# SISTEMAS EMBARCADOS

## Programação em C: Recursos do Sistema

Autores:
**Edson Mintsu Hung**
**Evandro Leonardo Teixeira**
**Tiago Alves da Fonseca**

**Universidade de Brasília**

**Faculdade do Gama**

**Curso de Engenharia Eletrônica**

# SISTEMAS EMBARCADOS

***Conteúdo:***

1. Acesso a arquivos
2. Processos e Sinais
3. Threads POSIX e biblioteca pthreads
4. Comunicação e Sincronismo entre Processos
5. Programação usando sockets
6. Device Drivers

# 1. Acesso a arquivos

Costuma-se definir arquivo como o "lugar" onde são gravados os dados. Em outros sistemas operacionais é feita uma distinção entre arquivos, diretórios, dispositivos e outros componentes do sistema. No GNU/Linux, entretanto, tudo aquilo que pode ser manipulado pelo sistema é tratado como arquivo.

Mas, se tudo é arquivo, como o sistema operacional faz a distinção entre arquivo regular, diretório, dispositivo, processo, etc? Muito simples: todo arquivo tem o que é chamado cabeçalho, que contém informações como: tipo, tamanho, data de acesso, modificação, etc. Através desse cabeçalho o GNU/Linux sabe então quando se trata de arquivo regular, diretório, dispositivo, e assim por diante.

Os tipos de arquivos existentes num sistema GNU/Linux são os seguintes:

1) Arquivo regular: tipo comum que contém dados somente. Os arquivos regulares podem ser dos mais variados tipos, guardando os mais diferentes tipos de informações. Existem arquivos de áudio, vídeo, imagem, texto, enfim. Os arquivos se dividem em 2 (duas) categorias principais:

1.1) Binários: arquivos binários são compostos por bits 1 e 0 e só podem ser interpretados pelo sistema operacional. Programas e bibliotecas são exemplos de arquivos binários;

1.2) Texto: arquivos do tipo texto são compostos por informações em forma de texto, que podem ser entendidas pelo usuário comum. Arquivos desse tipo não necessariamente contém texto propriamente dito. Quando se diz que um arquivo é do tipo texto isto significa que, se aberto em um editor de texto, serão exibidos informações legíveis (ainda que possam não fazer muito sentido).

2) Diretórios: os diretórios são utilizados para separar um grupo de arquivos de outros. Um diretório pode conter arquivos e outros diretórios, que serão chamados subdiretórios;

3) Dispositivos: todo componente de hardware instalável é chamado dispositivo. Placas de vídeo, som, rede, drives de CD-ROM, tudo o que se liga na interface USB do PC, memória RAM, são dispositivos. Os dispositivos podem ser:

3.1) De bloco: dispositivos de bloco utilizam buffer para leitura/gravação. Geralmente são unidades de disco, como HD's, CD's, etc;

3.2) De caracter: esses dispositivos não utilizam buffer para leitura/gravação. A maioria dos dispositivos PCI e outros dispositivos como impressoras, mouse, etc. são do tipo caracter;

3.3) Fifo: trata-se de um canal de comunicação, através do qual pode-se ver os dados que estão trafegando por um dispositivo;

4) Links: os links são arquivos utilizados para fazer referência a um outro arquivo localizado em outro local. Em outras palavras, são atalhos. Os links podem ser de 2 (dois) tipos:

4.1) Simbólicos: fazem uma referência ao arquivo através de seu endereço lógico no disco ou memória. São os links mais comuns;

4.2) Absolutos: fazem referência ao arquivo através do seu endereço físico no disco rígido ou memória.

5) FIFO: canal de comunicação, utilizado para direcionar os dados produzidos por

um processos para um outro processo.

Arquivos podem conter diferentes tipos de informações. Cada tipo de informação requer um método específico de trabalho. Assim, um arquivo de imagem não pode ser lido por um programa de reprodução de áudio, porque os dados de um arquivo de imagem são organizados de forma totalmente diferente da de um arquivo de áudio, e o programa em questão só é capaz de lidar com arquivos de áudio.

Tendo isso em vista, é necessário que o sistema operacional e os demais programas possam diferenciar os diversos tipos de arquivos disponíveis, para evitar que um programa tente manipular um tipo de arquivo que não suporta. O GNU/Linux faz essa diferenciação através da leitura do cabeçalho do arquivo. Assim, a extensão do arquivo geralmente não importa para o sistema operacional, mas é usada tão somente para fácil identificação do usuário, motivo pelo qual é muito comum encontrar-se arquivos sem extensão no GNU/Linux.

## 1.1 Acesso a arquivos utilizando **stdio**

O sistema de entrada e saída do ANSI C é composto por uma série de funções, cujos protótipos estão reunidos em **stdio.h** . Todas estas funções trabalham com o conceito de "ponteiro de arquivo". Este não é um tipo propriamente dito, mas uma definição usando o comando typedef. Esta definição também está no arquivo **stdio.h**. Pode-se declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

**p** será então um ponteiro para um arquivo. É usando este tipo de ponteiro que pode-se manipular arquivos no C.

### fopen()

Esta é a função de abertura de arquivos. Seu protótipo é:

```
FILE *fopen (char *nome_do_arquivo,char *modo);
```

O nome_do_arquivo determina qual arquivo deverá ser aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O modo de abertura diz à função `fopen()` que tipo de uso você vai fazer do arquivo. A tabela abaixo mostra os valores de modo válidos:

| Modo | Significado |
|---|---|
| "r" | Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto. |
| "w" | Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído. |
| "a" | Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente. |
| "rb" | Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o |

| | |
|---|---|
| | arquivo é binário. |
| "wb" | Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário. |
| "ab" | Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário. |
| "r+" | Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado. |
| "w+" | Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado. |
| "a+" | Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente. |
| "r+b" | Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário. |
| "w+b" | Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário. |
| "a+b" | Acrescenta dados ou cria uma arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário |

Pode-se então, para abrir um arquivo binário para escrita, escrever:

```
FILE *fp;                /* Declaração da estrutura
fp=fopen ("exemplo.bin","wb");  /* o arquivo se chama exemplo.bin e está
localizado no diretório corrente */

if (!fp)
        printf ("Erro na abertura do arquivo.");
```

A condição !fp testa se o arquivo foi aberto com sucesso porque no caso de um erro a função `fopen()` retorna um ponteiro nulo (NULL).

Uma vez aberto um arquivo, pode-se ler ou escrever nele utilizando as funções que serão apresentadas a seguir.

Toda vez que manipula-se arquivos, há uma espécie de posição atual no arquivo. Esta é a posição de onde será lido ou escrito o próximo caractere. Normalmente, num acesso sequencial a um arquivo, não é necessário alterar esta posição pois quando se lê um caractere a posição no arquivo é automaticamente atualizada. Num acesso randômico tem-se que alterar esta posição (vide `fseek()`).

### exit()

A função **exit()** possui um protótipo do tipo:

```
void exit (int codigo_de_retorno);
```

Para utilizá-la deve-se colocar um include para o arquivo de cabeçalho stdlib.h. Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no

programa e faz com que o programa termine e retorne, para o sistema operacional, o código_de_retorno. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema. A função `exit()` se torna importante em casos como alocação dinâmica e abertura de arquivos pois nestes casos, se o programa não conseguir a memória necessária ou abrir o arquivo, a melhor saída pode ser terminar a execução do programa. Pode-se reescrever o exemplo da seção anterior usando agora o exit() para garantir que o programa não deixará de abrir o arquivo:

```
#include <stdio.h>
#include <stdlib.h> /* Para a função exit() */

main (void)
{
FILE *fp;
...
fp=fopen ("exemplo.bin","wb");
if (!fp)
        {
        printf ("Erro na abertura do arquivo. Fim de programa.");
        exit (1);
        }
...
return 0;
}
```

**fclose()**

Quando termina-se de usar um arquivo que aberto, deve-se fechá-lo. Para tanto usa-se a função `fclose()`:

```
int fclose (FILE *fp);
```

O ponteiro fp passado à função `fclose()` determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

Fechar um arquivo faz com que qualquer caracter que tenha permanecido no "buffer" associado ao fluxo de saída seja gravado. Mas, o que é este "buffer"? Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o "buffer") em vez de serem escritos em disco imediatamente. Quando o "buffer" estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caracter que se fosse gravar, tivesse que posicionar a cabeça de gravação em um ponto específico do disco, apenas para gravar aquele caracter, as gravações seriam muito lentas. Assim estas gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

A função `exit()` fecha todos os arquivos que um programa tiver aberto.

**putc()**

A função putc é a primeira função de escrita de arquivo que veremos. Seu protótipo é:

```
int putc (int ch,FILE *fp);
```

Escreve um caractere no arquivo.

O programa a seguir lê uma string do teclado e escreve-a, caractere por caractere em um arquivo em disco (o arquivo arquivo.txt, que será aberto no diretório corrente).

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

int main()
{
 FILE *fp;
 char string[100];
 int i;
 fp = fopen("arquivo.txt","w");   /* Arquivo ASCII, para escrita */
 if(!fp)
 {
    printf( "Erro na abertura do arquivo");
    exit(0);
 }
 printf("Entre com a string a ser gravada no arquivo:");
 gets(string);
 for(i=0; string[i]; i++) putc(string[i], fp); /* Grava a string, caractere a
caractere */
 fclose(fp);
 return 0;
}
```

Depois de executar este programa, verifique o conteúdo do arquivo arquivo.txt (você pode usar qualquer editor de textos ou visualizador). Você verá que a string que você digitou está armazenada nele.getc

Retorna um caractere lido do arquivo. Protótipo:

```
int getc (FILE *fp);
```

**feof()**

EOF ("End of file") indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto utiliza-se a função `feof()`. Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

```
int feof (FILE *fp);
```

Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por getc com EOF. O programa a seguir abre um arquivo já existente e o lê, caracter por caracter, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
FILE *fp;
char c;
fp = fopen("arquivo.txt","r"); /* Arquivo ASCII, para leitura */
if(!fp)
```

```
{
    printf( "Erro na abertura do arquivo");
    exit(-1);
}
while((c = getc(fp) ) != EOF)  /* Enquanto não chegar ao final do arquivo */
    printf("%c", c);           /* imprime o caracter lido */
fclose(fp);
return 0;
}
```

A seguir é apresentado um programa onde várias operações com arquivos são realizadas, usando as funções dantes vistas. Primeiro o arquivo é aberto para a escrita, e imprime-se algo nele. Em seguida, o arquivo é fechado e novamente aberto para a leitura. Verifique o exemplo.

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

void main()
{
        FILE *p;
        char c, str[30], frase[80] = "Este e um arquivo chamado: ";
        int i;
        /* Le um nome para o arquivo a ser aberto: */
        printf("\n\n Entre com um nome para o arquivo:\n");
        gets(str);
        if (!(p = fopen(str,"w")))  /* Caso ocorra algum erro na abertura do
arquivo..*/
        {                           /* o programa aborta automaticamente */
          printf("Erro! Impossivel abrir o arquivo!\n");
          exit(1);
        }
        /* Se nao houve erro, imprime no arquivo e o fecha ...*/
        strcat(frase, str);
        for (i=0; frase[i]; i++)
         putc(frase[i],p);
        fclose(p);
        /* Abre novamente para  leitura  */
        p = fopen(str,"r");
        c = getc(p);            /* Le o primeiro caracter */
        while (!feof(p))        /* Enquanto não se chegar no final do arquivo
*/
        {
                printf("%c",c); /*   Imprime o caracter na tela */
                c = getc(p);    /* Le um novo caracter no arquivo */
        }
        fclose(p);              /* Fecha o arquivo */
}
```

Arquivos pré-definidos

Quando se começa a execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos:

.* stdin: dispositivo de entrada padrão (geralmente o teclado)

.* stdout: dispositivo de saída padrão (geralmente o vídeo)

.* stderr: dispositivo de saída de erro padrão (geralmente o vídeo)

.* stdaux: dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)

.* stdprn : dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

Cada uma destas constantes pode ser utilizada como um ponteiro para FILE, para acessar os periféricos associados a eles. Desta maneira, pode-se, por exemplo, usar:

```
ch =getc(stdin);
```

para efetuar a leitura de um caracter a partir do teclado, ou :

```
putc(ch, stdout);
```

para imprimí-lo na tela.

**fgets()**

Para se ler uma string num arquivo pode-se usar `fgets()` cujo protótipo é:

```
char *fgets (char *str, int tamanho,FILE *fp);
```

A função recebe três argumentos: a string a ser lida, o limite máximo de caracteres a serem lidos e o ponteiro para FILE, que está associado ao arquivo de onde a string será lida. A função lê a string até que um caracter de nova linha seja lido ou tamanho-1 caracteres tenham sido lidos. Se o caracter de nova linha ('\n') for lido, ele fará parte da string, o que não acontecia com gets. A string resultante sempre terminará com '\0' (por isto somente tamanho-1 caracteres, no máximo, serão lidos).

A função fgets é semelhante à função `gets()`, porém, além dela poder fazer a leitura a partir de um arquivo de dados e incluir o caracter de nova linha na string, ela ainda especifica o tamanho máximo da string de entrada. Como vimos, a função gets não tinha este controle, o que poderia acarretar erros de "estouro de buffer". Portanto, levando em conta que o ponteiro fp pode ser substituído por stdin, como visto anteriormente, uma alternativa ao uso de *gets* é usar a seguinte construção:

```
fgets (str, tamanho, stdin);
```

onde str é a string que se está lendo e tamanho deve ser igual ao tamanho alocado para a string subtraído de 1, por causa do '\0'.

**fputs()**

Protótipo:

```
char *fputs (char *str,FILE *fp);
```

Escreve uma string num arquivo.

ferror e perror

Protótipo de ferror:

```
int ferror (FILE *fp);
```

A função retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo.

`ferror()` se torna muito útil quando almeja-se verificar se cada acesso a um arquivo teve sucesso, de modo a garantir a integridade dos nossos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não consegue-se ler, etc.

Uma função que pode ser usada em conjunto com `ferror()` é a função `perror()` (*print error*), cujo argumento é uma string que normalmente indica em que parte do programa o problema ocorreu.

No exemplo a seguir, faz-se o uso de `ferror()`, `perror()` e `fputs()`.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *pf;
    char string[100];
    if((pf = fopen("arquivo.txt","w")) ==NULL)
    {
        printf("\nNao consigo abrir o arquivo ! ");
        exit(1);
    }
    do
    {
        printf("\nDigite uma nova string. Para terminar, digite <enter>: ");
        gets(string);
        fputs(string, pf);
        putc('\n', pf);
        if(ferror(pf))
        {
            perror("Erro na gravacao");
            fclose(pf);
            exit(1);
        }
    } while (strlen(string) > 0);
    fclose(pf);
}
```

**fread()**

Com esta função pode-se escrever e ler blocos de dados. Para tanto, existem as

funções `fread()` e `fwrite()`. O protótipo de `fread()` é:

```
unsigned fread (void *buffer, int numero_de_bytes, int count, FILE *fp);
```

O buffer é a região de memória na qual serão armazenados os dados lidos. O argumento numero_de_bytes contém o tamanho da unidade a ser lida. count indica quantas unidades devem ser lidas. Isto significa que o número total de bytes lidos é:

```
numero_de_bytes*count
```

A função retorna o número de unidades efetivamente lidas. Este número pode ser menor que count quando o fim do arquivo for encontrado ou ocorrer algum erro.

Quando o arquivo for aberto para dados binários, fread pode ler qualquer tipo de dados.

**fwrite()**

A função `fwrite()` funciona como a sua companheira `fread()`, porém escrevendo no arquivo. Seu protótipo é:

```
unsigned fwrite(void *buffer,int numero_de_bytes,int count,FILE *fp);
```

A função retorna o número de itens escritos. Este valor será igual a count a menos que ocorra algum erro.

O exemplo abaixo ilustra o uso de fwrite e fread para gravar e posteriormente ler uma variável float em um arquivo binário.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
FILE *pf;
float pi = 3.1415;
float pilido;
if((pf = fopen("arquivo.bin", "wb")) == NULL) /* Abre arquivo binário para escrita */
{
    printf("Erro na abertura do arquivo");
    exit(1);
}
if(fwrite(&pi, sizeof(float), 1,pf) != 1)     /* Escreve a variável pi */
    printf("Erro na escrita do arquivo");
fclose(pf);                                    /* Fecha o arquivo */
if((pf = fopen("arquivo.bin", "rb")) == NULL) /* Abre o arquivo novamente para leitura */
{
    printf("Erro na abertura do arquivo");
    exit(1);
```

```
}
if(fread(&pilido, sizeof(float), 1,pf) != 1)   /* Le em pilido o valor da
variável armazenada anteriormente */
    printf("Erro na leitura do arquivo");
printf("\nO valor de PI, lido do arquivo e': %f", pilido);
fclose(pf);
return(0);
}
```

Note-se o uso do operador sizeof, que retorna o tamanho em bytes da variável ou do tipo de dados.

**fseek()**

Para se fazer procuras e acessos randômicos em arquivos usa-se a função `fseek()`. Esta move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado. Seu protótipo é:

```
int fseek (FILE *fp,long numbytes,int origem);
```

O parâmetro origem determina a partir de onde os numbytes de movimentação serão contados. Os valores possíveis são definidos por macros em `stdio.h` e são:

| *Nome* | *Valor* | *Significado* |
|---|---|---|
| SEEK_SET | 0 | Início do arquivo |
| SEEK_CUR | 1 | Posição corrente do arquivo |
| SEEK_END | 2 | Fim do arquivo |

Tendo-se definido a partir de onde irá se contar, numbytes determina quantos bytes de deslocamento serão dados na posição atual.

**rewind()**

A função `rewind()` de protótipo

```
void rewind (FILE *fp);
```

retorna a posição corrente do arquivo para o início.

**remove()**

Protótipo:

```
int remove (char *nome_do_arquivo);
```

Apaga um arquivo especificado.

O código a seguir exemplifica a utilização das seguintes funções: `fgets()`, `fputs()`, `fwrite()`, `fread()`, `fgets()` e `fputs()`.

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

int main()
{
        FILE *p;
        char str[30], frase[] = "Este e um arquivo chamado: ", resposta[80];
        int i;
        /* Le um nome para o arquivo a ser aberto: */
        printf("\n\n Entre com um nome para o arquivo:\n");
        fgets(str,29,stdin);/* Usa fgets como se fosse gets */


        for(i=0; str[i]; i++)
              if(str[i]=='\n')
                str[i]=0; /* Elimina o \n da string lida */
              if (!(p = fopen(str,"w"))) /* Caso ocorra algum erro na
abertura do arquivo..*/
     {
              /* o programa aborta automaticamente */
                printf("Erro! Impossivel abrir o arquivo!\n");
                exit(1);
        }
        /* Se nao houve erro, imprime no arquivo, e o fecha ...*/
        fputs(frase, p);
        fputs(str,p);
        fclose(p);
        /* abre novamente e le */
        p = fopen(str,"r");
        fgets(resposta, 79, p);
        printf("\n\n%s\n", resposta);
        fclose(p);                      /* Fecha o arquivo */
        remove(str);                    /* Apaga o arquivo */
        return(0);
}
```

As funções que resumem todas as funções de entrada e saída formatada no C são as funções `printf()` e `scanf()`. Um domínio destas funções é fundamental ao programador.

**printf()**

Protótipo:

```
int printf (char *str,...);
```

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle **str**, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato. Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de **%**. A cada comando de formato deve corresponder um argumento na função `printf()`. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.

Abaixo é apresentada a tabela de códigos de formato:

| Código | Formato |
|---|---|
| %c | Um caracter (char) |
| %d | Um número inteiro decimal (int) |
| %i | O mesmo que %d |
| %e | Número em notação científica com o "e"minúsculo |
| %E | Número em notação científica com o "e"maiúsculo |
| %f | Ponto flutuante decimal |
| %g | Escolhe automaticamente o melhor entre %f e %e |
| %G | Escolhe automaticamente o melhor entre %f e %E |
| %o | Número octal |
| %s | String |
| %u | Decimal "unsigned" (sem sinal) |
| %x | Hexadecimal com letras minúsculas |
| %X | Hexadecimal com letras maiúsculas |
| %% | Imprime um % |
| %p | Ponteiro |

Exemplificando:

| Código | Imprime |
|---|---|
| `printf ("Um %%%c %s",'c',"char");` | `Um %c char` |
| `printf ("%X %f %e",107,49.67,49.67);` | `6B 49.67 4.967e1` |
| `printf ("%d %o",10,10);` | `10 12` |

É possível também indicar o tamanho do campo, justificação e o número de casas decimais. Para isto usa-se códigos colocados entre o **%** e a letra que indica o tipo de formato.

Um inteiro indica o tamanho mínimo, em caracteres, que deve ser reservado para a saída. Se colocar **%5d** indica que o campo terá cinco caracteres de comprimento *no mínimo*. Se o inteiro precisar de mais de cinco caracteres para ser exibido então o campo terá o comprimento necessário para exibi-lo. Se o comprimento do inteiro for menor que cinco então o campo terá cinco de comprimento e será preenchido com espaços em branco. Se se quiser um preenchimento com zeros pode-se colocar um zero antes do número. Tem-se então que **%05d** reservará cinco casas para o número e se este for menor então se fará o preenchimento com zeros.

O alinhamento padrão é à direita. Para se alinhar um número à esquerda usa-se um sinal - antes do número de casas. Então **%-5d** será o nosso inteiro com o número mínimo de cinco casas, só que justificado a esquerda.

Pode-se indicar o número de casas decimais de um número de ponto flutuante. Por exemplo, a notação **%10.4f** indica um ponto flutuante de comprimento total dez casas decimais inteiras e com quatro casas decimais de precisão. Entretanto, esta mesma notação, quando aplicada a tipos como inteiros e strings indica o número mínimo e máximo de casas. Então **%5.8d** é um inteiro com comprimento mínimo de cinco e máximo de oito.

Exemplificando:

| Código | Imprime |
|---|---|
| `printf ("%-5.2f",456.671);` | `\|456.67\|` |
| `printf ("%5.2f",2.671);` | `\| 2.67\|` |
| `printf ("%-10s","Ola");` | `\|Ola  \|` |

Nos exemplos o "pipe" ( | ) indica o início e o fim do campo mas não são escritos na tela.

**scanf()**

Protótipo:

```
int scanf (char *str,...);
```

A string de controle str determina, assim como com a função `printf()`, quantos parâmetros a função vai necessitar. Deve-se lembrar que a função `scanf()` deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador &. Os especificadores de formato de entrada são muito parecidos com os de `printf()`. Os caracteres de conversão *d, i, u* e *x* podem ser precedidos por *h* para indicarem que um apontador para *short* ao invés de *int* aparece na lista de argumento, ou pela letra *l* (letra ele) para indicar que que um apontador para *long* aparece na lista de argumento. Semelhantemente, os caracteres de conversão *e, f* e *g* podem ser precedidos por *l* para indicarem que um apontador para *double* ao invés de *float* está na lista de argumento. Exemplos:

| Código | Formato |
|---|---|
| %hi | Um short int |
| %li | Um long int |
| %lf | Um double |

**sprintf() e sscanf()**

`sprintf` e `sscanf` são semelhantes a `printf` e `scanf`. Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou leem em uma string. Os

protótipos são:

```
int sprintf (char *destino, char *controle, ...);

int sscanf (char *destino, char *controle, ...);
```

Estas funções são muito utilizadas para fazer a conversão entre dados na forma numérica e sua representação na forma de strings. No programa abaixo, por exemplo, a variável i é "impressa" em string1. Além da representação de i como uma string, string1 também conterá “Valor de i=” .

```
#include <stdio.h>

int main()
{
    int i;
    char string1[20];
    printf( " Entre um valor inteiro: ");
    scanf("%d", &i);
    sprintf(string1,"Valor de i = %d", i);
    puts(string1);
    return 0;
}
```

Já no programa abaixo, foi utilizada a função `sscanf` para converter a informação armazenada em string1 em seu valor numérico:

```
#include <stdio.h>

int main()
{
    int i, j, k;
    char string1[]= "10 20 30";
    sscanf(string1, "%d %d %d", &i, &j, &k);
    printf("Valores lidos: %d, %d, %d", i, j, k);
    return 0;
}
```

Os fluxos padrão em arquivos permitem ao programador ler e escrever em arquivos da maneira padrão com a qual se lê e escreve na tela.

### fprintf()

A função `fprintf()` funciona como a função `printf()`. A diferença é que a saída de `fprintf()` é um arquivo e não a tela do computador. Protótipo:

```
int fprintf (FILE *fp,char *str,...);
```

Como já é de se esperar, a única diferença do protótipo de `fprintf()` para o de `printf()` é a especificação do arquivo destino por meio do ponteiro de arquivo.

### fscanf()

A função `fscanf()` funciona como a função `scanf()`. A diferença é que `fscanf()` lê de um arquivo e não do teclado do computador. Protótipo:

```
int fscanf (FILE *fp,char *str,...);
```

Obviamente, a única diferença do protótipo de `fscanf()` para o de `scanf()` é a especificação do arquivo destino através do ponteiro de arquivo.

Talvez a forma mais simples de escrever o programa, seja usando `fprintf()` e `fscanf()`. Resultando em:

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

int main()
{
        FILE *p;
        char str[80],c;
        /* Le um nome para o arquivo a ser aberto: */
        printf("\n\n Entre com um nome para o arquivo:\n");
        gets(str);
        if (!(p = fopen(str,"w"))) /* Caso ocorra algum erro na abertura do
arquivo..*/
        {         /* o programa aborta automaticamente */
                printf("Erro! Impossivel abrir o arquivo!\n");
                exit(1);
        }
        /* Se nao houve erro, imprime no arquivo, fecha ...*/
        fprintf(p,"Este e um arquivo chamado:\n%s\n", str);
        fclose(p);
        /* abre novamente para a leitura*/
        p = fopen(str,"r");
        while (!feof(p))
        {
        fscanf(p,"%c",&c);
        printf("%c",c);
        }
        fclose(p);
        return(0);
}
```

# 2. Processos e sinais

Processo é a forma de representar um programa em execução em um sistema operacional. É o processo que utiliza os recursos do computador - processador, memória, dispositivos, etc - para a realização das tarefas para as quais a máquina é destinada. Tipicamente o sistema UNIX compartilha códigos e bibliotecas entre processos, de modo que a memória seja ocupada sem redundância.

Aplicativos desenvolvidos utilizando multi-processos permitem executar mais de uma operação simultâneamente, ou até mesmo utilizar programas prontos.

## 2.1.1. Composição de um processo

O sistema operacional lida com uma infinidade de processos e, por isso, é necessário ter meios que permitam controlá-los. Para isso, os processos contam com um conjunto de características, dentre as quais:

- Proprietário do processo;

- Estado do processo (em espera, em execução, etc);

- Prioridade de execução;

- Recursos de memória.

O trabalho de gerenciamento de processos precisa contar com as informações acima e com outras de igual importância para que as tarefas sejam executadas da maneira mais eficiente. Um dos meios usados para isso é atribuir a cada processo uma identificação denominada PID (Process Identifier).

O PID é são valores inteiros de 16 bits que são alocados sequencialmente pelo sistema operacional à medida que novos processos são criados, evitando assim a colisão entre suas identificações.

Os sistemas baseados em Unix precisam que um processo já existente se duplique para que a cópia possa ser atribuída a uma tarefa nova. Quando isso ocorre, o processo "copiado" recebe o nome de "processo pai", enquanto que o novo é denominado "processo filho". É nesse ponto que o PPID (Parent Process Identifier) passa a ser usado: o PPID de um processo nada mais é do que o PID de seu processo pai.

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
   printf ("O identificador do processo - PID é: %d\n", (int) getpid ());
   printf ("O identificador do processo pai - PPID é: %d\n", (int) getppid
());
   return 0;
}
```

### 2.1.2. UID e GID

Conforme já mencionado, cada processo precisa de um proprietário, um usuário que seja considerado seu dono. A partir daí, o sistema saberá, através das permissões fornecidas pelo proprietário, quem pode e quem não pode executar o processo em questão. Para lidar com os donos, o sistema usa os números UID e GID.

O Linux gerencia os usuários e os grupos através de números conhecidos como UID (*User Identifier*) e GID (*Group Identifier*). Como é possível perceber, UID são números de usuários e GID são números de grupos. Os nomes dos usuários e dos grupos servem apenas para facilitar o uso humano do computador.

Cada usuário precisa pertencer a um ou mais grupos. Como cada processo (e cada arquivo) pertence a um usuário, logo, esse processo pertence ao grupo de seu proprietário. Assim sendo, cada processo está associado a um UID e a um GID.

Os números UID e GID variam de 0 a 65536. Dependendo do sistema, o valor limite pode ser maior. No caso do usuário root, esses valores são sempre 0 (zero). Assim, para fazer com que um usuário tenha os mesmos privilégios que o root, é necessário que seu GID seja 0.

### 2.1.3. Sinais de processos

Os sinais são meios usados para que os processos possam se comunicar e para que o sistema possa interferir em seu funcionamento. Por exemplo, se o usuário executar o comando `kill` para interromper um processo, isso será feito por meio de um sinal.

Quando um processo recebe um determinado sinal e conta com instruções sobre o que fazer com ele, tal ação é colocada em prática. Se não houver instruções pré-

programadas, o próprio Linux pode executar a ação de acordo com suas rotinas.

## 2.2. Criando processos

Existem duas técnicas comuns utilizadas para criar um novo processo. A primeira é relativamente simples, mas deve ser utilizada com certa restrição, devido às ineficiências (uma vez que podem ocorrer falhas na execução), além de possuir um risco considerável de segurança. Já a segunda técnica é mais complexa, porém provê maior flexibilidade, velocidade e segurança.

### 2.2.1. Utilizando system()

A função system oriunda da biblioteca padrão do C (`stdlib.h`) permite, de maneira muito simples executar um comando dentro do programa em execução. A partir dele, o sistema cria um sub-processo onde o comando é executado em um shell padrão.

```
#include <stdlib.h>
int main ()
{
int retorna_valor;
retorna_valor = system ("ls -l /");
return retorna_valor;
}
```

A função system retorna em sua saída o status do comando no shell. Se o shell não puder ser executado, o `system()` retorna o valor 127; se um outro erro ocorre, a função retorna -1.

Como a função system utiliza o shell para invocar um comando, ela fica sujeita às características, limitações e falhas de segurança inerentes do shell do sistema. Além disso, não se pode garantir que uma versão particular do shell Bourne (por exemplo) esteja disponível. Ou até mesmo, restrições devido aos privilégios do usuário podem inviabilizar o sistema em questão.

### 2.2.2. Utilizando fork() e exec()

Diferente do que ocorre no DOS e no Windows, que possuem uma família de funções específicas denominada `spawn`, onde estas funções cujo argumento é o nome do programa a ser executado, criando um novo processo a partir do programa em execução. O Linux não possui uma função única que inicia e executa um novo processo em um único passo. No caso, o Linux disponibiliza uma função denominada `fork`, que cria um processo filho que é uma cópia exata do processo pai. Além disso, o Linux disponibiliza um outro conjunto de funções, a família `exec`, que chaveia entre uma instância entre dois processos. Ou seja, para se criar um novo processo, se utiliza um `fork` para fazer a cópia do processo em questão. Em seguida o `exec` transforma um destes processos em instância de outro programa.

#### 2.2.2.1. Chamando fork()

Quando um programa chama o `fork()`, uma duplicação de processos, denominada

processo filho (*child process*) é criada. O processo pai continua a executar normalmente o programa de onde o `fork()` foi chamado. Assim como o processo filho também continua a execução desde o `fork()`.

Então como é feita a distinção entre estes dois processos? Primeiro, o processo filho é um novo processo e isso implica em um novo PID – diferente de seu pai. Uma maneira de distinguir o filho do pai em um programa é simplesmente fazer uma chamada com a função `getpid()`. Entretanto, a função `fork()` retorna valores distintos. O valor de retorno no processo pai é o PID do processo filho, ou seja, retorna um novo PID. Já o valor do retorno do filho é zero.

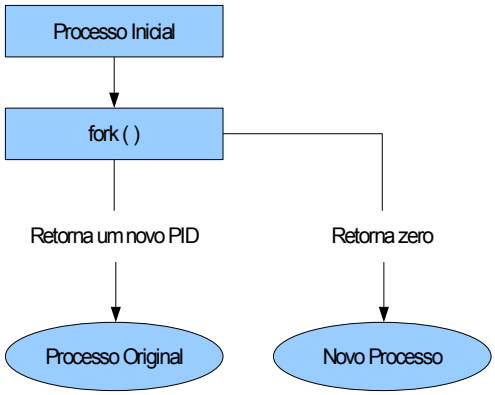


Figura 2.1: Duplicação de processo com o `fork`

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

int main ()
{
  pid_t child_pid;
  printf ("the main program process ID is %d\n", (int) getpid ());
  child_pid = fork ();
  if (child_pid != 0) {
   printf ("this is the parent process, with id %d\n", (int) getpid
());
   printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
   printf ("this is the child process, with id %d\n", (int) getpid
());
  return 0;
}
```

### 2.2.2.2. Utilizando a família exec()

A função `exec` substitui o programa em execução de um processo por outro programa. Quando um programa chama a função `exec`, o processo cessa imediatamente a execução do programa corrente e passa a executar um novo programa do início, isso se assumir que a chamada não possua ou encontre nenhum erro.

A família `exec` são funções que variam sutilmente na sua funcionalidade e também na maneira em que são chamados.

- Funções que contém a letra 'p' em seus nomes (`execvp` e `execlp`) aceitam que o nome ou procura do programa esteja no *current path*; funções que não possuem o 'p' devem conter o caminho completo do programa a ser executado.
- Funções que contém a letra 'v' em seus nomes (`execv`, `execvp` e `execve`) aceitam que a lista de arqumentos do novo programa sejam nulos. Funções que contém a letra 'l' aceitam em sua lista de argumentos a utilização de mecanismos *varargs* em linguagem C.
- Funções que contém a letra 'e' em seus nomes (`exece` e `execle`) aceitam um argumento adicional.

Como a função `exec` substitui o programa em execução por um outro, ele não retorna valor algum, exceto quando um erro ocorre.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/* Spawn a child process running a new program. PROGRAM is the name
of the program to run; the path will be searched for this program.
ARG_LIST is a NULL-terminated list of character strings to be
passed as the program's argument list. Returns the process ID of
the spawned process. */
int spawn (char* program, char** arg_list)
{
pid_t child_pid;
/* Duplicate this process. */
child_pid = fork ();
if (child_pid != 0)
/* This is the parent process. */
return child_pid;
else {
/* Now execute PROGRAM, searching for it in the path. */
execvp (program, arg_list);
/* The execvp function returns only if an error occurs. */
fprintf (stderr, "an error occurred in execvp\n");
abort ();
```

```
}
}
int main ()
{
/* The argument list to pass to the "ls" command. */
char* arg_list[] = {
"ls", /* argv[0], the name of the program. */
"-l",
"/",
NULL /* The argument list must end with a NULL. */
};
/* Spawn a child process running the "ls" command. Ignore the
returned child process ID. */
spawn ("ls", arg_list);
printf ("done with main program\n");
return 0;
}
```

## 2.3. Sinais

Um sinal é uma interrupção por software enviada aos processos pelo sistema para informá-los da ocorrência de eventos "anormais" dentro do ambiente de execução (por exemplo, falha de segmentação, violação de memória, erros de entrada e saída, etc). Deve-se notar que este mecanismo possibilita ainda a comunicação e manipulação de processos.

Um sinal (com exceção do SIGKILL) pode ser tratado de três maneiras ditintas em UNIX:
1) pode ser simplesmente ignorado. Por exemplo, o programa pode ignorar as interrupções de teclado geradas pelo usuário (e exatamente o que se passa quando um processo e executado em background).
2) pode ser interceptado. Neste caso, na recepção do sinal, a execução de um processo é desviado para o procedimento especificado pelo usuário, para depois retomar a execução no ponto de onde foi interrompido.
3) o comportamento padrão (default) pode ser aplicado à um processo após a recepção de um sinal.

### 2.3.1. Tipos de sinal

Os sinais são identificados pelo sistema por um número inteiro. O arquivo `/usr/include/signal.h` contém a lista de sinais acessíveis ao usuário. Cada sinal é caracterizado por um mnemônico. Os sinais mais usados nas aplicações em UNIX são listados a seguir:

SIGHUP (1) Corte: sinal emitido aos processos associados a um terminal quando este se "desconecta". Este sinal também é emitido a cada processo de um grupo quando o chefe termina sua execução.

SIGINT (2) Interrupção: sinal emitido aos processos do terminal quando as teclas de interrupção (por exemplo: INTR, CTRL+c) do teclado são acionadas.

SIGQUIT (3)* Abandono: sinal emitido aos processos do terminal quando com a tecla de abandono (QUIT ou CTRL+d) do teclado são acionadas.

SIGILL (4)* Instrução ilegal: emitido quando uma instrução ilegal é detectada.

SIGTRAP (5)* Problemas com *trace*: emitido após cada intrução em caso de geração de *traces* dos processos (utilização da primitiva `ptrace()`)

SIGIOT (6)* Problemas de instrução de E/S: emitido em caso de problemas de hardware.

SIGEMT (7) Problemas de intrução no emulador: emitido em caso de erro material dependente da implementação.

SIGFPE (8)* Emitido em caso de erro de cálculo em ponto flutuante, assim como no caso de um número em ponto flutuante em formato ilegal. Indica sempre um erro de programação.

SIGKILL (9) Destruição: "arma absoluta" para matar os processos. Não pode ser ignorada, tampouco interceptada (existe ainda o SIGTERM para uma morte mais "suave" para processos).

SIGBUS (10)* Emitido em caso de erro sobre o barramento.

SIGSEGV (11)* Emitido em caso de violação da segmentação: tentativa de acesso a um dado fora do domínio de endereçamento do processo.

SIGSYS (12)* Argumento incorreto de uma chamada de sistema.

SIGPIPE (13) Escrita sobre um pipe não aberto em leitura.

SIGALRM (14) Relógio: emitido quando o relógio de um processo pára. O relógio é colocado em funcionamento utilizando a primitiva alarm().

SIGTERM (15) Terminação por software: emitido quando o processo termina de maneira normal. Pode ainda ser utilizado quando o sistema quer por fim à execução de todos os processos ativos.

SIGUSR1 (16) Primeiro sinal disponível ao usuário: utilizado para a comunicação entre processos.

SIGUSR2 (17) Outro sinal disponível ao usuário: utilizado para comunicação interprocessual.

SIGCLD (18) Morte de um filho: enviado ao pai pela terminção de um processo filho.

SIGPWR (19) Reativação sobre pane elétrica.

Observação: Os sinais marcados por * geram um arquivo *core* no disco quando eles não são corretamente tratados.

Dica: para maior portabilidade dos programas que utilizam sinais, pode-se pensar em aplicar as seguintes heurísticas: evitar os sinais SISIOT, SIGEMT, SIGBUS e SIGSEGV que são dependentes da implementação. O mais correto seria interceptá-los para imprimir uma mensagem relativa à eles, mas não se deve jamais tentar atribuir uma signicação qualquer que seja para a ocorrência destes sinais.

### 2.3.2. Tratamento dos processos zumbis

O sinal SIGCLD se comporta diferentemente dos outros. Se ele e ignorado, a terminação de um processo filho, sendo que o processo pai não está em espera, não irá acarretar a criação de um processo zumbi.
Exemplo: O programa a seguir gera um processo zumbi quando o pai é informado da morte do filho por meio do sinal SIGCLD.

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

int main() {
  if (fork() != 0) while(1) ;
  return(0);
}
```

Para executá-lo basta:

```
# ./test_sigcld &
# ps
```

No próximo programa, o pai ignora o sinal SIGCLD, e seu filho não vai mais se tornar um processo zumbi.

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

int main() {
  signal(SIGCLD,SIG_IGN) ;/* ignora o sinal SIGCLD */
  if (fork() != 0)
  while(1) ;
return(0);
}
```

Observação: A primitiva `signal()` será mais detalhada no decorrer do texto.

### 2.3.3. Sinal SIGHUP: tratamento de aplicações duráveis

O sinal SIGHUP pode ser incômodo quando o usuário deseja que um processo continue a ser executado após o fim de sua seção de trabalho (aplicação durável). De fato, se o processo não trata esse sinal, ele será interrompido pelo sistema no instante de "deslogagem". Existem diferentes soluções para solucionar este problema:
1) Utilizar o comando shell `at` ou `@` que permite de lancar uma aplicação numa certa data, via um processo do sistema, denominado *deamon*. Neste caso, o sinal SIGHUP não terá nenhuma influência. sobre o processo, uma vez que ele não está ligado a nenhum terminal.
2) Incluir no código da aplicação a recepção do sinal SIGHUP.
3) Executar o programa em *background* (na verdade, um processo executado em *background* trata automaticamente o sinal SIGHUP

4) Executar a aplicação associada ao comando `nohup`, que provocará uma chamada a primitiva `trap`, e que redireciona à saída padrão sobre `nohup.out`.

### 2.3.4 Tratamento dos sinais

#### 2.3.4.1. Emissão de um sinal

##### 2.3.4.1.1. Primitiva `kill()`

```
#include <signal.h>
int kill(pid,sig) /* emissao de um sinal */
int pid ; /* id do processo ou do grupo de destino */
int sig ; /* numero do sinal */
```

Valor de retorno: 0 se o sinal foi enviado, -1 se não foi.
A primitiva kill() emite ao processo de número **pid** o sinal de número **sig**. Além disso, se o valor inteiro sig é nulo, nenhum sinal é enviado, e o valor de retorno vai informar se o número de pid é um número de um processo ou não.

Utilização do parâmetro pid:
Se pid > 0: pid designará o processo com ID igual a pid.

Se pid = 0: o sinal é enviado a todos os processos do mesmo grupo que o emissor. Esta possibilidade é geralmente utilizada com o comando shell `kill`. O comando `kill -9 0` irá matar todos os processos rodando em background sem ter de indicar os IDs de todos os processos envolvidos.

Se pid = 1:
{ Se o processo pertence ao super-usuário, o sinal é enviado a todos os processos, exceto aos processos do sistema e ao processo que envia o sinal.
{ Senão, o sinal é enviado a todos os processos com ID do usuário real igual ao ID do usuário efetivo do processo que envia o sinal (é uma forma de matar todos os processos que se é proprietário, independente do grupo de processos ao qual se pertence).

Se pid < 1: o sinal é enviado a todos os processos para os quais o ID do grupo de processos (pgid) é igual ao valor absoluto de pid.
Note finalmente que a primitiva `kill()` e na maioria das vezes executada via o comando `kill` no shell.

##### 2.3.4.1.2. Primitiva `alarm()`

```
#include <unistd.h>
unsigned int alarm(unsigned int secs) /* envia um sinal SIGALRM */
```

Valor de retorno: tempo restante no relógio se já existir um alarme armado anteriormente ou 0 se não existir. Se o secs for igual a 0, ele retorna o valor do tempo restante no relogio, sem portanto rearmar o alarme.
A primitiva alarm() envia um sinal SIGALRM ao processo chamando após um intervalo de tempo **secs** (em segundos) passado como argumento, depois reinicia o relógio de alarme. Na chamada da primitiva, o relógio é reiniciado a **secs** segundos e é decrementado até 0. Esta primitiva pode ser utilizada, por exemplo, para forçar a leitura do teclado dentro de um dado intervalo de tempo. O tratamento do sinal deve

estar previsto no programa, senão o processo será finalizado ao recebê-lo.

```
/* testa os valores de retorno de alarm() */
/* assim que seu funcionamento */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void it_horloge(int sig) /* rotina executada na recepc~ao de SIGALRM
*/
  { printf("recepc~ao do sinal %d :SIGALRM\n",sig) ; }

main() {
  unsigned sec ;
  signal(SIGALRM,it_horloge) ; /* interceptac~ao do sinal */
  printf("Fazendo alarm(5)\n") ;
  sec = alarm(5) ;
  printf("Valor retornado por alarm: %d\n",sec) ;
  printf("Principal em loop infinito (CTRL+c para acabar)\n") ;
  for(;;) ;
}
```

Outro exemplo:

```
/* teste dos valores de retorno de alarm() quando 2
* chamadas a alarm() sao feitas sucessivamente
*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void it_horloge(int sig) /* tratamento do desvio sobre SIGALRM */
{
  printf("recepcao do sinal %d : SIGALRM\n",sig) ;
  printf("atencao, o principal reassume o comando\n") ;
}

void it_quit(int sig) /* tratamento do desvio sobre SIGALRM */
{
  printf("recepcao do sinal %d : SIGINT\n",sig) ;
  printf("Por que eu ?\n") ;
}

int main()
{
  unsigned sec ;
  signal(SIGINT,it_quit); /* interceptacao do ctrl-c */
  signal(SIGALRM,it_horloge); /* interceptacao do sinal de alarme */
  printf("Armando o alarme para 10 segundos\n");
  sec=alarm(10);
  printf("valor retornado por alarm: %d\n",sec) ;
  printf("Paciencia... Vamos esperar 3 segundos com sleep\n");
  sleep(3) ;
  printf("Rearmando alarm(5) antes de chegar o sinal precedente\n");
  sec=alarm(5);
```

```
  printf("novo valor retornado por alarm: %d\n",sec);
  printf("Principal em loop infinito (ctrl-c para parar)\n");
  for(;;);
}
```

Observação: A interceptação do sinal só tem a finalidade de fornecer uma maneira elegante de sair do programa, ou em outras palavras, de permitir um redirecionamento da saída padrão para um arquivo de resultados.
Pode-se notar que o relógio é reinicializado para o valor de 5 segundos durante a segunda chamada de `alarm()`, e que mais ainda, o valor retornado e o estado atual do relógio. Finalmente, pode-se observar que o relógio e decrementado ao longo do tempo. As duas últimas linhas da execução são geradas após um sinal CTRL+c do teclado.
Observação: A função `sleep()` chama a primitiva `alarm()`. Deve-se então utilizá-la com maior prudência se o programa já manipula o sinal SIGALRM.

Exemplo usando `sleep()`:
Implementação de uma versão da função `sleep()` que utiliza as primitivas `pause()` e `alarm()`. O princípio de funcionamento é simples: um processo arma um alarme (via `alarm()`) e se posiciona em pausa (via `pause()`). Na chegada do sinal SIGALRM, `pause()` será interrompida e o processo termina sua execução.

```
/* utilizacao de pause() e de alarm() para
* implementar uma primitiva sleep2 */
#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void nullfcn() /* define-se aqui uma funcao executada quando */
{ } /* o sinal SIGALRM e interceptado por signal() */
/* esta funcao nao faz absolutamente nada */

void sleep2(int secs) /* dorme por secs segundos */
{
  if( signal(SIGALRM,nullfcn) )
  {
    perror("error: reception signal") ;
    exit(-1) ;
  }
  alarm(secs) ; /* inicializa o relogio a secs segundos */
  pause() ; /* processo em espera por um sinal */
}

int main() /* so para testar sleep2() */
{
  if(fork()==0)
  {
    sleep(3) ;
    printf("hello, sleep\n") ;
  }
  else /* pai */
  {
```

```
    sleep2(3) ;
    printf("hello, sleep2\n") ;
  }
  return 0;
}
```

Observação: O interesse da função `nullfunc()` e de se assegurar que o sinal que desperta o processo não provoque o comportamento *default* e que não seja ignorado, de forma a garantir que a pausa (via `pause()`) possa ser interrompida.

## 2.3.4.2. Recepção de sinais:

### 2.3.4.2.1. Primitiva `signal()`

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Valor de retorno: o valor anterior do manipulador do sinal, ou SIG ERR (normalmente -1) quando houver erro. A chamada de sistema `signal()` define um novo manipulador (*handler*) para o sinal especificado pelo número signum. Em outras palavras, ela intercepta o sinal signum. O manipulador do sinal e "setado" para handler, que é um ponteiro para uma função que pode assumir um dos três seguintes valores: SIG DFL: indica a escolha da ação *default* para o sinal. A recepção de um sinal por um processo provoca a terminação deste processo, menos para SIGCLD e SIGPWR, que são ignorados por *default*. No caso de alguns sinais, existe a criação de um arquivo de imagem core no disco.
SIG IGN: indica que o sinal deve ser ignorado: o processo e imunisado contra este sinal. Lembrando sempre que o sinal SIGKILL nunca pode ser ignorado.
Um ponteiro para uma função (nome da função): implica na captura do sinal. A função é chamada quando o sinal chega, e após sua execução, o tratamento do processo recomeça onde ele foi interrompido.
Não se pode proceder um desvio na recepção de um sinal SIGKILL pois esse sinal não pode ser interceptado, nem para SIGSTOP.
Pode-se notar então que é possível modicar o comportamento de um processo na chegada de um dado sinal. E exatamente isso que se passa para um certo número de processos canônicos do sistema: o shell, por exemplo, ao receber um sinal SIGINT irá escrever na tela o prompt (e não sera interrompido).

### 2.3.4.2.2. Primitiva `pause()`

```
#include <unistd.h>
int pause(void) /* espera de um sinal qualquer */
```

Valor de retorno: sempre retorna -1.

A primitiva `pause()` corresponde a uma espera simples. Ela não faz nada, nem espera nada de particular. Entretanto, uma vez que a chegada de um sinal interrompe toda primitiva bloqueada, pode-se dizer que `pause()` espera simplesmente a chegada de um sinal.
Observe o comportamento de retorno clássico de um primitiva bloqueada, isto é o posicionamento de errno em EINTR. Note que, geralmente, o sinal esperado por

`pause()` é o relógio de `alarm()`.

Exemplo:

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

void it_main(sig) /* tratamento sobre o 1o SIGINT */
int sig ;
{
  printf("recepção do sinal número : %d\n",sig) ;
  printf("vamos retomar o curso ?\n") ;
  printf("é o que o os profs insistem em dizer geralmente!\n") ;
}

void it_fin(sig) /* tratamento sobre o 2o SIGINT */
int sig ;
{
  printf("recepção do sinal número : %d\n",sig) ;
  printf("ok, tudo bem, tudo bem ...\n") ;
  exit(1) ;
}
int main()
{
  signal(SIGINT,it_main) ; /* interceptacao do 1o SIGINT */
  printf("vamos fazer uma pequena pausa (cafe!)\n") ;
  printf("digite CTRL+c para imitar o prof\n") ;
  printf("retorno de pause (com a recepcao do sinal): %d\n",pause()) ;
  printf("errno = %d\n",errno) ;
  signal(SIGINT,it_fin) ; /* rearma a interceptacao: 2o SIGINT */
  for(;;) ;
  exit(0) ;
}
```

### 2.3.5. Processos manipulando sinais: Exemplos

#### 2.3.5.1. Herança de sinais com fork()

Os processos filhos recebem a imagem da memória do pai, herdando seu comportamento em relação aos sinais. O próximo exemplo descreve este fenômeno:

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

void fin()
{
  printf("SIGINT para os processos %d\n",getpid()) ;
  exit(0) ;
}
```

```
int main()
{
  signal(SIGQUIT,SIG_IGN) ;
  signal(SIGINT,fin) ;
  if (fork()>0)
  {
    printf("processo pai : %d\n",getpid()) ;
    while(1) ;
  }
  else
  {
    printf("processo filho : %d\n",getpid()) ;
    while(1) ;
  }
  exit(0);
}
```

### 2.3.5.2. Comunicação entre processos

O programa a seguir é um exemplo simples da utlização das primitivas de emissão e recepção de sinais com o objetivo de permitir a comunicação entre dois processos. A execução deste programa permite ainda assegurar que o processo executando a rotina de desvio é mesmo aquele que recebeu o sinal.

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

void it_fils()
{
    printf("- Sim, sim. E darei um jeito nisso eu mesmo...
ARGHH...\n") ;
  kill (getpid(),SIGINT) ;
}

void fils()
{
  signal(SIGUSR1,it_fils) ;
  printf("- Papai conte mais uma vez, como foi que voce me fez?\n") ;
  while(1) ;
}

int main()
{
  int pid ;
  if ((pid=fork())==0) fils() ;
  else {
    sleep(2) ;
    printf("- Filhinho, quer ir passear no reino dos mortos?\n") ;
    kill (pid,SIGUSR1) ;
    sleep(1);
  }
```

```
  exit(0);
}
```

### 2.3.5.3. Controle da progressão de uma aplicação

Todos àqueles que já lançaram programas de simulação ou de cálculo númerico muito longos devem ter pensado numa forma de saber como está progredindo a aplicação durante a sua execução. Isto é perfeitamente possível por meio do envio do comando shell `kill` aos processos associados ao sinal. Os processos podem então, após a recepção deste sinal, apresentar os dados desejados. O exemplo a seguir mostra um programa que ajuda a resolver este problema:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
/* as variaveis a serem editadas devem ser globais */
long somme = 0 ;
void it_verificacao()
{
  long t_date ;
  signal(SIGUSR1, it_verificacao) ;/* reativo SIGUSR1 */
  time(&t_date) ;
  printf("\n Data do teste : %s ", ctime(&t_date)) ;
  printf("valor da soma : %d \n", (int) somme) ;
}

int main()
{
  signal(SIGUSR1,it_verificacao) ;
  printf ("Enviar o sinal USR1 para o processo %d \n",getpid()) ;
  while(1) {
    sleep(1);
    somme++ ;
  }
  exit(0);
}
```

Se o programa for lançado em background, se o usuário digitar o comando shell `kill -USR1` **`pid`**, ele irá obter as variáveis de controle desejadas. A primitiva `ctime()` usada no programa retorna um ponteiro para uma cadeia de caracteres contendo a data sob a forma:

"Data do teste : Wed Jul 26 17:52:42 2006"

## 2.4. Conclusão

Com exceção do sinal SIGCLD, os sinais que são recebidos por um processo não são memorizados: ou eles são ignorados, ou eles põem fim na execução dos processos, ou ainda eles são interceptados e tratados por algum procedimento. Por esta razão, os sinais não são apropriados para a comunicação interprocessos. Uma mensagem

sob a forma de sinal pode ser perdida se o sinal e recebido num momento onde o tratamento para esse tipo de sinal e simplesmente ignorá-lo. Após a captura de um sinal por um processo, esse processo vai readotar seu comportamento *default* em relação a esse sinal. Assim, no caso de se desejar captar um mesmo sinal várias vezes, é conveniente fazer a redefinição do comportamento do processo pela primitiva `signal()`. Geralmente, a interceptação do sinal deve ser rearmada o mais cedo possível (deve ser a primeira instrução efetuada no procedimento de desvio para tratamento do sinal).
Um outro problema é que os sinais têm um comportamento excessivamente abrupto comparado à execução do programa: na sua chegada, eles interrompem o trabalho em curso. Por exemplo, a recepção de um sinal enquanto o processo espera um evento (algo que pode acontecer durante a utilização das primitivas `open()`, `read()`, `write()`, `pause()`, `wait()`,...), laça a execução imediata da rotina de desvio; em seu retorno, a primitiva interrompida reenvia uma mensagem de erro, mesmo sem ser totalmente completada (errno e posicionado em EINTR). Por exemplo, quando um processo pai que intercepta os sinais de interrupção e de abandono está em espera da terminação de um filho, é possível que um sinal de interrupção ou de abandono tire o pai da espera no `wait()` antes que o filho tenha terminado sua execução. Neste caso, um processo <defuntc> será criado. Uma forma de contornar esse problema é ignorar certos sinais antes da chamada de tais primitivas (levando irremediavelmente a outros problemas, uma vez que esses sinais não serão tratados de forma alguma).

# 3. Threads

Assim como os processos, threads são mecanismos que permitem um programa realizar mais de uma operação “simultâneamente”. Além de executar threads concorrentemente; o kernel do Linux os organiza assincronamente, interrompendo cada thread de tempos em tempos de forma que todos tenham chance de ser executados.

Conceitualmente as threads co-existem com o processo, pois threads são unidades menores de execução de um processo. Quando um programa é invocado no Linux, um novo processo é criado e neste mesmo processo uma nova thread é criada, que executa o programa sequencialmente. A referida thread possui a capacidade de criar novas threads de modo que essas threads executam o mesmo código (possívelmente em segmentos de código distintos) no mesmo processo de maneira 'simultânea'.

Diferente do que ocorre quando se cria um novo processo com a função `fork` (vide capítulo referente a processos), durante a criação de uma nova thread, nada é copiado. Ou seja, as threads existente a criada dividem o mesmo espaço de memória, descrição de arquivos e outros resquícios do sistema, do mesmo modo que a thread original. Logo, se em uma dada instância alguma thread muda o valor de alguma variável, a thread subseqüente utilizará a variável modificada. Assim como se uma thread fecha um arquivo, outras threads não poderão ler ou escrever nele. Entretanto, podem-se tirar algumas vantagens desta característica da thread, pois não são necessários mecanismos de comunicação e sincronização complexas.

O GNU/Linux implementa uma API (*application program interface*) conhecida como ptheads um padrão IEEE de threads denominado POSIX (*Portable Operation System Interface*). Todas as funções de threads e tipos de dados são declaradas no arquivo de header `<pthread.h>`. Contudo as funções do pthread não são incluídas nas bibliotecas padrões do C. Ao invés disso, deve ser incluído a implementação das funcionalidades do libpthread, então é necessário adicionar à linha de comando o argumento `-lpthread` para conectar à compilação do código.

## 3.1 Criação de threads

Cada thread de um processo é identificado por um thread ID, que são referidos nos programas escritos em C e C++ pelo tipo `pthread_t`.

Após a criação, cada thread executa uma "*thread function*", que é uma função ordinária que contém o código que a thread deve executar. E depois termina a thread quando a mesma retornar o valor da função (termina de executar a função). No GNU/Linux, funções do thread utilizam apenas um parâmetro do tipo void*, e seu retorno também é do tipo void*.

Para criar uma nova thread utiliza-se a função `pthread_create`, que nos disponibiliza os seguintes elementos:

1) Um ponteiro para variável pthread_t, no qual o número de identificação (ID) da nova thread é armazenado.
2) Um ponteiro para o objeto atribuído à thread. Este objeto controla detalhes de como as threads interagem com o resto do programa.
3) Um ponteiro para a função da thread, que é uma função ordinária do tipo:

   void* (*) (void*)

4) Um valor de argumento para a thread do tipo void*; que é passado para a função somente quando a thread é iniciada.

Uma chamada na função `pthread_create` retorna imediatamente, enquanto a thread original continua a execução do programa. Enquanto isso a nova thread começa executando a *thread function*. Como o Linux agenda as threads assíncronamente, o programa não distingue a ordem de execução das instruções das threads. Uma maneira de se verificar isso é rodando o seguinte programa:

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

/* Imprime 'x' em stderr. */
void* print_xs (void* unused)
{
  while (1)
  fputc ('x', stderr);
  return NULL;
}

int main ()
{
  pthread_t thread_id;
  /* Cria um novo thread. A nova therad irá chamar a função print_xs*/
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Imprime 'o' continuamente em stderr. */
  while (1)
  fputc ('o', stderr);
  return 0;
}
```

Código: CriandoThread.c

Para compilar e conectar esse programa basta utilizar a seguinte linha de comando:

```
$ gcc CriandoThread.c -o CriandoThread -lpthread
```

## 3.2 Passando dados para threads

O argumento da thread possui um método conveniente de passar dados para a thread. Como o tipo de argumento da thread é um void*, não é possível passar um grande conjunto de dados por meio dos argumentos. Mas existem algumas maneiras de contornar esta situação, como passar um ponteiro para uma dada estrutura ou matriz de dados. Uma técnica comum seria definir uma estrutura para cada função thread, que contém os parâmetros que a função thread “espera”.

Utilizar o argumento da thread implica em facilitar o reuso das funções threads em várias threads. Onde todas essas threads executam o mesmo código, mas utilizando dados diferentes. O programa abaixo é similar ao apresentado anteriormente; o programa cria duas threads, onde uma delas imprime 'x' e a outra 'o'. Porém, ao invés de imprimí-los infinitamente, cada um imprime um número de caracteres e depois sai retornando para a função thread. A mesma função thread char_print, é utilizado por ambas as thread, mas cada uma é configurada de maneira distinta ao utilizar a estrutura char_print_parms.

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

struct char_print_parms
{
  char character;
  int count;
};

void* char_print (void* parameters)
{
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;
  for (i = 0; i < p->count; ++i)
  fputc (p->character, stderr);
  return NULL;
}

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
  thread1_args.character = 'x';
  thread1_args.count = 30000;
```

```
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
  return 0;
}
```

Entretanto, existe um problema sério neste código!!! A thread principal (que roda a função main) cria as estruturas de parâmetro da thread (thread1_args e thread2_args) como variáveis locais, e depois passam esses ponteiros para a estrutura dos threads que foram criados. E o que previne o Linux de agendar as threads de modo que a função main termine a execução antes de finalizar as outras threads? Simplesmente nada. Mas, isso ocorrer a memória que contém o parâmetro das estruturas são desalocadas enquanto as outras threads ainda estão acessando.

## 3.3 Juntando threads

Uma solução para forçar que a função main fique ativa até que as outras threads do programa terminem é utilizar uma função que possui funcionalidade equivalente ao wait. Essa função é a `pthread_join()`, que possui dois argumentos: a thread ID da thread que irá esperar e um ponteiro para uma variável void* que receberá o valor de retorno da thread.

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

struct char_print_parms
{
  char character;
  int count;
};

void* char_print (void* parameters)
{
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;
  for (i = 0; i < p->count; ++i)
  fputc (p->character, stderr);
  return NULL;
}

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
  thread1_args.character = 'x';
  thread1_args.count = 30000;
```

```
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
  pthread_join (thread1_id, NULL);
  pthread_join (thread2_id, NULL);
  return 0;
}
```

Por fim, deve-se ter cuidado para que todo dado que irá ser passado como referência para uma thread não será desalocado, por qualquer thread, até que se tenha certeza que a execução da thread tenha terminado.

## 3.4 Cancelando threads

Em circunstâncias normais, uma thread termina quando sua execução finaliza, ou por meio do retorno da função thread ou pela chamada da `pthread_exit()`. Mas é possível que uma thread requisite o término de outra, denomina-se a essa funcionalidade o cancelamento de thread.

Para cancelar uma thread, deve-se chamar a função `pthread_cancel()`, passando como parâmetro a thread ID da thread a ser cancelada.

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

void *thread_function(void *arg);
  int main(){
  int res;
  pthread_t a_thread;
  void *thread_result;
  res = pthread_create(&a_thread, NULL, thread_function, NULL);
  if (res != 0){
    perror("Não foi possível criar a thread!");
    exit(EXIT_FAILURE);
  }
  sleep(5);
  printf("Cancelando a thread ... \n");
  res = pthread_cancel(a_thread);
  if (res != 0){
    perror("Não foi possível cancelar a thread!");
    exit(EXIT_FAILURE);
  }
  printf("Esperando o fim da execução da thread ... \n");
  res = pthread_join(a_thread,&thread_result);
  if (res != 0){
    perror("Não foi possível juntar as threads!");
    exit(EXIT_FAILURE);
  }
```

```
  exit(EXIT_SUCCESS);
}

void *thread_function(void *arg){
  int i, res;
  res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  if (res != 0){
    perror("Falha na pthread_setcancelstate");
    exit(EXIT_FAILURE);
  }
  res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
  if (res != 0){
    perror("Falha na pthread_setcanceltype");
    exit(EXIT_FAILURE);
  }
  printf("Função thread executando. \n");
  for (i = 0; i < 10; i++){
    printf("Thread em execução (%d) ... \n", i);
    sleep(1);
  }
  pthread_exit(0);
}
```

## 3.5. Sincronização de threads

Até então toda abodagem é aplicável apenas à execução de threads concorrente que ocorrem “simultâneamente”, cujo jargão é “pooling” entre threads (supondo que se trata de sistemas mono-processados e “single thread”). Uma vez que não foi inserido nenhuma função de sincronização de threads, o que torna o programa ineficiente. Para tanto, utiliza-se canônicamente um conjunto de funções específicas para melhor manipular e controlar a execução de threads e acessos à segmentos críticos do código.

Serão explorados neste material dois métodos básicos para a solução do problema supracitado. O primeiro é o semáforo que age como um gerenciador de tráfego num cruzamento, onde as vias seria correspondentes à trechos de códigos. O segundo serão as mutexes, que pelo próprio mnemônico sugere - “mutual exclusion”, o mesmo atua como um bloqueador de trechos de códigos.

### 3.5.1. Sincronização utilizando Semáforos POSIX

Uma vez que os sistemas operacionais atuais permitem a existência de um recurso compartilhado entre dois ou mais threads, onde este recurso seja modificado pelos mesmos, exige-se a utilização de mecanismos de sincronização entre os threads. A API POSIX fornece para este fim os semáforos e os mutexes, cujo princípio de funcionamento é semelhante: existe uma sessão crítica de código onde somente um thread poderá estar executando em um dado instante. Isto garante a atomicidade de uma operação, assegurando o comportamento esperado no programa.

Para isso, o POSIX fornece o uso de semáforos e mutex (Mutual Exclusion). Ambos garantem a execução de um único thread em um certo trecho de código, com uma diferença semântica: mutex são indicados para travar (lock) acesso a um recurso comum sendo geralmente utilizados para sincronizar dois threads, enquanto semáforos podem atuam como “porteiros” controlando o acesso de 'n' threads a um recurso. De fato, é possível implementar um mecanismo em função do outro, porém existem problemas onde a semântica dos semáforos se adapta melhor se comparada a mutex.
Os semáforos discutidos abaixo fazem parte do padrão POSIX de extensões para tempo real, servindo unicamente para threads. O outro tipo de semáforos (conhecidos como Semáforos System V) servem para sincronizar processos diferentes e não serão discutidos neste material.
Existem muitas semelhanças na implementação e uso de ambas as técnicas de sincronização, seguindo de forma geral os passos:

- Definição de uma variável de controle;
- Acesso a variável de controle para barrar acesso de outros threads na sessão crítica;
- Liberar outros threads.

Em uma solução mais genérica, desenvolvida por Dijkstra em 1965, os semáforos possuem as seguintes características:

- Seja um semáforo **s**, uma estrutura de dados contendo um contador e um apontador para uma fila de processos bloqueados no semáforo;
- A estrutura de dados pode somente ser acessado por duas operações atômicas (**P** e **V**);
- A operação **P** bloqueia o processo (neste caso, uma thread) que a executa se o valor do semáforo é nulo;
- A operação **V** incrementa o valor do semáforo. Existindo processos ou threads bloqueados, o primeiro da fila do semáforo é desbloqueado;
- As modificações no valor do semáforo são executadas atomicamente;
- Se os dois processos ou threads tentam executar **P**(**s**) ou **V**(**s**), essas operações serão executadas seqüencialmente, em uma ordem arbitrária.
- Os semáforos podem ser usados para exclusão mútua com 'n' processos, quando iniciados com o valor '1'.

As bibliotecas pthread inclui de definição de semáforos não binários e contém primitivas que permitem a inicialização e utilização de semáforos (operções para a inicialização de **P** e **V**). A biblioteca semaphore.h contém a definição do semáforo **s** realizada da seguinte forma:

```
#include <semaphore.h>

sem_t s;
```

Após ter sido declarado, a atribuição do valor inicial do semáforo é feita com a primitiva

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

onde:
`sem` é o endereço da variável semáforo;
`pshared` indica que o semáforo não é compartilhado com threads em outro processo quando for 0;
`value` indica o valor inicial do semáforo.

As operações **P** e **V** em um semáforo são respectivamente, `sem_wait` e `sem_post`, como definidos a seguir:

```
int sem_wait(sem_t *s);
int sem_post(sem_t *s);
```

Geralmente utilizamos os semáforos quando estamos interessados neste tipo de sistema, para um processo denominado `Process`:

```
Process
{
   sem_wait(&s)
     // processo crítico
   sem_post(&s)
    // processo não crítico
}
```

Exemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
sem_t bin_sem;

void *thread_function(void *arg)
{
  sem_wait(&bin_sem); //espera ate o valor do semaforo for
diferente de zero e decrementa ... utilizado na inicializacao!!!

   while(strncmp("end", work_area,3) != 0)//compara com a string
de finalizacao
   {
     printf("You input %d characters\n", strlen(work_area)-
1);//mostra o numero de caracteres digitados
       sem_wait(&bin_sem); //espera ate o valor do semaforo for
diferente de zero e decrementa
   }
   pthread_exit(NULL); //termina a thread
}

int main()
{
   pthread_t a_thread;
   void *thread_result;
   sem_init(&bin_sem, 0 , 0); //inicia o semaforo
   pthread_create(&a_thread, NULL, thread_function, NULL);//cria
uma thread
   printf("Input some text. Enter 'end' to finish\n"); //
```

```
imprime as instrucoes de uso
   while(strncmp("end", work_area, 3) != 0) //compara com a
string de finalizacao
   {
      fgets(work_area, WORK_SIZE, stdin); //le a string do
standard input
      sem_post(&bin_sem); //incrementa o valor do semaforo
   }
   printf("Waiting for thread to finish ...\n");
   pthread_join(a_thread, &thread_result); //juntando a thread -
para garantir que a funcao main termine depois da thread
   printf("Thread joined\n");

   sem_destroy(&bin_sem); //destruindo o semaforo
   exit(0); //sai da funcao main
}
```

O exemplo clássico do produtor/consumidor:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

pthread_t tid1, tid2;
sem_t full, empty, mutex;
#define N 10

int buffer[N];
int i = 0, j = 0; //produtor produz na posicao i e consumidor
consome na posicao j

void *produtor()
{
  for (;;)
    {
      sem_wait(&empty);
      sem_wait(&mutex);
      buffer[i] = 50;
      i = (i + 1) % N;
      sem_post(&mutex);
```

```
      sem_post(&full);
    }
}

void *consumidor()
{
  int j, c;
  for (;;)
    {
      sem_wait(&full);
      sem_wait(&mutex);
      c = buffer[j];
      j = (j + 1) % N;
      sem_post(&mutex);
      sem_post(&empty);
    }
}

int main()
{
  sem_init(&mutex, 0, 1);
  sem_init(&full, 0, 0);
  sem_init(&empty, 0, 10);
  pthread_create(&tid1, NULL, produtor, NULL);
  pthread_create(&tid2, NULL, consumidor, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

Neste exemplo, o semáforo `s0` é inicializado com '`1`' e o `s1` com '`0`'. Se o processo consumidor atuar no processador, o semáforo `s1` ficará bloqueado, até que o produtor tenha depositado um elemento no buffer e executar a operação `sem_post(&s1)`. O produtor por sua vez, ficará bloqueado no semáforo `s0` até que o consumidor execute a operação `sem_post(&s0)`, que ocorrerá sempre que o consumidor retirar o elemento do buffer.

### 3.5.2. Sincronização utilizando Mutex

Uma das formas mais comumente usadas é o "dispositivo de exclusão mútua" (*mutex*). Um *mutex* serve para proteger regiões críticas contra acesso concorrente e para implementar formas mais sofisticadsa de sincronização, como monitores.

O *mutex* funciona como uma trava parecida com as encontradas em armários públicos em aeroportos ou alguns bancos. Se a porta estiver aberta, é só usar (trancar). Se estiver fechada, você deve esperar a sua vez.

O primeiro passo para se usar um *mutex* é criá-lo. Para tanto, basta definir uma variável do tipo `pthread_mutex_t` e inicializá-la com um dos seguintes valores (para Linux):

- `PTHREAD_MUTEX_INITIALIZER` -- Um *mutex* deste tipo suspende indefinidamente a *thread* que tenta usá-lo recursivamente. Isto é, se **A** travar o *mutex* e tentar travá-lo novamente, antes de liberá-lo. Este tipo é o default.
- `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP` -- Para este tipo uma solicitação feita pela *thread* "dona" do *mutex* provoca um erro (`EDEADLK`).
- `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP` -- Este é um *mutex* recursivo.

Uma outra possibilidade é fazer a inicialização manualmente, com

`pthread_mutex_init`

Para requisitar a ``trava'', usa-se a função:

`pthread_mutex_lock(&`mutex`)`

O comportamento desta função depende do tipo do *mutex*, como explicado acima. Se o *mutex* estiver livre, a *thread* passa ser a "dona" e obtém a trava. Se estiver travado, esta chamada bloqueia a *thread* até chegar sua vez.

`pthread_mutex_trylock` faz o mesmo, mas não bloqueia a *thread* se o *mutex* estiver travado. Ao invés disso, ela retorna um código de erro (`EBUSY`).

Para liberar um *mutex* a função é

`pthread_mutex_unlock(&`mutex`)`

Para destruir um objeto *mutex*:

`pthread_mutex_destroy(&`mutex`)`

*Mutex*es **não são** seguros em ambiente assíncronos! As chamadas de travamento e liberação devem estar emparelhadas corretamente. Nunca coloque *mutex*es em tratamento de sinais!

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

int  x;

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void loop() {
  int y,i;
  for (i=0; i< 5000000;i++) {
        pthread_mutex_lock(&mut);
        y  =  x;
```

```
        y += 1;
        x  = y;
        pthread_mutex_unlock(&mut);
  }
}

int  main(int ac)
{
  pthread_t id1,id2;
  x = 0;
  pthread_create(&id1, NULL, (void*) loop, NULL);
  pthread_create(&id2, NULL, (void*) loop, NULL);
  pthread_join(id1, NULL);
  pthread_join(id2, NULL);
  printf("%d\n", x);
}
```

# 4. Comunicação/Sincronismo entre processos

## 4.1. Tubos (*pipes*)

Os tubos (ou pipes) constituem um mecanismo fundamental de comunicação unidirecional entre processos. Eles são um mecanismo de E/S com duas extremidades, ou socketc, correspondendo na verdade a filas de caractereres do tipo FIFO (First In First Out): as informações são introduzidas numa das extremidades (num socket) e retiradas em outra (outro socket). Conectando dois fluxos de dados de um processo para outro Esse tipo de abordagem é familiar para os usuários de UNIX/Linux, pois usualmente se concatena comandos no *shell*, além de alimentar a saída de um processo à entrada de outro. Para concatenar comandos no shell, a seguinte sintaxe deve ser empregada:

```
$ comando1 | comando2
```

Os tubos são implementados como arquivos (eles possuem um i-node na tabela de indexação), mesmo se eles não têm um nome definido no sistema. Assim, o programa especifica sua entrada e sua saída somente como um descritor na tabela de índices, o qual aparece como uma variável ou constante, podendo a fonte (entrada) e o destino (saída) serem alteradas sem que para isso o texto do programa tenha que ser alterado. No caso do exemplo, na execução de comando1 a saída padrão (stdout) é substituda pela entrada do tubo. No comando2, de maneira similar, a entrada padrão (stdin) é substituída pela saída de comando1.

Figura 1: Exemplo do esquema de um pipe.

No caso, o shell organiza as entradas e saídas padrões do sistema, onde:

1) A entrada do comando1 é oriunda do teclado do terminal.

2) A saída do comando1 alimenta a entrada do comando2.

3) A saída do comando2 é conectada ao monitor do terminal.

No decorrer deste capítulo, abordar-se-á a metodologia de se escrever um programa para obter um resultado similar ao supracitado, além de utilizar pipes para conectar múltiplos processos juntos de modo a permitir a implementação de um sistema simples do tipo cliente/servidor.

### 4.1.1. Particularidades dos tubos

Uma vez que eles não têm nomes, os tubos de comunicação são temporários, existindo apenas em tempo de execução do processo que os criou;

A criação dos tubos é feita através de uma primitiva especial: `pipe()`;

Vários processos podem fazer leitura e escrita sobre um mesmo tubo, mas nenhum mecanismo permite diferenciar as informações na saída do tubo;

A capacidade é limitada (em geral a 4096 bytes). Se a escrita sobre um tubo continua mesmo depois do tudo estar completamente cheio, ocorre um bloqueio (*dead-lock*);

Os processos comunicando-se por meio dos tubos devem ter uma ligação de parentesco, e os tubos religando processos devem ter sidos abertos antes da criação dos filhos (veja a passagem de descritores de arquivos abertos durante a execução do `fork()`;

E impossível fazer qualquer "movimentação" no interior de um tubo.

Com a finalidade de estabelecer um diálogo entre dois processos usando tubos, é necessário a abertura de um tubo em cada direção.

### 4.1.2. Criação de um tubo

A Primitiva `pipe()`

```
#include <unistd.h>
int pipe(int desc[2]);
```

Valor de retorno: 0 se a criação tiver sucesso, e -1 em caso de falha. A primitiva `pipe()` cria um par de descritores, apontando para um i-node, e coloca-os num vetor apontado por desc:

desc[0] contém o número do descritor pelo qual pode-se ler no tubo;

desc[1] contém o número do descritor pelo qual pode-se escrever no tubo

Assim, a escrita sobre desc[1] introduz dados no tubo, e a leitura em desc[0] extrai dados do tubo.

### 4.1.3. Seguranca do sistema

No caso em que todos os descritores associados aos processos susceptíveis de ler num tubo estiverem fechados, um processo que tenta escrever neste tubo deve receber um sinal SIGPIPE, sendo então interrompido se ele não possuir uma rotina de tratamento deste sinal.

Se um tubo esta vazio, ou se todos os descritores susceptíveis de escrever sobre ele estiverem fechados, a primitiva `read()` retornará o valor 0 (fim de arquivo lido).

Exemplo de emissão de um sinal SIGPIPE:

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

void it_sigpipe()
{
  printf("Sinal SIGPIPE recebido \n") ;
}

int main()
{
  int p_desc[2] ;
  signal(SIGPIPE,it_sigpipe) ;
  pipe(p_desc) ;
  close(p_desc[0]) ; /* fechamento do tubo em leitura */
  if (write(p_desc[1],"0",1) == -1)
    perror("Error write") ;
  exit(0);
}
```

Neste exemplo, tenta-se escrever num tubo sendo que ele acaba de ser fechado em leitura; o sinal SIGPIPE é emitido e o programa e desviado para a rotina de tratamento deste sinal. No retorno, a primitiva `write()` retorna -1 e `perror()` imprime na tela a mensagem de erro.

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

int main()
{
  int i, p_desc[2] ;
  char c ;
  pipe(p_desc) ; /* criacao do tubo */
  write(p_desc[1],"AB",2) ; /* escrita de duas letras no tubo */
  close(p_desc[1]) ; /* fechamento do tubo em escrita */
  /* tentativa de leitura no tuno */
  for (i=1; i<=3; i++) {
    if ( (read(p_desc[0],&c,1) == 0) )
      printf("Tubo vazio\n") ;
    else
      printf("Valor lido: %c\n",c) ;
  }
  exit(0);
}
```

Este exemplo mostra que a leitura num tubo e possível, mesmo se este tiver sido fechado para a escrita. Obviamento, quando o tubo estiver vazio, `read()` vai retornar o valor 0.

### 4.1.4. Aplicações das primitivas de entrada e saída

É possível utilizar as funções da biblioteca padrão sobre um tubo já aberto, associando a esse tubo – por meio da função `fopen()` - um ponteiro apontando sobre uma estrutura do tipo FILE:

`write()` : os dados são escritos no tubo na ordem em que eles chegam. Quando o tubo está cheio, `write()` se bloqueia esperando que uma posição seja liberada. Pode-se evitar este bloqueio utilizando-se o flag O_NDELAY.

`read()` : os dados são lidos no tubo na ordem de suas chegadas. Uma vez retirados do tubo, os dados não poderão mais serem relidos ou restituídos ao tubo.

`close()` : esta função é mais importante no caso de um tubo que no caso de um arquivo. Não somente ela libera o descritor de arquivo, mas quando o descritor de arquivo de escritura está fechado, ela funciona como um fim de arquivo para a leitura.

`dup()` : esta primitiva combinada com `pipe()` permite a implementação dos comandos religados por tubos, redirecionando a saída padrão de um comando para a entrada padrão de um outro.

### 4.1.5. Implementação de um comando com tubos

Este exemplo permite observar como as primitivas `pipe()` e `dup()` podem ser combinadas com o objetivo de produzir comandos *shell* do tipo `ls|wc|wc`. Note que e necessário fechar os descritores não utilizados pelos processos que executam a rotina.

```
/* este programa e equivalente ao comando shell ls|wc|wc */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int p_desc1[2] ;
int p_desc2[2] ;

void faire_ls()
{
  /* saida padrao redirecionada para o 1o. tubo */
  close (1) ;
  dup(p_desc1[1]) ;
  close(p_desc1[1]) ;
  /* fechamento dos descritores nao-utilizados */
  close(p_desc1[0]) ;
  close(p_desc2[1]) ;
  close(p_desc2[0]) ;
  /* executa o comando */
  execlp("ls","ls",0) ;
  perror("impossivel executar ls ") ;
}

void faire_wc1()
{
  /* redirecionamento da entrada padrao para o 1o. tubo*/
  close(0) ;
  dup(p_desc1[0]) ;
```

```
  close(p_desc1[0]) ;
  close(p_desc1[1]) ;
  /* redirecionamento da saida padrao para o 2o. tubo*/
  close(1) ;
  dup(p_desc2[1]) ;
  close(p_desc2[1]) ;
  close(p_desc2[0]) ;
  /* executa o comando */
  execlp("wc","wc",0) ;
  perror("impossivel executar o 1o. wc") ;
}

void faire_wc2()
{
  /* redirecionamento da entrada padrao para o 2o. tubo*/
  close (0) ;
  dup(p_desc2[0]) ;
  close(p_desc2[0]) ;
  /* fechamento dos descritores nao-utilizados */
  close(p_desc2[1]) ;
  close(p_desc1[1]) ;
  close(p_desc1[0]) ;
  /* executa o comando */
  execlp("wc","wc",0) ;
  perror("impossivel executar o 2o. wc") ;
}

int main()
{
  /* criacao do primeiro tubo*/
  if (pipe(p_desc1) == -1)
    perror("Impossivel criar o 1o. tubo") ;
  /* criacao do segundo tubo */
  if (pipe(p_desc2) == -1)
    perror("impossivel criar o 1o. tubo") ;
  /* lancamento dos filhos */
  if (fork() == 0) faire_ls() ;
    if (fork() == 0) faire_wc1() ;
```

```
      if (fork() == 0) faire_wc2() ;
        exit(0);
}
```

### 4.1.6. Comunicação entre pais e filhos usando um tubo: exemplo

Este exemplo enfoca mais uma vez a herança dos descritores por meio do `fork()`. O programa cria um tubo para a comunicação entre um processo pai e seu filho.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define DATA "Testando envio de mensagem usando pipes"

int main()
{
  int sockets[2], child;
  char buf[1024];
  /* criacao de um tubo */
  if ( pipe(sockets) == -1 ) {
    perror("Error opening stream socket pair") ;
    exit(10);
  }
  /* criacao de um filho */
  if ( (child = fork()) == -1)
    perror ("Error fork") ;
  else if (child) {
    /* Esta ainda e a execuc~ao do pai. Ele l^e a mensagem do filho */
     if ( close(sockets[1]) == -1) /* fecha o descritor nao utilizado
*/
      perror("Error close") ;
    if (read(sockets[0], buf, 1024) < 0 )
      perror("Error: reading message");
      printf("-->%s\n", buf);
      close(sockets[0]);
  }
  else {
```

```
    /* Esse e o filho. Ele escreve a mensagem para seu pai */
     if ( close(sockets[0]) == -1) /* fecha o descritor nao utilizado
*/
      perror("Error close") ;
    if (write(sockets[1], DATA, sizeof(DATA)) < 0 )
      perror("Error: writing message");
    close(sockets[1]);
  }
  sleep(1);
  exit(0);
}
```

Ao excutar este programa, um tubo é criado pelo processo pai, o qual logo após faz um *fork*. Quando um processo faz um *fork*, a tabela de descritores do pai é automaticamente copiada para o processo filho. No programa, o pai faz um chamada de sistema `pipe()` para criar um tubo. Esta rotina cria um tubo e inclui na tabela de descritores do processos os descritores para os *sockets* associados às duas extremidades do tubo. Note que as extremidades do tubo não são equivalentes: o socket com índice 0 está sendo aberto para leitura, enquanto que o de índice 1 esta sendo aberto somente para a escrita. Isto corresponde ao fato de que a entrada padrão na tabela de descritores é associada ao primeiro descritor, enquanto a saída padrão é associada ao segundo.

Após ser criado, o tubo será compartilhado entre pai e filho após a chamada *fork*. A tabela de descritores do processo pai aponta para ambas as extremidades do tubo. Após o *fork*,

ambas as tabelas do pai e do filho estarão apontando para o mesmo tubo (herança de descritores). O filho então usa o tubo para enviar a mensagem para o pai.

### 4.1.6. Utilização dos tubos

É possível que um processo utilize um tubo tanto para a escrita quanto para a leitura de dados. Este tubo não tem mais a função específica de fazer a comunicação entre processos, tornando-se muito mais uma implementação da estrutura de um arquivo. Isto permite, em certas máquinas, de ultrapassar o limite de tamanho da zona de dados. O mecanismo de comunicação por tubos apresenta um certo número de incovenientes como o não armazenamento da informação no sistema e a limitação da classe de processos podendo trocar informações via tubos.

## 4.2. As FIFOS ou tubos com nome

Uma FIFO combina as propriedades dos arquivos e dos tubos: como em um arquivo, ela tem um nome e todo processo que tiver as autorizações apropriadas pode abrí-lo em leitura ou escrita (mesmo se ele não tiver ligação de parentesco com o criador do tubo). Assim, um tubo com nome, se ele não estiver destruído, persistirá no sistema,

mesmo após a terminação do processo que o criou. Uma vez aberto, uma FIFO se comporta muito mais como um tubo do que como um arquivo: os dados escritos são lidos na ordem "First In First Out", seu tamanho é limitado, e além disso, é impossível de se movimentar no interior do tubo.

### 4.2.1. Criação de um tubo com nome

Primitiva `mknod()`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Valor de retorno: 0 se a criação do tubo tem sucesso, e -1 caso contrário. A primitiva `mknod()` permite a criação de um nó (arquivo normal ou especial, ou ainda um tubo) cujo nome é apontado por *pathname*, especificado por *mode* e *dev*. O argumento *mode* especifica os direitos de acesso e o tipo de nó a ser criado. O argumento *dev* não é usado na criação de tubos com nome, devendo ter seu valor igual a 0 neste caso.

A criação de um tubo com nome é o único caso onde o usuário normal tem o direito de utilizar esta primitiva, reservada habitualmente ao super-usuário. Afim de que a chamada `mknod()` tenha sucesso, é indispensável que o flag S_IFIFO esteja “setado” e nos parâmetros de *mode* os direitos de acesso ao arquivo estejam indicados: isto vai indicar ao sistema que uma FIFO vai ser criada e ainda que o usuário pode utilizar `mknod()` mesmo sem ser root.

Dentro de um programa, um tubo com nome pode ser eliminado utilizando a primitiva `unlink(const char *pathname)`, onde *pathname* indica o nome do tubo a ser destruído.

Exemplo:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
```

```
{
  printf("Vou criar um tubo de nome 'fifo1'\n") ;
  printf("Vou criar um tubo de nome 'fifo2'\n") ;
  if (mknod("/tmp/fifo1",S_IFIFO|0666, 0) == -1) {
    perror("Criacao de fifo1 impossivel") ;
    exit(1) ;
  }
  if (mknod("/tmp/fifo2",S_IFIFO|0666, 0) == -1) {
    perror("Criacao de fifo2 impossivel") ;
    exit(1) ;
  }
  sleep(10) ;
  printf("Vou apagar o tubo de nome 'fifo1'\n") ;
  unlink("/tmp/fifo1") ;
  exit(0);
}
```

Para verificar o funcionamento do programa descrito acima, deve-se executá-lo em background e pode-se verificar utilizando o comando shell `$ ls -l /tmp/fifo*` que os tubos denominados fifo1 e fifo2 foram criados e depois de dez segundos, que o tubo fifo1 será destruído. Resultando em:

```
$ ls -l /tmp/fifo*
prw-r--r-- 1 root root 0 2006-08-01 19:13 /tmp/fifo1
prw-r--r-- 1 root root 0 2006-08-01 19:13 /tmp/fifo2
```

No instante em que as fifos foram criadas até que o fifo1 é destruída, quando nos resta apenas o pipe fifo2. Observe que a presença do bit 'p' indica que fifo1 e fifo2 são tubos (pipes) com nome. Pode-se notar ainda que o tubo denominado fifo2 permanece no sistema, mesmo após destruição do processo que o criou. Mas a eliminação de um tubo com nome pode ser feita a partir do shell, como no caso de um arquivo comum, usando-se o comando `rm`.

### 4.2.2. Manipulação das FIFOs

As instruções `read()` e `write()` são bloqueantes:

Na tentativa de leitura de uma FIFO vazia, o processo ficará em espera até que haja um preenchimento suficiente de dados dentro da FIFO;

Na tentativa de escrita de uma FIFO cheia, o processo irá esperar que a FIFO seja sucessivamente esvaziada para comecar a preenchê-la com seus dados. Neste caso ainda, a utilização do flag O_NDELAY permite manipular o problema de bloqueio, uma

vez que nesse caso, as funções `read()` e `write()` vão retornar um valor nulo.

# 5. Sockets

## 5.1. Introdução:

Será abordado neste capítulo, um outro método de comunicação entre processos, mas com algumas diferenças. Até o presente momento, o interesse era restrito apenas ao espaço de arquivos, compartilhamento de memória fisica ou informações apenas de processos executando em uma máquina.

Uma nova ferramenta introduzida pela versão UNIX da Universidade de Berkley, denominada interface de socket, que seria uma extensão do conceito de pipe, com a possibilidade de comunicação através de rede de computadores. Um processo em uma máquina pode utilizar socket para comunicar de um processo para outro, utilizando um modelo cliente/servidor no qual é utilizado em redes distribuídas. Note que este modelo também viabiliza comunicação entre processos de uma mesma máquina.

Interfaces de sockets também são utilizados em sistemas da Microsoft Windows, especificamente chamados de Windows Socket ou Winsock. Este serviço é disponibilizado pela biblioteca dinâmica Winsock.dll. Cuja implementação é compativel com os sistemas UNIX.

Portanto, socket nada mais é que um mecanismo de comunicação que permitem sistemas do tipo cliente/servidor que podem ser aplicabilidade local (em uma única máquina) ou em uma rede de computadores. Permitindo ainda a implementação de múltiplos clientes comunicando com um único servidor.

Formalmente, sockets são definidos por um grupo de quatro elementos:

1) Número de identificação ou endereço do host remoto.
2) Número da porte do host remoto.
3) Número de identificação ou endereço do host local.
4) Número da porte do host local.

Uma das primeiras formas de se desenvolver aplicações distribuídas em um ou mais computadores foi com o uso de sockets. Com isso foram desenvolvidas diversas aplicações cliente/servidor onde cliente(s) e servidor poderiam estar em máquinas diferentes, distantes umas das outras. Atualmente tem-se outras ferramentas de linguagem para implementar software distribuído, mas é interessante notar como vários dos conceitos da API de sockets permanecem verdadeiros ainda hoje. No texto a seguir será visto o que é a API de socket, as seus principais funções e procedimentos e uma aplicação exemplo escrita em C.

## 5.2. Interface de Programa Aplicativo (API) de sockets

Aplicativos cliente e servidor utilizam protocolos de transporte para se comunicarem. Quando um aplicativo interage com o software de protocolo, ele deve especificar

detalhes, como por exemplo: se é um servidor ou um cliente (isto é, se esperará passivamente ou iniciará ativamente a comunicação). Além disso, os aplicativos que se comunicam devem especificar detalhes adicionais (por exemplo, o remetente deve especificar os dados a serem enviados, e o receptor deve especificar onde os dados recebidos devem ser colocados).

A interface que um aplicativo usa quando interage com o software de protocolo de transporte é conhecida como Interface de Programa Aplicativo (*Application Program Interface*, API). Uma API define um conjunto de operações que um aplicativo pode executar quando interage com o software de protocolo. Deste modo, a API determina a funcionalidade que está disponível a um aplicativo e também a dificuldade de se criar um programa para usar aquela funcionalidade.

A maioria dos sistemas de programação define uma API dando um conjunto de procedimentos que o aplicativo pode chamar e os argumentos que cada procedimento espera. Normalmente, uma API contém um procedimento separado para cada operação básica. Por exemplo, uma API poderia conter um procedimento que é usado para estabelecer uma comunicação e outro procedimento que é usado para enviar dados.

Os padrões de protocolos de comunicação usualmente não especificam uma API que os aplicativos devem usar para interagir com os protocolos. Em vez disso, os protocolos especificam as operações gerais que devem ser fornecidas e permitem que cada sistema operacional defina a API específica que um aplicativo deve usar para executar as operações. Deste modo, um padrão de protocolo poderia sugerir que uma operação seja necessária para permitir que um aplicativo envie dados, e a API especifica o nome exato da função e o tipo de cada argumento.

Embora os padrões de protocolo permitam que os projetistas de sistema operacional escolham uma API, muitos adotaram a API de sockets (sockets). A API de sockets está disponível para muitos sistemas operacionais, incluindo sistema usados em computadores pessoais (por exemplo, Microsoft Windows) como também vários sistemas UNIX (por exemplo, Solaris da Sun Microsystems).

A API de sockets se originou como parte do sistema operacional BSD UNIX. O trabalho foi financiado por uma bolsa do governo americano, através da qual a University of California em Berkeley desenvolveu e distribuiu uma versão de UNIX que continha protocolos de ligação inter-redes TCP/IP. Muitos vendedores de computadores portaram o sistema BSD para seu hardware, e o usuaram como base em seus sistema operacionais comerciais. Deste modo, a API de sockets se tornou o padrão de fato na indústria.

## 5.3. Sockets e Bibliotecas de Socket

No UNIX BSD e nos sistemas que derivaram dele, as funções de sockets fazem parte do próprio sistema operacional. Como os sockets se tornaram mais extensamente usados, os vendedores de outros sistemas decidiram acrescentar uma API de sockets a seus sistemas. Em muitos casos, em vez de modificar seu sistema operacional básico, os vendedores criaram uma biblioteca de sockets que fornece a API de sockets. Isto é, o vendedor criou uma biblioteca de procedimentos onde cada procedimento tem o mesmo nome e argumentos que as funções de socket.

Do ponto de vista de um programador de aplicativos, uma biblioteca de sockets

fornece a mesma semântica que uma implementação de sockets no sistema operacional. O programa chama procedimentos de sockets, que são então providos por procedimentos do sistema operacional ou por rotinas de biblioteca. Deste modo, um aplicativo que usa sockets pode ser copiado para um novo computador, compilado, carregado junto com a biblioteca de sockets do computador e então executado - o código de origem não precisa ser mudado quando o programa for portado de um sistema de computador para outro (na prática, bibliotecas de sockets são raramente perfeitas, e às vezes ocorrem diferenças menores entre a implementação padrão de uma API de sockets e uma biblioteca de sockets - por exemplo, na forma em que os erros são tratados).

Apesar de semelhanças aparentes, bibliotecas de socket têm uma implementação completamente diferente que uma API de sockets nativa provida por um sistema operacional. Diferentemente de rotinas de socket nativo, que são parte do sistema operacional, o código para procedimentos de biblioteca de sockets é unido ao programa aplicativo e reside no espaço de endereçamento do aplicativo. Quando um aplicativo chamar um procedimento da biblioteca de sockets, o controle passa para a rotina de biblioteca que, por sua vez, faz uma ou mais chamadas para as funções de sistema operacional subjacente para obter o efeito desejado. É interessante perceber que funções providas pelo sistema operacional subjacente não precisam de maneira alguma se assemelhar à API de sockets - as rotinas na biblioteca de sockets escondem do aplicativo o sistema operacional nativo e apresentam somente uma interface de sockets. Para resumir:uma biblioteca de sockets pode fornecer a aplicativos uma API de sockets em um sistema de computador que não forneça sockets nativos. Quando um aplicativo chama um dos procedimentos de socket, o controle passa para uma rotina de biblioteca que faz uma ou mais chamadas para o sistema operacional subjacente para implementar a função de socket.

## 5.4. Comunicação de Socket e E/S do UNIX

Como os sockets foram originalmente desenvolvidos como parte do sistema operacional UNIX, eles empregam muitos conceitos encontrados em outras partes do UNIX. Em particular, sockets são integrados com a E/S - uma aplicação se comunica através de uma rotina similar ao socket que forma um caminho para a aplicação transferir dados para um arquivo. Deste modo, compreender sockets exige que se entenda as facilidades de E/S do UNIX.

UNIX usa um paradigma *open-read-write-close* para toda E/S; o nome é derivado das operações de E/S básicas que se aplicam tanto a dispositivos como a arquivos. Por exemplo, um aplicativo deve primeiro chamar *open* para preparar um arquivo para acesso. O aplicativo então chama *read* ou *write* para recuperar dados do arquivo ou armazenar dados no arquivo. Finalmente, o aplicativo chama *close* para especificar que terminou de usar o arquivo.

Quando um aplicativo abre um arquivo ou dispositivo, a chamada *open* retorna um descritor, um inteiro pequeno que identifica o arquivo; o aplicativo deve especificar o descritor ao solicitar transferência de dados (isto é, o descritor é um argumento para o procedimento de *read* ou *write*). Por exemplo, se um aplicativo chama *open* para acessar um arquivo de nome qualquer, o procedimento de abertura poderia retornar o descritor 4. Uma chamada subsequente para write que especifica o descritor 4 fará com que sejam escritos dados no arquivo nomeado anteriormente; o nome de arquivo não aparece na chamada para *write*.

Para exemplificar o conteúdo exposto nesta seção, far-se-á o uso de dois programas: o primeiro de trata de um servidor de socket localizado em "/tmp/tst" ou em um argumento (ao comentar a linha contendo o `argv[1]` e comentando a linha seguinte) que lê e permanece aberto até que receber a mensagem "quit."

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

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    text = (char*) malloc (length);
    read (client_socket, text, length);
    fprintf (stderr,"%s", text);
    if (!strcmp (text, "quit."))
      return 1;
    free (text);
  }
}

int main (int argc, char* const argv[])
{
  //const char* const socket_name = argv[1];
  const char* const socket_name = "/tmp/tst";
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind(socket_fd, &name, SUN_LEN (&name));
```

```
  listen (socket_fd, 5);
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;
             client_socket_fd   =   accept(socket_fd,   &client_name,
&client_name_len);
    client_sent_quit_message = server (client_socket_fd);
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);
  close (socket_fd);
  unlink (socket_name);
  return 0;
}
```

Já o segundo código se trata de um socket cliente, que se conecta com o servidor supracitado ou qualquer outro enviando textos por meio de conexões socket na linha de comando, i.e., `$ ./socket_client /tmp/tst mensagem`. Mandando assim a mensagem de texto "mensagem" para /tmp/tst.

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

void write_text (int socket_fd, const char* text)
{
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
```

```
  int socket_fd;
  struct sockaddr_un name;
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  connect (socket_fd, &name, SUN_LEN (&name));
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```

### 5.5. Sockets, Descritores e E/S de Rede

A comunicação de sockets usa também a abordagem de descritor. Antes de uma aplicativo poder usar protocolos para se comunicar, o aplicativo deve solicitar ao sistema operacional que crie um socket que será usado para comunicação. O aplicativo passa o descritor como argumento quando ele chama procedimentos para transferir dados através da rede; o aplicativo não precisa especificar detalhes sobre o destino remoto cada vez que transfere dados.

Em uma implementação UNIX, os sockets são completamente integrados com o restante da E/S. O sistema operacional fornece um único conjunto de descritores para arquivos, dispositivos, comunicação entre processos e comunicação de rede. Como resultado, procedimentos como read e write são bastante gerais - uma aplicação pode usar o mesmo procedimento para enviar dados para outro programa, um arquivo ou através de uma rede. Em terminologia corrente, o descritor representa um objeto, e o procedimento write representa o método aplicado àquele objeto. O objeto subjacente determina como o método é aplicado.

A vantagem principal de um sistema integrado reside em sua flexibilidade: pode ser escrito um único aplicativo que transfira dados para uma localização arbitrária. Se o aplicativo recebe um descritor que corresponde a um dispositivo, o aplicativo envia dados para o dispositivo. Se o aplicativo recebe um descritor que corresponde a um arquivo, o aplicativo armazena dados no arquivo. Se o aplicativo recebe um descritor que corresponde a um socket, o aplicativo envia dados através de uma inter-rede para uma máquina remota. Ou seja, quando um aplicativo cria um socket, o aplicativo recebe um descritor; um inteiro pequeno, usado para referenciar o socket. Se um sistema usa o mesmo espaço de descritores para sockets e demais E/S, pode ser usado um único aplicativo para comunicação de rede bem como para transferência local de dados.

### 5.6. Parâmetros e a API de Sockets

A programação de socket difere da E/S convencional porque um aplicativo deve especificar diversos detalhes para usar um socket. Por exemplo, um aplicativo deve escolher um protocolo de transporte em particular, fornecer o endereço de protocolo de uma máquina remota e especificar se o aplicativo é um cliente ou um servidor. Para

acomodar todos os detalhes, cada socket tem diversos parâmetros e opções - um aplicativo pode fornecer valores para cada um deles.

Como opções e argumentos deveriam ser representados em uma API? Para evitar que exista uma única função de socket com argumentos separados para cada opção, os projetistas da API de sockets escolheram definir muitas funções. Essencialmente, um aplicativo cria um socket e então invoca funções para especificar em detalhes como será usado o socket. A vantagem da abordagem de sockets é que a maioria das funções tem três ou menos argumentos; a desvantagem é que um programador deve se lembrar de chamar múltiplas funções ao usar sockets.

## 5.7. Procedimentos que implementam a API de Sockets

### O Procedimento `socket()`

O procedimento socket cria um socket e retorna um descritor inteiro:

```
descriptor = socket( protofamily, type, protocol )
```

O argumento protofamily especifica a família de protocolos a ser usada com o socket. Por exemplo, o valor PF_INET é usado para especificar o suíte de protocolo TCP/IP, e PF_DECnet é usado para especificar protocolos da Digital Equipment Corporation.

O argumento type especifica o tipo de comunicação que o socket usará. Os dois tipos mais comuns são a transferência de stream orientada à conexão (especificada com o valor SOCK_STREAM) e uma transferência sem conexão orientada a mensagens (especificada com o valor SOCK_DGRAM).

O argumento protocol especifica um protocolo de transporte particular usado com o socket. Ter um argumento protocol além de um argumento type permite a um único suíte de protocolo incluir dois ou mais protoclos que forneçam o mesmo serviço. Naturalmente, os valores que podem ser usados com o argumento protocol dependem da família de protocolos. Por exemplo, embora o suíte de protocolos TCP/IP inclui o protocolo TCP, o suíte AppleTalk não o inclue.

### O Procedimento `close()`

O procedimento close informa ao sistema para terminar o uso de um socket (a interface de Sockets do Windows usa o nome closesocket em vez de close). Ele assume a forma:

```
close(socket)
```

onde socket é o descritor para um socket sendo fechado. Se o socket está usando um protocolo de transporte orientado à conexão, o close termina a conexão antes de fechar o socket. O fechamento de um socket imediatamente termina seu uso - o descritor é liberado, impedindo que o aplicativo envie mais dados, e o protocolo de transporte pára de aceitar mensagens recebidas direcionadas para o socket, impedindo que o aplicativo receba mais dados.

### O Procedimento `bind()`

Quando criado, um socket não tem um endereço local e nem um endereço remoto. Um servidor usa o procedimento bind para prover um número de porta de protocolo em que o servidor esperará por contato. O bind leva três argumentos:

```
bind( socket, localaddr, addrlen )
```

O argumento socket é o descritor de um socket que foi criado, mas não previamente amarrado (com bind); a chamada é uma requisição que ao socket seja atribuído um número de porta de protocolo particular. O argumento localaddr é uma estrutura que especifica o endereço local a ser atribuído ao socket, e o argumento addrlen é um inteiro que especifica o comprimento do endereço. Como os sockets podem ser usados com protocolos arbitrários, o formato de um endereço depende do protocolo sendo usado. A API de sockets define uma fomra genérica usada para representar endereços, e então exige que cada família de protocolos especifique como seus endereços de protocolo usam a forma genérica.

O formato genérico para representar um endereço é definido como uma estrutura sockaddr. Embora várias versões tenham sido liberadas, o código de Berkely mais recente define uma estrutura sockaddr com três campos:

```
struct sockaddr {
   u_char sa_len;       /* comprimento total do endereço */
   u_char sa_family;    /* família do endereço           */
   char sa_data[14];    /* o endereço propriamente dito  */
};
```

O campo sa_len consiste em um único octeto que especifica o comprimento do endereço. O campo sa_family especifica a família à qual um endereço pertence (a constante simbólica AF_INET é usada para endereços TCP/IP). Finalmente, o campo sa_data contém o endereço. Cada família de protocolos define o formato exato dos endereços usados com o campo sa_data de uma estrutura sockaddr. Por exemplo, os protocolos TCP/IP usam a estrutura sockaddr_in para definir um endereço:

```
struct sockaddr_in {
   u_char  sin_len;          /* comprimento total do endereço      */
   u_char  sin_family;       /* família do endereço                */
   u_short sin_port;         /* número de porta de protocolo       */
   struct  in_addr sin_addr; /* endereço IP de computador          */
   char    sin_zero[8];      /* não usado (inicializado com zero)  */
};
```

Os primeiros dois campos da estrutura sockaddr_in correspondem exatamente aos dois primeiros campos da estrutura genérica sockaddr. Os últimos três campos definem a forma exata do endereço que o protocolo TCP/IP espera. Existem dois pontos a serem observados. Primeiro, cada endereço identifica tanto um computador como um aplicativo particular naquele computador. O campo sin_addr contém o endereço IP do computador, e o campo sin_port contém o número da porta de protocolo de um aplicativo. Segundo, embora o TCP/IP necessite somente de seis octetos para armazenar um endereço completo, a estrutura genérica sockaddr reserva quatorze octetos. Deste modo, o campo final da estrutura sockaddr_in define um campo de 8 octetos de zeros, que preenchem a estrutura para o mesmo tamanho que

sockaddr.

Diz-se que um servidor chama *bind* para especificar o número da porta de protocolo em que o servidor aceitará um contato. Porém, além de um número de porta de protocolo, a estrutura sockaddr_in contém um campo para um endereço IP. Embora um servidor possa escolher preencher o endereço IP ao especificar um endereço, fazer isso causa problemas quando um host tiver múltiplas interfaces (multihomed) porque significa que o servidor aceita apenas requisições enviadas a um endereço específico. Para permitir que um servidor opere em um host com múltiplas interfaces, a API de sockets inclui uma constante simbólica especial, INADDR_ANY, que permite a um servidor usar uma porta específica em quaisquer dos endereços IP do computador. Para resumir:

A estrutura sockaddr_in define o formato que o TCP/IP usa para representar um endereço. Embora a estrutura contenha campos para endereços IP e número de porta de protocolo, a API de sockets inclui uma constante simbólica que permite a um servidor especificar uma porta de protocolo em quaisquer dos endereços IP do computador.

### O Procedimento `listen()`

Depois de especificar uma porta de protocolo, um servidor deve instruir o sistema operacional para colocar um socket em modo passivo para que o socket possa ser usado para esperar pelo contato de clientes. Para fazer isso, um servidor chama o procedimento listen, que toma dois argumentos:

```
listen( socket, queuesize )
```

O argumento socket é o descritor de um socket que foi criado e amarrado a um endereço local, e o argumento queuesize especifica um comprimento para a fila de requisição do socket.

O sistema operacional cria uma fila de requisição separada para cada socket. Inicialmente, a fila está vazia. À medida que chegam requisições de clientes, elas são colocadas na fila; quando o servidor pede para recuperar uma requisição recebida do socket, o sistema retorna a próxima requisição da fila. Se a fila está cheia quando chega uma requisição, o sistema rejeita a requisição. Ter uma fila de requisições permite que o sistema mantenha novas requisições que chegam enquanto o servidor está ocupado tratando de uma requisição anterior. O argumento permite que cada servidor escolha um tamanho máximo de fila que é apropriado para o serviço esperado.

### O Procedimento `accept()`

Todos os servidores iniciam chamando socket para criar um socket e bind para especificar um número de porta de protocolo. Depois de executar as duas chamadas, um servidor que usa um protocolo de transporte sem conexão está pronto para aceitar mensagens. Porém, um servidor que usa um protocolo de transporte orientado à conexão exige passos adicionais antes de poder receber mensagens: o servidor deve chamar listen para colocar o socket em modo passivo, e deve então aceitar uma requisição de conexão. Uma vez que uma conexão tenha sido aceita, o servidor pode usar a conexão para se comunicar com um cliente. Depois de terminar a comunicação, o servidor fecha a conexão.

Um servidor que usa transporte orientado à conexão deve chamar o procedimento accept para aceitar a próxima requisição de conexão. Se uma requisição está presente

na fila, accept retorna imediatamente; se nenhuma requisição chegou, o sistema bloqueia o servidor até que um cliente forme uma conexão. A chamada accept tem a forma:

```
newsock = accept( socket, cadddress, caddresslen )
```

O argumento socket é o descritor de um socket que o servidor criou e amarrou (bound) a uma porta de protocolo específica. O argumento cadddress é o endereço de uma estrutura do tipo sockaddr e caddresslen é um ponteiro para um inteiro. Accept preenche os campos do argumento caddresscaddresslen o comprimento do endereço. Finalmente, accept cria um novo socket para a conexão e retorna o descritor do novo socket para quem chamou. O servidor usa o novo socket para se comunicar com o cliente e então fecha o socket quando terminou. Enquanto isso, o socket original do servidor permanece inalterado - depois de terminar a comunicação com um cliente, o servidor usa o socket original para aceitar a próxima conexão de um cliente.

### O Procedimento `connnect()`

Os clientes usam o procedimento connect para estabelecer uma conexão com um servidor específico. A forma é:

```
connect( socket, saddress, saddresslen )
```

O argumento socket é o descritor de um socket no computador do cliente a ser usado para a conexão. O argumento saddress é uma estrutura sockaddr que especifica o endereço do servidor e o número de porta de protocolo (a combinação de um endereço IP e um número de porta de protocolo é às vezes chamado de um endereço de endpoint). O argumento saddresslen especifica o comprimento do endereço do servidor medido em octetos.

Quando usado com um protocolo de transporte orientado à conexão como TCP, connect inicia uma conexão em nível de transporte com o servidor especificado. Na essência, connect é o procedimento que um cliente usa para contatar um servidor que tinha chamado accept.

É interessante observar que um cliente que usa um protocolo de transporte sem conexão pode também chamar connect. Porém, fazer isso não inicia uma conexão ou faz com que pacotes cruzem a inter-rede. Em vez disso, connect meramente marca o socket conectado e registra o endereço do servidor.

Para entender por que faz sentido conectar com um socket que usa transporte sem conexão, lembre que protocolos sem conexão exigem que o remetente especifique um endereço de destino com cada mensagem. Em muitos aplicativos, porém, um cliente sempre contata um único servidor. Deste modo, todas as mensagens vão ao mesmo destino. Em tais casos, um socket conectado fornece uma taquigrafia - o cliente pode especificar o endereço do servidor uma única vez, não sendo necessário especificar o endereço com cada mensagem. A questão é:

O procedimento connect, que é chamado por clientes, tem dois usos. Quando usado com transporte orientado à conexão, connect estabelece uma conexão de transporte com um servidor especificado. Quando usado com transporte sem conexão, connect registra o endereço do servidor no socket, permitindo ao cliente enviar muitas mensagens para o mesmo servidor sem exigir que o cliente especifique o endereço de destino em cada mensagem.

### Os Procedimentos `send()`, `sendto()` e `sendmsg()`

Tanto os clientes quanto os servidores precisam enviar informações. Normalmente, um cliente envia uma requisição, e um servidor envia uma resposta. Se o socket está conectado, o procedimento send pode ser usado para transferir dados. send tem quatro argumentos:

```
send( socket, data, length, flags )
```

O argumento socket é o descritor do socket a ser usado, o argumento data é o endereço em memória dos dados a serem enviados, o argumento length é um inteiro que especifica o número de octetos de dados, e o argumento flags contém bits que requisitam opções especiais (muitas opções visam à depuração do sistema e não estão disponíveis para programas convencionais cliente e servidor).

Os procedimentos sendto e sendmsg permitem a um cliente ou servidor enviar uma mensagem usando um socket não-conectado; ambos exigem que um destino seja especificado. sendto toma o endereço de destino como um argumento. Ele toma a forma:

```
sendto( socket, data, length, flags, destaddress, addresslen )
```

Os primeiros quatro argumentos correspondem aos quatro argumentos do procedimento send. Os últimos dois argumentos especificam o endereço de um destino e o comprimento daquele endereço. O formato do endereço no argumento destaddress é a estrutura sockaddr (especificamente, a estrutura sockaddr_in quando usada com TCP/IP).

O procedimento sendmsg executa a mesma operação que sendto, mas abrevia os argumentos definindo uma estrutura. A lista de argumentos mais curta pode tornar os programas que usam sendmsg mas fáceis de ler:

```
sendmsg( socket, msgstruct, flags )
```

O argumento msgstruct é uma estrutura que contém informações sobre o endereço de destino, o comprimento do endereço, a mensagem a ser enviada e o comprimento da mensagem:

```
struct msgstruct {                /* estrutura usada por sendmsg
*/
    struct sockaddr *m_saddr;/* pointer para endereço de destino
*/
    struct datavec *m_dvec;    /* pointer para mensagem (vetor)
*/
    int m_dvlength;                  /* num. de itens em vetor
*/
   struct access *m_rights; /* pointer para acessar lista de direitos
*/
    int m_alength;                   /* num. de itens em lista
*/
};
```

Os detalhes da estrutura de mensagem não têm importância - deve ser visto como um modo de combinar muitos argumentos em uma única estrutura. A maioria dos aplicativos usa apenas os primeiros três campos, que especificam um endereço de protocolo de destino e uma lista de itens de dados que inclui a mensagem.

### Os Procedimentos `recv()`, `recvfrom()` e `recvmsg()`

Um cliente e um servidor precisam receber dados enviados pelo outro. A API de sockets fornece vários procedimentos que podem ser usados. Por exemplo, um aplicativo pode chamar recv para receber dados de um socket conectado. O procedimento tem a forma:

```
recv( socket, buffer, length, flags )
```

O argumento socket é descritor de um socket a partir do qual dados devem ser recebidos. O argumento buffer especifica o endereço em memória em que a mensagem recebida deve ser colocada e o argumento length especifica o tamanho do buffer. Finalmente, o argumento flags permite que se controle detalhes (por exempjlo, para permitir a um aplicativo extrair uma cópia de uma mensagem recebida sem remover a mensagem do socket).

Se um socket não está conectado, ele pode ser usado para receber mensagens de um conjunto arbitrário de clientes. Em tais casos, o sistema retorna o endereço do remetente junto com cada mensagem recebida. Os aplicativos usam o procedimento recvfrom para receber uma mensagem e o endereço do seu remetente:

```
recvfrom( socket, buffer, length, flags, sndraddr, saddrlen )
```

Os primeiro quatro argumentos correspondem aos argumentos de recv; os dois argumentos adicionais, sndraddr e saddrlen, são usados para registrar o endereço IP do remetente. O argumento sndraddr é um ponteiro para uma estrutura sockaddr em que o sistema escreve o endereço do remetente, e o argumento saddrlen é um ponteiro para um inteiro que o sistema usa para registrar o comprimento do endereço. recvfrom registra o endereço do remetente exatamente da mesma forma que sendto espera. Deste modo, se um aplicativo usa recvfrom para receber uma mensagem, enviar uma resposta é fácil - o aplicativo simplesmente usa o endereço registrado como um destino para a resposta.

A API de sockets inclui um procedimento de entrada equivalente ao procedimento de saída sendmsg. O procedimento recvmsg opera como recvfrom, mas exige menos argumentos. Ele tem a forma:

```
recvmsg( socket, msgstruct, flags )
```

onde o argumento msgstruct dá o endereço de uma estrutura que possui o endereço para uma mensagem recebida como também posições para o endereço IP do remetente. O msgstruct registrado por recvmsg usa exatamente o mesmo formato que a estrutura esperada por sendmsg. Deste modo, os dois procedimentos funcionam bem para receber uma mensagem e enviar uma resposta.

## 5.8. Ler e Escrever com Sockets

A API de sockets foi originalmente projetada para ser parte do UNIX, que usa read e write para E/S. Consequentemente, sockets permitem que aplicativos também usem read e write para transferir dados. Como send e recv, read e write não têm argumentos que permitam a especificação de um destino. Em vez disso, read e write têm cada um três argumentos: um descritor de socket, a localização de um buffer em memória usado para armazenar os dados e o comprimento do buffer de memória. Deste modo, read e write devem ser usados com sockets conectados.

A vantagem principal de se usar read e write é a generalidade - um programa aplicativo pode ser criado para transferir dados de ou para um descritor sem saber se o descritor corresponde a um arquivo ou a um socket. Deste modo, um programador pode usar um arquivo em disco local para testar um cliente ou servidor antes de tentar se comunicar através de uma rede. A desvantagem principal de se usar read e write é que uma implementação de biblioteca de sockets pode introduzir sobrecarga adicional na E/S de arquivo em qualquer aplicativo que também use sockets.

## 5.9. Outros Procedimentos de Socket

A API de socktes contém outros procedimentos úteis. Por exemplo, após um servidor chamar o procedimento accept para aceitar uma requisição de conexão recebida, o servidor pode chamar o procedimento getpeername para obter o endereço completo do cliente remoto que iniciou a conexão. Um cliente ou servidor pode chamar também gethostname para obter informações sobre o computador em que ele está executando.

Os sockets têm muitos parâmetros e opções. Dois procedimentos de propósito geral são usados para configurar opções de socket ou obter uma lista de valores correntes. Um aplicativo chama o procedimento setsockopt para armazenar valores em opções de socket, e o procedimento getsockopt para obter os valores correntes das opções. As opções são principalmente usadas para tratar dos casos especiais (por exemplo, para aumentar o desempenho mudando o tamanho do buffer interno que o software de protocolo usa).

Dois procedimentos são usados para traduzir entre endereços IP e nomes de computador. O procedimento gethostbyname retorna o endereço IP para um computador dado o seu nome. Os clientes usam frequentemente gethostbyname para traduzir um nome inserido por um usuário em um endereço IP correpondente exigido pelo software de protocolo.

O procedimento gethostbyaddr fornece um mapeamento inverso - dado um endereço IP referente a um computador, ele retorna o nome do computador. Os clientes e servidores podem usar gethostbyaddr ao mostrar informações para uma pessoa ler.

## 5.10. Sockets, Threads e Herança

Uma vez que muitos servidores são concorrentes, a API de sockets é projetada para funcionar com programas concorrentes. Embora os detalhes dependam do sistema operacional subjacente, as implementações da API de sockets aderem ao seguinte princípio: Cada novo thread criado herda uma cópia de todos os sockets abertos do thread que os criou.

Para entender como os servidores usam herança de sockets, é importante saber que os sockets usam um mecanismo de contagem de referências. Quando um socket é primeiramente criado, o sistema inicializa a contagem de referências do socket para 1; o socket existe desde que a contagem de referências permaneça positiva. Quando um programa cria um thread adicional, o sistema fornece ao thread uma lista de todos os sockets que o programa possui e incrementa a contagem de referências de cada socket em 1. Quando um thread chama close para um socket, o sistema decrementa a contagem de referências no socket em 1 e remove o socket da lista de threads (se o thread termina antes de fechar os sockets, o sistema chama automaticamente close

em cada um dos sockets abertos que o thread tinha).

O thread principal de um servidor concorrente cria um socket que o servidor usa para aceitar conexões recebidas. Quando chega uma requisição de conexão, o sistema cria um novo socket para a nova conexão. Logo depois do thread principal criar um thread de serviço para tratar da nova conexão, ambos os threads têm acesso aos sockets novo e velho, e a contagem de referências de cada socket tem o valor 2. Porém, o thread principal não usará o novo socket, e o thread de serviço não usará o socket original. Portanto, o thread principal chama close para o novo socket, e o thread de serviço chama close para o socket original, reduzindo a contagem de referência de cada um para 1.

Depois de um thread de serviço terminar, ele chama close no novo socket, reduzindo a contagem de referência para zero e fazendo com que o socket seja apagado. Deste modo, o tempo de vida dos sockets em um servidor concorrente pode ser resumido:

O socket que um servidor concorrente usa para aceitar conexões existe desde que o thread servidor principal execute; um socket usado para uma conexão específica existe apenas enquanto existe o thread para tratar daquela conexão.

Até agora foi descrita a API de sockets, incluindo procedimentos individuais que podem ser chamados, os argumentos usados com cada procedimento e a operação que o procedimento executa. Agora API de sockets será abordada examinando um cliente e um servidor simples que usam sockets para se comunicar. Embora não mostre todos os projetos possíveis de cliente e servidor, o código exemplo elucidará o propósito dos procedimentos de sockets mais importantes. Em particular, o exemplo mostra uma sequência de chamadas de procedimento com sockets e mostra a diferença entre as chamadas usadas em um cliente e as chamadas usadas em um servidor.

## 5.11. Comunicação Orientada à Conexão

Um cliente e um servidor devem selecionar um protocolo de transporte que suporte serviço sem conexão ou um que suporte serviço orientado à conexão. O serviço sem conexão permite a um aplicativo enviar uma mensagem a um destino arbitrário a qualquer momento; o destino não precisa concordar que ele aceitará a mensagem antes da transmissão acontecer. Em contraste, um serviço orientado à conexão exige que dois aplicativos estabeleçam uma conexão de transporte antes que possam ser enviados dados. Para estabelecer uma conexão, os aplicativos interagem com o software de protocolo de transporte em seus computadores locais, e os dois módulos de transporte trocam mensagens através da rede. Após ambos os lados concordarem que uma conexão seja estabelecida, os aplicativos podem enviar dados.

Um Serviço Exemplo:

Um cliente e servidor exemplo ajudarão a explicar muitos detalhes da interação orientada à conexão e mostrarão como o software de um serviço orientado à conexão usa sockets. Para manter os programas pequenos e manter-se o foco nas chamadas de sockets, escolhe-se um serviço trivial: o servidor sempre que um cliente contata o servidor e o cliente por meio de eco recebe o texto enviado para o servidor.

Para simplificar a implementação e depuração, o serviço é projetado para usar ASCII. Um cliente forma uma conexão com um servidor e espera por uma saída. Sempre que chega uma requisição de conexão, um servidor cria uma mensagem em forma de

ASCII imprimível, envia a mensagem através da conexão e então fecha a conexão. O cliente apresenta os dados que recebe e termina.

Sequência de Chamadas de Procedimento com Sockets:

A figura a seguir mostra a sequência de procedimentos com sockets que chamam o cliente e servidor.

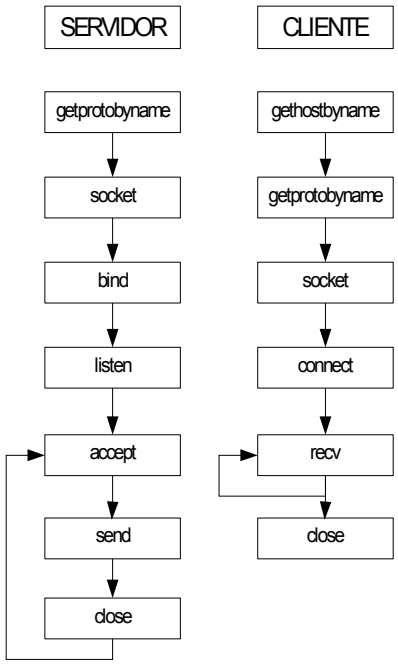


**Figura 5.1**: Esquemático de procedimentos de socket para servidor e cliente.

Como mostra a figura, o servidor chama sete procedimentos de socket e o cliente chama seis. O cliente começa chamando os procedimentos de biblioteca gethostbyname para converter o nome de um computador para um endereço IP e getprotobyname para converter o nome de um protocolo para a forma binária interna usada pelo procedimento de socket. O cliente então chama socket para criar um socket e connect para conectar o sockt a um servidor. Uma vez que for estabelecida a conexão, o cliente chama repetidamente recv para receber os dados que o servidor envia. Finalmente, após os dados terem sido recebidos, o cliente chama close para fechar o socket.

O servidor chama também getprotobyname para gerar o identificador binário interno para o protoco1lo antes de chamar socket para criar um socket. Uma vez que um socket foi criado, o servidor chama bind para especificar uma porta de protocolo local para o socket e listen para colocar o socket em modo passivo. O servidor então insere um laço infinito em que ele chama accept para aceitar a próxima requisição de conexão recebida, send para enviar uma mensagem para o cliente e close para fechar a nova conexão. Depois de fechar uma conexão, o servidor chama accept para extrair a próxima conexão recebida.

A sequência de chamadas de procedimento com sockets no cliente e servidor exemplo. O servidor deve chamar listen antes que o cliente chame connect.

Código-exemplo para cliente

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

#define RCVBUFSIZE 32

void DieWithError(char *errorMessage)
{
    printf("\n DieWithError \n");
    exit(1);
}

int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in echoServAddr;
    unsigned short echoServPort;
    char *servIP;
    char *echoString;
    char echoBuffer[RCVBUFSIZE];
    unsigned int echoStringLen;
    int bytesRcvd, totalBytesRcvd;

    if ((argc < 3) || (argc > 4))
    {
          fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo
Port>]\n", argv[0]);
       exit(1);
    }

    servIP = argv[1];
    echoString = argv[2];
```

```
    if (argc == 4)
        echoServPort = atoi(argv[3]);
    else
        echoServPort = 7;

    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    memset(&echoServAddr, 0, sizeof(echoServAddr));
    echoServAddr.sin_family      = AF_INET;
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);
    echoServAddr.sin_port        = htons(echoServPort);

          if (connect(sock, (struct sockaddr *) &echoServAddr,
sizeof(echoServAddr)) < 0)
        DieWithError("connect() failed");

    echoStringLen = strlen(echoString);

    if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
          DieWithError("send() sent a different number of bytes than
expected");

    /* Receive the same string back from the server */
    totalBytesRcvd = 0;
    printf("Received: ");
    while (totalBytesRcvd < echoStringLen)
    {
        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <=
0)
                    DieWithError("recv() failed or connection closed
prematurely");
        totalBytesRcvd += bytesRcvd;
        echoBuffer[bytesRcvd] = '\0';
        printf(echoBuffer);
    }

    printf("\n");
    close(sock);
```

```
    exit(0);
}
```

Código-exemplo para servidor:

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

#define MAXPENDING 5

void DieWithError(char *errorMessage);
void HandleTCPClient(int clntSocket);

int main(int argc, char *argv[])
{
    int servSock;
    int clntSock;
    struct sockaddr_in echoServAddr;
    struct sockaddr_in echoClntAddr;
    unsigned short echoServPort;
    unsigned int clntLen;

    if (argc != 2)
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);

    if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    memset(&echoServAddr, 0, sizeof(echoServAddr));
```

```
    echoServAddr.sin_family = AF_INET;
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    echoServAddr.sin_port = htons(echoServPort);

          if (bind(servSock, (struct sockaddr *) &echoServAddr,
sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    if (listen(servSock, MAXPENDING) < 0)
        DieWithError("listen() failed");

    for (;;)
    {
        clntLen = sizeof(echoClntAddr);

              if ((clntSock = accept(servSock, (struct sockaddr *)
&echoClntAddr, &clntLen)) < 0)
            DieWithError("accept() failed");

                                    printf("Handling client %s\n",
inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock);
    }
    /* NOT REACHED */
}
```

### 5.12. Serviço de Stream e Múltiplas Chamadas `recv()`

Embora o servidor faça somente uma chamada a send para enviar dados, o código cliente faz iterações para receber dados. Durante cada iteração, o cliente chama recv para obter dadso; a iteração pára quando o cliente obtiver uma condição de fim de arquivo (isto é, uma contagem de zero).

Na maioria dos casos, o TCP no computador do servidor colocará a mensagem inteira em um único segmento de TCP e então transmitirá o segmento através de uma inter-rede TCP/IP em um datagrama IP. Porém, o TCP não garante que os dados serão enviado sem um único segmento, nem garante que cada chamada de recv retornará exatamente a mesma quantidade de dados que o servidor transferiu em uma chamada a send. Em vez disso, o TCP afirma somente que os dados serão entregues em ordem, com cada chamada de recv retornando um ou mais octetos de dados. Consequentemente, um programa que chama recv deve estar preparado para fazer repetidas chamadas até que todos os dados tenham sido extraídos.

## 5.13. Procedimentos de Socket e Bloqueamento

A maioria dos procedimentos na API de sockets é síncrona ou bloqueante da mesma maneira que a maioria das chamadas de E/S. Isto é, quando um programa chama um procedimento de socket, o programa é suspenso até que o procedimento seja completado (enquanto suspenso, um programa não usa a CPU). Não existe nenhum prazo para a suspensão - a operação pode levar um tempo arbitrariamente longo.

Para entender como clientes e servidores usam procedimentos bsloqueantes, primeiro cnsidere o servidor. Depois de criar um socket, amarrar uma porta de protocolo e colocar o socket em modo passivo, o servidor chama accept. Se um cliente já solicitou uma conexão antes de o servidor chamar accept, a chamada retorna imediatamente. Se o servidor aslcança a chamada accept antes que qualquer cliente requisite uma conexão, o servidor será suspenso até que uma requisição chegue. De fato, o servidor gasta a maior parte de seu tempo suspenso na chamada accept.

As chamadas para procedimentos de socket no código podem também bloquear. Por exemplo, algumas implementações do procedimento de biblioteca gethostbyname enviam uma mensagem através de uma rede para um servidor e esperam por uma resposta. Em tais casos, o cliente permanece suspenso até que a resposta seja recebida. Semelhantemente, a chamada para connect bloqueia até que o TCP possa executar o 3-way handshake para estabelecer uma conexão.

Talvez a suspensão mais importante aconteça durante a trasmissão de dados. Depois da conexão ser estabelecida, o cliente chama recv. Se nenhum dado foi recebido na conexão, a chamada bloqueia. Deste modo, se o servidor tem uma fila de requisições de conexão, o cliente permanecerá bloqueado até que o servidor envie dados.

## Bibliografia:

[1] R. Stone, N. Matthew, Beginning Linux Programming, 2nd Edition, Wrox, 2001.

[2] M. Mitchell, J. Oldman, A. Samual, Advanced Linux Programming, 1st Edition, New Riders Publishing 2001.

[3] C. A. S. Santos. Programação em Tempo Real, UFRN, 2000.

[3] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux Device Drivers, 3rd Edition, O'Reilly. 2005.

[4] A. E. M. Brito, Sistemas Operacionais, UFCG, 2004.