z* seja interpretada como um *int*. A área de memória do tamanho do *int* é criada porém apenas 1/4 é preenchido com o conteúdo de *z*.

Figura 1.3: Utilizando Type Casting



Qualquer operação matemática como soma, subtração, multiplicação e divisão pode ser realizada com eles. Também não há limites com os operadores lógicos.

1.5 A função *sizeof()*

Utilizada pra informar o tamanho em *bytes* de um tipo ou variável. Podemos utilizar o *sizeof()* de diversas maneiras e em alguns casos devemos tomar cuidado, especialmente com os vetores.

Exemplo:

```
char str[10];
int  val[10];
```

Ao executar *sizeof(str)* teremos o tamanho do tipo *char* multiplicado pela quantidade de membros na variável, resultando em 10. Mas quando fazemos *sizeof(val)* temos o tamanho do *int* multiplicado pela quantidade de membros, resultando em 40.

O *sizeof()* também pode ser utilizado em tipos criados pelo programador, como no caso da Sessão 1.3.8 onde os tipos *mes* e *pessoa* foram criados. É perfeitamente possível executar *sizeof(pessoa)*, ou *sizeof(p1)*, ou ainda *sizeof(struct pessoa)*.

Para imprimir o tamanho das variáveis em *bits*, basta multiplicar o resultado de *sizeof()* por oito. (dã!)

Capítulo 2

Asteriscos na Memória

Pegue seu papel e caneta. A hora dos ponteiros chegou.

Conhecendo um pouco do sistema operacional tudo se torna mais fácil. Quando você já sabe o que é a "memória" do programa, entende os ponteiros como bebe um copo d'água.

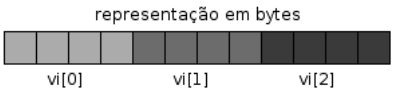
Esta é outra barreira para os estudantes. Esse mito sobre os ponteiros é apenas uma balela, e sem eles não é possível programar e nem mesmo entender a linguagem C.

Aqui eles serão apresentados da maneira mais clara possível. É fato que muitos estudantes abandonam a linguagem C por não conseguir desenvolver aplicações úteis e isso simplesmente porque não entendem os ponteiros.

2.1 Enxergando os vetores

Para o programa os vetores são áreas de memória lineares e o tamanho depende da quantidade de membros. Uma definição do tipo *int vi[3]* tem o tamanho de *sizeof(int) * 3*.

Figura 2.1: Representação gráfica de vetor



Então, quando um programa acessa o membro 0 desta variável está acessando apenas um pedaço da área de memória. Quando o índice aumenta, vira 2, ele está acessando outro pedaço da área de memória da mesma variável. Assim são os vetores.

Não existe outro delimitador de tamanho senão o próprio tipo da variável.

Os outros tipos de dados são exatamente iguais. O tipo *char*, especificamente, possui apenas 1 *byte* portanto seu incremento acontece de um em um.

O programa a seguir será utilizado para imprimir o endereço de cada membro do vetor, utilizando o tipo *char*.

charaddr.c

```
/*
 * charaddr.c: imprime o endereço e cada membro do vetor
 *
 * Para compilar:
 *  cc -Wall charaddr.c -o charaddr
 *
 * Alexandre Fiori
```

```
 */

#include <stdio.h>
#include <string.h>

int main()
{
    int i;
    char str[] = "sou um vetor";

    for(i = 0; i < strlen(str); i++)
        fprintf(stdout, "%p -> %c\n", &str[i], str[i]);

    return 0;
}
```

Executando:

```
$ ./charaddr
0xbfe1822f -> s
0xbfe18230 -> o
0xbfe18231 -> u
0xbfe18232 ->
0xbfe18233 -> u
0xbfe18234 -> m
0xbfe18235 ->
0xbfe18236 -> v
0xbfe18237 -> e
0xbfe18238 -> t
0xbfe18239 -> o
0xbfe1823a -> r
```

Se prestar atenção, notará que a diferença do endereço entre os caracteres incrementa de um em um - o tamanho de um *char*.

Agora vamos ao exemplo utilizando o tipo *int*.

intaddr.c

```
/*
 * intaddr.c: imprime o endereço e cada membro do vetor de caracteres
 *            utilizando o tipo int
 *
 * Para compilar:
 *  cc -Wall intaddr.c -o intaddr
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main()
{
    int i;
    int str[] =
        { 's', 'o', 'u', ' ', 'u', 'm', ' ', 'v', 'e', 't', 'o', 'r' };

    for(i = 0; i < sizeof(str)/sizeof(int); i++)
        fprintf(stdout, "%p -> %c\n", &str[i], (char) str[i]);

    return 0;
}
```

Executando:

```
$ ./intaddr
0xbfa1874c -> s
0xbfa18750 -> o
```

```
0xbfa18754 -> u
0xbfa18758 ->
0xbfa1875c -> u
0xbfa18760 -> m
0xbfa18764 ->
0xbfa18768 -> v
0xbfa1876c -> e
0xbfa18770 -> t
0xbfa18774 -> o
0xbfa18778 -> r
```

Aqui, o intervalo entre cada membro do vetor é do tamanho de um *int*, ou seja, 4.

O único cuidado com este programa é com relação à função *sizeof()*. Ela sempre retorna o tamanho de uma variável ou tipo em *bytes*. Se chamarmos *sizeof(int)*, o resultado é 4. Porém, se criarmos um vetor como *int vi[3]* e chamarmos *sizeof(vi)* o resultado será 12. Como neste programa utilizamos o *int* para um vetor de caracteres, estamos disperdiçando memória pois a cada membro de 4 *bytes* utilizamos apenas 1.

O *type casting* utilizado em *(char) str[i]* faz com que apenas o último *byte* dos 4 de cada membro seja utilizada, para que *fprintf()* imprima o caracter em *%c*.

A diferença básica os dois programas é que as variáveis do tipo *char* foram preparadas para receber caracteres, e os vetores do tipo *char* sempre terminam em *'\0'*.

Isso torna possível fazer *loops* que andam por cada elemento do vetor terminando ao encontrar um *'\0'*.

Com vetores do tipo *int* ou qualquer outro, isso também é possível, mas deve ser feito manualmente, enquanto que no tipo *char* esse procedimento é o padrão gerado por todos os compiladores em variáveis constantes.

2.2 Enxergando os ponteiros

Toda variável tem um endereço no *address space*, e este endereço é do tamanho de um *int* - são endereços de 32 *bits*.

Portanto, todo ponteiro tem o tamanho de um *int*. Mas um ponteiro é um vetor de apenas um membro, e o índice 0 deste vetor tem o tamanho do tipo.

Uma definição como *double *p* signfica que *p* tem o tamanho de um *int* e *p[0]* tem um tamanho de um *double*.

Como os ponteiros são vetores, não podemos apontá-los para variáveis que não são vetores, a menos que utilizemos o endereço da variável.

Exemplo:

ptrprint.c

```
/*
 * ptrprint.c: imprime endereço de ponteiro e conteúdo de variável através
 *             do ponteiro
 *
 * Para compilar:
 *  cc -Wall ptrprint.c -o ptrprint
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main()
{
    char *ptr;
```

```
    char  tmp = 'A';

    fprintf(stdout, "ptr(%p) tem %d bytes.\n", &ptr, sizeof(ptr));
    fprintf(stdout, "tmp(%p) tem %d bytes.\n", &tmp, sizeof(tmp));

    ptr = &tmp;

    fprintf(stdout, "agora ptr(%p) aponta para tmp: %c\n", ptr, ptr[0]);
    fprintf(stdout, "mas o endereço de ptr(%p) é o mesmo!\n", &ptr);

    return 0;
}
```

Executando:

```
$ ./ptrprint
ptr(0xbfd199b4) tem 4 bytes.
tmp(0xbfd199b3) tem 1 bytes.
agora ptr(0xbfd199b3) aponta para tmp: A
mas o endereço de ptr(0xbfd199b4) é o mesmo!
```

Quando a variável *tmp* foi criada, passou a ter um endereço na memória com o tamanho de 1 *byte*. O conteúdo atribuído a este *byte* foi a letra *'A'*.

Ao executar *ptr = &tmp*, imediatamente *ptr* passou a apontar para a área de memória de *tmp* e portanto *ptr[0]* foi utilizado para acessar essa área.

É importante saber que para acessar o membro 0 de um vetor podemos utilizar o asterisco, veja:

```
fprintf(stdout, "agora ptr(%p) aponta para tmp: %c\n", ptr, *ptr);
```

Então, concluímos que *ptr[0]* e **ptr* são a mesma coisa.

2.2.1 Aritmética dos ponteiros

Sabendo o que é um ponteiro fica muito fácil de compreender programas como o que será apresentado aqui.

Uma das diferenças entre as variáveis comuns e os ponteiros é que podemos utilizar operadores aritméticos com eles, como incremento, decremento, soma e subtração.

Exemplo:

ptraddr.c

```
/*
 * ptraddr.c: imprime o endereço e cada membro do vetor utilizando ponteiro
 *
 * Para compilar:
 *  cc -Wall ptraddr.c -o ptraddr
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main()
{
    char *p, str[] = "sou um vetor";

    p = str;

    while(*p != '\0') {
        fprintf(stdout, "%p -> %c\n", p, *p);
        p++;
    }

    return 0;
}
```

Executando:

```
$ ./ptraddr
0xbf8ba80f -> s
0xbf8ba810 -> o
0xbf8ba811 -> u
0xbf8ba812 ->
0xbf8ba813 -> u
0xbf8ba814 -> m
0xbf8ba815 ->
0xbf8ba816 -> v
0xbf8ba817 -> e
0xbf8ba818 -> t
0xbf8ba819 -> o
0xbf8ba81a -> r
```

O resultado deste programa utilizando ponteiro para fazer o *loop* é exatamente o mesmo do programa *charaddr.c* na Sessão 2.1, utilizando vetor com índice.

Analisando o código você verá que *ptr = str* não precisa de & pois ambos são vetores, portanto são do mesmo tipo. Quando um ponteiro aponta para uma variável comum, precisamos utilizar o endereço da variável. Quando ele aponta para outro vetor, não precisamos.

Outro detalhe é que **p* sempre representa *p[0]*, porém *p++* faz com que o ponteiro avance um *byte*, então **p* ou *p[0]* cada hora aponta para um dos membros do vetor.

Se o ponteiro fosse do tipo *int*, ele iria avançar *sizeof(int)* a cada incremento.

Em todos os tipos pode-se fazer `p++`, `p--`, `p+=x`, `p-=y`.

Veja:

ptrset.c

```
/*
 * ptrset.c: ajusta posição do ponteiro utilizando aritmética de ponteiros
 *
 * Para compilar:
 *  cc -Wall ptrset.c -o ptrset
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main()
{
    char *p, str[] = "sou um vetor";

    p = str;
    p += 5;

    fprintf(stdout, "%c, %c\n", *p, p[2]);

    return 0;
}
```

Executando:

```
$ ./ptrset
m, v
```

Quando *p = str* o ponteiro aponta para o primeiro *byte* do vetor *str.* Ao incrementar 5, em *p += 5*, este ponteiro avança 5 *bytes*.

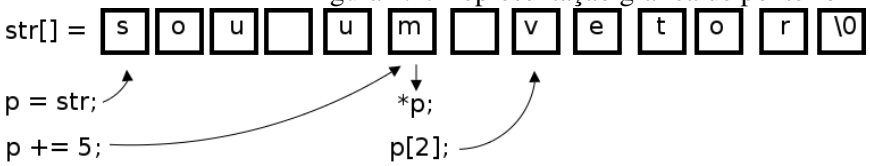
Se cada caracter é um *byte* e você está no primeiro *byte* de *"sou um vetor",* ao avançar 5 em qual caracter você irá cair?

Por isso **p*, ou *p[0]* imprimiu o caracter 'm'.

A partir dali, *p[2]* referencia dois *bytes* à frente, mas não avança. Por isso o caracter 'v' é impresso.

Veja:

Figura 2.2: Representação gráfica de ponteiro



2.3 Particularidades

Saber o que é o *address space* é fundamental para entender os ponteiros. Ter em mente que os endereços de memória são as variáveis do seu programa, apenas no *address space* do seu processo, é trivial.

Lembrar que um asterisco corresponde a um vetor (* == []) irá ajudar na hora de resolver problemas com os ponteiros.

Agora veremos algumas particularidades dos ponteiros.

2.3.1 Ponteiro para estrutura

As estruturas de dados também são áreas de memória lineares para a aplicação, tendo como único delimitador o tipo das variáveis.

Ao criar um ponteiro para estrutura, o método de acessar os membros é diferente.

ptrstruct1.c

```
/*
 * ptrstruct1.c: ponteiro para estrutura
 *
 * Para compilar:
 *  cc -Wall ptrstruct1.c -o ptrstruct1
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>

int main()
{
    struct pessoa {
        int idade;
        char nome[20];
    };

    struct pessoa bozo;
    struct pessoa *p;

    memset(&bozo, 0, sizeof(bozo));

    bozo.idade = 33;
    snprintf(bozo.nome, sizeof(bozo.nome), "Palhaço Bozo");

    p = &bozo;

    fprintf(stdout, "Nome: %s, Idade: %d\n", p->nome, p->idade);
```

```
    return 0;
}
```

Executando:

```
$ ./ptrstruct1
Nome: Palhaço Bozo, Idade: 33
```

Para apontar, *p = &bozo*, utilizamos & pelo fato de *bozo* não ser um vetor. Então apontamos para seu endereço.

Os ponteiros para estruturas utilizam '->' para acessar os membros e também é possível incrementá-las assim como fazemos com *char*, veja:

ptrstruct2.c

```
/*
 * ptrstruct2.c: ponteiro para vetor de estruturas
 *
 * Para compilar:
 *  cc -Wall ptrstruct2.c -o ptrstruct2
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>

int main()
{
    struct pessoa {
        int idade;
        char nome[20];
    };

    struct pessoa bozo[2];
    struct pessoa *p;

    memset(&bozo, 0, sizeof(bozo));

    bozo[0].idade = 33;
    snprintf(bozo[0].nome, sizeof(bozo[0].nome), "Palhaço Bozo (#1)");

    bozo[1].idade = 28;
    snprintf(bozo[1].nome, sizeof(bozo[1].nome), "Palhaço Bozo (#2)");

    p = bozo;

    fprintf(stdout, "Nome: %s, Idade: %d\n", p->nome, p->idade);

    p++;

    fprintf(stdout, "Nome: %s, Idade: %d\n", p->nome, p->idade);

    return 0;
}
```

Executando:

```
$ ./ptrstruct2
Nome: Palhaço Bozo (#1), Idade: 33
Nome: Palhaço Bozo (#2), Idade: 28
```

Agora *bozo* é um vetor. Não precisamos de & para o ponteiro e quando incrementamos *p++* avançamos a quantidade de *bytes* relativa a *sizeof(struct pessoa)*, então *p* aponta para o próximo membro.

É simples... muito mais do que parece.

2.3.2 Ponteiro para função

Também é possível criar ponteiros para funções, muito útil no desenvolvimento de código dinâmico.

Aqui, os caracteres (*nome) devem ser utilizados na definição da variável, a qual deve ter o mesmo protótipo da função.

O & nunca é utilizado pois não existem funções vetores.

ptrfunc.c

```
/*
 * ptrfunc.c: ponteiro para função
 *
 * Para compilar:
 *  cc -Wall ptrfunc.c -o ptrfunc
 *
 * Alexandre Fiori
 */

#include <stdio.h>

static int soma(int x, int y)
{
    return x + y;
}

int main()
{
    int (*somaptr)(int x, int y);

    somaptr = soma;

    fprintf(stdout, "Resultado: %d\n", somaptr(8, 5));

    return 0;
}
```

Executando:

```
$ ./ptrfunc
Resultado: 13
```

Os ponteiros para funções normalmente são utilizados na definição de estruturas para que o objeto de uma estrutura seja capaz de executar funções - procedimento semelhante ao do C++ com métodos em classes.

2.3.3 Ponteiro em argumento de função

São os mais utilizados. Na definição de funções, quando os argumentos são ponteiros, é possível enviar o endereço de uma variável de um escopo a outro, de maneira quase que imperceptível.

argchar.c

```
/*
 * argchar.c: ponteiro char em argumento de função
 *
 * Para compilar:
 *  cc -Wall argchar.c -o argchar
 *
 * Alexandre Fiori
 */
```

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

static int my_strlen(char *str)
{
     int count = 0;

     while(*str++ != '\0') count++;

     return count;
}

int main()
{
     char temp[128];

     memset(temp, 0, sizeof(temp));
     snprintf(temp, sizeof(temp), "sou um vetor");

     fprintf(stdout, "string \"%s\" tem %d bytes.\n", temp, my_strlen(temp));

     return 0;
}
```

Executando:

```
$ ./argchar
string "sou um vetor" tem 12 bytes.
```

A variável *temp* está no escopo da função *main()*. Quando a função *my_strlen(temp)* é chamada, o ponteiro *str* dela terá o endereço do vetor *temp*.

Então, lá no escopo da função *my_strlen()* poderemos acessar a área de memória de *temp* e contar cada um de seus *bytes* através de um *loop*.

Quando isso ocorre, podemos ler ou gravar no ponteiro. Gravando nele, estaremos alterando o vetor *temp*.

No *loop*, alguns detalhes devem ser levados em consideração:

- **str* representa o membro atual, e quando este for *'\0'* o *loop* termina;
- o incremento na variável só é executado depois da comparação, portanto a função compara **p* com *'\0'* e depois avança um *byte*.

Agora veremos um exemplo com ponteiro para estrutura como argumento de função.

argstruct.c

```
/*
 * argstruct.c: ponteiro para estrutura em argumento de função
 *
 * Para compilar:
 *  cc -Wall argstruct.c -o argstruct
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>

struct pessoa {
     int idade;
     char nome[20];
     char desc[50];
```

```
};

static void my_print(struct pessoa *ptr)
{
     fprintf(stdout, "Nome: %s\nIdade: %d\nDescrição: %s\n",
             ptr->nome, ptr->idade, ptr->desc);
}

int main()

{
     struct pessoa bozo;

     memset(&bozo, 0, sizeof(bozo));
     bozo.idade = 31;
     snprintf(bozo.nome, sizeof(bozo.nome), "Bozo");
     snprintf(bozo.desc, sizeof(bozo.desc), "O palhaço de todos vocês!");

     my_print(&bozo);

     return 0;
}
```

Executando:

```
$ ./argstruct
Nome: Bozo
Idade: 31
Descrição: O palhaço de todos vocês!
```

2.3.4 Ponteiro para ponteiro

É totalmente comum apontar um ponteiro para outro ponteiro. O procedimento é exatamente o mesmo, como se estivesse lidando com um vetor.

2.3.5 Matriz de Ponteiros

A matriz mais conhecida é ***argv*, provida por *main()*. Não é nada além de um vetor de vetores.

Especialmente nesta matriz, o índice *argv[0]* aponta para uma *string* que contém o nome do próprio programa executável. Por isso utilizamos *argv.

argcargv.c

```
/*
 * argcargv.c: manuseia a matriz de ponteiros **argv
 *
 * Para compilar:
 *  cc -Wall argcargv.c -o argcargv
 *
 * Alexandre Fiori
 */

#include <stdio.h>

int main(int argc, char **argv)
{
     char **fn;

     if(argc < 2) {
         fprintf(stderr, "use: %s string1 [string2]...\n", *argv);
         return 1;
     } else
         fn = ++argv;

     while(--argc)
```

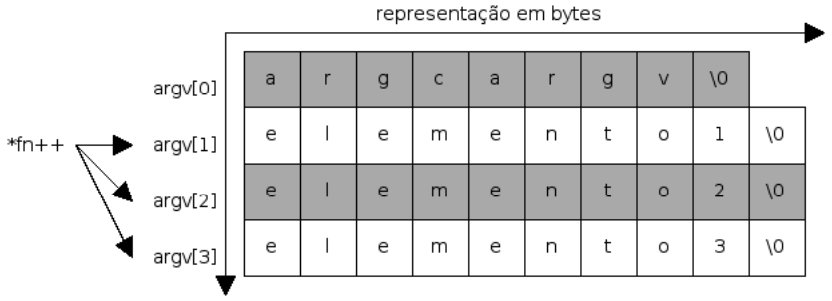
```
            fprintf(stdout, "argumento: %s\n", *fn++);

        return 1;
}
```

Executando:

```
$ ./argcargv
use: ./argcargv string1 [string2]...

$ ./argcargv elemento1 elemento2 elemento3
argumento: elemento1
argumento: elemento2
argumento: elemento3
```

Vamos analisar este programa...

Figura 2.3: Representação gráfica de matriz de ponteiros



Em primeiro lugar, temos *argc* com a quantidade de argumentos digitados na linha de comando, mais um - referente ao próprio nome do programa. Quando *fn = ++argv* significa que *fn* passa a apontar para *argv[1]*. O *loop* decrementa o valor de *argc* até que chegue em zero. Durante o decremento, **fn++* é impresso. Significa que **fn* diz respeito a *argv[1]*, depois *argv[2]* e depois *argv[3]*, onde cada um deles possui outro vetor.

2.4 Técnicas com ponteiros

Os ponteiros são o ponto forte da linguagem C. Com eles você pode manipular qualquer tipo de dados e criar aplicações rápidas e inteligentes.

2.4.1 Inicialização

A inicialização de ponteiros é muito importante. Quando criamos uma variável do tipo ponteiro e o programa é executado, o sistema operacional o coloca em algum lugar da memória RAM onde previamente havia outro programa - o que chamamos de lixo na memória.

Se o ponteiro não for inicializado, ele poderá conter lixo, veja:

```
...
char *p;
fprintf(stdout, "p = %s\n", p);
...
```

Este programa pode funcionar ou não. O conteúdo em *p* é imprevisível. Quando a função *fprintf()* for ler o conteúdo do ponteiro, ela irá imprimir cada *byte* até que encontre *'\0'*. Como não sabemos o que há em *p*, ela pode imprimir a memória toda, para sempre.

Se o sistema operacional detectar que *fprintf()* está violando a área do processo, *address space*, irá finalizar o programa emitindo um sinal do tipo SIGSEGV[1], resultando no fim da aplicação com a

[1] SIGNAL(7) - Linux Programmer's Manual

mensagem Falha de Segmentação.

Portanto, ao criarmos variáveis do tipo ponteiro, devemos inicializá-las com NULL:

```
...
char *p = NULL;
...
```

Aí criamos outro problema. Se uma função tentar ler esta variável, que não aponta para lugar algum, também irá violar o acesso à memória resultando em SIGSEGV[2].

2.4.2 Funções

Algumas funções da *libc* são importantes na utilização de ponteiros e vetores. Aqui veremos algumas delas.

2.4.2.1 A função *memset()*

Segue seu protótipo[3]:

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Esta função é utilizada para preencher determinada área de memória com o caracter definido pelo usuário no argumento *c*.

Basicamente, quando criamos variáveis do tipo vetor ou estruturas de dados, devemos limpar a área de memória antes de utilizar.

Exemplo:

```
...
char temp[1024];
struct stat st;

memset(temp, 0, sizeof(temp));
memset(&st, 0, sizeof(st));
...
```

Como ela recebe um ponteiro em *s*, *temp* pode ser utilizado sem &. Já a variável *st* não é um vetor, portanto utilizamos &.

Neste exemplo as variáveis foram preenchidas com 0, o que chamamos de limpeza.

No caso de *temp*, qualquer valor que lhe for atribuído após o *memset()* deverá ser do tamanho de *sizeof(temp)-1*, para sempre manter o último caracter com o *'\0'* indicando fim de *string*.

2.4.2.2 A função *signal()*

Esta função permite que o programador execute uma determinada função quando o processo recebe um sinal do sistema operacional[4].

Esses sinais podem ser bons ou ruins, ou podem ajudar no controle da aplicação.

Os tipos mais comuns de sinais são:

[2]NULL é normalmente um *#define NULL 0*, um endereço nulo.
[3]MEMSET(3) - Linux Programmer's Manual
[4]SIGNAL(2) - Linux Programmer's Manual

Tabela 2.1: Tabela de sinais *(signal)*

Sinal	Descrição
SIGHUP	Quando o terminal é desligado
SIGINT	Quando o usuário interrompe com CTRL-C
SIGKILL	Somente enviado pelo comando *kill*
SIGSEGV	Quando a aplicação viola o *address space*
SIGPIPE	Quando há problemas com descritores de arquivo
SIGTERM	Quando a aplicação é fechada normalmente
SIGUSR1	Definido pelo usuário
SIGUSR2	Definido pelo usuário
SIGSTOP	Quando o usuário interrompe com CTRL-Z
SIGCONT	Quando volta do SIGSTOP

Todos esses sinais podem ser tratados. Quando a aplicação recebe algum deles, é possível executar uma função do seu código.

A maioria pode ser ignorado, mas alguns como SIGKILL e SIGSEGV não podem.

signal.c

```
/*
 * signal.c: interpreta sinais do sistema operacional
 *
 * Para compilar:
 *  cc -Wall signal.c -o signal
 *
 * Alexandre Fiori
 */

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

static void my_signal(int sig)
{
    fprintf(stdout, "recebendo sinal %d...\n", sig);
    exit(1);
}

int main()
{
    char temp[128];

    signal(SIGHUP,  SIG_IGN);
    signal(SIGPIPE, SIG_IGN);
    signal(SIGINT,  my_signal);
    signal(SIGTERM, my_signal);
    signal(SIGKILL, my_signal);
    signal(SIGSEGV, my_signal);

    while(!feof(stdin)) {
        memset(temp, 0, sizeof(temp));
        if(fgets(temp, sizeof(temp)-1, stdin) == NULL)
            break;
        else
            fprintf(stdout, "texto: %s", temp);
    }

    return 0;
}
```

Executando:

```
$ ./signal
teste, escrevendo no stdin
texto: teste, escrevendo no stdin
recebendo sinal 2... <- quando CTRL-C foi pressionado
```

Neste exemplo, a função *signal()* foi utilizada diversas vezes para especificar a ação a ser tomada mediante determinado sinal.

Os sinais SIGHUP e SIGPIPE serão ignorados por este programa e os demais utilizarão a função *my_signal()* como *callback* - chamada automaticamente pela função *signal()*.

Como é notável, a função *signal()* recebe um ponteiro para função como argumento.

Se durante a execução deste programa o usuário for a outro terminal e executar *killall -TERM signal*, nosso programa irá receber o sinal e imprimí-lo no terminal.

2.4.2.3 A função *atexit()*

Esta função opera como um *scheduler*, agendando funções para serem executadas no término do programa, pela chamada de *exit()* ou *return* na função *main()*.

Segue seu protótipo[5]:

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

Ela também recebe uma função como argumento, portanto temos uma aplicação para testá-la:

atexit.c

```
/*
 * atexit.c: agenda funções para serem executadas ao término do programa
 *
 * Para compilar:
 *  cc -Wall atexit.c -o atexit
 *
 * Alexandre Fiori
 */

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

static void my_finish()
{
    fprintf(stdout, "finalizando dispositivos...\n");
}

static void my_cleanup()
{
    fprintf(stdout, "fechando arquivos de configuração...\n");
}

int main()
{
    atexit(my_finish);
    atexit(my_cleanup);

    return 0;
}
```

Executando:

[5] ATEXIT(3) - Linux Programmer's Manual

```
$ ./atexit
fechando arquivos de configuração...
finalizando dispositivos...
```

Simples e muito eficiente para controlar o término da aplicação. Fechar arquivos abertos, desalocar recursos, etc.

Agora nossa própria versão desta função (apenas educativo):

my-atexit.c

```
/*
 * my-atexit.c: implementação própria de atexit()
 *
 * Para compilar:
 *  cc -Wall my-atexit.c -o my-atexit
 *
 * Alexandre Fiori
 */

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

/* numero máximo de callbacks permitidos */
#define MAX_CALLBACKS 8

/* matriz de ponteiros com os callbacks */
static void *callbacks[MAX_CALLBACKS];
static int   callbacks_pos = 0;

/* implementação de my_atexit() */
static int my_atexit(void (*func)(void))
{
    /* caso haja posição disponível na matriz,
     * guarda o ponteiro */
    if(callbacks_pos < MAX_CALLBACKS) {
        callbacks[callbacks_pos++] = func;
        return 0;
    }

    /* caso já existam MAX_CALLBACKS-1 funções agendadas,
     * retorna erro */
    return -1;
}

/* implementação de my_exit() */
static void my_exit(int status)
{
    void (*func)(void);

    /* executa todas as funções agendadas, de trás pra frente */
    while(callbacks_pos != -1)
        if((func = callbacks[callbacks_pos--]) != NULL)
            func();

    /* termina o programa */
    exit(status);
}

/* funções de teste... */
static void func0(void) { fprintf(stdout, "função 0...\n"); }
static void func1(void) { fprintf(stdout, "função 1...\n"); }
static void func2(void) { fprintf(stdout, "função 2...\n"); }
static void func3(void) { fprintf(stdout, "função 3...\n"); }
static void func4(void) { fprintf(stdout, "função 4...\n"); }
static void func5(void) { fprintf(stdout, "função 5...\n"); }
static void func6(void) { fprintf(stdout, "função 6...\n"); }
static void func7(void) { fprintf(stdout, "função 7...\n"); }
static void func8(void) { fprintf(stdout, "função 8...\n"); }
static void func9(void) { fprintf(stdout, "função 9...\n"); }
```

```
int main()
{
     my_atexit(func0);
     my_atexit(func1);
     my_atexit(func2);
     my_atexit(func3);
     my_atexit(func4);
     my_atexit(func5);
     my_atexit(func6);
     my_atexit(func7);
     my_atexit(func8);
     my_atexit(func9);

     my_exit(0);

     /* nunca será executado! */
     return 0;
}
```

Executando:

```
$ ./my-atexit
função 7...
função 6...
função 5...
função 4...
função 3...
função 2...
função 1...
função 0...
```

Embora o programa não tenha funcionalidade real, pode ser utilizado como refência para a criação de um *scheduler* próprio.

2.4.3 Alocação dinâmica de memória

Trata-se de solicitar determinada quantidade de memória RAM ao sistema operacional para que seja utilizada na aplicação. Normalmente essa memória não faz parte do *address space* do programa até que seja alocada.

Quando o sistema operacional aloca memória para uma aplicação, a área alocada passa a pertencer ao *address space* do programa, porém não é linear, podendo ser alocada em outras páginas de memória bem distantes daquelas onde a aplicação está.

A alocação dinâmica de memória só deve ser feita quando a aplicação irá receber dados de tamanhos imprevisíveis, então após calcular o tamanho necessário pode-se alocar a memória.

Depois de utilizá-la é necessário avisar o sistema operacional, liberando-a para que seja utilizada por outros processos.

2.4.3.1 A função *malloc()*

Esta função tenta alocar *n* bytes de memória RAM no sistema operacional e atribuí-la ao *address space* do programa que a solicitou, através de um ponteiro.

Veja:

```
/* alocação dinâmica de 8192 bytes */
char *p = malloc(8192);
```

O código acima solicita 8192 *bytes* ao sistema operacional. Casa haja memória disponível, um vetor com esta quantidade de *bytes* será alocado em algum lugar da memória RAM e seu endereço será retornado por *malloc()*, que apontamos para *p*.

Agora, em *p,* há 8192 *bytes* para serem utilizados para qualquer fim. Se o ponteiro de *p* for perdido esta memória está perdida.

Exemplo:

```
char *p = malloc(1024);
p = NULL;
```

O endereço da memória estava guardado em *p*, que agora é NULL. Não temos mais como saber qual era o endereço e perdemos a memória. Depois de algum tempo o coletor de lixo *(garbage collector)* do sistema operacional poderá encontrá-la e permitir que seja utilizada por outros processo. Enquanto isso, ela estará inutilizável e disperdiçada.

Quando alocamos memória com *malloc()* e precisamos manipular seu conteúdo, é aconselhável criar um segundo ponteiro, sendo um ponteiro para ponteiro.

Veja:

```
char *p = malloc(2048);
char *temp = p;
...
while(*temp != '\0') { ... }
...
```

Pois mesmo se mudarmos o ponteiro *temp*, podemos a qualquer hora apontá-lo para o início da memória alocada com *temp = p*.

Caso não haja memória disponível no sistema operacional, *malloc()* irá retornar NULL, então é necessário prever esta possível falha.

Exemplo:

```
/* solicitando 8MB para o sistema */
char *p = malloc(8192*1024);
if(p == NULL) {
    perror("malloc");
    exit(1);
}
```

No exemplo imprimimos o erro e fechamos a aplicação com *exit()*, mas este procedimento não é obrigatório.

2.4.3.2 A função *calloc()*

Funciona de maneira semelhante à *malloc()*, porém esta função permite que se especifique o tamanho de cada membro do vetor.

Quando alocamos memória para o tipo *char*, é simples por este tipo tem 1 *byte*. Ao executar *char *p = malloc(20)* temos os exatos 20 *bytes* solicitados.

Mas quando alocamos memória dinâmica para o tipo *int*, por exemplo, devemos tomar cuidado.

Exemplo:

```
int *p = malloc(1024);
```

A função *malloc()* aloca 1024 *bytes*, mas cada *int* tem 4 *bytes*. Quando *p++* for executado, avançará de quatro em quatro *bytes.* Em outras palavras, alocamos um vetor de 256 posições *int* quando solicitamos 1024!

Solução:

```
int *p = malloc(1024 * sizeof(int));
```

Agora sim temos um vetor *int* com 1024 posições.
Entre essas e outras, é aconselhável utilizar *calloc()* por dois motivos:

1. *calloc()* permite definir o tamanho de cada membro do vetor a ser alocado;
2. *calloc()* limpa a memória alocada com *'\0'* antes de retorná-la.

Com *malloc()*, teríamos que executar *memset()* a cada alocação de memória.
Veja um exemplo de *calloc()*:

```
...
char *x = calloc(256, sizeof(char));
...
int *y = calloc(256, sizeof(int));
...
double z = calloc(256, sizeof(double));
...
```

Notável diferença.

2.4.3.3 A função *free()*

Esta é utilizada para desalocar a memória mapeada por *malloc()* ou *calloc()*, mas deve ser utilizada com cuidado.
Um dos maiores problemas encontrados pelos programadores é gerenciar a memória e manter as aplicações estáveis, diminuindo o trabalho do coletor de lixo do sistema operacional.
Exemplo:

```
...
char *p = malloc(1024);
...
free(p);
...
```

Este é o uso correto de *free()*. Caso o argumento passado para *free()* seja uma variável que não contém memória alocada por *malloc()* ou *calloc()*, o sistema operacional irá emitir um sinal SIGSEGV e finalizar o processo imediatamente.
Uma boa maneira de utilizar *free()* é colocá-la em uma função agendada para executar no final do programa através de *atexit()*, descrito na Sessão 2.4.2.3.

2.4.3.4 A função *realloc()*

Segue seu protótipo[6]:

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Esta função recebe em *ptr* um endereço previamente alocado por *malloc()* e reajusta o tamanho da memória para *size*. Caso o novo tamanho seja menor que o anterior, parte dos dados (o final do vetor) serão perdidos. Caso o novo tamanho seja maior, a nova memória não estará limpa.
Se *ptr* for NULL, a chamada a esta função é equivalente a *malloc(size)*. Caso *ptr* seja válido e *size* seja 0, a chamada a esta função é equivalente a *free(ptr)*.

[6] MALLOC(3) - Linux Programmer's Manual

2.4.3.5 As funções *mmap()*, *munmap()* e *msync()*

Permite mapear um descritor de arquivo na memória especificando suas permissões e tipo de acesso[7]. Retorna um ponteiro para a memória alocada do tamanho solicitado na chamada.

Esta função é útil para acessar dispositivos do sistema, como por exemplo os dispositivos do Video4Linux que dão acesso às imagens de placas digitalizadores através de arquivos no */dev/videoN*. Com *mmap()* pode-se acessar o *buffer* das imagens capturadas e processá-las sem que haja *read()*.

A função *munmap()* serve para desalocar a memória mapeada por *mmap()*.

A função *msync()* deve ser utilizada para sincronizar a memória mapeada por *mmap()* foi modificada de volta no arquivo de origem[8].

Segue uma demonstração de seu uso:

cat-mmap.c

```
/*
 * cat-mmap.c: mapeia um arquivo na memória e imprime seu conteúdo no terminal
 *
 * Para compilar:
 *  cc -Wall cat-mmap.c -o cat-mmap
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>

static void cat_mmap(char *filename)
{
    int fd;
    struct stat st;
    char *mem = NULL;

    /* abre o arquivo */
    if((fd = open(filename, O_RDONLY)) == -1) {
        perror("open");
        return;
    }

    /* obtem informações do arquivo */
    if(fstat(fd, &st) == -1) {
        perror("fstat");
        close(fd);
        return;
    }

    /* mapeia */
    mem = mmap(0, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if(mem == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return;
    } else
        close(fd);

    /* imprime no terminal */
    fprintf(stdout, "-- %s (%d bytes) --\n",
            filename, (int) st.st_size);
    fwrite(mem, sizeof(char), st.st_size, stdout);
```

[7] MMAP(2) - Linux Programmer's Manual
[8] MSYNC(2) - Linux Programmer's Manual

```
    /* desaloca a memória */
    munmap(mem, st.st_size);
}

int main(int argc, char **argv)
{
    char **fn;

    if(argc < 2) {
        fprintf(stderr, "use: %s arquivo[s]...\n", *argv);
        return 1;
    } else
        fn = ++argv;

    while(--argc)
        cat_mmap(*fn++);

    return 0;
}
```

2.4.4 Listas ligadas

As listas ligadas são estruturas de dados definidas pelo programador que permitem alocar novos membros sob demanda. São utilizadas em casos semelhantes às matrizes, porém as matrizes tem quantidade fixa de membros e as listas não.

Normalmente definimos o tipo lista com código semelhante a este:

```
typedef struct _list Pessoa;
struct _list {
    int idade;
    char nome[128];

    Pessoa *next;
}
```

Agora temos os tipos *struct _list* e também *Pessoa*, e no código só será utilizado o tipo *Pessoa*. Quando um ponteiro deste tipo for criado, a quantidade de memória *sizeof(Pessoa)* deve ser alocada a ele, então teremos um local para armazenar dados para seus membros *idade* e *nome*. Veja:

```
Pessoa *p = malloc(sizeof(Pessoa));
p->idade = 30;
snprintf(p->nome, sizeof(p->nome), "Nome da pessoa");
```

Sendo assim, caso seja necessário criar mais cadastros de pessoas, ao invés de criarmos outra variável como *p*, podemos simplesmente usar *p->next* que é um ponteiro vazio, que não aponta para lugar algum.

Veja como podemos criar um novo membro:

```
p->next = malloc(sizeof(Pessoa));
p = p->next;
p->idade = 25;
snprintf(p->nome, sizeof(p->nome), "Nome da outra pessoa");
```

Agora alocamos mais memória mas temos um problema: perdemos o endereço do ponteiro da primeira memória ao executar *p = p->next* e nunca mais teremos acesso a ele.

Esse exemplo é baseado em uma lista do tipo simples, que só possui o ponteiro *next*. Existem as listas duplamente ligadas que possuem *next* e *prev*, onde um aponta para o próximo e outro aponta para o anterior, respectivamente.

Ao criar as listas devemos tomar o cuidado de nunca perder o endereço do primeiro membro, pois sem ele não teremos como acessar nem desalocar a memória alocada por *malloc()*. O resultado

disso é que quando o programa terminar, deixará memória alocada no sistema resultando em *memory leak*.

Segue o código de uma lista ligada simples:

simplelist.c

```
/*
 * simplelist.c: lista ligada simples com alocação dinâmica
 *
 * Para compilar:
 *  cc -Wall simplelist.c -o simplelist
 *
 * Alexandre Fiori
 */

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

/* tipo Pessoa é a lista ligada */
typedef struct _pessoa Pessoa;
struct _pessoa {
    int idade;
    char nome[128];

    Pessoa *next;
};

/* aloca memória para um tipo 'Pessoa' */
static Pessoa *list_init(void);

/* adiciona um novo membro na lista
 * caso 'list' seja NULL, significa que devemos criar
 * a lista com os dados informados
 * caso 'list' já seja uma lista criada, devemos adicionar
 * um novo membro no final da lista */
static Pessoa *list_add(int idade, char *nome, Pessoa *list);

int main()
{
    Pessoa *root, *bkp;

    /* adiciona membros na lista */
    root = list_add(22, "José Maria", NULL);
    root = list_add(31, "Clebiana Rosine", root);
    root = list_add(35, "Josivan Alencar", root);
    root = list_add(27, "Raildes Carolina", root);

    /* como 'root' é sempre o primeiro membro da lista
     * podemos utilizá-lo para andar por cada cadastro
     * e imprimir seu conteúdo */

    while(root) {
        fprintf(stdout, "Nome: %s, idade: %d\n", root->nome, root->idade);
        if(root->next) {
            /* guarda cópia do endereço em 'root' */
            bkp = root;

            /* 'root' será o próximo membro */
            root = root->next;

            /* desaloca a memória do membro que já foi impresso
             * e ficou guardado em 'bkp' */
            free(bkp);
        } else {
            /* desaloca a memória do último membro */
            free(root);
            root = NULL;
            break;
        }
    }
```

```
    return 1;
}

static Pessoa *list_init(void)
{
    Pessoa *mem = malloc(sizeof(Pessoa));

    if(!mem) {
        perror("malloc");
        exit(1);
    } else
        memset(mem, 0, sizeof(Pessoa));

    return mem;
}

static Pessoa *list_add(int idade, char *nome, Pessoa *list)
{
    Pessoa *root, *temp;

    /* 'root' sempre aponta para o início da lista */

    /* ao adicionar o primeiro membro precisamos
     * criar a lista */
    if(list == NULL) root = temp = list_init();
    else {
        /* caso a lista já exista, devemos alocar
         * memória no último membro */
        root = temp = list;
        while(temp)
            if(temp->next == NULL) break;
            else temp = temp->next;

        /* agora temp->next é um membro vazio
         * e será utilizado */
        temp->next = list_init();
        temp = temp->next;
    }

    /* copia os dados no novo membro da lista */
    temp->idade = idade;
    strncpy(temp->nome, nome, sizeof(temp->nome));

    return root;
}
```

Executando:

```
$ ./simplelist
Nome: José Maria, idade: 22
Nome: Clebiana Rosine, idade: 31
Nome: Josivan Alencar, idade: 35
Nome: Raildes Carolina, idade: 27
```

Todos os membros foram alocados, seus dados foram atribuídos aos respectivos locais (*nome* e *idade*) e depois foram impressos. Logo após imprimir a memória já foi desalocada, para evitar *memory leak*.

Existem ainda listas ligadas mais complexas que permitem remover alguns membros durante seu uso. Um bom exemplo de lista ligada está disponível na biblioteca *glib*[9], distribuída com o GTK. Lá a biblioteca possui funções para criar e manipular listas.

[9] GTK e GLIB - http://www.gtk.org

Capítulo 3

Manipulação de Dados

Trabalhar com arquivos é tarefa fundamental na linguagem C. Nos sistemas *Unix-like*, toda configuração dos programas é feita através de arquivos - texto ou binário.

Para entender o que são os arquivos é necessário conhecer um pouco sobre como o sistema operacional lida com eles, e o que são os sistemas de arquivos *(filesystems)*.

Sistemas operacionais diferentes utilizam sistemas de arquivos diferentes, sendo eles o formato utilizado para gerenciar o conteúdo dos discos rígidos e outros dispositivos de bloco. No Linux, por exemplo, temos como padrão o formato *ext3*, que é uma versão aprimorada do *Second Extended* conhecido como *ext2*.

Existem ainda outros formatos, como por exemplo o FAT16, FAT32, NTFS, UFS, ISO9660 que é utilizado em CDs e o UDF, utilizado em DVDs.

Um sistema de arquivos é algo semelhante a um protocolo, composto por um conjunto de regras e características que permite o gerenciamento de dados lógicos em ambientes físicos, como é o caso dos discos rígidos.

3.1 Os sistemas de arquivos

Quando um disco é formatado, leva consigo um padrão definido pelo sistema de arquivos. Significa que discos com sistemas de arquivos diferentes têm a informação organizada de maneira diferente. Como o sistema de arquivos é parte fundamental do sistema operacional, muito do que há formato dos discos é relacionado com as características do próprio sistema operacional.

Porém, existem algumas características padronizadas entre a maior parte dos sistemas de arquivos: a segmentação do disco.

Formatar um disco nem sempre significa apagar todos os dados. Boa parte dos sistemas simplesmente colocam no disco um cabeçalho com suas informações e na medida que novos arquivos são gravados no disco, seguem o padrão do sistema de arquivos. Essas informações são relacionadas ao tamanho dos blocos utilizados para ler e gravar dados *(IO Block)*. Além disso, junto com cada arquivo há uma série de dados adicionais relativos ao sistema operacional: usuário dono do arquivo, grupo, permissões, data de criação, data de modificação, etc.

O padrão POSIX define as funções *stat()* e *fstat()*[1], utilizadas para preencher uma estrutura *struct stat* com os dados de determinado arquivo. De maneira genérica, é fácil entender os sistemas de arquivos conhecendo essa estrutura, veja:

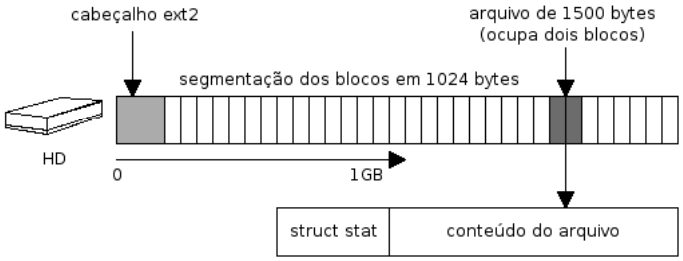
```
struct stat {
  dev_t         st_dev;      /* device */
  ino_t         st_ino;      /* inode */
```

[1] STAT(2) - Linux Programmer's Manual

```
  mode_t        st_mode;     /* protection */
  nlink_t       st_nlink;    /* number of hard links */
  uid_t         st_uid;      /* user ID of owner */
  gid_t         st_gid;      /* group ID of owner */
  dev_t         st_rdev;     /* device type (if inode device) */
  off_t         st_size;     /* total size, in bytes */
  blksize_t     st_blksize;  /* blocksize for filesystem I/O */
  blkcnt_t      st_blocks;   /* number of blocks allocated */
  time_t        st_atime;    /* time of last access */
  time_t        st_mtime;    /* time of last modification */
  time_t        st_ctime;    /* time of last status change */
};
```

O conteúdo de *struct stat* é adicionado a cada arquivo gravado no disco e contém toda a informação necessária para manipulá-lo.

Figura 3.1: Representação gráfica de sistema de arquivos



A figura acima é uma representação simbólica dos arquivos no disco e não significa que os sistemas de arquivos trabalhem extamente assim. O importante é saber que quando o disco é segmentado, os arquivos utilizam blocos. No exemplo, temos um arquivo de 1500 *bytes* no sistema de arquivos com blocos de 1024. Consequentemente esse arquivo ocupa dois blocos e um pedaço do disco é disperdiçado. Porém, para ambientes que possuem arquivos muito grandes, ler os blocos a cada 1024 *bytes* pode tornar o sistema lento. O aconselhável é manter a formatação do disco com blocos de 4096 *bytes*, tornando o sistema ~4x mais rápido que o anterior, com blocos de 1024.

O nome do arquivo e o local hierárquico na estrutura de diretórios não está na *struct stat*. Essas informações estão em outra estrutura do próprio sistema de arquivos. Podemos concluir que um arquivo ocupa mais *bytes* que seu tamanho real, pois informações adicionais são atibuídas a ele.

O comando *stat* permite visualizar todo o conteúdo da *struct stat*, veja:

```
$ ls -l
-rw-r--r--  1 bozo bozo 5000 2005-11-24 16:06 meuarquivo

$ stat meuarquivo
  File: 'meuarquivo'
  Size: 5000            Blocks: 16         IO Block: 4096   regular file
Device: 806h/2054d      Inode: 786105      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/    bozo)   Gid: ( 1000/    bozo)
Access: 2005-11-24 16:06:17.000000000 -0200
Modify: 2005-11-24 16:06:17.000000000 -0200
Change: 2005-11-24 16:06:17.000000000 -0200
```

Note que o tamanho dos blocos do sistema de arquivos *(IO Block)* é de 4096 *bytes*, mas a quantidade de blocos é 16. Por que um arquivo de 5000 *bytes* utiliza 16 blocos quando deveria utilizar 2?

Porque na *struct stat* a representação dos blocos é sempre em unidades de 512. Podemos calcular a quantidade real de blocos de uma maneira muito simples:

```
512 * blocos /  IO Block
```

Veja:

```
512 * 16 / 4096 = 2
```

Os diretórios normalmente são arquivos do tamanho do *IO Block* do sistema de arquivos.

3.2 A função *stat()*

Utilizamos a função *stat()* muitas vezes para obter informações como permissões ou tamanho dos arquivos. No campo *st_mode* da *struct stat* temos a informação do tipo de arquivo além do modo em si. O tipo do arquivo indica se ele é um arquivo comum, um diretório, etc. O modo diz respeito às permissões, como leitura, gravação e execução para o usuário, grupo ou todos os demais.

O arquivo de cabeçalho *sys/stat.h* provê algumas *macros* para identificar o tipo de arquivo, veja:

Tabela 3.1: *Macros* e tipos de arquivos

Macro	Tipo
S_ISREG	Arquivo comum
S_ISDIR	Diretório
S_ISCHR	Dispositivo de caracteres *(character device)*
S_ISBLK	Dispositivo de blocos *(block device)*
S_ISFIFO	*Fifo*
S_ISLNK	*Link* simbólico
S_ISSOCK	*Unix Domain Socket*

Segue um exemplo:

filetype.c

```
/*
 * filetype.c: imprime o tipo de arquivo e tamanho utilizando stat()
 *
 * Para compilar:
 *  cc -Wall filetype.c -o filetype
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

/* imprime o tipo do arquivo e tamanho */
static void filetype(const char *filename);

/* necessita ao menos dois argumentos */
int main(int argc, char **argv)
{
    char **fn;

    if(argc < 2) {
        fprintf(stderr, "use: %s arquivo [arquivo...]\n", *argv);
        return 1;
    } else
        fn = ++argv;

    while(--argc)
```

```
        filetype(*fn++);

    return 0;
}

static void filetype(const char *filename)
{
    int size;
    struct stat st;
    char temp[1024];

    /* obtem os dados do arquivo e preenche a estrutura
     * em &st */
    if(lstat(filename, &st) == -1) {
        perror("stat");
        return;
    }

    size = (int) st.st_size;
    memset(temp, 0, sizeof(temp));

    if(S_ISREG(st.st_mode))
        fprintf(stdout, "%s: arquivo comum, %d bytes\n", filename, size);
    else
    if(S_ISDIR(st.st_mode))
        fprintf(stdout, "%s: diretório, %d bytes\n", filename, size);
    else
    if(S_ISCHR(st.st_mode))
        fprintf(stdout, "%s: character device\n", filename);
    else
    if(S_ISBLK(st.st_mode))
        fprintf(stdout, "%s: block device\n", filename);
    else
    if(S_ISFIFO(st.st_mode))
        fprintf(stdout, "%s: fifo\n", filename);
    else
    if(S_ISLNK(st.st_mode)) {
        readlink(filename, temp, sizeof(temp));
        fprintf(stdout, "%s: link para %s\n", filename, temp);
    } else
    if(S_ISSOCK(st.st_mode))
        fprintf(stdout, "%s: unix domain socket\n", filename);
}
```

Executando:

```
$ ./filetype /etc/passwd
/etc/passwd: arquivo comum, 915 bytes

$ ./filetype /var
/var: diretório, 4096 bytes

$ ./filetype /dev/ttyS0 /dev/hda
/dev/ttyS0: character device
/dev/hda: block device

$ ./filetype /etc/rc2.d/S20ssh
/etc/rc2.d/S20ssh: link para ../init.d/ssh

$ ./filetype /tmp/.X11-unix/X0
/tmp/.X11-unix/X0: unix domain socket
```

A diferença entre as funções *stat()* e *lstat()* é que a primeira reconhece os *links* simbólicos como arquivos comuns. Já a segunda, *lstat()*, reconhece perfeitamente mas não segue o padrão POSIX.1.1996.

3.3 A função *perror()*

Boa parte das chamadas de sistema e funções da *libc* podem gerar erros. Quando isso acontece, elas preenchem uma variável do tipo *int* chamada *errno* com o número do erro. Esta variável normalmente não está presente no código pois é estática dentro da *libc*. Caso queira ter acesso a ela, basta chamar *#include <errno.h>*.

A função *perror()* é utilizada para imprimir a descrição do erro baseado em *errno*.

Exemplo:

```
int fd = open("/tmp/meuarquivo", O_RDONLY);
if(fd == -1) perror("open");
```

O argumento em *perror()* não é relacionado com a função que gerou o erro. Ele simplesmente imprime a descrição do erro presente em *errno*. Portanto, é possível utilizá-lo assim:

```
int fd = open("/tmp/meuarquivo", O_RDONLY);
if(fd == -1) perror("abrindo arquivo");
```

Para mais informações consulte PERROR(3).

3.4 Funções para manipular arquivos

Nosso sistema operacional é constituído de camadas. São apenas duas: *kernel level* e *user level*. Na primeira camada, o *kernel level*, os arquivos são estruturas do tipo *struct inode* e as operações como leitura e gravação de dados são estruturas do tipo *struct file_operations*[2].

Já no *user level* os arquivos são representados por números inteiros, do tipo *int*. Esse número é relacionado a uma *struct inode* do *kernel space*.

Ainda no *user level* temos duas maneiras de acessar os arquivos: pelas chamadas de sistema *(syscalls)* ou pelas chamadas da *libc*. Quando utilizamos as chamadas de sistema trabalhamos diretamente com o descritor de arquivos, e quando utilizamos as chamadas da *libc* trabalhamos através de um tipo ponteiro *FILE*.

As funções relacionadas à manipulação de arquivos possuem muitas propriedades e são muito bem documentadas nas páginas de manual do sistema. Como o objetivo aqui não é reproduzir os manuais, as funções serão apresentadas apenas na tabela 3.2.

Todas as chamadas de sistema estão na sessão 2 do manual, enquanto as chamadas da *libc* estão na sessão 3.

[2]Podem ser encontradas no arquivo */usr/include/linux/fs.h*

Tabela 3.2: Funções para manipulação de arquivos

Nome	Manual	Descrição
open()	OPEN(2)	Abre um arquivo para leitura, gravação ou ambos
read()	READ(2)	Lê dados de arquivo aberto
write()	WRITE(2)	Escreve dados em arquivo aberto
lseek()	LSEEK(2)	Reposiciona o *offset* de arquivo aberto
dup()	DUP(2)	Duplica descritor de arquivo
fcntl()	FCNTL(2)	Manipula descritor de arquivo
close()	CLOSE(2)	Fecha arquivo previamente aberto
fopen()	FOPEN(3)	Abre um arquivo para leitura, gravação ou ambos
fread()	FREAD(3)	Lê dados binários de arquivo aberto
fwrite()	FREAD(3)	Escreve dados em arquivo aberto
fseek()	FSEEK(3)	Reposiciona o *offset* de arquivo aberto
ftell()	FSEEK(3)	Retorna o *offset* de arquivo aberto
fgets()	GETS(3)	Lê linha por linha de arquivo aberto
fprintf()	PRINTF(3)	Escreve dados em arquivo aberto
feof()	FERROR(3)	Retorna positivo quando encontra fim de arquivo
fileno()	FERROR(3)	Retorna o descritor de arquivo de *FILE*
fclose()	FCLOSE(3)	Fecha arquivo previamente aberto

Essas são apenas algumas das funções mais utilizadas na manipulação de arquivos. Todas (ou quase) serão utilizadas em diversos exemplos ao longo dos capítulos, então basta ler o exemplo e consultar a página de manual.

Para acessar os manuais deve-se informar a sessão:

```
$ man 2 open
$ man 3 fseek
```

3.5 Os arquivos *stdin*, *stdout* e *stderr*

São arquivos especiais representados de maneiras diferentes para as chamadas de sistema e chamadas da *libc*.

A linguagem C provê os três arquivos e são representados pelos números 0, 1 e 2, sendo *stdin, stdout* e *stderr*, respectivamente.

Na *libc* são criados com os nomes ao invés de números, e são do tipo *FILE* *.

Para escrever dados no terminal utilizando a função *write()*, podemos escrever no arquivo 1, veja:

```
write(1, "teste\n", 6);
```

Para realizar a mesma tarefa pela função *fprintf()*, escrevemos no arquivo chamado *stdout*, veja:

```
fprintf(stdout, "teste\n");
```

Internamente ambas fazem a mesma coisa.

3.6 Lista de argumentos variáveis

Podemos criar funções onde a lista de argumentos tem tamanho indefinido, assim como as funções *printf()* e *fprintf()*.

Para tornar isso possível devem utilizar a função *va_start()*[3], que permite a criação da lista de acordo com a quantidade de argumentos lá presentes.

Esse sistema é muito útil na manipulação de erros e muitos outros casos.

Segue um exemplo:

```
/*
 * debug.c: cria sistema de depuração ativado por variável de ambiente
 *
 * Para compilar:
 *  cc -Wall debug.c -o debug
 *
 * Alexandre Fiori
 */

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

/* função com lista de argumentos variáveis */
static void debug(const char *fmt, ...)
{
    va_list ap;

    if(getenv("DEBUG")) {
        va_start(ap, fmt);
        fprintf(stdout, "debug: ");
        vfprintf(stdout, fmt, ap);
        va_end(ap);
    }
}

int main()
{
    int x = 1, y = 2;

    fprintf(stdout, "realizando procedimento X...\n");
    debug("calculando resultado de x + y: %d\n", x + y);

    fprintf(stdout, "realizando procedimento Y...\n");
    debug("calculando resultado de y - x: %d\n", y - x );

    return 0;
}
```

Executando:

```
$ ./debug
realizando procedimento X...
realizando procedimento Y...

$ export DEBUG=1
$ ./debug
realizando procedimento X...
debug: calculando resultado de x + y: 3
realizando procedimento Y...
debug: calculando resultado de y - x: 1
```

O sistema *stdarg* é útil em diversas situações.

3.7 Interpretando arquivos texto

As partes dos programas que interpretam dados são conhecidas como *parsers*. Para interpretar o conteúdo de um arquivo é necessário abrí-lo, ler linha por linha e interpretá-la. De acordo com o valor da linha, toma-se alguma decisão.

[3] STDARG(3) - Linux Programmer's Manual

Os *parsers* costumam ser complicados e enigmáticos, pelo fato de utilizarem ponteiros para lá e para cá, além de tornar quase que invisível o tratamento dos dados que realmente acontece. Normalmente são dinâmicos, estão prontos para lidar com qualquer tipo de informação padronizada seguindo algumas regras estabelecidas pelo próprio programador.

Para criar um *parser* é necessário conhecer os ponteiros muitos bem, pois só eles permitem o tratamento direto com as *strings*, normalmente linhas de configuração de determinado arquivo.

3.7.1 Arquivos de configuração

Existem diversos tipos de arquivos de configuração. No diretório */etc* do sistema há pelo menos 10 tipos de arquivos diferentes, com formatos diferentes - embora a maioria seja texto.

Se compararmos por exemplo o arquivo */etc/sysctl.conf* com o arquivo */etc/hosts*, teremos tipos diferentes de configuração. No primeiro, o padrão é caracterizado por "opção = valor", enquanto no segundo é "opção valor".

Para arquivos com padrões diferentes são necessários *parsers* diferentes. Além disso, mesmo quando um padrão é definido, o usuário que configura o arquivo pode cometer enganos ou na tentativa de deixar a configuração bonita, inserir espaços ou TABs a mais.

Para entendermos os *parsers*, teremos um arquivo de configuração com o padrão "opção = valor", veja:

sample.conf

```
#
# sample.conf: arquivo de configuração de demonstração
#

language=ANSI C
compiler =gcc
parser= for(), feof() and fgets()
editor = vi
os version=Linux 2.6.14 SMP
programmer name =    Alexandre Fiori
programmer home =    /home/fiorix
programmer uid = 1000
```

Apesar de seguir o padrão, ele está todo bagunçado. Em algumas linhas o usuário configurou "opção=valor", em outras linhas "opção =valor", ainda em outras "opção= valor", e para completar "opção = (TAB) valor".

Imagine, como programador, prever todos esses detalhes para escrever o *parser*. É, parceiro, é necessário prever tudo que o usuário poderá fazer na hora de configurar um determinado programa através desses arquivos.

3.7.2 Os *parsers*

Como fazemos para interpretar arquivos como o da sessão anterior, *sample.conf*? Não existe uma maneira genérica de interpretar linhas senão usando ponteiros. Ao olhar para um arquivo de configuração, como programador, é necessário criar um pequeno algoritmo antes de começar a escrever o código.

Quando sua aplicação precisar de um arquivo de configuração, nunca comece pelo código. Sempre, antes de mais nada, crie um arquivo de configuração que pareça ser válido para o que deseja.

Vamos analisar os passos para criar um *parser* para *sample.conf*.

1. Criar um local de armazenamento *(buffer)* para cada opção da configuração

2. Prever todas as possíveis opções
3. Abrir o arquivo de configuração
4. Em um *loop*, ler cada linha para uma variável
5. Se a linha iniciar com '\r' ou '\n' ou '#' significa linha em branco ou comentário, devemos ignorar
6. Devemos verificar se a linha possui o caracter '=', então assumimos que o que está para trás é opção e o que está para frente é valor
7. Entre o caracter '=' e o valor pode haver espaços ou TABs, devemos levar em consideração
8. Comparar o começo da linha, a opção, com cada uma das possíveis opções
9. Guardar o valor no local correto de armazenamento

Assim, conseguiremos interpretar esse simples arquivo.
Segue o código:

parsefile.c

```
/*
 * parsefile.c: interpreta arquivo de configuração
 *
 * Para compilar:
 *  cc -Wall parsefile.c -o parsefile
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>

#define CONFIG_FILE "sample.conf"

/* define um tipo para ser utilizado como
 * local de armazenamento da configuração */
typedef struct {
    char language[10];
    char compiler[10];
    char parser[50];
    char editor[10];
    char osver[50];
    char name[128];
    char home[128];
    char uid[5];
} config_t;

/* função para remover '\r' ou '\n' do final
 * das linhas */
static char *strfix(char *str)
{
    char *p;

    /* anda por 'str' até encontrar '\r' ou '\n'
     * e quando encontra troca por '\0' */
    for(p = str; *p != '\0'; p++)
        if(*p == '\r' || *p == '\n') {
            *p = '\0';
            break;
        }

    return str;
}
```

```
int main()
{
    int i;
    FILE *fp;
    char temp[128], *p;

    /* local para armazenar a configuração
     * do arquivo CONFIG_FILE */
    config_t conf;

    /* estrutura genérica para definir as
     * opções de configuração */
    typedef struct {
        char *optname;
        char *buffer;
        int  size;
    } parser_t;

    /* todas as opções de configuração e o
     * respectivo local de armazenamento */
    parser_t opts[] = {
        { "language",        conf.language, sizeof(conf.language) },
        { "compiler",        conf.compiler, sizeof(conf.compiler) },
        { "parser",          conf.parser,   sizeof(conf.parser)   },
        { "editor",          conf.editor,   sizeof(conf.editor)   },
        { "os version",      conf.osver,    sizeof(conf.osver)    },
        { "programmer name", conf.name,     sizeof(conf.name)     },
        { "programmer home", conf.home,     sizeof(conf.home)     },
        { "programmer uid",  conf.uid,      sizeof(conf.uid)      }
    };

    /* tamanho real de opts */
    const int opts_size = sizeof(opts)/sizeof(opts[0]);

    /* abre o arquivo de configuração */
    if((fp = fopen(CONFIG_FILE, "r")) == NULL) {
        perror("open");
        return 1;
    }

    /* limpa o local de armazenamento */
    memset(&conf, 0, sizeof(conf));

    /* loop até o fim do arquivo */
    while(!feof(fp)) {
        memset(temp, 0, sizeof(temp));

        /* lê uma linha do arquivo e encerra o loop
         * caso ocorra algum erro */
        if(fgets(temp, sizeof(temp), fp) == NULL) break;

        /* volta ao começo do loop se encontrar linhas
         * em branco ou comentários...
         * (ignora a linha) */
        p = temp;
        if(*p == '\r' || *p == '\n' || *p == '#')
            continue;

        /* procura o caracter '=' e caso não exista
         * ignora a linha */
        while(*p != '=' && *p != '\0') p++;
        if(*p == '=') p++;
        else continue;

        /* depois do caracter '=' pode ter espaço ou
         * tab, então pula todos até encontrar o
         * conteúdo da opção */
        while((*p == ' ' || *p == '\t') && *p != '\0') p++;
        if(*p == '\0') continue;

        /* compara os dados lidos com todas as possíveis
         * opções... quando encontra, grava no local de
         * armazenamento correto */
```

```
            for(i = 0; i < opts_size; i++)
                if(!strncmp(opts[i].optname, temp, strlen(opts[i].optname)))
                    strncpy(opts[i].buffer, strfix(p), opts[i].size);
        }

        /* fecha o arquivo de configuração */
        fclose(fp);

        /* imprime todas as opções lidas */
        for(i = 0; i < opts_size; i++)
            if(*opts[i].buffer != '\0')
                fprintf(stdout, "[%s] = [%s]\n",
                        opts[i].optname, opts[i].buffer);

        return 0;
    }
```

Executando:

```
$ ./parsefile
[language] = [ANSI C]
[compiler] = [gcc]
[parser] = [for(), feof() and fgets()]
[editor] = [vi]
[os version] = [Linux 2.6.14 SMP]
[programmer name] = [Alexandre Fiori]
[programmer home] = [/home/fiorix]
[programmer uid] = [1000]
```

Ele pode até parecer complicado, mas não é. As funções que lidam com *strings*, sendo *strncmp()* e *strncpy()* têm propriedades importantes neste programa. A documentação de ambas está disponível nas páginas de manual STRCMP(3) e STRCPY(3).

Depois de alguma prática, códigos como este passam a fazer parte da rotina. No início pode haver dificuldade, mas nada que um pedaço de papel e uma caneta não resolva.

3.8 Interpretando arquivos XML

O XML é um formato de arquivo texto que permite criar estruturas com dados organizados. Não há um padrão para organizar o conteúdo no XML, o que há é a maneira de estruturar o arquivo. A sintaxe do XML é padronizada, mas o conteúdo varia de acordo com as características dos dados que serão colocados lá.

Hoje é comum entre os sistemas a utilização de XML para troca de dados em ambientes diferentes, como por exemplo em programas na Web e bancos de dados. Porém, os programas que escrevem o XML e os programas que interpretam precisam conhecer a estrutura como os dados foram organizados lá.

O XML no Linux é interpretado através de uma biblioteca chamada *libxml*[4] que hoje faz parte do projeto GNOME[5].

Para escrever arquivos XML em C o programador pode optar por utilizar a biblioteca ou não, pois como são arquivos texto é possível escrevê-los até com *fprintf()*.

3.8.1 Criando arquivos XML

Normalmente os arquivos XML são criados manualmente por programadores ou usuários utilizando editores de texto comuns.

[4] Biblioteca XML - http://www.xmlsoft.org
[5] Projeto GNOME - http://www.gnome.org

Para um ambiente de aprendizado, teremos o seguinte arquivo:

sample.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    sample.xml: arquivo de configuração XML
-->

<config>
    <system name="leibniz">
        <kernel version="2.6.14"/>
        <kernel bogomips="5611.93"/>
        <kernel image="/boot/bzImage-2.6.14"/>
    </system>

    <system name="galois">
        <kernel version="2.6.13"/>
        <kernel bogomips="5192.12"/>
        <kernel image="/boot/bzImage-2.6.13"/>
    </system>
</config>
```

Esse arquivo foi criado utilizando o editor *vim* e pode ser modificado desde que siga o mesmo padrão estabelecido, caso contrário o *parser* não será capaz de interpretar o conteúdo e informará um erro.

3.8.2 O *parser* XML

A documentação completa das funções da *libxml* estão disponível em seu site, onde também há exemplos.

Basicamente, a biblioteca interpreta o arquivo XML inteiro e cria uma lista ligada para que o programador possa navegar pelas sessões do arquivo.

Primeiro é necessário interpretar o XML utilizando a função *xmlParseFile()*, depois obter o ponteiro para o documento, utilizando *xmlDocGetRoot()*.

Esse ponteiro, então, será utilizado para encontrar o primeiro *node* do arquivo, sendo o <*config*> do nosso arquivo de exemplo *sample.xml*. Esse *node* é o local principal onde estão as configurações.

Como ele é uma lista hierárquica ligada, podemos acessar o conteúdo dentro do *node* utilizando *node->xmlChildrenNode*, então caímos no mesmo ambiente de <*system name="nome"*>. Para navegar dentro de cada *node* <*system*> basta utilizar o mesmo procedimento e para ir de um node a outro a biblioteca conta com *node->next* e *node->prev*.

Segue o código:

parsexml.c

```
/*
 * parsexml.c: interpreta arquivo XML
 *
 * Para compilar:
 *  cc -Wall $(xml-config --cflags) parsexml.c -o parsexml $(xml-config --libs)
 *
 * Alexandre Fiori
 */

/* sistema */
#include <stdio.h>
#include <string.h>
```

```
#include <stdarg.h>
#include <stdlib.h>

/* libxml */
#include <xmlmemory.h>
#include <parser.h>

#define CONFIG_FILE "sample.xml"

int main()
{
    xmlDocPtr doc = NULL;
    xmlChar *value = NULL;
    xmlNodePtr node = NULL, sub = NULL;

    /* imprime erro e finaliza o programa */
    void error(const char *fmt, ...) {
        va_list ap;

        va_start(ap, fmt);
        vfprintf(stderr, fmt, ap);
        va_end(ap);

        if(doc) xmlFreeDoc(doc);

        exit(1);
    }

    /* abre o arquivo XML */
    if((doc = xmlParseFile(CONFIG_FILE)) == NULL)
        error("Arquivo %s inválido (doc).\n", CONFIG_FILE);

    /* obtem o ROOT do XML */
    if((node = xmlDocGetRootElement(doc)) == NULL)
        error("Arquivo %s inválido (node).\n", CONFIG_FILE);

    /* caso o primeiro node não seja 'config', retorna erro */
    if(xmlStrcmp(node->name, (const xmlChar *) "config"))
        error("Entrada 'config' não encontrada.\n");

    /* entra no node 'config' */
    node = node->xmlChildrenNode;

    /* anda por todos os nodes dentro de 'config' */
    while(node) {
        /* caso o nome do node não seja 'system', vai pro próximo */
        if(xmlStrcmp(node->name, (const xmlChar *) "system")) {
            fprintf(stderr, "node inválido: %s\n", node->name);
            node = node->next;
        }

        fprintf(stdout, "system: %s\n",
                xmlGetProp(node, (const xmlChar *) "name"));

        /* anda por cada 'system' */
        sub = node->xmlChildrenNode;
        while(!xmlStrcmp(sub->name, (const xmlChar *) "kernel") && sub) {

            if((value = xmlGetProp(sub, (const xmlChar *) "version")) != NULL)
                fprintf(stdout, "version: %s\n", value);

            if((value = xmlGetProp(sub, (const xmlChar *) "bogomips")) != NULL)
                fprintf(stdout, "bogomips: %s\n", value);

            if((value = xmlGetProp(sub, (const xmlChar *) "image")) != NULL)
                fprintf(stdout, "image: %s\n", value);

            /* vai pro próximo */
            if((sub = sub->next) == NULL) break;
        }

        /* vai pro próximo */
        if((node = node->next) == NULL) break;
```

```
            else
            fprintf(stdout, "\n");
        }

        if(doc) xmlFreeDoc(doc);
        return 0;
    }
```

Executando:

```
$ ./parsexml
system: leibniz
version: 2.6.14
bogomips: 5611.93
image: /boot/bzImage-2.6.14

system: galois
version: 2.6.13
bogomips: 5192.12
image: /boot/bzImage-2.6.13
```

3.9 Manipulando *strings*

A manipulação de *strings* é sempre feita por *parsers*. Aqui teremos como exemplo um programa que faz a função de um *shell*.

O *shell* é o interpretador de comandos do sistema operacional, um programa que lê dados do *stdin* (terminal), interpreta e executa através da família de funções EXEC(3)[6].

Aqui teremos o *xshell*, um interpretador de comandos de demonstração com alguns comandos internos providos por *callback* em uma estrutura com ponteiros para funções.

O interpretador do texto digitado pelo usuário também é baseado em ponteiros, quebrando uma *string* em diversos pedaços, trocando o espaço ' ' por *'\0'* e mantendo uma matriz de ponteiros para cada argumento de maneira compatível com *execvp()*.

3.9.1 Interpretando dados digitados pelo usuário

Segue o código fonte do interpretador de comandos:

xshell.c

```
/*
 * xshell.c: interpretador de comandos
 *
 * Para compilar:
 *  cc -Wall xshell.c -o xshell
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/wait.h>

/* troca '\r' ou '\n' por '\0' */
static char *fixstr(char *str);

/* aponta cada membro da matriz para cada
```

[6] EXEC(3) - Linux Programmer's Manual

```
 * argumento do buffer utilizando espaço
 * como delimitador */
static void split(char *str, char **lines);

/* estrutura genérica com os comandos locais */
typedef struct {
    char  *cmd;
    void (*func)(char **lines);
} cmdptr;

/* comandos locais */
void builtin_cd(char **lines);
void builtin_exit(char **lines);

/* estrutura com ponteiros para comandos locais */
static cmdptr cmdlist[] = {
    { "cd",   builtin_cd   },
    { "exit", builtin_exit },
};

/* função principal */
int main()
{
    cmdptr *ptr;
    int i, is_builtin;
    char temp[4096], *lines[sizeof(temp)/2];
    const int cmdlen = sizeof(cmdlist)/sizeof(cmdlist[0]);

    /* ignora CTRL-C */
    signal(SIGINT, SIG_IGN);

    for(;;) {
        /* imprime o prompt */
        fprintf(stdout, "$ ");
        fflush(stdout);

        /* lê os comandos do usuário */
        memset(temp, 0, sizeof(temp));
        if(fgets(temp, sizeof(temp)-1, stdin) == NULL) break;

        /* ignora linhas começadas em '\r', '\n' e '#' */
        switch(*temp) {
            case '\r':
            case '\n':
            case '#':
                continue;
        }

        /* remove caracteres de controle do final da linha */
        fixstr(temp);

        /* divide a linha de comando colocando cada
         * um na matriz de ponteiros */
        split(temp, lines);

        /* checa se o comando digitado é local ou externo */
        is_builtin = 0;
        for(i = 0, ptr = cmdlist; i < cmdlen; i++, ptr++) {
            if(!strcmp(*lines, ptr->cmd)) {
                ptr->func(&lines[1]);
                is_builtin = 1;
            }
        }

        /* se for comando interno, volta ao começo do loop */
        if(is_builtin) continue;

        /* executa o comando externo */
        switch(fork()) {
            case -1:
                perror("fork");
                exit(1);
            case 0:
```

```
                execvp(*lines, lines);
                perror("exec");
                _exit(1);
            default:
                wait(NULL);
        }
    }

    return 0;
}

static char *fixstr(char *str)
{
    char *p;

    for(p = str; *p != '\0'; p++)
        if(*p == '\r' || *p == '\n') {
            *p = '\0';
            break;
        }

    return str;
}

static void split(char *str, char **lines)
{
    while(*str != '\0') {
        while(*str == ' ' && *str != '\0') *str++ = '\0';
        *lines++ = str;
        while(*str != ' ' && *str != '\0') str++;
    }

    *lines = NULL;
}

void builtin_cd(char **lines)
{
    if(chdir(*lines) == -1) {
        perror("chdir");
    }
}

void builtin_exit(char **lines)
{
    fprintf(stdout, "oooops!\n");
    exit(0);
}
```

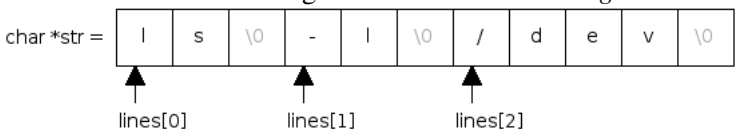
Executando:

```
$ ./xshell
xshell$ ls
argcargv.c   atexit.c    intaddr.c    ptrfunc.c   ptrstruct1.c  xshell
argchar.c    cat-mmap.c  my-atexit.c  ptrprint.c  ptrstruct2.c  xshell.c
argstruct.c  charaddr.c  ptraddr.c    ptrset.c    signal.c
xshell$ exit
oooops!
$
```

Neste código é a função *split()* quem faz a mágica na *string*. Ela simplesmente troca todos os espaços por *'\0'* e aponta cada elemento da matriz lines para o início de uma palavra.

Caso o usuário digite um comando como por exemplo *ls -l /dev*, a função *split* que recebe *temp = "ls -l /dev"* aponta *lines[0]* para *temp[0]*, *lines[1]* para *temp[3]* e *lines[2]* para *temp[6]*. Como todos os espaços se tornaram *'\0'*, cada elemento da matriz *lines* é uma *string* inteira, veja:

Figura 3.2: Divisão de *string* em matriz de *strings*



Como a função *split* tem o conteúdo de *temp* no ponteiro *str*, é a variável *temp* quem sofre a alteração. Já a matriz *lines* não tem nenhum conteúdo, apenas aponta para *temp* em lugares estratégicos fazendo com que cada um de seus membros seja uma *string* completa.

3.10 Expressões Regulares

As expressões regulares são *strings* onde alguns caracteres têm significado especial e são utilizadas normalmente para encontrar texto em outra *string*.

Imagine o administrador de sistemas que precisa encontrar o cadastro do usuário João no arquivo */etc/passwd* mas não lembra se ele foi cadastrado com letra maiúscula ou minúscula, com ou sem o acento. As expressões regulares permitem encontrar esta palavra: João, Joao, joão ou joao.

Existem dois tipos de expressões regulares: as modernas que foram definidas no padrão POSIX 1003.2 como "extendidas" e as obsoletas, definidas no 1003.2 como "básicas".

As expressões regulares são conhecidas como REs *(Regular Expressions)* e o suporte às básicas só existe para manter compatibilidade com programas antigos, portanto não serão tratadas aqui.

O comando *egrep* utiliza as expressões regulares extendidas para encontrar texto em arquivos e será utilizado em alguns exemplos.

Uma expressão regular moderna é composta por uma ou mais sessões não vazias, separadas por 'l'. Ela casa com qualquer coisa que for válida para uma das sessões, veja:

```
$ egrep "stdout|stderr" parsexml.c
        vfprintf(stderr, fmt, ap);
            fprintf(stderr, "node inválido: %s\n", node->name);
        fprintf(stdout, "system: %s\n",
                fprintf(stdout, "version: %s\n", value);
                fprintf(stdout, "bogomips: %s\n", value);
                fprintf(stdout, "image: %s\n", value);
        fprintf(stdout, "\n");
```

O comando *egrep* só imprime as linhas que casaram com a expressão regular *"stdout|stderr"*, onde *stdout* pode ser considerado uma sessão e *stderr* outra.

Alguns caracteres possuem significado especial nas expressões, como por exemplo o caracter '.'. Ele representa qualquer caracter.

A expressão regular *"Eli.a"* casa com os textos *Eliza*, *Elisa*, *Elixa*, e assim por diante, pois onde há o '.' pode haver qualquer caracter.

Também é permitido criar grupos de possíveis caracteres, utilizando '[...]'. A expressão regular *"Eli[sz]a"* casa apenas com *Elisa* ou *Eliza*.

Os grupos podem ser negativos, desde que seu primeiro caracter seja '^'. A expressão regular *"Eli[^sz]a"* casa com qualquer coisa menos *Elisa* ou *Eliza*.

No início da expressão regular, o caracter '^' representa início de linha, veja:

```
$ egrep "^s" filetype.c
static void filetype(const char *filename);
static void filetype(const char *filename)
```

Se a linha começar com espaço e o primeiro caracter for a letra 's', a expressão *"^s"* não será válida.

No final da expressão regular, o caracter '$' representa fim de linha. A expressão regular *"[^;]$"* casa com todas as linhas que não terminam em ';'.

Os caracteres e grupos podem ser quantificados, utilizando '*', '+', '?' ou '{...}'. Os quantificadores são válidos apenas para o caracter ou grupo anterior a eles.

O quantificador '*' é utilizado para casar com o caracter ou grupo anterior a ele 0 ou mais vezes. Sendo assim, a expressão regular *"cass*"* casa com as palavras *casa*, *casado*, *casamento*, *cassação* ou *cassiopéia*, pois o caracter 's' antes do '*' pode existir ou não. De fato, a famosa expressão regular *".*"* casa com qualquer coisa.

O quantificador '+' é semelhante ao '*', porém só casa com o caracter ou grupo anterior a ele 1 ou mais vezes. A expressão regular *"std.+"* casa com *stdin*, *stdout*, *stderr*, *stdio*, *stdlib*, e assim por diante. Mas esta expressão não casa com *unistd*, a menos que seja *unistd.h*.

O '?' é utilizado para fazer com que o caracter ou grupo anterior a ele exista ou não. A expressão regular *"bac?ia"* casa com *bacia* ou *baia*, e a expressão regular *"ba[ch]?ia"* casa com *bacia*, *bahia* ou *baia*.

O '{...}' é utilizado para delimitar a quantidade mínima, máxima ou intervalo de vezes que o caracter ou grupo anterior pode existir. Com a expressão regular *"[0-9]{2}"* exigimos número de 0 a 9 duas vezes. Ela casa com 10, 11, 20, 50. Já a expressão regular *"[0-9]{2,}"* casa com números de dois ou mais dígitos. Para intervalos, a expressão regular *"[0-9]{2,5}"* casa com números de dois a cinco dígitos.

O limite para as expressões regulares é sua própria imaginação. A documentação completa e oficial das expressões regulares POSIX pode ser encontrada na página de manual REGEX(7).

3.10.1 Utilizando expressões regulares

A *libc* conta com funções para o uso de expressões regulares básicas e extendidas[7]. A primeira é *regcomp()* que compila uma expressão regular e armazena suas informações em uma estrutura do tipo *struct re_pattern_buffer*, com *typedef* para o tipo *regex_t*.

A segunda função, *regexec()*, é utilizada para executar a expressão regular já compilada em uma *string* e retorna 0 quando a expressão casa com a *string* ou *REG_NOMATCH* quando não casa.

Com essas duas funções podemos utilizar o recurso de expressões regulares em nossos programas para diversos fins.

O exemplo que segue é uma versão caseira do comando *egrep* que solicta ao menos um argumento na linha de comando: uma expressão regular. Quando executado com apenas um argumento o programa lê os dados do *stdin* e compara cada linha com a expressão regular e só imprime as linhas que casam com ela. Quando executado com mais de um argumento, os argumentos dois em diante são tratados como arquivos que serão abertos e lidos, e cada linha será comparada com a expressão regular. Apenas as linhas que casam serão impressas na tela.

Depois de utilizar a expressão compilada, deve-se desalocar a memória chamando *regfree()*.

my-grep.c

```
/*
 * my-grep.c: pesquisa uma expressão regular definida pelo usuário no
 *            conteúdo de um arquivo ou no stdin
 *
 * Para compilar:
 *  cc -Wall my-grep.c -o my-grep
```

[7] REGCOMP(3) - Linux Programmer's Manual

```
 *
 * Alexandre Fiori
 */

#include <errno.h>
#include <stdio.h>
#include <regex.h>
#include <string.h>
#include <sys/types.h>

/* aplica 'regex' em 'filename' */
static void match(const regex_t *regex, const char *filename);

/* necessita ao menos um argumento */
int main(int argc, char **argv)
{
    regex_t reg;
    int r, c = argc - 2;
    char err[128], *regex, **files;

    if(argc < 2) {
        fprintf(stderr, "use: %s eregex [file1 file2...]\n", *argv);
        return 1;
    }

    regex = argv[1];
    files = &argv[2];

    /* compila a expressão regular */
    if((r = regcomp(&reg, regex, REG_EXTENDED|REG_NEWLINE|REG_NOSUB))) {
        memset(err, 0, sizeof(err));
        regerror(r, &reg, err, sizeof(err));
        regfree(&reg);
        fprintf(stderr, "regex error: %s\n", err);
        return 1;
    }

    if(c) {
        /* procura no[s] arquivo[s] */
        while(c--) match(&reg, *files++);
    } else
        /* procura no stdin */
        match(&reg, NULL);

    /* desaloca a memória da expressão compilada */
    regfree(&reg);
    return 0;
}

static void match(const regex_t *regex, const char *filename)
{
    int line = 0;
    char temp[1024];
    FILE *fp = filename ? fopen(filename, "r") : stdin;

    /* caso não seja possível abrir o arquivo... */
    if(!fp) {
        fprintf(stderr, "%s: %s\n", filename, strerror(errno));
        return;
    }

    while(!feof(fp)) {
        memset(temp, 0, sizeof(temp));
        if(fgets(temp, sizeof(temp), fp) == NULL) break;

        line++;

        /* executa a expressão regular na linha */
        if(regexec(regex, temp, 0, 0, 0) == REG_NOMATCH)
            continue;
        else
            fprintf(stdout, "%s(%d): %s",
                    filename ? filename : "-stdin-",
```

```
                    line, temp);
    }

    if(filename) fclose(fp);
}
```

Executando:

```
$ ./my-grep "^static.*" debug.c parsefile.c
debug.c(15): static void debug(const char *fmt, ...)
parsefile.c(30): static char *strfix(char *str)

$ ps -e f | ./my-grep "^.*pts/[0-9].*bash"
-stdin-(64):  3662 pts/1    Ss+    0:00      \_ -bash
-stdin-(65):  3980 pts/2    Ss     0:00      \_ -bash
-stdin-(68):  3997 pts/3    Ss     0:00      \_ -bash
```

O programa *my-grep* sempre imprime o nome do arquivo onde encontrou as linhas que casaram com a expressão regular e entre parênteses o número da linha.

3.10.2 Expressões Regulares em *parsers*

Um dos recursos mais importantes das expressões regulares são as *substrings*. Com elas, podemos selecionar uma parte da *string* e obter seus valores como variáveis. O comando *sed* é um dos que permite o uso de *substrings*.

As *substrings* são definidas por '(...)' e posteriormente cada uma é identificada por '\número'. Na expressão regular *"([0-9]+)([a-z]+)"*, a primeira *substring* é identificada por *\1* e a segunda por *\2*.

Em diversos casos na administração de sistemas o administrador precisa manipular arquivos texto para encontrar ou modificar linhas ou colunas. O *sed* provê a melhor interface para essas tarefas através do sistema de substituição de campos por *substrings*.

Imagine-se com o conteúdo abaixo:

```
$ tail -5 /etc/protocols
l2tp    115     L2TP            # Layer Two Tunneling Protocol [RFC2661]
isis    124     ISIS            # IS-IS over IPv4
sctp    132     SCTP            # Stream Control Transmission Protocol
fc      133     FC              # Fibre Channel
```

Por qualquer motivo o administrador precisa fazer com que a primeira coluna seja invertida com a segunda e o restante seja ignorado. Em arquivos de 5 ou 50 linhas é possível realizar essa tarefa manualmente mas em arquivos de 500 ou 5000 linhas é uma tarefa impraticável.

A melhor maneira é utilizar expressões regulares, veja:

```
$ tail -5 /etc/protocols | sed -r "s/^(.+)[ \t]+([0-9]+).+$/\2  \1/g"
115  l2tp
124  isis
132  sctp
133  fc
```

A opção *-r* faz com que a expressão seja tratada como extentida ao invés de básica. O comando *s/expressão1/expressão2/g* faz com que a *expressão1* seja substituída pela *expressão2*. Como na primeira foram criadas duas *substrings*, todo o conteúdo foi substituído por pela segunda e depois pela primeira.

As funções *regcomp()* e *regexec()* também permitem o uso de *substrings* - de fato, elas são utilizadas tanto no *egrep* quanto no *sed*.

Quando usamos *substrings* a função *regexec()* solicita dois parâmetros novos: um vetor do tipo *regmatch_t* e o número máximo de *substrings* possíveis.

Ao executar a expressão regular em uma *string*, o vetor do tipo *regmatch_t* terá a posição 0 com os dados da *string* inteira, a posição 1 com os dados da primeira *substring*, e assim por diante.

O tipo *regmatch_t* é uma estrutura com apenas dois membros: *rm_so* e *rm_eo*, sendo *start offset* e *end offset*. O primeiro contém o número do caracter do início da *substring* e o segundo contém o número do caracter do final da *substring*.

Embora pareça complicado, não é. O segredo é ler o código com calma e interpretar cada parte.

parsereg.c

```
/*
 * parsereg.c: executa expressões regulares em strings
 *
 * Para compilar:
 *  cc -Wall parsereg.c -o parsereg
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <regex.h>
#include <string.h>
#include <sys/types.h>

/* procura a expressão 'regex' em 'str' e retorna o resultado */
static char *match(char *str, char *regex);

int main()
{
    char temp[128], *p;

    /* cria string para primeiro teste */
    memset(temp, 0, sizeof(temp));
    snprintf(temp, sizeof(temp), "opção = valor");

    /* executa primeiro teste */
    p = match(temp, "^opção = (.*)$");
    fprintf(stdout, "p = %s\n", p);

    /* cria string para segundo teste */
    memset(temp, 0, sizeof(temp));
    snprintf(temp, sizeof(temp), "nome: bozo");

    /* executa segundo teste */
    p = match(temp, "^nome: (.*)$");
    fprintf(stdout, "p = %s\n", p);

    return 0;
}

static char *match(char *str, char *regex)
{
    int r;
    char err[128], *p;
    const int nmatch = 2;

    regex_t reg;
    regmatch_t result[nmatch];

    memset(err, 0, sizeof(err));
    memset(&result, 0, sizeof(result));

    /* compila a expressão 'regex' */
    if((r = regcomp(&reg, regex, REG_EXTENDED))) {
        regerror(r, &reg, err, sizeof(err));
        regfree(&reg);
        fprintf(stderr, "regcomp: %s\n", err);
```

```
        return NULL;
    }

    /* executa a expressão em 'temp' */
    if(regexec(&reg, str, nmatch, result, 0) == REG_NOMATCH) {
        regfree(&reg);
        return NULL;
    }

    /* aponta 'p' para o offset de início da substring */
    p = str+result[1].rm_so;

    /* coloca '\0' no final da substring */
    p[result[1].rm_eo] = '\0';

    /* retorna 'p' contendo apenas a substring */
    return p;
}
```

Executando:

```
$ ./parsereg
p = valor
p = bozo
```

3.11 Unicode

Tudo começou com o ASCII, um mapa de caracteres baseado em 7 *bits* que só previa 128 símbolos, mais que suficiente na época de seu nascimento na década de 1960. Com a evolução e popularização dos computadores esse mapa de caracteres se tornou obsoleto pelo fato de não permitir a escrita de línguas com símbolos e alfabetos diferentes.

Anos depois surgiu uma nova família de mapas de caracteres, chamada ISO-8859. Esta família foi baseada em 8 *bits*, sendo capaz de representar até 256 caracteres - entre eles alguns imprimíveis e outros não, como os caracteres de controle. Com a nova tecnologia era possível escrever texto em diversas línguas com seus próprios símbolos. Esta família ainda é largamente utilizada hoje e tem um problema crítico: não é possível escrever um único documento texto com diversas línguas. Em outras palavras, não podemos escrever no mesmo documento texto uma parte em português, outra em russo, e outra em japonês, pois os símbolos dessas línguas estão em mapas diferentes da família ISO-8859.

Com os arquivos de texto simples *(plain text)* não há uma maneira de fazer os programas detectarem qual o mapa de caracteres que foi utilizado na criação do arquivo, resultando na impressão de símbolos estranhos no meio do conteúdo. Na maioria dos casos é necessário configurar os editores de texto e informar qual o mapa de caracteres que deve ser utilizado para interpretar os arquivos.

Com as línguas Asiáticas em geral, o problema se agravou de forma que a computação teve de evoluir para permitir o uso de seus alfabetos, que normalmente não são baseados nos símbolos latinos.

A única maneira de resolver todos esses problemas era a criação de um mapa de caracteres que fosse capaz de prever todos os símbolos de todas as línguas existentes (e muitas línguas mortas) e ainda alguns símbolos utilizados na matemática e engenharia.

Seu nome é Unicode.

O Unicode é um mapa de caracter universal - *Universal Character Set (UCS)*. Ele foi desenvolvido com um único propósito: acabar com a abundância de mapas de caracteres usados para escrever texto em diferentes línguas.

Os mapas de caracteres até então eram divididos por regiões geográficas e símbolos comuns entre elas. O mapa mais utilizado no Brasil, ISO-8859-1 ou latin1, é uma definição de todos os caracteres presentes em nosso alfabeto. Porém, esse mapa não prevê caracteres russos nem japoneses.

As fontes instaladas no sistema - tanto no terminal quanto no ambiente gráfico - são sempre baseadas em um mapa de caracter, veja:

```
$ ls /usr/X11R6/lib/X11/fonts/misc/|grep ISO8859
10x20-ISO8859-11.pcf.gz
10x20-ISO8859-16.pcf.gz
10x20-ISO8859-1.pcf.gz
4x6-ISO8859-10.pcf.gz
4x6-ISO8859-13.pcf.gz
4x6-ISO8859-14.pcf.gz
4x6-ISO8859-15.pcf.gz
4x6-ISO8859-16.pcf.gz
4x6-ISO8859-1.pcf.gz
4x6-ISO8859-2.pcf.gz
...
```

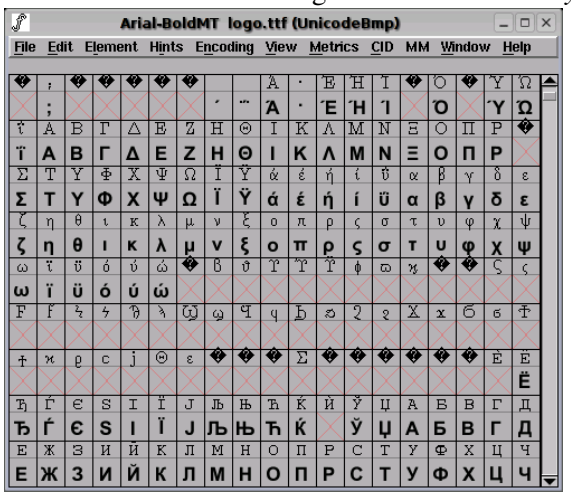
O mapa de caracteres ISO8859-10 é diferente de ISO8859-15 que é diferente de ISO8859-1. Sendo assim, cada uma dessas fontes possui símbolos diferentes em ordens diferentes, de acordo com o mapa.

Significa que para o Unicode são necessárias novas fontes, baseadas nesse novo mapa de caracteres. Uma das vantagens é que qualquer mapa de caracteres pode ser convertido para Unicode.

Duas organizações são responsáveis pelo padrão Unicode, sendo *Unicode Consortium* e *International Organization for Standardization (ISO)*. O nome Unicode também é representado por ISO/IEC 10646. Esse mapa de caracteres é mais que um conjunto de símbolos organizados em uma ordem sequencial, ele também prevê escrita bidirecional e outras particularidades de cada língua.

O Unicode é basicamente uma tabela (um vetor) relativamente grande onde cada símbolo de cada língua é representado por um número, uma posição da tabela. Hoje, a maioria das fontes *TrueType* são baseadas no Unicode, portanto podemos ver este vetor utilizando um editor de fontes. No Linux, tem um editor relativamente bom chamado *FontForge*, veja:

Figura 3.3: Fonte *TrueType* baseada em Unicode



A maneira de organizar o vetor é uma questão de codificação. Os tipos de codificação do Unicode são conhecidos como UTF-n, onde *n* é o número de *bits* utilizados para um único símbolo. UTF significa *Unicode Transformation Format*.

O Unicode muda um dos conceitos mais antigos da computação, onde um caracter é sempre representado por um *byte*. Com o sistema de codificação do Unicode os caracteres podem ser representados por mais de um *byte*, afinal o próprio mapa de caracteres possui mais de 256 símbolos.

As codificações mais comuns são UTF-8 e UTF-16, sendo este segundo utilizado nos sistemas Microsoft Windows atuais onde um caracter é sempre representado por um ou dois *bytes*. Os

sistemas *Unix-like* como o Linux utilizam o esquema UTF-8, onde um único caracter pode ser representado por um ou mais *bytes* - permitindo até 4 *bytes* para um único caracter, o tamanho de um *long int*.

Como os sistemas *Unix-like* são completamente dependentes de arquivos texto simples *(plain text)* o UTF-8 mantém a codificação do primeiro *byte* exatamente igual no ASCII, fazendo com que os arquivos de configuração dos programas sempre sejam válidos independente do mapa de caracteres. Significa que um arquivo ASCII é sempre um arquivo UTF-8 válido, sem nenhuma modificação. Quando os arquivos UTF-8 possuem mais de um *byte* para representar um único caracter, os caracteres da tabela ASCII nunca são usados no segundo, terceiro ou quarto *byte*, pois caso arquele arquivo seja interpretado como ASCII não será comprometido.

Assim, os arquivos de configuração, *shellscripts* e outros arquivos de texto simples são válidos tanto no ASCII quanto no UTF-8, e são totalmente compatíveis.

3.11.1 Implementação do Unicode

De fato, a *libc* suporta Unicode em todas as funções relacionadas a *strings*. Um novo tipo *wchar_t* deve ser utilizado no lugar de *char*. A declaração das funções desta implementação está definida em *wchar.h*.

O nome *wchar* é devido ao fato do tipo *char* ser expandido, passando a ser um *wide char*.

As funções como *strlen()* e *strcpy()* possuem suas equivalentes sendo *wcslen()* e *wcscpy()*. A função *memset()* também possui sua equivalente, *wmemset()*. As funções *printf()* e *fprintf()* possuem suas equivalentes *wprintf()* e *fwprintf()* e assim por diante.

Em suma, basta consultar rapidamente o arquivo *wchar.h* para encontrar uma função relacionada ao Unicode.

É importante passar a programar utilizando essas funções pois em poucos anos todos os sistemas serão baseados em Unicode e as famílias ISO-8859 serão obsoletas.

Projetos importantes como o GNOME[8] são totalmente compatíveis com UTF-8, independente do mapa de caracteres configurado no sistema. Internamente, todo código é baseado nas funções *wcs**.

[8] GNOME - *http://www.gnome.org*

Capítulo 4

Desenvolvimento de Projetos

Para desenvolver aplicações com qualidade o programador deve se preparar para um projeto: com arquivos de cabeçalho, arquivos de código fonte e um procedimento de compilação.

É comum que novos programadores tenham dificuldade em criar aplicações pois na maioria das vezes estão concentrados em um único arquivo *.c* onde está todo o código e o programa se torna uma grande bagunça.

Esta é, talvez, uma barreira maior que a programação em si.

Nas próximas páginas você irá encontrar diferentes maneiras de organizar projetos, aplicações separadas em diversos arquivos de cabeçalho e código fonte, além de procedimentos de compilação para otimizar o desenvolvimento.

Ferramentas, bibliotecas de uso padronizado e utilitários para internacionalização também serão apresentados.

4.1 Dividindo o projeto em arquivos

Para manter a organização do projeto, o ideal é dividir a aplicação em arquivos separados. Esta divisão é bem simples e faz com que o desenvolvimento e a manutenção sejam mais fáceis, além de tornar o programa mais elaborado.

4.2 Os arquivos de cabeçalho

Esses arquivos não são bibliotecas, estão longe disso! Os *headers*, arquivos de cabeçalho - aqueles arquivos *.h* - são utilizados pelo compilador para verificar a sintaxe das funções utilizadas e informar o programador caso uma delas tenha sido utilizada de maneira inadequada.

É perfeitamente possível criar um programa sem incluir nenhum arquivo de cabeçalho, pois nem o compilador nem a aplicação não depende deles.

Pode parecer estranho, mas é a realidade. Para que compreenda melhor este caso, deve entender o que são e como funcionam as bibliotecas.

4.2.1 Bibliotecas

As bibliotecas da linguagem C são programas não executáveis com grupos de funções para uso genérico, como manipulação de *strings*, de arquivos, de recursos do sistema operacional e outros.

Elas são arquivos binários já compilados, prontos para serem utilizados.

O fato é que para utilizar uma biblioteca é necessário conhecê-la, saber o tipo de retorno e cada argumento de cada função presente nela.

A biblioteca padrão do sistema, a *libc*, sozinha, possui mais de 500 funções. A menos que você seja um *expert*, não irá lembrar de todos os argumentos de todas as funções.

No Linux é possível consultar as funções através das páginas de manual[1], onde há toda a informação sobre o tipo de retorno e os argumentos.

Mesmo que você, programador, saiba utilizar as funções, ainda pode cometer erros durante a passagem de argumentos ou recebimento do retorno. Se isso acontecer, o compilador pode gerar um programa executável que funciona de maneira inadequada e você, programador, irá demorar horas para encontrar os erros cometidos.

Para facilitar tudo isso existem os arquivos de cabeçalho, os *headers*, aqueles arquivos *.h*. Eles são distribuídos junto às bibliotecas, que são arquivos binários. Mas os *headers* estão no formato texto, como código em linguagem C.

Os arquivos de cabeçalho distribuídos com as bibliotecas possuem uma lista do protótipo das funções disponíveis na biblioteca - isso mesmo, a declaração das funções providas pela biblioteca, com o tipo de retorno e todos os argumentos necessários para cada uma delas.

Durante o processo de compilação, a o pré-processador adiciona a seu programa todo o código contido nos arquivos incluídos com *#include*. A partir deste momento seu programa possui a declaração de todas aquelas funções, portanto se uma delas foi utilizada com parâmetros diferentes da declaração, irá resultar em um aviso ou erro de compilação.

Caso você não utilize nenhum *#include*, o compilador irá emitir um aviso sobre o uso de algumas funções não declaradas e irá gerar o programa executável normalmente.

O problema é que na hora de executá-lo, se uma das funções foi utilizada de maneira inadequada, esses valores serão passados para a biblioteca que pode gerar erros de execução, então a aplicação terá *bugs*.

Normalmente os arquivos de cabeçalho disponibilizados com as bibliotecas utilizam a palavra *extern* na declaração de suas funções, veja:

```
extern void exit(int status);
```

Esta palavra explica ao compilador que a função existe porém não está no código. Isso significa que ele deve compilar e na hora de gerar o programa executável irá encontrar esta função em alguma biblioteca.

Para gerar o programa executável, o próprio compilador utiliza um programa externo, um *linker*. O papel do *linker* é juntar todos objetos gerados pelo compilador e criar um arquivo executável dependente de algumas bibliotecas especificadas pelo programador.

O compilador *gcc* executa o comando *ld* para realizar esta tarefa, sendo ele o *linker*. Por padrão, o próprio *gcc* informa o *ld* da dependência da *libc*, portanto o programador não precisa se preocupar com ela.

Se o arquivo de cabeçalho foi incluído com *#include* e sua biblioteca não for informada ao *gcc*, ele também não informará o *ld*, resultando em erro.

lderr.c

```
/*
 * lderr.c: criptografa a string 'teste' usando crypt()
 *
 * Para compilar:
 *  cc -Wall lderr.c -o lderr -lcrypt
 *
 * Alexandre Fiori
 */
```

[1] MAN(1) - Manual pager utils

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <unistd.h>

int main()
{
     fprintf(stdout, "resultado: %s\n", crypt("teste", "A1"));

     return 0;
}
```

A função *crypt()*[2] foi utilizada como exemplo. Sua página de manual diz que para utilizar esta função primeiro deve-se definir a constante _XOPEN_SOURCE antes de incluir o arquivo de cabeçalho onde ela está declarada. Depois, para fazer o *link* deve-se informar o *gcc* do uso da biblioteca *libcrypt*.

Veja:

```
$ cc -Wall lderr.c -o lderr
/tmp/cc8l07ao.o: In function 'main':lderr.c:(.text+0x2a): undefined reference to 'crypt'
collect2: ld returned 1 exit status
```

Este tipo de erro significa que o código foi compilado perfeitamente mas o *link* falhou, pois existe uma referência indefinida ao símbolo *crypt* - ele quer dizer que a função *crypt()* não existe em nenhum objeto.

Apesar do arquivo de cabeçalho *unistd.h* possuir a declaração da função *crypt()*, seu código está em uma biblioteca que não é a *libc*, a única utilizada automaticamente pelo *gcc*.

Veja:

```
$ cc -Wall lderr.c -o lderr -lcrypt
$ ./lderr resultado: A1FrAIKkfAu6E
```

Agora o compilador *gcc* informou o *ld* do uso de uma biblioteca, chamada *libcrypt*. Embora a opção seja apenas *-lcrypt*, qualquer que seja o valor passado em *-lvalor* se torna *libvalor*.

Para verificar bibliotecas que um arquivo executável depende, utilize o comando *ldd*[3].

4.2.2 Criando arquivos de cabeçalho

A criação de arquivos de cabeçalho só é necessária no desenvolvimento de bibliotecas próprias ou de projetos com mais de um arquivo com código fonte, caso contrário são dispensáveis.

O conteúdo desses arquivos é simples, basta declarar as funções que existem em determinado arquivo *.c* que serão utilizadas em outro arquivo *.c*.

Abaixo teremos o código de um projeto dividido em quatro arquivos:

- *pr1.c*: arquivo principal, com a função *main()*
- *pr1.h*: definições gerais do projeto
- *info.c*: arquivo com a função *print_info()*
- *info.h*: declaração das funções em *info.c*

pr1.c

[2] CRYPT(3) - Library functions

[3] LDD(1) - Informa as dependências de bibliotecas dinâmicas

```
/*
 * pr1.c: demonstra o uso de arquivos de cabeçalho
 *
 * Para compilar:
 *  cc -Wall -c pr1.c  -o pr1.o
 *  cc -Wall -c info.c -o info.o
 *  cc -Wall -o pr1 pr1.o info.o
 *
 * Alexandre Fiori
 */

#include "pr1.h"

int main()
{
    /* função definida em info.c */
    print_info();

    return 0;
}
```

pr1.h

```
/*
 * pr1.h: definições gerais do projeto
 */

#ifndef _PR1_H
#define _PR1_H

/* sistema */
#include <stdio.h>

/* projeto */
#include "info.h"

/* constantes do projeto */
#define PR1_NAME "pr1"
#define PR1_VERSION "0.1b"

#endif /* pr1.h */
```

info.c

```
/*
 * info.c: código da função print_info()
 */

#include "pr1.h"

void print_info(void)
{
    fprintf(stdout, "%s %s\n", PR1_NAME, PR1_VERSION);
}
```

info.h

```
/*
 * info.h: definições das funções em info.c
 */

#ifndef _INFO_H
#define _INFO_H

extern void print_info(void);

#endif /* info.h */
```

Compilando:

```
$ cc -Wall -c pr1.c  -o pr1.o
$ cc -Wall -c info.c -o info.o
```

As duas linhas acima são utilizadas para compilar os arquivos *pr1.c* e *info.c* e gerar os objetos *pr1.o* e *info.o*, respectivamente.

A opção *-c* do *gcc* indica que os arquivos devem ser apenas compilados. Isso devido ao fato de que o *gcc* sempre executa o *linker* após compilar um arquivo. Neste caso ele não irá chamar.

Perceba que durante a compilação do arquivo *pr1.c* o *gcc* permitiu que a função *print_info()* fosse utilizada mesmo sem ter seu código presente. Isso devido ao uso da palavra *extern* em *info.h*, o qual é incluído em *pr1.h*, por sua vez incluído em *pr1.c*.

```
$ cc -Wall -o pr1 pr1.o info.o
```

A linha acima faz com que o *gcc* utilize o código já compilado dos objetos *pr1.o* e *info.o* para criar o arquivo executável *pr1*, através do *ld*.

A opção *-o* significa *output*, fazendo com que o argumento seguinte seja o nome do arquivo a ser gerado.

Agora, o *linker* irá procurar pelo símbolo da função *print_info()* para que seja possível gerar o executável *pr1*. De fato, ele irá encontrar este símbolo presente no objeto *info.o*.

Executando:

```
$ ./pr1
pr1 0.1b
```

O uso das constantes _PR1_H e _INFO_H nos arquivos *pr1.h* e *info.h* são mera convenção, são opcionais, mas são muito úteis em projetos maiores.

Quando um arquivo inclui outro, utilizando *#include*, seu código é copiado para o conteúdo corrente. Portanto, caso haja dois arquivos onde um inclui o outro, haverá duplicação de definições das funções.

Esse problema é evitado utilizando *#ifndef* e essas constantes.

4.3 Procedimento de compilação

Em projetos onde há diversos arquivos com código fonte é praticamente inviável executar o compilador diversas vezes a cada modificação de código durante o desenvolvimento.

Os procedimentos de compilação otimizam o desenvolvimento fazendo com que o programador se concentre no código e não perca tempo com a compilação.

Quando o projeto cresce e mais bibliotecas externas são utilizadas elas podem simplesmente ser incluídas no procedimento de compilação.

4.3.1 O arquivo *Makefile*

O arquivo *Makefile* é uma espécie de *script* onde há uma rotina de procedimentos definida pelo programador. Este arquivo pode ser utilizado para executar qualquer sequência de comandos e não é vinculado à linguagem C.

Ele é utilizado pelo programa *make*[4], um utilitário GNU que executa suas sessões.

[4] MAKE(1) - GNU make utility

4.3.2 Definição de sessões

Cada sessão de um *Makefile* é definida com um ou mais comandos, veja:

Makefile.sample

```
#
# Makefile.sample: procedimentos apenas para demonstração
#
# Para utilizar:
#  make -f Makefile.sample teste
#
# Alexandre Fiori
#

teste:
        clear
        ls -l

sessao1:
        date

sessao2:
        uname -a
```

Executando:

```
$ make -f Makefile.sample sessao1
date
Sun Nov 20 19:43:52 BRST 2005

$ make -f Makefile.sample sessao10
make: *** No rule to make target 'sessao10'.  Stop.
```

Ao chamar a sessão 'sessao1', o comando *make* utilizou o *Makefile.sample* para executar o comando definido lá, *date*.

Ao chamar a sessão 'sessao10', o comando *make* retornou erro informando que não existe esta sessão no *Makefile*.

Caso o comando *make* seja executado sem a opção *-f*, irá procurar por um arquivo com nome de *Makefile* no diretório atual.

Se nenhuma sessão for especificada, irá utilizar a primeira encontrada em *Makefile*.

4.3.3 Criação manual de *Makefile*

O projeto *pr1* definido na Sessão 4.2.2 necessita de um *Makefile*. Lá, existem dois arquivos com código fonte para compilar e depois executar o *linker* para gerar o arquivo executável.

Segue o arquivo:

Makefile

```
#
# Makefile: procedimento de compilação para pr1
#
# Para utilizar:
#  make
#
# Alexandre Fiori
#

# comandos do sistema
```

```
CC = cc
RM = rm -f

# variáveis do procedimento
CFLAGS = -Wall
LIBS = # -lsample

# nome do projeto
NAME = pr1

# objetos do projeto
OBJECTS = pr1.o info.o

# procedimento de compilação
.SUFFIXES: .c
.c.o:
        $(CC) $(CFLAGS) -c -o $@ $<

# sessão principal, dependente de $(OBJECTS) e $(NAME)
all: $(OBJECTS) $(NAME)

$(NAME):
        $(CC) $(CFLAGS) -o $(NAME) $(OBJECTS) $(LIBS)

clean:
        $(RM) $(OBJECTS) $(NAME)
```

Utilizando o procedimento de compilação:

```
$ make
cc -Wall -c -o pr1.o pr1.c
cc -Wall -c -o info.o info.c
cc -Wall -o pr1 pr1.o info.o

$ make clean
rm -f pr1.o info.o pr1
```

Também é possível executar mais de uma sessão do *Makefile* com uma única chamada ao comando *make*, veja:

```
$ make clean all
rm -f pr1.o info.o pr1
cc -Wall -c -o pr1.o pr1.c
cc -Wall -c -o info.o info.c
cc -Wall -o pr1 pr1.o info.o
```

Que tal?

4.3.4 Dica do editor *vim*

O editor de texto *vim* está preparado para lidar com a compilação de projetos com procedimento baseado em *Makefile*.

Dentro do editor, o comando *:make* irá executar o comando *make* e caso encontre erros na compilação, o editor irá diretamente para a o arquivo e linha onde está o primeiro erro encontrado.

Casa haja mais de um erro, para avançar para o próximo utilize *:cn* e para voltar ao anterior utilize *:cp*.

Essas opções significam *C Next* e *C Previous*, respectivamente.

O editor também suporta argumentos para o procedimento, como por exemplo *:make clean all*.

4.3.5 Ferramentas GNU

O kit de desenvolvimento GNU[5] conta com ferramentas especializadas na criação do procedimento de compilação de projetos.

Em outras palavras, essas ferramentas geram o *Makefile* automagicamente.

A vantagem de utilizar essas ferramentas é que no conceito de desenvolvimento da GNU estão alguns pré-requisitos importantes, como a portabilidade do código.

Além disso, o *script* que gera o *Makefile*, chamado *configure*, irá procurar todas as ferramentas, arquivos de cabeçalho e bibliotecas necessárias para a compilação. Caso alguma delas não seja encontrada, o *script* informa o usuário sobre o problema e uma possível solução.

Isso faz com que os usuários que baixam *software* livre da *Internet* possam compilar os programas sem incomodar os autores dos projetos com dúvidas ou problemas de compilação.

4.3.5.1 A ferramenta *m4*

Utilizada para processar *macros m4*. Esta ferramenta normalmente não é utilizada de modo direto, porém todas as outras apresentadas aqui dependem dela para funcionar.

4.3.5.2 A ferramenta *aclocal*

Utilizada para definir propriedades da plataforma - sistema operacional, versão, etc - e gerar o arquivo *aclocal.m4* à partir do arquivo *configure.ac*.

4.3.5.3 A ferramenta *automake*

Utilizada para gerar o arquivo *Makefile.in* à partir dos arquivos *configure.ac* e *Makefile.am*.

O arquivo *aclocal.m4* deve existir, bem como os arquivos *AUTHORS*, *README*, *NEWS* e *ChangeLog*.

4.3.5.4 A ferramenta *autoconf*

Utilizada para gerar o arquivo *configure* à partir do arquivo *configure.ac*.

O *script* gerado por ela depende do arquivo *Makefile.in*.

4.3.5.5 Os arquivos *AUTHORS*, *README*, *NEWS* e *ChangeLog*

Para utilizar a ferramenta *automake* esses arquivos devem existir no diretório do projeto. Este pré-requisito é devido ao conceito de *software* livre da GNU onde os programas precisam ser documentados para que sejam distribuídos na *Internet*.

O arquivo *AUTHORS* diz respeiro aos autores do projeto. Basta criá-lo e lá dentro escrever os nomes dos autores.

O arquivo *README* deve conter informações gerais sobre o projeto.

O arquivo *NEWS* deve trazer novidades entre uma versão e outra.

O arquivo *ChangeLog* deve possuir as modificações significativas das etapas de desenvolvimento do projeto - um histórico.

[5] GNU - http://www.gnu.org

4.3.5.6 O arquivo *configure.ac*

Este arquivo é utilizado por *aclocal*, *automake* e *autoconf*. Seu conteúdo é composto por *macros m4* que indicam quais arquivos de cabeçalho e bibliotecas fazem parte da dependência do projeto para que o *script configure* gerado por *autoconf* esteja preparado para verificar a existência antes de permitir a compilação.

4.3.5.7 O arquivo *Makefile.am*

Este arquivo é utilizado por *automake* para gerar *Makefile.in* que posteriormente será utilizado pelo *script configure* para criar o *Makefile* do projeto.

Seu conteúdo indica quais programas executáveis devem ser gerados e qual o código fonte de cada um deles.

Também é possível vincular parâmetros para a compilação, como por exemplo o uso de bibliotecas específicas para cada programa.

4.3.6 Exemplo

Este exemplo é baseado no código fonte de *pr1*, aprentado na Sessão 4.2.2. A única diferença é que seu nome foi modificado para *pr2* e portanto as constantes PR1_H, PR1_NAME e PR1_VERSION foram alteradas para PR2_H, PR2_NAME e PR2_VERSION respectivamente.

pr2.c

```
/*
 * pr2.c: demonstra o uso de GNU AutoTools
 *
 * Para compilar:
 *  ./configure
 *  make
 *
 * Alexandre Fiori
 */

#include "pr2.h"

int main()
{
    /* função definida em info.c */
    print_info();

    return 0;
}
```

pr2.h

```
/*
 * pr2.h: definições gerais da aplicação
 */

#ifndef _PR2_H
#define _PR2_H

/* sistema */
#include <stdio.h>

/* projeto */
#include "info.h"
```

```
/* constantes do projeto */
#define PR2_NAME "pr2"
#define PR2_VERSION "0.1b"

#endif /* pr2.h */
```

info.c

```
/*
 * info.c: código da função print_info()
 */

#include "pr2.h"

void print_info(void)
{
    fprintf(stdout, "%s %s\n", PR2_NAME, PR2_VERSION);
}
```

info.h

```
/*
 * info.h: definições das funções em info.c
 */

#ifndef _INFO_H
#define _INFO_H

extern void print_info(void);

#endif /* info.h */
```

AUTHORS, ***README***, ***NEWS*** e ***ChangeLog***

Podem ser gerados com *touch*, serão arquivos em branco:

```
$ touch AUTHORS README NEWS ChangeLog
```

Makefile.am

```
#
# Makefile.am: arquivo que deve ser processado com automake para criar
#              Makefile.in automaticamente
#
# Para utilizar:
#  automake
#
# Alexandre Fiori
#

bin_PROGRAMS = pr2
pr2_SOURCES  = pr2.c info.c
#pr2_LDFLAGS = -rdynamic
#pr2_LDADD   = -L/usr/local/lib -ltest

noinst_HEADERS = pr2.h info.h
```

configure.ac

```
#
# configure.ac: utilizado por aclocal, automake e autoconf
#               para gerar o script configure
#
# Para utilizar:
#  automake
#
# Alexandre Fiori
#

AC_INIT(pr2.h)
#AC_ARG_PROGRAM
#AC_PROG_INSTALL
AM_INIT_AUTOMAKE(pr2, 0.1)

dnl Checa o hostname e outras informações
AC_CANONICAL_HOST

dnl Checa alguns programas
AC_PROG_CC
AC_PROG_INSTALL
AC_PROG_MAKE_SET
#AC_CHECK_TOOL(AR, ar)
#AC_CHECK_TOOL(RANLIB, ranlib, :)

#dnl Inicializa libtool
#AM_PROG_LIBTOOL

dnl Checa os arquivos de cabeçalho
AC_STDC_HEADERS

dnl Gera Makefile
AC_OUTPUT(Makefile)
```

Processo de criação do *script configure*

Com todos esses arquivos no mesmo diretório é possível utilizar as ferramentas GNU para gerar o *script*, veja:

```
$ ls
AUTHORS    Makefile.am  README        info.c  pr2.c
ChangeLog  NEWS         configure.ac  info.h  pr2.h

$ aclocal

$ automake -a
configure.ac: installing `./install-sh'
configure.ac: installing `./missing'
configure.ac:17: installing `./config.guess'
configure.ac:17: installing `./config.sub'
Makefile.am: installing `./INSTALL'
Makefile.am: installing `./COPYING'
Makefile.am: installing `./depcomp'

$ autoconf

$ ls
AUTHORS    Makefile.am  aclocal.m4      configure     info.h      pr2.h
COPYING    Makefile.in  autom4te.cache  configure.ac  install-sh
ChangeLog  NEWS         config.guess    depcomp       missing
INSTALL    README       config.sub      info.c        pr2.c
```

Processo de compilação

O arquivo *Makefile* ainda não existe. Contudo, o *script* encarregado de criá-lo, *configure*, precisa ser executado pois irá checar se o sistema possui tudo que é necessário para compilar a aplicação.

Esta é função das ferramentas GNU, além de tornar a aplicação portável para diversas plataformas desde que o código esteja adaptado.

Veja:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for a BSD-compatible install... /usr/bin/install -c
checking whether make sets $(MAKE)... (cached) yes
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for ANSI C header files... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands

$ make
...
gcc  -g -O2   -o pr2  pr2.o info.o

$ ./pr2
pr2 0.1b
```

Este *Makefile* gerado pelo *script* também é capaz de instalar a aplicação no sistema. Basta executar o comando *make install.*

4.4 Funções e bibliotecas importantes

Durante o desenvolvimento de um projeto, sua aplicação deve ser semelhante às ferramentas do sistema.

Para isso, algumas funções e bibliotecas devem ser utilizadas.

4.4.1 Utilizando *getopt*

O *getopt* é utilizado para interpretar argumentos da linha de comando. Sua implementação está dividida em algumas funções providas pela *libc* e na maioria dos sistemas há também um comando *getopt* que pode ser utilizando em *shellscripts*.

Sua utilização é simples e muito eficiente, pois já conta com as diversas maneiras de interpretar argumentos de uma maneira genérica que vale para qualquer tipo de aplicação.

4.4.1.1 O programa *getopt*

O programa *getopt*[6] é baseado nas funções da *libc* portanto seu modo de funcionamento é muito semelhante nos *scripts*, veja:

getopt.sh

[6]GETOPT(1) - Interpretador de opções de comandos

```
#!/bin/sh
#
# getopt.sh: utiliza o programa getopt para interpretar os argumentos
#            da linha de comando
#
# Para utilizar:
#  ./getopt.sh
#
# Alexandre Fiori
#

help()
{
cat << EOF
getopt.sh v0.1
use: $0 [OPÇÃO]

OPÇÕES:
 -d             imprime data e hora do sistema
 -c             imprime características da CPU
 -h             imprime esta tela
 -u             imprime versão do sistema operacional
 -e ARQUIVO     abre o editor de textos padrão em ARQUIVO
 -l ARQUIVO     mostra informações de ARQUIVO [via stat]
EOF
}

# caso não haja argumento na linha de comando, imprime o help
[ ! "$1" ] && help

# caso haja, interpreta cada um
while getopts "e:l:dchu" OPT; do
    case $OPT in
    d)
        date
        ;;
    c)
        cat /proc/cpuinfo
        ;;
    u)
        uname -a
        ;;
    e)
        [ ! "$EDITOR" ] && EDITOR=vi
        $EDITOR $OPTARG
        ;;
    l)
        ls -l $OPTARG
        ;;
    *)
        help
    esac
done
shift $[OPTIND-1]
```

Executando:

```
$ ./getopt.sh -du
Sun Nov 20 22:34:34 BRST 2005
Linux leibniz 2.6.14.2 #1 SMP Fri Nov 11 22:54:43 BRST 2005 i686 GNU/Linux

$ ./getopt.sh -e
./getopt.sh: option requires an argument -- e
getopt.sh v0.1
use: ./getopt.sh [OPÇÃO]

OPÇÕES:
 -d             imprime data e hora do sistema
 -c             imprime características da CPU
 -h             imprime esta tela
 -u             imprime versão do sistema operacional
 -e ARQUIVO     abre o editor de textos padrão em ARQUIVO
 -l ARQUIVO     mostra informações de ARQUIVO [ls -l]
```

Agora este *script* atua como a grande maioria das ferramentas disponíveis no sistema. Note que em *'while getopts "e:l:dchu" OPT; do'* apenas as letras acompanhadas por *:* necessitam de argumento adicional para funcionar.

Executar *getopt.sh -du* é exatamente a mesma coisa que executar *getopt.sh -d -u*.

4.4.1.2 A função *getopt()*

Na *libc* existem três funções *getopt()*[7]. A primeira, propriamente dita, é utilizada em casos onde o programador precisa de uma aplicação que receba argumentos de maneira semelhante ao *script* apresentado na sessão anterior.

A segunda é *getopt_long()*, que permite o uso de opções como *–help*, *–info*, e assim por diante. Ela também é compatível com *getopt()* e suporta opções do tipo *-h* e *-i*.

A terceira é *getopt_long_only()*, que só suporta as opções extendidas.

O exemplo é baseado na segunda opção, por ser a mais conveniente. De fato, o conteúdo de cada opção não será implementado para encurtar o programa.

getopt-long.c

```
/*
 * getopt-long.c: demonstra o uso de getopt_long() para interpretar
 *                argumentos da linha de comando
 *
 * Para compilar:
 *  cc -Wall getopt-long.c -o getopt-long
 *
 * Alexandre Fiori
 */

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int main(int argc, char **argv)
{
    int c, optind = 0;
    const char *opts = "e:l:dchu";
    const struct option optl[] = {
        { "edit",     1, 0, 'e' },
        { "list",     1, 0, 'l' },
        { "date",     0, 0, 'd' },
        { "cpu-info", 0, 0, 'c' },
        { "help",     0, 0, 'h' },
        { "uname",    0, 0, 'u' },
        { 0, 0, 0, 0 },
    };

    void help() {
        fprintf(stderr,
                "getopt-long v0.1\n"
                "use: %s [OPÇÃO]\n\n"
                "OPÇÕES:\n"
                "-d, --date       Imprime data e hora do sistema\n"
                "-c, --cpu-info   Imprime características da CPU\n"
                "-h, --help       Imprime esta tela\n"
                "-u, --uname      Imprime versão do sistema operacional\n"
                "-e, --edit=ARQ   Edita arquivo ARQ\n"
                "-l, --list=ARQ   Lista arquivo ARQ\n", *argv);
        exit(1);
    }

    if(argc == 1) help();
```

[7] GETOPT(3) - Linux Programmer's Manual

```
    while((c = getopt_long(argc, argv, opts, optl, &optind)) != -1)
        switch(c) {
            case 'e':
                fprintf(stdout, "edita arquivo %s.\n", optarg);
                break;
            case 'l':
                fprintf(stdout, "lista arquivo %s.\n", optarg);
                break;
            case 'd':
                fprintf(stdout, "imprime a data.\n");
                break;
            case 'c':
                fprintf(stdout, "imprime dados da CPU.\n");
                break;
            case 'u':
                fprintf(stdout, "imprime dados do sistema operacional.\n");
                break;
            default:
                help();
        }

    return 0;
}
```

Executando:

```
$ ./getopt-long --date --uname -e bla
imprime a data.
imprime dados do sistema operacional.
edita arquivo bla.
```

Agora suas aplicações poderão seguir os padrões do sistema para lidar com argumentos.

4.4.2 Utilizando *gettext*

O *gettext*[8] faz parte do projeto de internacionalização do sistema e é utilizado para traduzir mensagens.

Na implementação da *libc*, a função *gettext()*[9] é utilizada para traduzir *strings* de texto para a língua nativa baseando-se em um catálogo de mensagens.

Para entender o *gettext* é necessário conhecer o *locale*.

4.4.2.1 Sistema de localização *locale*

O *locale*[10] é um programa que manipula algumas variáveis do sistema relacionadas a regionalização e internacionalização da *libc*.

O valor dessas variáveis diz respeito à linguagem do sistema, sistema monetário, tipo de papel padrão, unidade de medida, etc.

Para que o *locale* funcione é necessário habilitar as linguagens desejadas. O sistema pode ter catálogos de uma, algumas ou todas as linguagens disponíveis.

Veja:

```
$ locale -a
C
POSIX
es_ES
```

[8] GETTEXT(1) - Traduz mensagens

[9] GETTEXT(3) - Linux Programmer's Manual

[10] LOCALE(1) - Manipula o sistema de regionalização

```
es_ES.iso88591
it_IT
it_IT.iso88591
italian
pt_BR
pt_BR.iso88591
spanish
```

Essas são as linguagens habilitadas no sistema, não as disponíveis. Para obter uma lista de todas as disponíveis veja o arquivo */usr/share/i18n/SUPPORTED*.

Para adicionar mais linguagens ao *locale* basta adicionar uma nova linha no arquivo */etc/locale.gen* e executar o comando *locale-gen*[11].

Depois de ter a linguagem habilitada, basta alterar o valor das variáveis de ambiente LC_* para alterar o comportamento dos programas que utilizam o *locale*.

4.4.2.2 Implementando *gettext* nas aplicações

Se sua aplicação for desenvolvida utilizando o *gettext* poderá funcionar em várias línguas diferentes sem que haja necessidade de recompilar o código. Para isso, é necessário preparar todas as *strings* e gerar um catálogo de mensagens com elas.

A preparação das *strings* pode ser feita de maneira simples, utilizando uma *macro* do pré-processador.

Normalmente a implementação é feita em etapas, onde o programador define uma *macro* que será utilizada futuramente pelo sistema de internacionalização mas não se preocupa com ela e mantém toda a concentração no desenvolvimento da aplicação.

i18n.c

```
/*
 * i18n.c: define a macro para uso futuro do gettext
 *
 * Para compilar:
 *  cc -Wall i18n.c -o i18n
 *
 * Alexandre Fiori
 */

#include <stdio.h>

/* utilizado no desenvolvimento da aplicação */
#define _(str) (str)

int main()
{
    fprintf(stdout, _("Hello world!\n"));
    return 0;
}
```

Executando:

```
$ ./i18n
Hello world!
```

Com esta *macro* todas as *strings* serão representadas por seu próprio conteúdo. É como se ela não existisse, não serve pra nada. Mas durante o desenvolvimento da aplicação o programador irá utilizá-la em todas as *strings*, assim: *_("texto")* ao invés de simplesmente *"texto"*.

Depois da aplicação pronta, testada e funcionando, deve-se gerar o catálogo de mensagens para tradução, como segue:

[11] O *locale-gen* é um *shellscript* e não está presente em todas as distribuições. Caso não exista em seu sistema consulte o manual LOCALEDEF(1).

```
$ xgettext -o i18n-base.po -k_ i18n.c

$ cat i18n-base.po
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2005-11-21 21:44-0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: i18n.c:17
#, c-format
msgid "Hello world!\n"
msgstr ""
```

O programa *xgettext* é utilizado para capturar as *strings* baseadas na *macro* especificada em *-k* e gerar o arquivo especificado em *-o*.

Portanto, o arquivo *i18n-base.po* é nossa base para tradução. Os campos *msgid* não devem ser alterados pois é através deles que *gettext* irá encontrar a mensagem original para ser traduzida. O campo *msgstr* deve ser preenchido com a tradução desejada.

Estrutura de diretórios do *gettext*

O *gettext* utiliza domínios *(domains)*. Para ele, um domínio é o nome dado ao catálogo da aplicação. Portanto, é comum definirmos o domínio com o próprio nome da aplicação.

Quando uma aplicação é traduzida, os arquivos *.po* devem ser compilados utilizando a ferramenta *msgfmt* que será apresentada a seguir, que gera catálogos de mensagens binários com extensão *.mo (message object)*.

Posteriormente, esses catálogos devem ser instalados no sistema - normalmente são instalados junto com a aplicação - para que estejam disponíveis ao usuário que deseja utilizar a aplicação na língua nativa.

O local para encontrar esses objetos já instalados pode ser definido no código pela função *bindtextdomain()* que será explicada a seguir.

O diretório padrão na grande maioria dos sistemas é */usr/share/locale*.

Lá dentro, a estrutura é a seguinte:

```
língua/locale/domain.mo
```

Exemplo:

```
/usr/share/locale/pt_BR/LC_MESSAGES/intl.mo
```

Para nossa aplicação, o local com os catálogos será definido como *./locale*, ou seja, o diretório atual. Então, a estrutura será:

```
./locale/pt_BR/LC_MESSAGES/intl.mo
```

Caso seja traduzido para espanhol:

```
./locale/es/LC_MESSAGES/intl.mo
```

Traduzindo e compilando os catálogos

Agora o arquivo *i18n-base.po* será ser copiado para *i18n-pt_BR.po*, *i18n-es.po* e *i18n-it.po*, a estrutura de diretórios local será criada, todos serão editados, traduzidos e depois compilados.

Veja:

```
$ mkdir -p ./locale/pt_BR/LC_MESSAGES
$ cp i18n-base.po i18n-pt_BR.po
'i18n-base.po' -> 'i18n-pt_BR.po'

$ mkdir -p ./locale/es/LC_MESSAGES
$ cp i18n-base.po i18n-es.po
'i18n-base.po' -> 'i18n-es.po'

$ mkdir -p ./locale/it/LC_MESSAGES
$ cp i18n-base.po i18n-it.po
'i18n-base.po' -> 'i18n-it.po'

/* tradução de todos, um por um, manualmente */
$ vi i18n-pt_BR.po i18n-es.po i18n-it.po

/* compila os catálogos */
$ msgfmt i18n-pt_BR.po -o ./locale/pt_BR/LC_MESSAGES/i18n.mo
$ msgfmt i18n-es.po -o ./locale/es/LC_MESSAGES/i18n.mo
$ msgfmt i18n-it.po -o ./locale/it/LC_MESSAGES/i18n.mo
```

Agora já temos os catálogos traduzidos e instalados no local correto, voltamos às funções...

4.4.2.3 A função *setlocale()*

Utilizada para definir o *locale* da aplicação.

4.4.2.4 A função *bindtextdomain()*

Utilizada para definir o local com a estrutura do sistema de tradução para determinado domínio.

4.4.2.5 A função *textdomain()*

Utilizada para definir o domínio que será acessado pelas chamadas a *gettext()*.

4.4.2.6 A função *gettext()*

Pesquisa os catálogos de determinado domínio e retorna a *string* de acordo com a configuração do *locale*, presente nas variáveis de ambiente LC_*.

Em nosso exemplo, utilizamos LC_MESSAGES. A variável LANG é principal, e se for alterada irá alterar automaticamente o valor de todas as outras.

4.4.2.7 Código fonte internacionalizado

O programa abaixo é a evolução do código apresentado na Sessão 4.4.2.2, porém com a *macro* alterada e as funções do *gettext* já implementadas.

i18n.c

```
/*
 * i18n.c: programa internacionalizado!
 *
 * Para compilar:
 *  cc -Wall i18n.c -o i18n -lintl
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <locale.h>
#include <libintl.h>

/* chamada a gettext() */
#define _(str) gettext(str)

int main()
{
    /* informa que o domínio 'i18n' pode ser
     * encontrado em ./locale */
    bindtextdomain("i18n", "./locale");

    /* informa que gettext() deve utilizar
     * o domínio 'i18n' */
    textdomain("i18n");

    /* definir o locale usando "" significa
     * utilizar o valor das variáveis de ambiente */
    setlocale(LC_ALL, "");
    setlocale(LC_MESSAGES, "");

    /* imprime a mensagem utilizando a macro _(str) que
     * é a chamada a gettext(str) */
    fprintf(stdout, _("Hello world!\n"));
    return 0;
}
```

Executando:

```
$ ./i18n
Hello world!

$ export LANG=pt_BR
$ ./i18n
Olá mundo!

$ export LANG=es_ES
$ ./i18n
Hola mundo!

$ export LANG=it_IT
$ ./i18n
Ciao mondo!
```

Agora, parceiro, basta programar! Não esqueça de preparar um *Makefile* para automatizar tudo isso. Consulte a documentação do *automake* e *autoconf* pois eles possuem procedimentos para compilar e instalar os arquivos do *gettext*.

4.5 Criando bibliotecas

No início, quando comecei a programar, não entendia muito bem como essa coisa de biblioteca funcionava. Lembro que no *clipper* era necessário chamar o *linker* e informar sobre o uso de *clipper.lib*, a biblioteca dele.

Qualquer programa executável, por mais simples que fosse, não cabia em um disquete pois tinha sempre ~2MB.

Era o ano de 1993, meu aniversário de 13 anos estava chegando e havia pedido um livro de *clipper* para minha mãe como presente - era melhor que meias, pijama, essas coisas. Já estava de olho nele a meses, então escrevi todos os dados em um pedaço de papel e entreguei a ela.

No dia do aniversário, chegou ela com o presente. Pediu desculpas por ter perdido o papel e disse: *Acho que era esse... pelo menos o nome é parecido*.

Pensei: *Pronto! Comprou o livro errado!*

Quando abri o pacote, não entendi direito. O título era: *Integrando Clipper 5.2 com C, de William Pereira Alves - editora Ética*.

Eu não fazia a menor idéia do que era aquilo. Integrar uma linguagem com a letra C? C de *Clipper*??? Sei lá.

Meses depois, já tinha entendido que era uma linguagem relacionada a um tal de Unix, que era poderosa, etc e tal.

Consegui com um amigo, o Eduardo, uma cópia do Turbo C 2.0.

Instalei no DOS e copiei todos os exemplos do livro, todos! Comecei a gostar daquele Turbo C, ainda porque o livro explicava como criar bibliotecas em C para usar no *clipper*. Podia habilitar o mouse, fazer miséria.

Então percebi o seguinte: quando criava uma biblioteca não utilizava a famosa função *main()* do C. A biblioteca era um arquivo *.lib* que o *linker* utilizava para gerar o executável, e para isso copiava seu conteúdo dentro do executável. Os programas ficaram maiores ainda!!!

Meses depois abandonei o *clipper*, tive que aprender inglês pra ler o manual do Turbo C e estou aqui, até hoje, nesse tal de C - mas agora com *vi* e *gcc*!

O fato é que as bibliotecas, essas que são copiadas dentro do binário executável, são chamadas de bibliotecas estáticas. São compostas por conjuntos de funções agrupadas em objetos pré compilados, que por sua vez são agrupados em um único arquivo e indexados de alguma maneira.

Ou seja: é uma coleção de arquivos *.o* dentro de um arquivo (hoje) *.a*.

Com a evolução dos sistemas operacionais, as bibliotecas estáticas perderam campo e surgiram as bibliotecas dinâmicas, ou bibliotecas compartilhadas - que são arquivos *.so* presentes no sistema.

Vamos conhecê-las.

4.5.1 Bibliotecas estáticas

As bibliotecas estáticas são compostas por um ou mais objetos, arquivos pré compilados e agrupados em um único arquivo.

Atualmente essas bibliotecas são utilizadas apenas no escopo do próprio projeto e não são disponibilizadas no sistema. O programador organiza as funções que são de uso genérico para a aplicação e cria um biblioteca para separar o código genérico do código específico. É muito comum encontrar esse tipo de biblioteca em projetos que geram mais de um arquivo executável com as mesmas características, como por exemplo a aplicação *iproute*.

Como exemplo, o *iproute* é composto por vários arquivos, entre eles o executável *ip* e o executável *tc*. Esses dois programas fazem parte do mesmo projeto, porém com funcionalidades diferentes. Internamente, no código, utilizam algumas funções em comum. Então, para facilitar, o programador criou uma biblioteca com as funções genéricas e as utilizou nos dois programas. Quando são compilados, o código da biblioteca é copiado para dentro do executável e boas. A biblioteca não precisa existir no sistema pois já faz parte de cada executável.

4.5.1.1 Criando bibliotecas estáticas

A criação dessas bibliotecas é resultado da compilação de alguns arquivos de código fonte sem a presença de *main()* e o agrupamento dos objetos deve ser feito pelo programa *ar*. Depois, o

arquivo deve ser indexado utilizando *ranlib*.

O projeto de demonstração é uma biblioteca chamada *slib (simple lib)* que provê funções para controlar um *stack* interno.

Junto, há um programa chamado *app* que utiliza as funções da biblioteca.

No procedimento de compilação, o *Makefile*, iremos primeiro compilar e criar a biblioteca e posteriormente compilar o programa, utilizando a biblioteca.

app.c

```
/*
 * app.c: programa que utiliza a biblioteca slib
 *
 * Para compilar:
 *  make
 *
 * Alexandre Fiori
 */

/* sistema */
#include <stdio.h>

/* projeto */
#include "slib.h"

int main()
{
    slib_put(3);
    slib_put(5);
    slib_put(7);
    slib_put(9);

    fprintf(stdout, "valor1: %d\n", slib_get());
    fprintf(stdout, "valor2: %d\n", slib_get());
    fprintf(stdout, "valor3: %d\n", slib_get());
    fprintf(stdout, "valor4: %d\n", slib_get());

    return 0;
}
```

slib.c

```
/*
 * slib.c: biblioteca de demonstração
 */

#include <stdio.h>
#include <string.h>

/* utilizado para guardar valores inteiros */
static int slib_stack[3];
static int slib_stack_pos = 0;
const  int slib_stack_size = sizeof(slib_stack)/sizeof(slib_stack[0]);

/* inicializa a biblioteca */
void slib_init(void)
{
    /* limpa a área de memória */
    memset(slib_stack, 0, sizeof(slib_stack));
}

/* adiciona um valor ao stack */
int slib_put(int in)
{
    if(slib_stack_pos == slib_stack_size) return -1;
    else
        slib_stack[slib_stack_pos++] = in;
```

```
    return in;
}

/* retira um valor do stack */
int slib_get()
{
    if(!slib_stack_pos) return -1;

    return slib_stack[--slib_stack_pos];
}

/* finaliza a biblioteca */
void slib_quit(void)
{
    /* limpa a área de memória */
    slib_init();
}
```

slib.h

```
/*
 * slib.h: biblioteca de demonstração
 */

#ifndef _SLIB_H
#define _SLIB_H

extern void slib_init(void);
extern int  slib_put(int in);
extern int  slib_get();
extern void slib_quit(void);

#endif /* slib.h */
```

Makefile

```
#
# Makefile: procedimento de compilação para slib
#
# Para utilizar:
#  make
#
# Alexandre Fiori
#

# comandos do sistema
CC = cc
RM = rm -f
AR = ar rc
RANLIB = ranlib

# variáveis do procedimento
CFLAGS = -Wall
LIBS = -L./ -lslib

# nome do projeto
NAME = app
SLIB = libslib.a

# objetos do projeto
LIBOBJECTS = slib.o
APPOBJECTS = app.o

# procedimento de compilação
.SUFFIXES: .c
.c.o:
        $(CC) $(CFLAGS) -c -o $@ $<
```

```
# sessão principal
all: $(NAME)

$(NAME): $(SLIB) $(APPOBJECTS)
        $(CC) $(CFLAGS) -o $(NAME) $(APPOBJECTS) $(LIBS)

$(SLIB): $(LIBOBJECTS)
        $(AR) $(SLIB) $(LIBOBJECTS)
        $(RANLIB) $(SLIB)

clean:
        $(RM) $(NAME) $(SLIB) $(APPOBJECTS) $(LIBOBJECTS)
```

Compilando e executando

O processo de compilação está preparado no *Makefile*, portando basta executar *make* e ter a aplicação pronta, veja:

```
$ make
cc -Wall -c -o slib.o slib.c
ar rc libslib.a slib.o
ranlib libslib.a
cc -Wall -c -o app.o app.c
cc -Wall -o app app.o -L./ -lslib

$ ./app
valor1: 7
valor2: 5
valor3: 3
valor4: -1
```

As opções utilizadas em *ar rc* informam que o arquivo *libslib.a* deve ser criado com o conteúdo seguinte, no caso, *slib.o*. Normalmente as bibliotecas são compostas por mais de um arquivo objeto, mas neste exemplo apenas um foi utilizado.

No arquivo *app.c*, experimente remover as quatro funções *fprintf()* e adicionar apenas uma, esta:

```
fprintf(stdout, "valores: %d %d %d %d\n",
        slib_get(), slib_get(), slib_get(), slib_get());
```

Como o compilador processa o código fonte da direita para esquerda, o programa binário executável sofre as consequências.

Veja o resultado:

```
$ ./app valores: -1 3 5 7
```

Travou[12]?

Ok. Isso acontece devido à maneira como o compilador interpreta o código. Se prestar atenção no incremento e decremento de variáveis notará que quando fazemos *var++* o resultado só é computado na próxima operação.

Por outro lado, quando fazemos *++var* ele é computado imediatamente.

Se interpretarmos o código da direita para esquerda, podemos nos deparar com um sinal de incremento ou decremento mas ainda não temos a variável onde esta operação deve ser executada, então deixamos pra próxima. Mas, quando encontramos uma variável e depois um incremento ou decremento, podemos realizar a operação imediatamente.

[12] Operação conhecida como *brain lock*

4.5.2 Bibliotecas dinâmicas

Com o passar do tempo as bibliotecas estáticas foram sendo deixadas de lado. Isso devido ao fato de que os programas executáveis eram muito grandes e na maioria das vezes boa parte de seu código binário era proveniente das mesmas bibliotecas - como por exemplo a *libc*, que estava em todos. Ao invés de copiar os objetos das bibliotecas estáticas dentro do código binário dos programas, um sistema mais evoluído foi adotado: as bibliotecas dinâmicas.

Também são conhecidas como DSO *(Dynamic Shared Object)* ou *Shared Library*, por ser um objeto dinâmico compartilhado. Isso significa que o *linker* não copia o conteúdo da biblioteca dentro do binário executável, ele simplesmente cria uma referência a ela.

Como grande parte dos programas usa funções genéricas da *libc*, por exemplo, a *libc* passou a ser uma biblioteca dinâmica. Com isso, todo o código com as funções genéricas ficou concentrado nela. Os programas executáveis ficaram menores e agora dependem de ter a *libc* instalada no sistema.

A vantagem desse sistema é que, além dos programas executáveis ficarem muito menores ficaram mais rápidos no momento da execução. Outro fato importante é que quando um programa é executado ele carrega o conteúdo da biblioteca na memória, e quando esse conteúdo está lá pode ser usado por outros programas! Tudo isso é feito automaticamente pelo sistema operacional através de MMAP(2)[13].

4.5.2.1 Criando bibliotecas dinâmicas

O procedimento para a criação da biblioteca dinâmica é exatamente o mesmo da biblioteca estática. Um arquivo ou mais arquivos de cabeçalho devem ser criados e também o código fonte. A diferença está na compilação.

Este exemplo utilizará o mesmo código fonte do anterior, na Sessão 4.5.1.1, mas o *Makefile* é diferente.

Makefile

```
#
# Makefile: procedimento de compilação para slib
#
# Para utilizar:
#  make
#
# Alexandre Fiori
#

# comandos do sistema
CC = cc
RM = rm -f

# variáveis do procedimento
CFLAGS = -Wall
LIBS = -L./ -lslib

# nome do projeto
NAME = app
SLIB = libslib.so

# objetos do projeto
LIBOBJECTS = slib.o
APPOBJECTS = app.o

# procedimento de compilação
.SUFFIXES: .c
```

[13] MMAP(2) - Linux Programmer's Manual

```
.c.o:
        $(CC) $(CFLAGS) -c -o $@ $<

# sessão principal
all: $(NAME)

$(NAME): $(SLIB) $(APPOBJECTS)
        $(CC) $(CFLAGS) -o $(NAME) $(APPOBJECTS) $(LIBS)

$(SLIB): $(LIBOBJECTS)
        $(CC) $(CFLAGS) -shared -Wl,-soname -Wl,$(SLIB) \
                -o $(SLIB) $(LIBOBJECTS)

clean:
        $(RM) $(NAME) $(SLIB) $(APPOBJECTS) $(LIBOBJECTS)
```

Compilando:

```
$ make
cc -Wall -c -o slib.o slib.c
cc -Wall -shared -Wl,-soname -Wl,libslib.so \
       -o libslib.so slib.o
cc -Wall -c -o app.o app.c
cc -Wall -o app app.o -L./ -lslib
```

Ao invés de agrupar os objetos com *ar* e indexar a biblioteca com *ranlib*, informamos ao *ld* por meio do *gcc* que o objeto a ser criado é do tipo *shared*, portanto não necessita uma função *main()*.

Agora, o arquivo *libslib.so* foi criado e o programa *app* foi compilado utilizando a biblioteca dinâmica, portanto depende dela para executar.

Veja:

```
$ ldd ./app
        linux-gate.so.1 =>  (0xffffe000)
        libslib.so => not found
        libc.so.6 => /lib/tls/libc.so.6 (0xb7e9c000)
        /lib/ld-linux.so.2 (0xb7fe3000)
```

A biblioteca *libslib.so* não foi encontrada. Sem ela, não podemos executar o programa, veja:

```
$ ./app
./app: error while loading shared libraries: libslib.so: cannot open
shared object file: No such file or directory
```

Isso acontece devido ao fato de que as bibliotecas dinâmicas devem estar presentes em específicos no sistema, como */lib* e */usr/lib*. Caso estejam em outro diretório, o administrador deve especificar seu caminho utilizando o arquivo */etc/ld.so.conf* e sempre fizer alteração deve executar o comando *ldconfig*[14]. O comando *ldconfig* escaneia os diretórios com bibliotecas dinâmicas e cria um índice em */etc/ld.so.cache* - arquivo binário que não deve ser manipulado diretamente.

A segunda maneira de informar o sistema sobre novos diretórios com bibliotecas dinâmicas é utilizar a variávem de ambiente LD_LIBRARY_PATH.

Então, se copiarmos o arquivo *libslib.so* para */lib* ou */usr/lib* o programa *app* irá funcionar. Se definirmos o diretório atual na variável LD_LIBRARY_PATH ele também irá funcionar - porém, apenas nesta sessão pois quando o terminal for fechado e aberto novamente esta variável não irá existir.

A terceira maneira é adicionar o diretório com o arquivo *libslib.so* na configuração do *ld*, em */etc/ld.so.conf*.

Exemplo:

[14] LDCONFIG(8) - Configure dynamic linker run-time bindings

```
$ export LD_LIBRARY_PATH=./
$ ./app
valor1: 7
valor2: 5
valor3: 3
valor4: -1
```

Para obter uma lista de todas as bibliotecas dinâmicas instaladas e válidas no sistema pode-se executar o comando *ldconfig –print-cache*.

4.6 Carregando bibliotecas manualmente

Prepare-se para uma longa e interessante história.

Quando descobri que as bibliotecas dinâmicas *(DSO)* podiam ser carregadas manualmente, foi uma loucura. Era o ano de 1999, já trabalhava como programador em uma empresa de automação criando aplicações com imagens digitais - usando Linux, claro.

Na época a aplicação era voltada para a captura e processamento de imagens digitais em tempo real, então era necessário criar mecanismos que possibilitavam o processamento das imagens e que hora faziam uma coisa, hora faziam outra.

As bibliotecas dinâmicas carregadas manualmente caíram como uma luva. Eu podia criar diversas bibliotecas com funções diferentes e carregar cada uma delas manualmente quando fosse necessário, sem informar ao *linker* que o programa executável dependia delas.

Imagine abrir um arquivo, ler seu conteúdo, utilizar e depois fechar. É exatamente isso, mas com bibliotecas. É possível abrir uma biblioteca *.so*, utilizar suas funções e depois fechá-la, manualmente. Como se fosse um *plug-in* para o programa.

A aplicação apresentada aqui irá manipular imagens utilizando bibliotecas dinâmicas, portanto é necessário entender o que é uma imagem digital.

4.6.1 Imagens digitais

As imagens digitais são muito populares hoje. Todo mundo conhece arquivos *jpeg*, *png* e muitos outros disponíveis na *Internet*. Todos esses arquivos são comprimidos, utilizando um algoritmo de compressão de imagem.

Mas, o que é comprimido? Exceto os vetores, todas as outras imagens digitais são criadas à partir de uma imagem pura, conhecida como *raw image*. Existem diversos padrões para imagens *raw*, sendo o mais comum entre eles o RGB.

As imagens RGB podem ser comprimidas para *jpeg*, *png*, *mpeg*, etc. O processo inverso também é possível, pois se descomprimirmos um *jpeg* teremos uma imagem RGB.

Embora existam diversos tipos de RGB, utilizaremos o mais simples e mais comum: RGB24.

O nome RGB24 é devido à quantidade de *bits* utilizados para criar um *pixel*, um ponto. Essas imagens são pesadas pelo fato de não serem comprimidas então dependendo da resolução uma única imagem pode chegar a vários *megabytes*.

Imagine uma imagem pequena, de 320x240 pontos no formato RGB24. Qual o tamanho em *bytes* desta imagem? Basta multiplicar o X por Y e os dois pelo tamanho de cada *pixel*. Se são 24 *bits*, estamos falando em 3 *bytes*, um para R*(red)*, um para G*(green)* e outro para B*(blue)*.

```
320 * 240 * 3 = 230.400 (225K)
```

Agora imagine uma imagem do tamanho da tela, por exemplo 1024x768.

```
1024 * 768 * 3 = 2.359.296 (2.3MB)
```

Cada 8 *bits* da imagem define um tom de cor, sendo R, G e B, e a cada 24 bits da imagem temos um *pixel*.

Primeiro: como fazemos para carregar essa imagem pela linguagem C?

1. Podemos alocar memória suficiente para ela via *malloc()* e ler o arquivo inteiro com *read()* ou *fread()*
2. Podemos simplesmente mapear o arquivo na memória via *mmap()*

Segundo: como interpretamos os *pixels*, se não temos um tipo de variável de 24 *bits*?

1. Podemos ler a imagem de 8 em 8 *bits*, usando *char pixel*
2. Podemos utilizar uma estrutura ou união com 3 variáveis do tipo *char* dentro

Em ambas questões ficamos com a opção número 2.

Veja o *pixel*:

```
typedef struct {
    char r;
    char g;
    char b;
} Pixel;
```

Acabamos de criar o tipo *Pixel*, que tem exatamente 3 *bytes*. Um ponteiro *Pixel* pode apontar para a memória retornada por *mmap()* e cada incremento faz com que tenhamos acesso ao próximo *pixel* da imagem, pois ele anda de 3 em 3!

Que tal?

4.6.2 O formato *portable anymap*

As imagens RGB24 normalmente são precedidas por um cabeçalho, para que possam ser identificadas pelos programas. O formato *portable anymap* é um arquivo com extensão *.ppm* ou *.pnm* e seu conteúdo é o seguinte:

```
P6
# possível comentário
320 240
255
(imagem no formato rgb24)
```

Na verdade ele é apenas um cabeçalho, onde os valores 320 e 240 são relativos à resolução da imagem anexada no arquivo. Programas como *gimp*[15] são capazes de ler e gravar esse formato.

Portanto, é necessário levar em consideração o cabeçalho antes de manipular a imagem.

4.6.3 Escrevendo a biblioteca que trata imagens

Nossa biblioteca será bem simples. Sua função é interpretar uma imagem RGB24 e criar o negativo da imagem.

Para isso, utilizaremos a operação lógica *NOT*. O *NOT* inverte todos os *bits* de uma variável, trocando 0 por 1 vice-versa.

rgb24-negative.c

[15] GNU Image Manipulation Program - http://www.gimp.org

```
/*
 * rgb24-negative.c: biblioteca que lê uma imagem RGB24 e retorna
 *                   seu negativo
 *
 * Para compilar:
 *  cc -shared -Wall -Wl,-soname -Wl,rgb24-negative.so \
 *     -o rgb24-negative.so rgb24-negative.c
 *
 * Alexandre Fiori
 */

/* define o tipo pixel */
typedef struct {
    unsigned char r;
    unsigned char g;
    unsigned char b;
} Pixel;

/* define a função que aplica o efeito */
void effect(void *image, int w, int h, int header_offset)
{
    char oldpix;
    int i, size = w * h;
    Pixel *pix = image + header_offset;

    for(i = 0; i < size; i++, pix++) {
        /* troca o R */
        oldpix = pix->r;
        pix->r = ~oldpix;

        /* troca o G */
        oldpix = pix->g;
        pix->g = ~oldpix;

        /* troca o B */
        oldpix = pix->b;
        pix->b = ~oldpix;
    }
}
```

Pronto. Esse código irá se tornar uma biblioteca dinâmica e a função *effect()* trata as imagens. O ponteiro do tipo *void* é utilizado para receber a imagem, enquanto as variáveis *w* e *h* recebem o tamanho da imagem.

O argumento *header_offset* contém o número de *bytes* do cabeçalho, quando atribuímos *pix = image + header_offset* estamos fazendo com que o ponteiro já avance esse bytes, e *pix* aponta direto para o primeiro *pixel* da imagem RGB24.

4.6.4 O programa que carrega as bibliotecas

Este programa sempre deverá receber dois argumentos na linha de comando: o nome de um *plug-in* e o nome de um arquivo *portable anymap*.

O *plug-in.so* será carregado usando de *dlopen()*[16]. Esta função permite carregar um *DSO* e mapear suas funções através de ponteiros.

Nosso procedimento é simples: recebemos o nome do arquivo na linha de comando, abrimos e depois tratamos o cabeçalho. Se for um arquivo válido, uma imagem RGB24 com cabeçalho *portable anymap*, carregamos o *plug-in DSO* e procuramos nele uma função chamada *effect()*. Caso ela exista, mapeamos a imagem na memória com *mmap()* e passamos o ponteiro para *effect()*.

Depois, descarregamos a biblioteca com *dlclose()*, descarregamos o arquivo com *munmap()* e terminamos o programa.

ppmrun.c

[16] DLOPEN(3) - Linux Programmer's Manual

```
/*
 * ppmrun.c: processa arquivos RGB24 por meio de plug-ins
 *
 * Para compilar:
 *  cc -Wall ppmrun.c -o ppmrun -ldl
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <fcntl.h>
#include <dlfcn.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>

#define RGB 3 /* bytes */

/* checa o cabeçalho da imagem */
static int ppm_header(int fd, int *w, int *h, int *off);

/* necessita ao menos dois argumentos */
int main(int argc, char **argv)
{
    void *plugin, *image;
    void (*effect)(void *image, int w, int h, int header_offset);
    int fd, w, h, offset, size;

    if(argc != 3) {
        fprintf(stderr, "use: %s plug-in.so arquivo.ppm\n", *argv);
        return 1;
    }

    /* abrimos a imagem (leitura e gravação) */
    if((fd = open(argv[2], O_RDWR)) == -1) {
        perror("open");
        return 1;
    }

    /* checa se o arquivo é realmente portable anymap */
    if(!ppm_header(fd, &w, &h, &offset)) {
        fprintf(stderr, "Arquivo %s inválido.\n", argv[2]);
        close(fd);
        return 1;
    } else
        size = w * h * RGB;

    /* carrega o plug-in */
    plugin = dlopen(argv[1], RTLD_NOW);
    if(!plugin) {
        fprintf(stderr, "dlopen: %s\n", dlerror());
        close(fd);
        return 1;
    }

    /* carrega a função */
    effect = dlsym(plugin, "effect");
    if(!effect) {
        fprintf(stderr, "dlsyn: %s\n", dlerror());
        dlclose(plugin);
        close(fd);
        return 1;
    }

    /* agora temos a função effect do plugin como
     * ponteiro aqui... portanto temos que mapear
     * a imagem e passar para ela */
    image = mmap(0, size, PROT_READ|PROT_WRITE,
            MAP_SHARED|MAP_LOCKED, fd, 0);
    if(image == MAP_FAILED) {
        perror("mmap");
```

```
            dlclose(plugin);
            close(fd);
            return 1;
        }

        /* processa e altera a imagem */
        effect(image, w, h, offset);

        /* termina o programa */
        munmap(image, size);
        dlclose(plugin);
        close(fd);
        return 0;
    }

    static int ppm_header(int fd, int *w, int *h, int *off)
    {
        char temp[128];
        struct stat st;
        FILE *fp = fdopen(fd, "r");
        int offset, check = 0, my_w = 0, my_h = 0;

        if(!fp) {
            perror("fdopen");
            return 0;
        } else
            /* obtem os dados do arquivo */
            fstat(fd, &st);

        while(!feof(fp)) {
            /* lê cada linha do arquivo */
            memset(temp, 0, sizeof(temp));
            if(fgets(temp, sizeof(temp), fp) == NULL) break;

            /* a primeira linha PRECISA ser identificada por P6 */
            if(strncmp(temp, "P6", 2) && !check) {
                fprintf(stderr, "DOH!: %s\n", temp);
                break;
            } else
                check = 1;

            /* ignora comentários... */
            if(*temp == '#')
                continue;
            else
            /* se encontrar 255 termina o loop */
            if(!strncmp(temp, "255", 3))
                break;
            else
            /* sempre tenta identificar a resolução */
                sscanf(temp, "%d %d\n", &my_w, &my_h);
        }

        /* se não conseguiu a resolução, retorna erro */
        if(!my_w || !my_h) return 0;

        /* realiza o checksum: compara o tamanho do arquivo
         * sem o cabeçalho com a resolução multiplicada por 3 */
        offset = ftell(fp);
        if(st.st_size - offset != my_w * my_h * RGB) {
            fprintf(stdout, "checksum falhou!\n");
            return 0;
        }

        /* se o arquivo está OK, grava a resolução em w e h */
        *w = my_w;
        *h = my_h;
        *off = offset;

        return 1;
    }
```

Bem, este é o programa.

Agora, é necessário ter uma imagem *portable anymap* em qualquer resolução. Caso não tenha, converta um *jpeg* ou qualquer outro.
Segue a imagem original que irá ser modificada:

Figura 4.1: Imagem *portable anymap* original

Agora iremos compilar e executar o programa.

```
$ cc -shared -Wall -Wl,-soname -Wl,rgb24-negative.so -o rgb24-negative.so rgb24-negative.c
$ cc -Wall ppmrun.c -o ppmrun -ldl

$ ./ppmrun
use: ./ppmrun plug-in.so arquivo.ppm

$ ./ppmrun ./rgb24-negative.so bacuri.ppm
```

O arquivo *bacuri.ppm* tem a mesma quantidade de *bytes*, mas seu conteúdo foi alterado. A imagem foi manipulada e agora temos lá o negativo dela.
Veja:

Figura 4.2: Imagem *portable anymap* processada

O mais interessante é que se o programa for executado novamente, a imagem voltará a seu estado original. Como houve apenas uma inversão dos *bits*, nenhum dado foi perdido.
Agora sinta-se a vontade para criar outros *plug-ins* seguindo a mesma estrutura de *rgb24-negative.c*, e nem precisará mexer no código de *ppmrun.c* nem recompilá-lo.
Vale a pena estudar a *libjpeg*[17], que permite comprimir e descomprimir RGB24 para *jpeg*.

4.7 Registrando eventos no *syslog*

O *syslog* é o serviço que registra eventos dos sistemas *Unix-like*. Boa parte das aplicações utilizam-no para gravar mensagens de diversos tipos ao invés de criar um próprio esquema de *log*.

[17] Independent JPEG Group - http://www.ijg.org/

A vantagem de usar o *syslog* é que ele é um serviço específico para registro de eventos e atua como um concentrador de *logs*. Ainda permite configurar a maneira de lidar com as mensagens enviadas pelos programas, como o local da gravação dos *logs* em disco ou em outro *syslog* via rede.

O local onde as mensagens são gravas depende da configuração deste serviço, que normalmente é feita através do arquivo */etc/syslog.conf*. Lá, o administrador do sistema separa as mensagens por *facility* e *priority(level)* e direciona cada uma para um arquivo.

Segue um exemplo da configuração do *syslog* padrão do Debian GNU/Linux[18]:

syslog.conf

```
#  /etc/syslog.conf     Configuration file for syslogd.
#
#                       For more information see syslog.conf(5)
#                       manpage.

#
# First some standard logfiles.  Log by facility.
#

auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslog
#cron.*                         /var/log/cron.log
daemon.*                        -/var/log/daemon.log
kern.*                          -/var/log/kern.log
lpr.*                           -/var/log/lpr.log
mail.*                          -/var/log/mail.log
user.*                          -/var/log/user.log
uucp.*                          /var/log/uucp.log

#
# Logging for the mail system.  Split it up so that
# it is easy to write scripts to parse these files.
#
mail.info                       -/var/log/mail.info
mail.warn                       -/var/log/mail.warn
mail.err                        /var/log/mail.err

# Logging for INN news system
#
news.crit                       /var/log/news/news.crit
news.err                        /var/log/news/news.err
news.notice                     -/var/log/news/news.notice

#
# Some 'catch-all' logfiles.
#
*.=debug;\
        auth,authpriv.none;\
        news.none;mail.none     -/var/log/debug
*.=info;*.=notice;*.=warn;\
        auth,authpriv.none;\
        cron,daemon.none;\
        mail,news.none          -/var/log/messages

#
# Emergencies are sent to everybody logged in.
#
*.emerg                         *

daemon.*;mail.*;\
        news.crit;news.err;news.notice;\
        *.=debug;*.=info;\
        *.=notice;*.=warn       |/dev/xconsole
```

É muito importante conhecer a configuração do *syslog* antes de utilizá-lo em sua aplicação.

[18] SYSLOG.CONF(5) - Linux System Administration

Quando o serviço *syslog* está ativo, ele cria um *socket* do tipo *Unix Domain Socket* para permitir que as aplicações se conectem no *daemon* e enviem suas próprias mensagens. Normalmente não há controle de acesso e qualquer programa pode enviar mensagens lá.

Veja:

```
$ sudo netstat -nxp | grep syslog
unix 7   [ ]     DGRAM       3119   2542/syslogd    /dev/log

$ ls -l /dev/log
srw-rw-rw-  1 root root 0 2005-11-25 13:33 /dev/log
```

É através do arquivo */dev/log* que a conexão entre os programas e o *daemon* é feita.

Para que seu programa se comunique com o *syslog* deve-se utilizar a família de funções do *syslog* da *libc*[19]. A função *openlog()* se conecta ao *daemon*. Depois, para enviar mensagens, deve-se utilizar *syslog()* que funciona de maneira semelhante a *fprintf()* e para finalizar a comunicação deve-se utilizar *closelog()*.

Exemplo:

syslog.c

```
/*
 * syslog.c: registra eventos no syslog
 *
 * Para compilar:
 *  cc -Wall syslog.c -o syslog
 *
 * Alexandre Fiori
 */

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

int main()
{
    openlog("meuprograma", LOG_PID, LOG_USER);

    syslog(LOG_INFO, "registrando eventos no syslog [%d]...\n", getuid());

    closelog();

    return 0;
}
```

Executando:

```
$ ./syslog
$ tail -1 /var/log/user.log
Nov 25 14:52:18 localhost meuprograma[3671]: registrando eventos no syslog [1000]...
```

Não esqueça de verificar a configuração do *syslog* pois o *facility* LOG_USER e *level* LOG_INFO podem estar configurados para gravar os registros em outro arquivo, ao invés de */var/log/user.log*.

[19] SYSLOG(3) - Linux Programmer's Manual

Capítulo 5

Redes Interconectadas

Hoje há diversas linguagens de programação com suporte a *sockets*. Através de APIs simplificadas e básicas o programador pode criar aplicações que trocam informações pela rede de maneira muito rápida.

Qual o grau de dificuldade de lidar com APIs de baixo nível? Depende da maneira como esta API for explicada, e da capacidade de compreendimento do programador.

Aqui, veremos os *sockets* no seu segundo nível. Segundo porque temos que considerar o mais baixo nível que está no sistema operacional, então o nível onde trabalharemos está acima - a *libc*.

Porém, para que seja possível lidar com *sockets* neste nível é extremamente necessário conhecer os princípios do protocolo IP, seu sistema de roteamento, algumas características dos protocolos TCP, UDP e ICMP, e assim por diante.

De passo em passo esses detalhes serão desmistificados com o objetivo de clarear a sua mente. Depois de aberta, as informações cairão nela como uma luva.

5.1 Origem das redes de computadores

A alguns anos atrás numa leitura de primeira, encontrei uma história genial[1], a qual decidi transcrever aqui.

Cerca de 30 anos atrás cientistas inventavam a rede de computador, conhecida como *ethernet*. A origem do termo vem de *ether* (em português, atmosfera), onde os segmentos de dados - conhecidos como pacotes - eram transmitidos quando o protocolo foi inventado.

Nesta época pessoas com diversos computadores nas ilhas do Hawaii queriam se comunicar entre si. O sistema de telefonia entre as ilhas não era confiável para suportar a comunicação *dial-up* (linha discada), então essas pessoas passaram a utilizar um *link* de rádio frequência (RF) e criaram um novo protocolo de comunicação designado a operar nesta nova topologia, chamado de *ALOHA System*.

Cada computador enviava seus dados como um pacote, uma *string* (sequência) de *bytes* com um cabeçalho informativo que incluía o endereço do remetente, destinatário e quantidade de *bytes* de dados enviados - sendo este último número utilizado para verificar se os dados que chegaram possuem a mesma quantidade de *bytes* que foram enviados. Este sistema hoje é conhecido como *checksum*.

Um pacote corrompido resultaria então no que conhecemos como colisão de rede, dois ou mais pacotes transmitidos ao mesmo tempo.

[1] Linux Cluster Architecture, Alex Vrenios - p. 23-26.

Este novo protocolo era tão simples e tão poderoso que muitos cientistas da computação se interessaram e começaram a pensar na possibilidade desta nova tecnologia de comunicação entre computadores ser usada largamente.

Um rádio *transceiver* (transmissor e receptor) é normalmente conectado a uma antena através de uma linha de transmissão, um cabo coaxial preparado para operar entre a saída do *transceiver* e entrada da antena - com o mínimo de perda de sinal. O rádio normalmente fica em uma sala, em cima de uma mesa, enquanto a antena muitas vezes fica em lugares mais altos como telhados e torres. A linha de transmissão torna esta separação possível.

No experimento original do *ALOHA System* cada nó da rede tinha um sistema com seu próprio *transceiver*, linha de transmissão e antena anexados ao computador e programas específicos para controlar o fluxo dos pacotes - envio e recebimento.

Foi aí que então um novo grupo de cientistas decidiu eliminar as antenas conectando todos os *transceivers* por um único cabo coaxial. Quando a distância entre os computadores não implicaria em um problema, o mesmo protocolo funcionaria para uma *Local Area Network* (ou *LAN*, rede local de computadores). Nascia o *ethernet*.

Os cabos coaxiais se tornaram mais finos e baratos e os mesmos conceitos funcionavam bem em equipamentos muito mais simples e baratos que os primeiros *transceivers* do experimento. Ainda eram cabos coaxiais, mas tão baratos quanto os cabos de energia elétrica.

Muitos edifícios comerciais já tinham em sua estrutura kilômetros de cabos da rede de telefonia interna, normalmente muitos pares a mais que o necessário para todos os telefones. Cada par, trançado entre si, era necessário para uma única linha de voz - então a expressão par trançado.

Para aplicar o novo sistema de rede de computadores alguém precisava conseguir utilizar esta infra-estrutura já presente em muitos locais. Não era difícil desenhar uma placa de acesso à rede que utilizasse um par de cabos.

Não demorou muito e já funcionava muito bem entre dois computadores, um *link* ponto a ponto. O problema é que com aquelas ferramentas parecia ser impossível criar uma rede onde um computador poderia se comunicar com qualquer outro da rede através daqueles cabos - seriam inúmeras conexões entre pares e se tornaria uma bagunça. Para resolver este problema foi criado um dispositivo concentrador, posteriormente chamado de *hub*.

Um *hub* é um equipamento pequeno e eletrônicamente simples que possui um número fixo de portas de comunicação, uma para cada nó da rede. Com este equipamento as redes passaram a ter uma topologia tipo estrela, onde o concentrador fica no meio e os nós nas pontas. Todos os nós conectados apenas no concentrador, capaz de permitir comunicação direta entre cada um.

O sistema eletrônico do *hub* simplesmente retransmite os sinais que recebe em uma porta para todas as outras. Levando em consideração as características do protocolo *ethernet*, cada computador que enviasse um pacote para determinado destinatário teria seu pacote entregue a todos os demais, porém só o nó com aquele endereço é que deveria responder.

Um sistema de convenção de nomes foi adotado. Cada cabo coaxial original era capaz de conectar dois pontos a uma distância de até 500 metros operando primeiro a 1, depois a 2, e finalmente a 10 *megabits* por segundo (Mbps). Foram nomeados como 1BASE-5, 2BASE-5 e 10BASE-5, respectivamente.

O primeiro número representa a velocidade do *link*, a base refere-se à banda de comunicação, onde apenas uma frequência é utilizada e o número final representa a distância aproximada do cabo, em centenas de metros.

Os conectores coaxiais são conhecidos como conectores UHF por serem preparados para o intervalo de frequências UHF (300Mhz ou mais) antes de chegarem perto de uma perda significativa de sinal. Técnicamente esses conectores são chamados de PL-259 (sendo PL de *plug*).

O termo par trançado vem do sistema de telefonia - em qualquer casa comum há quatro cabos do sistema telefônico e conectores RJ-11. Estes serviram de modelo para o sistema mais novo, com

oito cabos e conectores RJ-45.

É comum utilizarmos redes 10BASE-T, onde o T vem de *twisted pair* (par trançado). Há também 100BASE-T indicando rede 100Mbps e 1000BASE-T, indicando rede *gigabit*.

5.2 Equipamentos da infra-estrutura

Existem diversos tipos de equipamentos com funções importantes nas redes de computadores. Entre eles estão os *hubs*, cuja função já foi detalhada na Sessão 5.1.

5.2.1 O *switch*

Existem também os *hubs* inteligentes, conhecidos como *switches*.

Os *switches* são equipamentos que realizam as mesmas funções de um *hub*, com um número fixo de portas para comunicação numa topologia tipo estrela. Porém, possuem certo grau de inteligência.

Nos dias atuais com o grande crescimento do uso das redes de computadores é comum a necessidade de segurança nos dados. Um equipamento como o *hub* envia os dados de um computador para todos os demais, esperando uma única resposta do nó que possuir o endereço contido no cabeçalho do pacote.

Contudo, se um computador recebe todos os pacotes da rede e responde apenas alguns, não precisa nem ser Brasileiro para pensar em capturar todos e responder apenas os corretos.

Para eliminar esta deficiência de infra-estrutura deve-se utilizar *switch* ao invés de *hub*. Os *switches* são capazes de armazenar internamente o endereço *ethernet* de cada nó conectado a cada uma de suas portas[2]. Sendo assim, ao invés de replicar em todas as portas os pacotes recebidos, ele os envia apenas à porta cujo endereço de destino se encontra.

5.2.2 O roteador

Ele é quem permite a conexão entre duas ou mais redes lógicas. Em ambientes modernos utilizamos um conjunto de protocolos que operam sob a camada *ethernet*, conjunto conhecido como TCP/IP.

Em redes TCP/IP os computadores/nós possuem determinados endereços, onde cada intervalo de endereços pertence a uma rede lógica - Sessão 5.6. Os roteadores conectam essas redes lógicas através de rotas.

Detalhes sobre o sistema de roteamento serão explicados na Sessão 5.7.

5.2.3 O filtro de pacotes, *firewall*

Um *firewall* normalmente é composto por um ou mais computadores, onde há um conjunto de regras que permitem ou não a passagem de pacotes de acordo com critérios definidos pelo administrador da rede ou analista de segurança.

A função de um *firewall* é evitar que pacotes forjados cheguem aos servidores de dados, ou a alguns serviços disponibilizados por esses servidores. Na sua função de filtro de pacotes muitos trabalham apenas com o cabeçalho dos pacotes, baseando suas regras em endereço de origem e destino, protocolo e outros campos contidos lá.

O *firewall* normalmente não analisa o conteúdo dos pacotes, sendo muitas vezes incapaz de rejeitar vírus ou programas indevíduos.

[2]Função conhecida como *ARP cache*, disponível na maioria dos sistemas operacionais.

5.2.4 O *access point*

Este é o nome dado aos centralizadores de comunicação de redes sem fio *(wireless networks)*. Ele opera como um *hub* ou *switch* para essas redes e ainda pode ser conectado a um *hub* ou *switch* comum permitindo que os nós da rede sem fio façam parte da rede cabeada.

Muitos possuem as funções de roteador e *firewall* embutidas, facilitando no uso caseiro.

As redes sem fio caseiras atuais estão dividias em dois padrões[3], sendo seus dois pilares: 802.11b e 802.11g.

O 802.11b opera a 2.4Ghz a uma velocidade até 11Mbps. Já o 802.11g opera a 5.7Ghz a uma velocidade até 54Mbps e é compatível com o padrão mais antigo, 802.11b.

Este tipo de rede tem duas topologias: estrela ou ponto a ponto.

Na topologia tipo estrela, é necessário um *Access Point* e todos os nós operam no modo *Managed* - sendo gerenciáveis pelo centralizador. Já no ambiente ponto a ponto cada nó opera no modo *Ad-Hoc*, permitindo apenas dois nós.

5.3 Sistema Operacional

O Linux é um poderoso ambiente de aprendizado e produção, com todas as ferramentas adequadas para o bom compreendimento das coisas como elas são.

Muitos administradores de rede e programadores de hoje são apenas usuários avançados das ferramentas do sistema, e é exatamente esta barreira que devemos quebrar.

5.3.1 Características

Hoje um sistema GNU/Linux pode operar como qualquer um dos equipamentos mencionados na Sessão 5.2. Há *software* disponível gratuitamente e com o código aberto, pronto para ser utilizado pelos administradores de rede e lido pelos programadores.

Um bom administrador de rede deve conhecer pelo menos um pouco de programação, o que facilitará muito seu trabalho.

5.3.2 Ambiente

O ambiente de programação de rede do Linux é dividido em duas partes:

- *Kernel space*
  Lá estão as chamadas de sistema, conhecidas como *syscalls*. No *kernel* há todo o código que faz desde uma placa de rede funcionar até a implementação dos protocolos.
  As chamadas de sistema são funções do *kernel* disponibilizadas para os programadores, para que estes criem suas aplicações utilizando os recursos disponíveis no sistema operacional.
  Os programadores de sistema operacional trabalham neste ambiente, criando módulos *(drivers)* e disponibilizando chamadas de sistema para os programadores de *user space*.

- *User space*
  Este é o ambiente provido basicamente pela *libc*, biblioteca padrão. A *libc* possui diversas funções que ajudam no desenvolvimento das aplicações e muitas delas internamente fazem uso das chamadas de sistema.
  Os programadores de aplicações trabalham aqui, fazendo uso das chamadas de sistema e funções da *libc*.

[3] Definições do 802.11 - http://grouper.ieee.org/groups/802/11/

Além de muito rico, o ambiente de programação conta com o menor número possível de camadas de acesso ao *hardware*, sendo apenas as duas mencionadas acima. Significa que um programa acessa os dispositivos de maneira muito simples e rápida.

5.4 *OSI - Open System Interconnection*[4]

As redes lógicas operam por meio de *hardware* e *software*, sendo totalmente dependentes de uma infra-estrutura, conhecida como rede física (cabeamento ou ondas de rádio e placas de rede).

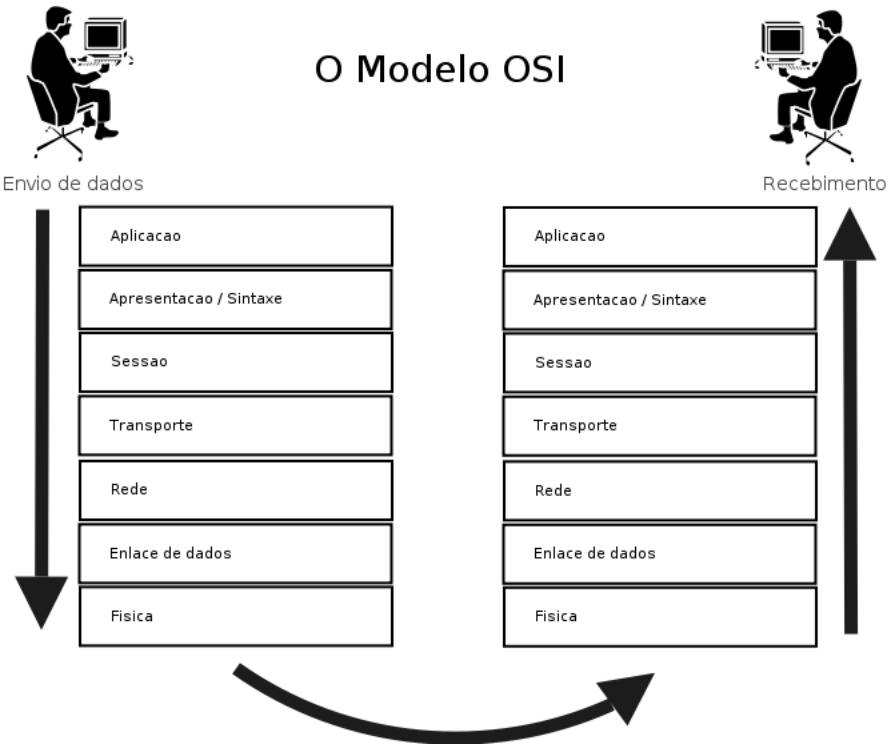
O modelo OSI define os aspectos de uma rede de comunicação de dados em sete camadas para a implementação de dispositivos, protocolos e aplicações.

Por ser um modelo, tem base em uma teoria e não significa que é obrigatório ou que está em todas as redes de comunicação.

O fato é que a *Internet* assim como o TCP/IP são baseados neste modelo.

O controle é sempre passado de uma camada a outra começando pela aplicação em um nó e prosseguindo até a camada mais baixa para ser transmitido ao outro nó, que faz todo o processo inverso.

Figura 5.1: Modelo OSI



5.4.1 Camada #7 - Aplicação

Esta é a camada das aplicações, dos programas. Os outros participantes do canal de comunicação - a conexão - são identificados, possivelmente criptografados e o formato dos dados e sua sintaxe são considerados.

[4]Referência para o Modelo OSI - http://pt.wikipedia.org/wiki/Modelo_OSI

5.4.2 Camada #6 - Apresentação / Sintaxe

Esta camada provê independência de diferenças na representação dos dados. Em outras palavras, é ela quem converte os dados da aplicação em um formato para ser transmitido via rede e vice-versa.

A camada de apresentação trabalha os dados criptografando-os, compactando ou simplesmente os convertendo de/para a aplicação.

5.4.3 Camada #5 - Sessão

Esta camada é a que gerencia conexões entre as aplicações: inicia, trafega dados e finaliza.

É nesta camada que a conexão é configurada e estabelecida, controlada e finalizada. Por aqui as aplicações trocam seus dados entre os nós.

Trata-se do gerenciamento de conexões, que é, na maioria das vezes, a própria sessão.

5.4.4 Camada #4 - Transporte

Responsável pela transferência de dados entre os nós. Esta camada lida com o gerenciamento de erros e o controle de fluxo dos dados.

Aqui controla-se os dados transmitidos e a transmitir.

5.4.5 Camada #3 - Rede

Esta camada provê a tecnologia de roteamento, responsável pela criação de caminhos lógicos entre as redes para que os dados possam então ser transmitidos de uma ponta a outra - de um nó ao outro.

Roteamento e repasse de pacotes *(packet forward)* são funções desta camada, bem como o endereçamento lógico dos nós, controle de erros, controle de congestionamento e sequenciamento de pacotes.

5.4.6 Camada #2 - Enlace de dados

Nesta camada os dados são codificados e decodificados em *bits*. Ela lida com o controle transmissão, sincronia e gerenciamento de erros da camada física.

Esta camada é dividida em duas sub-camadas:

- *MAC (Media Access Control)*
  Controla a maneira como um nó ganha acesso aos dados que chegam na rede e como ganha permissão para enviar dados à rede.
- *LCC (Logical Link Control)*
  Controla a sincronia, gerencia erros e faz o controle de fluxo.

5.4.7 Camada #1 - Física

Esta camada é onde ocorre a transmissão dos *bits* - pulsos elétricos, luzes infra-vermelho ou ondas de rádio - pela rede no que envolve a parte elétrica e mecânica.

Os protocolos *Fast Ethernet*, *ATM* e *RS232* são componentes desta camada.

5.5 O modelo real, da *Internet*

Embora baseado no Modelo OSI, o ambiente da *Internet* possui apenas quatro camadas:

5.5.1 Aplicação

Aplicações nos nós - programas nos computadores, sistemas dos roteadores, etc - já englobam as camadas 5, 6 e 7 do Modelo OSI.

5.5.2 Transporte

Exatamente a camada 4 do Modelo OSI.

5.5.3 *Internet*

A camada anteriormente chamada de Rede leva outro nome aqui, mas é idêntica à camada 3 do Modelo OSI.

5.5.4 Máquina-Rede

Uma junção das camadas 1 e 2 do Modelo OSI, que controla sincronia e gerenciamento de erros.

5.6 O Protocolo IP

É este protocolo, o *Internet Protocol,* que possibilita a criação de redes lógicas dando endereços a cada nó participante, de modo que o tráfego de dados seja direcionado para outro participante, de acordo com seu endereço.

No Modelo OSI, o protocolo IP se enquadra na camada 3, Rede.

Em suma, as principais funções deste protocolo são:

- Endereçar cada nó participante de uma rede lógica;
- Endereçar redes lógicas através de um sistema de roteamento.

Com os recursos do IP podemos criar:

- *LAN - Local Area Network*: uma rede local, caseira ou de escritório;
- *WAN - Wide Area Network*: uma rede expandida, normalmente formada por diversas *LANs*.

Atualmente utilizamos o IPv4, cujo nome deve-se ao formato de seu endereço: quatro octetos[5] com valores decimais.

Podemos concluir que endereços com quatro octetos são endereços de 32 *bits*!

Tabela 5.1: Notação decimal e binária de endereço IP

Base	**octeto 1**	**octeto 2**	**octeto 3**	**octeto 4**
10, decimal	192	168	19	5
2, binário	11000000	10101000	00010011	00000101

No dia a dia utilizamos apenas os endereços no formato decimal, porém é extremamente importante conhecê-los na notação binária[6].

[5] Um octeto é um número formato por 8 *bits*.

[6] Para ler os números binários mantenha em mente:

128	64	32	16	8	4	2	1

Quando ler um número binário, basta somar seus valores positivos.

00010010 == 18!

Sendo assim, o maior número decimal que um octeto pode ter é 255.
Em binário, 11111111.

5.6.1 Classes de IP

Os endereços são divididos em classes, sendo A, B, C e D.

Tabela 5.2: Classes de endereços IP

Classe	Início	Fim
A	**1**.0.0.0	**127**.255.255.255
B	**128**.0.0.0	**191**.255.255.255
C	**192**.0.0.0	**223**.255.255.255
D	**224**.0.0.0	**239**.255.255.255

A divisão dos endereços em classes dá-se, entre outros motivos, pela própria notação binária que é modificada conforme os números crescem.

- Na classe A, todos os endereços iniciam em 0;
- Na classe B, todos os endereços iniciam em 10;
- Na classe C, todos os endereços iniciam em 110;
- Na classe D, todos os endereços iniciam em 1110.

Veja:

Tabela 5.3: Divisão binária de classes IP

Decimal	Binário
1	**0**0000001
127	**0**1111111
128	**10**000000
191	**10**111111
192	**110**00000
223	**110**11111
224	**1110**0000
239	**1110**1111

5.6.2 Endereços IP especiais

O primeiro e mais comum é o endereço 127.0.0.1, conhecido como *loopback*. Qualquer sistema operacional com suporte ao protocolo IP possui este endereço que é utilizado para acessar serviços de rede no próprio computador.
Ele é atribuído a uma interface de rede virtual, simulando uma placa de rede.
Além disso, toda rede lógica tem três informações básicas:

- O endereço da rede *(network address)*;
- O endereço do fim da rede *(broadcast address)*;
- O endereço dos nós - número entre o endereço de rede e o *broadcast*.

Existem alguns intervalos de endereços que são reservados para redes locais e nunca existirão na *Internet*:

Tabela 5.4: Endereços IP não roteáveis na *Internet*

Classe	Início	Fim
A	**10**.0.0.0	**10**.255.255.255
B	**172.16**.0.0	**172.31**.255.255
C	**192.168**.0.0	**192.168**.255.255

Sendo assim, para construir redes caseiras ou de escritório, deve-se utilizar os endereços da tabela acima.

As empresas de telecomunicações fornecedoras de *link* de acesso disponibilizam endereços IP válidos na *Internet*[7].

5.6.3 Máscaras de Rede

Na configuração de uma rede lógica, cada nó deve atribuir pelo menos um endereço IP e uma máscara à placa de rede.

Embora exista uma divisão de classes de endereços, é a máscara que define o endereço da rede *(network address)* e o endereço do fim da rede *(broadcast address)*.

Cada classe possui uma máscara padrão:

Tabela 5.5: Máscaras de rede para classes IP

Classe	Máscara (base 10)	Máscara (base 2)	*CIDR*
A	255.0.0.0	11111111.00000000.00000000.00000000	/8
B	255.255.0.0	11111111.11111111.00000000.00000000	/16
C	255.255.255.0	11111111.11111111.11111111.00000000	/24

> **NOTA:** O *CIDR (Classless InterDomain Routing)* é um tipo de notação que permite escrever as máscaras de rede de maneira abreviada, mais eficiente e prática para administradores que pretendem realizar tarefas de roteamento e configuração de *firewall*. Basicamente, é o número de *bits* presentes na máscara de rede.

5.6.4 Simulando o endereçamento IP

Imagine-se configurando uma rede entre dois computadores. Ambos possuem placas de rede conectadas a um *hub* por meio de cabos, par trançado.

Para que ambos estejam na mesma rede lógica, você deve atribuir um endereço IP e uma máscara a cada um.

- Computador X: IP 192.168.10.1/24
- Computador Y: IP 192.168.10.2/24

Ao atribuir apenas essas informações, o sistema operacional[8] deve calcular os endereços de rede e fim da rede.

[7] Você também pode comprar endereços e até classes na Fapesp - http://www.registro.br.

[8] Qualquer sistema operacional moderno calcula o *network address* e *broadcast*.

É simples, utilizando a operação binária E (ou *AND*).

Exemplo, em binário:

Tabela 5.6: Notação binária de endereçamento IP

IP	11000000	10101000	00001010	00000001
Máscara	11111111	11111111	11111111	00000000
Network Address	**11000000**	**10101000**	**00001010**	**00000000**
Broadcast Address	**11000000**	**10101000**	**00001010**	**11111111**

Exemplo, em decimal:

Tabela 5.7: Notação decimal de endereçamento IP

IP	192	168	10	1
Máscara	255	255	255	0
Network Address	**192**	**168**	**10**	**0**
Broadcast Address	**192**	**168**	**10**	**255**

> **NOTA:** O cálculo do *network address* é apenas um *AND* binário entre o endereço IP e a máscara de rede. O cálculo do *broadcast address* é um *XOR* binário entre a máscara e um *NOT* no *network address* já calculado.

```
network_address = ip & máscara
broadcast_address = máscara ^ ~network_address
```

Se os cálculos mencionados foram aplicados aos endereços e máscaras dos computadores X e Y, você irá perceber que ambos pertencem à mesma rede. Agora faça o mesmo procedimento para os endereços seguintes:

- Computador X: IP 192.168.5.1/24
- Computador Y: IP 192.168.7.2/24

Então, são da mesma rede lógica?

5.7 Roteamento IP

O sistema de roteamento permite interconectar duas ou mais redes distintas através de regras para os pacotes IP em equipamentos denominados roteadores.

Para que a criação seja rotas seja possível, os roteadores devem estar conectados fisicamente um ao outro.

5.7.1 *Gateway*

Anteriormente foi mencionado o endereçamento IP e uma simulação de colocar dois ou mais computadores na mesma rede. Quando trata-se de uma rede local, atribuímos apenas duas informações para a interface de rede: endereço IP e máscara de rede.

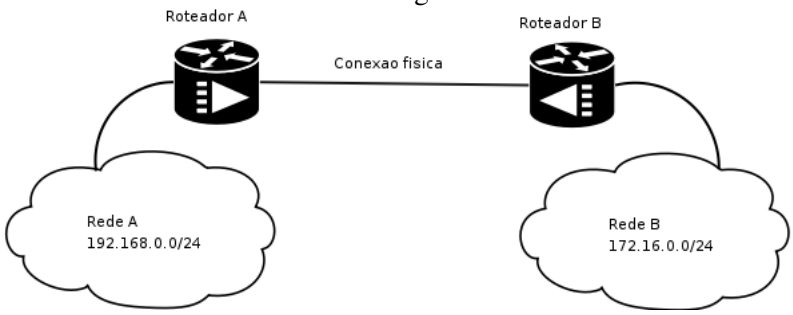
Para trabalhar em redes roteadas, as estações precisam de uma nova informação: o endereço do *gateway* - o endereço IP do roteador!

Quando uma estação tenta acessar um endereço que faz parte da mesma rede *(network address)* ela simplesmente envia seu pacote para o *hub* ou *switch*, então aguarda a resposta do destinatário. Se esta estação tentar acessar qualquer outro endereço que não faça parte de sua rede (intervalo de endereços entre *network e broadcast addresses*) este pacote irá para o *gateway*, encarregado de entregá-lo à rede de destino.

Mais detalhes a seguir.

5.7.2 Teoria do Roteamento IP

Figura 5.2: Redes IP roteadas

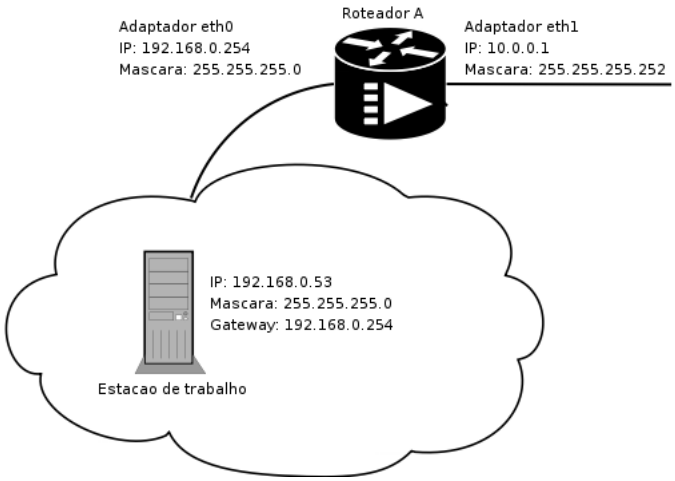


A figura acima ilustra um ambiente com duas redes, onde cada uma possui seu próprio roteador.

Em ambiente semelhante a este, cada roteador deve possuir ao menos duas interfaces de rede[9], sendo uma para a conexão com a rede local e outra para a conexão com o roteador remoto. Para conectar mais redes, o roteador precisa de mais interfaces de rede.

Os computadores nas redes A e B devem utilizar seus respectivos roteadores como *gateway*, como mostra a figura abaixo:

Figura 5.3: Ambiente de rede local



O adaptador[10] eth0 do roteador atende sua *LAN* pelo endereço IP 192.168.0.254, sendo apenas mais um nó na rede lógica.

Contudo, quando as estações passam a configurá-lo como *gateway*, ele poderá exercer sua função de roteador.

[9] Na nomenclatura de roteadores, as interfaces de rede também são conhecidas como **portas** - embora sejam placas de rede comuns.

[10] Adaptador, interface, porta - placa de rede

Se nenhuma estação configurar *gateway*, a rede funcionará normalmente, mas nunca será possível alcançar a rede B, e vice-versa.

Entre as tarefas de um administrador de redes está a criação de rotas. Para que isso seja possível é necessário ter um mapa físico da rede, um documento ou um desenho, ou um rascunho, ou qualquer rabisco que tenha as informações sobre qual cabo de rede se conecta a qual *hub* ou *switch*, ou a outro roteador.

De todos os cabos, ou rede sem fio, qual está conectado em cada interface do roteador.

Sem este mapa é praticamente impossível criar **rotas**.

É importante também considerar as características dos sistemas operacionais com relação à sua **tabela de roteamento**.

Basicamente, quando se atribui um endereço IP e máscara de rede a uma interface de rede, o sistema operacional cria uma rota em sua tabela, dizendo:

IP 192.168.0.5 MASK 255.255.255.0 DEV eth0

Significa que qualquer pacote com endereço da rede 192.168.0.0 até 192.168.0.255 deverá entrar e sair pela interface eth0.

5.7.3 *Subnet*

As sub-redes são extremamente importantes no ambiente de roteamento pois permitem dividir uma classe de endereços em pequenas redes.

Basta pensar nos fornecedores de *link*, que normalmente precisam fornecer intervalos de endereços a seus clientes, e graças às sub-redes é possível fornecer desde pequenos até grandes intervalos.

Até agora, a menor das redes conhecida é a rede do tipo Classe C, que permite 255 endereços - 253 removendo os dois especiais.

Utilizando sub-redes, a menor das redes é a divisão de uma Classe C em 64 pequenos pedaços, resultando em redes de 4 endereços.

Se você já está com um nó na cabeça, fique tranquilo, pois é exatamente o mesmo cálculo de IP e máscara de rede que será utilizado aqui.

Calculando o *network address e broadcast address*

Vamos utilizar como exemplo o endereço IP 10.0.0.5 e a máscara de rede 255.255.255.252.

Veja a notação binária:

Tabela 5.8: Cálculo de *subnet*

IP	00001010	00000000	00000000	00000101
Máscara	11111111	11111111	11111111	11111100
Network Address	00001010	00000000	00000000	**00000100**
Broadcast Address	00001010	00000000	00000000	**00000111**

No cálculo do octeto em questão, o último, realizamos um *AND* binário entre o IP e a máscara para chegar ao *network address*:

```
00000101 & 11111100 == 00000100 (decimal 4)
```

No cálculo do mesmo octeto, realizamos um *OR* binário entre a máscara e o *network address*, para chegar ao *broadcast address*!

```
11111100 ^ ~00000100 == 00000111 (decimal 7)
```

Sendo assim, podemos concluir:

- O *CIDR* da máscara 255.255.255.252 é /30;
- O computador com endereço 10.0.0.5/30 pertence à rede que vai de 10.0.0.4 até 10.0.0.7;
- As redes com máscara /30 possuem apenas 4 endereços;
- Redes com quatro endereços são ideais para roteamento, pois permitem um endereço para a rede, um para cada roteador e um para o final da rede.

Máscaras disponíveis em *Subnet*

Pela pura lógica, não é qualquer endereço que se pode utilizar nas máscaras para criar sub-redes.

A table abaixo mostra as possíveis máscaras e sua modificação em uma Classe C:

Tabela 5.9: Máscaras de *subnet*

Dividido em N pedaços	Máscara de rede	CIDR
2	255.255.255.128	/25
4	255.255.255.192	/26
8	255.255.255.224	/27
16	255.255.255.240	/28
32	255.255.255.248	/29
64	255.255.255.252	/30

Se o mesmo cálculo for realizado no octeto anterior, o penúltimo, temos então uma ***Supernet***, possibilitando o uso de máscaras de 255.255.128.0 (/17) até 255.255.252.0 (/22).

O endereço 192.168.100.12/17 pertence à rede 192.168.0.0 com *broadcast* 192.168.127.255. Por outro lado, o endereço 192.168.200.12/17 pertence à outra metade da rede, sendo 192.168.128.0 com *broadcast* 192.168.255.255.

Para um bom entendedor, é fácil notar que as máscaras são simplesmente uma evolução dos bits:

Tabela 5.10: Evolução dos *bits* em *subnets*

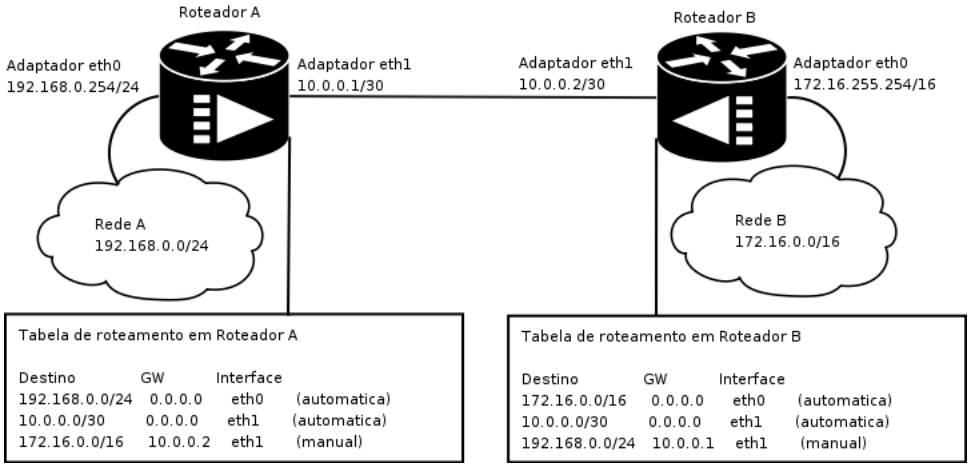
Decimal	Binário
128	10000000
192	11000000
224	11100000
240	11110000
248	11111000
252	11111100

5.7.4 Simulando o roteamento IP

Com todas essas informações, o roteamento fica simples.

A figura abaixo ilustra as mesmas redes A e B com suas tabelas de roteamento:

Figura 5.4: Rede IP funcional



Notavelmente, as rotas automáticas da tabela de roteamento são provenientes da configuração da interface de rede.

Para interconectar as redes A e B, o administrador criou UMA rota manual em cada roteador.

Passo a Passo

1. As estações da rede A utilizam o roteador A como *gateway;*
2. Quando uma estação da rede A, 192.168.0.20 envia um pacote a 172.16.0.7, este pacote vai diretamente para o *gateway* pois o endereço de destino não faz parte da rede A;
3. Quando o roteador A recebe o pacote (origem: 192.168.0.20, destino: 172.16.0.7) ele procura em sua tabela de roteamento uma interface para entregá-lo, e encontra a interface eth1;
4. Como há uma rede entre os roteadores, ao sair do roteador A o pacote chega ao rotador B, que consulta sua tabela de roteamento;
5. No rotador B, um de suas rotas diz que pacotes com destino à rede 172.16.0.0/16 deve ir para a interface eth0, ecoando no *hub* ou *switch* e sendo recebido pela estação com tal endereço;
6. O caminho de volta é exatamente o mesmo, porém os endereços de origem e destino do pacote são trocados.

5.7.5 Conclusão

A *Internet* é uma grande teia de roteadores com diversas *LANs* e *WANs* compondo sua estrutura. Nos exemplos citados foram utilizados IPs inválidos, porém na verdadeira rede os sistemas de roteamentos são feitos com IPs reais.

Em laboratório pode-se utilizar exatamente o esquema citado na Sessão 5.7.4 para efetuar testes.

5.8 Desenvolvendo aplicações simples

Então vamos ao desenvolvimento. Assume-se que você conhece o conteúdo mencionado até agora, a infra-estrutura e a lógica das redes.

Para começarmos com as aplicações algumas considerações devem ser levadas em conta:

1. Em computadores de plataforma x86[11] a ordem dos *bytes* é pelo menos significativo primeiro, enquanto no ambiente da *Internet* é pelo mais significativo primeiro[12];

2. Começaremos com programas que utilizam as funções disponibilizadas pela *libc* e não correspondem à conexão entre dois pontos diretamente, apenas tratamento de dados;

3. É necessário conhecer algumas estruturas de dados e arquivos de cabeçalho que serão apresentados aqui;

4. É de extrema importância ter os manuais da *libc* em mãos, pois não haverá cópia do conteúdo dos manuais aqui.

5.9 *Byte Order*

A diferença na ordem dos *bytes* é significativa no desenvolvimento de aplicações.
Veja:

Tabela 5.11: *Internet Byte Order*

Arquitetura	**IP**				
x86	**Base 10**	192	168	5	19
	Base 2	11000000	10101000	00000101	00010011
Internet	**Base 10**	19	5	168	192
	Base 2	00010011	00000101	10101000	11000000

Os octetos são organizados de forma diferente antes do pacote ir para a *Internet*. Caso seja necessário tratar uma sequência binária para conversão ou algo do tipo, é importante levar a ordem em consideração.

5.10 Conversão de *byte order*

Como já foi mencionado os endereços estão na notação do *byte* mais significativo primeiro, na *Internet*.

Para desenvolver aplicações que tratam endereços é necessário convertê-los para a notação correta. A *libc* disponibiliza uma família de funções INET(3) encarregadas da conversão.

5.10.1 A função *inet_aton()*

A função *inet_aton()* converte os endereços de *Internet* em sua notação comum - números e pontos, como 192.168.5.19 - para a notação binária. O resultado da conversão fica armazenado em uma *struct in_addr*. Se o resultado da conversão for inválido, retorna 0.

5.10.2 A função *inet_ntoa()*

A função *inet_ntoa()* faz a operação inversa da anterior. Aqui, ela recebe um endereço de *Internet* na notação binária e retorna uma *string* contendo o mesmo endereço na notação comum - números e pontos.

[11] X86 ou 80x86, o nome genérico para uma arquitetura de microprocessadores desenvolvidos e produzidos pela Intel.
[12] INET(3) - Linux Programmer's Manual

5.10.3 Programa para calcular endereço de rede

Baseando-se nas duas funções mencionadas até aqui, veremos um programa que faz o cálculo do *network address* e do *broadcast address* recebendo como parâmetro o endereço IP e a máscara de rede.

netcalc.c

```
/*
 * netcalc: recebe ip e netmask e calcula network address e broadcast
 *
 * Para compilar:
 *  cc -Wall netcalc.c -o netcalc
 *
 * Alexandre Fiori
 */

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

/* imprime o 'prefix' e depois a notação binária e decimal do 'addr' */
static void bitprint(const char *prefix, struct in_addr addr);

/* necessita de dois argumentos na linha de comando */
int main(int argc, char **argv)
{
    struct in_addr ipaddr, netmask, network, broadcast;

    if(argc != 3) {
        fprintf(stderr, "use: %s ipaddr netmask\n"
                        "exemplo: %s 192.168.15.2 255.255.255.128\n",
                        *argv, *argv);
        return 1;
    }

    /* converte o IP para network byte order, salva o resultado em 'ipaddr' */
    if(!inet_aton(argv[1], &ipaddr)) {
        fprintf(stderr, "endereço (ipaddr=%s) inválido!\n", argv[1]);
        return 1;
    }

    /* converte a máscara para network byte order */
    if(!inet_aton(argv[2], &netmask)) {
        fprintf(stderr, "endereço (netmask=%s) inválido\n", argv[2]);
        return 1;
    }

    /* calcula o network address */
    network.s_addr = ipaddr.s_addr & netmask.s_addr;

    /* calcula o broadcast */
    broadcast.s_addr = netmask.s_addr ^ ~network.s_addr;

    /* imprime o valor de todos os endereços */
    bitprint("ipaddr:    ", ipaddr);
    bitprint("netmask:   ", netmask);
    bitprint("network:   ", network);
    bitprint("broadcast: ", broadcast);

    return 0;
}

static void bitprint(const char *prefix, struct in_addr addr)
{
    int i;
    /* utiliza 'mask' para imprimir apenas os bits ligados
     * e 'dots' para saber quando deve imprimir um '.' */
    unsigned long int mask = 0x80000000, dots = 0x01010100;
```

```
    /* imprime o prefixo */
    fprintf(stdout, "%s", prefix);

    /* assume-se endereços 32 bits, sizeof(long int) */
    for(i = 0; i < 32; i++) {
        fprintf(stdout, "%d", addr.s_addr & mask ? 1 : 0);
        if(mask & dots) fprintf(stdout, ".");
        mask >>= 1;
    }

    /* imprime o endereço em notação decimal */
    fprintf(stdout, " (%s)\n", inet_ntoa(addr));
}
```

Executando:

```
$ ./netcalc 192.168.100.5 255.255.128.0
ipaddr:    00000101.01100100.10101000.11000000 (192.168.100.5)
netmask:   00000000.10000000.11111111.11111111 (255.255.128.0)
network:   00000000.00000000.10101000.11000000 (192.168.0.0)
broadcast: 11111111.01111111.10101000.11000000 (192.168.127.255)

$ ./netcalc 192.168.200.5 255.255.128.0
ipaddr:    00000101.11001000.10101000.11000000 (192.168.200.5)
netmask:   00000000.10000000.11111111.11111111 (255.255.128.0)
network:   00000000.10000000.10101000.11000000 (192.168.128.0)
broadcast: 11111111.11111111.10101000.11000000 (192.168.255.255)
```

5.11 Utilizando DNS para resolver endereços

A resolução de endereços IP para nome e vice-versa depende totalmente da configuração do sistema, principalmente do arquivo */etc/resolv.conf*.

Para que a resolução funcione corretamente você precisa ter acesso à rede e um servidor DNS configurado.

A *libc* possui uma família de funções[13] que faz a resolução por meio de DNS.

5.11.1 As funções *gethostbyname()* e *gethostbyaddr()*

Estas funções utilizam a configuração do sistema para se conectar a um servidor DNS e solicitar a resolução de endereços.

Elas retornam uma estrutura do tipo *struct hostent* com o resultado da operação. A primeira, *gethostbyname()* é utilizada quando você tem um nome, como *google.com* e precisa ter seu endereço IP.

A segunda, *gethostbyaddr()* é o inverso, quando você tem um endereço IP e precisa saber seu nome - processo conhecido como DNS reverso.

5.11.2 As funções *sethostent()* e *endhostent()*

Esta função permite que a conexão com o servidor DNS permaneça aberta para realizar pesquisas sucessivas. Antes de executar *gethostbyname()* você pode chamá-la ou não.

Caso seu programa vá fazer uma única consulta no servidor DNS, ela é totalmente dispensável. Já no caso de realizar duas ou mais pesquisas é extremamente manter a conexão aberta para evitar tráfego desnecessário.

Após fazer todas as pesquisas, deve-se avisar a *libc* para fechar a conexão, utilizando *endhostent()*.

[13] GETHOSTBYNAME(3) - Linux Programmer's Manual

5.11.3 A função *herror()*

Durante o processo de resolução de nome pelas funções *gethostbyname()* ou *gethostbyaddr()* pode ocorrer erros. Caso o sistema operacional não tenha um servidor DNS configurado, a resolução não será possível.

Se o sistema operacional possuir um servidor DNS mas este está com problemas ou fora do ar, a resolução também não será possível.

Caso o sistema tenha o servidor DNS configurado e este está respondendo adequadamente, ainda pode haver o erro de não encontrar o nome ou IP solicitado.

A função *herror()* é encarregada de imprimir a mensagem de erro corretamente caso as funções de resolução falhem.

5.11.4 Programa para resolver endereços para nome e vice-versa

O código abaixo solicita um endereço na forma de nome ou IP e faz a resolução DNS para seu IP ou nome, respectivamente.

Ele está preparado para operar com endereços IPv4.

resolver.c

```
/*
 * resolver.c: resolve endereços para nome e vice-versa
 *
 * Para compilar:
 *  cc -Wall resolver.c -o resolver
 *
 *  Alexandre Fiori
 */

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

/* necessita ao menos um argumento na linha de comando */
int main(int argc, char **argv)
{
    struct hostent *he;
    struct in_addr  ip;

    if(argc != 2) {
        fprintf(stderr, "use: %s addr\n"
                        "exemplo: %s google.com\n",
                        *argv, *argv);
        return 1;
    }

    /* argv[1] pode ser um nome como 'google.com' ou um número,
     * como '192.168.0.7';
     * para sabermos qual operação utilizar, basta tentar transformar
     * o conteúdo de argv[1] utilizando inet_aton();
     */

    if(inet_aton(argv[1], &ip)) {
        /* caso seja um endereço IP, devemos solicitar o DNS reverso */
        he = gethostbyaddr(&ip, sizeof(ip), AF_INET);
        if(!he) {
            herror("gethostbyaddr");
            return 1;
        }
    } else {
        /* caso seja um nome, devemos resolver para o IP */
```

```
        he = gethostbyname(argv[1]);
        if(!he) {
            herror("gethostbyname");
            return 1;
        }
    }

    /* imprime todos os endereços retornados */
    fprintf(stdout, "host: %s\n", he->h_name);

    while(*he->h_addr_list) {
        struct in_addr *in = (struct in_addr *) *he->h_addr_list;

        fprintf(stderr, "%s\n", inet_ntoa(*in));
        he->h_addr_list++;
    }

    return 0;
}
```

Executando:

```
$ ./resolver google.com
host: google.com
64.233.187.99
72.14.207.99

$ ./resolver www.uol.com.br
host: www.uol.com.br
200.221.2.45

$ ./resolver 200.221.2.45
host: home.uol.com.br
200.221.2.45
```

5.11.5 Conclusão

Todos ou 99.9% dos aplicativos existentes em ambiente *Unix-like* utilizam o mesmo procedimento para a resolução de nomes.

A estrutura *struct hostent* é genérica e possui todas as informações relacionadas à resolução de nomes.

Uma descrição detalhada de seu conteúdo pode ser encontrado na página de manual GETHOSTBYNAME(3).

```
$ man gethostbyname
```

5.12 Identificando serviços e protocolos

Os serviços de rede operam por meio de protocolos como TCP e UDP que serão vistos a seguir.

Esses protocolos utilizam portas para multiplexar o acesso a seus recursos e permitir que diversos clientes acessem os servidores.

Cada porta de serviço tem um respectivo nome.

5.12.1 O arquivo */etc/services*

Este arquivo está presente em 99% dos sistemas *Unix-like* e é uma espécie de base de dados com informações relacionadas a serviços e protocolos.

Nosso objetivo é conhecer as funções da *libc* que realizam pesquisa neste arquivo através da linguagem C.

5.12.2 As funções *getservbyname()* e *getservbyport()*

A maneira de utilizar estas funções é muito semelhante ao uso de *gethostbyname()* e *gethostbyaddr()*, porém a estrutura preenchida por elas é *struct servent.*

Para lidar com essas funções é necessário conhecer um mínimo sobre serviços, suas respectivas portas e protocolos, como por exemplo:

- O serviço HTTP opera por meio de TCP
- O serviço SMTP opera por meio de TCP
- O serviço SNMP opera por meio de UDP
- O serviço TFTP opera por meio de UDP

Essas informações podem ser facilmente obtidas por uma rápida leitura do arquivo */etc/services*, mas durante o desenvolvimento de aplicações é necessário conhecê-las.

Caso você não conheça exatamente (incompetente!) a porta dos serviços e seu protocolo, poderá omití-lo e as funções irão retornar a primeira entrada com qualquer protocolo.

5.12.3 As funções *setservent()* e *endservent()*

Ao realizar pesquisa no arquivo */etc/services* a *libc* precisa abrí-lo no modo de leitura e depois fechá-lo. Para que sucessivas pesquisas não realizem essas operações diversas vezes, você pode avisar quando ele deve ser aberto e quando deve ser fechado.

Caso haja uma única pesquisa a ser feita, o uso dessas funções é dispensável.

5.12.4 As funções *htons()* e *ntohs()*

Normalmente utilizamos *unsigned short int* para o número das portas, mas como já sabemos a ordem dos *bytes* na *Internet* é diferente do ambiente x86[14].

Para realizar a conversão correta dos números de porta devemos utilizar essas duas funções.

A porta de número 80, por exemplo, tem a seguinte notação:

```
00000000 01010000
```

Mas na *Internet* ela é:

```
01010000 00000000
```

Portanto, no código devemos utilizar sempre *htons()* e *ntohs()* para fazer e desfazer a conversão.

Também existem as versões para *unsigned long int*, úteis na conversão de enderecos IP, sendo *htonl()* e *ntohl()*.

Tente adaptá-las ao programa *netcalc.c* na Sessão 5.10.3.

[14] BYTEORDER(3) - Linux Programmer's Manual

5.12.5 Programa para identificar serviços e portas

Segue o programa que realiza pesquisa na base de serviços e portas por meio da *libc*.

servres.c

```
/*
 * servres.c: recebe nome ou número da porta e protocolo e informa
 *            todos os dados contidos na base, /etc/services.
 *
 *  Para compilar:
 *   cc -Wall servres.c -o servres
 *
 *  Alexandre Fiori
 */

#include <stdio.h>
#include <netdb.h>
#include <ctype.h>
#include <stdlib.h>

/* necessita dois argumentos na linha de comando */
int main(int argc, char **argv)
{
    struct servent *se;
    const char *proto = argv[2]; /* opcional */

    if(argc < 2) {
        fprintf(stderr, "use: %s porta [protocolo]\n"
                        "exemplo: %s 80 tcp\n"
                        "exemplo: %s smtp\n",
                        *argv, *argv, *argv);
        return 1;
    }

    /* argv[1] pode conter um nome ou número de porta
     * para definir qual operação devemos utilizar é necessário isdigit()
     */

    if(isdigit(*argv[1]))
        /* caso seja um número... */
        se = getservbyport(htons(atoi(argv[1])), proto);
    else
        /* caso seja um nome... */
        se = getservbyname(argv[1], proto);

    if(!se) {
        fprintf(stderr, "serviço ou protocolo não encontrado.\n");
        return 1;
    }

    /* imprime o resultado */
    fprintf(stdout, "porta %d (%s), protocolo %s\n",
            ntohs(se->s_port), se->s_name, se->s_proto);

    return 0;
}
```

Executando:

```
$ ./servres www
porta 80 (www), protocolo tcp

$./servres http
porta 80 (www), protocolo tcp

$ ./servres snmp
porta 161 (snmp), protocolo tcp

$ ./servres 53 udp
```

```
porta 53 (domain), protocolo udp

$ ./servres 110
porta 110 (pop3), protocolo tcp
```

5.12.6 O arquivo */etc/protocols*

Este arquivo é uma espécie de base de dados com informações sobre o nome e número dos protocolos - não dos serviços!

A *libc* também provê funções para pesquisa nele para que os programas funcionem de acordo com a configuração do sistema.

5.12.7 As funções *getprotobyname()* e *getprotobynumber()*

Assim como *getservbyname()* e *getservbyport()*, essas funções buscam o protocolo por nome ou número e retornam número ou nome, respectivamente.

A estrutura preenchida por elas é *struct protoent*.

5.12.8 As funções *setprotoent()* e *endprotoent()*

Como nas outras semelhantes, essas funções abrem e fecham o arquivo */etc/protocols* e só devem ser utilizadas em pesquisas sucessivas.

5.12.9 Programa para identificar protocolos

Segue o programa que realiza pesquisa na base de protocolos e imprime seus respectivos nomes e números por meio da *libc*.

protores.c

```
/*
 * protores.c: recebe nome ou número do protocolo e informa
 *             todos os dados contidos na base, /etc/protocols.
 *
 *  Para compilar:
 *   cc -Wall protores.c -o protores
 *
 *  Alexandre Fiori
 */

#include <stdio.h>
#include <netdb.h>
#include <ctype.h>
#include <stdlib.h>

/* necessita um argumento na linha de comando */
int main(int argc, char **argv)
{
    struct protoent *pe;

    if(argc != 2) {
        fprintf(stderr, "use: %s [nome|número]\n"
                        "exemplo: %s tcp\n"
                        "exemplo: %s 1\n",
                        *argv, *argv, *argv);
        return 1;
    }

    /* argv[1] pode conter um nome ou número do protocolo
     * para definir qual operação devemos utilizar é necessário isdigit()
```

```
     */

    if(isdigit(*argv[1]))
        /* caso seja um número... */
        pe = getprotobynumber(atoi(argv[1]));
    else
        /* caso seja um nome... */
        pe = getprotobyname(argv[1]);

    if(!pe) {
        fprintf(stderr, "protocolo não encontrado.\n");
        return 1;
    }

    /* imprime o resultado */
    fprintf(stdout, "protocolo %d (%s)\n",
            pe->p_proto, pe->p_name);

    return 0;
}
```

Executando:

```
$ ./protores ip
protocolo 0 (ip)

$ ./protores tcp
protocolo 6 (tcp)

$ ./protores 17
protocolo 17 (udp)

$ ./protores 50
protocolo 50 (esp)

$ ./protores 19
protocolo não encontrado.
```

5.13 Conexões lógicas: *Sockets*

Uma definição precisa para *socket* depende do ponto de vista:

1. No *kernel space* um *socket* é uma estrutura de dados *struct file_operations* - é um arquivo!
2. No *user space* um *socket* é um *int*, um número relacionado a determinada conexão - como um arquivo aberto com *open()*!

Para os desenvolvedores de aplicações, trabalhar com *sockets* é muito semelhante a trabalhar com arquivos. O sistema de leitura e escrita é praticamente o mesmo, porém abrir um arquivo é diferente de abrir um *socket*.

Na *libc* há uma família inteira de funções relacionadas aos *sockets*[15].

5.14 Famílias de *Socket*

Existem diversas famílias de *sockets* disponíveis na API de programação da *libc*[16], onde apenas duas serão utilizadas:

[15] SOCKET(7) - Linux Programmer's Manual
[16] SOCKET(2) - Linux Programmer's Manual

Tabela 5.12: Famílias de *Socket*

Address Family	***Protocol Family***	**Descrição**
AF_INET	PF_INET	*socket* da família *Internet*
AF_UNIX	PF_UNIX	*socket* da família *Unix*

Para uma lista completa das famílias pode-se consultar o arquivo de cabeçalho do *kernel*[17].

```
$ less /usr/include/linux/socket.h
```

Os *sockets* também têm um tipo, além da familia. Este tipo pode ser SOCK_STREAM ou SOCK_DGRAM e será visto a seguir.

5.14.1 *Sockets* da família AF_INET

São os mais comuns e mais utilizados. Servem para estabelecer conexão entre dois pontos utilizando um protocolo como TCP ou UDP.

Trabalham com base em servidor e cliente e são comums nos serviços da *Internet* como web, e-mail, ftp...

5.14.2 *Sockets* da família AF_UNIX

O *Unix Domain Socket* é muito semelhante à família AF_INET, porém não há conexão de rede física. Esta família é utilizada apenas para trocar informações entre dois ou mais processos do mesmo computador.

Ao invés de portas, são utilizados arquivos.

Em aplicações como bancos de dados esses *sockets* são utilizados para que o administrador possa se conectar ao terminal e executar comandos. Como você já deve saber, um banco de dados opera como um *daemon*[18] portanto para ter acesso às suas funções é necessário um terminal. Este *daemon* então cria um *socket* do tipo AF_UNIX utilizando um arquivo como por exemplo */tmp/db_socket*. O terminal do banco de dados se conecta a este arquivo utilizando TCP ou UDP e troca informações com o *daemon*.

O ambiente gráfico X11 também utiliza *Unix Domain Sockets* para se comunicar com os programas gráficos. Veja:

```
$ ls -l /tmp/.X11-unix/
total 0
srwxrwxrwx  1 root root 0 Nov 14 18:29 X0
```

São de extrema utilidade para nós, programadores.

5.15 Tipos de *Socket*

Segue a tabela com os tipos mais comuns de *sockets*:

Tabela 5.13: Tipos de *Socket*

Tipo	**Descrição**
SOCK_STREAM	Provê conexão por sessão, transmissão de ida-e-volta sequenciada
SOCK_DGRAM	Suporta datagramas sem conexão, envio de mensagens de tamnho fixo
SOCK_RAW	Acesso ao pacote todo, incluindo cabeçalhos

[17] É necessário ter o código fonte do *kernel* instalado.

[18] Programa que quando executado não tem interface com o usuário, é um processo em *background*.

5.15.1 *Sockets* do tipo SOCK_STREAM

Normalmente utilizado em conexões do tipo TCP.

5.15.2 *Sockets* do tipo SOCK_DGRAM

Normalmente utilizado em conexões do tipo UDP.

5.15.3 *Sockets* do tipo SOCK_RAW

Semelhante ao *Packet Socket*[19], este tipo é utilizado quando o programador precisa enviar ou receber os pacotes junto com seus cabeçalhos.

Nos outros tipos de *sockets* o programador também tem acesso aos cabeçalhos de pacotes, porém utilizando SOCK_RAW um pacote é uma *string* contendo o pacote inteiro, em nível de *driver* (OSI camada 2) - conhecido como *raw packet*.

Aqui utilizaremos para capturar ICMP.

5.16 Protocolos

Aqui temos os conjuntos de regras, chamados protocolos, respectivos à camada 4 do modelo OSI. Entre eles estão TCP, UDP e ICMP que serão detalhados.

5.16.1 O Protocolo TCP

Este protocolo é o principal ao lado do IP. Juntos, formam a dupla dinâmica - um protocolo de roteamento e um protocolo de transporte.

Chama-se *Transmission Control Protocol* e sua função principal é controlar o fluxo de dados entre dois *sockets* - normalmente cliente e servidor.

Pelo cabeçalho[20] do TCP é possível compreender sua função:

```
/*
 * TCP header.
 * Per RFC 793, September, 1981.
 */
struct tcphdr
  {
    u_int16_t th_sport;         /* source port */
    u_int16_t th_dport;         /* destination port */
    tcp_seq th_seq;             /* sequence number */
    tcp_seq th_ack;             /* acknowledgement number */
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;           /* (unused) */
    u_int8_t th_off:4;          /* data offset */
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;          /* data offset */
    u_int8_t th_x2:4;           /* (unused) */
#  endif
    u_int8_t th_flags;
#  define TH_FIN        0x01
#  define TH_SYN        0x02
#  define TH_RST        0x04
#  define TH_PUSH       0x08
#  define TH_ACK        0x10
#  define TH_URG        0x20
```

[19] PACKET(7) - Linux Programmer's Manual

[20] Cópia do original da *libc*, em */usr/include/netinet/tcp.h*

```
    u_int16_t th_win;           /* window */
    u_int16_t th_sum;           /* checksum */
    u_int16_t th_urp;           /* urgent pointer */
};
```

As conexões do tipo TCP são baseadas em sessão (cliente e servidor), portanto, para que isso seja possível o protocolo utiliza índices, conhecidos como portas, que nada mais são além de números do tipo *unsigned short int*, que vinculam um determinado programa a uma conexão.

Esta é a tal multiplexação, que funciona da seguinte maneira:

Quando um programa é executado, ele possui um número de indentificação no sistema operacional, chamado de *Proccess ID*, ou simplesmente *PID*. Este número, no *kernel space* é uma estrutura de dados com todas as informações daquele processo.

Um programa como o *apache*, por exemplo, solicita ao sistema operacional por meio de uma *syscall* a alocação da porta de número 80 do protocolo TCP.

A partir deste momento, ele se torna um servidor. O sistema operacional sabe que qualquer pacote que chegar a ele do tipo TCP, com a porta de número 80, deve ser entregue àquele PID. Portanto, o *apache*, um programa no *user space*, passa a receber requisições da rede.

Se o processo for finalizado, o serviço for desligado, aquela porta volta a ser livre para qualquer outro processo que solicitá-la.

Do outro lado, há um navegador, um *browser*. Quando o usuário digita uma URL do tipo http://, o navegador já sabe que trata-se de TCP/80. Ele então pega o nome da URL e faz uma conversão de DNS, obtém o endereço IP e cria um *socket* entre ele e o destinatário.

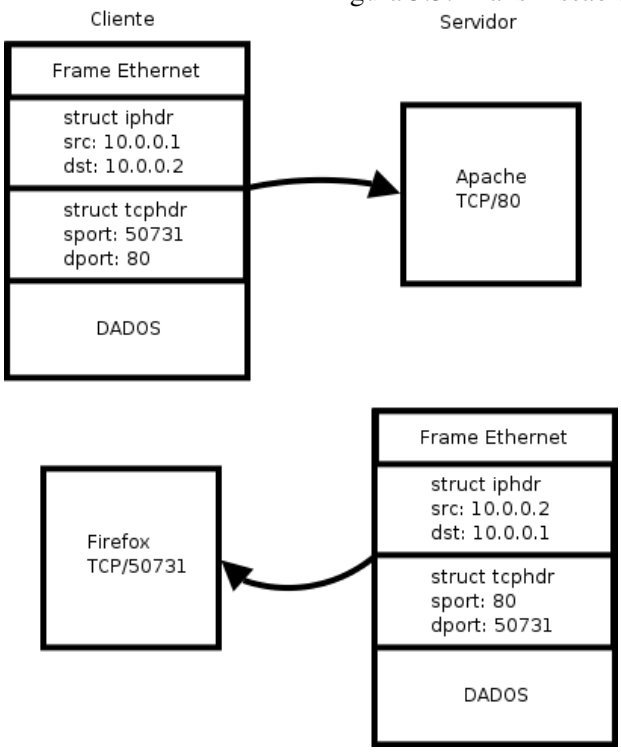
Deste lado, o cliente também cria uma porta (aleatória) no sistema operacional. Esta porta é normalmente um número entre 1024 e 65535, pois as portas *baixas*, de 1 a 1023 são convencionadas a ser de uso de servidores - mas não obrigatórias!

Enfim, agora o processo *Firefox* tem uma porta TCP/50731 vinculada a seu PID. Quando ele enviar um pacote ao servidor *apache,* TCP/80, este irá receber os dados, processá-los e devolver. Devolver onde? Oras, no IP de origem do pacote, o IP do computador onde está o navegador, que veio no cabeçalho IP do pacote, na porta de origem TCP/50731 que agora se tornou porta de destino.

Portas Lógicas

Ao olhar para o cabeçalho do TCP, especialmente para os dois primeiros campos que são *th_sport* e *th_dport*, podemos simular uma transmissão de dados:

Figura 5.5: Transmissão de dados do TCP



Então, podemos concluir que o pacote chega, o programa processa e devolve - onde ocorre a inversão dos campos no pacote.

Controle de Fluxo

Ainda no cabeçalho do TCP, encontramos os campos *th_seq* e *th_ack*. Eles controlam o fluxo dos dados na conexão.

Depois de conectado, o processo que envia um pacote atribui o valor 1 ao *th_seq* e 0 ao *th_ack*. O processo que recebe o pacote, processa e devolve com 1 no *th_seq* e agora 1 no *th_ack.*

E assim sucessivamente:

Tabela 5.14: Controle de fluxo do TCP

Cliente	Servidor
seq1, ack0	seq1, ack1
seq2, ack1	seq2, ack2
seq3, ack2	seq3, ack3

Caso um pacote chegue fora de ordem ou duplicado (colisão), este deve ser reenviado.

Controle de Sessão

Através do campo *th_flags* o TCP controla sessão. Quando o cliente se *conecta* no servidor ele envia um pacote sem dados, com o valor 0x02 (TH_SYN) no campo *th_flags*. O servidor então responde com 0x10, e aí está iniciada a sessão.

Para finalizar, um dos dois envia 0x01 e deve receber 0x04.

Portabilidade

Aqui entra o tamanho da janela TCP, o campo *th_win*. Nele, o emissor do pacote coloca o valor em *bytes* do tamanho máximo de pacote que deseja receber. Caso o receptor emita um pacote com mais *bytes* que o permitido, este pacote é descartado.

Isso possibilita que sistemas de diferentes arquiteturas e quantidade de memória troquem pacotes sem causar danos ao sistema operacional.

Confiabilidade

Através do campo *th_sum* o TCP realiza o *checksum* dos pacotes. Caso o emissor envie um pacote de 70 *bytes*, é este o valor do *th_sum* no cabeçalho.

Quando o receptor tiver o pacote inteiro em mãos, consulta seu tamanho e compara com o *th_sum*, que deve ser o mesmo. Caso seja diferente, o pacote deve ser reenviado.

Considerações

Todas essas características estão implementadas no *kernel space*, portanto programadores de aplicações não precisam se preocupar com todo este controle do protocolo, pois ele é feito internamente pelo sistema operacional.

Que tal escrever um?

5.16.2 O Protocolo UDP

Este protocolo, *User Datagram Protocol*, é bem mais simples que o TCP e opera através de datagramas, que são pacotes de tamanho fixo.

O UDP não trabalho baseado em sessão, um processo apenas envia o datagrama para a rede e outro processo o recebe.

O cabeçalho do UDP[21]:

```
/* UDP header as specified by RFC 768, August 1980. */
struct udphdr
{
  u_int16_t uh_sport;           /* source port */
  u_int16_t uh_dport;           /* destination port */
  u_int16_t uh_ulen;            /* udp length */
  u_int16_t uh_sum;             /* udp checksum */
};
```

Ele também utiliza multiplexação para que seja possível atender clientes simultâneos, exatamente como o TCP.

Também realiza *checksum* nos pacotes.

5.16.3 O Protocolo ICMP

Este serve para controle de mensagens e erros na rede. o *Internet Control Message Protocol* é famoso por ser utilizado no *ping*.

Todo pacote ICMP tem um tipo e um código, sendo este um sub-tipo.

No caso do *ping*, o programa cria um pacote ICMP com o tipo 8 (ICMP_ECHO) e o envia para o destinatário. Ao receber um pacote com estas características, o destinatário devolve ao remetente com o tipo 0 (ICMP_ECHOREPLY).

Segue o cabeçalho do ICMP[22]:

[21] Cópia do original da *libc*, em */usr/include/netinet/udp.h*

[22] Cópia do original da *libc*, em */usr/include/netinet/ip_icmp.h*

```
struct icmphdr
{
  u_int8_t type;                /* message type */
  u_int8_t code;                /* type sub-code */
  u_int16_t checksum;
  union
  {
    struct
    {
      u_int16_t id;
      u_int16_t sequence;
    } echo;                     /* echo datagram */
    u_int32_t   gateway;        /* gateway address */
    struct
    {
      u_int16_t __unused;
      u_int16_t mtu;
    } frag;                     /* path mtu discovery */
  } un;
};
```

5.17 Funções e estruturas

A família de funções da *libc* relacionadas aos *sockets* é bem grande[23]. Aqui, veremos apenas algumas que são consideravelmente importantes para escrever programas clientes e servidores.

5.17.1 A função *socket()*

Os *sockets* trabalham sempre em pares. Esta função cria uma ponta para comunicação.

Para criar um *socket* é necessário informar a família, o tipo e o protocolo a ser utilizado.

Exemplo:

```
...
struct protoent *pe = getprotobyname("tcp");
int fd = socket(AF_INET, SOCK_STREAM, pe->p_proto);
...
```

O código acima cria um *socket* do família AF_INET, tipo SOCK_STREAM para o protocolo TCP. Em *fd* temos os descritor de arquivo que nos dá acesso a este ponto da comunicação - embora ele ainda não esteja conectado em nenhum lugar.

Os *Unix Domain Sockets* são muito semelhantes, veja:

```
...
struct protoent *pe = getprotobyname("udp");
int fd = socket(AF_UNIX, SOCK_DGRAM, pe->p_proto);
...
```

O código acima cria um *socket* da família AF_UNIX, tipo SOCK_DGRAM para o protocolo UDP. Nada impede de utilizar esta família com o tipo SOCK_STREAM e o protocolo TCP assim como os *sockets* da família AF_INET também podem ser do tipo SOCK_DGRAM para o protocolo UDP.

Depois de ter o *socket* criado, precisamos fazê-lo se tornar algo: um servidor ou um cliente. Este *socket* também pode ser utilizado para extrair informações da placa de rede ou colocar lá novas propriedades, e também pode ser utilizado para enviar ou receber mensagens do protocolo ICMP.

As próximas funções, relacionadas a conexão, são dependentes de uma estrutura de dados do tipo *struct sockaddr*, que define as características do meio de comunicação para cada família de *socket*.

Antes de conhecer as funções é necessário conhecer as estruturas *struct sockaddr*, *struct sockaddr_in* e *struct sockaddr_un*.

[23] SOCKET(2) - Linux Programmer's Manual (na sessão SEE ALSO)

5.17.2 A estrutura *struct sockaddr*

A definição desta estrutura é a seguinte[24]:

```
typedef unsigned short  sa_family_t;
/*
 *      1003.1g requires sa_family_t and that sa_data is char.
 */
struct sockaddr {
        sa_family_t     sa_family;      /* address family, AF_xxx       */
        char            sa_data[14];    /* 14 bytes of protocol address */
};
```

Esta definição genérica tem um único campo em comum com as demais: a família. O campo *sa_data* é preenchido pelos valores definidos nas outras estruturas mencionadas, *struct sockaddr_in* ou *struct sockaddr_un*. Portanto, ela nunca é utilizada diretamente.

Esta é a maneira mais simples de fazer com que as funções como *connect()* sejam válidas para *sockets* de diferentes famílias, pois todas recebem como argumento uma *struct sockaddr* genérica e pela família são capazes de saber qual é o tipo de dados real que está lá.

O tamanho desta estrutura genérica pode ser calculado: um *unsigned short* tem normalmente 2 *bytes*, cada *char* tem 1 *byte*, portanto um vetor de 14 *chars* tem 14 *bytes*. Sendo assim, a estrutura tem 16 *bytes*.

5.17.3 A estrutura *struct sockaddr_in*

Segue a definição da desta estrutura[25]:

```
/* Structure describing an Internet (IP) socket address. */
#define __SOCK_SIZE__   16              /* sizeof(struct sockaddr)      */
struct sockaddr_in {
  sa_family_t           sin_family;     /* Address family               */
  unsigned short int    sin_port;       /* Port number                  */
  struct in_addr        sin_addr;       /* Internet address             */

  /* Pad to size of 'struct sockaddr'. */
  unsigned char         __pad[__SOCK_SIZE__ - sizeof(short int) -
                        sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

Os campos aqui são *sin_family*, *sin_port* e *sin_addr*. Parecem claros?

O campo *__pad* é apenas um enchimento, para fazer com que esta estrutura tenha exatamente o mesmo tamanho da *struct sockaddr* genérica.

5.17.4 A estrutura *struct sockaddr_un*

Esta, ao invés de ter porta e endereço, tem um caminho para o arquivo relacionado à conexão - como foi explicado na Sessão 5.14.2. Veja:

```
#define UNIX_PATH_MAX   108

struct sockaddr_un {
        sa_family_t sun_family; /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

Aqui, os campos são *sun_family* e *sun_path*, sendo este segundo uma *string* com o caminho para o arquivo que é o *socket*.

[24] Cópia da original no código fonte do *kernel*, em /usr/include/linux/socket.h

[25] Cópia da original no código fonte do *kernel*, em /usr/include/linux/in.h

5.17.5 As funções *shutdown()* e *close()*

Depois de utilizar um *socket*, ele deve ser fechado. Assim como um arquivo, o *socket* pode ser fechado com *close()*:

```
...
close(fd);
...
```

Já a função *shutdown()* é utilizada para fechar parte do *socket* ou ele todo. A característica desta função é cancelar parte de uma conexão *full-duplex*, recebendo dois parâmetros: o descritor de arquivo do *socket* e a parte que deseja fechar:

Tabela 5.15: Maneiras de eliminar parte da conexão

Nome	Número	Descrição
SHUT_RD	0	Desabilita o recebimento de dados
SHUT_WR	1	Desabilita o envio de dados
SHUT_RDWR	2	Desabilita ambos

Normalmente *shutdown()* é utilizada antes de *close()*, pois *close()* apenas elimina o descritor de arquivo do *kernel space*.

Veja:

```
...
shutdown(fd, 2);
close(fd);
...
```

É exatamente a mesma que:

```
...
shutdown(fd, SHUT_RDWR);
close(fd);
...
```

5.17.6 A função *connect()*

Esta é utilizada para fazer com que o *socket* previamente criado se conecte a outro *socket*, tornando a aplicação um cliente.

Aqui, é necessário definir qual é o *socket* remoto, utilizando uma estrutura de dados do tipo *struct sockaddr*.

Para *sockets* da família AF_INET utilizamos *struct sockaddr_in*, enquanto para a família AF_UNIX utilizamos *struct sockaddr_un*.

Exemplo com AF_INET:

connect.c

```
/*
 * connect.c: conecta no loopback utilizando TCP/80
 *
 * Para compilar:
 *  cc -Wall connect.c -o connect
 *
 * Alexandre Fiori
 */
```

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

/* define o destino da conexão, porta 80 em localhost */
#define DESTINO_ADDR "127.0.0.1"
#define DESTINO_PORT 80

int main()
{
    int fd;
    struct in_addr addr;
    struct sockaddr_in sock;
    struct protoent *proto = getprotobyname("tcp");

    /* cria o socket */
    fd = socket(AF_INET, SOCK_STREAM, proto->p_proto);
    if(fd == -1) {
        perror("socket");
        return 1;
    }

    /* limpa as estruturas que serão utilizadas */
    memset(&addr, 0, sizeof(addr));
    memset(&sock, 0, sizeof(sock));

    /* converte determinado endereço para network byte order */
    if(!inet_aton(DESTINO_ADDR, &addr)) {
        fprintf(stderr, "Endereço inválido!\n");
        close(fd); /* é importante fechar o socket! */
        return 1;
    }

    /* preenche a estrutura sockaddr_in: onde vamos conectar */
    sock.sin_family = AF_INET;
    sock.sin_addr   = addr;
    sock.sin_port   = htons(DESTINO_PORT);

    /* conecta */
    if(connect(fd, (struct sockaddr *) &sock, sizeof(sock)) == -1) {
        perror("connect");
        close(fd);
        return 1;
    }

    /* agora estamos conectados em DESTINO_ADDR, DESTINO_PORT utilizando
     * o protocolo TCP */
    fprintf(stdout, "Conectado a %s, %d...\n", DESTINO_ADDR, DESTINO_PORT);
    sleep(1);

    /* desconecta */
    shutdown(fd, 2);
    close(fd);

    return 0;
}
```

5.17.7 As funções *bind()*, *listen()* e *accept()*

Para que a aplicação se torne um servidor, é necessário utilizar as três funções: *bind()*, *listen()* e *accept()*.

Cada uma tem suas próprias características e serão explicadas separadamente.

A função *bind()*

Esta é a função que solicita ao sistema operacional a alocação de uma porta e protocolo para a aplicação. Internamente, no *kernel space*, esta função atualiza a tabela da informações do processo (PID) para que, quando qualquer pacote com tal porta e protocolo chegar no *kernel space* ser transmitido à aplicação.

Para informar o sistema operacional, a função *bind()* também utiliza uma estrutura do tipo *struct sockaddr*, mas aqui esta estrutura tem uma conotação diferente.

Em AF_INET os valores da estrutura *struct sockaddr_in* são utilizados da seguinte maneira:

Tabela 5.16: Campos da estrutura *struct sockaddr_in*

sin_addr	Interface de rede onde a porta deve existir
sin_port	Porta a ser alocada para esta aplicação

Normalmente temos uma ou mais interfaces de rede no sistema. A alocação de uma porta pode ser feita em todas as interfaces, utilizando o valor INADDR_ANY em *sin_addr*, fazendo com que qualquer pacote que chegue ao sistema operacional seja direcionado para a aplicação.

Caso o valor de *sin_addr* seja de um único endereço IP do sistema, a porta só estará disponível na interface de rede com aquele IP.

Exemplo:

```
struct in_addr interface;
struct sockaddr_in s;
...
inet_aton("127.0.0.1", &interface);
...
s.sin_family = AF_INET;
s.sin_addr = interface;
s.sin_port = htons(4040);
...
bind(fd, (struct sockaddr *) &s, sizeof(s));
```

O código acima faz com que a porta 4040 seja disponível *apenas* na interface *loopback*, a qual tem o endereço IP 127.0.0.1.

Se for utilizado *s.sin_addr = htonl(INADDR_ANY)* esta porta estará disponível em qualquer interface de rede do sistema.

Em AF_UNIX os valores em *struct sockaddr_un* são:

Tabela 5.17: Campos da estrutura *struct sockaddr_un*

sun_path	Caminho do arquivo que deve ser criado

Segue um exemplo:

```
struct sockaddr_un s;
...
s.sun_family = AF_UNIX;
snprintf(s.sun_path, sizeof(s.sun_path), "/tmp/mysock");
...
bind(fd, (struct sockaddr *) &s, sizeof(s));
```

Então, quando a função *bind()* for executada o arquivo */tmp/mysock* será criado no disco, para que clientes possam se conectar a ele através de *connect()*.

A função *listen()*

Depois de criar o *socket* e alocar sua porta ou arquivo no sistema operacional, é necessário avisá-lo que a aplicação já está pronta para receber conexões. Para isso temos *listen()*.

Esta função recebe dois parâmetros, sendo que o primeiro é o descritor de arquivo do *socket* e o segundo é chamado de *backlog*.

O *backlog* é utilizado para criar uma fila de chamadas pendentes no sistema operacional, veja:

```
...
listen(fd, 3);
...
```

Assim, caso a aplicação esteja atendendo um cliente e ainda não esteja preparada para receber outro, o sistema operacional mantém até 3 clientes na fila. Os próximos irão receber a mensagem de conexão recusada, como se não houvesse a porta solicitada.

A maioria dos programadores utiliza 5, e é este valor que estará presente nos exemplos.

Vale ressaltar que *listen()* não define a quantidade máxima de clientes a serem atendidos pelo servidor, define apenas uma fila para as conexões que chegam.

A função *listen()* só deve ser utilizada para *sockets* do tipo SOCK_STREAM ou SOCK_SEQPACKET.

A função *accept()*

Depois de tornar a aplicação um servidor, através de *socket()*, *bind()* e *listen()*, é necessário utilizar *accept()* para capturar as conexões que chegam - os clientes que se conectam a seu serviço.

Esta função opera por padrão no modo *blocking*, ou seja, pára o programa enquanto não receber uma conexão. Quando algum cliente se conecta à porta e protocolo previamente definidos por *bind()* então esta função retorna um descritor de arquivo - que é o *socket* do cliente - e ainda preenche uma estrutura do tipo *struct sockaddr*, com os dados do cliente.

Em *sockets* da família AF_UNIX não há necessidade de preencher a *struct sockaddr_un* pois o cliente não tem um endereço de origem. Já nos *sockets* do tipo AF_INET, os valores preenchidos em *sin_addr* e *sin_port* da estrutura *struct sockaddr_in* são o endereço IP e a porta do cliente que se conectou ao servidor.

Para que essa *struct sockaddr* seja preenchida de maneira correta por *accept()* é necessário passar ainda um inteiro *int* com o tamanho em *bytes* da estrutura. Este *int* então será alterado por *accept()* contendo o valor em *bytes* do endereço preenchido - parece complicado mas é simples.

Segue o exemplo de uma aplicação que se torna um servidor:

accept.c

```
/*
 * accept.c: faz da aplicação um servidor e aguarda por clientes
 *
 * Para compilar:
 *  cc -Wall accept.c -o accept
 *
 * Alexandre Fiori
 */

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

```
#include <netinet/in.h>
#include <arpa/inet.h>

/* define a porta onde o servidor deve operar */
#define SERVER_PORT 1905

/* define mensagem a ser enviada para os clientes */
#define SERVER_XMSG "Olá socket!\n"

/* define função para fechar o servidor de maneira correta */
static int server_fd = 0;
static void server_close(int sig)
{
    if(server_fd) close(server_fd);

    fprintf(stdout, "Fechando o servidor...\n");
    exit(0);
}

int main()
{
    int fd, fdc;
    struct sockaddr_in xsrv, xcli;
    struct protoent *proto = getprotobyname("tcp");

    /* cria o socket */
    if((fd = socket(AF_INET, SOCK_STREAM, proto->p_proto)) == -1) {
        perror("socket");
        return 1;
    }

    /* preenche a estrutura xsrv com os dados do servidor */
    memset(&xsrv, 0, sizeof(xsrv));
    xsrv.sin_family      = AF_INET;
    xsrv.sin_addr.s_addr = htonl(INADDR_ANY);
    xsrv.sin_port        = htons(SERVER_PORT);

    /* solicita a porta ao sistema operacional...
     * caso seja uma porta já alocada para outro processo bind()
     * irá retornar -1 */
    if(bind(fd, (struct sockaddr *) &xsrv, sizeof(xsrv)) == -1) {
        perror("bind");
        close(fd);
        return 1;
    }

    /* avisa o sistema operacional que já estamos aptos a receber
     * conexões dos clientes */
    if(listen(fd, 5) == -1) {
        perror("listen");
        close(fd);
        return 1;
    }

    /* prepara a aplicação para receber CTRL-C */
    fprintf(stdout, "Para fechar o servidor pressione CTRL-C\n");
    server_fd = fd;
    signal(SIGINT, server_close);

    /* loop infinito que recebe os clientes */
    for(;;) {
        int len = sizeof(xcli);
        memset(&xcli, 0, len);

        /* aguarda a conexão dos clientes... o programa fica parado
         * aqui até que um cliente se conecte então capturamos o
         * socket do cliente e seus dados */
        fdc = accept(fd, (struct sockaddr *) &xcli, (socklen_t *) &len);
        if(fdc == -1) break;

        /* imprime mensagem no terminal */
        fprintf(stdout, "novo cliente: %s:%d\n",
                inet_ntoa(xcli.sin_addr), ntohs(xcli.sin_port));
```

```
        /* envia mensagem ao cliente */
        send(fdc, SERVER_XMSG, strlen(SERVER_XMSG), 0);

        /* fecha a conexão */
        shutdown(fdc, 2);
        close(fdc);
    }

    return 0;
}
```

Para testar este programa são necessários dois terminais. No primeiro, pode-se executar o servidor e no segundo fazemos a conexão a ele utilizando *telnet* como cliente.

Caso seja possível, execute-os em computadores diferentes, veja:

```
/* computador 1 - servidor */
$ ip addr show eth0 | grep inet
    inet 192.168.0.1/24 brd 192.168.0.255 scope global eth0

$ ./accept
Para fechar o servidor pressione CTRL-C


/* computador 2 - cliente */
$ ip addr show eth0 | grep inet
    inet 192.168.0.2/24 brd 192.168.0.255 scope global eth0

$ telnet 192.168.0.1 1905
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Olá socket!
Connection closed by foreign host.
```

5.17.8 As funções *send()* e *recv()*

Essas funções são utilizadas para enviar e receber dados entre *sockets* previamente conectados (TCP) e funcionam de maneira similar a *write()* e *read()*, respectivamente.

A diferença principal entre elas é que *send()* e *recv()* possuem um campo chamado *flags* que é relacionado às opções do *socket*. Contudo, se o campo *flags* levar o valor 0, elas funcionam exatamente como *write()* e *read()*.

Exemplo:

```
...
/* envio de dados */
char *msgout = "Olá socket";
send(fd, msgout, strlen(msgout), 0);
...
...
/* recebimento de dados */
char msgin[20];
memset(msgin, 0, sizeof(msgin));

recv(fd, msgin, sizeof(msgin), 0);
fprintf(stdout, "mensagem recebida: %s\n", msgin);
...
```

As funções desta família podem gerar um *signal* SIGPIPE caso uma das pontas desconecte do *socket*. Se este *signal* não for tratado com a função *signal()* o programa será terminado.

Uma das maneiras de contornar este problema é avisar o programa para ignorar o SIGPIPE, então as funções irão retornar -1 quando uma das pontas desconectar.

Para ignorá-lo, basta utilizar:

```
...
signal(SIGPIPE, SIG_IGN);
...
```

5.17.9 As funções *sendto()* e *recvfrom()*

Essas funções são semelhantes a *send()* e *recv()*, porém só devem ser utilizadas entre *sockets* não dependentes de sessão (UDP).

A função *sento()* é leva os parâmetros de *connect()* e *send()* ao mesmo tempo, enquanto *recvfrom()* leva os parâmetros de *accept()* e *recv()*.

Veja:

```
...
/* envio de dados */
struct sockaddr_in to;
char *msgout = "Olá socket";
...
sendto(fd, msgout, strlen(msgout), 0,
       (struct sockaddr *) &to, sizeof(to));
...
...
/* recebimento de dados */
socklen_t len;
struct sockaddr_in from;
char msgin[20];

len = sizeof(from);
memset(msgin, 0, sizeof(msgin));
recvfrom(fd, msgin, sizeof(msgin), 0,
         (struct sockaddr *) &from, &len);
...
```

Em *sendto()*, a *struct sockaddr_in* deve ser preenchida antes de enviar os dados, pois lá estarão os dados do destinatário.

Em *recvfrom()* a *struct sockaddr_in* será preenchida durante o recebimento dos dados e terá as informações do remetente.

Embora o exemplo tenha sido formulado com base em *sockaddr_in*, essas funções também funcionam com *sockaddr_un*.

5.18 Técnicas

Durante o desenvolvimento de clientes e servidores há diversas maneiras de lidar com a conexão e o tráfego de dados. Deve-se evitar fluxo desnecessário, manter os *sockets* sincronizados, e os servidores devem estar aptos a controlar diversos clientes simultâneamente.

Apesar de parecer simples, o código de um servidor pode não trabalhar da maneira desejada quando chegam mais de 1000 conexões por segundo. E se forem 10.000? Talvez 100.000?

Portanto, nosso cuidado especial é com eles, os servidores. Já os clientes são mais simples pelo fato lidar apenas com uma ou duas conexões no mesmo processo.

Aqui serão apresentadas técnicas para a criação de aplicações que trafegam desde baixo até alto volume de dados.

5.18.1 A função *getpeername()*

Como já foi dito na Sessão 5.13 os *sockets* são representados por uma estrutura no *kernel space*. Lá estão todas as informações relacionadas à conexão, e no *user space* essa estrutura é identificada por um *int*, um descritor de arquivo.

No *loop* principal de um servidor TCP capturamos os *sockets* dos clientes através de *accept()*, que preenche uma estrutura do tipo *struct sockaddr* com os dados do cliente. Também no UDP, quando recebemos um pacote com *recvfrom*, temos exatamente o mesmo processo, preenchendo uma estrutura *struct sockaddr* com os dados do remetente do datagrama.

Nessas duas funções, *accept()* e *recvfrom()*, o programador pode simplesmente omitir essa estrutura passando NULL como argumento, veja:

```
...
fdc = accept(fd, NULL, 0);
...
recvfrom(fd, buff, sizeof(buff), 0, NULL, 0);
...
```

Posteriormente, em qualquer lugar do código, é possível obter os dados do cliente remoto utilizando *getpeername()*.

Esta função solicita três argumentos: o descritor de arquivo do cliente (*socket*), um estrutura do tipo *struct sockaddr* e um *int* com seu tamanho. Então ela preenche a estrutura e atualiza o tamanho dos dados lá gravados.

Vale lembrar que para *sockets* da família AF_UNIX esta função é dispensável pois os clientes deste tipo de *socket* se conectam a um arquivo portanto não há informações para coletar.

Exemplo:

```
...
struct sockaddr_in cli;
socklen_t len = sizeof(cli);
getpeername(fdc, (struct sockaddr *) &cli, &len);
...
```

5.18.2 A função *getsockname()*

Esta função recebe os mesmos parâmetros da anterior, *getpeername()*, porém preenche uma estrutura *struct sockaddr* com os dados do *socket* local, não remoto.

5.18.3 A função *fcntl()*

Originalmente criada para controlar arquivos, esta função também é útil para capturar e alterar as propriedades associadas aos *sockets*.

Quando criados pela função *socket()*, todos eles possuem uma série de propriedades padronizadas, como por exemplo a opção *blocking*.

Quando o *socket* é do tipo *blocking*, as funções que enviam ou recebem dados deste *socket* como *send()*, *sendto()*, *recv()*, *recvfrom()*, *accept()* e outras, ficam totalmente bloqueadas até que dados sejam transmitidos.

Por exemplo: no código de um servidor TCP, no *loop* principal, a função *accept()* bloqueia o código e só retorna quando um novo cliente se conecta. Em um servidor UDP a função *recvfrom()* faz exatamente a mesma coisa.

Caso o programador deseje executar código enquanto os *sockets* não recebem ou enviam dados, poderá torná-los *non-blocking* utilizando *fcntl()*.

Depois de alterar esta propriedade, qualquer função que ficaria bloqueando a execução do programa irá retornar EAGAIN ou EWOULDBLOCK caso não haja atividade no *socket*[26].

Exemplo:

[26] SOCKET(7) - Linux Programmer's Manual

```
#include <errno.h>
#include <fcntl.h>
...
/* captura as opções atuais */
int old_opts = fcntl(fd, F_GETFL, 0);

/* torna o socket non-blocking */
fcntl(fd, F_SETFL, old_opts | O_NONBLOCK);
...
/* aguarda conexões */
for(;;) {
     fdc = accept(fd, (struct sockaddr *) &cli, &len);
     if(fdc == -1 && (errno == EAGAIN || errno == EWOULDBLOCK))
     ...
     /* executa código */
     ...
}
...
```

5.18.4 A função *setsockopt()*

Esta função permite manipular as propriedades do funcionamento do *socket* e do protocolo utilizado por ele.

Agora veremos algumas opções importantes para configurar os *sockets* adequadamente.

SO_REUSEADDR

É muito comum a função *bind()* retornar o erro EADDRINUSE e *perror()* imprimir a mensagem *Address already in use*.

Isso ocorre quando o programa tenta alocar uma porta que já está alocada para outro processo no *kernel space*, ou quando há conexões no estado de TIME_WAIT com a porta selecionada.

Quando o servidor fecha a conexão com um cliente do tipo TCP, nem sempre o procedimento correto é o que ocorre. Muitas vezes faltam os pacotes que indicam o real fim da conexão, portanto no *kernel space* aquela conexão fica no estado de TIME_WAIT.

Se a opção SO_REUSEADDR for especificada no *socket*, *bind()* só retorna EADDRINUSE quando a porta já está alocada por outro processe. Se houver conexões no estado de TIME_WAIT, *bind()* irá ignorá-las e o *kernel* irá alocar a porta.

Para habilitar esta opção, deve-se utilizar o seguinte:

```
...
int opt = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
...
```

SO_RCVBUF e SO_SNDBUF

Estas opções permitem alterar os valores internos de envio e recebimento de pacotes para o *socket* em questão.

Exemplo:

```
...
int maxsnd = 4096;
int maxrcv = 2024;
setsockopt(fd, SOL_SOCKET, SO_SNDBUF, &maxsnd, sizeof(maxsnd));
setsockopt(fd, SOL_SOCKET, SO_RCVBUF, &maxrcv, sizeof(maxrcv));
...
```

Programas como o *samba* possuem essas opções no arquivo de configuração para permitir ao administrador da rede solucionar problemas de comunicação com *sockets* de outros sistemas operacionais que utilizam valores diferentes no *buffer* de envio e recebimento.

SO_KEEPALIVE

O *keep alive* é uma das falhas fundamentais do TCP. Na teoria ele serve para manter a conexão entre dois *sockets* aberta por um longo período de tempo sem que haja atividade ou tráfego de dados.

Muitas aplicações dependem desta opção, pois ela também permite que a conexão se mantenha ativa caso uma das pontas perca o acesso temporariamente à outra. Isso é causado muitas vezes por problemas com roteamento, regras de firewall e infra-estrutura em geral. Quando o problema é resolvido a conexão pode continuar no estado de ativa, se os *sockets* estão preparados com *keep alive*.

A falha nesse sistema é que se apenas uma das pontas ativar a opção, a outra não irá reconhecer os tais pacotes do *keep alive* que só são enviados em intervalos regulares quando não há atividade entre os *sockets*.

Para ativar o *keep alive* use:

```
...
int opt = 1;
setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &opt, sizeof(opt));
...
```

TCP_NODELAY

Na implementação do protocolo TCP há um algoritmo chamado *Nagle*, que faz com que o *kernel space* crie um *buffer* para os pacotes e só os envie quando houver uma quantidade suficiente de dados.

Esse algoritmo evita que pequenos pacotes sejam enviados frequentemente causando má utilização da rede.

A opção TCP_NODELAY desabilita o algoritmo e faz com que os pacotes sejam enviados o mais rápido possível ao *socket* remoto, independente de seu tamanho.

Exemplo:

```
...
int opt = 1;
setsockopt(fd, SOL_TCP, TCP_NODELAY, &opt, sizeof(opt));
...
```

5.18.5 A função *getsockopt()*

Ao contrário de *setsockopt()*, esta informação captura as propriedades de um *socket* para que o programador tenha conhecimento de suas características em qualquer ponto do programa.

Para saber o tamanho atual do *buffer* de envio, pode usar:

```
...
int size = 0;
socklen_t len = sizeof(opt);
getsockopt(fd, SOL_SOCKET, SO_SNDBUF, &size, &len);
fprintf(stdout, "sndbuf: %d\n", size);
...
```

Para saber se determinada opção está habilitada ou não o procedimento é o mesmo, mas o valor retornado em *size* (do exemplo) será 0 ou 1.

5.18.6 A função *select()*

Esta é uma das principais funções para utilizar com *sockets*. Ela permite que o programador crie um vetor com diversos descritores de arquivo e bloqueia a execução do programa enquanto não houver atividade em algum deles.

É uma função do tipo *blocking*, mas para permitir controle sobre o bloqueio *select()* utiliza uma estrutura do tipo *struct timeval*.

A estrutura *struct timeval* permite que o programador defina o tempo em que *select()* irá aguardar por atividade em um dos descritores de arquivo do vetor.

Ela também é preparada para informar o tipo da atividade, se um dos descritores de arquivo está pronto para receber dados ou se está enviando.

O protótipo desta função é o seguinte:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *utimeout);
```

É importante lembrar que os descritores de arquivo são estruturas no *kernel space*, e no *user space* são números *int* que atuam como índices entre esses dois diferentes ambientes.

Portanto, um arquivo aberto com *open()* ou um *socket()* são números. Por padrão, os descritores de arquivo 0, 1 e 2 são conhecidos como *stdin*, *stdout* e *stderr*. Ao abrir um novo arquivo ou *socket*, o número atribuído a ele tem 99% de chances de ser 3, e assim sucessivamente.

A primeira opção, *nfds* deve ser indica com o número do maior *socket* que se deve monitorar em algum dos vetores do *readfds*, *writefs* ou *exceptfs*, mais um.

Sim, são vetores. Este tipo *fd_set* é uma estrutura que o programador não precisa manipular diretamente, pois *select()* conta com um conjunto de macros para manipular seus vetores.

Seguem as macros:

FD_CLR(int fd, fd_set *set) é utilizada para remover o descritor de arquivo *fd* do vetor *set*.

FD_ISSET(int fd, fd_set *set) retorna 1 quando há atividade no descritor de arquivo *fd* previamente adicionado no vetor *set*.

FD_SET(int fd, fd_set *set) adiciona o descritor de arquivo *fd* no vetor *set*.

FD_ZERO(fd_set *set) limpa o vetor *set* e deve ser utilizada antes de qualquer outra operação com ele.

A estrutura *struct timeval* tem apenas dois campos, sendo *tv_sec* para indicar segundos e *tv_usec* para indicar microsegundos.

Imagine-se programando uma aplicação servidor e recebendo diversos clientes com a função *accept()*. É simples adicionar cada *fdc* retornado por ela a um vetor do tipo *fd_set* utilizando a macro FD_SET().

Depois, é simples monitorar atividade de todos os *sockets* dos clientes conectados utilizando *select()* e passando um vetor *fd_set* apenas no *readfds*.

Para clarear as idéias, segue um programa bem simples que monitora atividade no terminal, veja:

select.c

```
/*
 * select.c: aguarda por atividade no stdin durante 5 segundos e informa
 *           o resultado: sim ou não
 *
 * Para compilar:
 *  cc -Wall select.c -o select
 *
 * Alexandre Fiori
 */

#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>

int main()
{
    int fd = 0; /* stdin, mas poderia ser um socket */

    int ret;
    fd_set fds;
    struct timeval tv;

    /* limpa o vetor fds */
    FD_ZERO(&fds);

    /* adiciona fd ao vetor fds */
    FD_SET(fd, &fds);

    /* especifica 5 segundos em tv */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    ret = select(fd+1, &fds, NULL, NULL, &tv);
    if(ret == -1)
        perror("select");
    else
    if(ret && FD_ISSET(fd, &fds)) {
        char temp[128];
        memset(temp, 0, sizeof(temp));

        read(fd, temp, sizeof(temp));
        fprintf(stdout, "atividade no fd: %s\n", temp);
    }
    else
        fprintf(stdout, "nenhuma atividade em 5 segundos.\n");

    return 0;
}
```

Executando:

```
$ ./select
nenhuma atividade em 5 segundos.

$ ./select
teste <- texto digitado no terminal!
atividade no fd: teste
```

Um detalhe importante sobre esta função é que quando utilizada com *sockets* ela pode retornar positivo indicando que há atividade no descritor mas a função subsequente, normalmente *recv()*, pode bloquear a execução. Isso dá-se pelo fato de que um pacote pode chegar e *select()* irá retornar positivo, mas se esse pacote estiver corrompido ou algo do tipo, *recv()* irá bloquear porque os dados serão descartados no *kernel space*. Portanto, é aconselhável utilizar *select()* com *sockets* do tipo *non-blocking*.

Também é possível utilizar *select()* sem determinar a quantidade de tempo para aguardar por atividade, ignorando a *struct timeval* e passando NULL como argumento em seu lugar.

5.18.7 A função *fork()*

Muitos programadores optam por utilizar *sockets* do tipo padrão, *blocking*. Dependendo do tipo da aplicação essa técnica é válida, porém consome mais recursos do sistema operacional pois cada cliente que chega ao servidor possui um novo processo para atendê-lo.

A função *fork()* é utilizada para criar processos filhos, conhecidos como *child*.

No terminal é fácil identificar a hierarquia dos processos através do comando abaixo:

```
$ ps -e f
```

O servidor baseado em *fork()* normalmente tem uma função que é definida para atender um único *socket*, porém no *loop* principal logo após o *accept()*, um novo *child* é criado e a função que atende o *socket* é chamada.

Veja:

forksrv.c

```
/*
 * forksrv.c: faz da aplicação um servidor e aguarda por clientes, que serão
 *            atendidos em processos filhos
 *
 * Para compilar:
 *  cc -Wall forksrv.c -o forksrv
 *
 * Alexandre Fiori
 */

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

/* define a porta onde o servidor deve operar */
#define SERVER_PORT 1910

/* define mensagem a ser enviada para os clientes */
#define SERVER_XMSG "Olá socket!\n"

/* define função para fechar o servidor de maneira correta */
static int server_fd = 0;
static void server_close(int sig)
{
    if(server_fd) close(server_fd);

    fprintf(stdout, "Fechando o servidor...\n");

    /* desabilita o stdout para não replicar as mensagens de saída,
     * pois todos os childs executam esta função quando CTRL-C
     * é pressionado */
    close(1);

    /* fecha o processo atual */
    exit(0);
}

/* função para atender cada socket de cliente */
static void client(int fd)
{
    /* envia mensagem ao cliente */
    send(fd, SERVER_XMSG, strlen(SERVER_XMSG), 0);
```

```
    /* caso queira ver os processos criados para cada cliente
     * tire o comentário da linha abaixo... */
    /* sleep(10); */

    /* fecha a conexão */
    shutdown(fd, 2);
    close(fd);

    /* fecha o child */
    _exit(0);
}

int main()
{
    int fd, fdc;
    struct sockaddr_in xsrv, xcli;
    struct protoent *proto = getprotobyname("tcp");

    /* cria o socket */
    if((fd = socket(AF_INET, SOCK_STREAM, proto->p_proto)) == -1) {
        perror("socket");
        return 1;
    }

    /* preenche a estrutura xsrv com os dados do servidor */
    memset(&xsrv, 0, sizeof(xsrv));
    xsrv.sin_family      = AF_INET;
    xsrv.sin_addr.s_addr = htonl(INADDR_ANY);
    xsrv.sin_port        = htons(SERVER_PORT);

    /* solicita a porta ao sistema operacional...
     * caso seja uma porta já alocada para outro processo bind()
     * irá retornar -1 */
    if(bind(fd, (struct sockaddr *) &xsrv, sizeof(xsrv)) == -1) {
        perror("bind");
        close(fd);
        return 1;
    }

    /* avisa o sistema operacional que já estamos aptos a receber
     * conexões dos clientes */
    if(listen(fd, 5) == -1) {
        perror("listen");
        close(fd);
        return 1;
    }

    /* prepara a aplicação para ignorar os sinais
     * de controle de child */
    signal(SIGCHLD, SIG_IGN);

    /* prepara a aplicação para receber CTRL-C */
    fprintf(stdout, "Para fechar o servidor pressione CTRL-C\n");
    server_fd = fd;
    signal(SIGINT, server_close);

    /* loop infinito que recebe os clientes */
    for(;;) {
        int pid, len = sizeof(xcli);
        memset(&xcli, 0, len);

        /* aguarda a conexão dos clientes... o programa fica parado
         * aqui até que um cliente se conecte então capturamos o
         * socket do cliente e seus dados */
        fdc = accept(fd, (struct sockaddr *) &xcli, (socklen_t *) &len);
        if(fdc == -1) break;

        /* cria novo processo para atender o cliente */
        switch((pid = fork())) {
            case -1:
                perror("fork");
                exit(1);
            case 0:
```

```
                client(fdc);
                break;
        }

        /* imprime mensagem no terminal */
        fprintf(stdout, "novo cliente (pid %d): %s:%d\n",
                pid, inet_ntoa(xcli.sin_addr), ntohs(xcli.sin_port));
    }

    return 0;
}
```

O procedimento para teste é semelhante ao de *accept()*, na Sessão 5.17.7 - mas aqui a porta é 1910 ao invés de 1905.

5.18.8 A função *daemon()*

Os *daemons* são programas que não possuem interface com o usuário, não imprimem mensagens no terminal nem ficam presos a ele. Para fazer com que o processo execute em segundo plano, *background*, basta usar a função *daemon()*[27] seguida de *setsid()*[28] - responsável por criar uma nova sessão para o processo desanexando-o do terminal onde foi executado.

Exemplo:

```
...
daemon(0, 0);
setsid();
...
```

5.18.9 A função *sendfile()*

Deve ser utilizada para transferir dados entre dois descritores de arquivo.

Segue o protótipo da função[29]:

```
#include <sys/sendfile.h>

ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

Esta função necessita alguns argumentos, sendo *out_fd* um descritor de arquivo com permissão de escrita e *in_fd* um descritor de arquivo com permissão de leitura.

O *offset* deve ser um inteiro indicando à partir de qual *byte* deve-se ler *in_fd* e *count* a quantidade de *bytes* para transmitir.

Esta função será utilizada posteriormente como exemplo.

5.19 Aplicações reais

Até agora, diversas funções foram mencionadas, bem como alguns exemplos. O fato é que nem todas elas apareceram em aplicações reais.

A partir daqui teremos algumas aplicações de rede utilizando muito do que já foi explicado.

[27] DAEMON(3) - Linux Programmer's Manual
[28] SETSID(2) - Linux Programmer's Manual
[29] SENDFILE(2) - Linux Programmer's Manual

5.19.1 icmpd: recebendo pacotes puros (*raw packets)*

A aplicação seguinte tem como objetivo mostrar a você que é possível trafegar estruturas de dados inteiras por *send()*, *sendto()*, *recv()* e *recvfrom()*.

Basta criar uma estrutura com os dados que deseja enviar ou receber e passá-la com seu tamanho para uma das funções mencionadas.

Embora não seja obrigatório, pacotes puros serão utilizados para coletar informações do protocolo ICMP e mostrá-las na tela.

Esta aplicação captura PING.

5.19.1.1 Código fonte

O código é simples e enxuto, veja:

icmpd.c

```
/*
 * icmpd.c: captura PING e mostra no terminal
 *
 * Para compilar:
 *  cc -Wall icmpd.c -o icmpd
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

/* define a estrutura do pacote:
 * pelo fato de 'data' ter apenas 4096 bytes, esta aplicação
 * não é capaz de capturar pacotes maiores que isso */
typedef struct {
    struct iphdr ip;
    struct icmphdr icmp;
    char data[4096];
} packet;

int main()
{
    int r, fd;
    socklen_t len;
    struct sockaddr_in in;
    struct protoent *proto = getprotobyname("icmp");
    packet pkt;

    /* cria o socket */
    if((fd = socket(AF_INET, SOCK_RAW, proto->p_proto)) == -1) {
        perror("socket");
        return 1;
    }

    fprintf(stdout, "Aguardando pacotes...\n");

    /* loop principal */
    for(;;) {
        len = sizeof(pkt);
        memset(&pkt, 0, sizeof(pkt));

        /* recebe o pacote inteiro, com os cabeçalhos */
```

```
            r = recvfrom(fd, &pkt, sizeof(pkt), 0,
                    (struct sockaddr *) &in, &len);
            if(r == -1) {
                perror("recvfrom");
                close(fd);
                return 1;
            }

            /* descarta pacotes que não foram do tipo ICMP ECHO */
            if(pkt.icmp.type != ICMP_ECHO) continue;

            /* calcula o tamanho real do pacote,
             * sem o cabeçalho IP: como faz o ping */
            r -= sizeof(pkt.ip);

            /* imprime a informação no terminal */
            fprintf(stdout, "ICMP de %s com %d bytes e sequência %d.\n",
                    inet_ntoa(in.sin_addr), r, pkt.icmp.un.echo.sequence);
        }

        return 0;
    }
```

5.19.1.2 Notas

Como você pode ver, o sistema operacional repassa o pacote completo quando utilizamos SOC_RAW - menos o *ethernet frame*.

Para executar este programa é necessário fazê-lo com o usuário *root*, pois só ele é quem pode criar *sockets* do tipo SOCK_RAW.

Quando ele estiver executando, vá em outro computador e envie um *ping* para ele e veja esta informação na tela. Também é possível fazer *ping localhost* e capturar os pacotes do *loopback*.

Vale a pena enviar um *ping* com o tamanho do pacote modificado, utilizando a opção *-s*.

5.19.2 multid/multisend: recebendo e enviando *multicast*

O *multicast* é um tipo de pacote UDP que utiliza endereços IP da classe D. As aplicações que desejam receber esses pacotes devem se tornar membras de um grupo *multicast*, que nada mais é que um endereço IP.

Para enviar pacotes para grupos *multicast* basta enviar pacotes UDP comuns.

5.19.2.1 multid.c: *daemon* que recebe mensagens *multicast*

A aplicação que será apresentada aqui não é literalmente um *daemon* pois fica presa ao terminal imprimindo mensagens. Para que ela se torne um, é necessário utilizar a função *daemon()* seguida de *setsid()*, apresentadas na Sessão 5.18.8.

Para fazer com que a aplicação pertença a um grupo *multicast* utilizamos *setsockopt()* com uma estrutura de dados do tipo *struct ip_mreq*.

Código do servidor:

multid.c

```
/*
 * multid.c: daemon que aguarda pacotes multicast e os imprime no terminal
 *
 * Para compilar:
 *  cc -Wall multid.c -o multid
 *
 * Alexandre Fiori
```

```
 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MULTI_PORT  2020
#define MULTI_GROUP "225.0.0.1"

int main()
{
    int r, fd;
    socklen_t len;
    struct sockaddr_in s, in;
    struct ip_mreq req;
    char temp[4096];

    /* define as propriedades do servidor */
    memset(&s, 0, sizeof(s));
    s.sin_family      = AF_INET;
    s.sin_addr.s_addr = htonl(INADDR_ANY);
    s.sin_port        = htons(MULTI_PORT);

    /* cria o socket UDP */
    if((fd = socket(s.sin_family, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        return 1;
    }

    /* solicita a porta ao sistema operacional */
    if(bind(fd, (struct sockaddr *) &s, sizeof(s)) == -1) {
        perror("bind");
        close(fd);
        return 1;
    }

    /* cria o grupo multicast */
    req.imr_multiaddr.s_addr = inet_addr(MULTI_GROUP);
    req.imr_interface.s_addr = htonl(INADDR_ANY);
    if(setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            &req, sizeof(req)) == -1) {
        perror("setsockopt");
        close(fd);
        return 1;
    }

    fprintf(stdout, "Aguardando mensagens...\n");

    /* loop principal */
    for(;;) {
        len = sizeof(in);

        memset(temp, 0, sizeof(temp));
        r = recvfrom(fd, temp, sizeof(temp)-1, 0,
                (struct sockaddr *) &in, &len);
        if(r == -1) {
            perror("recvfrom");
            close(fd);
            return 1;
        }

        fprintf(stdout, "Mensagem de %s:%d com %d bytes:\n"
                        "%s\nFim da mensagem\n\n",
                inet_ntoa(in.sin_addr), ntohs(in.sin_port),
                r, temp);
    }

    return 0;
}
```

5.19.2.2 multisend.c: envia mensagens UDP

Este programa envia mensagens na forma de pacote UDP para qualquer endereço, portanto pode ser utilizado para se comunicar com o servidor *multicast*.

Código do cliente:

multisend.c

```
/*
 * multisend.c: envia mensagens texto para um grupo multicast
 *
 * Para compilar:
 *  cc -Wall multisend.c -o multisend
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MULTI_PORT  2020
#define MULTI_GROUP "225.0.0.1"

int main(int argc, char **argv)
{
    int fd;
    struct in_addr addr;
    struct sockaddr_in s;

    if(argc != 4) {
        fprintf(stderr, "use: %s grupo porta mensagem\n"
                        "exemplo: %s 225.0.0.1 2020 \"teste multicast\"\n",
                        *argv, *argv);
        return 1;
    }

    /* define as propriedades do destino da mensagem */
    if(inet_aton(argv[1], &addr) == -1) {
        fprintf(stderr, "Endereço %s inválido!\n", argv[1]);
        return 1;
    }

    memset(&s, 0, sizeof(s));
    s.sin_family = AF_INET;
    s.sin_addr   = addr;
    s.sin_port   = htons(atoi(argv[2]));

    /* cria o socket UDP */
    if((fd = socket(s.sin_family, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        return 1;
    }

    /* envia a mensagem */
    sendto(fd, argv[3], strlen(argv[3]), 0,
                (struct sockaddr *) &s, sizeof(s));

    close(fd);
    return 0;
}
```

5.19.2.3 Notas

Em redes roteadas há certa dificuldade de trafegar pacotes *multicast* porque quando uma aplicação passa a pertencer a um grupo, ela precisa avisar os clientes. Portanto, exige um sistema de roteamento dinâmico.

Os *switches* também devem suportar o ambiente *multicast*.

Em redes locais funcionam perfeitamente e podem fazer milagres: imagine uma aplicação que lê um DVD e envia o audio e video para um grupo multicast na rede. De outro lado, um servidor que aguarda pacotes *multicast*, passa o vídeo na tela e toca o audio.

Agora imagine 200 computadores com esse servidor e todos recebendo um único pacote com os dados do DVD.

5.19.3 minihttpd.c: mini servidor HTTP *non-blocking*

Este mini servidor HTTP é apenas para fins educativos e não segue por completo o RFC2606.

Para compreendê-lo é necessário conhecer um mínimo sobre o protocolo HTTP e ter alguma experiência com outros servidores, como o *apache*.

Ele possui apenas a implementação do modo GET e alguns *mime-types* como *text/plain*, *text/html*, *image/jpg* e *image/png*.

O código é enxuto e suporta diversos clientes simultâneos realizando requisições diferentes graças ao *select()* e todos os *sockets non-blocking*.

5.19.3.1 Ambiente do mini servidor

O *DocumentRoot* do servidor é definido no código para o diretório */tmp/minihttpd*, então lá devem estar os arquivos HTML e imagens disponíveis para os clientes.

A porta padrão é definida no código para 1908 ao invés de 80, portanto não esqueça de colocá-la na URL para acessar o servidor:

```
http://mini-servidor:1908/
```

5.19.3.2 Código fonte

Segue o código do minihttpd:

minihttpd.c

```
/*
 * minihttpd.c: torna a aplicação um mini web server utilizando select()
 *              para tratar todos os clientes em um único processo
 *
 * NOTA: esta implementação não está totalmente baseada no RFC2606
 *
 * Para compilar:
 *  cc -Wall minihttpd.c -o minihttpd
 *
 * Alexandre Fiori
 */

#define _GNU_SOURCE
#include <time.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
```

```
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/sendfile.h>
#include <arpa/inet.h>

#define WWW_ROOT "/tmp/minihttpd" /* DocumentRoot */
#define WWW_INDX "index.html"     /* Indexes */
#define WWW_PORT 1908             /* HTTP Port */
#define WWW_MAXB 4096             /* MAX read buffer */
#define WWW_E404 "Arquivo não encontrado no servidor."

/* funções */
static void  setnonblock(int fd);
static int   handle_request(int fd);
static char *mimetype(const char *filename, char *mime, int mime_len);
static void  finish(int sig);

int main()
{
    int i, fd, fdc, ndfs;
    struct sockaddr_in s;
    fd_set mset, rset;

    /* define as propriedades do servidor */
    memset(&s, 0, sizeof(s));
    s.sin_family      = AF_INET;
    s.sin_addr.s_addr = htonl(INADDR_ANY);
    s.sin_port        = htons(WWW_PORT);

    /* cria socket para o servidor (TCP) */
    if((fd = socket(s.sin_family, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return 1;
    } else {
        int opt = 1;
        setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    }

    /* aloca a porta */
    if(bind(fd, (struct sockaddr *) &s, sizeof(s)) == -1) {
        perror("bind");
        close(fd);
        return 1;
    }

    /* informa o tamanho da fila */
    if(listen(fd, 5) == -1) {
        perror("listen");
        close(fd);
        return 1;
    }

    /* zera mset e rset para select() */
    FD_ZERO(&mset);
    FD_ZERO(&rset);

    /* torna o servidor non-block e adiciona o socket no mset */
    setnonblock(fd);
    FD_SET(fd, &mset);
    ndfs = fd;

    /* configura os sinais */
    signal(SIGPIPE, SIG_IGN);
    signal(SIGTERM, finish);
    signal(SIGINT,  finish);
    fprintf(stdout, "Aguardando conexões TCP na porta %d...\n", WWW_PORT);
```

```
    /* loop principal */
    for(;;) {
        rset = mset; /* guarda cópia */

        /* aguarda atividade em qualquer socket */
        if(select(ndfs+1, &rset, NULL, NULL, NULL) == -1) {
            perror("select");
            close(fd);
            return 1;
        }

        /* checa por atividade em todos os sockets disponíveis */
        for(i = fd; i <= ndfs; i++) {
            if(FD_ISSET(i, &rset)) {
                /* caso seja o socket do servidor, chama accept() */
                if(i == fd) {
                    fdc = accept(fd, NULL, 0);
                    if(fdc == -1) {
                    /* ignora os erros de accept() */
                        perror("accept");
                    } else {
                        /* adiciona o cliente select() */
                        setnonblock(fdc);
                        FD_SET(fdc, &mset);
                        if(fdc > ndfs) ndfs = fdc;
                    }
                }
                else
                /* caso seja o socket de um cliente,
                 * interpreta a requisição HTTP */
                if(handle_request(i) == -1) {
                    close(i);
                    FD_CLR(i, &mset);
                }
            }
        }
    }

    return 0;
}

/* torna o socket non-blocking */
static void setnonblock(int fd)
{
    int opts = fcntl(fd, F_GETFL, 0);
    fcntl(fd, F_SETFL, opts | O_NONBLOCK);
}

/* lida com a requisição do cliente */
static int handle_request(int fd)
{
    int nl;
    socklen_t len;
    struct sockaddr_in in;
    char temp[WWW_MAXB], *lines[sizeof(temp)/2];

    /* lê a requisição */
    memset(temp, 0, sizeof(temp));
    if(recv(fd, temp, sizeof(temp)-1, 0) <= 0) {
        if(errno == EAGAIN || errno == EWOULDBLOCK)
            return 0;
        else
            return -1;
    }

    /* obtem as informações do cliente */
    len = sizeof(in);
    getpeername(fd, (struct sockaddr *) &in, &len);

    /* quebra a string temp trocando \r ou \n por \0 e mantendo uma
     * matriz de ponteiros para cada início de linha */
    int parse(char *buff, char **result) {
```

```
    int count = 0;
    while(*buff != '\0') {
        while((*buff == '\r' || *buff == '\n') && *buff != '\0')
            *buff++ = '\0';

        *result++ = buff;
        count++;

        while((*buff != '\r' && *buff != '\n') && *buff != '\0')
            buff++;
    }

    return count;
}

/* imprime mensagem no terminal */
void log(char *file, char *result) {
    time_t t = time(NULL);
    struct tm *tm = localtime(&t);

    if(!tm) {
        perror("localtime");
        return;
    }

    fprintf(stdout,
        "%04d.%02d.%02d %02d:%02d:%02d "
        "[%s:%d] GET %s - %s\n",
        tm->tm_year + 1900, tm->tm_mon, tm->tm_mday,
        tm->tm_hour, tm->tm_min, tm->tm_sec,
        inet_ntoa(in.sin_addr), ntohs(in.sin_port),
        file, result);
}

/* envia erro 404 */
int send404(int fdc, char *file) {
    dprintf(fdc, "HTTP/1.0 404 Not Found\n"
            "Content-type: text/plain\n"
            "Content-Length: %d\n"
            "Server: minihttpd\n"
            "Connection: Close\n\n%s",
            strlen(WWW_E404), WWW_E404);
    log(file, "404 Not Found");
    return -1;
}

nl = parse(temp, lines);

/* trata a requisição e envia o arquivo solicitado em GET */
{
    fd_set fds;
    struct stat st;
    int r, xfd, v1, v2, offset = 0;
    char mime[128], filename[WWW_MAXB], *path;

    memset(mime, 0, sizeof(mime));
    memset(filename, 0, sizeof(filename));

    sscanf(lines[0], "GET %s HTTP/%d.%d", filename, &v1, &v2);
    if(!strncmp(filename, "/", sizeof(filename)))
        path = WWW_INDX;
    else
        path = filename;
    path = mimetype(path, mime, sizeof(mime));

    if(stat(path, &st) == -1)
        return send404(fd, filename);
    else
    if(S_ISDIR(st.st_mode))
        return send404(fd, filename);

    if((xfd = open(path, O_RDONLY)) == -1)
        return send404(fd, filename);
```

```
        else
            log(filename, "200 OK");

        dprintf(fd, "HTTP/1.0 200 OK\n"
                    "Content-type: %s\n"
                    "Content-Length: %d\n"
                    "Server: minihttpd\n"
                    "Connection: Keep-Alive\n\n",
                    mime, (int) st.st_size);
        do {
            FD_ZERO(&fds);
            FD_SET(fd, &fds);
            if(select(fd+1, 0, &fds, 0, 0) == -1)
                break;
            else /* envia o arquivo por partes */
                r = sendfile(fd, xfd, (off_t *) &offset, st.st_size);

            if(r == -1 || !r) break;
        } while(offset != st.st_size);

        close(xfd);
    }

    return 1;
}

/* define o mime-type e retorna o caminho correto para o arquivo
 * solicitado na requisição */
static char *mimetype(const char *filename, char *mime, int mime_len)
{
    typedef struct {
        char *extension;
        char *mime_type;
    } MIME;

    /* o padrão é text/plain */
    static MIME mimelist[] = {
        { "txt",  "text/plain" },
        { "jpg",  "image/jpg"  },
        { "png",  "image/png"  },
        { "html", "text/html"  },
    };

    char *p;
    int i, found = 0;
    static char path[WWW_MAXB];
    const static int mimelist_len = sizeof(mimelist)/sizeof(mimelist[0]);

    /* escreve o caminho completo para o arquivo, dentro de WWW_ROOT */
    memset(path, 0, sizeof(path));
    snprintf(path, sizeof(path), "%s/%s", WWW_ROOT, filename);

    /* descobre o mime-type */
    for(i = 0; i < mimelist_len; i++) {
        if((p = strstr(filename, "."))) {
            if(p[1] != '\0') p++;
            if(!strncmp(p, mimelist[i].extension, strlen(p))) {
                strncpy(mime, mimelist[i].mime_type, mime_len);
                found = 1;
                break;
            }
        }
    }

    if(!found)
        strncpy(mime, mimelist[0].mime_type, mime_len);

    return path;
}

/* finaliza o servidor */
static void finish(int sig)
{
```

```
        fprintf(stdout, "Fechando o servidor...\n");
        exit(0);
    }
```

5.19.3.3 Testando conexões simultâneas no mini servidor

Abrir um navegador como o *Firefox* acessar a URL do servidor não é suficiente para saber se ele é estável.

Para realizar o *stress-test* utilizei um programa da *Jakarta*[30] feito em Java(TM) chamado *Jmeter*.

Com ele é possível criar procedimentos de teste que simulam diversos navegadores conectando no servidor e solicitando documentos diferentes.

No teste que realizei, criei 400 *threads* realizando 3 requisições aleatórias em *loop* durante 3 horas, e nas estatísticas do *Jmeter* o servidor não apresentou nenhum erro e atendeu a todas as conexões.

Durante esse teste ainda vinha ao *Firefox* e conectava no servidor, que respondia normalmente.

Pude concluir que para documentos básicos como HTML e imagens ele funciona, mas seu comportamento pode ser diferente com arquivos MP3 e outros, pois durante o envio de grandes arquivos por *sendfile()*, o servidor não atende outros clientes.

[30] Página oficial da Jakarta - http://jakarta.apache.org

Capítulo 6

Acesso a Banco de Dados

Banco de dados é definido como uma coleção de informações armazenadas nos computadores de maneira sistemática, possibilitando consulta para obter respostas.

Os programas que fazem a função de banco de dados devem permitir o gerenciamento dos dados bem como as pesquisas e são conhecidos como *Database Management Systems (DBMS)*.

Hoje há diversos tipos de bancos de dados gratuitos, código fonte aberto.

A grande maioria dos bancos de dados são desenvolvidos em linguagem C ou C++ pela eficiência e rapidez da linguagem. Sendo assim, todos eles disponibilizam bibliotecas com grupos de funções para gerenciar e acessar seu conteúdo.

Em linguagens interpretadas como *python*, *php* ou *perl*, o acesso interno ao banco de dados é sempre feito pela API de programação da linguagem C, pois o interpretador dessas linguagens é sempre feito em C ou C++.

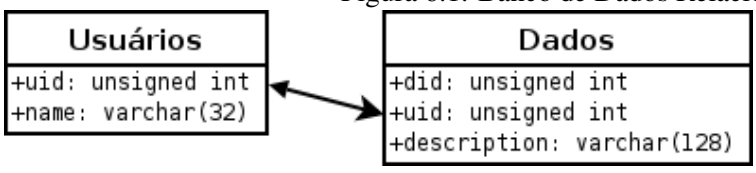
Os bancos de dados mais modernos permitem o relacionamento de dados e são conhecidos como *Relational Database Management Systems (RDBMS)*. O relacionamento dá-se pela maneira de organizar as informações e obter as respostas.

Podemos, por exemplo, criar uma área no banco de dados onde haja registros de usuários, e cada usuário deve possuir um número de identificação. Em outra área, temos dados. Cada dado pertence a um usuário, portanto naquela área dos dados ao invés de informar o nome do usuário deve-se apenas informar seu número.

Ao realizar uma pesquisa nesse tipo de banco de dados relacional, é possível obter todos os dados da área de dados e ao invés do número do usuário, seu nome, relacionando o tal número com o nome na área do banco onde estão os registros de usuários.

Segue o diagrama UML deste exemplo:

Figura 6.1: Banco de Dados Relacional



Essas áreas para armazenar dados nos bancos de dados são conhecidas como tabelas. No diagrama acima temos duas tabelas, sendo uma para usuários e outra para dados.

Os registros de usuários sempre estão na tabela “Usuários” e cada um tem um número de identificação.

Sempre que adicionarmos informações na tabela “Dados”, essas informações devem ser relacionadas a algum usuário, portanto o campo “uid” da tabela “Dados” deve ter um número relativo ao campo “uid” da tabela “Usuários”.

É relativamente simples.

A maioria dos bancos de dados atuais utiliza a mesma linguagem para gerenciar as informações, seu nome é *Structured Query Language (SQL)* - pronuncia-se "sequel".

A linguagem SQL provê interface para bancos de dados relacionais e foi padronizada pelo ISO e ANSI anos após sua criação na década de 1970 pela IBM - com intuito de gerenciar o *System R*.

Para o bom aproveitamento do conteúdo a seguir você deve conhecer pelo menos os três comandos principais desta linguagem: INSERT, DELETE e SELECT.

6.1 Bancos de Dados gratuitos

Hoje temos dois pilares: MySQL e PostgreSQL. Ambos são excelentes RDBMSs e são distribuídos com o código fonte aberto para que mais pessoas conheçam e trabalhem neles.

Ambos possuem API de programação para linguagem C e C++ e os grupos de funções são bem simples e diretos. Não há camadas para acessar os bancos de dados pois a biblioteca da API de programação em C ou C++ sempre acessa o banco por *Unix Domain Sockets* ou *Internet Sockets*, AF_UNIX ou AF_INET, respectivamente - vide Sessão 5.14.

Ambos possuem um terminal de acesso ao banco baseado na linguagem SQL onde o administrador pode gerenciar as informações, permissões, desempenho e até realizar pesquisas. O terminal de acesso utiliza a GNU *Readline* - assim como o *shell bash*.

Quando devenvolvemos uma aplicação que se conecta ao banco de dados, utilizamos exatamente as mesmas funções que foram utilizadas no desenvolvimento do terminal de acesso, ferramenta provida pelo próprio banco de dados.

As linguagens interpretadas como *python*, *php* e *perl* também utilizam essas mesmas funções.

Então, podemos concluir que estamos trabalhando no mais baixo nível para acessar os bancos de dados - a API de programação C ou C++.

Todo código a seguir é baseado em MySQL 5.0 e PostgreSQL 8.1.

6.2 MySQL

Começou de uma necessidade. Um grupo de administradores de rede e programadores teve a intenção de usar o *Mini SQL (mSQL)* para conectar algumas tabelas utilizando um algoritmo próprio, muito rápido, chamado ISAM.

Depois de alguns testes se deram conta de que o mSQL não rápido nem flexivel o bastante para suas necessidades e isso resultou em uma nova interface SQL para aquelas tabelas com uma API muito semelhante à do mSQL.

A nova API era desenhada para suportar código externo escrito para mSQL e facilmente seria portado para MySQL.

Naquele sistema antigo do mSQL todas as ferramentas e bibliotecas que eram desenvolvidas por esse grupo tinham o prexifo "my" no nome, e assim funcionaram por mais de 10 anos.

O co-fundador do projeto, Monty Widenius foi um dos que sugeriu o nome - apesar de ter uma filha chamada My.

Ninguém sabe muito bem se o nome veio das ferramentas e bibliotecas ou por causa da filha dele.

O fato é que hoje o MySQL é um dos bancos de dados mais populares e rápidos, utilizado largamente em diversos tipos sistemas.

Sua API de programação em C é simples, direta e bem documentada. Existem alguns procedimentos para fazer a conexão com o banco de dados e outros para realizar operações como inserção de dados e pesquisas.

6.3 PostgreSQL

Em 1986 na Universidade de Berkeley na Califónia o Professor Michael Stonebraker iniciou o projeto POSTGRES, patrocinado pela Agência de Pesquisa de Projetos Avançados de Defesa (DARPA), Escritório de Pesquisa do Exército (ARO), Fundação Nacional de Ciência (NSF) e a empresa ESL, Inc.

Na fase inicial do projeto foram preparados documentos especificando o desenho do banco de dados e os modelos de armazenamento de dados e modelos de regras.

A primeira versão operacional do POSTGRES só veio um ano depois, em 1987 e logo em seguida foi apresentada na conferência ACM-SIGMOD em 1988.

A primeira versão oficial do banco de dados só foi lançada em junho de 1989 e distribuída para poucos usuários que logo criticaram os modelos de regras do projeto original, fazendo com que em junho de 1990 viesse a segunda versão.

Naquela época o POSTGRES já era usado em diversos ambientes de pesquisa e produção, como sistemas de análise financeira, monitoramento de performance de turbinas, banco de dados de informações de asteróides, banco de dados de informações médicas e muitos outros sistemas de informação geográfica. Também havia se tornado ferramenta de ensino em Universades até que a empresa Illustra Information Technologies pegou o código e começou a comercializar no *Informix*, posteriormente adquirido pela IBM.

Em 1993 a quantidade de pessoas que usavam o POSTGRES já era grande e como consumia muito tempo da Universidade para manter o projeto, ele teve fim oficial naquele mesmo ano na versão 4.2.

Em 1994 os programadores Andrew Yu e Jolly Chen trocaram a linguagem PostQUEL por SQL e passaram a lançar novas versões na *Internet* com o nome de Postres95 - já com código fonte aberto, compilado com GNU *make* ou invés de BSD *make*, a ferramenta *psql* para acesso interativo (terminal de acesso), bibliotecas para C e TCL além de outros menores detalhes e correções.

O novo código era 100% ANSI C e 25% menor que o POSTGRES original, além de 30-50% mais rápido com diversas rotinas reescritas.

Em 1996 o nome Postgres95 não era mais adequado e projeto tomou rumo com o nome PostgreSQL, deixando claro o nome da linguagem de gerenciamento do banco.

Até hoje o PostgreSQL é um dos bancos de dados mais modernos, robustos e eficientes com código aberto, no mundo.

A API de programação chama-se *libpq* para C e *libpqxx* para C++. Ambas são simples e possuem todas as funções para interagir com o banco de dados.

6.4 Criação do ambiente de laboratório

6.4.1 MySQL

Em primeiro lugar deve-se instalar o MySQL. A instalação varia de uma distribuição para outra mas em todas os pacotes binários possuem nomes semelhantes.

No Debian GNU/Linux é necessário instalar: *mysql-server*, *mysql-client* e *libmysqlclient-dev*.

O primeiro é o banco de dados propriamente dito, o segundo é o terminal de acesso ao banco de dados e o terceiro, a biblioteca com a API de programação em C e seus arquivos de cabeçalho *(headers)*.

Dentro do servidor é possível criar vários bancos de dados e usuários com acesso diferenciado para cada um deles. Nos bancos de dados é possível criar tabelas que também podem ter acesso diferenciado para cada usuário.

Com as tabelas, construímos nosso sistema.

Quando criamos um novo banco de dados o servidor cria um diretório */var/lib/mysql/novobanco* - onde *novobanco* é o nome do *database*.

Dentro desse diretório o MySQL organiza as tabelas em arquivos, e para cada tabela temos 3 arquivos:

Tabela 6.1: Organização das tabelas do MySQL

Nome	Descrição
nome.frm	Definição da tabela, tipos dos campos
nome.MYD	Dados da tabela
nome.MYI	Arquivo de índices

Os índices são utilizados para aumentar a velocidade da pesquisa nas tabelas. Quando criados, o servidor os mantém em um *hash* e atualiza seu conteúdo a cada modificação.

Por padrão, o único usuário do sistema que pode acessar o MySQL é o usuário *root*, então devemos utilizá-lo para criar um novo usuário no MySQL e esse usuário será utilizado na aplicação.

Depois, devemos criar algumas tabelas de maneira que o sistema seja relacional. Ao invés de executarmos um procedimento por vez podemos colocar todos em um arquivo texto e informar o MySQL para executar este arquivo.

my-app.sql

```
--
-- my-app.sql: estrutura para ambiente de testes em MySQL
--
-- Para utilizar:
--  sudo mysql < my-app.sql
--
-- Alexandre Fiori
--

-- Cria o database myapp
use mysql;
CREATE DATABASE myapp;

-- Cria o usuário 'myuser' com senha 'mypass'
CREATE USER myuser@localhost IDENTIFIED BY 'mypass';

-- Acessa o database myapp
use myapp;

-- Cria tabela para armazenamento de usuários
CREATE TABLE users (
    uid int unsigned NOT NULL auto_increment,
    name varchar(128) UNIQUE NOT NULL,
    PRIMARY KEY (uid, name)
) TYPE=MyISAM;

-- Cria tabela para armazenamento de dados
CREATE TABLE data (
    did int unsigned NOT NULL auto_increment,
    uid int unsigned NOT NULL,
    description varchar(512) NOT NULL,
    PRIMARY KEY(did)
) TYPE=MyISAM;

-- Cria o usuário 'bozo' no sistema, na tabela 'users'
INSERT INTO users (name) VALUES ('bozo');

-- Cria uma entrada na tabela 'data' que pertence ao
-- usuário 'bozo' (com uid=1)
INSERT INTO data (uid, description) VALUES (1, 'Palhaço Bozo');
```

```
-- Garante permissão total ao usuário 'myuser' no database 'myapp'
GRANT ALL on myapp.* to myuser@localhost;

-- FIM
```

Executando:

```
$ sudo mysql < my-app.sql
$ sudo ls -l /var/lib/mysql/myapp/
-rw-rw----  1 mysql mysql   28 Nov 27 21:24 data.MYD
-rw-rw----  1 mysql mysql 2048 Nov 27 21:24 data.MYI
-rw-rw----  1 mysql mysql 8630 Nov 27 21:24 data.frm
-rw-rw----  1 mysql mysql   65 Nov 27 21:24 db.opt
-rw-rw----  1 mysql mysql   20 Nov 27 21:24 users.MYD
-rw-rw----  1 mysql mysql 3072 Nov 27 21:24 users.MYI
-rw-rw----  1 mysql mysql 8588 Nov 27 21:24 users.frm
```

Agora temos o novo *database* chamado *myapp* e para acessá-lo temos dois usuários: o usuário *root* ou o usuário *myuser* com senha *mypass*.

Podemos conferir os dados criados acessando o terminal do MySQL:

```
$ mysql -p -u myuser -D myapp
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 82 to server version: 5.0.16-Debian_1-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show tables;
+-----------------+
| Tables_in_myapp |
+-----------------+
| data            |
| users           |
+-----------------+
2 rows in set (0.00 sec)

mysql> describe data;
+-------------+------------------+------+-----+---------+----------------+
| Field       | Type             | Null | Key | Default | Extra          |
+-------------+------------------+------+-----+---------+----------------+
| did         | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| uid         | int(10) unsigned | NO   |     |         |                |
| description | varchar(512)     | NO   |     |         |                |
+-------------+------------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql> describe users;
+-------+------------------+------+-----+---------+----------------+
| Field | Type             | Null | Key | Default | Extra          |
+-------+------------------+------+-----+---------+----------------+
| uid   | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name  | varchar(128)     | NO   | PRI |         |                |
+-------+------------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> select * from users;
+-----+------+
| uid | name |
+-----+------+
|   1 | bozo |
+-----+------+
1 row in set (0.00 sec)

mysql> select * from data;
+-----+-----+--------------+
```

```
| did | uid | description  |
+-----+-----+--------------+
|   1 |   1 | Palhaço Bozo |
+-----+-----+--------------+
1 row in set (0.00 sec)

mysql> quit
Bye
```

Dentro do *database* temos duas tabelas para armazenamento de informações, sendo elas *users* e *data*. Na primeira, *users*, devemos efetuar o registro de usuários e na segunda, *data*, qualquer tipo de dados.

O relacionamento entre elas dá-se pelo campo *uid* presente nas duas. Sempre que houver dados é necessário relacionar a algum usuário.

6.4.2 PostgreSQL

A instalação do PostgreSQL é diferente nas distribuições Linux devido ao fato de cada distribuição ter seu próprio gerenciador de pacotes.

No Debian GNU/Linux os pacotes necessários são: *postgresql*, *postgresql-client* e *libpq-dev*.

O primeiro é o servidor PostgreSQL, o segundo é o pacote onde há o cliente do banco que provê acesso por meio de um terminal e o último é a biblioteca da API para linguagem C com os respectivos arquivos de cabeçalho *(headers)*.

O PostgreSQL possui vários níveis de acesso. Cada *database* criado nele pode ter usuários comuns ou administradores, com permissão para criação e acesso a recursos. Cada tabela também tem níveis de acesso, exatamente como os *databases*.

Além disso o PostgreSQL tem um arquivo chamado *pg_hba.conf* com listas de permissão para acessar o servidor. Lá é possível adicionar permissão de acesso por usuário à partir do mesmo computador ou de endereços IP em determinados *databases*.

Para nosso ambiente de laboratório é necessário inserir a seguinte linha no arquivo (normalmente) */etc/postgresql/pg_hba.conf*:

```
local   myapp    myuser    md5
```

Significa permitir acesso *local* (do próprio computador, via *Unix Domain Socket*) no *database* chamado *myapp* para o usuário *myuser* com senha, criptografada com o algoritmo *md5*.

É necessário informar o PostgreSQL após modificação neste arquivo.

```
/etc/init.d/postgresql reload
```

O terminal de acesso ao banco, um programa chamado *psql*, só permite acesso (no Debian) à partir do usuário de sistema *postgres*.

Para criarmos o *database* e as tabelas podemos nos conectar por ele e fazer os procedimentos manualmente ou preparar um *script*.

pg-app.sql

```
--
-- pg-app.sql: estrutura para ambiente de testes em PostgreSQL
--
-- Para utilizar:
--  (Debian GNU/Linux)
--  sudo su postgres -c "psql < pg-app.sql"
--
```

```
-- Alexandre Fiori
--

-- Cria o usuário 'myuser' com senha 'mypass'
CREATE USER myuser ENCRYPTED PASSWORD 'mypass';

-- Cria o database myapp
CREATE DATABASE myapp OWNER myuser;

-- Acessa o database myapp
\c myapp;

-- Cria tabela para armazenamento de usuários
CREATE TABLE users (
    uid serial NOT NULL,
    name varchar(128) UNIQUE NOT NULL,
    PRIMARY KEY (uid, name)
);

-- Cria tabela para armazenamento de dados
CREATE TABLE data (
    did serial NOT NULL,
    uid int NOT NULL,
    description varchar(512) NOT NULL,
    PRIMARY KEY(did)
);

-- Cria o usuário 'bozo' no sistema, na tabela 'users'
INSERT INTO users (name) VALUES ('bozo');

-- Cria uma entrada na tabela 'data' que pertence ao
-- usuário 'bozo' (com uid=1)
INSERT INTO data (uid, description) VALUES (1, 'Palhaço Bozo');

-- Garante permissão total ao usuário 'myuser' no database 'myapp'
-- e nas tabelas users e data
GRANT ALL on database myapp to myuser;
GRANT ALL on users, users_uid_seq, data, data_did_seq to myuser

-- FIM
```

Executando:

```
$ sudo su postgres -c "psql < pg-app.sql"
CREATE ROLE
CREATE DATABASE
You are now connected to database "myapp".
...
CREATE TABLE
INSERT 0 1
INSERT 0 1
GRANT
GRANT
```

Quando criamos um novo *database* o arquivo */var/lib/postgresl/main/global/pg_database* é atualizado e uma nova entrada com o nome é adicionado. Porém, todo nome de *database* é relacionado a um número de índice que deve estar presente no *main/base/número*.

Os campos do tipo *SERIAL* criam tabelas do tipo sequência no banco, fazendo com que sejam inteiros auto incrementados.

Para conferir o ambiente criado, o *database* chamado *myapp* com o usuário *myuser* e senha *mypass* utilizamos o terminal:

```
$ psql -WU myuser -d myapp
Password for user myuser:
Welcome to psql 8.1.0, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
```

```
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

myapp=> \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | data  | table | postgres
 public | users | table | postgres
(2 rows)

myapp=> \dt data
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | data | table | postgres
(1 row)

myapp=> \dt users
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

myapp=> select * from users;
 uid | name
-----+------
   1 | bozo
(1 row)

myapp=> select * from data;
 did | uid | description
-----+-----+--------------
   1 |   1 | Palhaço Bozo
(1 row)

myapp=> \q
```

Para obter informações detalhadas das tabelas deve-se usar o comando *\d tabela*. Como são muitos detalhes além dos campos e seus tipos, não estão presentes aqui.

O relacionamento das tabelas dá-se pelo campo *uid* presente tanto em *users* quanto em *data*. Cada entrada criada na tabela *data* deve ter o campo *uid* relacionado com um usuário válido na tabela *users*.

6.5 API de programação

6.5.1 MySQL

Antes de mais nada é necessário incluir um único arquivo de cabeçalho, *mysql.h*. Nele há definição para todas as funções relacionadas ao acesso ao MySQL servidor.

O procedimento para uso da API é o seguinte:

1. Inicializar a biblioteca com *mysql_init()*
2. Conectar ao banco de dados com *mysql_real_connect()*
3. Executar os comandos SQL - vistos a seguir
4. Fechar a conexão com o banco com *mysql_close()*
5. Finalizar a biblioteca com *mysql_library_end()*

A execução dos comandos SQL são feitos através da função *mysql_query()*. Essa função usa como argumento um tipo *const char* * que deve ser uma *string* terminada em *'\0'*. Também há uma função *mysql_real_query()* que não depende do *'\0'* mas depende de um *int* definindo o tamanho da *string* - para uso com dados binários onde o *'\0'* pode fazer parte do conteúdo.

Para comandos que não retornam dados como INSERT, UPDATE e DELETE podemos saber se tiveram sucesso ou não pela quantidade de linhas afetadas na tabela, usando a função *mysql_affected_rows()*. Caso seu retorno seja 0, significa que não houve alteração no conteúdo da tabela.

Para comandos como SELECT, SHOW, DESCRIBE e outros que normalmente retornam dados, temos diferentes maneiras de proceder. A primeira é chamando a função *mysql_store_result()* que copia o resultado do comando do servidor para o cliente de uma só vez. A segunda é chamando a função *mysql_use_result()* que simplesmente informa o servidor que os dados serão copiados por vez e não transfere nada.

Em ambos os casos acessamos o conteúdo das linhas retornadas pelo servidor através da função *mysql_fetch_row()*. Com *mysql_store_result()* essa função acessa a última linha do resultado previamente copiado do servidor. Com *mysql_use_result()* ela copia a linha do servidor para o cliente. A informação do tamanho dos dados em cada linha pode ser adquirida através da função *mysql_fetch_lenghts()*.

Depois de utilizar o resultado é necessário desalocar a memória dele chamando *mysql_free_result()*.

Para saber se o comando teve o resultado esperado podemos comparar o que solicitamos com o número de campos retornados da tabela, através da função *mysql_field_count()*.

Para mais informações específicas da API acesse o site do MySQL *(http://www.mysql.com)* na sessão *Developer Zone*, *Documentation*. Lá escolha a versão e terá acesso aos manuais da API.

6.5.2 PostgreSQL

O arquivo de cabeçalho da *libpq* que deve ser incluído é *libpq-fe.h*. Nele há definição para todas as funções relacionadas ao acesso ao servidor PostgreSQL. Uma sequência deve ser seguida para efetuar a conexão e os comandos no banco:

1. Conectar ao banco de dados com *PQconnectdb()*
2. Executar os comandos SQL - vistos a seguir
3. Fechar a conexão com o banco de dados com *PQfinish()*

São poucas estapas. A execução de comandos SQL dá-se pela função *PQexec()*, onde é passada uma *string* terminada em *'\0'*. Para comandos onde há dados binários e o *'\0'* pode fazer parte da sintaxe SQL deve-se utilizar *PQexecParams()*.

O retorno da chamada a *PQexec()* é sempre um *result set*.

Para comandos SQL que retornam dados como SELECT, SHOW e outros, esses dados já vêm direto no *result set* e para averiguar se o comando realmente foi bem sucedido deve-se chamar a função *PQresultStatus()* passando o *result set* como argumento. Quando o comando é bem sucedido o retorno desta função é um *typedef int* declarado como *PGRES_TUPLES_OK* - indicando que há linhas disponíveis no *result set*. Para tratar essas linhas e colunas temos as funções *PQntuples()* que informa o número de linhas e *PQnfields()* que retorna o número de campos. Ambas precisam do *result set* como argumento e retornam *int*.

O *result set* do PostgreSQL permite acesso aleatório aos dados através da chamada a *PQgetvalue()* que solicita como argumento o *result set*, o número da linha e número da coluna, retornando um tipo *char* * com o conteúdo daquele campo.

Para outros comandos que não retornam dados como INSERT, DELETE e UPDATE, o *result set* provê apenas o *status* de sucesso ou falha do comando. Para averiguar se o comando SQL foi bem sucedido ou não, deve-se chamar a função *PQcmdTuples()* passando o *result set* como argumento. O retorno dessa função é um *char* * que informa a quantidade de linhas afetadas.

Em caso de erro pode-se imprimir a mensagem relacionada ao tipo do erro com *PQresultErrorMessage()* passando o próprio *result set* como argumento. Em caso de erro na conexão com o servidor pode-se imprimir a mensagem relacionada ao erro com *PQerrorMessage()* passando o *handler* da conexão como argumento.

A documentação completa está disponível no site do PostgreSQL *(http://www.postgresql.org)* na sessão *Documentation* e lá *Online Manual* para cada versão.

6.6 Inserindo dados

6.6.1 MySQL

Seguindo os procedimentos da API é fácil inserir dados no MySQL. Depois de ter o ambiente de laboratório criado, basta programar.

Nosso programa irá inserir dados no *database* previamente criado, chamado *myapp*. Lá, temos as tabelas *users* e *data* onde adicionamos registros de usuários ou dados.

O programa de exemplo será capaz de popular as duas tabelas de acordo como for executado na linha de comando.

Caso queira adicionar usuário, basta especificar o nome - pois o número do usuário é automaticamente criado pelo banco de dados. Caso queira adicionar dados, deve informar o nome do usuário dono dos dados e os dados propriamente ditos.

my-insert.c

```
/*
 * my-insert.c: insere uma entrada no MySQL
 *
 * Para compilar:
 *  cc -Wall my-insert.c -o my-insert -lmysqlclient
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <mysql/mysql.h>

int main(int argc, char **argv)
{
    MYSQL db;
    char *user, *entry, query[1024];

    if(argc < 2) {
        fprintf(stderr, "uso: %s usuário [entrada]\n", *argv);
        return 1;
    }

    user = argv[1];
    entry = argv[2];

    /* inicia a estrutura do mysql */
    if(mysql_init(&db) == NULL) {
        fprintf(stderr, "init: %s\n", mysql_error(&db));
        return 1;
    }
```

```
    /* conecta no banco */
    if(mysql_real_connect(&db,
                "localhost", "myuser", "mypass", "myapp",
                0, NULL, 0) == NULL) {
        fprintf(stderr, "connect: %s\n", mysql_error(&db));
        return 1;
    }

    /* caso não haja 'entry', adiciona o usuário */
    memset(query, 0, sizeof(query));
    if(!entry)
        snprintf(query, sizeof(query),
                "INSERT INTO users (name) VALUES ('%s')", user);
    else
    /* adiciona 'entry' para 'user' */
        snprintf(query, sizeof(query),
                "INSERT INTO data (uid, description) "
                "SELECT uid, '%s' as description "
                "FROM users WHERE name='%s'",
                entry, user);

    fprintf(stdout, "executando: %s\n", query);

    /* executa query no servidor */
    if(mysql_query(&db, query)) {
        fprintf(stderr, "query: %s\n", mysql_error(&db));
        mysql_close(&db);
        return 1;
    }

    /* imprime resultado */
    fprintf(stdout, "resultado: %s\n",
            (unsigned long int) mysql_affected_rows(&db) ? "OK" : "FALHOU");

    /* fecha conexão */
    mysql_close(&db);
    mysql_library_end();
    return 0;
}
```

Executando:

```
$ ./my-insert
uso: ./my-insert usuário [entrada]

$ ./my-insert lili
executando: INSERT INTO users (name) VALUES ('lili')
resultado: OK

$ ./my-insert lili "Amiga do bozo"
executando: INSERT INTO data (uid, description)
 SELECT uid, 'Amiga do bozo' as description FROM users WHERE name='lili'
resultado: OK
```

Note que para adicionar um registro de usuário na tabela *users* é muito simples. Para adicionar dados na tabela *data* é necessário vincular o novo conteúdo a um usuário, portanto é necessário verificar se o usuário existe. Ao invés de executar dois comandos podemos fazer tudo de uma vez só, chamando INSERT...SELECT. Este é um dos recursos do banco.

Caso o usuário não exista, SELECT irá falhar e consequentemente INSERT também, veja:

```
$ ./my-insert qualquer "teste"
executando: INSERT INTO data (uid, description)
 SELECT uid, 'teste' as description FROM users WHERE name='qualquer'
resultado: FALHOU
```

Assim garantimos a integridade dos dados no *database*. É importante saber que em sistemas em produção onde os dados são relacionados, nunca devemos apagar os registros de usuários pois a

tabela de dados passaria a ter dados inválidos. Para solucionar esse problema é necessário criar um campo na tabela de usuários que define se ele está ativo ou inativo, assim todos os dados vinculados a ele permanecem no banco no estado de inativo.

6.6.2 PostgreSQL

Basta seguir os procedimentos da API de programação. Este ambiente de laboratório é exatamente igual ao que foi criado para o MySQL e em ambos bancos de dados a sintaxe dos comandos SQL é igual.

Nosso programa irá se conectar no *database* previamente criado, chamado *myapp* com o usuário *myuser* e senha *mypass*. A função que realiza a conexão com o servidor, *PQconnectdb()* solicita como argumento uma *string* com as informações relacionadas à conexão, do seguinte tipo:

```
char *conn = "host=localhost dbname=myapp user=myuser pass=mypass";
```

Quando a conexão é feita com *localhost*, internamente a biblioteca *libpq* utiliza *Unix Domain Sockets*. Para qualquer outro endereço IP a biblioteca passa a utilizar *Internet Sockets*.

Este programa será capaz de popular as tabelas *users* e *data* de acordo com os argumentos da linha de comando. Caso seja passado apenas um argumento, será tratado como nome de usuário e este nome será adicionado na tabela *users*. Caso sejam dois argumentos na linha de comando, o primeiro será tratado como nome de usuário e o segundo como dados relacionados ao usuário do primeiro argumento que deve existir no banco de dados para que os dados sejam adicionados.

pg-insert.c

```
/*
 * pg-insert.c: insere uma entrada no PostgreSQL
 *
 * Para compilar:
 *  cc -Wall pg-insert.c -o pg-insert -lpq
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <postgresql/libpq-fe.h>

int main(int argc, char **argv)
{
    PGconn *db;
    PGresult *rs;
    char query[128], *user, *entry,
      *conn = "host=localhost dbname=myapp user=myuser password=mypass";

    if(argc < 2) {
        fprintf(stderr, "uso: %s usuário [entrada]\n", *argv);
        return 1;
    }

    user = argv[1];
    entry = argv[2];

    /* conecta no banco */
    if((db = PQconnectdb(conn)) == NULL) {
        fprintf(stderr, "connect: %s\n", PQerrorMessage(db));
        return 1;
    }

    /* caso não haja 'entry', adiciona usuário */
    memset(query, 0, sizeof(query));
    if(!entry)
```

```
            snprintf(query, sizeof(query),
                    "INSERT INTO users (name) VALUES ('%s')", user);
        else
        /* adiciona 'entry' para 'user' */
        snprintf(query, sizeof(query),
                "INSERT INTO data (uid, description) "
                "SELECT uid, '%s' as description "
                "FROM users WHERE name='%s'",
                entry, user);

        fprintf(stdout, "executando: %s\n", query);

        /* executa a query */
        if((rs = PQexec(db, query)) == NULL) {
            fprintf(stderr, "query: %s\n", PQresultErrorMessage(rs));
            PQfinish(db);
            return 1;
        }

        /* imprime resultado */
        fprintf(stdout, "resultado: %s\n",
                *PQcmdTuples(rs) >= '1' ? "OK" : "FALHOU");

        /* desaloca a memória do result set */
        PQclear(rs);

        /* fecha conexão com o banco */
        PQfinish(db);
        return 0;
}
```

Executando:

```
$ ./pg-insert lili
executando: INSERT INTO users (name) VALUES ('lili')
resultado: OK

$ ./pg-insert lili "Amiga do bozo"
executando: INSERT INTO data (uid, description)
 SELECT uid, 'Amiga do bozo' as description FROM users WHERE name='lili'
resultado: OK
```

A tabela *users* do *database* chamado *myapp* tem o campo *uid* do tipo *SERIAL*. Esse tipo é auto incrementável portanto não precisamos nos preocupar em adicionar o número do usuário. Para adicionar dados relacionados a um usuário é necessário garantir a existência do usuário, e para isso utilizamos o comando INSERT...SELECT.

Caso o usuário não exista, SELECT irá falhar e consequentemente INSERT também, veja:

```
$ ./pg-insert mafalda "vovo mafalda"
executando: INSERT INTO data (uid, description)
 SELECT uid, 'vovo mafalda' as description FROM users WHERE name='mafalda'
resultado: FALHOU
```

É assim que garantimos a integridade dos dados no *database*. Só adicionamos dados para um usuário quando o mesmo existe na tabela *users* e graças à sintaxe da linguagem SQL podemos executar duas tarefas com um só comando.

6.7 Realizando pesquisas

6.7.1 MySQL

Para realizar pesquisas nas tabelas utilizamos SELECT. Porém, nosso exemplo de laboratório guarda o número do usuário dono dos dados ao invés do nome. Para mostrar os dados com o nome

correto do usuário temos que realizar a pesquisa em duas tabelas simultaneamente. O SELECT permite este tipo de operação.

my-select.c

```
/*
 * my-select.c: mostra entradas no MySQL com SELECT
 *
 * Para compilar:
 *  cc -Wall my-select.c -o my-select -lmysqlclient
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <mysql/mysql.h>

int main()
{
    MYSQL db;
    MYSQL_RES *rs;
    MYSQL_ROW row;
    char query[128];
    unsigned long int i, rows;

    /* inicia a estrutura do mysql */
    if(mysql_init(&db) == NULL) {
        fprintf(stderr, "init: %s\n", mysql_error(&db));
        return 1;
    }

    /* conecta no banco */
    if(mysql_real_connect(&db,
                "localhost", "myuser", "mypass", "myapp",
                0, NULL, 0) == NULL) {
        fprintf(stderr, "connect: %s\n", mysql_error(&db));
        return 1;
    }

    /* prepara a query */
    memset(query, 0, sizeof(query));
    snprintf(query, sizeof(query),
            "SELECT d.did, u.name, d.description "
            "FROM data AS d, users AS u "
            "WHERE d.uid = u.uid");

    fprintf(stdout, "executando: %s\n", query);

    /* executa a query */
    if(mysql_query(&db, query)) {
        fprintf(stderr, "query: %s\n", mysql_error(&db));
        mysql_close(&db);
        return 1;
    }

    /* se o resultado não possuir 3 colunas significa erro */
    if((i = mysql_field_count(&db)) != 3) {
        fprintf(stderr, "colunas: %lu\n", i);
        mysql_close(&db);
        return 1;
    }

    /* copia o resultado da query do server para o client */
    if((rs = mysql_store_result(&db)) == NULL) {
        fprintf(stderr, "store: %s\n", mysql_error(&db));
        mysql_close(&db);
        return 1;
    }

    /* obtém a matriz com as linhas */
    rows = mysql_num_rows(rs);
```

```
    /* anda pelas linhas do result set */
    for(i = 0; i < rows; i++) {
        row = mysql_fetch_row(rs);

        /* imprime as colunas do result set */
        fprintf(stdout, "id=%s, user=%s: %s\n", row[0], row[1], row[2]);
    }

    /* desaloca a memória do result set */
    mysql_free_result(rs);

    /* fecha a conexão com o banco */
    mysql_close(&db);
    mysql_library_end();
    return 0;
}
```

Executando:

```
$ ./my-select
executando: SELECT d.did, u.name, d.description
 FROM data AS d, users AS u WHERE d.uid = u.uid
id=1, user=bozo: Palhaço Bozo
id=2, user=lili: Amiga do bozo
```

Temos aqui o exemplo claro do relacionamento de dados em tabelas. Realizamos a pesquisa consultando duas tabelas simultaneamente e temos os dados da maneira adequada para esta aplicação.

O tipo *MYSQL_RES* é uma estrutura *struct st_mysql_res* e pode ser encontrado no arquivo de cabeçalho *mysql.h*. Nessa estrutura estão todas as linhas retornadas pelo comando SELECT e outros que retornam dados.

O tipo *MYSQL_ROW* é um *typedef* para *char* ** - vide Sessão 2.3.5.

É importante saber que embora alguns dados sejam do tipo *unsigned int* na tabela, na API de programação eles sempre serão *strings*.

6.7.2 PostgreSQL

As pesquisas nas tabelas de dados são feitas com SELECT. Como nosso ambiente foi criado com os dados da tabela *data* sempre vinculados a um usuário, temos de imprimir o nome deste usuário ao invés de seu número de identificação. Para isso é necessário realizar o SELECT em duas tabelas simultaneamente - exatamente como fizemos no MySQL.

Como é notável, a sintaxe SQL dos dois bancos de dados é a mesma na maioria das situações - mas existem algumas divergências.

pg-select.c

```
/*
 * pg-select.c: mostra entradas no PostgreSQL com SELECT
 *
 * Para compilar:
 *  cc -Wall pg-select.c -o pg-select -lpq
 *
 * Alexandre Fiori
 */

#include <stdio.h>
#include <string.h>
#include <postgresql/libpq-fe.h>

int main()
{
```

```
    PGconn *db;
    PGresult *rs;
    char query[128],
      *conn = "host=localhost dbname=myapp user=myuser password=mypass";
    int i, rows;

    /* conecta no banco */
    if((db = PQconnectdb(conn)) == NULL) {
        fprintf(stderr, "connect: %s\n", PQerrorMessage(db));
        return 1;
    }

    /* prepara a query */
    memset(query, 0, sizeof(query));
    snprintf(query, sizeof(query),
            "SELECT d.did, u.name, d.description "
            "FROM data AS d, users AS u "
            "WHERE d.uid = u.uid");

    fprintf(stdout, "executando: %s\n", query);

    /* executa a query */
    if((rs = PQexec(db, query)) == NULL) {
        fprintf(stderr, "query: %s\n", PQresultErrorMessage(rs));
        PQfinish(db);
        return 1;
    }

    /* checa se o comando retornou dados */
    if(PQresultStatus(rs) != PGRES_TUPLES_OK) {
        fprintf(stderr, "query: %s\n", PQresultErrorMessage(rs));
        PQclear(rs);
        PQfinish(db);
        return 1;
    }

    /* obtém a matriz com as linhas */
    rows = PQntuples(rs);

    /* anda pelas linhas do result set */
    for(i = 0; i < rows; i++) {
        /* imprime as colunas do result set */
        fprintf(stdout, "id=%s, user=%s: %s\n",
                PQgetvalue(rs, i, 0),
                PQgetvalue(rs, i, 1),
                PQgetvalue(rs, i, 2));
    }

    /* desaloca a memória do result set */
    PQclear(rs);

    /* fecha conexão com o banco */
    PQfinish(db);
    return 0;
}
```

Executando:

```
$ ./pg-select
executando: SELECT d.did, u.name, d.description
 FROM data AS d, users AS u WHERE d.uid = u.uid
id=1, user=bozo: Palhaço Bozo
id=2, user=lili: Amiga do bozo
```

Aí está o relacionamento de dados entre as tabelas. Embora a tabela *data* tenha apenas o *uid*, obtemos o nome do usuário consultando aquele *uid* na tabela *users*. Com SELECT construímos o resultado da maneira adequada para a aplicação com nomes ao invés de números.