Programação em C no GNU/Linux

Lucas Correia Villa Real
lucasvr@gobolinux.org

29 de Fevereiro de 2004

Conteúdo

Capítulo 1

Operação do GNU/Linux

1.1 Obtendo acesso ao GNU/Linux

Distribuições convencionais de GNU/Linux fornecem duas maneiras para que o usuário autentique-se no computador: uma delas é através de login gráfico, e a outra é através de login via console.

A segunda maneira é a mais comum, onde é exigido um nome de usuário e uma senha. Todo o sistema GNU/Linux mantém uma conta de super-usuário, normalmente associada ao nome *root*, e contas opcionais para cada usuário que irá operar este host.

Normalmente são utilizados dois arquivos para efetuar a autenticação de um usuário: o `passwd` e o `shadow`, localizados no diretório de configurações de programas da distribuição, freqüentemente o `/etc`. O primeiro arquivo mantém uma listagem com os nomes de usuários, seu identificador único no sistema (um valor inteiro), o grupo principal ao qual ele pertence, seu nome completo (além de outros dados opcionais) e a *shell* que será utilizada em suas sessões.

Após verificar a existência deste usuário a partir do `passwd`, o arquivo `shadow` é processado. Este arquivo contém uma listagem das senhas de cada usuário, criptografadas. A senha fornecida pelo usuário é processada pelo mesmo algoritmo, e é feito um *matching* para verificar se ambas senhas criptografadas são idênticas, encerrando o processo de autenticação do usuário no host e dando a ele o acesso a um console com uma shell.

Existem outras formas de autenticação, como o Linux-PAM, que faz uso de módulos de autenticação. Assim, é possível escrever módulos com rotinas específicas para autenticar o usuário, como a validação de algum cartão magnético, reconhecimento facial ou ainda pela voz.

1.2 Como executar comandos sobre o GNU/Linux

As distribuições GNU/Linux costumam ser compostas por um conjunto de pacotes base, que capacitam a utilização de diferentes serviços, além de interfaces para permitir que o usuário faça uso do sistema – normalmente compostas por uma *shell*, um servidor para a interface gráfica e um gerenciador de janelas.

Shells são interfaces em modo texto, onde comandos digitados pelo usuário são interpretados e executados. Existem diferentes implementações destas interfaces, dentre as quais podem-se destacar o `ASH, BASH, KSH, CSH, TCSH` e o `ZSH`. Estas

implementações diferenciam-se pela sintaxe na qual o usuário utiliza para a execução de tarefas e pelos recursos que cada uma oferece. Esta sessão trata da operação básica sobre o BASH, um dos interpretadores mais comuns, presente em grande parte das distribuições atuais.

Após realizar o `login` no sistema, um prompt fica disponível ao usuário, aguardando a entrada de algum comando. A sintaxe padrão para a execução de um programa no BASH é:

```
$ programa
```

A execução deste comando irá criar um novo processo, enquanto o processo pai (o BASH) irá aguardar pela sua finalização para continuar a execução de outros comandos. Nota-se que essa é a mesma semântica obtida com comandos executados no MS-DOS, onde a shell (command.com, no caso) executa o programa de forma síncrona, ou seja: Espera que os filhos terminem para retornar o controle do terminal ao usuário.

Em alguns casos é desejável que o programa seja executado de forma asíncrona, ou seja, deseja-se continuar a ter acesso ao console antes mesmo que o programa tenha terminado. Nestes casos, pode-se executar o programa usando a seguinte forma:

```
$ programa &
```

O "&" indica que o processo filho relativo à execução de "programa" será criado, mas que o processo pai (o BASH) irá continuar seu fluxo de execução, sem aguardar pelo término do processo filho.

Bem como a execução de programas, freqüentemente é desejável redirecionar a saída do programa para algum outro dispositivo que não a saída padrão (*stdout*). Da mesma forma, isto pode querer ser feito para redirecionar a saída de erro padrão (*stderr*), ou modificar a entrada de dados para que seja feita a partir de um outro fluxo que não a entrada padrão (*stdin*).

A forma que o BASH utiliza para fazer o controle de redirecionamentos é através dos operadores "<"e ">", processados na ordem em que aparecem, da esquerda para a direita. Caso o primeiro caracter do operador de redirecionamento seja ">", o redirecionamento refere-se à saída padrão (*stdout*).

Também há casos (em caso de depuração de mensagem de erros, por exemplo) onde-se quer direcionar ambas saídas (*stdout* e *stderr*) para um arquivo, o que pode ser obtido com:

```
$ programa >& arquivo_com_mensagens
```

1.3 Obtendo ajuda sobre comandos

Cada programa *base* do GNU/Linux costuma apresentar uma página de manual, contendo explicações sobre a sua operação.

Existem dois formatos predominantes como forma de documentação de aplicativos no GNU/Linux: as *man pages* e as *info pages*.

O que as diferencia é a forma na qual a informação é apresentada ao usuário. As *man pages* apresentam basicamente:

- A sinopse do programa;
- Uma descrição do que ele faz;

- Opções e argumentos que são tratados por este programa;
- Referências à páginas manuais semelhantes que sejam relevantes;
- Problemas conhecidos;
- Autor(es).

Além disso, uma *man page* é disposta na forma de uma página contínua de informações, podendo muitas vezes tornar difícil a localização de determinados assuntos.

As *info pages* solucionam este problema, apresentando a documentação em um formato que permite uma navegação mais fácil, principalmente quando existe um grande volume de dados a ser informado. Este formato normalmente é utilizado para detalhar a documentação de um programa, enquanto as *man pages* explicam os recursos do programa de forma mais simplista.

1.4 Consoles virtuais

O kernel Linux oferece suporte a dispositivos de terminais virtuais que suportam dispositivos de vídeo e teclado. Eles são chamados de "virtuais" pela possibilidade de executar várias instâncias de terminais virtuais – também chamados de consoles virtuais – em um único terminal físico. Este recurso é bastante útil, por permitir que um terminal virtual colete mensagens do sistema e *warnings*, enquanto outro terminal virtual seja utilizado para gerenciar uma sessão de um usuário em modo texto, e um terceiro terminal rodando uma sessão X, todas em paralelo.

A maneira utilizada para alternar entre os diferentes consoles virtuais é normalmente realizada através da combinação de teclas `Alt-<Fn>`.

Para modificar as fontes utilizadas em um terminal, utiliza-se o comando `setfont(8)`.

1.5 Processos

Qualquer tarefa atribuída ao GNU/Linux é um processo. Todo comando executado gera um número de processo (PID), e através deste número é possível manipular este processo. Alguns destes programas que permitem a manipulação de processos são:

- `top(1)`: lista os processos em execução em tempo real. Ele mostra algumas informações sobre o sistema, bem como uma lista das tarefas sendo gerenciadas pelo kernel Linux. Algumas das informações incluem o nome do processo e a quantidade de memória e CPU consumidas por ele.
- `ps(1)`: lista os processos atuais. Vários parâmetros podem ser utilizados, como a opção "-x", que lista todos os processos da máquina, e a opção "-u", que inclui o autor do processo. Outros parâmetros úteis são o "-C *nome*", que listam os processos com um nome específico, e "-U *usuário*", que lista todos os processos pertencentes à *usuário*.
- `nice(1)`: executa um comando com uma prioridade diferente de escalonamento. Prioridades no Linux vão de -20 (prioridade alta) à 19 (prioridade baixa). A prioridade padrão para execução de processos é 0.

- `renice(1)`: altera a prioridade de escalonamento de algum processo.
- `kill(1)`: envia um sinal para um processo. Uma lista de sinais encontra-se disponível na *man page* do `signal(7)`.
- `killall()`: envia um sinal para todos os processos especificados na linha de comando através de seu nome.
- `nohup()`: roda algum comando, ignorando sinais de *hangup*. Este comando é útil quando deseja-se rodar algum programa e logo após efetuar o logoff da máquina: o programa continuará em execução até o término de sua execução.
- `jobs`: caso tenha algum programa (job) sendo executado em *background*, este comando mostra-os. Este comando é implementado pela shell (BASH e ZSH o implementam). A forma de deixar um programa em background é disparando-o com o parâmetro "&", como visto na Sessão 1.2, ou pressionando a tecla CTRL+Z durante a execução do programa.
- `fg`: também pertinente à controle de programas em background, este comando permite trazer de volta algum job que esteja em background. Caso haja mais de um, é necessário especificar o seu número, listado com o comando `jobs`. Algumas shells utilizam o nome do procsso, ao invés do seu número, para tornar a dar o controle a ele.
- `bg`: assim como o `fg`, faz com que o controle seja dado a um processo. No entanto, este modo permite que o programa continue sua execução em uma *sub-shell*. Assim, a shell fica liberada para que o usuário continue a execução de comandos nela, e o programa sai do estado de background e torna a executar. Este comando é bastante utilizado para chamar programas gráficos no ambiente *X* sem bloquear o terminal.

Capítulo 2

Ambiente de desenvolvimento

2.1 Compilador GCC

"GCC" [STA 2002] é uma abreviação do termo *GNU Compiler Collection*. Ele leva este nome pelo fato de que várias versões do compilador estão integradas, com suporte à linguagens como C, C++, Objective-C, Ada, Fortran, Java e Treelang. Além disso, ele representa tanto o nome geral do compilador quanto o nome do compilador de programas C, onde a abreviação passa a ter o significado de "*GNU C Compiler*". Este é o compilador padrão de qualquer distribuição GNU/Linux.

Quando o GCC é invocado, ele normalmente realiza quatro etapas para gerar o executável: pré-processamento, compilação, montagem e ligação, sempre nesta ordem, e sendo que é possível parar o processo no final de cada etapa. Os primeiros três estágios aplicam-se a um único arquivo fonte, e encerram produzindo um arquivo objeto. A ligação combina todos os arquivos objetos especificados como entrada em um arquivo executável. Estes passos podem ser melhor descritos como:

- *Pré-processamento:* Esta etapa é responsável pela resolução de diretrizes do pré-processador, como `#define, #if, #include`. Nesta fase, o GCC utiliza o utilitário `cpp`.
- *Compilação:* Nesta fase é produzida a linguagem de montagem dos arquivos de entrada.
- *Montagem:* Produz o arquivo objeto `.o`, levando em conta a linguagem de montagem dos arquivos de entrada. Nesta etapa, o GCC utiliza o utilitário `gas` (*GNU Assembler*), ou o montador nativo `as`, caso ele não esteja disponível.
- *Ligação:* Nesta fase os arquivos `.o` e as bibliotecas são "colocados" no executável. O utilitário usado nessa fase é o `ld` (*GNU Linker*).

Para parar o GCC em alguma determinada etapa, podem ser utilizados os seguintes comandos:

- *Pré-processamento:*
  `$ gcc -E teste.c -o teste.i`
  Este comando redireciona a saída do pré-processador para o arquivo `teste.i`.

- *Compilação:*
  `$ gcc -S teste.c`
  O arquivo `teste.s` será gerado, já em linguagem assembly da arquitetura.

- *Montagem:*
  `$ gcc -c teste.c`
  Gera o arquivo objeto `teste.o`.

- *Ligação:*
  `$ gcc -o teste teste.c`
  Gera o arquivo executável `teste`.

Assim, para gerar um executável para um simples programa (`teste.c`, por exemplo), utiliza-se o seguinte comando:

```
$ gcc -o teste teste.c
```

No entanto, a maioria dos programas consistem em vários arquivos de código. Caso hajam dois arquivos, `arquivo1.c` e `arquivo2.c`, a seguinte linha de comando poderia ser utilizada para compilar o programa final:

```
$ gcc -o teste arquivo1.c arquivo2.c
```

O GCC interpreta os arquivos de acordo com a sua extensão. Algumas das principais podem ser vistas na Tabela 2.1.

Tabela 2.1: Algumas extensões reconhecidas pelo GCC

Extensão	Interpretação
.c	Programa em linguagem C
.C .cc	Programa em linguagem C++
.i	Programa em C pré-processado
.ii	Programa em C++ pré-processado
.S .s	Programa em linguagem Assembly
.o	Programa objeto
.a .so	Bibliotecas compiladas

Alguns parâmetros importantes suportados pelo GCC foram vistos, como o -c e -o. No entanto, outras opções podem ser importantes para compilar um projeto. Algumas destas estão listadas na Tabela 2.2.

2.2 Arquivos *headers*

Arquivos *headers* contém basicamente definições utilizadas por bibliotecas do sistema e protótipos de funções, de forma que um programa sendo compilado possa realizar a verificação de sintaxe e tipos corretamente.

A localização dos arquivos headers em uma distribuição GNU/Linux convencional fica no diretório `/usr/include`. Alguns outros diretórios importantes são:

- `/usr/X11R6/include/X11`: contém headers necessários para compilar programas que usam o *X11 Window System*.
- `/usr/include/X11`: headers para compilar programas que usam o X11. Normalmente é um symlink para o diretório `/usr/X11R6/include/X11`;

Tabela 2.2: Algumas extensões reconhecidas pelo GCC

Opção	**Descrição**
-o FILE	Especifica o nome do arquivo compilado (padrão `a.out`).
-c	Gera somente o objeto (.o).
-I <diretório>	Busca arquivos de headers primeiro em *diretório*.
-L <diretório>	Busca bibliotecas primeiro em *diretório*.
-lnome	Liga a biblioteca *libnome* ao executável.
-static	Executa a ligação usando bibliotecas estáticas.
-g, -ggdb	Inclui informações para o depurador.
-O	Otimiza o código.
-On	Especifica um nível de otimização entre 0 e 3 (0 não otimiza).
-ansi	Desligando recursos do GCC incompatíveis com o padrão ANSI.
-w	Inibe as mensagens de warning (aviso).
-Wall	Emite todas as mensagens de warning.
-Werror	Converte todas as mensagens de warning em erro.
-v	Mostra os comandos usados em cada passo da compilação.

- `/usr/include/g++`: headers para compilar programas C++ que utilizam o compilador GNU C++;
- `/usr/include/asm`: Arquivos contendo as funções, parâmetros e definições específicas arquitetura na qual os binários gerados serão criados;
- `/usr/include/linux`: link simbólico para o diretório contendo os headers utilizados na versão do Linux usada para compilar a GNU libc. Algumas distribuições ainda praticam o erro de apontar para o diretório do kernel Linux *atual* sendo utilizado no host.

Estes diretórios são úteis para localizar a definição de uma função que não apresenta uma man page ou uma info page. Um comando que auxilia a localização de definições é o `grep(1)`.

2.3 Utilizando os recursos de debug

Estes recursos são muito úteis, pois freqüentemente erros acontecem. A opção -g do GCC permite que sejam gerados códigos de informação para o depurador. Para compilar um programa com suporte ao uso em um depurador, é necessário utilizar a seguinte linha de comando:

```
$ gcc -g -o teste teste.c
```

Para retirar os símbolos gerados pela opção -g, pode-se utilizar o comando `strip(1)` com a opção –strip-debug sobre o arquivo binário (neste caso chamado de *teste*):

```
$ strip --strip-debug teste
```

O depurador padrão da GNU é o GDB [GIL 2003], o *GNU Debugger*. Para utilizar o depurador, chama-se o executável a partir da shell, passando o nome do programa a ser depurado como argumento. Em alguns casos, quando o programa está sendo executado e realiza uma operação ilegal, sua execução é abortada e um arquivo *core* é gerado, contendo informações sobre o contexto atual do processo quando o erro ocorreu. Neste caso, é possível passar o arquivo *core* como parâmetro extra, de forma que o depurador *salte* imediatamente para a instrução que causou o erro. Vale ressaltar que esta operação só gera algum dado relevante quando o programa que gerou esta excessão tiver sido compilado com suporte a símbolos de depuração.

O GDB pode fazer quatro tipos de operações (além de outras operações de suporte a estas) para ajudar a encontrar problemas no código:

- Iniciar o programa, especificando algum parâmetro que possa afetar seu comportamento;
- Parar o programa em uma condição específica;
- Examinar o que houve, quando o programa tiver encerrado a sua execução;
- Modificar variáveis no programa, de forma a poder corrigir os efeitos causados por um problema e avançar para poder aprender sobre um novo.

A utilização padrão do GDB então é feita da seguinte maneira, tomando como exemplo um programa chamado *teste*:

```
$ gdb teste

GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

O prompt de comando agora passa a ser *(gdb)*, onde comandos específicos do GDB poderão ser utilizados para entender o que está acontecendo no código do programa. O seguinte programa será utilizado como exemplo para as explicações de operação do GDB:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int
5  calcula (int a, int b)
6  {
7      int resultado;
8      resultado = a + b;
9      return resultado;
10 }
11
12 int
13 main()
14 {
15     int a, b, resultado;
16     a = 1;
17     b = 2;
18     resultado = calcula (a, b);
19     printf ("Resultado: %d\n", resultado );
20     exit (0);
21 }
```

Para executar este programa no GDB, basta executar a operação *run*:

```
$ gdb teste
  ...
(gdb) run
```

```
Starting program: /Users/lucasvr/teste
Resultado: 3

Program exited normally.
```

Como não houve nenhum problema durante a execução deste programa, ele encerrou e retornou ao prompt do GDB. No entanto, podemos querer executar o programa passo a passo. Para isto, invoca-se o comando `break <linha>`. Para listar o código sendo depurado e descobrir a linha que deseja-se utilizar como ponto de parada (*breakpoint*), os comandos "`list`" e "`list -`" podem ser usados para listar as próximas 10 linhas de código, ou as últimas 10 linhas de código, de forma progressiva. Outra forma bastante utilizada é marcar o breakpoint em determinadas funções. No nosso exemplo, para iniciar a execução passo a passo a partir da função *calcula*, ao invés de especificar a linha desejada para o comando break, pode ser informada o nome da função a ser utilizada como breakpoint:

```
(gdb) break calcula
Breakpoint 1 at 0x8048446: file teste.c, line 7.
(gdb) run
Starting program: /Users/lucasvr/teste

Breakpoint 1, calcula (a=1, b=2) at teste.c:7
7               resultado = a + b;
```

A partir deste ponto, podemos continuar a execução do programa até que ele encerre normalmente, com o uso do comando `continue`, ou executá-lo passo a passo. Para isto, podemos utilizar o comando `step`, que continua executando o programa até encontrar uma outra linha do código, retornando então o controle ao GDB. Uma característica importante do `step [count]` é que ele não entra em funções ou loops: caso haja uma chamada a uma função, por exemplo, ele irá retornar o controle ao GDB apenas após a execução completa dessa função. Caso deseja-se ter um controle maior sobre a execução de funções, loops e switches, pode-se utilizar o comando `next [count]`, que segue o fluxo do programa.

Outros comandos interessantes são os referentes à variáveis. O GDB permite que o usuário modifique o valor de uma variável com o comando `set variable <variável=valor>`. No caso do breakpoint realizado anteriormente na função *calcula*, poderíamos alterar o valor de *a* para 2, alterando o resultado da soma. Para isto, basta executar:

```
(gdb) break calcula
Breakpoint 1 at 0x8048446: file teste.c, line 7.
(gdb) run
Starting program: /Users/lucasvr/teste

Breakpoint 1, calcula (a=1, b=2) at teste.c:7
7               resultado = a + b;
(gdb) set variable a=2
(gdb) continue
Continuing.
Resultado: 4

Program exited normally.
(gdb)
```

Outro recurso bastante utilizado é o watchpoint, que interrompe a execução do programa quando uma determinada expressão for realizada. Ele pode ser usado para verificar quando uma variável tem seu valor modificado, como no exemplo abaixo, onde uma mudança no valor de *b* é a condição para o retorno do comando ao depurador:

```
(gdb) break main
Breakpoint 1 at 0x804846a: file teste.c, line 14.
(gdb) run
Starting program: /Users/lucasvr/teste

Breakpoint 1, main () at teste.c:14
14              a = 1;
(gdb) watch b
Hardware watchpoint 2: b
(gdb) continue
Continuing.
Hardware watchpoint 2: b

Old value = 0
New value = 2
main () at teste.c:16
16              resultado = calcula (a, b);
(gdb) print b
$1 = 2
```

Nota-se que o controle é retomado logo após a execução da operação que modificou *b*, logo a execução continua tendo como indicação a próxima linha a ser executada pelo programa. Utilizamos ainda o comando `print`, que é usado para imprimir o valor de alguma variável – neste caso a variável *b*.

O GDB permite ainda ser anexado a um processo em execução. Para isto, o GDB deve ser executado passando o PID do processo como segundo argumento, seguindo o nome do programa. Como o programa a ser depurado estará em um ponto desconhecido de execução, é interessante verificar *onde* ele encontra-se. O comando `backtrack` do GDB imprime os *stack frames* do processo, e assim é possível saber onde o programa está sendo executado. Cada frame é um dado associado à uma chamada de função, e contém os argumentos enviados à ela, as variáveis locais a esta função e o endereço no qual a função está executando.

Para fazer um teste, modificaremos nosso programa exemplo para que ele faça a leitura de uma tecla para continuar a execução. Assim, poderemos depurá-lo enquanto estiver em execução, já que ele executa muito rápido para podermos anexar o GDB a ele. A função *calcula* deverá ficar da seguinte forma:

```
int
calcula (int a, int b)
{
    int resultado;
    resultado = a + b;
    getchar ();
    return resultado;
}
```

Após recompilar o programa – não esquecendo da opção -g –, iremos executá-lo:

```
$ ./teste
```

Agora, em outro terminal, iremos anexar o GDB à sua instância. Para isto, é necessário obter o PID do processo, executando o GDB em seguida:

```
$ ps -C teste
  PID TTY          TIME CMD
15247 pts/6    00:00:00 teste
```

```
$ gdb teste 15247 -silent
Attaching to program: /Users/lucasvr/teste, process 15247
Reading symbols from /System/Links/Libraries/libc.so.6...done.
Loaded symbols for /System/Links/Libraries/libc.so.6
Reading symbols from /System/Links/Libraries/ld-linux.so.2...done.
Loaded symbols for /System/Links/Libraries/ld-linux.so.2
0x400e76d4 in read () from /System/Links/Libraries/libc.so.6
(gdb)
```

Neste exemplo, o processo *teste* encontrava-se no PID 15247, e este valor foi informado como parâmetro para a execução do GDB. Nesse ponto temos controle sobre o processo, mas não sabemos ainda onde ele está. Para descobrir, vamos listar os stack frames do processo:

```
(gdb) backtrace
#0  0x400e76d4 in read () from /System/Links/Libraries/libc.so.6
#1  0x40146238 in sys_sigabbrev () from /System/Links/Libraries/libc.so.6
#2  0x40085b63 in _IO_file_underflow () from /System/Links/Libraries/libc.so.6
#3  0x40087ffd in _IO_default_uflow () from /System/Links/Libraries/libc.so.6
#4  0x40087e46 in __uflow () from /System/Links/Libraries/libc.so.6
#5  0x4008230a in getchar () from /System/Links/Libraries/libc.so.6
#6  0x08048487 in calcula (a=1, b=2) at teste.c:8
#7  0x080484b8 in main () at teste.c:17
#8  0x4002cfe4 in __libc_start_main () from /System/Links/Libraries/libc.so.6
```

Esta stack deve ser interpretada a partir do maior valor, que indica a base de execução do programa – neste caso a função *__libc_start_main()*, da biblioteca padrão do C, a GNU libc. Em seguida a função *main()* foi executada na linha 17 do programa teste.c, chamando a função *calcula()* a seguir, na linha 8, e finalmente realizando a chamada *getchar()*. A partir deste ponto, as funções referentes à implementação desta chamada na GNU libc foram executadas, e pode-se verificar que o processo encontra-se aguardando a leitura de algum dado com a chamada de sistema *read()*. Nota-se que para cada função chamada, um novo frame é criado, e a cada retorno de função, o frame para esta função é eliminado. Caso a função seja recursiva, podem existir vários frames para a mesma função.

Para escolher algum frame específico, pode-se utilizar o comando `frame`, seguido do número do frame desejado. Pode-se ainda utilizar os comandos `up [n]` e `down [n]`, que avançam ou retrocedem a escolha do frame desejado na stack. Para modificarmos nosso programa agora, iremos modificar o valor do resultado a ser retornado para a função *main()*. Para isso, iremos escolher o frame referente à função *calcula()* (frame 6, como mostrou o backtrace), modificar o valor da variável *resultado* e continuar a execução do programa:

```
(gdb) frame 6
#6  0x08048487 in calcula (a=1, b=2) at teste.c:8
8               getchar ();
(gdb) set variable resultado=10
(gdb) continue
Continuing.
```

Agora, ao retornar ao terminal utilizado para disparar o processo `teste` e pressionar a tecla ENTER, veremos que nossa modificação no GDB foi refletida no processo que foi modificado. A partir deste ponto pode-se retornar ao GDB e encerrá-lo, com o comando `quit`.

```
(gdb) quit

$
```

2.4 Autoconf e Automake

O Autoconf e Automake são duas ferramentas que auxiliam o programador a gerar arquivos *Makefile* para o projeto. O Autoconf é responsável por realizar uma série de checagens no host, como a versão do compilador, tamanhos de tipos de variáveis, existência de headers, bibliotecas (bem como suas versões), entre muitos outros. Estas verificações são importantes, pois muitas vezes os programas rodam nas plataformas mais diversas, em arquiteturas diferentes da utilizada pelo programador. Após a realização destas verificações, o Automake é utilizado para gerar um arquivo Makefile a partir das exigências feitas pelo Autoconf.

O processo para gerar arquivos Makefile utilizando estas ferramentas consiste nos seguintes passos:

- Escrever um arquivo *Makefile.am*;
- Escrever um arquivo *configure.in*;
- Rodar `aclocal` para criar cópias locais de macros utilizadas pelo `autoconf`;
- Rodar `automake -a -c` para gerar um arquivo *Makefile.in* a partir do arquivo *Makefile.am* (o *automake* verifica o arquivo *configure.in* para obter informações sobre o projeto);
- Rodar `autoconf` para gerar o script `configure`.

A estrutura utilizada em projetos GNU segue a seguinte hierarquia de arquivos e diretórios, a partir do diretório base dos fontes do projeto:

- `AUTHORS`: arquivo contendo os nomes dos autores do projeto;
- `BUGS`: listagem dos problemas conhecidos do projeto, e de formas de submeter bugs encontrados pelo usuário;
- `COPYING`: contém a licença pela qual o projeto é distribuído;
- `ChangeLog`: contém um histórico com modificações realizadas no projeto;
- `FAQ`: perguntas freqüentemente feitas a respeito de alguma característica do programa;
- `INSTALL`: processo de instalação do programa;
- `Makefile.am`: o arquivo a ser gerado pelo programador, informando alguns dados importantes para que o arquivo Makefile seja construído;
- `NEWS`: contém novidades sobre o programa;
- `README`: informações sobre o programa, como seu propósito e formas de contato com o autor e/ou mailing lists;
- `configure.in`: arquivo a ser gerado pelo programador, contendo verificações que devem ser feitas antes de compilar o programa;
- `include/`: diretório contendo os arquivos headers do programa;
- `src/`: diretório contendo os arquivos fontes do programa;
- `src/Makefile.am`: regras de compilação dos fontes, como nome do binário e arquivos fontes usados para construí-lo.

2.4.1 Escrevendo um arquivo *Makefile.am*

Como visto, existem dois arquivos Makefile.am. O que reside no diretório raiz do projeto informa os subdiretórios que contém os fontes e algumas opções extras. O conteúdo dele normalmente é:

```
AUTOMAKE_OPTIONS = foreign gnu
SUBDIRS = src
EXTRA_DIST = BUGS COPYING ChangeLog INSTALL README FAQ
```

O campo *AUTOMAKE_OPTIONS* aceita alguns níveis, dentre os quais:

- *foreign:* verifica se o projeto está de acordo com os padrões exigidos para distribuir o software;
- *gnu:* verifica se o projeto inclui os arquivos exigidos pelo GNU standards para pacotes de software;
- *no-dependencies:* usado em situações onde não existem informações suficientes para fazer a detecção automática de dependências no pacote.

As opções suportadas são muitas, e aconselha-se a leitura da *info page* do `automake` para consultá-las.

O campo *SUBDIRS* informa diretórios onde residem arquivos fontes do programa, e *EXTRA_DIST* informa que os arquivos ali listados devem ser ignorados pelo `automake` e pelo `autoconf`.

O arquivo *Makefile.am* interno ao diretório *src/* pode conter informações complexas com regras que devem ser utilizadas para cada arquivo, mas no entanto o seguinte é suficiente para gerar pacotes com estas ferramentas:

```
bin_PROGRAMS = Programa
Programa_SOURCES = arquivo1.c arquivo2.c arquivo3.c

EXTRA_DIST = config.sample
```

Após gerar estes arquivos, a próxima etapa é a criação de um arquivo *configure.in*.

2.4.2 Escrevendo um arquivo *configure.in*

A maneira mais fácil de gerar um arquivo *configure.in* é através do comando `autoscan(1)`. Ele examina os arquivos fonte a partir do diretório corrente, recursivamente, e gera um arquivo *configure.scan* base no qual pode ser gerado o arquivo *configure.in*. Após renomeado para *configure.in*, algumas mudanças devem ser feitas, como informar o nome e versão do projeto, e-mail para reportar bugs, entre outros.

Para uma estrutura contendo apenas um arquivo chamado *teste.c* no diretório *src/* com um *printf* e dois includes, *stdin.h* e *unistd.h*, este é o resultado gerado pelo autoscan:

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.57)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/teste.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.
```

```
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

Após preparados estes dois arquivos, *configure.in* e *Makefile.am*, o projeto pode ser gerado executando `aclocal, automake` e `autoconf`, como explicado anteriormente.

2.5 DDD - *Data Display Debugger*

Apesar do GDB ser um poderoso debugger, sendo até possível com ele debuggar o próprio kernel do sistema operacional, assim com aplicativos *multi-threaded*, a sua interface em linha de comando não é amigável para usuários pouco familirizados com a ferramenta, o que frustra a maioria dos usuários iniciantes em GNU/Linux e não permite que toda a potencialidade do debugger seja aproveitada em sua totalidade.

Para suprir tal dificuldade, foi desenvolvido o DDD [ZEL 2000], que é simplesmente uma interface gráfica para o debugger.

Porém, o DDD possui ainda extensões ao processo de debugging tradicional, permitindo que gráficos sejam plotados a partir do valor de variáveis, como matrizes e arrays. Um exemplo de operação do DDD pode ser visto na Figura 2.5

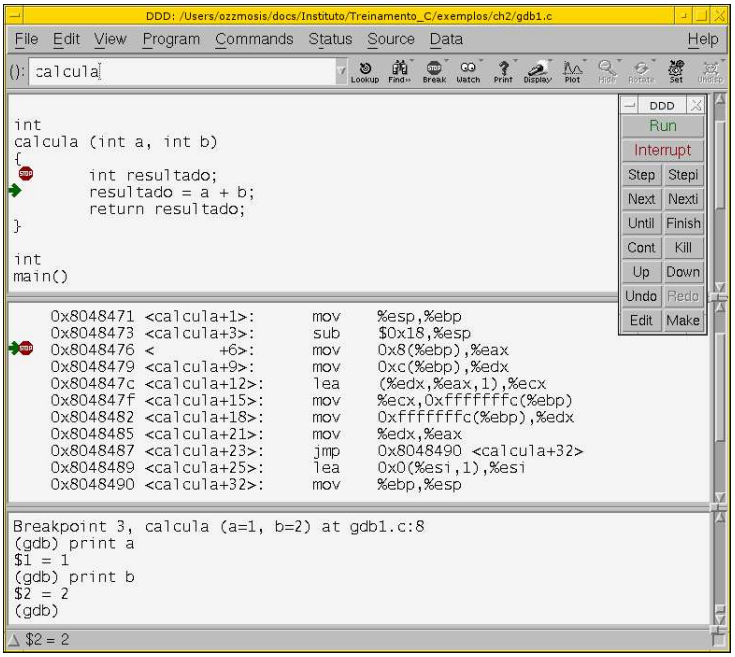

Figura 2.1: Operando com o DDD sobre o programa exemplo

Vale notar que mesmo com todas as extensões do DDD ao GDB, o mesmo ainda continua sendo um mero *front-end* para o debugger da GNU, sendo possível executar **todos** os comandos do GDB na janela no final da janela principal do DDD, como pode ser visto na janela inferior da Figura 2.5

2.6 Criando aplicativos que rodem tanto no GNU/Linux como no Windows

Não existe nenhuma regra específica a ser seguida para executar o mesmo código em sistemas operacionais diferentes, apenas alguns cuidados que devem ser atendidos. O mais comum é apenas utilizar funções ANSI, que estão disponíveis em ambos, e evitar ao máximo o uso de funções específicas de um determinado sistema operacional. Assim, pode-se usar a flag *-ansi* do GCC para restringir as funções usadas às definidas e suportadas pelo ANSI.

Em alguns casos, entretanto, isto não é possível, e então pode ser necessário utilizar diretivas de compilação `#ifdef` para verificar o sistema em uso. É bastante comum isolar códigos especificos de uma determinada arquitetura ou sistema operacional desta forma. Por exemplo:

```
#if defined ( __linux__ )
# define DIR_SEPARATOR '\\'
#else if defined ( _WIN32)
# define DIR_SEPARATOR '/'
#endif
```

Existe ainda uma forma de portar código nativo criado no GNU/Linux para o Windows, usando um ambiente no Windows chamado Cygwin[1]. Ele consiste de uma camada de bibliotecas que atuam como *wrappers*, de forma a mapear nomes de dispositivos existentes apenas no GNU/Linux para o Windows, como dispositivos IDE, SCSI e seriais, entre outros.

Uma grande variedade de bibliotecas e interfaces e aplicativos já encontram suporte neste ambiente, como bibliotecas de threads, a biblioteca GTK+, o servidor Apache e o servidor XFree86.

[1] ver `http://www.cygwin.com`

Capítulo 3

Explorando os recursos básicos do sistema operacional

3.1 Lendo parâmetros curtos e longos passados pela linha de comando

Em "C" é possível a um programa coletar parâmetros passados na linha de comando no momento de sua ativação. Assim, podemos escrever um programa chamado `soma` que receberá os parâmetros *1* e *2* que serão passados via linha de comando:

```
$ soma 1 2
```

Estes parâmetros são passados para o programa através dos argumentos da função `main()`. O módulo principal do programa (`main`) pode receber dois parâmetros externos opcionais, que são o *argv*, um vetor de ponteiros para strings, e o *argc*, que indica a quantidade de strings contidas neste array. Caso um programa não receba nenhum parâmetro, existirá apenas uma string no array (no índice 0), contendo o nome do programa executado, e *argc* terá o valor 1.

Tomando por exemplo o programa *soma*, poderíamos escrevê-lo da seguinte forma:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int
5  main (int argc, char **argv)
6  {
7      int i;
8      printf ("Numero de argumentos: %d\n", argc);
9      for (i = 0; i < argc; i++)
10         printf ("Argumento %d: %s\n", i, argv[i]);
11     exit (0);
12 }
```

Este programa irá imprimir a quantidade de argumentos recebidos, e em seguida irá mostrar cada argumento recebido como parâmetro. É normal que os programas realizem a consistência dos parâmetros recebidos. Dessa forma, é comum encontrar o seguinte tipo de código:

```
1  #include <stdio.h>
2
3  int
```

```
4  main (int argc , char **argv)
5  {
6      if ( argc != 3) {
7          fprintf ( stderr , "Sintaxe: %s valor1 valor2\n", argv [0]);
8          exit (1);
9      }
10 }
```

O padrão POSIX.2 provê uma função para tratar argumentos de entrada, o `getopt(3)`, definida pelo seguinte protótipo:

```
#include <unistd.h>
int getopt (int argc , char * const argv [], const char * optstring );
extern char *optarg;
extern int optind , opterr , optopt;
```

A função `getopt(3)` analisa os argumentos da linha de comando. Seus argumentos *argc* e *argv* são, respectivamente, a contagem de argumentos e o array de strings passados para a função `main()` na invocação do programa.

Um elemento de *argv* que inicia com "-" (e não é exatamente "-" ou "–") é um elemento de opção. Os caracteres deste elemento (fora o "-" inicial) são caracteres de opção. Se `getopt(3)` é chamado repetidamente, ele retorna sucessivamente cada um dos caracteres de opção de cada um dos elementos de opção.

Se a função `getopt(3)` encontra um outro caractere de opção, ela retorna este caractere, atualizando a variável externa *optind* e uma variável estática *nextchar* de forma que a próxima chamada a `getopt(3)` pode continuar a busca com o caractere de opção seguinte ou um elemento de *argv*.

Se não há mais caracteres de opção, `getopt(3)` retorna -1. Então *optind* é o índice em *argv* do primeiro elemento de *argv* que não é uma opção.

optstring é uma string contendo os caracteres de opção legítimos. Se tal caractere é seguido por dois-pontos, a opção requer um argumento, então `getopt` atribui um ponteiro para o texto que segue no mesmo elemento de *argv*, ou para o texto do elemento seguinte de *argv*, em *optarg*. Dois dois-pontos significam que uma opção recebe um argumento opcional; se há texto no elemento de *argv* atual, ele é retornado em *optarg*, caso contrário *optarg* é zerado. Esta é uma extensão GNU. Se *optstring* contém *W* seguido de um ponto-e-vírgula, então *-W foo* é tratado como a opção longa *–foo*. (A opção *-W* é reservada pelo POSIX.2 para extensões de implementação.) Este comportamento é uma extensão GNU, não disponível em bibliotecas anteriores à GNU libc 2.

A função `getopt_long(3)` funciona como `getopt(3)`, exceto que ele também aceita opções longas, iniciadas por dois traços. Nomes de opções longas podem ser abreviados se a abreviação for única ou se iguala exatamente à alguma opção definida. Uma opção longa pode requerir um parâmetro da forma *–arg=param* ou *–arg param*. O seguinte protótipo a define:

```
#define _GNU_SOURCE
#include <getopt.h>
int getopt_long (int argc , char * const argv [], const char * optstring ,
        const struct option *longopts , int *longindex );
```

Neste caso, *longopts* é um ponteiro para o primeiro elemento de um array de *struct option*, declarado em <getopt.h> como:

```
struct option {
    const char *name;
    int has_arg;
    int * flag;
    int val;
};
```

O significado destes campos são:

- *nome:* é o nome da opção longa;
- *has_arg:* no_argument (ou 0) se a opção não recebe um argumento, required_argument (ou 1) se a opção requer um argumento, ou optional_argument (ou 2) se a opção recebe um argumento opcional;
- *flag:* especifica como os resultados são retornados para uma opção longa. Se *flag* é NULL, então `getopt_long(3)` retorna *val*. Caso contrário, `getopt_long(3)` retorna 0, e *flag* aponta para uma variável, a qual é ajustada para *val* se a opção é encontrada, ou deixada intocada se a opção não é encontrada;
- *val:* é o valor a retornar, ou a carregar para a variável apontada por flag.

O último elemento do array deve ser preenchido com zeros.

O seguinte exemplo ilustra o uso de `getopt_long(3)` com a maioria de seus recursos:

```
#include <stdio.h>
#include <unistd.h>
#define _GNU_SOURCE
#include <getopt.h>

static struct option long_options [] = {
    {"add",      1, 0, 0},
    {"append",   0, 0, 0},
    {"delete",   1, 0, 0},
    {"verbose" , 0, 0, 0},
    {"create",   1, 0, 'c'},
    {"file",     1, 0, 0},
     {0, 0, 0, 0}
};

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

    while (1) {
        int option_index = 0;

        c = getopt_long (argc, argv, "abc:d:",
                         long_options, &option_index);
        if (c == -1)
            break;

        switch (c) {
            case 0:
                printf ("opcao %s", long_options [option_index].name);
                if (optarg)
                    printf (" com argumento %s", optarg);
                printf ("\n");
                break;
            case 'a':
                printf ("opcao a\n");
                break;
            case 'b':
```

```
40                      printf ("opcao b\n");
41                      break;
42                  case 'c':
43                      printf ("opcao c com valor '%s'\n", optarg);
44                      break;
45                  case 'd':
46                      printf ("opcao d com valor '%s'\n", optarg);
47                      break;
48                  case '?':
49                      break;
50                  default:
51                      printf ("erro: getopt retornou %d\n", c);
52              }
53          }
54
55          if (optind < argc) {
56              printf ("parametros nao referentes a opcoes: ");
57              while (optind < argc)
58                  printf ("\t%s\n", argv[optind++]);
59          }
60          exit (0);
61      }
```

3.2 Trabalhando com arquivos

Arquivos em C podem ser manipulados com o uso de descritores de arquivos, representados por objetos do tipo *int*, ou streams, representados por objetos do tipo *FILE **.

Descritores de arquivos provêem uma interface primitiva e de baixo-nível para operações de entrada e saída. Tanto descritores de arquivos como streams podem representar uma conexão a um dispositivo (como um terminal), ou um pipe ou socket para comunicar com outro processo, u ainda para representar um arquivo *normal*. Os descritores de arquivos são usados quando é necessário realizar operações de controle sobre algum dispositivo específico, ou ainda quando precisam realizar I/O em modos especiais, como não-bloqueante, dado que streams não oferecem recursos de baixo nível.

A maior vantagem de utilizar a interface de streams é que o conjunto de funções para realizar operações de I/O, ao oposto de operações de controle, é muito mais rico e que as facilidades correspondentes para os descritores de arquivos. A interface de descritores de arquivos provê apenas operações simples para transferência de blocos de caracteres, enquanto a interface de streams provê funções poderosas para formatação de funções de I/O, bem como funções orientadas a caracter e linhas.

É importante ressaltar que streams são mais portáveis que descritores de arquivos, pois nem todas as operações sobre descritores de arquivos estão implementadas em outros sistemas operacionais – sobretudo o conjunto de operações GNU, apesar de que grande parte delas encontram-se no padrão POSIX 1.1

3.3 Streams

Quando a função `main` do programa é invocada, ela já disponibiliza três streams abertas e prontas para o uso. Elas representam os canais *standard* de entrada e saída que foram estabelecidos para o processo, e estão declaradas no arquivo header *stdio.h*:

- *FILE *stdin:* é a stream de entrada padrão, que é a origem normal de entrada de dados em um programa;

- *FILE *stout:* é a stream padrão de saída, que é usada para imprimir os dados de saída do programa;
- *FILE *stderr:* da mesma forma que a stream *stdout*, porém para imprimir mensagens de erro e de diagnóstico pelo programa.

Algumas das operações mais comuns de serem realizadas sobre streams são:

- `FILE *fopen (const char *path, const char *mode);`
  abre o arquivo apontado pela string "path" e retorna uma stream associada a ele. Os seguintes modos são utilizados para abrir o arquivo:
  - *r:* abre um arquivo texto para leitura, posicionando a stream no início do arquivo.
  - *r+:* abre um arquivo texto para leitura e escrita, posicionando a stream no início do arquivo.
  - *w:* trunca o arquivo para zero bytes e cria o arquivo texto para escrita. A stream é posicionada no início do arquivo.
  - *w+:* abre o arquivo para leitura e escrita. O arquivo é criado caso ele não exista, ou truncado caso contrário. A stream é posicionada no início do arquivo.
  - *a:* abre o arquivo para *append* (escrita no final do arquivo). Ele é criado caso ele não exista, e a stream é posicionada no final do arquivo.
  - *a+:* abre o arquivo para leitura e *append*. Ele é criado caso ele não exista, e a stream é posicionada no final do arquivo.

  A string *modo* pode ainda incluir a letra "b" após indicar o modo desejado para explicitar que o arquivo a ser trabalhado será binário, e não texto. Apesar de não ter efeito em um programa que irá operar no GNU/Linux, ela é muito importante caso o programa irá rodar em algum ambiente não-UNIX.
- `int fclose (FILE *stream);`
  desassocia a stream "stream" do arquivo ao qual ela estava operando. Caso a stream estivesse sendo usada para dados de saída, os dados em buffer são escritos antes, com o uso da operação `fflush(3)`.
- `int fflush (FILE *stream);`
  força a escrita de todos os dados em espaço de usuário que estejam mantidos em buffer para a saída informada ou atualizam a stream "stream" através da função *write* associada a ela.
- `int feof (FILE *stream);`
  testa o indicador de fim-de-arquivo para a stream apontada por "stream", retornando um valor não-zero caso este indicador esteja setado, ou 0 caso ainda não seja fim de arquivo.
- `int fileno (FILE *stream);`
  examina o argumento "stream" e retorna um inteiro relativo ao seu descritor de arquivo.
- `int fputs (const char *s, FILE *stream);`
  escreve a string *s* para *stream*, sem o caracter "
  0" de final de string.
- `char *fgets (char *s, int size, FILE *stream);`
  lê *size* caracteres a partir de *stream* e os armazena no buffer apontado por *s*, que já deve estar pré-alocado. A leitura encerra após encontrar EOF ou uma nova linha. Se um caracter de nova linha for lida, ele é armazenada no buffer. Um caracter "
  0" é armazenado após o último caracter do buffer.
- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
  lê *nmemb* elementos de dados, cada um tendo o tamanho de *size* bytes, a partir da stream apontada por *stream*, e armazenando-os no local apontado por *ptr*. Esta operação costuma ser utilizada para operações em arquivos binários. `fread` retorna o número de itens lidos com sucesso.

- `size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);`
  escreve *nmemb* elementos de dados, cada um tendo o tamanho de *size* bytes, para a stream apontada por *stream*, obtidos a partir do local apontado por *ptr*. Esta operação costuma ser utilizada para operações em arquivos binários. `fwrite` retorna o número de itens escritos com sucesso.
- `int fseek (FILE *stream, long offset, int whence);`
  posiciona o indicador de posição de arquivo para a stream apontada por *stream*. A nova posição, medida em bytes, é obtida através da soma de *offset* bytes para a posição especificada por *whence*. Caso *whence* seja definido como SEEK_SET, SEEK_CUR ou SEEK_END, o offset é relativo ao início do arquivo, à posição corrente do indicador ou ao final do arquivo, respectivamente.
- `long ftell (FILE *stream);`
  obtém o valor atual do indicador de posição de arquivo para a stream apontada por *stream*.
- `void rewind (FILE *stream);`
  aponta o indicador de posição de arquivo para a stream apontada por *stream* para o início do arquivo. Ela é equivalente à executar a função `fseek (stream, 0L, SEEK_SET)`.

O programa a seguir ilustra o uso destas operações: ele abre um arquivo para leitura e um outro para escrita, escrevendo apenas as linhas pares para o segundo.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int
5  copia_arquivo (char *origem, char *destino)
6  {
7      FILE *stream_origem, * stream_destino;
8      char buffer[1024];
9      int i;
10
11     /* abre arquivos de origem e destino, criando este ultimo */
12     stream_origem = fopen (origem, "r");
13     if (! stream_origem) {
14         perror ("fopen");
15         return -1;
16     }
17
18     stream_destino = fopen ( destino , "w");
19     if (! stream_destino ) {
20         perror ("fopen");
21         fclose (stream_origem);
22         return -1;
23     }
24
25     /* processa linhas do arquivo origem */
26     i = 1;
27     while (! feof (stream_origem)) {
28         fgets ( buffer , sizeof ( buffer ), stream_origem);
29
30         if ( i % 2 == 0) /* numero par */
31             fputs ( buffer , stream_destino );
```

```
32
33          i++;
34      }
35
36      /* fecha as streams e retorna */
37      fclose (stream_origem);
38      fclose ( stream_destino );
39      return 0;
40  }
41
42  int
43  main (int argc , char **argv)
44  {
45      int res;
46
47      if ( argc != 3) {
48          printf ("Sintaxe: %s <arq_origem> <arq_destino>\n", argv[0]);
49          exit (1);
50      }
51
52      res = copia_arquivo ( argv [1], argv [2]);
53      if ( res < 0) {
54          fprintf ( stderr , "Um erro ocorreu durante a copia\n");
55          exit (1);
56      }
57      exit (0);
58  }
```

3.4 Descritores de Arquivos

Assim como quando operando sobre *streams*, quando utilizando descritores de arquivos temos os três canais de entrada e saída abertos automaticamente (stdin, stdout e stderr, ou 0, 1 e 2, respectivamente).

Um conceito importante é o de que em sistemas operacionais UNIX, tudo é um arquivo. Ou seja, dispositivos de rede, de terminais, de som, vídeo, diretórios e os próprios arquivos de dados do usuário são representados por arquivos. Desta forma é possível manipular todos os dispositivos do sistema com o uso de operações sobre arquivos: por isto as operações sobre descritores de arquivos são tão importantes.

As operações básicas sobre descritores de arquivos são:

- `int open(const char *pathname, int flags);`
  abre o arquivo especificado por *pathname*, retornando um descritor de arquivo representado por um inteiro. As flags devem ser O_RDONLY, O_WRONLY ou O_RDWR, que requisitam a abertura do arquivo no modo somente leitura, somente escrita, ou leitura e escrita, respectivamente. Estes modos podem ser combinados via *bitwise OR* com zero ou mais opções, dentre as quais estão:
  - *O_CREAT:* se o arquivo não existe, ele será criado. Neste caso, o protótipo da função `open` listado abaixo será utilizado.
  - *O_EXCL:* quando usado em conjunto com O_CREAT e o arquivo já existir, a função retornará um erro.
  - *O_TRUNC:* caso o arquivo esteja sendo aberto em modo de escrita em conjunto com este modo, o arquivo será truncado para 0 bytes.

  - *O_APPEND:* abre o arquivo no modo *append*, para adicionar novos dados ao final do arquivo.
  - *O_NOFOLLOW:* caso *pathname* seja um link simbólico, a chamada é interrompida e retorna um erro.
  - *O_DIRECTORY:* caso *pathname* não seja um diretório, a chamada retorna um erro.

- `int open(const char *pathname, int flags, mode_t mode);`
  caso o modo O_CREAT seja passado como flag para a função `open`, este protótipo deverá ser utilizado, para explicitar o modo no qual o arquivo será aberto. Maiores informações sobre os parâmetros podem ser consultados na página manual do `open(2)`.
- `int creat(const char *pathname, mode_t mode);`
  esta função é equivalente à chamada `open` com as flags iguais a O_CREAT | O_WRONLY | O_TRUNC.
- `int close(int fd);`
  fecha o descritor de arquivo indicado por *fd*.

Algumas outras operações costumam ser realizadas sobre descritores de arquivos. Podemos destacar:

- `ssize_t read(int fd, void *buf, size_t count);`
  lê até *count* bytes a partir da posição corrente do arquivo representado pelo descritor *fd* para o buffer apontado por *buf*. O valor retornado informa quantos bytes foram lidos com sucesso, ou -1 em caso de erro.
- `ssize_t write(int fd, const void *buf, size_t count);`
  escreve até *count* bytes no arquivo representado pelo descritor *fd* a partir do buffer apontado por *buf*. O valor retornado informa quantos bytes foram escritos com sucesso, ou -1 em caso de erro.
- `off_t lseek(int fildes, off_t offset, int whence);`
  opera da mesma forma que o `fseek(3)`, porém sobre descritores de arquivos. *fildes* refere-se ao descritor de arquivo que será utilizado na operação de *seek*, *offset* informa quantos bytes serão movimentados na operação e *whence* informa a posição relativa à qual será feito o *seek*.
- `int fstat(int filedes, struct stat *buf);`
  retorna informações sobre o arquivo representado pelo descritor *filedes*, armazenando-as no endereço da struct stat apontada por *buf*. Entre as informações retornadas, estão a quantidade de hard links feitas para este arquivo, o seu *inode number*, a data de última alteração e de último acesso, entre outros.

O exemplo abaixo lida com um arquivo binário através de descritores de arquivos. Neste exemplo, ele adiciona um novo campo do mesmo tipo de estrutura da qual ele é composto após seu último byte de dados, e em seguida lê e imprime o seu conteúdo.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>       /* atoi (3) */
4  #include <sys/types.h>    /* open(2) */
5  #include <sys/ stat .h>   /* open(2) */
6  #include <sys/ fcntl .h>  /* open(2) */
7
8  struct informacao {
9      int     quantidade;
10     float   valor;
11     char    nome[128];
12 };
13
```

```
14 int
15  cria_arquivo  ( char *nome)
16 {
17       int  fd;
18
19       fd = open (nome, O_RDWR | O_CREAT | O_TRUNC, S_IRWXU | S_IRWXG | S_IRWXO);
20       if  ( fd < 0) {
21            perror  ("open");
22            exit  (1);
23       }
24       return  fd;
25 }
26
27 void
28  le_arquivo  ( int  fd)
29 {
30        size_t  n;
31       struct  informacao  info;
32
33       lseek  ( fd , 0,  SEEK_SET);
34       while  (1) {
35            n = read  ( fd, &info ,  sizeof  ( info ));
36            if  ( n == 0) {
37                 /* fim de arquivo */
38                 break;
39            }
40
41            printf  ("nome: %s\nvalor: %f\nquantidade: %d\n\n",
42                      info.nome, info.valor ,  info.quantidade );
43       }
44 }
45
46 void
47  adiciona_entrada  ( int  fd ,  char *numero, char *arquivo)
48 {
49        ssize_t  n;
50       int  num;
51       struct  informacao  info;
52
53       num = atoi  ( numero);
54
55       info.quantidade  = 10 *  num;
56       info.valor  = 123.45 * (  float  ) num;
57        snprintf  ( info.nome,  sizeof  ( info.nome), "arquivo %s",  arquivo );
58
59       lseek  ( fd , 0,  SEEK_END);
60       n = write  ( fd, &info ,  sizeof  ( info ));
61       if  ( n < 0)
62            perror  ("write");
63 }
64
65 int
66 main (int  argc ,  char **argv)
67 {
```

```
68      int fd;
69
70      if (argc != 3) {
71          fprintf (stderr, "Sintaxe: %s <arquivo destino> <algum numero>\n", argv[0]);
72          exit (1);
73      }
74
75      fd = open (argv[1], O_RDWR);
76      if (fd < 0) {
77          perror ("open");
78          printf ("tentando criar arquivo ...\ n\n");
79          fd = cria_arquivo (argv[1]);
80      }
81
82      adiciona_entrada (fd, argv[2], argv[1]);
83      le_arquivo (fd);
84
85      close (fd);
86      exit (0);
87  }
```

Capítulo 4

CGI: Common Gateway Interface

O CGI define uma maneira de interação entre o web browser e programas externos de geração de conteúdo, normalmente referidos como programas CGI ou scripts CGI.

Existem duas principais diferenças entre criar um programa convencional e um CGI. Primeiramente, toda a saída do programa CGI deve ser precedida por um header do tipo MIME. Isto é um cabeçalho HTTP que informa ao client o tipo de conteúdo que ele está recebendo. Na maior parte do tempo, o conteúdo será:

```
Content-type:  text/html
```

Em segundo lugar, toda a saída precisa estar formatada em HTML ou em algum outro formato que o browser será capaz de mostrar. Na maior parte do tempo isto é HTML, mas ocasionalmente poderá ser necessário escrever algum programa que gere a saída de uma imagem GIF ou algum outro conteúdo não-HTML.

Fora estes dois detalhes, um programa em CGI parece-se com um programa convencional qualquer. O exemplo abaixo lista o conteúdo de um arquivo texto:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  #define ARQUIVO "/tmp/teste.txt"
5
6  int
7  main (int argc, char **argv)
8  {
9      char buffer[256];
10     FILE *fd;
11
12
13     printf ("Content-type: text/html\r\r\n\n");
14     printf ("<html><head><title> Conteudo do arquivo %s </title></head>\n",
15             ARQUIVO);
16
17     fd = fopen (ARQUIVO, "r");
18     if (!fd) {
19         printf ("<b> ERRO </b>: arquivo %s ãno existe\n", ARQUIVO);
20         printf ("</body></html>\n");
21         exit (1);
```

```
22      }
23
24
25      while (! feof (fd)) {
26          fgets (buffer, sizeof (buffer), fd);
27          printf ("%s <br>\n", buffer);
28      }
29
30      fclose (fd);
31      printf ("</body></html>\n");
32      exit (0);
33  }
```

Após a compilação do programa, basta copiá-lo para o diretório `cgi-bin` do Apache, ou do `httpd` em uso, e acessá-lo via browser, como no exemplo abaixo:

```
http://localhost/cgi-bin/teste
```

No entanto, é bastante comum o uso de alguma biblioteca para facilitar a formatação dos dados em HTML, para evitar erros e agilizar o desenvolvimento do programa CGI. Existem várias bibliotecas disponíveis cadastradas no site da Freshmeat[1].

[1] ver `http://www.freshmeat.net`

Capítulo 5

Bibliotecas dinâmicas

O objetivo de bibliotecas dinâmicas é a centralização de implementação de funções em bibliotecas específicas, permitindo que o gerenciamento e atualização das mesmas se torne propagável a todos os programas que usam suas funções de forma trivial.

No GNU/Linux, existe o conceito de *shared objects*, que é análogo ao conceito de DLL do Microsoft Windows.

Para utilizar bibliotecas dinâmicas no GNU/Linux, se usam as funções `dlopen(3)` e `dlclose(3)`. Para carregar os símbolos das funções a serem utilizadas a função `dlsym(3)` deve ser utilizada.

Um exemplo do uso de bibliotecas dinâmicas pode ser visto a seguir:

```
1  #include <stdio.h>
2  #include <dlfcn.h>
3
4  int
5  main (int argc, char **argv)
6  {
7      void * biblioteca;
8      double (*seno)(double);
9      char *erro;
10
11     biblioteca = dlopen ("libm.so", RTLD_LAZY);
12     if (! biblioteca) {
13         fprintf (stderr, "%s\n", dlerror ());
14         exit (1);
15     }
16
17     seno = dlsym (biblioteca, "simbolo_nao_existe");
18     if ((erro = dlerror ()) != NULL) {
19         fprintf (stderr, "%s\n", erro);
20         exit (1);
21     }
22
23     printf ("%f\n", (*seno)(2.0));
24     dlclose (biblioteca);
25     exit (0);
26 }
```

Existem basicamente 2 modos para resolução dos símbolos disponibilizados por uma biblioteca: `RTLD_LAZY`, que significa que os ponteiros para funções não resolvidos em tempo de

compilação devem ser apenas resolvidos pela biblioteca dinâmica em tempo de execução, ou `RTLD_NOW`, que significa que todos os símbolos da biblioteca dinâmica devem ser resolvidos na abertura da mesma, sendo que a abertura da biblioteca deverá falhar caso algum símbolo não possa ser definido.

```
1  #include <stdio.h>
2  #include <dlfcn.h>
3
4  int
5  main (int argc, char **argv)
6  {
7      void * biblioteca;
8      double (*seno)(double);
9      char *erro;
10
11     biblioteca = dlopen ("libm.so", RTLD_NOW);
12     if (! biblioteca) {
13         fprintf (stderr, "%s\n", dlerror ());
14         exit (1);
15     }
16
17     seno = dlsym (biblioteca, "sin");
18     if ((erro = dlerror ()) != NULL) {
19         fprintf (stderr, "%s\n", erro);
20         exit (1);
21     }
22
23     printf ("%f\n", (*seno)(2.0));
24     dlclose (biblioteca);
25     exit (0);
26 }
```

Para usar as funções de bibliotecas dinâmicas a biblioteca `dl` deve ser especificada em tempo linkagem, tal como:

```
$ gcc dl-now.c -o dl-now -ldl
```

5.1 Criação de bibliotecas dinâmicas

Para criar bibliotecas dinâmicas no GNU/Linux o processo é simples: basta compilar o código com a flag `-shared`, gerando a biblioteca.

Um exemplo de uma biblioteca simples que exporta apenas uma função pode ser visto a seguir:

```
1  #include <stdio.h>
2
3  int
4  mundo (void)
5  {
6      printf ("Bem-vindo ao GNU/Linux\n");
7  }
```

Para compilar a biblioteca, basta:

```
$ gcc minha_biblioteca -o libminha_biblioteca.so -shared
```

Capítulo 6

Processos e threads

Existem duas maneiras de criar novos fluxos de processos no GNU/Linux. Uma delas é através da chamada de sistema `fork(2)`, e a outra através da criação de threads, que podem ser vistas como processos leves [STE 98]

6.1 Processos

É possível criar processos através da chamada de sistema `fork(2)`. A chamada de sistema não recebe nada como parâmetro, retorna os seguintes valores:

- **0:** Sinalizando que o processo que está executando no momento é o do filho criado;
- **>0:** Sinalizando que o processo que está executando no momento é o do pai;
- **<0:** Sinaliza que a chamada de sistema não pôde ser completada com sucesso.

Um exemplo de criação de um sub-processo pode ser visto no exemplo abaixo:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  int
9  main (int argc, char **argv)
10 {
11     int status;
12     pid_t pid;
13
14     pid = fork ();
15     if (pid == 0) {
16         char **args;
17
18         printf ("Processo filho executando \"ls -a\"\n");
19         args = (char **) malloc (sizeof (char *) * 3);
20         args[0] = strdup ("ls");
21         args[1] = strdup ("-a");
22         args[2] = NULL;
23
```

```
24          execvp ( args [0], args );
25          free ( args [0]);
26          free ( args [1]);
27          free ( args );
28
29      } else if ( pid > 0) {
30          printf ("Processo pai aguardando processo filho ..\n");
31          waitpid ( pid, & status , WUNTRACED);
32
33          if ( WIFEXITED(status))
34              printf ("processo filho executou normalmente\n");
35          else
36              printf ("processo filho nao executou normalmente\n");
37
38      } else if ( pid < 0) {
39          /* algum erro ocorreu */
40          perror ("fork");
41      }
42
43      exit (0);
44  }
```

A função `execvp(3)` é utilizada para trocar a imagem do processo corrente pela imagem de um novo processo. O primeiro argumento para ela é o *pathname* para um programa, e o segundo argumento é um array terminado por NULL, descrevendo os argumentos que este programa irá receber. O conjunto do uso do `fork(3)` com o `execvp(3)` e `waitpid(3)` apresenta uma semântica semelhante à função `spawnl()` do Microsoft Windows, com a excessão de que a `spawnl()` já explicita em seus argumentos se o processo pai deve aguardar pelo seu término de execução ou não.

Outra função utilizada na listagem do programa acima é a `waitpid(3)`, que aguarda pelo término da execução de algum processo. Os parâmetros enviados a ela são o *pid* do processo a ser aguardado, um ponteiro para um inteiro, onde serão armazenas informações sobre a execução do processo (como sinais que foram entregues a ele para que ele fosse encerrado e valor de retorno da execução do processo), e uma constante chamada `WUNTRACED`, que significa que ela deve parar de aguardar pelo filho que já teve sua execução parada e se encontra em um status desconhecido.

6.2 Threads

Threads têm o mesmo propósito de um processo: gerar um novo fluxo de execução. No entanto, elas são mais adequadas para uso em programas que precisam disparar muitos processos filhos, devido ao fato de elas implementarem o conceito de *processos leves*.

Todas as threads em um processo compartilham o estado deste processo. Elas residem no mesmo endereçamento de memória, vêem as mesmas funções e os mesmos dados. As maiores vantagens de utilizar threads em programas são:

- Ganhos de performance em hardware multiprocessado;
- Aumento da responsividade do programa;
- Mudança de métodos de comunicação entre os processos;
- Uso eficiente dos recursos do sistema;
- Apenas um binário executa bem tanto em arquiteturas SMP (*Symmetric Multi Processors*) quanto UP (*Uni Processor*);
- Possibilidade de criar programas bem estruturados;

- Apenas um único arquivo fonte pode ser usado em múltiplas plataformas.

Uma thread consiste de uma *stack* e um *stack pointer*, um *program counter* e algumas informações a respeito de sua instância, como prioridades de escalonamento e máscaras de sinal, armazenadas na sua estrutura. Além disso, os registradores da CPU são também armazenados (sendo que a stack pointer e o program counter são alguns destes registradores).

O Linux utiliza, até a série 2.4, o modelo de threads POSIX, aprovado pelo padrão IEEE 1003.1b-1993 [Ame 94]. Para dar suporte a este modelo, as distribuições GNU/Linux fornecem a biblioteca *LinuxThreads*, que implementa grande parte das exigências feitas por este padrão. A série 2.6 do kernel Linux está fazendo uso de um outro modelo de threads, chamado *NPTL* (*Native POSIX Threading Library*) [The 2003]. Este texto baseia-se na implementação das LinuxThreads.

Em relação às threads Win32, as threads POSIX são muito mais leves, contendo primitivas mais simples de uso. Uma outra característica é a de que threads Win32 mantém uma dependência entre janelas e threads. Como nem todas threads constroem e usam janelas, isto acarreta em um overhead desnecessário para os fluxos de execução.

Para criar um novo fluxo de execução com o uso de threads no GNU/Linux, utiliza-se a função `pthread_create(3)`, definida pelo seguinte protótipo:

```
#include <pthread.h>
int pthread_create (pthread_t * thread, pthread_attr_t * attr,
                    void * (* start_routine )(void *), void * arg);
```

`pthread_create(3)` cria uma nova thread de controle que executa concorrentemente com a thread que a gerou. A nova thread aplica a função *start_routine*, passando *arg* como seu argumento. A nova thread termina explicitamente, chamando a função `pthread_exit(3)`, ou implicitamente, apenas retornando da função *start_routine*. O último caso é equivalente a chamar `pthread_exit(3)` com o resultado retornado pela função *start_routine* como retorno.

O argumento *attr* especifica algum atributo a ser aplicado para a nova thread, ou pode ser NULL caso os atributos padrão devam ser utilizados: a thread criada é *joinable*, ou seja, seus recursos de memória não serão liberados até que algum processo sincronize com esta thread, e tem uma política de escalonamento normal (não-realtime). Os atributos a serem passados para a thread devem ser inicializados com a função `pthread_attr_init(3)`, e após configurados com o uso de alguma das funções listadas na página manual `pthread_attr_init(3)`.

Para sincronizar um processo (ou uma thread) com o resultado de uma outra thread, utiliza-se a função `pthread_join(3)`:

```
#include <pthread.h>
int pthread_join (pthread_t th, void ** thread_return);
```

`pthread_join(3)` suspende a execução da thread corrente até que a thread identificada por *th* termine, tanto explicitamente, via `pthread_exit(3)`, ou sendo cancelada, através da função `pthread_cancel(3)`. Caso o argumento *thread_return* seja não nulo, ele é utilizado para armazenar o valor de retorno da thread *th*, devendo ser desalocado após ter sido utilizado pelo programa.

O exemplo abaixo ilustra a criação e sincronização de uma thread no GNU/Linux. Para compilar este programa, é necessário passar a flag `-lpthread`, para que o programa seja linkado com a biblioteca de threads `libpthread.so` e possa fazer uso de suas funções.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <pthread.h>
5
6 void *
7 calcula (void *dados)
8 {
```

```
9       int * ret = NULL;
10      int * valor = ( int *) dados;
11
12      ret = ( int *) malloc ( sizeof ( int ));
13      *ret = * valor * * valor ;
14
15      pthread_exit ( ret );
16  }
17
18  int
19  main ( int argc , char **argv)
20  {
21      pthread_t tid ;
22      int *dados , * resultado ;
23
24      dados = ( int *) malloc ( sizeof ( int ));
25      *dados = 5;
26
27      pthread_create (& tid , NULL, calcula , dados );
28      pthread_join ( tid , ( void **) & resultado );
29
30      printf ("%d * %d = %d\n", *dados, *dados, * resultado );
31
32      free (dados );
33      free ( resultado );
34      exit (0);
35  }
```

Como threads podem compartilhar recursos entre si, não é necessário fazer uso de mecanismos pesados de intercomunicação entre processos. A forma mais comum de compartilhar dados entre diferentes threads é através de variáveis globais a elas, e mantendo a coerência das leituras e escritas destes dados com o uso de mecanismos de exclusão mútua (*mutexes*) e semáforos.

O uso de mutexes é feito basicamente através das primitivas `pthread_mutex_lock(3)` e `pthread_mutex_unlock(3)`. Na primeira primitiva, caso alguma outra thread já tenha adquirido o lock, a thread que tentou pegá-lo pela segunda vez tem sua execução interrompida até que a outra thread o libere. Caso não desejáva-se que a execução seja interrompida, `pthread_mutex_trylock(3)` pode ser usado: caso o lock não esteja disponível ela continua a execução do programa.

O exemplo abaixo mostra como compartilhar um dado entre duas threads de execução, utilizando mecanismos de exclusão mútua:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <pthread.h>
5
6   static int valor = 0;
7   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9   void *
10  incrementa ( void *dados)
11  {
12      int i;
13
14      for ( i = 0; i < 10000; i++) {
```

```
15          pthread_mutex_lock (&mutex);
16          valor++;
17          pthread_mutex_unlock (&mutex);
18      }
19      pthread_exit (NULL);
20  }
21
22  void *
23  decrementa (void *dados)
24  {
25      int i;
26
27      for (i = 0; i < 10000; i++) {
28          pthread_mutex_lock (&mutex);
29          valor--;
30          pthread_mutex_unlock (&mutex);
31      }
32      pthread_exit (NULL);
33  }
34
35
36  int
37  main (int argc, char **argv)
38  {
39      pthread_t tid, tid2;
40
41      pthread_create (&tid, NULL, incrementa, NULL);
42      pthread_create (&tid2, NULL, decrementa, NULL);
43      pthread_join ( tid, NULL);
44      pthread_join ( tid2, NULL);
45
46      printf ("valor da variavel global: %d\n", valor);
47      exit (0);
48  }
```

Semáforos também tem um uso bastante comuns. Eles funcionam como contadores para recursos compartilhados entre threads. As operações básicas realizadas em semáforos são incrementar o contador atomicamente e aguardar até que o contador seja não nulo e decrementá-lo atomicamente. Nestes casos, podem ser usadas as funções `sem_init(3)`, `sem_wait(3)`, `sem_post(3)` e `sem_destroy(3)`:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_destroy(sem_t * sem);
```

`sem_init(3)` inicializa o semáforo apontado por *sem* com o valor especificado em *value*. Este valor indica quantas threads poderão entrar na sessão crítica ao mesmo tempo. Nos casos mais comuns são utilizados semáforos binários, e então este valor é iniciado com 1. O argumento *pshared* indica se o semáforo é local ao processo atual (neste caso *pshared* deve ser zero) ou se ele não é compartilhado entre vários processos (neste caso *pshared* deve ser não zero). No entanto, a implementação das LinuxThreads não implementa o compartilhamento de semáforos entre vários processos, retornando um erro na inicialização do semáforo caso *pshared* seja um valor diferente de zero.

O exemplo abaixo apresenta o uso de semáforos para controle de uma sessão crítica:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  #include <semaphore.h>
6
7  static int valor = 0;
8  sem_t semaforo;
9
10 void *
11 incrementa (void *dados)
12 {
13     int i;
14
15     for (i = 0; i < 10000; i++) {
16         sem_wait (&semaforo);
17         valor++;
18         sem_post (&semaforo);
19     }
20     pthread_exit (NULL);
21 }
22
23 void *
24 decrementa (void *dados)
25 {
26     int i;
27
28     for (i = 0; i < 10000; i++) {
29         sem_wait (&semaforo);
30         valor--;
31         sem_post (&semaforo);
32     }
33     pthread_exit (NULL);
34 }
35
36
37 int
38 main (int argc, char **argv)
39 {
40     pthread_t tid, tid2;
41
42     sem_init (&semaforo, 0, 1);
43
44     pthread_create (&tid, NULL, incrementa, NULL);
45     pthread_create (&tid2, NULL, decrementa, NULL);
46     pthread_join (tid, NULL);
47     pthread_join (tid2, NULL);
48
49     printf ("valor da variavel global: %d\n", valor);
50     sem_destroy (&semaforo);
51     exit (0);
52 }
```

Capítulo 7

TCP/IP

O conjunto de funções disponíveis para serem utilizadas na comunicação através do protocolo TCP/IP é muito grande. Este capítulo visa apresentar a utilização de sockets no GNU/Linux para comunicação entre cliente e servidor, nos modos bloqueante e não-bloqueante no IPv4 [STE 98a].

Provavelmente a maneira mais simples de explicar como programar sockets neste ambiente é com o uso de um exemplo comentado. Abaixo encontra-se listada uma implementação de um cliente TCP de uma aplicação do tipo *timeofday*. Este cliente estabelece uma conexão TCP com um servidor, e o servidor apenas envia de volta a hora e data atuais, em um formato formatado e legível.

```
1  #include "timeofday.h"
2
3  int
4  main (int argc, char **argv)
5  {
6      int sockfd, n;
7      char recvline[MAXLINE + 1];
8      struct sockaddr_in servaddr;
9
10     if (argc != 2) {
11         fprintf (stderr, "Sintaxe: %s <endereco IP>\n", argv[0]);
12         exit (1);
13     }
14
15     if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
16         err_quit ("socket");
17
18     memset (&servaddr, 0, sizeof (servaddr));
19     servaddr.sin_family = AF_INET;
20     servaddr.sin_port = htons (13);  /* servidor de daytime */
21
22     if (inet_pton (AF_INET, argv[1], &servaddr.sin_addr) <= 0)
23         err_quit ("inet_pton");
24
25     if (connect (sockfd, (struct sockaddr *) &servaddr, sizeof (servaddr)) < 0)
26         err_quit ("connect");
27
28     while ((n = read (sockfd, recvline, MAXLINE)) > 0) {
29         recvline[n] = 0;                /* terminada por NULL */
```

```
30          if ( fputs ( recvline , stdout ) == EOF)
31              err_quit ("fputs");
32      }
33
34      if ( n < 0)
35          err_quit ("read");
36
37      exit (0);
38  }
```

Este exemplo, quando compilado, deve utilizar a flag `-lnsl`, para ser linkado com as funções de sockets da Glibc.

Existem vários detalhes a serem considerados neste programa:

- 1: o cabeçalho em comum com a aplicação do servidor. Os detalhes de seu conteúdo são mostrados mais a seguir;
- 15: criação de um socket TCP. A função *socket* cria uma stream (SOCK_STREAM) de Internet (AF_INET), que é o nome dado para um socket TCP. Esta função retorna o descritor de um socket, utilizado para realizar as chamadas de funções futuras, como as chamadas ao *connect* e ao *read*;
- $18 - 23$: especificação do endereço IP de um servidor e uma porta. Uma estrutura do tipo Internet socket address (a *sockaddr_in sockaddr*) é preenchida com o endereço IP do servidor e com o número da porta. A estrutura é zerada com o uso da função *memset*, configurada para usar a família de endereços AF_INET e para usar a porta 13 (que é a porta conhecida para o servidor de daytime em qualquer host TCP/IP que suporte este serviço). O IP fornecido e a porta nesta estrutura precisam estar em formatos específicos: para isto, a função *htons* (*host to network short*) para converter o número da porta, e é utilizada a função *inet_pton* (*presentation to numeric*) para converter o endereço IP em ASCII para o formato necessário;
- $25 - 26$: estabelecimento da conexão com o servidor. A função *connect*, quando aplicada a um socket TCP, estabelece uma conexão TCP com o servidor especificado pela estrutura *socket address* apontada pelo segundo argumento;
- $28 - 35$: lê e mostra o reply do servidor. Como o TCP é um protocolo *byte-stream*, ele não tem registro sobre limites do conteúdo sendo transportado, ou seja: os bytes retornados podem vir de uma única vez, ou em n pequenas leituras, até que os últimos bytes sejam lidos. Desta forma, nem sempre é garantido que com um único *read* venha toda a resposta do servidor. Assim, *sempre* que estivermos lendo o conteúdo de um socket TCP, é necessário fazê-lo em um loop, terminando-o quando *read* retornar 0 (o outro lado encerrou a conexão) ou um valor menor do que 0 (quando ocorre um erro).

Como o programa anterior não pode executar corretamente sem um servidor, é necessário escrever um. A versão do servidor para este cliente encontra-se listado logo abaixo.

```
1   #include "timeofday.h"
2
3   int
4   main (int argc , char **argv)
5   {
6       int listenfd , connfd;
7       struct sockaddr_in servaddr;
8       char buff[MAXLINE];
9       time_t ticks;
10
11      if (( listenfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
12          err_quit ("socket");
```

```
13
14      memset (&servaddr , 0,  sizeof  ( servaddr ));
15      servaddr . sin_family  = AF_INET;
16      servaddr . sin_addr . s_addr  = htonl  ( INADDR_ANY);
17      servaddr . sin_port  = htons  (13);        /* servidor de daytime */
18
19      if  (( bind  (  listenfd  , ( struct  sockaddr *) & servaddr ,  sizeof  ( servaddr ))) < 0)
20           err_quit  ("bind");
21
22      if  ((  listen  (  listenfd  , MAX_CONNECT)) < 0)
23           err_quit  (" listen ");
24
25      for  (;;) {
26           if  (( connfd = accept  (  listenfd  , ( struct  sockaddr  *)  NULL, NULL)) < 0)
27                err_quit  ("accept");
28
29           ticks  = time  ( NULL);
30           snprintf  ( buff ,  sizeof  ( buff ),  "%.24s\r\n", ctime (& ticks ));
31           if  (( write  ( connfd , buff ,  strlen  ( buff ))) < 0)
32                err_quit  ("write");
33
34           close  ( connfd );
35      }
36 }
```

Assim como no cliente, alguns detalhes precisam ser considerados na versão do servidor:

- $11 - 12$: cria o socket TCP, idêntico à versão do cliente;
- $19 - 20$: realiza um *binding* (amarra) da porta conhecida com o socket. A porta conhecida do servidor (13, para o serviço de daytime) é ligada ao socket, preenchendo uma estrutura de Internet socket address e chamando o *bind*. É especificado o endereço IP como INADDR_ANY, que permite que o servidor aceite uma conexão em qualquer interface, caso o host tenha mais de uma;
- $22 - 23$: converte o socket para um *listening socket*. Quando é feita a chamada ao *listen*, conexões ao socket passam a ser aceitas pelo kernel. Estes três passos, *socket, bind* e *listen* são os passos utilizados para preparar qualquer *listening descriptor* (descritor de escuta) em um servidor TCP (que neste caso é o *listenfd*). A constante MAX_CONNECT está especificada no header *timeofday.h*, e especifica a quantidade máxima de clientes que irão conectar-se ao *listening descriptor*.
- $26 - 32$: aceita conexões dos clientes e envia um reply. Normalmente o processo do servidor é posto para dormir na chamada à *accept*, aguardando que uma conexão de um cliente chegue e seja aceita. Uma conexão TCP usa o que é chamado de um *three-way handshake* para estabelecer uma conexão, e quando este *handshake* é completado, a função *accept* retorna, e o valor de retorno desta função é um novo descritor (*connfd*), chamado de *connected descriptor* (descritor do cliente conectado). O novo descritor é usado para comunicar com o novo cliente. Um novo descritor é retornado pelo *accept* a cada cliente que conecta ao servidor.

  A função *time* retorna a hora e data atual, na forma dos segundos que ocorreram desde a época do Unix: 1^o de janeiro de 1970, às 00:00:00 horas UTC (*Coordinated Universal Time*). A função *ctime* converte o valor inteiro retornado em uma forma legível para humanos. Esta string é então escrita de volta para o cliente.
- 34: a conexão com o cliente é encerrada.

O arquivo header utilizado em ambos exemplos encontra-se abaixo:

```
1  #ifndef _timeofday_h
2  #define _timeofday_h 1
3
4  #include <stdio.h>
5  #include <unistd.h>
6  #include <stdlib.h>
7  #include <sys/types.h>
8  #include <sys/socket.h>  /* socket () */
9  #include <arpa/inet.h>   /* inet_pton () */
10 #include <netinet/in.h>  /* htons () */
11 #include <string.h>      /* memset() */
12 #include <time.h>        /* time () */
13
14 #define err_quit (msg) ({ perror(msg); exit (1); })
15
16 /* SOMAXCONN fica definido em bits/socket.h */
17 #define MAX_CONNECT SOMAXCONN
18 #define MAXLINE 1024
19
20 #endif /* _timeofday_h */
```

Após ter o servidor compilado e rodando em um terminal, um outro terminal pode ser utilizado para efetuar a comunicação entre o cliente e o servidor. Para isto, o cliente deve ser executado passando um endereço IP válido como parâmetro. Como estamos especificando qualquer interface para escuta no servidor, pode ser utilizada a interface *loopback*:

```
$ timeofday-client 127.0.0.1
Mon Oct 13 00:50:12 2003
```

Estes exemplos funcionam e podem servir como base para programas maiores. No entanto, eles ainda usam valores numéricos para referenciar o servidor, e muitas vezes é desejável utilizar o nome do host que disponibiliza o serviço. Para isto algumas funções podem ser utilizadas, como:

- *struct hostent *gethostbyname(const char *name):* retorna um ponteiro para uma estrutura do tipo *hostent* para o host *name*, que pode estar representado tanto na notação de um endereço IP numérico quanto como um hostname. A estrutura *hostent*, definida no <netdb.h>. é composta pelos seguintes elementos:

```
struct hostent {
    char    *h_name;        /* nome oficial do host */
    char    **h_aliases ;   /* lista de aliases , terminada por NULL */
    int     h_addrtype ;    /* tipo do endereco do host */
    int     h_length ;      /* tamanho do endereco */
    char    **h_addr_list ; /* lista de anderecos , terminada por NULL */
}
#define h_addr  h_addr_list [0]  /* backward compatibility */
```

- *struct hostent *gethostbyaddr(const char *addr, int len, int type):* retorna um ponteiro para uma estrutura do tipo *hostent* para o endereço do host especificado em *addr* de tamanho *len* e de tipo *type*. O único tipo válido de endereço é atualmente *AF_INET*.
- *struct servent *getservbyname(const char *name, const char *proto):* retorna um ponteiro para uma estrutura do tipo *servent* para a linha do arquivo `services` do diretório de configurações da distribuição, normalmente `/etc/services`, que coincida com o serviço *name* que utiliza o protocolo *proto*. Caso *proto* seja NULL, qualquer protocolo será aceito.

A estrutura *servent* é especificada no arquivo <netdb.h> como segue:

```
struct servent {
    char    *s_name;        /* nome oficial do servico */
    char    **s_aliases ;   /* lista de aliases , terminada por NULL */
    int     s_port ;        /* numero da porta */
    char    *s_proto ;      /* protocolo a ser usado */
}
```

- *struct servent *getservbyport(int port, const char *proto):* retorna um ponteiro para uma estrutura do tipo *servent* para a linha do arquivo `services` que coincida com a porta especificada em *port*, usando o protocolo *proto*. Caso o protocolo seja NULL, qualquer um será aceito no *matching*. A porta deve estar informada em *network byte order* (`htonl(3)`).

O próximo exemplo mostra como estes recursos podem ser incorporados ao programa anterior:

```
1  #include "timeofday.h"
2
3  int
4  main (int argc , char **argv)
5  {
6      int sockfd , n;
7      char recvline [MAXLINE + 1];
8      struct sockaddr_in servaddr ;
9      struct hostent *ht;
10
11     if ( argc != 3) {
12         fprintf ( stderr , "Sintaxe: %s <endereco IP> <porta>\n", argv[0]);
13         exit (1);
14     }
15
16     if (( sockfd = socket ( AF_INET, SOCK_STREAM, 0)) < 0)
17         err_quit ("socket");
18
19     if (( ht = gethostbyname (argv [1])) < 0)
20         err_quit ("gethostbyname");
21
22     memset (&servaddr , 0, sizeof ( servaddr ));
23     servaddr . sin_family = AF_INET;
24     servaddr . sin_port = htons ( atoi ( argv [2]));
25     memcpy (&servaddr.sin_addr , ht->h_addr_list [0], sizeof ( struct in_addr ));
26
27     if ( connect ( sockfd , ( struct sockaddr *) & servaddr , sizeof ( servaddr )) < 0)
28         err_quit ("connect");
29
30     while (( n = read ( sockfd , recvline , MAXLINE)) > 0) {
31         recvline [n ] = 0;              /* terminada por NULL */
32         if ( fputs ( recvline , stdout ) == EOF)
33             err_quit ("fputs");
34     }
35
36     if ( n < 0)
37         err_quit ("read");
38
```

```
39      exit (0);
40  }
```

Podemos ver na linha 19 que agora é feita uma chamada para `gethostbyname(3)`, que realiza o lookup do nome fornecido como parâmetro na linha de comando. Na linha 24, informamos a porta através do parâmetro passado pelo usuário (a função `atoi(3)` realiza a conversão de uma string para um inteiro). A forma de realizar a cópia da interface a ser utilizada também foi mudada: na linha 25, vemos que a partir de agora o endereço é copiado a partir das informações disponíveis na estrutura *hostent− >h_addr_list*. Utilizamos o primeiro endereço referente às interfaces neste host, e conectamos a ele. O restante do programa mantém-se inalterado.

A seguir vamos analisar as mudanças na versão do servidor:

```
1  #include "timeofday.h"
2
3  int
4  main (int argc, char **argv)
5  {
6       int listenfd, connfd;
7       struct sockaddr_in servaddr, cliaddr;
8       socklen_t len;
9       char buff[MAXLINE];
10      time_t ticks;
11
12      if (argc != 2) {
13           printf ("Sintaxe: %s <porta>\n", argv[0]);
14           exit (1);
15      }
16
17      if ((listenfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
18           err_quit ("socket");
19
20      memset (&servaddr, 0, sizeof (servaddr));
21      servaddr.sin_family = AF_INET;
22      servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
23      servaddr.sin_port = htons (atoi (argv[1]));
24
25      if ((bind (listenfd, (struct sockaddr *) &servaddr, sizeof (servaddr))) < 0)
26           err_quit ("bind");
27
28      if ((listen (listenfd, MAX_CONNECT)) < 0)
29           err_quit ("listen");
30
31      for (;;) {
32           if ((connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &len)) < 0)
33                err_quit ("accept");
34
35           printf ("conexao estabelecida com %s na porta %d\n",
36                inet_ntop (AF_INET, &cliaddr.sin_addr, buff, sizeof (buff)),
37                ntohs (cliaddr.sin_port));
38
39           ticks = time (NULL);
40           snprintf (buff, sizeof (buff), "%.24s\r\n", ctime (&ticks));
41           if ((write (connfd, buff, strlen (buff))) < 0)
42                err_quit ("write");
```

```
43
44          close (connfd);
45      }
46  }
```

As modificações foram semelhantes às realizadas no cliente. Na linha 23, é feita a conversão da string relativa à porta a ser usada para inteiro, passando o resultado ao campo *sin_port* da estrutura *servaddr* para que esta porta seja usada. A função `accept` agora também informa um ponteiro para a estrutura *sockaddr cliaddr*. Isto permite que o `accept` armazene dados sobre a conexão do cliente. Estes dados são então usados nas linhas $35-37$: o `inet_ntop` converte o endereço passado em *&cliaddr.sin_addr* que usa a família *AF_INET* para uma string, que é copiada para *buffer*. Um ponteiro para este buffer é retornado, e usamos este ponteiro para imprimir o hostname do client. A função `htohs` é usada apenas para converter o número em *network byte order* para *host byte order*, e então usa seu resultado para imprimí-lo.

O header usado foi levemente modificado, declarando o cabeçalho *<netdb.h>* para prover suporte à chamada `gethostbyname(3)`, além de já incluir headers para suporte à chamada `select(2)`:

```
21  #ifndef _timeofday_h
22  #define _timeofday_h 1
23
24  #include <stdio.h>
25  #include <unistd.h>
26  #include <stdlib.h>
27  #include <sys/types.h>
28  #include <sys/socket.h> /* socket () */
29  #include <arpa/inet.h>  /* inet_pton () */
30  #include <netinet/in.h> /* htons () */
31  #include <string.h>     /* memset() */
32  #include <netdb.h>      /* gethostbyname () */
33  #include <time.h>       /* time () */
34
35  #include <sys/select.h> /* select () */
36  #include <sys/stat.h>
37  #include <fcntl.h>
38  #include <errno.h>
39
40  #define err_quit (msg) ({ perror(msg); exit (1); })
41
42  #define MAXLINE 1024
43  #define MAX_CONNECT 100
44
45  #endif /* _timeofday_h */
```

7.1 Select

Uma conexão não-bloqueante envolve várias operações sobre os descritores. Antes de apresentá-las, é necessário introduzir a função `select(2)`. Essa função aguarda até que um número de descritores de arquivo mudem seu status. Seu protótipo é definido como segue:

```
/* de acordo com o POSIX 1003.1-2001 */
include <sys/select.h>

/* de acordo com padroes mais novos */
```

```
include <sys/time.h>
include <sys/types.h>
include <unistd.h>

int  select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
             struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

A função `select(2)` usa um timeout que é uma *struct timeval* (com segundos e microsegundos), que pode ser atualizado para indicar quanto tempo já se passou do timeout especifidado. Três conjuntos independentes de descritores são monitorados. Os que estão listados em *readfds* serão monitorados para verificar se algum caracter fica disponível para leitura (para verificar se a leitura não vai bloquear). Os listados em *writefds* serão monitorados para verificar se alguma escrita não irá bloquear, e os em *exceptfds* serão monitorados quanto a exceções. Na saída, os conjuntos são modificados para indicar os descritores que tiveram seu status alterado.

Para operar esses conjuntos, quatro macros são disponibilizadas. *FD_ZERO* zera um conjunto. *FD_SET* e *FD_CLEAR* adicionam ou removem um descritor de um conjunto, e *FD_ISSET* testa para ver se um descritor faz parte do conjunto. Isso é útil após o retorno de `select(2)`, para garantir que o descritor que foi modificado é o mesmo que tinha-se interesse em monitorar.

O valor *n* indica o descritor de maior número em qualquer um dos três conjuntos, mais 1. *timeout* é o tempo que deve ser aguardado até que `select(2)` retorne. Caso *timeout* seja NULL, a chamada pode bloquear indefinidamente.

A seguinte estruturas é usada para definir os timeouts:

```
#include <sys/time.h>
struct timeval {
    long tv_sec;     /* segundos */
    long tv_usec;    /* micro segundos */
}
```

O valor de retorno de uma chamada ao `select(2)` é o número de descritores contigo no conjunto de descritores, que pode ser zero caso o timeout tenha expirado antes que qualquer coisa acontecesse.

O exemplo abaixo mostra o uso de `select(2)` para aguardar até que algum dado seja lido da *stdin*. O timeout é configurado para 5 segundos.

```
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int
7  main (int argc, char **argv)
8  {
9      fd_set rfds;
10     struct timeval tv;
11     int retval;
12
13     /* monitora a stdin (0) */
14     FD_ZERO (&rfds);
15     FD_SET (0, &rfds);
16
17     tv.tv_sec = 5;
```

```
18      tv.tv_usec = 0;
19
20      retval = select (1, &rfds, NULL, NULL, &tv);
21      if ( retval ) {
22          printf ("Dados disponiveis.\n");
23      }
24      else
25          printf ("Nenhum dado veio em 5 segundos\n");
26
27      exit (0);
28  }
```

Vale notar que após a chamada à `select(2)`, o valor de *tv* é desconhecido, e não deve ser usado para obter informações a respeito do tempo que se passou. Um tutorial completo no uso do `select(2)` é fornecido na *man page* do `select_tut(2)`.

7.2 Conexões não-bloqueantes

Após visto como funciona o uso da operação `select(2)`, fica mais fácil entender o processo de uma conexão não-bloqueante. Antes de apresentar os exemplos de um cliente e um servidor escritos seguindo este modelo, algumas funções são importantes de serem vistas:

- `int getsockopt (int s, int level, int optname, void *optval, socklen_t *optlen);`
  `int setsockopt (int s, int level, int optname, const void *optval, socklen_t *optlen);`
  Estas funções manipulam as opções associadas a um socket *s*, que podem existir em múltiplos níveis do protocolo, e estão sempre presentes no nível mais alto de sockets. Quando as opções do socket são manipuladas, o nível em que a opção reside e o nome desta opção devem ser especificadas. Para manipular opções no nível de socket, o parâmetro *level* é especificado como *SOL_SOCKET*. Para manipular opções em algum outro nível do protocolo, o número do protocolo deve ser informado. Por exemplo, para indicar que uma opção será interpretada pelo protocolo TCP, o parâmetro *level* deve conter o valor *TCP*.

  Os parâmetros *optval* e *optlen* são usados para acessar os valores das opções, quando usado com `setsockopt(2)`. Para `getsockopt(2)` elas identificam um buffer no qual o valor para a opção requisitada deverá ser retornado. Neste caso, *optlen* é um parâmetro informando o tamanho do buffer apontado por *optval*. Caso não haja nenhum valor para ser informado ou retornado, *optval* pode ser NULL. As opções disponíveis estão documentadas na *man page* do `socket(7)`.

  optname é passado para o módulo apropriado do protocolo, para que lá ele seja interpretado. O arquivo header <*sys/socket.h*> contém definições para opções a serem utilizadas no nível de sockets.

- `int fcntl(int fd, int cmd);`
  `int fcntl(int fd, int cmd, long arg);`
  `int fcntl(int fd, int cmd, struct flock *lock);`
  A função `fcntl(2)` realiza operações sobre descritores de arquivo (neste caso, especificados por *fd*). A operação é determinada por *cmd*, e dependendo da operação, pode ser utilizado um parâmetro extra, que pode ser um *long* ou um ponteiro para uma *struct flock*. Esta função permite operar com operações do tipo *close-on-exec*, além de permitir obter as flags de status de um arquivo, realizar *advisory lockings* (locks baseados em arquivos) e manipular sinais, entre outros.

A seguir é visto o código de um servidor que faz o uso de conexões não-bloqueantes:

```
1  #include "timeofday.h"
2
3  void
4   processa_clientes (int listenfd, int connfd)
5  {
6       int select_sock_in = 1;
7       int select_sock_out = 1;
8       int ret;
9       char buff[MAXLINE];
10
11      int max_fd = connfd;
12      int sock_EOF = 0;
13      int in_size = 0;
14      int done = 0;
15
16
17      if ( listenfd > max_fd)
18          max_fd = listenfd;
19      max_fd++;
20
21      while (! done) {
22          fd_set write_set, read_set;
23
24          FD_ZERO (&write_set);
25          FD_ZERO (&read_set);
26
27          if ( select_sock_in )
28              FD_SET (connfd, &read_set);
29
30          if ( select_sock_out )
31              FD_SET (connfd, &write_set);
32
33          ret = select (max_fd, &read_set, &write_set, NULL, NULL);
34          if ( ret < 0)
35              err_quit ("select");
36
37          if (FD_ISSET (connfd, &read_set)) {
38              in_size = read (connfd, buff, sizeof (buff));
39              if ( in_size == 0)
40                  sock_EOF = 1;
41              else if ( in_size < 0)
42                  err_quit ("read");
43          }
44
45          if ( in_size > 0) {
46              ret = write (connfd, buff, in_size);
47              if ( ret < in_size ) {
48                  if ( errno != EAGAIN) {
49                      err_quit ("write");
50                  } else {
51                      printf ("EAGAIN em write\n");
52                  }
53              } else {
```

```
54                      in_size = 0;
55                  }
56              }
57
58              if (sock_EOF && in_size == 0) {
59                  close (connfd);
60                  done = 1;
61              }
62
63              select_sock_out = (in_size > 0 || sock_EOF);
64              select_sock_in = (!sock_EOF && in_size == 0);
65          }
66      }
67
68      int
69      main (int argc, char **argv)
70      {
71          int listenfd, optval, connfd;
72          struct sockaddr_in servaddr;
73          long flags;
74          fd_set read_set;
75
76          if (argc != 2) {
77              printf ("Sintaxe: %s <porta>\n", argv[0]);
78              exit (1);
79          }
80
81          if ((listenfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
82              err_quit ("socket");
83
84          optval = 1;
85          if (setsockopt (listenfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval)) < 0)
86              err_quit ("setsockopt");
87
88          memset (&servaddr, 0, sizeof (servaddr));
89          servaddr.sin_family = AF_INET;
90          servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
91          servaddr.sin_port = htons (atoi (argv[1]));
92
93          if (bind (listenfd, (struct sockaddr *) &servaddr, sizeof (servaddr)) < 0)
94              err_quit ("bind");
95
96          if (listen (listenfd, 5) < 0)
97              err_quit ("listen");
98
99          flags = fcntl (listenfd, F_GETFL);
100         fcntl (listenfd, F_SETFL, flags | O_NONBLOCK);
101
102         /* prepara o select para leitura em listenfd */
103         FD_ZERO (&read_set);
104         FD_SET (listenfd, &read_set);
105         if (select (listenfd + 1, &read_set, NULL, NULL, NULL) <= 0)
106             err_quit ("select");
107
```

```
108     if (! FD_ISSET (listenfd, & read_set ))
109         err_quit ("select");
110
111     if ((connfd = accept ( listenfd , NULL, NULL)) < 0)
112         err_quit ("accept");
113
114     flags = fcntl ( listenfd , F_GETFL);
115     fcntl (connfd, F_SETFL, flags | O_NONBLOCK);
116     processa_clientes ( listenfd , connfd);
117     exit (0);
118 }
```

Ao receber uma requisição de um cliente, este servidor lê os dados enviados pelo cliente e os retorna ao cliente. Este é um serviço conhecido como um serviço de *echo*. A análise do código será feita a partir da função *main()*.

- $81 - 82$: cria um socket TCP;
- $84-85$: configura as opções deste socket com a função `setsockopt(2)`. Os parâmetros passados a ela indicam que estamos trabalhando no nível de socket, com a constante *SOL_SOCKET*, e com a opção *SO_REUSEADDR* indicamos que as regras usadas para a validação do endereço usada no `bind(2)` devem permitir a reutilização de um endereço local. Neste caso em que utilizamos um socket da família AF_INET, isto significa que o socket deve realizar o *bind*, a não ser que exista um outro socket ativo escutando no mesmo endereço;
- $87 - 90$: especificação do protocolo, interfaces e portas para usar no *bind*;
- $92 - 93$: faz o *bind*;
- $95 - 96$: prepara o socket para aceitar requisições;
- $98 - 99$: configura as opções do descritor *listenfd* de forma a manter as opções atuais (linha 98), adicionando uma constante informando que ele deverá operar no modo não-bloqueante;
- $102 - 108$: prepara o uso do `select(2)` para o descritor *listenfd*;
- $110 - 111$: aguarda pela conexão de algum cliente;
- $113 - 114$: após receber a conexão de um cliente, prepara o descritor novamente para operar em modo não bloqueante para as próximas operações;
- 115: processa a requisição, explicada abaixo;
- $17 - 19$: define o valor máximo dos descritores em uso para utilizar na chamada ao `select(2);`
- $24-35$: configura os conjuntos de descritores de leitura e escrita para uso no `select(2)`. Após, é aguardado até que o status de algum deles seja modificado. Nota-se que não está sendo usado nenhum timeout nessa operação;
- $37 - 43$: caso o descritor *connfd* tenha sido modificado, o servidor lê os dados a partir do socket representado por este descritor para o buffer *buff*;
- $45 - 56$: se algum dado foi lido anteriormente, então o devolve no mesmo socket para o cliente, escrevendo a quantidade de dados lidas até o momento;
- $58-61$: caso já tenha lido a stream toda e reconhecido o final de arquivo, fecha o descritor e encerra a execução da função.

O arquivo utilizado como header é o mesmo *timeofday.h*, visto que os programas nos exemplos precisam dos mesmos headers e usam a mesma constantes MAXLINE e a macro *err_quit()*.

A seguir é mostrado o programa cliente para este servidor. Ele envia uma stream de bytes para o servidor, contendo os dados de um arquivo texto. A seguir, realiza a leitura da resposta do servidor, e a imprime. Como visto no exemplo do servidor, a resposta vinda do servidor é o próprio conteúdo do arquivo.

```
1  #include "timeofday.h"
2
3  void
4   conecta_ao_servidor (char *filename, int sockfd)
5  {
6       int   select_sock_in  = 1;
7       int   select_sock_out  = 1;
8       int  fd, ret;
9       int   in_size  = 0;
10      int   out_size  = 0;
11     char buff[MAXLINE];
12     char outbuff[MAXLINE];
13
14     int max_fd = sockfd;
15     int file_EOF  = 0;
16     int sock_EOF = 0;
17     int shutdown_write_sock  = 0;
18     int done = 0;
19
20
21     if ((fd = open(filename, O_RDONLY)) <= 0)
22          err_quit ("open");
23
24     if (fd > max_fd)
25          max_fd = fd;
26     max_fd++;
27
28
29     while (!done) {
30          fd_set   write_set, read_set;
31
32          FD_ZERO (&write_set);
33          FD_ZERO (&read_set);
34
35          if ( select_sock_in )
36               FD_SET (sockfd, &read_set);
37
38          if ( select_sock_out )
39               FD_SET (sockfd, &write_set);
40
41          ret = select (max_fd, &read_set, &write_set, NULL, NULL);
42          if (ret < 0)
43               err_quit ("select");
44
45          /* le conteudo do arquivo para o buffer buff */
46          if (! file_EOF && in_size == 0) {
47               in_size = read (fd, buff, sizeof (buff));
48               if (in_size == 0) {
49                    file_EOF = 1;
50                    close(fd);
51               } else if (in_size < 0) {
52                    err_quit ("read");
53               }
```

```
54          }
55
56          /* escreve o conteudo de buff para o sockfd */
57          if ( in_size > 0) {
58               ret = write ( sockfd , buff , in_size );
59               if ( ret < in_size ) {
60                    if ( errno == EAGAIN) {
61                         printf ("EAGAIN em write\n");
62                    } else {
63                         err_quit ("write");
64                    }
65               } else {
66                    in_size = 0;
67               }
68          }
69          if ( file_EOF && in_size == 0 && !shutdown_write_sock ) {
70               shutdown (sockfd , SHUT_WR);
71               shutdown_write_sock = 1;
72          }
73
74          /* le reply do servidor */
75          if ( FD_ISSET (sockfd, &read_set )) {
76               out_size = read ( sockfd , outbuff , sizeof ( outbuff ));
77               if ( out_size == 0) {
78                    sock_EOF = 1;
79                    shutdown (sockfd , SHUT_RD);
80               }
81          }
82
83          if ( out_size > 0)
84               out_size = 0;
85
86          if ( sock_EOF)
87               done = 1;
88
89          select_sock_in = (! sock_EOF && out_size == 0);
90     }
91     printf ("%s", outbuff );
92 }
93
94
95 int
96 main ( int argc , char **argv)
97 {
98     int sockfd;
99     struct sockaddr_in servaddr;
100    struct timeval tv;
101    long flags ;
102    fd_set write_set ;
103
104    int sock_error = 0;
105    socklen_t sock_error_len ;
106
107
```

```
108     if ( argc < 3) {
109         printf ("Sintaxe: %s <arquivo_in> <porta>\n", argv[0]);
110         exit (0);
111     }
112
113     if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
114         err_quit ("socket");
115
116     flags = fcntl (sockfd, F_GETFL);
117     fcntl (sockfd, F_SETFL, flags | O_NONBLOCK);
118
119     memset (&servaddr, 0, sizeof (servaddr));
120     servaddr.sin_family = AF_INET;
121     servaddr.sin_addr.s_addr = htonl (INADDR_LOOPBACK);
122     servaddr.sin_port = htons (atoi (argv[2]));
123
124     /* aguarda no maximo 30 segundos */
125     tv.tv_sec = 30;
126     tv.tv_usec = 0;
127
128     if (connect (sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) == 0) {
129         /* conexao ocorreu imediatamente, nao e' necessario usar select() */
130     } else {
131         switch (errno) {
132             case EINPROGRESS: /* select para escrita */
133                 FD_ZERO (&write_set);
134                 FD_SET (sockfd, &write_set);
135                 if (select (sockfd + 1, NULL, &write_set, NULL, &tv) <= 0)
136                     err_quit ("select");
137
138                 if (! FD_ISSET (sockfd, &write_set))
139                     err_quit ("select");
140                 break;
141
142             default:
143                 err_quit ("connect");
144         }
145     }
146
147     sock_error_len = sizeof (sock_error);
148     if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &sock_error, &sock_error_len) < 0)
149         err_quit ("getsockopt");
150
151     conecta_ao_servidor (argv[1], sockfd);
152     exit (0);
153 }
```

Nesta versão do cliente temos também algumas informações relevantes e que valem a pena serem descritas. Novamente, começando a partir da função *main()*:

- $113 - 117$: cria um socket TCP e configura opções para o socket;
- $119 - 122$: conecta-se ao host local. Uma máquina em particular pode ter mais de um endereço IP, cada um correpondendo a uma interface de rede diferente. O uso da constante INADDR_ANY permite que o *bind* seja feito em todas elas simultaneamente;
- $125 - 126$: especifica o timeout a ser usado na chamada ao `select(2)`;

- 128: tenta conectar-se ao servidor. Caso consiga de imediato, segue adiante, ou aguarda pela mudança do estado do descritor de escrita do socket *sockfd* até receber o reply do servidor. O timeout imposto no exemplo é de 30 segundos;
- $147 - 149$: pega e limpa qualquer erro que tenha acontecido no socket;
- 151: conecta-se ao servidor, executando os passos descritos abaixo;
- $21 - 22$: abre o arquivo origem a ser enviado para o servidor;
- $24 - 26$: determina o maior descritor utilizado, para uso no `select(2)`;
- $32 - 43$: prepara os conjuntos de descritores de leitura e escrita a serem monitorados pelo `select(2)`; Esta chamada é desbloqueada assim que algum deles tiver seu valor modificado.
- $46 - 54$: lê o conteúdo do arquivo para o buffer especificado por *buff*;
- $57 - 68$: caso tenha sido lido algum conteúdo, escreve-o para o socket *sockfd*, onde está estabelecida a conexão com o servidor;
- $69 - 72$: caso o fim de arquivo tenha sido encontrado, chama `shutdown(2)`. Esta função finaliza totalmente ou parcialmente uma conexão full-duplex no socket especifiado como primeiro argumento. Caso o segundo parâmetro seja SHUT_RD, não são permitidas recepções futuras neste socket. Ele pode ainda ser SHUT_WR, que impede transmissões sobre o socket, ou ainda SHUT_RDWR, que impede recepções e transmissões sobre este socket. Neste exemplo, ele finaliza o socket para escrita;
- $75 - 80$: lê o reply do servidor para o buffer *outbuff*. Caso ele tenha lido o sinal de fim de arquivo, finaliza o socket para leitura, usando a chamada `shutdown(2)`;
- 91: imprime o buffer lido, que é o mesmo que foi enviado.

Capítulo 8

Bancos de dados

8.1 Implementações

Existem diversas implementações de servidores de banco de dados disponíveis para o GNU/Linux. Entre eles, pode-se citar o MySQL, PostgreSQL e o Firebird, três implementações excelentes e bastante estáveis de banco de dados relacionais para o GNU/Linux. O banco de dados usado nos exemplos será o MySQL.

8.2 Programando com Bancos de Dados em C

A API para programação em C é distribuída com o MySQL. Ele inclui a biblioteca *mysqlclient* e permite que programas em C acessem bancos de dados [MyS 2003].

Para conectar ao servidor, é feita uma chamada à função `mysql_init()` para inicializar o handler da conexão, e então a função `mysql_real_connect()` é chamada com este handler (além de outras informações como o hostname, nome do usuário e senha). Sobre esta conexão, `mysql_real_connect()` configura o valor da flag *reconnect* (parte da estrutura do MySQL) para 1. Esta flag indica que, caso uma *query* não possa ser efetuada devido à uma conexão perdida, uma nova tentativa de reconexão deve ser feita antes de retornar um erro. Quando a conexão estiver encerrada, a função `mysql_close()` a termina.

Enquanto uma conexão estiver ativa, o cliente pode enviar queries SQL para o servidor usando `mysql_query()` ou `mysql_real_query()`. A diferença entre estas duas funções está em que `mysql_query()` espera que a query seja especificada como uma string terminada por NULL, enquanto a função `mysql_real_query()` espera uma string de tamanho já contado. Caso a string contenha algum dado binário, deve ser utilizada a `mysql_real_query()`.

Para queries que não envolvem seleção, como INSERT, UPDATE e DELETE, é possível ver quantas colunas foram afetadas através da função `mysql_affected_rows()`. Para queries do tipo SELECT, as colunas selecionadas são obtidas na forma de um conjunto de resultados.

Estes conjuntos de resultados podem ser processados de duas formas pelo cliente. Uma delas é pela chamada à `mysql_store_result()`. Essa função obtém do servidor todas as colunas retornadas pela query e as armazena no cliente. A segunda forma é pelo uso da função `mysql_use_result()`, que inicia uma recuperação coluna a coluna a partir do conjunto de resultados retornado.

Em ambos casos, o acesso às colunas é feito usando a chamada `mysql_fetch_row()`. Com `mysql_store_result()`, `mysql_fetch_row()` acessa as colunas que já foram lidas do servidor. O tamanho dos dados em cada coluna pode então ser informado através da função `mysql_fetch_lengths()`.

Após o processamento do conjunto de resultados, a função `mysql_free_result()` libera a memória utilizada durante ele.

Com estas descrições já é possível escrever um aplicativo que comunique-se com o servidor de banco de dados, mas antes é necessário criar o banco de dados e a tabela. Para isto, é necessário executar alguns passos, descritos logo abaixo:

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 4.0.15a

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> CREATE DATABASE tabela_teste;
Query OK, 1 row affected (0.06 sec)

mysql> GRANT select, insert, update, delete
    -> ON tabela_teste.* TO user@localhost IDENTIFIED by 'senha'
```

Com isto o banco de dados *tabela_teste* foi criada, e foi garantido acesso de inserção, update e exclusão para o usuário *user@localhost*. A senha deve ser informada entre aspas simples.

Após criado o banco de dados, é necessário criar a tabela sobre a qual as operações serão feitas:

```
mysql> USE tabela_teste;
Database changed
mysql> CREATE TABLE produtos (
    -> name varchar(80) not null,
    -> num int not null,
    -> PRIMARY KEY (num)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW TABLES;
+------------------------+
| Tables_in_tabela_teste |
+------------------------+
| produtos               |
+------------------------+
1 row in set (0.00 sec)

mysql> DESCRIBE produtos;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| name  | varchar(80) |      |     |         |       |
| num   | int(11)     |      | PRI | 0       |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

Uma tabela chamada *produtos* foi criada sobre o banco *tabela_teste*, com uma string chamada *varchar* e um inteiro chamado *num*. A partir de agora é possível operar sobre estas estruturas utilizando um programa em C. O exemplo abaixo conecta-se ao servidor MySQL e realiza uma inserção de dados sobre a tabela criada, *produtos*. Para compilar este exemplo basta:

```
gcc insert_test.c -o insert_test -Wall `mysql_config --libs`
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mysql/mysql.h>
4
5  #define INSERT_QUERY "insert into produtos (name,num) values ('item %d', %d)"
6
7
8  int
9  main(int argc, char **argv)
10 {
11     MYSQL *sock = NULL;
12     MYSQL *mysql = NULL;
13     char qbuf[160];
14     int count, num;
15
16     if (argc != 3) {
17         fprintf (stderr, "Sintaxe: %s <dbname> <numero>\n", argv[0]);
18         exit (1);
19     }
20
21     mysql = mysql_init (mysql);
22     if (!mysql) {
23         fprintf (stderr, "mysql_init: %s\n", mysql_error (mysql));
24         exit (1);
25     }
26
27     sock = mysql_real_connect (mysql, "localhost", "lucasvr", "senha", argv [1], 0, NULL, 0);
28     if (!sock) {
29         fprintf (stderr, "mysql_real_connect: %s\n", mysql_error (mysql));
30         exit (1);
31     }
32
33     num = atoi (argv [2]);
34     count = 0;
35     while (count < num) {
36         sprintf (qbuf, INSERT_QUERY, count, count);
37         if (mysql_query (sock, qbuf)) {
38             fprintf (stderr, "mysql_query: %s\n", mysql_error (sock));
39             exit (1);
40         }
41         count++;
42     }
43
44     mysql_close (sock);
45     exit (0);
46 }
```

A execução deste programa pode ser feita da seguinte maneira:

```
$ ./insert_test tabela_teste 5
```

E o resultado será refletido imediatamente no banco de dados:

```
mysql> SELECT * from produtos;
+--------+-----+
| name   | num |
+--------+-----+
| item 0 |   0 |
| item 1 |   1 |
| item 2 |   2 |
| item 3 |   3 |
| item 4 |   4 |
+--------+-----+
5 rows in set (0.00 sec)
```

Capítulo 9

Interface Gráfica

9.1 Servidor X

O *X Window System*[SCH 86] foi criado em 1984 no Massachussets Institute of Technology (MIT) para dar suporte a dois projetos: o projeto Athena, que precisava de um sistema de janelas que pudesse ser usado em centenas de estações de trabalho e o projeto Argus, que precisava de um ambiente de depuração para aplicações multiprocessadas distribuídas [KOJ 2000].

O *X Window System* define padrões que devem ser adotados por implementações deste ambiente. O ambiente que o implementa e dá suporte para os sistemas operacionais UNIX e GNU/Linux é o XFree86. Nestas plataformas ele é distribuído sob uma licença livre, e em outras ele o é sobre a forma de uma licença comercialial, como no OS/2 e no QNX.

O sistema possui arquitetura cliente-servidor. O servidor localiza-se na estação de trabalho do usuário e provê acesso transparente ao hardware gráfico, mecanismos de comunicação entre clientes e entrada de dados por dispositivos como mouse e teclado. Os clientes podem ser executados na própria estação ou remotamente, conectados ao servidor através de uma rede.

Qualquer coisa que se queira apresentar na tela deve ser desenhada dentro de uma janela. Uma aplicação pode usar muitas janelas de cada vez. Menus e outros elementos da interface podem ser construídos com janelas que se sobrepõem às demais. As janelas são organizadas em uma hierarquia, o que facilita o gerenciamento. Qualquer janela, exceto uma, chamada de raiz, é subjanela de outra.

O X foi projetado de forma a fornecer mecanismos para que diferentes políticas de interface com o usuário pudessem ser implementadas, ou seja, é como um driver de vídeo, teclado e mouse. Portanto, o X não é nem possui uma interface com o usuário. A função de prover tal interface é delegada aos clientes e bibliotecas especiais externos ao servidor X, chamadas toolkits.

A seguinte estrutura deve ser respeitada para a criação de uma aplicação no X:

- Inicialização: durante a inicialização, o programa fará o tratamento de opções de linha de comando, conexão com o servidor X e alocação de estruturas internas. Tudo isto é feito internamente pelo toolkit, caso seja utilizado um.
- Montagem da interface gráfica: após aberta a conexão com o servidor, o programa pode criar e montar sua interface (janelas, menus, botões etc.) utilizando os widgets (elementos da interface) oferecidos pelo toolkit, ou usando as funções primitivas da Xlib, explicada na Sessão 9.2
- Laço de tratamento de eventos: finalmente, o programa entra em um laço, onde fica esperando por eventos vindos do servidor, como pressionamento de teclas ou botões do mouse, movimento do mouse ou requisição do servidor para redesenhar o conteúdo de suas janelas. Em toolkits, estes eventos são primeiro tratados internamente e, dependendo

do caso (quando o mouse é clicado em um widget botão, por exemplo), são repassados ao programa através de callbacks ou mecanismos similares.

9.2 A biblioteca Xlib

A Xlib é a API (Application Programs Interface) de mais baixo nível disponível para se programar aplicações para X (a não ser que a aplicação utilize o protocolo X diretamente para se comunicar com o servidor). Ela foi definida como padrão pelo X Consortium e é uma biblioteca em linguagem C, que oferece funções com correspondência direta, ou quase, com o protocolo X. Possui funções para abrir uma conexão com um servidor (display), criação, destruição e manipulação de janelas, desenho e recebimento de eventos, entre outros.

Não é aconselhável usar apenas a Xlib para escrever aplicações complexas, que possuam interfaces com usuário sofisticadas, pois além disso ser muito trabalhoso levará à existência de aplicações com interfaces inconsistentes umas com a as outras. É por isto que existem os toolkits, que utilizam a Xlib na sua implementação, mas oferecem um maior grau de abstração e funções de mais alto nível. Os toolkits permitem, além da construção de interfaces com o usuário, tratar de outras necessidades da aplicação, tais como configuração, tratamento de erros e acesso a serviços do sistema operacional.

9.3 Gerenciadores de Janelas

Programas que realizam operações de *baixo nível*, tais como gerenciadores de janelas e processadores de imagens, terão obrigatoriamente que usar as funções da Xlib. Uma abordagem bastante comum em aplicações complexas é usar um toolkit para construir a interface e funções da Xlib para operações de baixo nível.

Gerenciadores de janelas são muito úteis porque implementam as políticas de atribuição do foco, redimensionamento e sobreposição de janelas. Eles fazem isto com base em um conjunto de convenções estabelecidas entre ele e os clientes, usando mecanismos diversos como trocas de mensagens e eventos.

Normalmente o gerenciador de janelas utilizado no desktop do programador o influencia na tomada de decisão sobre qual biblioteca de *widgets* usar para criar suas aplicações gráficas. Alguns exemplos são o KDE, que utiliza o toolkit *Qt* (C++) e o GNOME, que usa o *GTK+* (C).

A Sessão 9.4 apresenta o modelo de programação usado com a toolkit *GTK+*.

9.4 GTK+

GTK+ é uma biblioteca de widgets escrita em C, com inspiração em orientação a objetos. Devido a essa característica, a programação nela pode se tornar um tanto tediosa, pois é necessário emular a programação orientada a objetos "na mão", por meio de uma disciplina de programação.

A criação de um programa GTK+ é feita conforme os seguintes passos mostrados no exemplo abaixo (este programa deve ser compilado com a flag `-lgtk` para ser linkado com a biblioteca GTK+):

```
1 #include <unistd.h>
2 #include <gtk/gtk.h>
3
4 void
5  quit_hello  (GtkWidget *widget, gpointer  data)
6 {
7       gtk_main_quit  ();
8 }
9
```

```
10 int
11 main (int argc , char **argv)
12 {
13     GtkWidget *window, *button;
14
15     gtk_init (&argc, &argv );
16
17     window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
18     gtk_signal_connect ( GTK_OBJECT (window), "destroy",
19             GTK_SIGNAL_FUNC (quit_hello), NULL);
20
21     button = gtk_button_new_with_label ("Ola");
22     gtk_signal_connect_object ( GTK_OBJECT (button), "clicked",
23             GTK_SIGNAL_FUNC (gtk_widget_destroy),
24             GTK_OBJECT (window));
25
26     gtk_container_add ( GTK_CONTAINER (window), button);
27     gtk_widget_show ( button );
28     gtk_widget_show ( window);
29
30     gtk_main ();
31     exit (0);
32 }
```

Este é um dos programas mais simples que pode ser escrito em GTK+, que faz o seguinte:

- $4-8$: declara a função que será chamada quando a janela receber o sinal *destroy*;
- 15: inicialização do GTK+. Nesta etapa é realizada a conexão com o servidor X, o tratamento de opções de linha de comando padrão e outras inicializações internas;
- 17: criação de uma janela principal (*TOPLEVEL*);
- $18-19$: conexão do sinal *destroy* à função *quit_hello()*;
- 21: criação de um botão;
- $22-24$: quando o botão receber o sinal *clicked*, chama a função *gtk_widget_destroy*, passando como parâmetro o objeto *window*;
- 26: o botão é colocado dentro da janela principal;
- 27: faz com que o widget seja visível;
- 28: faz com que a janela seja visível na tela;
- 30: entra em um loop infinito de processamento de eventos.

Como pode-se imaginar, na medida em que o programa cresce, torna-se inviável escrevê-lo diretamente sobre o GTK+, pois o usuário terá o compromisso de escrever todas as rotinas de posicionamento de janelas e botões manualmente, o que acaba tomando muito tempo e levando à ocorrência dos bugs mais diversos.

Uma solução que vários projetos têm criado é a adoção de uma interface gráfica para manipular as bibliotecas de widgets. No caso do QT, pode ser utilizado o *Designer* para desenvolver aplicações gráficas muito rapidamente, e com o GTK+ pode ser usado o *Glade*, que também aumenta muito a produção de códigos baseados no ambiente de janelas do XFree86.

As Figuras 9.1 e 9.2 mostram o processo de criação de um botão em uma nova janela e uma janela mais complexa, contendo botões e *combo boxes*.

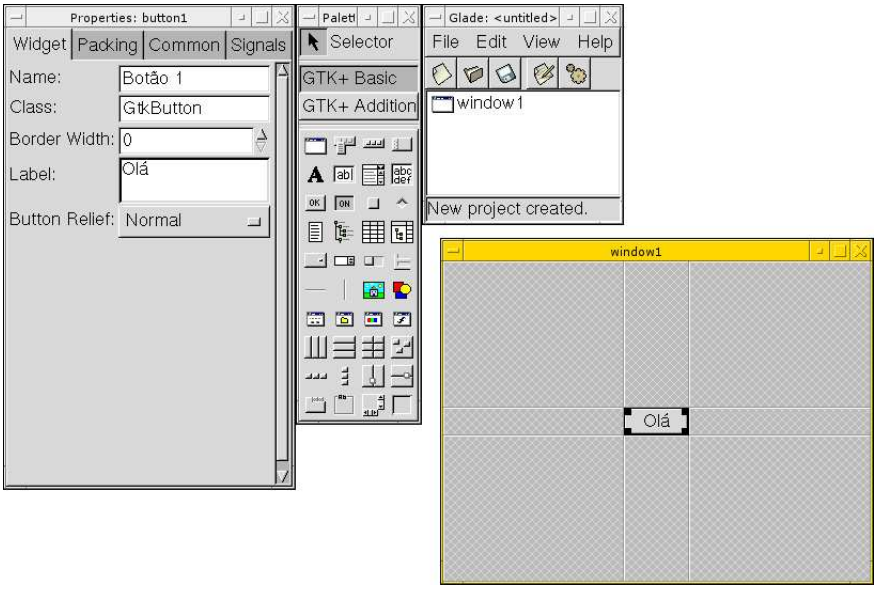

Figura 9.1: Inserindo um botão em uma nova janela com o Glade

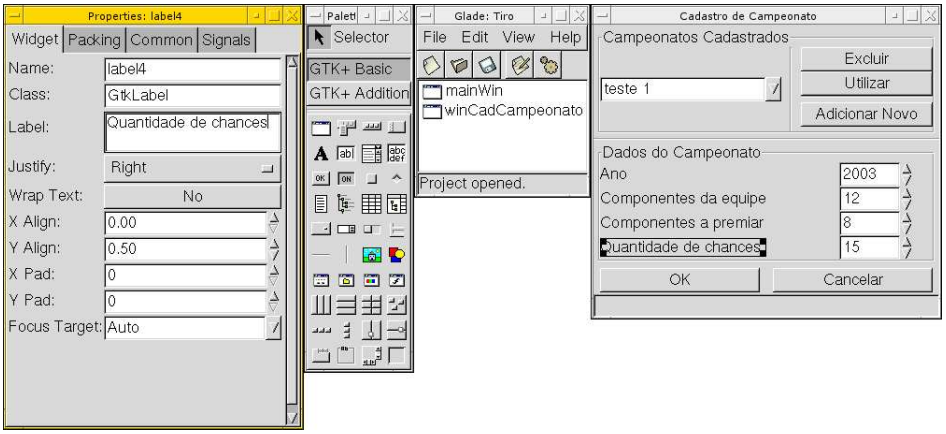

Figura 9.2: Construções mais complexas são construídas rapidamente no Glade

Capítulo 10

Comunicação

Em UNIX, tudo é um arquivo. Seguindo esta filosofia, a maneira utilizada para programar dispositivos é a mesma usada para tratar arquivos utilizando operações com descritores de arquivos. No entanto, operações sobre certos dispositivos podem ser triviais o suficiente para não necessitar o uso destes recursos, ou complexas demais a ponto de ser importante utilizar alguma biblioteca para auxiliar as operações sobre estes dispositivos.

A seguir são apresentadas as formas de acesso aos dispositivos serial, paralelo e USB.

10.1 Porta Paralela

Existem duas formas de acesso à dispositivos na porta paralela. O que acontece com ela no Linux é que existe um driver padrão chamado `lp`, que já dá suporte ao acesso à porta. Neste caso, o dispositivo criado na entrada `/dev` (ou outro diretório usado para armazenar os dispositivos, dependendo da distribuição) é o `lp0` para a primeira porta paralela, `lp1` para a segunda e assim sucessivamente. A porta paralela também pode ser programada utilizando diretamente o dispositivo `port`, usando operações sobre descritores de arquivos como `read(2)` e `write(2)`.

No entanto, a forma mais simples e rápida de programá-la é com o uso da família `inb(2)` e `outb(2)` [RUB 2001]. A seguir são definidos os protótipos das macros associadas à elas:

- `unsigned char inb (unsigned short int port);`
  `void outb (unsigned char value, unsigned short int port);`
  lêem ou escrevem bytes (8 bits) à porta *port*. O argumento *port* é definido como *unsigned long* para algumas plataformas, e *unsigned short* para outras. O tipo de retorno de *inb* também difere entre diferentes arquiteturas.
- `unsigned short int inw (unsigned short int port);`
  `void outw (unsigned char value, unsigned short int port);`
  acessam portas de 16-bits. Elas não estão disponíveis nas plataformas *M68K* e *S390*, que suportam apenas I/O orientado a bytes.
- `unsigned int inl (unsigned short int port);`
  `void outl (unsigned char value, unsigned short int port);`
  estas macros acessam portas de 32-bits. *value* é declarado *unsigned long* ou *unsigned int*, de acordo com a plataforma. Assim como em I/O de 16-bits, esta operação não está disponível nos *M68K* e *S390*.
- `void insb (unsigned short int port, void *addr, unsigned long int count);`
  `void outsb (unsigned short int port, void *addr, unsigned long int count);`
  lêem ou escrevem *count* bytes, iniciando no endereço de memória *addr*. Os dados são

lidos e escritos na porta *port*.

Estas macros {in,out}s* operam com os mesmo tipos das macros anteriores, com a diferença de que estas são orientadas a strings. Alguns processadores implementam instruções especiais para transferir uma seqüência de bytes, words ou longs de e para uma porta de I/O. No caso da arquitetura não suportar este tipo de instrução, elas são implementadas via software na forma de um loop.

- `void insw (unsigned short int port, void *addr, unsigned long int count);`
  `void outsw (unsigned short int port, void *addr, unsigned long int count);`
  lêem ou escrevem valores de 16-bits para uma porta de 16-bits.
- `void insl (unsigned short int port, void *addr, unsigned long int count);`
  `void outsl (unsigned short int port, void *addr, unsigned long int count);`
  lêem ou escrevem valores de 32-bits para uma porta de 32-bits.
- `unsigned char inb_p (unsigned short int port);`
  `void outb_p (unsigned char value, unsigned short int port);`
  `unsigned short int inw_p (unsigned short int port);`
  `void outw_p (unsigned char value, unsigned short int port);`
  `unsigned int inl_p (unsigned short int port);`
  `void outl_p (unsigned char value, unsigned short int port);`
  Algumas plataformas – principalmente o i386 – pode ter problemas quando o processador tenta transferir dadas muito rapidamente do ou para o bus. Os problemas podem ocorrer porque o processador é *overclocked* em respeito ao bus ISA, que podem acontecer quando os dispositivos da placa são muito lentos. A solução nestes casos é inserir um pequeno delay entre cada instrução de I/O. Nas arquiteturas que não enfrentam este problema, esta macro é expandida para as funções correspondentes sem o "*_p*".

Estas funções são na verdade definidas como macros inline, e precisam ser compiladas com nível de otimização habilitado para serem substituídas, caso contrário as referências não poderão ser resolvidas em tempo de linkagem do programa. Podem ser utilizadas as flags *-O* ou *-O2*, por exemplo.

Para poder operar sobre as portas seriais e paralelas em espaço de usuário, é necessário requisitar acesso para o kernel para as portas de I/O em questão. Para isto são utilizados os programas `ioperm(2)` ou o `iopl(2)`:

```
#include <unistd.h> /* para a libc5 */
#include <sys/io.h> /* para a glibc */
int ioperm (unsigned long from, unsigned long num, int turn_on);
```

`ioperm(2)` configura os bits de acesso à porta para o processo para *num* bytes, começando na porta *from*, para o valor *turn_on*. O uso desta função requer privilégios de superusuário. As permissões de acesso não são herdadas no `fork(2)`, mas são no `exec(3)`. Além disto, `ioperm(2)` permite acesso para as primeiras 0x3ff portas de I/O. Para portas fora deste intervalo, o `iopl(2)` deve ser utilizado:

```
#include <sys/io.h>
int iopl(int level);
```

Esta função altera os privilégios de I/O do processo atual para o nível especificado em *level*. Assim, o processo tem acesso irrestrito à portas de acesso I/O, inclusive permitindo que o processo desabilite interrupções (o que provavelmente irá causar um crash no sistema). Diferente da `ioperm(2)`, esta função permite que as permissões sejam herdadas tanto pelo `fork(2)` como pelo `exec(3)`. O nível padrão para um processo normal é 0, e vai até 3.

10.2 Porta Serial

Portas seriais são identificadas no Linux através dos dispositivos `ttySn`, onde *n* indica o número da porta serial. Por exemplo, a porta `ttyS0` referencia a primeira porta serial, equivalente à COM1 no DOS. No entanto, pelo fato de ter características extras a serem consideradas quando programadas – como a *baud rate* e paridade de bits –, a programação sobre portas paralelas envolve alguns passos extras.

Existem três modos de operação de leitura sobre portas seriais, que são:

- *Modo Canônico*: trabalha sobre *linhas*, ou seja: uma operação de leitura só irá retornar quando tiver lido um caracter de fim de linha (\n) ou um caracter de *carriage return* (\r). Nesse modo, caracteres especiais como *erase, kill, delete* são reconhecidos e tratados como tal.
- *Modo Não-Canônico (raw)*: o processamento não é realizado sobre linhas, mas sobre quantidades fixas de caracteres, e a leitura nào é afetada caso caracteres especiais sejam lidos, como o *erase* e o *delete*.
- *Modo Assíncrono*: este modo permite que a chamada `read(2)` retorne imediatamente, sendo que um sinal é enviado ao programa (SIGIO) quando os dados estiverem disponíveis para leitura.

Outro detalhe a ser visto é o da configuração da porta serial: como o programa irá modificá-la, é interessante salvar o estado da porta antes de alterá-la, e então restaurar o seu estado ao término do programa. Feito isso, basta especificar as informações necessárias para configurar a porta serial e iniciar a leitura e escrita de dados à porta com o uso da chamada `read(2)` e `write(2)`.

10.2.1 A interface Termios

Para lidar com a serial no Linux, uma estrutura chamada *struct termios* é usada. Ela apresenta os seguinte membros:

- *tcflag_t c_iflag*: opções de entrada de dados;
- *tcflag_t c_oflag*: opções para a saída de dados;
- *tcflag_t c_cflag*: descreve opções de controle;
- *tcflag_t c_lflag*: flags para o modo local (canônico, não-canônico);
- *cc_t c_line*: disciplina da linha;
- *cc_t c_cc[]*: caracteres de controle.

As opções de controle, representadas pela c_cflag, controlam a *baud rate*, a quantidade de bits de dados, paridade, stop bits e o controle de fluxo de hardware. A constantes disponíveis para serem usadas podem ser vistas na Tabela 10.1

Duas opções da *c_cflag* devem estar sempre habilitadas: `CLOCAL` e `CREAD`. Elas irão garantir que o programa não virá a ser o "proprietário" da porta relativo à controles esporádicos de job e de sinais de *hangup*, e que o driver da interface serial será capaz de ler dados de entrada.

Devido às diferentes interfaces que são disponíveis para configurar a *baud rate* entre vários sistemas operacionais, costuma-se usar as funções `cfsetispeed(3)` e `cfsetospeed(3)` para informar a taxa de entrada e saída, respectivamente.

Relativo à paridade, basta combinar as flags sobre a *c_cflag* para obter a paridade desejada. Na listagem abaixo assume-se que a *c_cflag* foi previamente inicializada com algum outro campo (como o CLOCAL | CREAD):

- Sem paridade (8N1): options.c_cflag |= CS8;
- Com paridade par (7E1): options.c_cflag |= CS7 | PARENB;
- Com paridade ímpar (7O1): options.c_cflag |= CS7 | PARENB | PARODD;

Tabela 10.1: Constantes para o membro c_cflag

Constante	**Descrição**
B0	0 baud (descarta informação de Data Terminal Ready)
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud
B57600	57,600 baud
B76800	76,800 baud
B115200	115,200 baud
EXTA	Taxa externa de clock
EXTB	Taxa externa de clock
CS5	5 bits de dados
CS6	6 bits de dados
CS7	7 bits de dados
CS8	8 bits de dados
CSTOPB	2 stop bits (1 otherwise)
CREAD	Habilita recepção de dados
PARENB	Habilita bit de paridade
PARODD	Usa paridade ímpar ao invés de par
HUPCL	Hangup (descarta DTR) no último close
CLOCAL	Linha local - não modifica o "proprietário" da porta
LOBLK	Bloqueia saída de controle de jobs
CRTSCTS	Habilita fluxo de controle em hardware

- Com paridade de espaço (7S1): utiliza a mesma configuração do modo sem paridade (8N1).

As opções locais podem ser configuradas sobre o membro *c_lflag*, e ditam como os caracteres de entrada (leitura) serão manipulados pelo driver serial. Normalmente ele será configurado para o modo canônico ou não-canônico (raw). A Tabela 10.2 mostra as opções disponíveis para este campo.

As opções de entrada são informadas pelo membro *c_iflag*, de acordo com os valores vistos na Tabela 10.3.

O membro *c_oflag* contém opções de filtro de saída. Como nos modos de entrada, pode ser escolhido modo processado ou raw. A Tabela 10.4 mostra as opções disponíveis.

De todas estas opções, provavelmente a única utilizada será a `ONLCR` para mapear novas linhas para pares CR-LF. O restante das opções são mantidas por razões históricas e datam da época em que impressoras e terminais não conseguiam manter-se em sincronia com as streams

Tabela 10.2: Constantes para o membro c_lflag

Constante	Descrição
ISIG	Habilita sinais SIGINTR, SIGSUSP, SIGDSUSP e SIGQUIT
ICANON	Habilita modo de entrada canônico (raw mode caso contrário)
XCASE	Mapeia uppercase para lowercase (obsoleto)
ECHO	Habilita eco de caracteres de entrada
ECHOE	Ecoa o caracter de erase como BS-SP-BS
ECHOK	Ecoa nova linha (NL) após o caracter de kill
ECHONL	Ecoa NL
NOFLSH	Desabilita flush de buffers de enrada após caracteres de interrupt ou quit
IEXTEN	Habilita funções estendidas
ECHOCTL	Ecoa caracteres de controle como ĉhar e delete como ?
ECHOPRT	Ecoa caracteres apagados como um caractere apagado
ECHOKE	BS-SP-BS em toda a linha quando receber um line kill
FLUSHO	Saída sendo flushed
PENDIN	Processa entrada pendente na próxima leitura ou caracter de entrada
TOSTOP	Envia SIGTTOU para saída em background

Tabela 10.3: Constantes para o membro c_iflag

Constante	Descrição
INPCK	Habilita checagem de paridade
IGNPAR	Ignora erros de paridade
PARMRK	Marca erros de paridade
ISTRIP	Tira bits de paridade
IXON	Habilita controle de fluxo de software para saída
IXOFF	Habilita controle de fluxo de software para entrada
IXANY	Permite que qualquer caracter inicie o fluxo novamente
IGNBRK	Ignora condição de break
BRKINT	Envia um sinal SIGINT quando uma condição de break for detectada
INLCR	Mapeia nova linha (NL) para *carriage return* (CR)
IGNCR	Ignora caracter CR
ICRNL	Mapeia CR para NL
IUCLC	Mapeia uppercase para lowercase
IMAXBEL	Ecoa BEL para linhas de entrada muito longas

de dados da porta serial.

O último conjunto de opções que restam para permitir a configuração da porta serial são os caracteres de controle. Eles são configuráveis em um array chamado *c_cc* que contém tanto definições de caracteres de controles como parâmetros de timeout. Alguns ds dados suportados neste array estão listados na Tabela 10.5. O restante pode ser consultado a partir do header `bits/termios.h`.

10.2.2 Modo canônico

Para comunicar-se com um modem configurável via serial, pode-se utilizar o modo canônico – orientado a linhas. O programa a seguir pode ser utilizado para resetar o modem e ler os dados enviados por ele. Neste exemplo, não há uma forma de encerrar o programa de forma natural,

Tabela 10.4: Constantes para o membro c_oflag

Constante	**Descrição**
OPOST	Pós-processa a saída (não setado = saída raw)
OLCUC	Mapeia lowercase para uppercase
ONLCR	Mapeia NL para CR-NL
OCRNL	Mapeia CR para NL
NOCR	Não gera CR na saída na coluna 0
ONLRET	NL realiza a função de CR
OFILL	Usa caracteres de preenchimento (fill) para delay
OFDEL	Caractere de preenchimento é DEL
NLDLY	Máscara para delay entre linhas
NL0	Nenhum delay para NLs
NL1	Delay de novas saídas após NL para 100 mili-segundos
CRDLY	Máscara para delay para retornar o carro para a coluna da esquerda
CR0	Nenhum delay para CRs
CR1	Delay após CRs dependendo da posição da coluna atual
CR2	Delay de 100 mili-segundos após enviar CRs
CR3	Delay de 150 mili-segundos após enviar CRs
TABDLY	Máscara para delay após TABs
TAB0	Nenhum delay para TABs
TAB1	Delay após TABs dependendo da posição da coluna atual
TAB2	Delay de 100 mili-segundos após enviar TABs
TAB3	Expande caracteres de TAB para espaços
BSDLY	Máscara para delay após BSs
BS0	Nenhum delay para BSs
BS1	Delay de 50 mili-segundos após enviar BSs
VTDLY	Máscara para delay após VTs
VT0	Nenhum delay para VTs
VT1	Delay de 2 segundos após enviar VTs
FFDLY	Máscara para delay após FFs
FF0	Nenhum delay para FFs
FF1	Delay de 2 segundos após enviar FFs

Tabela 10.5: Constantes para o membro c_cc[]

Constante	**Descrição**	**Valor default**
VINTR	Interrupt	CTRL-C
VQUIT	Quit	CTRL-Z
VERASE	Erase	Backspace (BS)
VKILL	Kill-line	CTRL-U
VEOF	Fim de arquivo	CTRL-D
VEOL	Fim de linha	Carriage return (CR)
VEOL2	Segundo fim de linha	Line feed (LF)
VMIN	Número mínimo de caracteres para serem lidos	-
VTIME	Tempo para aguardar por dados (décimos de segundos)	-

pois o loop de leitura é infinito. Em um caso real, seria necessário verificar o conteúdo do buffer lido para decidir quando encerrar o loop.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <termios.h>
9
10 /* Configuracoes de baud rate sao definidas em <asm/termbits.h> */
11 #define BAUDRATE B9600
12 #define DEVICE "/dev/ttyS0"
13
14 int
15 main (int argc, char **argv)
16 {
17     int fd, res;
18     struct termios oldtio, newtio;
19     char buf[255];
20
21     /*
22      * Abre porta para leitura/escrita e nao como uma tty de controle, para
23      * evitar que o programa encerre se vier um CTRL+C na linha
24      */
25     fd = open (DEVICE, O_RDWR | O_NOCTTY);
26     if (fd < 0) {
27         perror ("open");
28         exit (1);
29     }
30
31     /* Salva a configuracao atual da porta serial e inicia uma nova */
32     tcgetattr (fd, &oldtio);
33     memset (&newtio, 0, sizeof (newtio));
34
35     /*
36      * CRTSCTS : Controle de fluxo de saida do hardware
37      * CS8     : 8n1 (8 bit, no parity, 1 stopbit)
38      * CLOCAL  : Conexao local, sem controle do modem
39      * CREAD   : Habilita a recepcao de caracteres
40      */
41     newtio.c_cflag = CRTSCTS | CS8 | CLOCAL | CREAD;
42
43     /*
44      * IGNPAR : ignora bytes com erros de paridade
45      * ICRNL  : mapeia CR (carriage return) para NL (new line), senao um CR nos
46      *          dados de entrada pode nao terminar a leitura
47      */
48     newtio.c_iflag = IGNPAR | ICRNL;
49
50     /* Seta baud rate de entrada e saida */
51     cfsetispeed (&newtio, BAUDRATE);
```

```
52      cfsetospeed (&newtio, BAUDRATE);
53
54      /* Saida em modo 'raw' */
55      newtio.c_oflag = 0;
56
57      /*
58       * ICANON: habilita entrada em modo canonico
59       * desabilita funcionalidades de 'echo', e nao envia sinais para o programa
60       */
61      newtio.c_lflag = ICANON;
62
63      /*
64       * Inicializa todos os caracteres de controle.
65       * Os valores padrao podem ser encontrados em '/usr/include/termios.h'
66       */
67      newtio.c_cc[VINTR]    = 0;     // 'Ctrl-C
68      newtio.c_cc[VQUIT]    = 0;     // Ctrl-'\'
69      newtio.c_cc[VERASE]   = 0;     // DEL
70      newtio.c_cc[VKILL]    = 0;     // @
71      newtio.c_cc[VEOF]     = 4;     // Ctrl-D */
72      newtio.c_cc[VTIME]    = 0;     // timeout entre leituras (t=VTIME * 0.1 segs)
73      newtio.c_cc[VMIN]     = 5;     // bloqueia leitura ate' ler 5 caracteres
74      newtio.c_cc[VSWTC]    = 0;     // '\0'
75      newtio.c_cc[VSTART]   = 0;     // Ctrl-Q
76      newtio.c_cc[VSTOP]    = 0;     // Ctrl-S
77      newtio.c_cc[VSUSP]    = 0;     // Ctrl-Z
78      newtio.c_cc[VEOL]     = 0;     // CR
79      newtio.c_cc[VREPRINT] = 0;     // Ctrl-R
80      newtio.c_cc[VDISCARD] = 0;     // Ctrl-U
81      newtio.c_cc[VWERASE]  = 0;     // Ctrl-W
82      newtio.c_cc[VLNEXT]   = 0;     // Ctrl-V
83      newtio.c_cc[VEOL2]    = 0;     // LF
84
85      /* 'Limpa' a linha e ativa as configuracoes para a porta */
86      tcflush (fd, TCIFLUSH);
87      tcsetattr (fd, TCSANOW, &newtio);
88
89      /* reseta o dispositivo (modem) */
90      if (write (fd, "ATZ\r", 4) < 4)
91          perror ("write");
92
93      while (1) {
94          res = read (fd, buf, 255);
95          if (res < 0) {
96              perror ("read");
97              exit (1);
98          }
99
100         /* Seta final de string, para poder imprimir o conteudo com printf */
101         buf[res] = 0;
102         fprintf (stdout, "[%s] (%d)\n", buf, res);
103     }
104
105     /* Restaura as configuracoes antigas da porta */
```

```
106     tcsetattr (fd, TCSANOW, &oldtio);
107     close (fd);
108     exit (0);
109 }
```

Nesse exemplo, vale notar que alguns caracteres de controle são inicializados com 0 mesmo após a estrutura ter sido preenchida com 0's através da função `memset(3)`. Isto foi feito apenas para mostrar no código a possibilidade de trocar os caracteres de controle. Os valores default estão comentados à direita de cada um.

As funções `cfsetispeed(3)` e `cfsetospeed(3)` permitem que seja utilizada uma taxa de *baud rate* diferente para a entrada e a saída de dados. Neste exemplo foi utilizada a mesma taxa, o que seria equivalente a passar a constante `BAUDRATE` juntamente na variável *newtio.c_cflag* no Linux.

10.2.3 Modo não-canônico (raw)

A diferença básica entre o modo canônico e o não-canônico, no que diz respeito ao código, é o não-mapeamento do *carriage return* para o caracter de nova linha (pois não estamos trabalhando baseados em linhas, mas sim em caracteres) e da especificação da variável *newtio.c_lflag* como 0, habilitando a entrada em modo não-canônico.

O exemplo abaixo mostra como trabalhar neste modo. Ele configura um timeout entre leituras de caracteres (após o sinal de nova linha) de 0.5 segundos, além de aguardar a leitura de pelo menos 5 caracteres.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <termios.h>
9
10 /* Configuracoes de baudrate sao definidas em <asm/termbits.h> */
11 #define BAUDRATE B9600
12 #define DEVICE "/dev/ttyS0"
13
14 int
15 main (int argc, char **argv)
16 {
17     int fd, res;
18     struct termios oldtio, newtio;
19     char buf[255];
20
21     fd = open (DEVICE, O_RDWR | O_NOCTTY);
22     if (fd < 0) {
23         perror ("open");
24         exit (1);
25     }
26
27     /* Salva a configuracao atual da porta serial e inicia uma nova */
28     tcgetattr (fd, &oldtio);
29     memset (&newtio, 0, sizeof (newtio));
30
31     /*
```

```
32        * BAUDRATE: Configura o bps rate.
33        * CRTSCTS : Controle de fluxo de saida do hardware
34        * CS8      : 8n1 (8 bit , no parity , 1 stopbit )
35        * CLOCAL : Conexao local , sem controle do modem
36        * CREAD : Habilita a recepcao de caracteres
37        */
38      newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
39      newtio.c_iflag = IGNPAR;
40
41      /* Saida em modo 'raw' */
42      newtio.c_oflag = 0;
43
44      /* habilita entrada em modo nao-canonico */
45      newtio.c_lflag = 0;
46
47      newtio.c_cc[VTIME]    = 5;   // timeout entre leituras ( t=VTIME * 0.1 segs)
48      newtio.c_cc[VMIN]     = 5;   // bloqueia leitura ate' ler 5 caracteres
49
50      /* 'Limpa' a linha e ativa as configuracoes para a porta */
51       tcflush ( fd , TCIFLUSH);
52       tcsetattr ( fd , TCSANOW, &newtio);
53
54      /* reseta o dispositivo ( modem) */
55      if ( write ( fd , "ATZ\r", 4) < 4)
56            perror ("write");
57
58      while (1) {
59            res = read ( fd , buf , 255);
60            if ( res < 0) {
61                  perror ("read");
62                  exit (1);
63            }
64
65            /* Seta final de string , para poder imprimir o conteudo com printf */
66            buf[ res ] = 0;
67             fprintf ( stdout , "[%s] (%d)\n", buf , res );
68      }
69
70      /* Restaura as configuracoes antigas da porta */
71       tcsetattr ( fd , TCSANOW, &oldtio);
72      close ( fd );
73      exit (0);
74  }
```

10.2.4 Modo assíncrono

Este modo é utilizado quando não deseja-se ter uma chamada `read(2)` bloqueante. Nesse caso, configura-se o `select(2)` para monitorar o descritor da porta serial, avisando-nos assim que ela tiver algum dado para ser lido. Além disso, um tratador para o sinal SIGIO é criado; dessa forma, assim que o `select(2)` nos informar que existem dados disponíveis para a leitura, o tratador do sinal será chamado, e poderá preparar quaisquer buffers e opções antes de efetuar a leitura de fato.

Abaixo segue um exemplo de código para usar o modo assíncrono para a leitura da porta

serial.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <termios.h>
9  #include <signal.h>
10
11 /* Configuracoes de baudrate sao definidas em <asm/termbits.h> */
12 #define BAUDRATE B9600
13 #define DEVICE "/dev/ttyS0"
14
15 void
16  trata_sinal (int signum)
17 {
18      printf ("agora posso ler da porta\n");
19 }
20
21 int
22 main (int argc, char **argv)
23 {
24      int fd, res;
25      struct termios oldtio, newtio;
26      char buf[255];
27      fd_set readfds;
28
29      fd = open (DEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
30      if (fd < 0) {
31          perror ("open");
32          exit (1);
33      }
34
35      /* Instala tratador de sinal para SIGIO */
36      signal (SIGIO, trata_sinal);
37
38      /* configura o descritor para operacoes assincronas */
39      fcntl (fd, F_SETFL, FASYNC);
40
41      /* Salva a configuracao atual da porta serial e inicia uma nova */
42      tcgetattr (fd, &oldtio);
43      memset (&newtio, 0, sizeof (newtio));
44
45      /*
46       * BAUDRATE: Configura o bps rate.
47       * CRTSCTS : Controle de fluxo de saida do hardware
48       * CS8     : 8n1 (8 bit, no parity, 1 stopbit)
49       * CLOCAL : Conexao local, sem controle do modem
50       * CREAD  : Habilita a recepcao de caracteres
51       */
52      newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
```

```
53      newtio. c_iflag  = IGNPAR;
54
55      /* Saida em modo 'raw' */
56      newtio. c_oflag  = 0;
57
58      /* habilita entrada em modo nao-canonico */
59      newtio. c_lflag  = 0;
60
61      newtio.c_cc[VTIME]    = 5;      // timeout entre leituras ( t=VTIME * 0.1 segs)
62      newtio.c_cc[VMIN]     = 5;      // bloqueia leitura ate' ler 5 caracteres
63
64      /* 'Limpa' a linha e ativa as configuracoes para a porta */
65       tcflush  ( fd , TCIFLUSH);
66       tcsetattr  ( fd , TCSANOW, &newtio);
67
68      /* reseta o dispositivo ( modem) */
69      if ( write ( fd , "ATZ\r", 4) < 4)
70           perror ("write");
71
72      while (1) {
73           FD_SET (fd, &readfds );
74           select  ( fd + 1, & readfds , NULL, NULL, NULL);
75           if  (! FD_ISSET (fd, &readfds )) {
76                fprintf  ( stderr , "Erro em select ()!\n");
77                exit  (1);
78           }
79           res  = read ( fd , buf , 255);
80           if ( res  < 0) {
81                perror ("read");
82                exit  (1);
83           }
84
85           /* Seta final de string , para poder imprimir o conteudo com printf */
86           buf[ res ] = 0;
87           fprintf  ( stdout , "[%s] (%d)\n", buf , res );
88      }
89
90      /* Restaura as configuracoes antigas da porta */
91       tcsetattr  ( fd , TCSANOW, &oldtio);
92      close ( fd );
93      exit  (0);
94 }
```

10.3 USB

O acesso à dispositivos USB pelo Linux pode ser feito através da LibUSB [1]. Ela é uma biblioteca para ser usada por aplicações em espaço de usuário para acessar dispositivos USB 1.1 independente do sistema operacional.

A respeito de dispositivos e interfaces, a API da libusb associa um dispositivo aberto a uma interface em específico. Isso significa que caso deseje-se abrir múltiplas interfaces em um dispositivo, ele deverá ser aberto múltiplas vezes para receber um manipulador para cada interface.

[1] http://libusb.sourceforge.net/

As seguintes funções são usadas para preparar o uso da biblioteca para comunicar com algum dispositivo:

- *void usb_init(void)*: inicializa estruturas internas da biblioteca;
- *int usb_find_busses(void)*: scaneia todos os BUSses USB no sistema, retornando a quantidade de modificações desde a última chamada a esta função (o total de BUSses adicionados e removidos);
- *int usb_find_devices(void)*: scaneia todos os dispositivos em cada BUS. Essa função deve ser chamada logo após `usb_find_busses()`. O valor de retorno indica a quantidade de dispositivos removidos ou adicionados até a última chamada à função;
- *struct usb_bus *usb_get_busses(void)*: retorna a lista dos busses USB encontrados;

A *struct usb_bus* apresenta os seguintes campos:

```
struct usb_bus {
    struct usb_bus *next, *prev;
    char dirname[PATH_MAX + 1];
    struct usb_device *devices;
};
```

O campo *dirname* é relativo ao diretório na entrada USB do `procfs` – normalmente localizado em /proc/bus/usb. Para cada BUS, um sub-diretório é criado neste diretório, sendo que *dirname* indica qual é o sub-diretório associado a um determinado BUS. Esta informação não chega a ser útil enquanto lida-se com o dispositivo pela libusb. Os campos *next* e *prev* são usados para iterar na lista duplamente encadeada que é retornada pela função *usb_get_busses()*.

Já a *struct usb_device* apresenta informações sobre os dispositivos encontrados. Ela é descrita como segue:

```
struct usb_device {
    struct usb_device *next, *prev;
    char filename[PATH_MAX + 1];
    struct usb_bus *bus;
    struct usb_device_descriptor descriptor;
    struct usb_config_descriptor *config;
    void *dev;            /* Darwin support */
};
```

Agora, o campo *filename* informa qual o arquivo dentro do diretório informado na *struct usb_bus* está associado ao dispositivo atual. Destes campos, o que será útil para as aplicações será o *descriptor*. Ele nos dá as seguintes características sobre o dispositivo:

```
struct usb_device_descriptor {
    u_int8_t  bLength;
    u_int8_t  bDescriptorType;
    u_int16_t bcdUSB;
    u_int8_t  bDeviceClass;
    u_int8_t  bDeviceSubClass;
    u_int8_t  bDeviceProtocol;
    u_int8_t  bMaxPacketSize0;
    u_int16_t idVendor;
    u_int16_t idProduct;
    u_int16_t bcdDevice;
    u_int8_t  iManufacturer;
    u_int8_t  iProduct;
    u_int8_t  iSerialNumber;
    u_int8_t  bNumConfigurations;
};
```

A partir destas informações, é possível obter o identificador do dispositivo que deseja-se trabalhar. Feito isso, as seguintes funções são usadas:

- *usb_dev_handle *usb_open(struct *usb_device dev)*: abre o dispositivo *dev* para uso. Esta função deve ser chamada antes de realizar qualquer operação sobre o dispositivo. Ela retorna um handle para comunicar-se com ele;
- *int usb_close(usb_dev_handle *dev)*: fecha um dispositivo aberto com *usb_open()*. Retorna 0 ou um valor < 0 caso ocorra um erro;
- *int usb_claim_interface(usb_dev_handle *dev, int interface)*: reivindica a interface para o sistema operacional. O parâmetro *interface* é o valor especificado no campo *bInterfaceNumber* do descritor. Este valor pode ser recuperado a partir de *config→interface→altsetting*, presente na *struct usb_device*. Essa função deve ser chamada antes de realizar qualquer operação relacionada à interface requisitada (como *usb_set_altinferface()* ou *usb_bulk_write()*;
- *int usb_release_interface(usb_dev_handle *dev, int interface)*: libera uma interface previamente requisitada pela *usb_claim_interface()*;
- *int usb_set_configuration(usb_dev_handle *dev, int configuration)*: escolhe a configuração ativa de um dispositivo. O parâmetro *configuration* é o valor especificado no campo bConfigurationValue do descritor;
- *int usb_set_altinterface(usb_dev_handle *dev, int alternate)*: escolhe a configuração ativa alternada da interface atual. O parâmetro *alternate* é o valor especificado no campo bAlternateSetting do descritor;
- *int usb_reset(usb_dev_handle *dev)*: reseta o dispositivo especificado, enviando um RESET para a porta que ele estiver conectado. Vale notar que após chamar essa função, o dispositivo precisará ser renumerado, sendo que o handle usado para chamar a *usb_reset()* não funcionará mais. Para tornar a usar o dispositivo, é necessário procurá-lo novamente e abrir um novo handle;
- *int usb_resetep(usb_dev_handle *dev, unsigned int ep)*: reseta todos os estados para o endpoint especificado em *ep*. O valor *ep* é o encontrado no campo bEndpointAddress do descritor;
- *int usb_clear_halt(usb_dev_handle *dev, unsigned int ep)*: limpa qualquer *halt status* do endpoint especificado por *ep*.

Para realizar transferência de dados sobre *bulk pipes*, podem ser usadas as seguintes funções:

- *int usb_bulk_write(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout)*: realiza uma requisição de escrita em volume (*bulk write*) para o endpoint *ep*, escrevendo *size* bytes do array *bytes* para o dispositivo *dev*. O timeout deve ser especificado em milisegundos. Essa função rtorna o número de bytes escritos ou um valor < 0 caso ocorra um erro;
- *int usb_bulk_read(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout)*: realiza uma requisição de leitura em volume (*bulk read*) para o endpoint *ep*. Retorna o número de bytes lidos ou um valor < 0 caso ocorra um erro.

A biblioteca permite também realizar transferência de dados sobre *interrupt pipes*:

- *int usb_interrupt_write(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout)*: realiza uma requisição de escrita interruptível para o endpoint *ep*. Retorna o número de bytes escritos com sucesso ou um valor < 0 caso haja um erro;
- *int usb_interrupt_read(usb_dev_handle *dev, int ep, char *bytes, int size, int timeout)*: realiza uma requisição de leitura interruptível para o endpoint *ep*. Retorna o número de bytes lidos com sucesso ou um valor < 0 caso haja um erro;

Algumas outras funções são providas pela biblioteca, com opções para obter strings de identificação para dispositivos e realizar transferências de controle. São elas:

- *int usb_control_msg(usb_dev_handle *dev, int requesttype, int request, int value, int index, char *bytes, int size, int timeout)*: realiza uma requisição de controle para o pipe padrão de controle em um dispositivo. Os parâmetros são os de mesmo nome na especificação USB. Retorna o número de bytes escritos/lidos ou um valor < 0 caso ocorra um erro;
- *int usb_get_string(usb_dev_handle *dev, int index, int langid, char *buf, size_t buflen)*: obtém o descritor de string especificado pelos parâmetros *index* e *langid* de um dispositivo. A string será retornada em formato Unicode, de acordo com a especificação USB. O valor de retorno é o número de bytes em *buf* ou < 0 caso haja um erro;
- *int usb_get_string_simple(usb_dev_handle *dev, int index, char *buf, size_t buflen)*: esta função é um wrapper para a *usb_get_string()* que retorna a string de descrição especificada pelo *index* na primeira língua para o descritor, convertendo-o para ASCII. Essa função retorna o número de bytes em *buf* ou um valor < 0 se houver um erro;
- *int usb_get_descriptor(usb_dev_handle *dev, unsigned char type, unsigned char index, void *buf, int size)*: obtém o descritor para o dispositivo identificado por *type* e *index* do descritor a partir do pipe de controle padrão. Retorna o número de bytes lidos para o descritor ou um valor < 0 em caso de erro;
- *int usb_get_descriptor_by_endpoint(usb_dev_handle *dev, int ep, unsigned char type, unsigned char index, void *buf, int size)*: idem à função *usb_get_descriptor()*, porém o obtém a partir do pipe de controle identificado por *ep*;
- *int usb_get_driver_np(usb_dev_handle *dev, int interface, char *name, int namelen)*: essa função irá obter o nome do driver associado à interface especificada pelo parâmetro *interface* e irá armazená-lo no buffer *name* de tamanho *namelen*. Essa função está implementada apenas para o Linux;
- *int usb_detach_kernel_driver_np(usb_dev_handle *dev, int interface)*: desconecta o driver de kernel da interface especificada pelo parâmetro *interface*. Aplicações usando a libusb podem então realizar a *usb_claim_interface()*. Ela está implementada apenas no Linux.

O conjunto de funções da API permitem que se manipule de forma bastante abstrata os dispositivos USB. O código abaixo busca todos os dispositivos e, para cada um, obtém as suas strings de informação e as imprime na saída padrão. Durante a procura por dispositivos, é buscado um em especial (como pode ser visto nas linhas 45 e 46), e este dispositivo será usado depois para realizar a escrita de um comando e a leitura de uma resposta. No entanto, vale lembrar que como os comandos (e os identificadores do dispositivo) são específicos, o código precisa ser adaptado para lidar com algum outro dispositivo. Para compilar este exemplo, basta rodar:

```
$ gcc teste_usb.c -o teste_usb -Wall `libusb-config --cflags --libs`
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <usb.h>
4
5  void show_info (usb_dev_handle *handle) {
6      int i;
7      char buffer[1024];
8
9      /* mostra ate 3 linhas de informacao sobre o dispositivo */
10     for (i = 0; i < 3; i++) {
11         int len = usb_get_string_simple (handle, i, buffer, 1024);
12         if (len < 0)
13             break;
14
15         buffer[len] = '\0';
```

```
16              printf ("%s\n", buffer );
17          }
18          printf ("\n");
19      }
20
21      int
22      main (int argc , char **argv)
23      {
24          struct usb_bus *busses;
25          struct usb_bus *bus;
26          struct usb_device *dev;
27          struct usb_device *gp32 = NULL;
28          usb_dev_handle *handle;
29          char buffer [1024];
30          int result ;
31
32          /* Inicializa a biblioteca e procura os devices */
33          usb_init ();
34          usb_find_busses ();
35          usb_find_devices ();
36
37          busses=usb_get_busses ();
38          for (bus=busses; bus; bus=bus->next) {
39              for (dev=bus->devices; dev; dev=dev->next) {
40
41                  handle = usb_open (dev);
42                  show_info (handle);
43                  usb_close (handle);
44
45                  if ((dev->descriptor.idVendor == 0xeb6) &&
46                      (dev->descriptor.idProduct == 0x3232)) {
47                      printf ("Achei dispositivo que procurava\n");
48                      gp32 = dev;
49                  }
50              }
51          }
52
53          if (gp32 == NULL) {
54              fprintf ( stderr , "Nao encontrei o dispositivo na USB\n");
55              exit (1);
56          }
57
58          /* Abre a interface representada pelo dispositivo 'gp32' */
59          handle = usb_open (gp32);
60          result = usb_claim_interface (handle, 0x0);
61          if ( result < 0) {
62              fprintf ( stderr , "%s\n", usb_strerror ());
63              exit (1);
64          }
65
66          /*
67           * Estamos prontos para fazer leituras e escritas ao device.
68           * Note que os dados a serem escritos ão especificos de cada dispositivo ,
69           * e devem ser consultados na especificacao dos mesmos. Neste exemplo,
```

```
70        * estao sendo usados os endpoints 0x03 para escrita e 0x81 para leitura .
71        */
72       memset(buffer, 0, sizeof(buffer));
73        snprintf(buffer, sizeof(buffer), "comandos especificos");
74
75        result = usb_bulk_write(handle, 0x03, buffer, sizeof(buffer), 100000);
76        if (result < 0) {
77              fprintf(stderr, "%s\n", usb_strerror());
78              exit(1);
79        }
80
81       /* le a resposta */
82       memset(buffer, 0, sizeof(buffer));
83        result = usb_bulk_read(handle, 0x81, buffer, sizeof(buffer), 100000);
84        if (result < 0) {
85              fprintf(stderr, "%s\n", usb_strerror());
86              exit(1);
87        }
88
89       /* libera a interface e encerra o programa */
90         usb_release_interface(handle, 0x0);
91        usb_close(handle);
92
93        exit(0);
94  }
```

Capítulo 11

Timers

11.1 Sinais

Timers são recursos que permitem sinalizar o processo em intervalos de tempo fixos. Como eles são baseados em sinais, esta sessão os apresenta, permitindo fazer um programa que lide com sinais diversos.

A chamada de sistema usada para capturar sinais é a `signal(2)`. Esta chamada de sistema instala um novo *signal handler* para o sinal com o número *signum*. O *signal handler* é configurado para *sighandler*, que pode ser qualquer função escrita pelo usuário, ou SIG_IGN para ignorá-lo ou SIG_DFL, para que a ação padrão seja tomada (descrita na *man page* do `signal(7)`.

O seu protótipo é o seguinte:

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

Alguns dos sinais mais comuns de serem tratados são:

- *SIGINT*: interrupção do teclado (CTRL+C);
- *SIGKILL*: processo recebeu um sinal de kill;
- *SIGTERM*: processo recebeu sinal de término.

Seu uso é bastante simples, como pode ser visto no exemplo abaixo. O programa declara um handler para quando receber o sinal *SIGINT*, e aguarda até que seja pressionada alguma tecla para encerrar normalmente, ou através da recepção do sinal especificado.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4
5  void
6  suicide (int signum)
7  {
8      printf ("Recebi o sinal %d, encerrando..\n", signum);
9      exit (1);
10 }
11
```

```
12 int
13 main (int argc , char **argv)
14 {
15       signal (SIGINT, suicide );
16       getchar ();
17       printf ("Encerrando normalmente\n");
18       exit (0);
19 }
```

11.2 Utilizando timers

O sistema provê a cada processo três timers, cada um decrementando em um domínio de tempo diferente. Quando um timer expira, um sinal é enviado para o processo, e o timer potencialmente reinicia. Os timers disponíveis são:

- *ITIMER_REAL*: decrementa em tempo real, e entrega o sinal SIGALRM quando expira;
- *ITIMER_VIRTUAL*: decrementa apenas enquanto o processo está executando, e entrega o sinal SIGVTALARM quando expira;
- *ITIMER_PROF*: decrementa quando o processo executa e quando o sistema está executando no contexto do processo. Este método costuma ser usado para fazer profile do tempo que um processo passou executando tanto em espaço de usuário quanto de kernel. O sinal SIGPROF é entreque quando este timer expira.

O protótipo para uso de timers é o seguinte:

```
#include <sys/time.h>
int getitimer (int which, struct itimerval *value );
int setitimer (int which, const struct itimerval *value , struct itimerval *ovalue );
```

Os valores que são usados são definidos pelas estruturas:

```
struct itimerval {
    struct timeval it_interval ; /* proximo valor */
    struct timeval it_value ;    /* valor atual */
};
struct timeval {
    long tv_sec ;                /* segundos */
    long tv_usec ;               /* micro segundos */
};
```

A função `getitimer(2)` preenche a estrutura indicada por *value* com as configurações do timer indicado por *wich*, que pode ser uma das ITIMER_REAL, ITIMER_VIRTUAL ou ITIMER_PROF. O elemento *it_value* é configurado para a quantidade de tempo restante no timer, ou zero caso o timer seja desabilitado. Similarmente, *it_interval* é configurado para o valor de reset.

A função `setitimer(2)` configura o timer indicado para o valor em *timer*. Se *ovalue* é não-zero, o valor antigo do timer é armazenado ali.

Timers decrementam de *it_value* até zero, gerando um sinal, e resetam para *it_interval*. Um timer que é configurado para um *it_value* zero expira.

Como exemplo de uso, podemos tomar o programa abaixo, que executa um timer de 2 em 2 segundos.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <sys/time.h>
5
6  void
7  timer_func (int signum)
8  {
9       printf ("recebi o sinal %d\n", signum);
10 }
11
12 int
13 main (int argc, char **argv)
14 {
15      struct itimerval tempo;
16
17      /* como vamos usar ITIMER_REAL, preparamos um signal handler para SIGALRM */
18      signal (SIGALRM, timer_func);
19
20      /* especifica o intervalo e seta o timer */
21      tempo.it_interval.tv_sec = 2;
22      tempo.it_interval.tv_usec = 0;
23      tempo.it_value.tv_sec = 2;
24      tempo.it_value.tv_usec = 0;
25      setitimer (ITIMER_REAL, &tempo, NULL);
26
27      while (1) {
28           usleep (100);
29      }
30      exit (0);
31 }
```

Bibliografia

[Ame 94] American National Standards Institute. **IEEE standard for information technology**: Portable Operating Sytem Interface (POSIX). part 1, system application program interface (API) — amendment 1 — realtime extension [C language]. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1994. xxiii + 590p. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[GIL 2003] GILMORE, J. **GDB Internals**. [S.l.]: Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA, 2003.

[KOJ 2000] KOJIMA, A. K.; ANDRADE, P. C. P. de; SANTOS, C. A. M. dos. **Desenvolvimento de Aplicações Gráficas para o X Window System**. Mini-curso apresentado no Software Livre 2000.

[MyS 2003] MySQL AB. **MySQL Reference Manual**. http://www.mysql.com/documentation/.

[RUB 2001] RUBINI, A.; CORBET, J. (Eds.). **LINUX device drivers**. [S.l.]: O'REILLY, 2001. v.2.

[SCH 86] SCHEIFLER, R.; GETTYS, J. The X Window System. **ACM Transactions on Graphics**, v.5, n.2, p.79–109, 1986.

[STA 2002] STALLMAN, R. M. **Using the GNU Compiler Collection (GCC)**. [S.l.]: Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA, 2002.

[STE 98] STEVENS, W. R. (Ed.). **UNIX Network Programming - Interprocess Communication**. [S.l.]: Prentice Hall, 1998.

[STE 98a] STEVENS, W. R. (Ed.). **UNIX Network Programming - Networking APIs:** Sockets and XTI. [S.l.]: Prentice Hall, 1998.

[The 2003] The Native POSIX Thread Library for Linux, 2003, http://people.redhat.com/drepper/nptl-design.pdf. **Anais...** [S.l.: s.n.], 2003.

[ZEL 2000] ZELLER, A. **Debugging with DDD**. http://www.gnu.org/manual/ddd/.