Introdução ao LINUX

e

Programação em Script-Shell

Programa de Educação Tutorial

Telecomunicações

PET TELE)))

Universidade Federal Fluminense

Niterói-RJ

2004

Prefácio

O Programa de Educação Tutorial (PET) possui como objetivo maior realizar atividades de pesquisa, ensino e extensão de forma não dissociada, isto é, desenvolver projetos que integrem em si estes três aspectos. Neste contexto, o grupo PET-Tele desenvolve diversas apostilas, não como um fim em si mesmas, mas como forma de realizar também outras atividades.

O desenvolvimento dessas apostilas é iniciado com um trabalho de pesquisa sobre o tema escolhido. Após este período inicial, produz-se material didático sobre o assunto, visando transmitir o conhecimento adquirido para os alunos do curso de graduação e outros interessados. Eventualmente, quando ocorre alguma solicitação, os alunos do grupo fornecem cursos sobre os assuntos abordados na apostila.

Até o momento, as seguintes apostilas estão disponíveis no site do PET:

HTML Linguagem de programação para hipertextos, principalmente empregada na construção de páginas da Internet (*webpages*).

LaTeX Sistema de edição de texto largamente utilizado em meios acadêmicos e científicos, bem como por algumas editoras nacionais e internacionais.

LINUX e *Script-Shell* Introdução ao sistema operacional GNU/Linux e programação em utilizando o *shell*.

MATLAB Ambiente de simulação matemática, utilizado em diversas áreas profissionais.

SPICE Ambiente de simulação de circuitos elétricos (analógicos e digitais), utilizado em projeto de circuitos discretos e integrados.

Nota desta Apostila

Esta apostila visa introduzir o usuário ao ambiente do sistema operacional Linux da família UNIX-Like, explicando seu sistema de arquivos, seus processos característicos e seus comandos. Além disso, tem por objetivo apresentar programação *Script-Shell*, ensinando a criar e executar programas, apresentando os comandos mais utilizados na construção de *scripts*, além de manipulação de variáveis, uso de estruturas básicas de decisão e controle e exemplos diversos.

A apostila está dividida em 3 partes:

O conteúdo da primeira parte abrange um breve histórico, alguns conceitos e características dos Sistemas Operacionais, *hardware* e *software* havendo maior destaque para o Sistema Operacional UNIX.

A segunda contém os aspectos básicos do Linux, assim como os principais comandos utilizados para realizar tarefas no Linux.

A última parte da apostila envolve os principais conceitos de programação utilizando o *shell* do Linux.

O principal motivo da abordagem desses 3 temas em uma única apostila foi o de levar o usuário iniciante a ter uma visão mais ampla do funcionamento de um sistema operacional quando um comando é digitado na tela de um terminal ou quando executado um *script*.

Não se pretende com este manual apresentar todos os comandos do Linux e suas aplicações mais usadas. Cada usuário usará mais um conjunto de comandos do que outros dependendo de seus objetivos. Existem comandos específicos para redes, para manipulação de arquivos, para configuração de dispositivos, etc. Um livro seria pouco para esgotar todas as facilidades e funções que o Linux oferece.

Espera-se que com essa apostila o usuário iniciante conheça o básico do Linux de forma clara e o usuário mais experiente a use como referência.

Quanto ao aspecto de programação, vale lembrar que um usuário não se tornará um grande programador apenas lendo uma apostila ou um livro. É preciso prática, fazer programas, testar funções, estudar programas prontos para entender seu funcionamento, etc. Uma grande vantagem do Linux é que suas distribuições vêm com os *scripts* abertos, para que o usuário possa ler, entender e adaptar às suas necessidades, podendo até mesmo fazer uma nova distribuição com base na outra. Bastando para isso, vontade de aprender e curiosidade. As *man pages* do Unix e dos interpretadores de comandos como o *bash* possuem tudo o que for necessário para entender os comandos de uma forma mais completa.

No fim da apostila, se encontra um guia de sites, livros e apostilas usados como base para a construção desta. E outras referências importantes contendo um amplo e importante conteúdo para o leitor usar afim de aprimorar seu conhecimento no Linux e programação utilizando o *shell*.

Sumário

Capítulo 1

Conceitos Gerais

1.1 Introdução ao Hardware

1.1.1 Resolução de problemas X Mapeamento de domínios

Como já foi visto em outras disciplinas, é possível descrever situações através de equações. Por exemplo, na Física, temos as leis da natureza mapeadas por equações, permitindo que antecipemos os resultados. Conhecendo a equação que descreve o problema, que pode estar em um domínio complicado, podemos transportá-la para outro domínio mais simples (que já pode ter uma solução conhecida), onde obtemos resultados que podem ser levados de volta ao domínio inicial. A resolução do problema pode ser definida em quatro etapas:

1. Mapeamento do problema real em um problema matemático;
2. Equacionamento do problema matemático;
3. Mapeamento do equacionamento em um algoritmo;
4. Adequação do algoritmo a uma linguagem de programação.

1.1.2 Funções lógicas e aritméticas básicas

É possível a representação de todo um sistema numérico utilizando apenas "zeros"e "uns". Chamamos esse sistema de sistema binário e foi desenvolvida com ele uma lógica, a lógica BOOLEANA. Através dela, podemos implementar soluções num domínio de fácil manipulação, uma vez que podemos considerar, por exemplo, uma lâmpada acesa como nível lógico '1' e uma apagada como '0'. Do mesmo modo, podemos trabalhar com tensão elétrica:

com tensão - '1';
sem tensão - '0'.

Tabela das funções lógicas:

A	B	AND	OR	XOR	NAND	NOR	XNOR	NOT A
0	0	0	0	0	1	1	1	1
0	1	0	1	1	1	0	0	1
1	0	0	1	1	1	0	0	0
1	1	1	1	0	0	0	1	0

Tabela 1.1: Tabela de Funções Lógicas

1.1.3 Implementação de funções matemáticas por circuitos

Uma vez conhecida a Lógica Booleana podemos, através da combinação de OR's, AND's, etc, construir circuitos capazes de somar, multiplicar, subtrair, enfim, realizar toda e qualquer operação matemática.

1.1.4 Computador X Máquina de níveis

Um computador é uma máquina digital capaz de solucionar problemas através de instruções que lhe são fornecidas. Cada computador tem sua linguagem de máquina, que consiste em todas as instruções primitivas que a máquina pode executar. Na verdade, pode ser dito que uma máquina define uma linguagem e uma linguagem define uma máquina.

Essas instruções mais básicas do computador geralmente se resumem a adicionar, verificar se um número é zero ou mover um dado. E estas são realizadas pelo circuito eletrônico. Seria complicado para um humano escrever um programa com base nestas simples instruções. Uma solução para esse problema é escrever o programa em uma linguagem (que chamaremos de `L2`) diferente da linguagem de máquina (`L1`). Então a execução de um programa escrito em `L2` deverá acontecer em duas etapas. Primeiro `L2` é transformada [1] em `L1`. Em seguida `L1` é executada diretamente. Assim, poderíamos pensar na existência de uma **máquina virtual** cuja linguagem de máquina é `L2`. Estentendo esse exemplo, poderiam ser criadas outras linguagens `L3`, `L4`, `L5`, etc..., até atingir um nível de linguagem conveniente ao ser humano. Como cada linguagem utiliza sua anterior como base, podemos pensar no computador como sendo composto por várias camadas ou níveis, uma **máquina multinível**.

Em grande parte dos computadores atuais há cerca de 7 níveis. A seguir há uma breve apresentação de cada um desses níveis. Veja Fig. 1.1.

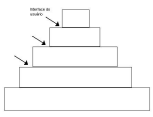

Figura 1.1: Esquema de níveis de um computador.

- Nível 0 - É o *hardware* da máquina. Consiste nas portas lógicas formando os circuitos.
- Nível 1 - É o nível onde estão os **microprogramas**. É o verdadeiro nível de linguagem de máquina. Esses microprogramas interpretam as instruções do nível 2, para o nível 0 executar.

[1] Entende-se por transformação de uma linguagem em outra como tradução ou interpretação.

- Nível 2 - Nível de máquina convencional. Contém o conjunto de instruções que serão interpretados pelo microprograma.
- Nível 3 - É um nível híbrido pois possui instruções do nível 2 e outras características novas. É chamado de **sistema operacional**.
- Nível 4 - Nível de montagem.
- Nível 5 - Contém as linguagens usadas pelos programadores para resolver programas.
- Nível 6 - Consiste nos programas feitos para trabalhar em determinada aplicação. Ex: editores de texto, simuladores matemáticos, elétricos, etc.

1.2 Introdução ao *software*

O *software* de um computador pode ser dividido, basicamente, em duas categorias:

- Programas de sistema - Gerenciam a operação do computador. O mais importante destes é o **Sistema Operacional**.
- Programas de aplicação - São aqueles que o usuário trabalha usualmente para resolver determinados problemas como editores de imagem, texto, jogos, etc.

1.2.1 Computador X Sistema Operacional

O Sistema Operacional controla todos os recursos do computador e fornece a base sobre a qual os programas e aplicativos são escritos. Ele é a interface do usuário e seus programas com o computador e responsável pelo gerenciamento de recursos e periféricos (como memória, discos, arquivos, impressoras, CD-ROMs, etc.) e a execução de programas.

A parte mais baixa desta interface com o *hardware* é considerada o ***Kernel*** do sistema operacional, traduzindo: Cerne do SO. É no *kernel* que estão definidas funções para operação com periféricos (mouse, disco, impressora, interfaces serial/paralela), gerenciamento de memória, entre outros.

O *kernel* é a parte mais importante do sistema operacional, pois, sem ele, a cada programa novo que se criasse seria necessário que o programador se preocupasse em escrever as funções de entrada/saída, de impressão, entre outras, em baixo nível. Por isso, quando há periféricos que o kernel não tem função pronta, é necessário escrever a interface para eles.

1.2.2 Interface de comunicação com o Sistema Operacional (SO)

O SO pode ser visto como uma máquina estendida, quando apresenta ao usuário uma máquina virtual equivalente ao hardware, porém muito mais simples de programar. Ele isola o usuário dos detalhes de operação do disco.

Ele também pode ser visto como gerente de recursos ao passo que ele gerencia os usuários de cada um dos recursos da máquina, combinando o tempo de uso de cada um e garante o acesso ordenado de usuários a recursos através da mediação dos conflitos entre as requisições dos diversos processos usuários do sistema. Por exemplo, o controle que o sistema operacional faz com os pedidos de diferentes usuários, em uma rede, para utilizar a impressora.

1.2.3 Interpretador de comandos (*shell*) de um SO

No caso do SO UNIX, o interpretador de comandos é chamado *Shell* e exerce a função de um programa que conecta e interpreta os comandos digitados por um usuário. É a interface que o usuário utiliza para enviar comandos para o sistema.

Dos vários programas *Shell* existentes, o *Bourne Shell*, o *Korn Shell* e o *C Shell* se destacam por serem os mais utilizados e conhecidos. Mas qualquer programador pode fazer o seu *Shell*. Estes *shells* tornaram-se conhecidos pois já vinham com o sistema, exceto o *Korn* que tinha que ser adquirido separadamente. O *Bourne Shell* vinha com o *System V* e o *C Shell* com o *BSD*. O *Korn Shell* é uma melhoria do *Bourne Shell*. Há também o *Bash* (*Bourne Again Shell*). Mais informações sobre estes *shells* podem ser vistas no capítulo 4.

Os comandos podem ser enviados de duas maneiras para o interpretador: de forma interativa e não-interativa.

Interativa: os comandos são digitados no *prompt* de comando e passados ao interpretador de comandos um a um. Neste modo, o computador depende do usuário para executar uma tarefa ou próximo comando.

Não-interativa: são usados arquivos de comandos criados pelo usuário (*scripts*) para o computador executar os comandos na ordem encontrada no arquivo. Neste modo, o computador executa os comandos do arquivo um por um e dependendo do término do comando, o *script* pode checar qual será o próximo comando a ser executado. O capítulo 4 descreve a construção de *scripts*.

1.3 Família UNIX

1.3.1 Sistemas UNIX

O UNIX é um sistema operacional multitarefa, ou seja, permite a utilização do processador entre várias tarefas simultaneamente, multiusuário, disponível para diversos *hardwares*. Ele possui a capacidade de criar opções específicas para cada usuário, as quais são ativadas quando o usuário se loga ao computador.

As raízes do UNIX encontraram-se na necessidade, na década de 70, de um sistema multitarefa confiável e aplicável ao ambiente dominante na época, um *mainframe* (um grande computador central) e uma série de terminais ligados a ele. No meio da programação, costuma-se dizer que o UNIX foi projetado por programadores para programadores já que possui certas características desejadas por eles como o de ser um sistema simples com grande flexibilidade.

Histórico do UNIX

Abaixo temos um cronograma da criação do UNIX:

- Décadas de 40 e 50:

  Todos os computadores eram pessoais quanto a forma de utilização. O usuário reservava um horário para utilizá-lo.

- Década de 60:

  Através dos sistemas *Batch* , o usuário enviava ao centro de processamento de dados um *job* em cartões perfurados para que esse fosse processado pelo computador. Aproximadamente uma

hora depois da submissão, o usuário podia buscar os resultados de seu programa. Porém, se houvesse algum erro no programa escrito, o usuário só saberia horas depois, perdendo seu tempo.

Para resolver esse problema, foi criado um sistema de compartilhamento de tempo, o **CTSS**, no MIT, onde cada usuário tinha um terminal *on-line* à sua disposição.

Após o CTSS, os Laboratórios Bell com o MIT e a General Eletric começaram um programa grandioso de criar um novo sistema operacional que suportasse centenas de usuários simultaneamente em regime de compartilhamento de tempo (*timesharing*). Tal sistema foi denominado **MULTICS** (MULTiplexed Information and Computing Service), que seria multi-usuário, multitarefa e teria um sistema de arquivos hierárquico.

- Década de 70:

  A AT&T, controladora da Bell Labs, insatisfeita com o progresso do MULTICS cortou o projeto e alguns programadores da Bell que trabalharam no projeto, como Ken Thompson, implementaram a versão monousuário do MULTICS em linguagem de montagem em um minicomputador, o PDP-7. Brian Kernighan, outro programador da Bell, deu o nome do novo sistema de **UNICS** como deboche ao nome do sistema anterior. Mais tarde o nome foi mudado para o conhecido **UNIX**.

  Em seguinda, o UNIX foi escrito para máquinas do tipo PDP-11. E como era necessário reescrever o sistema toda vez que ele fosse transportado para outra máquina, Thompson reescreveu o UNIX em uma linguagem de alto nível desenvolvida por ele, a Linguagem **B**. Para corrigir algumas imperfeições, Denni Ritchie, que também trabalhava na Bell Labs, desenvolveu a Linguagem **C**. Juntos, em 1973, eles reescreveram o UNIX em C. Dessa forma, foi garantida a portabilidade.

  Em 1974, Thompson e Ritchie publicaram um artigo sobre o novo sistema operacional UNIX, o que gerou um grande entusiasmo no meio acadêmico. Como a AT&T era um monopólio controlado atuante na área das telecomunicações, não foi permitida sua entrada no ramo da computação. Assim, a Bell Labs não colocou objeções para licenciar o UNIX para as universidades e empresas.

  Em 1977 existiam cerca de 500 computadores com UNIX no mundo todo.

  Pelo fato do UNIX ser fornecido com seu código fonte completo, muitas pessoas passaram a estudá-lo e organizar seminários para trocas de informações, visando a eliminação de *Bugs* e inclusão de melhoramentos. A primeira versão padrão do UNIX foi denominda **Versão 6** pois estava descrita na sexta edição do Manual do Programador UNIX.

- Década de 80:

  Com a divisão da AT&T em várias companhias independentes, imposta pelo gorverno americano em 1984, foi possível a criação de uma subsidiária dela no ramo da computação. Então foi lançado pela AT&T a primeira versão comercial do UNIX, o **System III**, seguida pelo **System V** , sendo que cada versão era maior e mais complicada que a antecessora.

  Auxiliada pela DARPA (*Defense Advanced Research Projects Agency*), a Universidade de *Berkeley* passou a distribuir uma versão melhorada da Versão 6, chamada **1BSD** (*First Berkeley Software Distribution*), seguida pela 2BSD, 3BSD e 4BSD. Esta última possuindo muitos melhoramentos, sendo o principal, o uso de memória virtual e da paginação, permitindo que programas

fossem maiores que a memória física. Também, a ligação de máquinas UNIX em rede, desenvolvida em Berkeley, fez com que o padrão de rede BSD, o TCP/IP, se tornasse padrão universal. Essas modificações fizeram com que fabricantes de computadores, como a Sun Microsystems e a DEC, baseassem suas versões no UNIX de Berkeley.

Em 1984 já existiam cerca de 100.000 computadores com UNIX rodando em diferentes plataformas.

No final da década, estavam circulando duas versões diferentes e incompatíveis, a 4.3BSD e o System V *Release* 3. Isso tornou praticamente impossível aos fornecedores de *software* ter a certeza de que seus programas pudessem rodar em qualquer sistema UNIX.

Então, o Comitê de Padronização do IEEE fez a primeira tentativa de unificação dos dois padrões. O nome escolhido do projeto foi **POSIX** (Portable Operating System). Foi criado após grandes discussões o padrão denominado **1003.1**, contendo a interseção das características tanto da 4.3 BSD quanto do System V. Porém, um consórcio, denominado **OSF** (*Open Software Foundation*), foi formado principalmente pela IBM (que comercializava o sistema **AIX**), DEC, Hewlett-Packard com o intuito de formar um sistema de acordo com o padrão 1003.1, além de características adicionais. Em reação, em 1988 a AT&T e Sun se unem no consórcio **UI** (UNIX International) para desenvolver Solaris e UNIXWare.

Com isso, passaram a existir, novamente, versões diferentes oferecidas pelos consórcios, tendo como característica comum o fato de serem grandes e complexas, contrariando a idéia principal do UNIX

Uma tentativa de fuga desses sistemas foi a criação de uma nova versão UNIX-*like* mais simples e menor, o sistema do tipo **MINIX**.

- Em 1997, foram vendidos cerca de 4 milhões de sistemas UNIX no mundo todo.

Alguns dos Sistemas Operacionais UNIX atuais são: BSD (FreeBSD, OpenBSD e NetBSD), Solaris (anteriormente conhecido por SunOS), IRIX, AIX, HP-UX, Tru64, Linux (nas suas milhares de distribuições), e até o Mac OS X (baseado num kernel Mach BSD chamado Darwin).

1.3.2 Interfaces do UNIX

Há 3 tipos de interface do sistema UNIX que podem ser identificadas:

- Interface de chamadas de sistema;
- Interface de biblioteca de procedimentos;
- Interface formada pelo conjunto de programas utilitários, considerada erroneamente por alguns usuários como sendo a verdadeira interface do UNIX.

Capítulo 2

Introdução ao Linux

Diferentemente do que se é levado a pensar, o Linux é uma implementação independente do sistema operacional UNIX. O Linux propriamente dito é um *kernel*, não um sistema operacional completo. Sistemas completos construídos em torno do *kernel* do Linux usam o sistema GNU que oferece um *shell*, utilitários, bibliotecas, compiladores e ferramentas, bem como outros programas como os editores de texto. Por essa razão, Richard M. Stallman, do projeto GNU, pede aos usuários que se refiram ao sistema completo como GNU/Linux. No apêndice, há mais informações sobre o projeto GNU.

O desenvolvimento do Linux iniciou a partir de um projeto pessoal de um estudante da Univeridade de Helsinki, na Finlândia, chamado Linus Torvalds. Ele pretendia criar um sistema operacional mais sofisticado do que o Minix, um UNIX relativamente simples cujo código fonte ele tinha disponível. O Linux obedece ao padrão estabelecido pelo governo norte americano, POSIX. O POSIX é o padrão da API (*Application Programming Interface*) UNIX, referências para desenvolvedores da família UNIX-*like*. Desde a apresentação do Linux em 5 de outubro de 1991, por Linus Torvalds, um grande número de pessoas envolvidas com programação começou a desenvolver o Linux. O nome Linux deriva da junção Linus + UNIX = Linux.

Portabilidade

Linux é hoje um dos *kernels* de sistema operacional mais portados, rodando em sistemas desde o iPaq (um computador portátil) até o IBM S/390 (um massivo e altamente custoso mainframe), embora este tipo de portabilidade não fosse um dos objetivos principais de Linus Torvalds. Seu esforço era tornar seu sistema portátil no sentido de ter habilidade de facilmente compilar aplicativos de uma variedade de fontes no seu sistema. Portanto, o Linux originalmente se tornou popular em parte devido ao esforço para que as fontes GPL ou outras favoritas de todos rodassem no Linux.

Distribuições

O sistema operacional completo (GNU/Linux) é considerado uma **Distribuição Linux**. É uma coleção de softwares livres (e às vezes não-livres) criados por indivíduos, grupos e organizações ao redor do mundo, e tendo o kernel como seu núcleo. Atualmente, companhias como a **Red Hat**, a **SuSE**,a **MandrakeSoft** ou a **Conectiva**, bem como projetos de comunidades com a **Debian** ou a **Gentoo**, compilam o *software* e fornecem um sistema completo, pronto para instalação e uso. Além disso, há projetos pessoais como o de Patrick Volkerding que fornece uma distribuição Linux, a **Slackware** e Carlos E. Morinoto que lançou a distribuição chamada **Kurumin**. Está última roda em CD, bastando as configurações do computador aceitarem o *boot* pelo *drive* do CD.

Logo que Linus Torvalds passou a disponibilizar o Linux, ele apenas disponibilizava o Kernel com alguns comandos básicos. O próprio usuário devia arrumar os outros programas, compilá-los e configurá-los. Para evitar esse trabalho, começou então a disponibilização de programas pré-compilados para o usuário apenas instalar. Foi assim que surgiu a MCC (Manchester Computer Centre), a primeira distribuição Linux, feita pela Universidade de Manchester. Algumas distribuições são maiores, outras menores, dependendo do número de aplicativos e sua finalidade. Algumas distribuições de tamanhos menores cabem em um disquete com 1,44 MB, outras precisam de vários CDs. Todas elas tem seu público e sua finalidade, as pequenas (que ocupam poucos disquetes) são usadas para recuperação de sistemas danificados ou em monitoramentos de redes de computadores. O que faz a diferença é como estão organizados e pré-configurados os aplicativos.

2.1 Aspectos básicos do Linux

2.1.1 *Startup* e *shutdown*

Correspondem respectivamente aos procedimentos de ligar e desligar o computador. O primeiro diz respeito basicamente ao fornecimento de energia para o funcionamento dos circuitos eletrônicos da máquina e não requer do usuário grandes precauções. Por outro lado o processo de *Shutdown* requer cuidado, uma vez que o desligamento inadequado do computador pode causar danos ao sistema operacional destruindo as tabelas internas de que ele necessita para funcionar. Quando o computador "trava", portanto, desligá-lo deve ser a última escolha. Uma opção é iniciar outra sessão, verificar onde está o problema e encerrar a sessão travada com um comando apropriado.

2.1.2 Abertura de seção no Linux

Para usar o Linux é preciso em primeiro lugar, que o usuário digite seu nome e sua senha, que são lidos e verificados pelo programa *login*. No UNIX um arquivo de senha é usado para guardar informações possuindo uma linha para cada usuário, contendo sua identificação alfabética e numérica, sua senha criptografada, seu diretório *home*, além de outras informações. Quando o usuário se identifica, o programa *login* criptografa a senha que acabou de ser lida do terminal e a compara com a senha do arquivo de senhas para dar permissão ao usuário. Esse arquivo com as informações dos usuários normalmente é encontrado no arquivo: `/etc/passwd`. As senhas criptografadas ficam no arquivo `/etc/shadow`.

2.1.3 Usuário, superusuário (*root*), grupos, acesso, proteção

O princípio da segurança do sistema de arquivo UNIX está baseado em usuários, grupos e outros usuários.

Usuário : é a pessoa que criou o arquivo. O dono do arquivo.

Grupo : é uma categoria que reúne vários usuários. Cada usuário pode fazer parte de um ou mais grupos, que permitem acesso a arquivos que pertencem ao grupo correspondente. Por padrão, o grupo de usuários inicial é o mesmo de seu nome de usuário. A identificação do grupo é chamada de **gid** (*group id*).

Outros : é a categoria de usuários que não se encaixam como donos ou grupos do arquivo.

As permissões de acesso para donos, grupos e outros usuários são independentes uma das outras, permitindo assim um nível de acesso diferenciado.

A conta ***root*** é também chamada de *super usuário*. Este é um login que não possui restrições de segurança. A conta *root* somente deve ser usada para fazer a administração do sistema. Utilize a conta de usuário normal ao invés da conta *root* para operar seu sistema. Uma razão para evitar usar privilégios *root* é devido à facilidade de se cometer danos irreparáveis ao sistema.

2.1.4 Sessão, *login*, *password*, *logout*

No Linux, a entrada no sistema é feita com um *login* e um *password*, onde *login* é o nome do usuário e *password* é uma senha de segurança. Uma vez efetuado o *login*, o usuário entra em sua conta e é aberta uma sessão onde se interage diretamente com o *Shell*. O *logout* (para finalizar a conta) é efetuado através do comando '`exit`'.

2.1.5 Consoles virtuais

Terminal (ou *console*) é o teclado e tela conectados em seu computador. O Linux faz uso de sua característica multi-usuário usando principalmente "terminais virtuais". Um terminal virtual é uma segunda sessão de trabalho completamente independente de outras que pode ser acessado no computador local ou remotamente via `telnet`, `rsh`, `rlogin`, etc.

No Linux, em modo texto, você pode acessar outros terminais virtuais segurando a tecla <ALT> e pressionando <F1> a <F6>. Cada tecla de função corresponde a um número de terminal do 1 ao 6 (o sétimo é usado por padrão pelo ambiente gráfico X). O Linux possui mais de 63 terminais virtuais, mas apenas 6 estão disponíveis inicialmente por motivos de economia de memória RAM. Se estiver usando o modo gráfico, você deve segurar <CTRL>+<ALT> enquanto pressiona uma tecla de <F1> a <F6>.

Um exemplo prático: se você estiver usando o sistema no Terminal 1 com um login qualquer e desejar entrar como *root* para instalar algum programa é só abrir um terminal virtual e realizar a tarefa desejada.

2.2 Sistema de arquivos

Arquivos são centrais para o UNIX de uma maneira não encontrada em outros sistemas operacionais. Comandos são arquivos executáveis, usualmente encontrados em locais previsíveis na árvore de diretórios. Privilégios do sistema e permissões são controlados em grande parte através de arquivos. Dispositivos I/O e arquivos I/O não são distinguidos nos níveis mais altos. Até mesmo a comunicação entre processos ocorre através de entidades similares a arquivos. Toda a segurança do sistema depende, em grande parte, da combinação entre a propriedade e proteções setadas em seus arquivos e suas contas de usuários.

2.2.1 Sistema hierárquico, árvore de diretórios, montagem de ramificações

O UNIX tem uma organização de diretórios hierárquica em formato conhecido como *filesystem*. A base desta árvore é um diretório chamado '*root directory*'. Em sistemas UNIX, todo espaço em disco disponível é combinado em uma única árvore de diretório abaixo do /, sendo que o local físico onde um arquivo reside não faz parte da especificação do UNIX.

2.2.2 Tipos básicos de arquivos: *plain files*, *directory*

Os arquivos são onde os dados estão gravados. Um arquivo pode conter um texto, uma música, programa, etc. Todo sistema UNIX reconhece pelo menos três tipos de arquivos:

- **Arquivos comuns**: Usados para armazenar dados. Os usuários podem acrescentar dados diretamente em arquivos comuns, como, por exemplo, através de um editor. Os programas executáveis também são guardados como arquivos comuns.
- **Arquivos de diretório**: Um arquivo de diretório contém uma lista de arquivos. Cada inserção na lista consiste em duas partes: o nome do arquivo e um ponteiro para o arquivo real em disco. Por outro lado, os diretórios se comportam exatamente como arquivos comuns, exceto pelo fato de que alguns comandos usados para manipulação de arquivos comuns não funcionarem para arquivos de diretório.
- **Arquivos especiais**: Estes arquivos são usados para fazer referência a dispositivos físicos como os terminais, as impressoras, os discos. Eles são lidos e gravados como arquivos comuns, mas tais associações causam a ativação do dispositivo físico ligado a ele.

O Linux é *case sensitive*, ou seja, ele diferencia letras maiúsculas e minúsculas nos arquivos. O diretório, como qualquer outro arquivo, também é *case sensitive*. Em um mesmo diretório, não podem existir dois arquivos com o mesmo nome ou um arquivo com mesmo nome de um subdiretório. Os diretórios no Linux são especificados por uma "/".

OBS: Estamos habituados a ver os nomes de arquivos como sendo: `nome.ext`, onde `ext` é a extensão do tipo de arquivo, por exemplo `.txt, .html, .doc`. Porém, no Linux os nomes não precisam seguir essa regra. Os nomes podem podem ser formados por várias extensões, como: `lista.ord.txt, nomes.maius.classe, livros.meu.ord.txt`.

O Linux organiza seu *filesystem* através de *inodes*. *Inodes* são estruturas de dados em disco que descrevem e armazenam os atributos do arquivo, incluindo sua localização. Campos de um *inode*: *user*, *group*, tipo do arquivo, tempo de criação, acesso, modo (modificação), número de links, tamanho e endereço no disco.

Existem mecanismos que permitem que vários *filenames* refiram-se a um único arquivo no disco. Esses mecanismos são os *links*. Existem dois tipos de *links*: *hard link* (ou simbólico) ou *soft link*.

- ***hard link***: associa dois ou mais *filenames* como mesmo *inode*. Os *hard links* compartilham o mesmo bloco de dados embora funcionando como entradas de diretório independentes.
- ***soft link***: estes são *pointers files* que apontam para outro *filename* no *filesystem*.

2.2.3 Permissões para acesso a arquivos

O sistema UNIX fornece um meio fácil de controlar o acesso que os usuários do sistema possam ter aos três tipos de arquivos. Isso é feito para permitir ou restringir o acesso a arquivos importantes do sistema e para garantir a privacidade de cada usuário que possua uma conta no sistema. O sistema diferencia três classes de usuários. Primeiro, todo arquivo possui um dono, um proprietário, designado no sistema por *user*. O proprietário tem controle total sobre a restrição ou permissão de acesso ao arquivo a qualquer hora. Além da posse individual do arquivo, é possível que um ou mais usuários do sistema possuam o arquivo coletivamente, em um tipo de propriedade de grupo. O usuário que não for proprietário do arquivo pode ter acesso a ele caso pertença ao grupo de usuários que tem permissão

para isso. Porém esse usuário não pode restringir ou permitir o acesso ao arquivo. Os usuários que não são nem proprietários nem pertençam a um grupo que tenha acesso ao arquivo formam a última categoria, conhecida simplesmente como "outros".

- **Proprietário** (designado por **u**, de *user*): quem criou o arquivo.
- **Grupo** (designado por **g**, de *group*): o grupo é formado por um ou mais usuários que podem ter acesso ao arquivo.
- **Outros** (designado por **o**, de *others*): refere-se a qualquer outro usuário do sistema.

O sistema Unix permite ainda três modos de acesso aos arquivos: leitura, escrita e execução. Os três modos de acesso são relativamente lógicos, porém o significado desses três modos de acesso é diferente para arquivos de diretórios. O usuário com permissão de leitura pode ler o conteúdo do diretório, por exemplo com o comando "`ls`". O usuário com permissão de escrita pode usar alguns programas privilegiados para gravar em um diretório. A permissão de gravação é necessária para criar ou remover arquivos do diretório.

Um usuário deve ter permissão de execução em um diretório para ter acesso aos arquivos ali alocados. Se um usuário tem permissão para leitura e escrita em um arquivo comum que está listado em um diretório mas não tem permissão de execução para aquele diretório, o sistema não o deixa ler nem gravar o conteúdo daquele arquivo comum.
Assim temos:

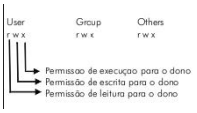

Figura 2.1: Permissões de Arquivos

Um arquivo que tiver suas permissões como as do exemplo acima poderia ser lido, escrito ou executado pelo dono ou por qualquer outra pessoa. Já no exemplo abaixo, o dono poderá ler e escrever, os outros só poderão ler e ninguém poderá executá-lo.

```
User        Group       Others
r w -       r - -       r - -
```

Para alterar essas permissões, utliza-se o comando `chmod`. Veja explicação do uso desse comando no próximo capítulo.

2.2.4 Diretórios

Chamamos de *árvore de diretórios* à organização dos arquivos de diretórios, fazendo uma alusão às suas ramificações, semelhantes aos galhos de uma árvore. Damos o nome de "*raiz*" ao diretório principal que contém todos os outros subdiretórios.

- *Diretório corrente* é o diretório em que o usuário se encontra naquele determinado momento.

- *Diretório "home"* é onde a conta do usuário está registrada.
- *Diretório ascendente* é o diretório onde determinado arquivo ou diretório está contido.
- *Diretório descendente* é o diretório contido em outro diretório.
- O "*path*" consiste na lista de diretórios que precisa ser atravessada, desde o diretório raiz até o arquivo, com barras separando os componentes. *Path* absoluto é o caminho completo desde a raiz até o arquivo. Se o caminho completo desde a raiz até o arquivo for `/UFF/pet/cursos/Linux` e o diretório de trabalho do usuário fosse `pet` no momento, bastaria que ele digitasse `cursos/Linux`. Este nome é chamado caminho relativo.

Figura 2.2: Padrão dos principais diretórios do UNIX.

Os diretórios principais, além de outros, com seu conteúdo estão listados abaixo:
`/etc` - Arquivos para administração do sistema.

`/bin` - Comandos UNIX mais comumente usados.

`/usr` - Todas as contas de usuários e alguns comandos.

`/dev` - Arquivos de dispositivos de entrada e saída.

`/lib` - Arquivos bibliotecas para programação em C.

`/tmp` - Armazenamento de arquivos temporários.

`/mmt` - Montagem de discos e periféricos.

`/sbin` - Diretório usado na inicialização do sistema. Demais pacotes para a administração do sistema devem ficar em /usr/sbin ou /usr/local/sbin.

`/var` - Diretório que contém arquivos variáveis, tais como *spool* (filas de *e-mail*, *crontab*, impressão) e *logs*. Este diretório existe para que os arquivos que necessitem ser modificados fiquem nele e não no `/usr`.

`/home` - Contém os diretórios pessoais dos usuários.

`/root` - É o diretório que contém os arquivos do administrador (seu *home*). Porém alguns sistemas Unix-*like* utilizam `/home/root`.

`/proc` - Diretório virtual onde o *kernel* armazena suas informações.

`/boot` - Contém a imagem do kernel e tudo o que for necessário ao processo de boot, menos configurações.

2.3 Processos

2.3.1 Processos e subprocessos

Um processo é um simples programa que está rodando em seu espaço de endereçamento virtual próprio. É distinto de um '*job*' ou comando, que, em sistemas UNIX, pode ser composto de muitos processos realizando uma única tarefa. Comandos simples como '`ls`' são executados como simples processos. Um comando composto, contendo *pipes*, irá executar um processo por segmento *pipe*. Para acompanhar o status de cada processo o sistema cria um **PID** (*Process* ID) para cada processo aberto.

Existem elementos chamados guias de controle de execução que ajudam a controlar a execução do programa.

Fork : é a chamada de sistema cuja execução cria um processo (filho) idêntico àquele que o chamou.

Processo pai : processo que gera um novo processo.

Processo filho : Processo gerado pelo processo pai.

Pid (*Process* ID) : é um número que identifica unicamente cada processo e é usado para referir-se a ele.

PPid (*Parent Process ID*) : é o Pid do processo pai.

2.3.2 Controle de processos

Aqui serão apresentados alguns tópicos que permitirão ao usuário realizar algumas tarefas interessantes como:

- executar mais de um processo ao mesmo tempo;
- rodar um processo em baixa prioridade.

Se você estiver usando um programa que irá demorar muito para terminar e você quer começar a trabalhar em outra tarefa você pode chamar seu programa para ser executado no que é conhecido como segundo plano (***background***). Isso pode ser feito colocando no final da linha de comando o símbolo '`&`'. Pode-se também, enquanto o programa está sendo executado, teclar `CTRL + Z`. Com isso o comando será suspenso e o sistema largará o *prompt*.

Chamamos de ***foreground*** o ato de fazer com que um programa rode em primeiro plano. Neste modo os processos podem interagir com os usuários e exibem a execução no monitor.

Quando o comando `nice` é colocado antes de uma linha de comando, o tempo dedicado pela CPU para execução daquela tarefa é reduzido, fazendo com que ele não atrase tanto outros processos mais importantes. Ou seja, a prioridade daquele processo é reduzida. Esse comando deve ser usado quando há outros processos mais importantes rodando no sistema.

Outros comandos relacionados a processos:

`jobs` lista os *jobs* em *background*.

`fg [job]` leva o *job* para *foreground*.

`bg [job]` leva o *job* para *background*.

`ps` mostra os processos que estão sendo rodados no computador.

`top` mostra continuamente o *status* de cada processo que está rodando no computador.

`kill [Pid]` encerra o processo.

O exemplo abaixo ilustra o uso de alguns desses comandos. Existe o seguinte *script* rodando no sistema:

```
#!/bin/bash

#Aviso

sleep 10m
echo Tá na Hora de Sair!!
date
```

Este programa emite um aviso após 10 minutos de ter sido colocado para rodar, em seguida mostra a data e a hora. Não teria sentido deixar esse programa rodando travando o terminal enquanto você poderia estar fazendo outras coisas, então ele pode ser colocado em background para liberar o terminal. Após 10 minuntos a mensagem aparece. O programa se chama `trava.sh`.

```
$ ./trava.sh &
[1] 2906
$ jobs
[1]+  Running                 ./trava.sh &
$ ps
 PID TTY          TIME CMD
1916 ttyp1    00:00:00 bash
2906 ttyp1    00:00:00 trava.sh
2907 ttyp1    00:00:00 sleep
2909 ttyp1    00:00:00 ps
$
$ fg 1
./trava.sh
aqui foi digitado: ctrl + z
[1]+  Stopped                 ./trava.sh
$ jobs
[1]+  Stopped                 ./trava.sh
$ jobs
```

```
[1]+  Stopped                 ./trava.sh
$ bg 1
[1]+ ./trava.sh &
$ jobs
[1]+  Running                 ./trava.sh &
$ ... comandos diversos do usuário enquanto ele espera a mensagem aparecer ...
$ Tá na Hora de Sair!!
Ter Jan 10 16:01:51 BRST 2006
aqui foi digitado: enter
[1]+  Done                    ./trava.sh
$ jobs
$ ps
 PID TTY          TIME CMD
1916 ttyp1    00:00:00 bash
2935 ttyp1    00:00:00 ps
$
```

Como pode ser visto, o programa foi executado em *background*. O comando jobs mostrou o job `trava.sh` em todos os seus estados: rodando, suspenso e executando, assim como fez o comando `ps`.

Capítulo 3

Comandos

Os sistemas operacionais UNIX e os que são derivados do UNIX (FreeBSD, Linux, etc) possuem muitos comandos e aplicativos. Em geral a sintaxe de uma comando é da seguinte forma:

`$ comando -opções argumentos`

Podem ser inseridos comandos seguidos na mesma linha separando-os por ponto-vírgula.

Ex: `$ cd;pwd`

3.1 Comandos de ajuda

As distribuições UNIX já vem com manuais incorporados ao sistema. Esses manuais de comandos se localizam no diretório `/usr/man`. A utilização da página de manual é bem simples, digite, por exemplo:

Sintaxe: `$ man [opção] <comando>`

Ex:
`$ man ls` → exibe o manual do comando '`ls`'

Um modo de procura por palavra-chave pode ser usado para encontrar o nome do comando que execute determinada tarefa. Para isso digite:

`$ man -k  palavra-chave`

Outros comandos são: `info, whatis, apropos`. Consulte o manual para informações sobre esses comandos. .

`apropos` : **procura por comandos/programas pela descrição**

Sintaxe: `apropos <descrição>`

É útil quando precisamos fazer alguma coisa mas não sabemos qual comando usar. Ele faz sua pesquisa nas páginas de manual e suas descrições e lista os comandos/programas que atendem à consulta.

3.2 Comandos e utilitários básicos

Os comandos serão exemplificados supondo haver inicialmente um diretório chamado `teste` contendo os seguinte arquivos: **arqu1.txt, arqu2.txt, cap2.tex, parI.tex, parII.tex, parIII.tex, imag.jpg, page.html**.

3.2.1 Comandos de manipulação de arquivos

`cat` **concatena e/ou exibe arquivos na saída *default*.**

Também pode exibir o conteúdo de vários arquivos em sucessão.

Sintaxe: `cat [opção] <arquivo>`

As principais opções são:

`-n` – Numera as linhas.

`-E` – Exibe $ ao final de cada linha.

`-A` – Exibe todo o conteúdo incluindo caracteres especiais, como acentos e espaços na forma de códigos.

Ex: Mostra o conteúdo do arquivo com as linhas numeradas.

```
$ cat -n arqu1.txt
     1
     2  "Software Livre" é uma questão
     3   de liberdade, não de preço.
     4   Para entender o conceito,
     5   você deve pensar em
     6  "liberdade de expressão",
     7   não em "cerveja grátis".
     8
```

Ex: Concatena na saída padrão o conteúdo dos dois arquivos.

```
$ cat  arqu1.txt arqu2.txt

"Software Livre" é uma questão
 de liberdade, não de preço.
 Para entender o conceito,
 você deve pensar em
"liberdade de expressão",
 não em "cerveja grátis".
```

```
=> A liberdade de executar o programa, para qualquer
   propósito (liberdade no. 0)
=> A liberdade de estudar como o programa funciona,
   e adaptá-lo para as suas necessidades
   (liberdade no. 1). Aceso ao código-fonte
   é um pré-requisito para esta liberdade.
=> A liberdade de redistribuir cópias de modo
   que você possa ajudar ao seu próximo
   (liberdade no. 2).
=> A liberdade de aperfeiçoar o programa e
   liberar os seus aperfeiçoamentos, de modo
   que toda a comunidade se beneficie.
   (liberdade no. 3). Acesso ao código-fonte
   é um pré-requisito para esta liberdade.
```

`tac` - **Semelhante ao comando `cat`, porém exibe o conteúdo em ordem reversa.**

Sintaxe: `tac [opção] <arquivo>`

`file` - **Identifica o conteúdo e mostra na tela o tipo de arquivo.**

É realizado um exame dos primeiros *bytes* do arquivo, seguido de uma comparação com as regras definidas nos arquivos:

`/usr/share/misc/magic` ou `/etc/magic`

O segundo arquivo, `/etc/magic`, pode ser editado pelo próprio usuário para identificar seus arquivos.

Ex:

```
$ file arqu1.txt
arqu1.txt: ISO-8859 text
$ file image.jpg
image.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI),
100 x 100 $ file h264 h264: directory
```

Pelo exemplo acima, vemos que os tipos de arquivos foram: text, image e directory. Além desses podem haver: executable, (para arquivos executáveis resultado da compilação de um programa), data (contendo dados geralmente não imprimíveis).

Veja mais opções no manual do sistema.

`cp (copy)` **Copia arquivos ou diretórios.**

Sintaxe: `cp [opções] <arqfont> <arqdest>`

Onde:

`arqfont` é o arquivo a ser copiado;

`arqdest` é o nome da cópia a ser criada. O nome do `arqdest` deve ser diferente do `arqfont` caso se esteja fazendo uma cópia para o mesmo diretório. Se o arquivo destino não existir, ele é criado com o nome `arqdest`. E caso exista e não seja um diretório, seu conteúdo será sobrescrito.

As opções podem ser:

`-i` – Pede confirmação para copiar o arquivo.

`-p` – Mantém os dados, como por exemplo permissões e datas do arquivo original.

`-r` – Copia os arquivos e diretórios recursivamente.

Ex:

```
$ cp arqu1.txt novoarq.txt
$ ls -l
total 44
 -rw-r--r--    1 kurumin  kurumin      395 2005-03-08 21:46 apend.tex
 -rw-r--r--    1 kurumin  kurumin      167 2005-03-08 22:53 arqu1.txt
 -rw-r--r--    1 kurumin  kurumin      606 2005-03-08 22:43 arqu2.txt
 -rw-r--r--    1 kurumin  kurumin        0 2005-03-08 20:09 cap2.tex
 -rwxrwxrwx    1 kurumin  kurumin     5684 2005-03-05 19:29 imag.jpg
 -rw-r--r--    1 kurumin  kurumin      167 2005-03-08 23:23 novoarq.txt
 -rw-r--r--    1 kurumin  kurumin     5740 2005-03-08 20:08 page.html
 -rw-r--r--    1 kurumin  kurumin      113 2005-03-08 21:42 parIII.tex
 -rw-r--r--    1 kurumin  kurumin      125 2005-03-08 21:41 parII.tex
 -rw-r--r--    1 kurumin  kurumin      214 2005-03-08 21:40 parI.tex
```

Como pode ser visto no exemplo, foi criado um arquivo chamado `novoarq.txt`

`rm (remove)` **Remove arquivos ou diretórios**

Sintaxe: `rm [opções] <arq1> <arq2>`

Onde cada arquivo é separado por espaços em branco.

Algumas das opções podem ser:

`-f` – Remove todos os arquivos mesmo não tendo permissão de escrita sem pedir confirmação do usuário.

`-i` – Remove o arquivo interativamente, ou seja, pede a confirmação do usuário.

`-r` – Remove um diretório e todo o seu conteúdo recursivamente.

Obs: Cuidado ao apagar os arquivos, pois uma vez usado o comando `rm` não é possível recuperar o arquivo.

Ex: Supondo o mesmo diretório exemplo com os arquivos descritos. Agora vamos apagar o arquivo `cap2.tex`.

```
$ rm -i cap2.tex
rm: remove arquivo comum ‘cap2.tex’? y
$ ls
apend.tex  arqu2.txt  novoarq.txt  parIII.tex  parI.tex
arqu1.txt  imag.jpg   page.html    parII.tex
$
```

Foi pedida confirmação para a exclusão do arquivo, a letra `y` foi digitada para confirmar, em seguida foi usado o comando `ls` para listar os arquivos. Como pode ser visto, o arquivo foi apagado.

`mv (move)` **Move ou renomeia arquivos**

Sintaxe: `mv [opcao] <origem> <destino>`

Remove o arquivo da `origem` para `destino` ou renomeia arquivo `origem` para arquivo `destino`.

A opção -i pede confirmação antes de mover um arquivo que irá sobrescrever. Exemplo:

`$mv imag.jpg nova.jpg` Renomeia o arquivo `imag.jpg` para `nova.jpg`.

`$mv list.tex ~/teste/` Move o arquivo `list.tex` para o diretório `~/teste/`.

```
$ls
apend.tex  arqu2.txt  nova.jpg     page.html   parII.tex
    arqu1.txt  list.tex   novoarq.txt  parIII.tex  parI.tex
```

`ln` **Cria *links* entre arquivos.**

Sintaxe: `ln [opção] arquivo1 arquivo2`

Sem opção, o comando `ln` dá um outro nome para um mesmo arquivo na memória. Ou seja, um mesmo arquivo pode ser referenciado de duas maneiras diferentes.

Já usando a opção `-s` é criado um link simbólico, ou seja, um arquivo que aponta para a área onde está o arquivo original.

Ex: Tendo o arquivo chamado `arq2.txt`, será criado um link de hardware e simbólico para ele. Observe a diferença no tamanho dos arquivos criados. O link simbólico é apenas um "atalho"para o arquivo original, por isso seu tamanho é menor. Atente também para o fato do uso do comando ln sem opção não é igual ao comando `cp` como pode parecer, pois os arquivos continuam interligados. A modificação de um implica na modificação do outro.

```
$ ls -l
total 4
-rw-r--r--    1 kurumin  kurumin        27 2006-02-03 10:36 original.txt
$ ln original.txt hard.txt
$ ls -l
total 8
-rw-r--r--    2 kurumin  kurumin        27 2006-02-03 10:36 hard.txt
-rw-r--r--    2 kurumin  kurumin        27 2006-02-03 10:36 original.txt
$ cat hard.txt
Esse é o arquivo original

$ cat >> hard.txt
Primeira modificação

$ cat  hard.txt
Esse é o arquivo original

Primeira modificação

$ cat  original.txt
Esse é o arquivo original

Primeira modificação

$ ls -l
total 8
-rw-r--r--    2 kurumin  kurumin        49 2006-02-03 10:38 hard.txt
-rw-r--r--    2 kurumin  kurumin        49 2006-02-03 10:38 original.txt
$ ln -s original.txt simb.txt
$ ls -l
total 9
-rw-r--r--    2 kurumin  kurumin        49 2006-02-03 10:38 hard.txt
-rw-r--r--    2 kurumin  kurumin        49 2006-02-03 10:38 original.txt
lrwxrwxrwx    1 kurumin  kurumin        12 2006-02-03 10:39 simb.txt -> original.txt
$ cat simb.txt
Esse é o arquivo original

Primeira modificação

$ cat >> simb.txt
Segunda modificação

$ cat simb.txt
Esse é o arquivo original

Primeira modificação
```

```
Segunda modificação

$ cat original.txt
Esse é o arquivo original

Primeira modificação

Segunda modificação

$
```

`diff` **Exibe a diferença entre dois arquivos.**

Sintaxe: diff [opção] arquivo1 arquivo2

Algumas das opções são:

`-b` – Ignora espaços e caracteres de tabulação.

`-i` – Não diferencia maiúscula de minúscula.

`-r` – Processa subdiretórios quando diretórios são comparados.

Ex:

```
$ cat parI.tex
parte um
esta linha esta igual nos dois
mas a linha seguinte não.
$ cat parII.tex
parte dois
esta linha esta igual nos dois
porem a linha seguinte não
$ diff parI.tex parII.tex
1c1
< parte um
---
> parte dois
3c3
< mas a linha seguinte não.
---
> porem a linha seguinte não
$
```

`chmod` **Altera a permissão de acesso aos arquivos.**

Sintaxe: `chmod [opção] <permissões> <arquivo>`

Algumas das `opções` podem ser:

`-R` – Se o arquivo for um diretório, o comando muda recursivamente o modo de acesso a todos os seus arquivos e subdiretórios.

`-c` – Mostra o resultado do uso do comando após seu uso.

`permissões` é composto pela classe do usuário (`u` para dono, `g` para grupo, `o` para outros tipos e `a` todos), pelos caracteres operadores (+ para acrescentar permissões, - para retirar permissões e = para retirar todas as permissões) e pelos caracteres de permissão (`r` para leitura, `w` para escrita e `x` para execução).

Exemplos:

```
$ ls -l
--wx-wx-wx    1 ze ze      5684 2005-03-05 19:29 nova.jpg
$ chmod -c a+rwx
modo de ‘nova.jpg’ mudado para 0777 (rwxrwxrwx)

$ chmod -c og-x nova.jpg
modo de ‘nova.jpg’ mudado para 0766 (rwxrw-rw-)
```

A mudança de permissão também pode ser feita colocando o código `rwx` na forma de números octais. Abaixo segue a equivalência do código nas letras rwx pra números.

- - - ⇒ 0
- - x ⇒ 1
- w - ⇒ 2
- w x ⇒ 3
r - - ⇒ 4
r - x ⇒ 5
r w - ⇒ 6
r w x ⇒ 7

Ex: Supondo que o arquivo parI.tex.

```
$ ls -l parI.tex
-r--r--r--    1 kurumin  kurumin        10 2005-03-15 15:18 parI.tex
$ chmod 754 parI.tex
$ ls -l parI.tex
-rwxr-xr--    1 kurumin  kurumin        10 2005-03-15 15:18 parI.tex
```

Comandos para manipulação de diretórios

`ls` Lista o conteúdo de um diretório e informações relativas aos arquivos.

Deriva da palavra '*list*'; quando se digita '`ls [nome do arquivo]`', o programa procura o arquivo desejado dentro do diretório corrente.

Ex:

```
$ ls
apend.tex  arqu2.txt  imag.jpg   parIII.tex  parI.tex
arqu1.txt  cap2.tex   page.html  parII.tex
```

O `ls` sem argumentos mostra apenas os nomes dos arquivos.

Sintaxe: `ls [opção] [arquivo]`

Os parâmetros opcionais podem ser:

`-l` – lista ordenada pelo nome e em formato longo

`-F` – mostra barra de diretórios

`-R` – mostra o conteúdo de todos os subdiretórios

`-x` – lista o resultado em várias colunas na horizontal

`-a` – lista todos os arquivos, inclusive os ocultos

`-i` – exibe o número do inode na primeira coluna

`-t` – lista em ordem cronológica em função da hora da última modificação

`-1` – lista somente os nomes dos arquivos ordenados.

Obs: podem ser usados vários parâmetros opcionais em conjunto.

Ex:

```
$ ls -la
total 48
drwxr-xr-x    2 ze ze  4096 2005-03-08 21:46 .
drwxr-xr-x   31 ze ze  4096 2005-03-08 21:59 ..
-rw-r--r--    1 ze ze   395 2005-03-08 21:46 apend.tex
-rw-r--r--    1 ze ze   159 2005-03-08 21:34 arqu1.txt
```

```
-rw-r--r--    1 ze ze  1188 2005-03-08 21:36 arqu2.txt
-rw-r--r--    1 ze ze     0 2005-03-08 20:09 cap2.tex
-rwxrwxrwx    1 ze ze  5684 2005-03-05 19:29 imag.jpg
-rw-r--r--    1 ze ze  5740 2005-03-08 20:08 page.html
-rw-r--r--    1 ze ze   113 2005-03-08 21:42 parIII.tex
-rw-r--r--    1 ze ze   125 2005-03-08 21:41 parII.tex
-rw-r--r--    1 ze ze   214 2005-03-08 21:40 parI.tex
```

Segue abaixo a explicação do resultado do comando `ls -la`:

`-la` É a combinação dos parâmetros `l` e `a`.

total 48 É o número de blocos ocupados pelos arquivos no diretório. Em algumas versões o tamanho de cada bloco é de 512 *Bytes* e em outras de 1K *Byte*.

Primeiro Campo Exibe as permissões dos arquivos.

Segundo Campo Exibe o número de ligações dos arquivos.

Terceiro Campo Exibe o proprietário do arquivo.

Quarto Campo Exibe o grupo do arquivo.

Quinto Campo Exibe o tamanho do arquivo em *Bytes*.

Sexto Campo Exibe a data e hora da última modificação do arquivo.

Sétimo Campo Exibe informação sobre o arquivo ou diretório.

`pwd` **(Present Working Directory)** : **mostra o diretório corrente segundo o percurso absoluto de localização.**

Sintaxe: `pwd`

```
$ pwd
 /home/ze/teste
```

`cd(change directory)` : **Troca de diretório corrente**

Sintaxe: `cd <diretório>`

Exemplo:

`$ cd /` Vai para o diretório raiz.

`$ cd ..` Vai para o diretório pai.

`$ cd` Vai para o diretório *home* do usuário.

`$ cd -` Vai para o último diretório acessado.

Ex: Suponha que o diretório atual seja:

```
$ pwd
 /home/ze/teste
```

Fazendo

```
$ cd /usr/bin
$ pwd
  /usr/bin
```

Vale a pena testar as várias opções deste comando, pois, conhecendo alguns atalhos, ele agiliza a navegação dentro do sistema. Veja a seção 3.3. Nela podem ser encontrados os caracteres especiais mais usados.

`mkdir` : **Cria diretórios**

Sintaxe: `mkdir [opções] <diretório>`

Exemplo: Será criado um diretório chamado `teste`

```
$ ls
  curso
  notas
  artigos
$ mkdir teste
$ ls
  curso
  notas
  artigos
  teste
```

`rmdir` : **Remove diretórios vazios**

Sintaxe: rmdir <diretório>

Exemplo: O diretório teste criado no exemplo anterior será removido.

```
$ rmdir teste
$ ls
  urso
  notas
  artigos
```

Comando para manipulação de contas e usuários

`id` : **Mostra o número de identificação do usuário e dos grupos a que ele pertence.**

`whoami` : **Mostra o nome do usuário.**

`who` : **Mostra os usuários que estão utilizando os terminais.**

`w` : **Mostra os usuários que estão utilizando os terminais com informações adicionais.**

`su` : **Troca o usuário.**

Esse comando é usado para mudar o usuário no mesmo terminal. Se o comando for executado sem qualquer nome do usuário, o padrão é mudar o usuário para *root*.

Ex:

```
   $ su
Password:
   #
```

```
   $ su user
Password:
   /home/user$
```

`sudo` : **Executa comandos como root.**

`adduser` : **Adiciona um usuário ou grupo no sistema**

Sintaxe: `adduser [opções] <usuário/grupo>`

Por padrão, quando um usuário é adicionado, é criado um grupo com o mesmo nome do usuário.

`addgroup` : **adiciona um novo grupo de usuários no sistema**

Sintaxe: `addgroup <usuário/grupo> [opções]`

`passwd` : **Altera a senha do usuário**

Sintaxe: `passwd <usuário> [opções]`

Se a opção `-g` for especificada a senha do grupo será alterada.

Obs: apenas o usuário *root* pode alterar a senha de outros usuários.

`newgrp` : **Altera a identificação de grupo do usuário**

Sintaxe: `newgrp <grupo>`

Comandos diversos

`alias` : **Cria um apelido para o comando e seus parâmetros**

Sintaxe: `alias <apelido>=‘<comando + parâmetros>’`

Exemplo: Será criado um apelido para aprimorar o `ls`

```
$ alias ls=‘ls -al -color -F’
$ ls
RESULTADO
```

Exemplo: Será criado um apelido para montar a unidade de disquete e entrar na tela.

```
$alias a=‘mount -t /dev/fd0/mnt/floppy/   cd/mnt/floppy’
```

`cal` : **Imprime o calendário para um determinado mês/ano**

Sintaxe: `cal [mês] [ano]`

Exemplo:
`$cal 6 2000`

`$cal 2000`

Algumas opções podem ser usadas através da sintaxe: `cal [opção]`

`-3` - Mostra o mês atual, o anterior e o seguinte.
`-y` - Mostra o calendário do ano inteiro.

Veja o manual para outras opções.

`clear` : **Limpa o terminal**

Sintaxe: `clear`

Quando o terminal estiver cheio de comandos já digitados ou de resultados de outros comandos, é recomendável utilizar o `clear` para aumentar o campo visual.

`date` : **Mostra a data e a hora atuais do sistema**

Sintaxe: `date`

Se digitarmos '`date -u`', o comando irá mostrar o horário de Greenwich.
Confira o manual para ver como modificar a formatação de apresentação da data.
Para modificar a hora, é necessário estar logado como *root*
`$ date mes/dia/hora/minuto/ano`
Então para mudar a data atual para 25 de dezembro de 2020, às duas e meia da tarde, digite:

```
$ date 122514302020
```

`df (disk free)` : **Mostra o espaço livre no disco**

Sintaxe: `df [opções]`,

onde opções podem ser os parâmetros listados abaixo:
`-a` – mostra o espaço ocupado por todos os arquivos

`-b` – mostra o espaço ocupado em *bytes*

`-c` – faz uma totalização de todo o espaço listado

`-D` – não conta *links* simbólicos

`-k` – mostra o espaço ocupado em *kbytes*

`-m` – mostra o espaço ocupado em *Mbytes*

`-S` – não calcula o espaço ocupado por subdiretórios

`du (disk usage)` : **Mostra o espaço utilizado por arquivos e diretórios do diretório atual**

Sintaxe: `du [opções]`,

onde opções podem ser os parâmetros listados abaixo:
`-s` – relata apenas o número de blocos

`-a` – informa o tamanho de cada arquivo

`-k` – lista o tamanho em *Kbytes*

`echo` : **Escreve no terminal**

Sintaxe: `echo [string]`

Exemplos do uso do comando `echo` serão vistos na seção 4.4.

`find` : **Procura arquivos por nome**

Sintaxe: `find <caminho> [opções]`

Exemplos:
`$find / -name prova` (procura no diretório raiz e subdiretórios um arquivo chamado `prova`)

`$find / -name prova -maxdepth 3` (procura no diretório raiz e subdiretórios, até o 3° nível, um arquivo chamado `prova`)

`$find / -mmin 10''` (procura no diretório raiz e subdiretórios um arquivo modificado há 10 minutos atrás)

`$find / -links 4''` (procura no diretório raiz e subdiretórios, todos os arquivos que possuem 4 *links* como referência)

`sync` : **Grava os dados do *cache* de disco na memória RAM para os disco rígidos e flexíveis**

Sintaxe: `sync`

O *cache* é um mecanismo de aceleração que permite que um arquivo seja armazenado na memória ao invés de ser imediatamente gravado no disco. Quando o sistema estiver ocioso, o arquivo é gravado para o disco. O Linux pode usar toda memória RAM disponível para o *cache* de programas acelerando seu desempenho de leitura/gravação. O uso do `sync` é útil em disquetes quando gravamos um programa e precisamos que os dados sejam gravados imediatamente para retirar o disquete da unidade.

Opções de compactação

O Linux possui programas de armazenamento de arquivos que podem ou não compactar os mesmos.

`tar` : **armazena vários arquivos e diretórios dentro de um único arquivo (`*.tar`) ou dispositivo**

Sintaxe: `tar [opções] <arquivo>`,

As opções podem ser os parâmetros listados abaixo:

`-c` – cria um novo arquivo `tar`

`-h` – armazena os arquivos apontados por *links*, ao invés dos *links*

`-r` – inclui arquivos ao final (*append*) de um arquivo `tar`

`-t` – lista o conteúdo de um arquivo `tar`

`-u` – somente inclui arquivos que não estão presentes ou são mais novos do que o arquivo `tar`

`-w` – pede confirmação para cada ação a ser tomada

`-v` – mostra na tela o que está sendo feito

`-x` – extrai arquivos armazenados em um arquivo `tar`

```
$ ls
arq1.txt  arq2.txt  hard.txt  original.txt  simb.txt
$ tar -cvf junto.tar arq*.txt
arq1.txt
arq2.txt
$ ls
arq1.txt  arq2.txt  hard.txt  junto.tar  original.txt  simb.txt
$
```

A fim de reduzir o espaço ocupado no disco, o uso da compactação deste arquivo se torna útil e eficiente. Para esta tarefa usamos o comando `gzip`.

`gzip` : **compactador de arquivos mais usado atualmente em sistemas UNIX**

Sintaxe: `gzip [opções] <arquivo>`,

onde opções podem ser os parâmetros listados abaixo:

`-c` – concatena saída de dados

`-h` – lista todas as opções

`-l` – lista os nomes no conteúdo do arquivo `.gz`

`-r` – exibe em modo recursivo

`-t` – usado para testar arquivos

`-v` – modo verbose, exibe informações sobre compactação.

```
$ ls
arq1.txt  arq2.txt  hard.txt  junto.tar  original.txt  simb.txt
$ gzip -v junto.tar
junto.tar:       97.6% -- replaced with junto.tar.gz
$ ls
arq1.txt  arq2.txt  hard.txt  junto.tar.gz  original.txt  simb.txt
```

`gunzip` : **descompactador de arquivos** `.gz`

Sintaxe: `gunzip [opções] <arquivo.gz>`

```
$ ls
arq1.txt  arq2.txt  hard.txt  junto.tar.gz  original.txt  simb.txt
$ gunzip junto.tar.gz
$ ls
arq1.txt  arq2.txt  hard.txt  junto.tar  original.txt  simb.txt
$
```

Existem outras opções de compactadores em sistemas UNIX. Cada um define um arquivo com uma extensão diferente, como os exemplos:

Compactador	Descompactador	Extensão
gzip	gunzip	.gz
pack	unpack	.z
compress	uncompress	.Z
bzip2	unbzip2	.bz2
zip	uzip	.zip

Tabela 3.1: Compactadores

3.2.2 Redirecionamento de entrada e saída

O UNIX trata todos os periféricos conectados ao sistema como arquivos. O teclado, por exemplo, é um "arquivo"de entrada; o vídeo é um arquivo de saída, assim como a impressora.

O arquivo padrão de saída (**stdout**) é o dispositivo no qual o UNIX deseja os resultados por *default*. Esse dispositivo geralmente é o vídeo. Praticamente todos os programas componentes do sistema que apresentam dados, o fazem como **stdout**, como por exemplo, o comando `ls`. Assim como o dispositivo padrão de entrada (**stdin**) de dados geralmente é o teclado. Há, também, uma saída padrão para os erros (**stderr**).

Redirecionando saída padrão para um arquivo

Utilizando o símbolo > podemos redirecionar a saída que seria para o vídeo para, por exemplo, um arquivo em disco.

Abaixo, a saída do comando `date` é gravada para um arquivo de nome `datas`.

Exemplo:

```
$ date > datas
```

```
$ cat datas
```

```
Qui Mar 17 21:20:06 BRT 2005
```

Acrescentando dados em um arquivo

O operador > cria um novo arquivo, caso ele não exista. Se ele já existe, o arquivo é sobreposto. Usando », a saída é acrescentada ao final do arquivo se o mesmo já existe ou é criado um novo arquivo caso ele não exista.

Exemplo:

```
$date > datas

$cat datas

Qui Mar 17 21:22:31 BRT 2005

$echo FIM DO ARQUIVO >> datas

$ cat datas

Qui Mar 17 21:22:31 BRT 2005 FIM DO ARQUIVO
```

No caso de erros, deve-se utilizar os símbolos 2> ou 2» para direcionar o erro da saída padrão para um determinado arquivo.

Exemplo:

```
$ write 2> erros
$ cat erros
write: write: you have write permission turned off.

$ cd f 2>> erros
$ cat erros
write: write: you have write permission turned off.

bash: cd: f: No such file or directory
$
```

Da mesma forma que a saída de um comando pode ser direcionada para um arquivo, um arquivo pode ser direcionado como entrada de um comando. Isso é feito utilizando o símbolo < . Na próxima seção teremos um exemplo de como fazer isso.

Também é possível fazer redirecionamento de entrada utilizando o sinal « . Este símbolo indica ao shell que o escopo de um comando começa na linha seguinte e termina quando ele encontrar o *label* que o segue. Este comando é muito usado para exibir um texto em várias linhas, sem precisar de ficar usando o comando `echo` para cada linha.

Exemplo:

```
$ cat <<FIM
> 1 - Opção UM
> 2 - Opção DOIS
> 3 - Opção TRÊS
> FIM
1 - Opção UM
2 - Opção DOIS
3 - Opção TRÊS
$
```

Organizando dados em um arquivo

O "`sort`"é um programa que classifica alfabeticamente as linhas de um arquivo texto. Ele obtém sua entrada em *stdin* e apresenta a saída em *stdout*. Se quisermos ordenar os nomes contidos no arquivo **nomes** utilizando *sort*, podemos fazer:

```
$cat  nomes
Deri
Cod
Pato
Brunaldo
KK
$

$sort < nomes
Brunaldo
Cod
Deri
KK
Pato
$
```

A saída poderia ser redirecionada ao mesmo tempo para um arquivo, como mostrado abaixo:

```
$sort < nomes >   nomesord
$cat nomesord
Brunaldo
Cod
Deri
KK
Pato
$
```

O `sort` leu o conteúdo do arquivo "`nomes`", classificou-os e a saída, que seria feita na tela, foi jogada para um novo arquivo chamado "`nomesord`". Um detalhe importante: os arquivos de entrada e saída devem ser diferentes, pois a primeira parte do comando é a criação do arquivo de saída. Se este for igual ao de entrada, ele é zerado.

Filtros

Os chamados programas "*filtro*"pegam sua entrada de *stdin*, fazem alguma modificação nesses dados e colocam a saída em um *stdout* (tela do terminal).

O mais simples programa filtro é o **cat**, que faz uma cópia fiel da entrada na saída. Outros programas desse tipo são: **sort**, **wc**, que conta as linhas, caracteres e palavras, **grep**, que mostra as linhas de um arquivo contendo uma seqüência de caracteres, e muitos outros comandos UNIX.

Não são programas filtro: **ls**, **who**, **date**, **cal**, pois, embora a saída seja para *stdout*, a entrada não é *stdin*.

`uniq` : **Remove linhas duplicadas dos arquivos.**

Mas há uma condição: a lista deve estar ordenada.

A opção `-d` faz com que sejam listados somente os nomes repetidos.

Ex: Observe que o comando `uniq` não dá resultado quando a lista não está ordenada.

```
$ cat lista
Alexandre
José
Felipe
Tatiana
Beatriz
Eduardo
Enrico
Aline
Erica
José

$ uniq lista
Alexandre
José
Felipe
Tatiana
Beatriz
Eduardo
Enrico
Aline
Erica
José

$ sort lista | uniq
Alexandre
Aline
Beatriz
Eduardo
Enrico
```

```
Erica
Felipe
José
Tatiana

$ sort lista | uniq -d
José
$
```

`split` : **Divide o arquivo em partes.**

As principais opções são:

`-bn` – Divide o arquivo em partes com n bytes de tamanho. A unidade de n pode ser em k (*kilobytes*) ou M (Megabytes).

`-ln` – Divide o arquivo em partes com n linhas de tamanho.

Sintaxe: `split [opções] <arquivo> [prefixo]`

Ex: O arquivo arqu2.txt será dividido em arquivos de 3 linhas cada.

```
$ split -l3 arqu2.txt arqlin
$ ls -l
 total 1080
-rw-r--r--    1 kurumin  kurumin       139 2006-01-11 11:36 arqlinaa
-rw-r--r--    1 kurumin  kurumin       131 2006-01-11 11:36 arqlinab
-rw-r--r--    1 kurumin  kurumin       111 2006-01-11 11:36 arqlinac
-rw-r--r--    1 kurumin  kurumin       130 2006-01-11 11:36 arqlinad
-rw-r--r--    1 kurumin  kurumin        89 2006-01-11 11:36 arqlinae
-rwxrwxrwx    1 kurumin  kurumin       161 2005-04-09 21:42 arqu1.txt
-rwxrwxrwx    1 kurumin  kurumin       600 2006-01-11 11:27 arqu2.txt
-rwxrwxrwx    1 kurumin  kurumin        10 2005-03-27 12:57 cap2.tex
-rwxrwxrwx    1 kurumin  kurumin      5615 2005-03-27 13:02 image.jpg
-rwxrwxrwx    1 kurumin  kurumin        71 2005-03-27 13:00 page.html
-rwxrwxrwx    1 kurumin  kurumin        10 2005-03-27 12:58 parIII.tex
-rwxrwxrwx    1 kurumin  kurumin        69 2006-01-10 16:42 parII.tex
-rwxr-xr--    1 kurumin  kurumin        67 2006-01-10 16:40 parI.tex
-rwxrwxrwx    1 kurumin  kurumin       161 2005-04-09 21:42 texto.txt
$ cat arqlinaa
=> A liberdade de executar o programa, para qualquer
   propósito (liberdade no. 0)
=> A liberdade de estudar como o programa funciona,
$ cat arqlinab
  e adaptá-lo para as suas necessidades
  (liberdade no. 1). Aceso ao código-fonte
  é um pré-requisito para esta liberdade.
```

```
$
```

`cut` : **Extrai colunas ou campos de um arquivo.**

Sintaxe: `cut [opções] <arquivo>`

As opções podem ser:

`-b` – Seleciona campos por bytes.

`-c` – Seleciona campos por caracteres.

`-f campos` – Seleciona lista de campos, onde podem ser números separados por vírgula ou faixas de números (`n1-n2`), ou combinação de ambas.

`-d` – Especifica o delimitador de campo.

E: O delimitador de campo informado no exemplo abaixo é um espaço em branco. Veja o resultado da escolha do campo 4 do comando `date`.

```
$ date
Qua Jan 11 14:04:49 BRST 2006
$ date|cut -d" " -f 4
14:06:31
$ date|cut  -c 5
J
$
```

`tr` : **Substitui caracteres**

Pode ser usado para trocar as letras de maiúsculas para minúsculas.

Sintaxe: `tr [opções] <string1> <string2>`

As opções principais são:

`-c` – Realiza a troca de todos os caracteres, exceto da string1.

`-d` – Elimina os caracteres especificados em string1, ignorando string2.

`-s` – Comprime a seqüência de caracteres repetidos da string2.

Ex: Converte as letras minúsculas para maiúsculas.

```
$ cat  capital.txt|tr "[a-z]" "[A-Z]"
 RIO DE JANEIRO
 SãO PAULO
 VITóRIA
 BELO HORIZONTE
 $
```

Ex: Converte múltiplos espaços em branco em um único espaço.

```
 $ cat capital.txt
 Rio      de      Janeiro
 São          Paulo
 Vitória
 Belo           Horizonte
 $ cat  capital.txt|tr -s " "
 Rio de Janeiro
 São Paulo
 Vitória
 Belo Horizonte
 $
```

`paste` : **Exibe lado a lado conteúdo de arquivos.**

Sintaxe: `paste [opções] <arquivo1> <arquivo2>`

As opções são:

`-s` – exibe as linhas de um arquivo em série em vez de uma abaixo da outra.

`-d c` – Especifica o delimitador de campos como sendo o caracter `c` em vez da tabulação.

Ex: Sejam os arquivos estado.txt e capital.txt. O estado será colocado ao lado da capital, separados por ": ".

```
 $ cat estado.txt
 Rio de Janeiro
 São Paulo
 Espírito Santo
 Minas Gerais
 $ cat capital.txt
 Rio de Janeiro
 São Paulo
 Vitória
 Belo Horizonte
```

```
$ paste -d: estado.txt capital.txt
Rio de Janeiro:Rio de Janeiro
São Paulo:São Paulo
Espírito Santo:Vitória
Minas Gerais:Belo Horizonte
$
```

`wc` : **Conta o número de linhas.**

Com as opções:

`-l` – Mostra apenas o número de linhas.

`-w` – Apenas o número de palavras

`-c` – Apenas o número de caracteres.

Sintaxe: `wc [opções] <arquivo>`

```
$ wc arqu2.txt
    14      90     600 arqu2.txt
$ wc -l arqu2.txt
    14 arqu2.txt
$ wc -w arqu2.txt
    90 arqu2.txt
$ wc -c arqu2.txt
   600 arqu2.txt
$
```

`tail` : **Exibe as últimas linhas de um arquivo**

Sintaxe: `tail [opções] <arquivo>`

Exemplo:

```
$ tail -n 5 arqu1.txt
Para entender o conceito,
você deve pensar em
"liberdade de expressão",
 não em "cerveja grátis".
$
```

As opções são semelhantes a do comando `head`, porém serão mostradas as últimas `m` linhas ou últimos `m` bytes.

`grep` : **Localiza cadeias de caracteres em uma entrada definida.**

Essa localização é baseada em expressões regulares. A entrada do comando `grep` pode ser um arquivo ou a saída de outro comando.

Sintaxe: `grep [opções] "expressão"[arquivo]`

Exemplo: Serão listadas todas as linhas dos arquivos `.txt` que contêm a palavra "liberdade".

```
$ grep liberdade *.txt
arqu1.txt: de liberdade, não de preço.
arqu1.txt:"liberdade de expressão",
arqu2.txt:=> A liberdade de executar o programa, para qualquer
arqu2.txt:   propósito (liberdade no. 0)
arqu2.txt:=> A liberdade de estudar como o programa funciona,
arqu2.txt:   (liberdade no. 1). Aceso ao código-fonte
arqu2.txt:   é um pré-requisito para esta liberdade.
arqu2.txt:=> A liberdade de redistribuir cópias de modo
arqu2.txt:   (liberdade no. 2).
arqu2.txt:=> A liberdade de aperfeiçoar o programa, e
arqu2.txt:   (liberdade no. 3). Acesso ao código-fonte
arqu2.txt:   é um pré-requisito para esta liberdad
```

Exemplo: Serão listadas as informações referentes aos arquivos .jpg do diretório corrente.

```
$ grep ls|grep *.jpg
-rwxrwxrwx    1 ze  ze      5615 2005-03-27 13:02 image.jpg
```

Há também comandos variantes do `grep`: o `egrep` e o `fgrep`. O primeiro tem como característica adicional a possibilidade do uso de metacaracteres. O segundo não apresenta essa funcionalidade. A vantagem deste último sobre o comando `grep` é sua rapidez.

Exemplo: Serão localizadas as linhas que começam com um caracter qualquer seguido das letras S ou P do arquivo arq1.txt.

```
$ egrep '^.(S|P)' arqu1.txt
"Software Livre" é uma questão
 Para entender o conceito,
```

Observe que foram usados apóstrofos ' para limitar a expressão procurada. Seu uso é importante para evitar erros de interpretação pelo *shell*, já que há caracteres com significado especial.

O caracter '^' foi usado para indicar o início da linha, o ponto . para indicar um caracter qualquer e a barra | foi usada para representar `ou` lógico. A expressão `^.(S|P)`, então, significa início de linha com qualquer caracter seguido das letras S ou P. As expressões com uso desses caracteres

especiais são chamadas de expressões regulares. A seção 3.3 tem mais sobre o uso de expressões regulares.

As principais opções do *grep* são listadas abaixo:

`-i` – Ignora a diferença entre maiúsculas e minúsculas

`-c` – Mostra o número de linhas em que a expressão foi encontrada

Exemplo: Voltando ao exemplo anterior, vamos ver em quantas linhas ocorreu a palavra liberdade.

```
$ grep -c liberdade *.txt
 arqu1.txt:2
     arqu2.txt:10
     nome.ord.txt:0
```

`-l` – Lista somente o nome dos arquivos que contêm a expressão

`-n` – Numera cada linha que contém a expressão procurada

`head` : **Mostra as linhas iniciais de um arquivo**

Sintaxe: `head [opções] <arquivo>`

As opções podem ser:

`-c m` – Mostra o texto que ocupa o tamanho de `m` bytes.

`-n m` – Mostra o texto das `m` primeiras linhas. Por padrão, são mostradas as 10 primeiras linhas se não for especificada nenhuma opção.

Exemplo:

```
$ head -c 23 arqu1.txt
  "Software Livre" é uma
$
```

`more` : **Mostra arquivos texto tela a tela**

Sintaxe: `more [opção] <arquivo>`

As principais opções são:

+n : exibe o arquivo a partir da linha n especificada.
-s : exibe múltiplas linhas em branco como sendo apenas uma.

As teclas associadas ao more são:

- <espaço>: para visualizar a próxima tela.
- <return>: para visualizar a próxima linha.
- <b>: para visualizar a tela anterior.
- <f>: para avançar uma tela.
- <q>: sair do comando.

`less` : **Mostra arquivos texto página a página com rolagem**

Sintaxe: `less <arquivo>`

Tem a mesma função do comando `more`.

Pipes

Os *Pipes* são a forma pela qual podemos construir uma conexão de dados entre a saída de um comando e a entrada de outro. A saída do comando anterior serve de entrada para o posterior.

Forma de um comando pipe: `comando1 | comando2 | ...`

Comandos conectados por *pipe* formam um *pipeline*.

Por exemplo: '`ls -la|more`'. Este comando faz a listagem longa de arquivos, que é enviada ao comando `more`. O comando `more` tem a função de efetuar uma pausa a cada 25 linhas do arquivo.

Comando `tee`

Envia o resultado do programa para a saída padrão do terminal e, ao mesmo tempo, para uma saída escolhida, por exemplo, um arquivo. Este comando deve ser usado com o comando *pipe*.

Exemplo:

```
ls -la|tee listagem.txt
```

A saída do comando será mostrada normalmente no terminal e, ao mesmo tempo, gravada no arquivo `listagem.txt`.

3.3 Expressões Regulares e Metacaracteres

Expressão Regular é uma seqüência de caracteres que simboliza diversas outras seqüências sem precisar listá-las. Nestas expressões, alguns caracteres recebem um significado especial, sendo chamados de metacaracteres. É através da utilização destes caracteres que podemos simbolizar uma quantidade enorme de palavras ou frases com uma expressão bem simples.

A Tabela 3.2 mostra alguns caracteres e seus significados. Os mais utilizados são os colchetes, o asterisco, o ponto e a interrogação.

Caracter	**Descrição**	**Exemplos**
.	Coringa de um caracter	vi.a
`[ ]`	Coincide com qualquer um dos caracteres listados	`[gpr]`ato
`[^]`	Coincide com qualquer um dos caracteres, exceto os listados	`[^mf]`ato
?	O caracter anterior pode aparecer ou não	meios?
*	O caracter anterior pode aparecer em qualquer quantidade	go*gle
+	O caracter anterior deve aparecer no mínimo uma vez	go+gle
{}	O caracter anterior deve aparecer na quantidade indicada	go`{1,5}`gle
`^`	Coincide com o começo da linha	`^`rio
$	Coincide com o fim da linha	mente$
`\b`	Limita uma palavra (letras, números e sublinhado)	`\b`(meu)
\	Torna os metacaracteres caracteres comuns.	sério`\?`
|	Atua como operador "ou".	(co|fu)mo
( )	Faz com que vários caracteres sejam vistos como um só.	(sai)?rei
.*	Qualquer caracter, em qualquer quantidade.	eu.*você
*?	Semelhante ao asterisco.	
+?	Semelhante ao sinal de mais.	
{}?	Semelhante às chaves simples.	

Tabela 3.2: Expressões Regulares

Capítulo 4

Introdução ao *script-shell* para LINUX

4.1 Aspectos básicos

4.1.1 Script e Script Shell

O *shell* é um interpretador de comandos que possui uma linguagem utilizada por diversas pessoas para facilitar a realização de inúmeras tarefas administrativas no Linux (como efetuar *backup* regularmente, procurar textos, criar formatações), e até mesmo para criar programas um pouco mais elaborados. A linguagem *shell* é interpretada, não havendo necessidade de compilar para gerar um arquivo executável.

Um ***script shell***, ou simplesmente ***script***, é um arquivo contendo uma seqüência de um ou mais comandos. Este arquivo é diretamente executável quando chamado pelo nome.

O *Shell* foi escrito em diferentes versões. Dos vários programas *Shell* existentes, o *Bourne Shell*, o *Korn Shell* e o *C Shell* se destacam por serem os mais utilizados e conhecidos.

O *Bourne Shell* é conhecido como *Shell* padrão, sendo o mais utilizado e estando na maioria dos sistemas *Unix like*.

O *Korn Shell* é uma versão melhorada do *Bourne Shell*.

O *C Shell* possui uma estrutura bastante parecida com a linguagem C e é também uma versão modificada do *Bourne Shell*.

Além desses, há um *shell* padrão do Linux, chamado *Bourne-Again Shell*. Este pode ser considerado o mais completo, sendo compatível com todos os *shells* citados anteriormente.

Mas qualquer programador pode fazer o seu *Shell*. Estes *shells* tornaram-se conhecidos pois já vinham com o sistema, exceto o *Korn*, que tinha que ser adquirido separadamente. O *Bourne Shell* vinha com o *System V* e o *C Shell* com o *BSD*.

Algumas características:

Shell		Prompt	Representação
Bourn Shell	⇒	\$	**sh**
Bourn-Again Shell	⇒	\$	**bash**
Korn Shell	⇒	\$	**ksh**
C shell	⇒	%	**csh**

4.2 Execução do programa

Um programa pode ser escrito em um editor de sua preferência como vi, kWrite, KEdit entre outros. O arquivo é salvo como texto comum. No ínicio do arquivo deve vir escrito:

`#!/bin/bash`

Os caracteres especiais #! (chamados *hash-bang*) informam ao kernel que o próximo argumento é o programa utilizado para executar este arquivo. No caso, `/bin/bash` é o shell que utilizamos. O kernel lê o *hash-bang* no início da linha, então ele continua lendo os caracteres seguintes e inicia o *bash*. Quando o *shell* lê o *hash-bang*, ele o interpreta como uma linha de comentário e a ignora, iniciando a execução do programa.

É preciso mudar a permissão do arquivo para executável para ele funcionar. Isso é feito pelo comando `chmod`. Para que o programa seja executável de qualquer parte do sistema é necessário salvá-lo em algum diretório que esteja no PATH (variável do sistema que contém a lista de diretórios onde o *shell* procura pelo comando digitado). Entretanto, é permitido salvá-lo em um diretório qualquer (como o diretório home do usuário), porém na hora de executá-lo pelo prompt, é necessário que seja indicado todo o caminho desde a raiz, ou, estando no mesmo diretório do arquivo, digitar no prompt: ./nomearquivo.

OBS: Também é possível modificar o PATH para incluir um diretório de sua escolha.
Por exemplo: vamos supôr que o usuário criou uma pasta em seu diretório para salvar seus scripts com o nome de `scripts`. Para tornar esse diretório como parte do PATH faça o seguinte:

Digite no prompt:

$ `echo $PATH`
Esse é seu PATH atual. Em seguida digite:
$ `PATH=$HOME/scripts:$PATH`
e
$ `echo $PATH`

Esse é seu novo PATH, com seu diretório de exemplos incluído. Agora seus scripts podem ser executados de qualquer diretório. Porém, dessa forma, a mudança da variável PATH só vale enquanto o *shell* corrente estiver aberto. Se for aberto outro *shell*, a mudança não terá efeito.

Existem arquivos que são inicializados assim que o *shell* é aberto:

`/etc/profile` : Tem as configurações básicas do sistema e vale para todos os usuários. Somente o *root* tem permissão para modificar esse arquivo.

`.bash_profile` ou `.bash_login` ou `.profile` ou `.bashrc` : Estes arquivos ficam no diretório home do usuário. As modificações feitas nesse arquivo só valem para o próprio usuário. Podemos, então, abrir o arquivo `.bashrc` e colocar nele o novo PATH. Além disso, podemos incluir também *aliases*.

Ex: Se tivéssemos um diretório chamado `scripts` e quiséssemos colocá-lo no PATH, bastaria acrescentar a linha abaixo ao arquivo `.bashrc`. Também foram colocados alguns *aliases*.

```
PATH=$PATH:~/scripts
alias c='clear'
alias montar='mount /dev/fd0'
```

Vamos fazer um exemplo passo-a-passo agora para criar e executar um programa.
Escreva em seu editor de texto o programa abaixo e salve como `um.sh`.

```
#!/bin/bash

echo "Programa UM!"
```

Agora no prompt, mude as permissões do arquivo.

```
$ chmod a+x um.sh
```

Agora é só chamar o programa no prompt.

```
$ sh um.sh
```

Neste caso o arquivo estava no mesmo diretório de trabalho. E se o arquivo estivesse em um diretório diferente? Lembre-se de mudar o PATH!

4.2.1 Erros na execução

Para o usuário iniciante, é bem provável que ele se depare com alguns erros bem comuns e fáceis de resolver. Os principais são:

- "*command not found* - Esse quer dizer que o shell não encontrou seu programa.

  A razão para isso pode ser que o nome do comando foi digitado de forma diferente do nome do arquivo. Certifique-se de que o nome está igual. Outra razão possível é que o arquivo está em um local diferente do PATH padrão. Nesse caso, deve-se proceder conforme explicado na seção anterior, que explica como salvar o arquivo.

- "*Permission denied*- A permissão para execução do arquivo foi negada. O usuário deve mudar a permissão do arquivo para executável.

- Outro erro comum é o de sintaxe. Nesse caso o *shell* encontra o *script* e indica a linha onde ocorreu o erro no programa.

4.2.2 *Quoting*

Denomina-se *quoting* a função de remover a predefinição ou significado especial dos metacaracteres. Dessa forma, é possível escrever os caracteres literalmente.
Há 3 tipos de mecanismos para *quoting*:

1. **Barra invertida** (\) - Chamada de caracter de escape, ela remove o significado especial do caracter seguinte à ela.

   Exemplo:

```
$ echo 0 # é um caracter especial
  0
$ echo 0 \# é um caracter especial
  0 # é um caracter especial
```

Observe a diferença que o uso da barra provoca.

```
$ echo agora a barra está sendo\
> usada para que seja possível\
> continuar escrevendo em outra linha\
> Veja que apareceu um prompt secundário \
> Mas na hora da impressão as linhas saem seguidas.
 agora a barra está sendousada para que seja possívelcontinuar
 escrevendo em outra linhaVeja que apareceu um prompt
 secundário Mas na hora da impressão as linhas saem seguidas.
```

Neste exemplo a barra suprimiu o significado de fim de linha.

2. **Aspas simples** (') - Todos os caracteres que vem entre estas aspas tem seu significado removido.

   Exemplo:

```
$ echo 'Com aspas simples não é necessário colocar\
> barra para cada caracter # $ * !'
  Com aspas simples não é necessário colocar barra para
  cada caracter # $ * !
```

3. **Aspas duplas** (") - É semelhante à anterior, porém não remove o significado de $, \, ", ' e {variável}.

4.3 Comentários

Para deixar seu programa mais claro e fácil de entender, o usuário não só pode como deve acrescentar comentários em seu código-fonte utilizando o caracter # no início da linha. Esta linha não será executada pelo *shell* quando ele o encontrar. Isso é bastante útil para permitir a leitura posterior do arquivo, correção de erros, mudanças no programa entre outras tarefas.

Por exemplo, no início de um *script* poderia ser escrita a sua função.

```
# Script para administrar as contas dos novos usuários

 <comandos>
```

Um bom código-fonte deve ter um cabeçalho identificando o autor e as funções do programa. É importante lembrar que, independentemente da linguagem usada, comentários são essenciais para um bom entendimento do programa escrito, tanto pelo autor como por outros usuários.

Abaixo segue uma lista de recomendações que um programador deve seguir:

- Escreva os comentários no momento em que estiver escrevendo o programa.
- Os comentários devem acrescentar algo e não apenas descrever o funcionamento do comando.
- Utilize espaços em branco para aumentar a legibilidade.
- Coloque um comando por linha, caso a situação permita.
- Escolha nomes representativos para as variáveis.

4.4 Impressão na tela

O comando `echo` permite mostrar na tela seus argumentos.

Ex:

```
$echo Escrevendo seu argumento
 Escrevendo seu argumento
```

Existem caracteres especiais que são interpretados pelo comando `echo`. Em algumas versões do Linux deve ser usado o parâmetro opcional `-e`. Esta opção habilita a interpretação dos caracteres de escape listados abaixo.

- `\\` - Barra invertida (*backslash*).
- `\nnn` - Escreve o caracter cujo octal é nnn.
- `\xHH` - Escreve o caracter cujo hexadecimal é HH.
- `\a` - Caracter de alerta sonoro (*beep*).
- `\b` - *Backspace*.
- `\c` - *Suprime newline*, forçando a continuação na mesma linha.
- `\f` - Alimentação de formulário.
- `\n` - Inicia um nova linha.
- `\r` - *Carriege return*. Retorna ao início da linha.
- `\t` - Equivale a um espaço de tabulação horizontal.
- `\v` - Equivale a um espaço de tabulação vertical.

Ex:

```
$ echo -e "Este exemplo mostra o uso de alguns dos caracteres mostrados:
> Começando pela contra-barra \\
> Caracter hexadecimal \100
> Usando backspace\b
> iniciando \n nova linha
> apagando o que foi \r escrito anteriormente na linha
> e tabulando \t horizontalmente e \v verticalmente."
Este exemplo mostra o uso de alguns dos caracteres mostrados:
Começando pela contra-barra \
Caracter hexadecimal @
Usando backspace
iniciando
 nova linha
 escrito anteriormente na linha
e tabulando      horizontalmente e
                                   verticalmente.
```

OBS: Para saber a representação octal e hexadecinal correspondente ao caracter desejado, consulte a tabela ASCII no manual: man ascii.

Quando o usuário não desejar que a saída do comando `echo` pule de linha, ele deve usar a opção `-n`.

4.5 Passagem de parâmetros e argumentos

Parâmetros são variáveis passadas como argumentos para o programa ou variáveis "especiais"que guardam certas informações.

Quando um programa recebe argumentos em sua linha de comando, estes são chamados de **parâmetros posicionais**. Eles são numerados de acordo com a ordem em que foram passados.

Exemplo:

```
$ cat local.sh
#Programa recebe e mostra dados

echo Cidade: $1
echo Estado: $2
echo País:   $3

$ ./local.sh Niterói "Rio de Janeiro" Brasil
Cidade: Niterói
Estado: Rio de Janeiro
País: Brasil
```

Como pode ser visto no exemplo, o programa chamado `local` recebe 3 argumentos. O primeiro, `Cidade`, é guardado na variável $1, o segundo, `Estado`, em $2 e o terceiro, `País`, em $3. Porém, o shell limita em 9 o número de argumentos. Para trabalhar com essa limitação, existe o comando `shift n` que faz com que os primeiros `n` argumentos passados não façam parte na contagem de argumentos.

Exemplo: Serão passados mais de 9 argumentos para o programa, mas através do uso do comando `shift` será possível listar todos.

```
$ cat shift.sh
#Programa recebe e mostra dados usando shift

echo Argumento1: $1
echo Argumento2: $2
echo Argumento3: $3
echo Argumento4: $4
echo Argumento5: $5
echo Argumento6: $6
echo Argumento7: $7
echo Argumento8: $8
echo Argumento9: $9
shift 9
echo Argumento10: $1
echo Argumento11: $2

$ ./shift.sh a b c d e f g h i j l m n
Argumento1: a
Argumento2: b
Argumento3: c
Argumento4: d
Argumento5: e
Argumento6: f
Argumento7: g
Argumento8: h
Argumento9: i
Argumento10: j
Argumento11: l
```

Há também os **parâmetros especiais** que podem ser usados pelo programa:

$ * - Este parâmetro informa uma string com todos os argumentos passados para o programa, onde cada argumento aparece separado pelo IFS (veja a seção 5.8, que fala de variáveis de sistema).

$ @ - Este parâmetro é semelhante ao anterior, porém os argumentos aparecem separados por espaços em branco.

$# - Este informa o número de argumentos passados na chamada do programa.

$? - Este guarda o valor de retorno do último comando executado. Quando a execução do programa acontece normalmente, é retornado 0, e se tiver ocorrido algum erro é retornado um valor diferente de zero.

$$ - Informa o número do processo de um determinado programa (PID).

$! - Informa o número do processo do último programa sendo executado em *background*.

$0 - Informa o nome do *shell script* executado.

Exemplo: Será visto o uso de alguns desses parâmetros especiais.

```
$ cat parametros.sh
# Mostra os dados relativos aos parametros passados

echo Foram passados $# argumentos.
echo Os argumentos foram: $@.

$ ./parametros.sh um dois três quatro
Foram passados 4 argumentos.
Os argumentos foram: um dois três quatro.
```

4.5.1 Leitura de parâmetros

Quando for necessário passar algum dado para o programa, usa-se o comando `read`, que lê a entrada escrita no terminal.

Exemplo:

```
$ cat read.sh
# uso do comando read

echo Digite uma palavra:
read algo
echo Você digitou: \"$algo\"

$ ./read.sh
Digite uma palavra:
Por que não uma frase?
Você digitou: "Porque não uma frase?"
```

Este comando permite ainda a passagem de uma lista de variáveis, desde que estas venham separadas entre espaços em branco.

Exemplo: Agora o programa `read` sofreu uma modificação para receber uma lista de variáveis.

```
$ cat read.sh
# uso do comando read

echo Digite umas palavras:
read prim segun ter
echo Você digitou:
```

```
echo      \"$prim\"
echo      \"$segun\"
echo      \"$ter\"

$ ./read.sh
Digite umas palavras:
Mas agora eu quero digitar uma frase!
Você digitou:
"Mas"
"agora"
"eu quero digitar uma frase!"
```

Se forem passadas mais variáveis do que o comando `read` vai ler, a variável excedente é interpretada como se fizesse parte da última variável. Como pode ser visto pelo exemplo anterior, onde foram passadas mais variáveis do que as 3 que o programa leria. Então, a terceira variável ficou com uma frase.

Algumas opções podem ser usadas com o comando `read`. São elas:

- `-p` – Nos exemplos acima foi preciso usar o comando `echo` toda vez antes do comando `read`. Porém, isso não é necessário, basta usar esta opção. O próximo exemplo mostra como.

  Exemplo:

  ```
  $ cat read.sh
  # uso do comando read

  read -p "Digite umas palavras:" prim segun ter
  echo Você digitou:
  echo      \"$prim\"
  echo      \"$segun\"
  echo      \"$ter\"

  $ ./read.sh
  Digite umas palavras:Dessa vez o echo não foi usado antes
  Você digitou:
  "Dessa"
  "vez"
  "o echo não foi usado antes"
  ```

- `-s` – Esse parâmetro serve para não ecoar o que foi digitado. Seu uso principal é na leitura de senhas.

- `-n` – Este parâmetro permite limitar o número de caracteres que serão lidos pelo `read`. Sua sintaxe é: `read -n N string`. Lerá apenas os N caracteres da *string* digitada.

4.6 Funções

Funções são estruturas que reúnem comandos na forma de blocos lógicos, permitindo a separação do programa em partes. Quando o nome de uma função é chamado dentro do *script* como um comando, todos os comandos associados a esta função são executados.

A sintaxe para o uso de funções é da forma:

```
function nome () {
    ...
    <comandos>
    ...
}
```

Onde `nome` é o nome que será dado à função criada. E `comandos` definem o corpo da função.

A principal vantagem de usar funções é a possibilidade de organizar o código do programa em módulos.

O exemplo a seguir demonstra o uso de funções em *script*. Um menu de opções é mostrado, sendo que cada opção leva à execução de uma função diferente.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#
#
#

clear
menu()
  {
     echo "         Calculadora Básica          "
     echo "     Operação apenas com inteiros    "
     echo "|-----------------------------------|"
     echo "|   Escolha uma das opções abaixo:  |"
     echo "|-----------------------------------|"
     echo "|  1) Soma                          |"
     echo "|  2) Subtração                     |"
     echo "|  3) Multiplicação                 |"
     echo "|  4) Divisão                       |"
     echo "|  5) Sair                          |"
     echo "|-----------------------------------|"
     echo "|-----------------------------------|"

  read opcao
  case $opcao in
```

```
     1) soma ;;
     2) subtra ;;
     3) mult ;;
     4) div ;;
     5) exit ;;
     *) "Opção Inexistente" ;
         clear ;
     menu ;;
  esac
     }



soma()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
     echo "Resposta = ‘expr $num1 "+" $num2‘"
     menu

  }

subtra()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "-" $num2‘"
     menu
  }

mult()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "*" $num2‘"
     menu
```

```
  }

div()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "/" $num2‘"
     menu
  }

menu
```

Repare no *script* acima que a função `menu` foi colocada no final do programa. Experimente chamar a função `menu` no início e veja o que acontece.

4.6.1 Execução de *script* por outro *script*

É possível executar um script de outro arquivo com se ele fosse uma função qualquer dentro de outro programa. Para isso, basta escrever: . `nomescript` no lugar onde ele deve ser executado.

Tendo como base o exemplo anterior, podemos mostrar a execução de *scripts* dentro de outros.

O primeiro programa é o *script* principal referente ao `menu` de opções. Observe como as operações são chamadas.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#  PARTE -> MENU
#
#

clear


menu(){

     echo "         Calculadora Básica          "
     echo "     Operação apenas com inteiros    "
     echo "|-----------------------------------|"
     echo "|   Escolha uma das opções abaixo:  |"
     echo "|-----------------------------------|"
```

```
      echo "|  1) Soma                          |"
      echo "|  2) Subtração                     |"
      echo "|  3) Multiplicação                 |"
      echo "|  4) Divisão                       |"
      echo "|  5) Sair                          |"
      echo "|-----------------------------------|"
      echo "|-----------------------------------|"

  read opcao
  case $opcao in
     1) . soma.sh ;;
     2) . subtra ;;
     3) . mult ;;
     4) . div ;;
     5) exit ;;
     *) "Opção Inexistente" ;
         clear ;
     menu ;;
  esac

}
menu
```

O *script* seguinte refere-se a operação de soma.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#  PARTE -> SOMA

soma()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
     echo "Resposta = `expr $num1 "+" $num2`"
     menu

  }
  soma
```

4.7 Depuração

Para verificar os problemas e possíveis causas de erros que acontecem nos *scripts*, basta rodar o programa da seguinte forma:

```
$ sh -x programa
```

A opção -x faz com que o programa seja executado passo-a-passo, facilitando a identificação de erros.

Capítulo 5

Manipulação de variáveis

Variável é uma posição nomeada de memória, usada para guardar dados que podem ser manipulados pelo programa.

Em *shell* não é necessário declarar a variável como em outras linguagens de programação.

5.1 Palavras Reservadas

Palavras reservadas são aquelas que possuem um significado especial para o *shell*.

O *Shell* possui comandos próprios (intrínsecos) como:

```
! case   do  done  elif  else  esac  fi  for  function  if  in
select  then   until while   {    }    time   [[   ]]
```

Além disso, o Unix possui outros comandos, vistos nos capítulos anteriores.

Em programação, geralmente trabalhamos com manipulação de variáveis. Dessa forma, é importante lembrar que o uso dessas palavras deve ser evitado, tanto no nome dado às variáveis quanto no nome dado ao *script*.

5.2 Criação de uma variável

Uma variável é criada da seguinte forma:

```
$  nomevar=valor
```

Uma variável é reconhecida pelo *shell* quando ela vem precedida pelo símbolo $. Quando este símbolo é encontrado, o *shell* substitui a variável pelo seu conteúdo. `nomevar` é o nome da variável e *valor* é o conteúdo que será atribuído à variável. É importante assegurar que não haja espaço antes e depois do sinal "="para evitar possíveis erros de interpretação. No exemplo acima, foi criada uma variável local. Para criar um variável global, é usado o comando `export`.

$ `export nomevar`

ou

$ `export nomevar`=valor

Exemplo: Será atribuído um valor à variável chamada `var`, e em seguida este valor será mostrado pelo comando `echo`.

```
$ var=Pensamento
$ echo "O conteúdo é o $var"
O conteúdo é o Pensamento
```

Vale lembrar algumas regras para a nomenclatura de variáveis que se aplicam às linguagens de programação:

- O nome da variável só pode começar com letras ou *underline*.
- São permitidos caracteres alfanuméricos.
- Não devem haver espaços em branco nem acentos.

  Exemplo: Veja o uso de aspas simples, duplas e crases com variáveis:

```
$ variavel="Meu login é: $user"
$ echo $variavel
 Meu login é: ze
$ variavel='Meu hostname é: $HOSTNAME'
$ echo $variavel
 Meu hostname é: $HOSTNAME
$ variavel="O diretorio de trabalho  é: `pwd`"
$ echo $variavel
 O diretorio de trabalho é: /home/ze
```

Quando vamos executar um script ou comando, um outro *shell* é chamado, executa os comandos e então retorna ao *shell* original onde foi feita a chamada. Por isso é importante lembrar de exportar suas variáveis para que elas sejam reconhecidas pelo "*shell* filho".

```
$ cat dir.sh
#!/bin/bash

echo "Veja que o diretório vai mudar"
echo "Inicialmente o diretório era: $PWD"
# neste ponto o diretorio mudou
cd /usr/bin
echo "Agora o diretório atual é $PWD"
echo "Mas terminado o programa parece que nada aconteceu.
O diretório continua sendo o inicial."

$ sh dir.sh
Veja que o diretório vai mudar
Inicialmente o diretório era: /home/kurumin/scripts
Agora o diretório atual é /usr/bin
Mas terminado o programa parece que nada aconteceu.
O diretório continua sendo o inicial.
$
```

5.3 Deleção de uma variável

Uma variável é apagada quando for usado o comando `unset`.

`$unset nomevar`

Exemplo: Vamos ver o que acontece quando a variável `var`, criada anteriormente, for deletada.

```
$ unset var
    $ echo "O conteúdo é o $var"
    O conteúdo é o
```

5.4 Visualização de variáveis

Utilizando o comando `set` é possível visualizar as variáveis locais e com o comando `env` as variáveis globais podem ser vistas.

5.5 Proteção de uma variável

Para evitar alterações e deleção de uma determinada variável usa-se o comando `readonly`. Dessa forma, a variável ganha atributo de somente leitura.

`$ readonly nomevar`

Todas as variáveis readonly, uma vez declaradas, não podem ser "unsetadas"ou ter seus valores modificados. O único meio de apagar as variáveis readonly declaradas pelo usuário é saindo do shell (logout).

5.6 Substituição de variáveis

Além de substituição de variáveis (variável pelo seu conteúdo, visto em exemplos anteriores), outros tipos de substituições são possíveis no *shell*. As principais são:

- **Substituição de comando** - Nesse caso, o nome do comando é substituído pelo resultado de sua operação quando ele for precedido pelo símbolo $ e entre () ou vier entre sinais de crase (‘).

```
$(comando)

ou

‘comando‘
```

Geralmente este tipo de manipulação é utilizada na passagem do resultado de um comando para uma variável ou para outro comando.

Exemplo:

```
$ dir=‘pwd‘
    $ echo "O diretório atual tem o seguinte caminho: $dir"
    O diretório atual tem o seguinte caminho: /home/kurumin/scripts
```

5.7 Variáveis em vetores

O *shell* permite o uso de variáveis em forma de *array*. Ou seja, vários valores podem ser guardados em uma variável seguindo a ordem de uma indexação. Assim como não é necessário declarar o tipo de variável no início do programa, também não é preciso declarar que uma variável será usada como vetor.

Um array é criado automaticamente se for atribuído um valor em uma variável da seguinte forma: `nomevar[indice]=valor`, onde índice é um número maior ou igual a zero.

Exemplo:

```
$ camada[0]=Física
$ camada[1]=Enlace
$ camada[2]=Redes
$ echo "As 3 camadas mais baixas da internet são: ${camada[*]}"
As 3 camadas mais baixas da internet são: Física Enlace Redes
```

Outra forma de se atribuir valores em *array* é:
`nomevar=(valor1, valor2, ..., valorn)`.

```
$ camada=(Física, Enlace, Redes)
$ echo "As 3 camadas mais baixas da internet são: ${camada[*]}"
As 3 camadas mais baixas da internet são: Física, Enlace, Redes
```

Para fazer referência a um elemento do array é só fazer: `${nomevar[indice]}`.

Para ver toda a lista de valores do *array* basta fazer `${nomevar[*]}` ou `${nomevar[@]}`. A diferença entre o uso do * ou @ é semelhante a diferença vista no uso de parametros especiais.

5.8 Variáveis do sistema

Existem algumas variáveis que são próprias do sistema e outras que são inicializadas diretamente pelo *shell*.

Algumas dessas variáveis, denominadas **variáveis do shell** são explicadas abaixo:

HOME – Contém o diretório *home* do usuário.

LOGNAME – Contém o nome do usuário atual.

IFS – Contém o separador de campos ou argumento (*Internal Field Separator*). Geralmente, o IFS é um espaço, tab, ou nova linha. Mas é possível mudar para outro tipo de separador.

Exemplo:

```
$ num=(1 2 3 4 5)
$ echo "${num[*]}"
1 2 3 4 5
$ echo "${num[@]}"
1 2 3 4 5
$ OLDIFS=$IFS
$ IFS='-'
$ echo "${num[*]}"
1-2-3-4-5
$ echo "${num[@]}"
1 2 3 4 5
$ echo $IFS

$ echo "$IFS"
-
$ IFS=$OLDIFS
$ echo "$IFS"
```

Vamos entender o que foi feito: foi criado um vetor para ilustrar o IFS quando forem usados os caracteres e * para listar o *array*. O IFS inicial é um espaço em branco, então tanto pelo uso do como pelo uso do *, os elementos foram listados separados por um espaço em branco. Em seguida uma variável chamada OLDIFS recebeu o conteúdo de IFS e IFS recebeu um hífen (-). A separação na listagem dos elementos saiu diferente, ou seja, o novo separador foi usado quando foi usado o asterisco. Finalmente, o IFS recebeu seu valor inicial, espaço em branco. Essa mudança permanece somente na seção em que foi modificada e até que ela seja fechada.

PATH – Armazena uma lista de diretórios onde o shell procurará pelo comando digitado.

PWD – Contém o diretório corrente.

PS1 – Esta é denominada *Primary Prompting String*. Ou seja, ela é a string que está no *prompt* que vemos no *shell*. Geralmente a string utilizada é: `\u@\h:\w $`. O significado desses caracteres e de outros principais está explicado abaixo:

- `\s` O nome do shell.
- `\u` Nome do usuário que está usando o shell.
- `\h` O *hostname*
- `\w` Contém o nome do diretório corrente desde a raiz.
- `\d` Mostra a data no formato: `dia_da_semana mês dia`.
- `\nnn` Mostra o caracter correspondente em números na base octal.
- `\t` Mostra a hora atual no formato de 24 horas, HH:MM:SS.
- `\T` Mostra a hora atual no formato de 12 horas, HH:MM:SS.

- `\W` Contém somente o nome do diretório corrente.

PS2 – Esta é denominada *Secondary Prompting String*. Ela armazena a string do *prompt* secundário. O padrão usado é o caracter `>`. Mas pode ser mudado para os caracteres mostrados acima.

MAIL – É o nome do arquivo onde ficam guardados os e-mails.

COLUMNS – Contém o número de colunas da tela do terminal.

LINES – Contém o número de linhas da tela do terminal.

Existem muitas outras variáveis que são descritas na página do manual do *Bash*, na seção *Shell Variables*.

Capítulo 6

Testes e Comparações em *Script-Shell*

6.1 Código de retorno

Antes de falar sobre testes e comparações é importante que o usuário entenda como as decisões são tomadas dentro de um programa.

Todo comando do UNIX retorna um código e este é chamado *código de retorno*. Quando o comando é executado sem erros, o código retornado vale 0. Porém, se houver alguma falha, é retornado um número diferente de 0.

O caracter especial ? funciona como uma variável que guarda o código de retorno do comando anterior. O exemplo abaixo mostra o resultado da operação de alguns comandos.

Exemplo: Quando acontece algum erro na execução do comando, o código de retorno é diferente de zero.

```
$ echo "$PS1"
\u@\h:\w\$
$ echo $?
0
$ rm documento.txt
rm: cannot lstat ‘documento.txt’: No such file or directory
$ echo $?
1
```

6.2 Avaliação das expressões

As expressões são avaliadas no *shell* através do comando `test` ou pelo uso da expressão entre colchetes [ ], uma maneira mais prática do uso do comando `test`.

```
$ var=Z
$ test $var = w
$ echo $?
1
$ [ $var = Z ]
$ echo $?
0
```

O resultado da expressão é retornado 0 para verdadeiro ou não 0 para falso.

6.3 Operadores *booleanos*

Os operadores *booleanos* são relacionados à lógica `e`, `ou`, `negação` entre outras relações. Os operadores que podem ser usados em expressões no *shell* são:

- `-a` - e (*and*).
- `-o` - ou (or).
- `!` - negação (not).

A combinação desses 3 operadores pode gerar outras funções lógicas.
Várias condições também podem ser agrupadas com o uso de *condio*".
Ex:

```
$ cat boole.sh
#!/bin/bash

read -p "Informe um número e uma letra: " num letra

if [ \( "$num" -gt 0 -a "$num" -lt 10 \) -o \( $letra = v \) ]
 then
   echo "Acertou a faixa do numero ou a letra."
 else
   echo "Errou as duas informações."
fi
$ sh boole.sh
Informe um número e uma letra: 15 v
Acertou o faixa do numero ou a letra.
$
```

6.4 Testes Numéricos

As relações utilizadas para testes numéricos são as descritas abaixo:

- `-eq` – Igual à (*equal to*).
- `-gt` – Maior que (*greater than*).
- `-ge` – Maior ou igual (*greater or equal*).
- `-lt` – Menor que (*less than*).
- `-le` – Menor ou igual (*less or equal*).
- `-ne` – Não-igual a (*not equal to*).

A sintaxe para teste é:
`[ var/número relação var/número ]`
onde `var/número` indica o conteúdo da variável ou um número.
Exemplo:

```
$ num=10
$ [ $num -eq 9 ]; echo $?
1
$ [ $num -le 9 ]; echo $?
1
$ [ $num -ge 9 ]; echo $?
0
```

Outra maneira de realizar o teste é colocando `var/número` entre aspas. Dessa forma evitamos a ocorrência de erros.
`[ "var/número" relação "var/número" ]`
Exemplo: Observe o que acontece quando a expressão é comparada com o valor nulo sem aspas

```
$ [ $num -ge  ]; echo $?
bash: [: 10: unary operator expected
2
$ [ "$num" -ge " " ]; echo $?
0
```

6.4.1 O Comando `let`

Este comando permite outra maneira de fazer testes numéricos. Em vez de usar as relações citadas anteriormente, são usados os símbolos:

- == – Igual à
- > – Maior que
- >= – Maior ou igual
- < – Menor que
- <= – Menor ou igual
- != – Não-igual à

A sintaxe é:
`let expressão`
Exemplo:

```
$ let "0 != 1"
$ echo $?
0
```

Uma variação desse comando é o uso de parênteses duplo:
`((expressão))`

```
$ ((0 <= 5)) ; echo $?
0
```

Essa é uma sintaxe semelhante a da linguagem C. Outro uso comum é no incremento de variáveis:

```
let var++  # equivalente a "var=$[ $var + 1 ]"

let var--  # equivalente a "var=$[ $var - 1 ]"
```

Exemplo:

```
$ num=102
$ let num++
$ echo $num
103
$ num=$[$num + 1]
$ echo $num
104
$ num=$((num+1))
$ echo $num
105
$ let num=num+1
$ echo $num
106
$ let num+=4
$ echo $num
110
```

6.5 Testes de *Strings*

O tamanho de uma *string* pode ser obtido pelo uso do comando `expr length` *`string`*.
Ex:

```
$ expr length palavra
7
$
```

Ex:

```
#!/bin/bash

echo "Digite a senha: "
read -s  senha
```

```
comp=$(expr length $senha)

if [ "$comp" -lt 6 -o "$comp" -gt 9 ]
 then
    echo "Senha Inválida."
    echo "Por segurança não são aceitas senhas com menos de
    6 caracteres ou mais que 9."
    echo "Informe uma nova senha."
 else
    echo "Senha aceita."
fi
```

Os operadores para testes de *string* podem ser:

Binários

- `=` - Retorna verdadeiro se as duas *strings* forem iguais.

```
$ str=palavra
$ [ "$str" = "cadeia" ]
$ echo $?
1
```

- `!=` - Retorna verdadeiro se as duas *strings* forem diferentes.

Unários

- `-z` - Retorna verdadeiro se o comprimento da *string* é igual à 0.
- `-n` - Retorna verdadeiro se o comprimento da *string* é diferente de 0.

```
$ [ -n $str ]
$ echo $?
0
```

A sintaxe para a comparação de *string* segue o mesmo modelo para a comparação numérica:
`["var/string" relação "var/string"]`
A preferência para o uso de aspas foi dada porque geralmente as *strings* contêm espaços em branco. Assim, são evitados erros de interpretação pelo *shell*.

6.6 Testes de arquivos

Sempre que vamos trabalhar com arquivos é necessário realizar testes como os mesmos para evitar erros. Para testar arquivos existem as opções:

- -d – O arquivo é um diretório.
- -e – O arquivo existe.
- -f – É um arquivo normal.
- -s – O tamanho do arquivo é maior que zero.
- -r – O arquivo tem permissão de leitura.
- -w – O arquivo tem permissão de escrita.
- -x – O arquivo tem permissão de execução.

A sintaxe usada deve ser:
`[ opção arquivo ]`
Ex:

```
$ ls -l  arqu1.txt data
-rwxrwxrwx    1 pet  linux       161   2005-04-09 21:42   arqu1.txt
-rw-r--r--    1 pet  linux        95   2005-04-07 22:00   data
drwxr-xr-x    3 pet  linux      1312   2005-01-07 22:57   scripts
$ [ -d scripts ]
$ echo $?
0
$ [ -d arqu1.txt ]
$ echo $?
1
$ [ -e arqu1.txt ]
$ echo $?
0
$ [ -f arqu1.txt ]
$ echo $?
0
$ [ -r data ]
$ echo  $?
0
$ [ -x data ]
$ echo $?
1
```

Capítulo 7

Controle de fluxo

Para as linguagens de programação, uma das mais importantes estruturas é o controle de fluxo. Com o *shell* não é diferente. Com ele, a execução do programa pode ser alterada de forma que diferentes comandos podem ser executados ou ter sua ordem alterada de acordo com as decisões tomadas. São realizados saltos, desvios ou repetições. Nas próximas seções explicaremos cada tipo de estrutura.

7.1 Decisão simples

A estrutura de decisão simples é aquela que realiza desvios no fluxo de controle, tomando com base o teste de uma condição dada, uma opção entre duas que podem ser escolhidas.

Uma decisão simples é uma construção com os comandos `if/then`. Isso representa **se** condição **então** realiza determinado comando.

Sintaxe:

```
if[expressão]; then

    comando
fi
```

Ex: Este programa bem simples mostra o uso do **if**. Há um arquivo chamado `livro.txt` cujo conteúdo segue abaixo:

```
#LIVRO          #EXEMPLARES
##########################
eletromagnetismo 5
redes            4
cálculo          8
física           6
eletrônica       7
```

O programa abaixo mostra o número de exemplares de um determinado assunto. Mas, primeiro, é verificado se o livro está na lista.

```
#!/bin/bash

echo -n  "Qual livro você deseja? "
```

```
read Livro

if grep $Livro livro.txt>>/dev/null
  then echo "O livro $Livro possui `grep $Livro livro.txt|cut -f2` exemplares."

  else echo "Este livro não está na lista"

fi
```

Obs: O diretório `/dev/null` é um lugar para onde redirecionamos a saída de um comando quando não é desejável que ela apareça no *prompt*.

7.2 Decisão múltipla

Este tipo de estrutura engloba, além dos comandos vistos anteriormente, os comandos `elif` e `else`. Neste caso, se a condição dada não for satisfeita, há outro caminho a ser seguido, dado pelo `elif`, que seria a combinação de **else** com **if** (senão se...). A diferença entre o uso de `elif` e `else if` é que, se fosse usado o último, seria necessário usar o `fi`.

Sintaxe:

```
if [expressão]; then

    comando

elif [expressão]; then

    comando

elif [expressão]; then

    comando
...

else

    comando

fi
```

7.2.1 O comando `case`

Outra forma de fazer desvios múltiplos é pelo uso do comando `case` . Ele é semelhante ao `if` pois representa tomada de decisão, mas permite múltiplas opções. Esta estrutura é bastante usada quando é necessário testar um valor em relação a outros valores pré- estabelecidos, onde cada um desses valores tem um bloco de comando associado.

Ex: Este exemplo dá o Estado de acordo com o DDD digitado.

```
echo "Insira o código DDD: "

read cod

case $cod in
  21) echo "Rio de Janeiro";;
  11) echo "São Paulo";;
  3[0-8]) echo "Minas Gerais";;
  *) Echo "Insira outro código";;

esac
```

Decisão com && e || Esses caracteres permitem a tomada de decisões de uma forma mais reduzida. A sintaxe usada é:

```
comando1 && comando2

comando1 || comando2
```

O && faz com que o `comando2` só seja executado se `comando1` retornar 0, ou seja, se `comando1` retornar verdadeiro o `comando2` é executado para testar se a condição toda é verdadeira.

O || executa o `comando2` somente se o `comando1` retornar uma valor diferente de 0, ou seja, somente se `comando1` retornar falso é que `comando2` será executado para testar se a condição toda é verdadeira.

Ex: Se o arquivo `livro.txt` realmente existir será impresso na tela: O arquivo existe.

```
[ -e livro.txt ] && echo "Arquivo Existe"
```

Ex: Se o diretório `NovoDir` não existir é criado um diretório com o mesmo nome.

```
cd NovoDir 2> /dev/null || mkdir NovoDir
```

7.3 Controle de *loop*

Existem 3 tipos de estruturas de *loop*, que serão vistas na próxima seção. Esse tipo de estrutura é usada quando é preciso executar um bloco de comandos várias vezes.

7.3.1 *While*

Nesta estrutura é feito o teste da condição, em seguida ocorre a execução dos comandos. A repetição ocorre enquanto a condição for verdadeira.

```
while <condição> do

    <comandos>

done
```

`condição` pode ser um teste, uma avaliação ou um comando.
Ex:

```
#!/bin/bash

echo "Tabela de Multiplicação do 7: "

i=7;
n=0;
while [ $n -le 10 ]
do
  echo $i x $n = $(($i * $n))
  let n++
done
```

7.3.2 *Until*

Neste caso, a repetição ocorre enquanto a condição for falsa. Ou seja, primeiramente a condição é testada, se o código de retorno for diferente de zero os comandos são executados. Caso contrário, a execução do programa continua após o *loop*.

```
until <condição> do

    <comandos>

done
```

`condição` pode ser um teste, uma avaliação ou um comando.
Ex: Este exemplo é semelhante ao exemplo anterior do comando `while`. O que mudou foi a condição de teste.

```
#!/bin/bash

echo "Tabela de Multiplicação do 7: "

 i=7;
n=10;
until [ $n -eq 0 ]
do
   echo $i x $n = $(($i * $n))
   let n--
done
```

7.3.3 *For*

A sintaxe da estrutura `for` é a seguinte:

```
for variavel in lista do

    <comandos>

done
```

Seu funcionamento segue o seguinte princípio: `variavel` assume os valores que estão dentro da lista durante os *loops* de execução dos `comandos`. As listas podem ser valores passados como parâmetros, dados de algum arquivo ou o resultado da execução de algum comando. Com o exemplo abaixo fica mais fácil de entender isso.

Ex: Este programa cria diretórios com o nome diretorioNUMERO, onde NUMERO vai de 1 à 5.

```
#!/bin/bash


for i in ‘seq 1 5‘
do
    mkdir diretorio$i
done
```

O comando `seq NumInicial NumFinal` faz uma contagem seqüencial do número inicial dado até o número final.

Para estes 3 tipos de construção de *loops*, existem dois comandos que permitem alterar a rotina de sua execução . São eles:

- `break [n]` - Este comando aborta a execução do *loop* e salta para a próxima instrução após o *loop*.
- `continue [n]` - Este comando faz com que o fluxo de execução do programa volte para o início do *loop* antes de completá-lo.

  Ex: O Exemplo abaixo ilustra o uso de `break` e do `continue`

```
#!/bin/bash

echo "Tente acertar o número "
echo "Dica: Ele está entre 10 e 50. "

i=1
while true
do
  echo "Digite o Número: "
  read num
  if [ $num != 30 ]
    then
      echo "Você errou. Tente outra vez"
      let i++
      continue
```

```
    fi

    if [ $num == 30 -a $i == 1 ]
      then
        echo Você acertou de primeira. Parabéns!
        break
    fi
    if [ $num == 30 ]
      then
        echo Você acertou após $i tentativas.
        break
    fi

done
```

Apêndice A

O Projeto GNU e o Linux

As informações completas que estão neste capítulo podem ser encontradas na página oficial do Projeto GNU, veja ref. [11].

O Projeto GNU foi idealizado em 1983 como uma forma de trazer de volta o espírito cooperativo que prevalecia na comunidade de informática nos seus primórdios – para tornar a cooperação possível uma vez mais, removendo os obstáculos impostos pelos donos do software proprietário.

Em 1971, quando Richard Stallman iniciou a sua carreira no MIT, ele trabalhava em um grupo que usava *software* livre exclusivamente. Mesmo as empresas de informática distribuíam *software* livre. Programadores eram livres para cooperar entre si, e freqüentemente faziam isso.

Nos anos 80, quase todo o *software* era proprietário, o que significava que ele tinha donos que proibiam e impediam a cooperação entre os usuários. Isso tornou o Projeto GNU necessário.

Todo usuário de computadores necessita de um sistema operacional; se não existe um sistema operacional livre, você não pode nem mesmo iniciar o uso de um computador sem recorrer ao *software* proprietário. Portanto, o primeiro item na agenda do *software* livre é um sistema operacional livre.

Nos anos 90, este objetivo foi atingido quando um *kernel* livre foi desenvolvido por Linus Torvalds: este kernel era o Linux. A combinação do Linux com o quase completo sistema GNU resultou em um sistema operacional completo: um sistema GNU baseado no Linux (GNU/Linux), incluindo Slackware, Debian, Red Hat, entre outros. Leia o artigo: *why-gnu-linux* no site do projeto GNU.

Um sistema UNIX-like é composto por muitos programas diferentes. Alguns componentes já estão disponíveis como *software* livre, por exemplo, X Window, TeX, GNU Emacs, GNU C, Bash, Ghostscript. Atualmente, se tem feito um esforço para fornecer *software* para usuários que não são especialistas em computadores. Portanto, estamos trabalhando em um *desktop* baseado em ícones, utilizando arrastar-e-soltar para ajudar os iniciantes a utilizar o sistema GNU.

O projeto GNU não é somente desenvolvimento e distribuição de *softwares* livres. O coração do projeto GNU é uma idéia: que o *software* deve ser livre.

A.1 *Software* Livre

A expressão "Software livre"se refere à liberdade dos usuários executarem, copiarem, distribuírem, estudarem, modificarem e aperfeiçoarem o software. Mais precisamente, ela se refere a quatro tipos de liberdade, para os usuários do software:

1. A liberdade de executar o programa, para qualquer propósito.

2. A liberdade de estudar como o programa funciona e adaptá-lo para as suas necessidades. Acesso ao código-fonte é um pré-requisito para esta liberdade.

3. A liberdade de redistribuir cópias de modo que você possa ajudar ao seu próximo.

4. A liberdade de aperfeiçoar o programa e liberar os seus aperfeiçoamentos, de modo que toda a comunidade se beneficie. Acesso ao código-fonte é um pré-requisito para esta liberdade.

É importante ressaltar que *Software* Livre está ligada à liberdade de expressão, e não de preço.

Apêndice B

Editor de textos vi

O programa VI é o mais famoso editor de texto ASCII do UNIX. Foi desenvolvido em *Berkeley University California* por *Willian Joy*, a partir do editor UNIX Ed.

O VI trabalha em três modos:

- modo digitação ("a"ou "i")
- modo comando interno (<ESC>)
- comando na última linha (<ESC>":")

B.1 Comandos internos - vi

*** teclar <ESC> e a letra correspondente:**. As letras que podem ser usadas são listadas abaixo:

`h` – move o cursor para a esquerda
`l` – move o cursor para a direita
`j` – move o cursor para baixo
`k` – move o cursos para cima
`^f` – move uma tela para frente
`^b` – move uma tela para trás
`a` – insere caracter à direita do cursor
`A` – insere caracter no final da linha
`i` – insere caracter à esquerda do cursor
`I` – insere caracter no início da linha
`o` – insere linha abaixo do cursor
`O` – insere linha acima do cursor
`u` – desfaz última modificação
`U` – desfaz todas as modificações feitas na linha
`x` – apaga caracter
`dw` – apaga palavra
`dd` - apaga linha
`s` – substitui caracter
`cw` – substitui palavra
`/string` – procura *string*
`? string` – move o cursor para a ocorrência anterior da palavra

`n` – repete o último / ou ?

B.2 Comandos da última linha - vi

*** teclar `<ESC>`: e o comando correspondente:**. Os comandos podem ser os listados abaixo:
`set num` – enumera o texto
`set nonu` – retira numeração do texto
`5,10 d` – apaga da linha 5 até a linha 10
`1,2 co 4` – copia linhas 1 e 2 para depois da linha 4
`4,5 m 6` – move linhas 4 e 5 para depois da linha 6
`1` – posiciona o cursor na primeira linha do texto
`$` – posiciona o cursor no final do texto
`w` – salva o arquivo e continua editando
`q!` – sai do editor de textos sem salvar o arquivo
`x` – sai do editor de textos e salva o arquivo
Existem também outros editores de texto em sistemas UNIX como o **vim, emacs, joe, jed, pico**.

Referências Bibliográficas

[1] Neves, Julio Cezar, *Programação Shell Linux*, Editora Brasport, Rio de Janeiro, 2000.

[2] Araujo, Jáiro, *Comandos do Linux - Uso eficiente e avançado*, Editora Ciência Moderna Ltda., Rio de janeiro, 2001

[3] Tanenbaum, Andrew S., *Sistemas Operacionais*, Tradução D. A. Polli, USP, 2000.

[4] Raimundo, Rodivaldo Marcelo, *Curso Básico de programação em POSIX-Shell Script*, Editora Book Express, Rio de janeiro, 2000.

[5] Tanenbaum, Andrew S., *Organização Estruturada de Computadores* , Terceira edição, Editora Prentice/Hall do Brasil , Rio de Janeiro, 1992.

Sites

[6] **Guia focalinux** , *www.focalinux.org*, Este site contém guias sobre GNU/Linux de todos os níveis.

[7] **Bash FAQ**, *ftp://ftp.cwru.edu/pub/bash/FAQ*, Perguntas e dúvidas mais freqüentes relacionadas ao *Bash*.

[8] **Expressões Regulares** , *http://guia-er.sourceforge.net*, Guia sobre expressões regulares.

[9] **Lista de discussão** , *http://br.groups.yahoo.com/group/shell-script*, Lista de discussão do yahoogrupos.

[10] **Shell Script - Por que programar tem que ser divertido**, *www.aurelio.net*, Aurélio Marinho Jargas, página pessoal. Contém uma vasta lista para referências sobre *script-shell*.

[11] **Projeto GNU**, *www.gnu.org*, Página oficial do projeto GNU

[12] *www.br-linux.org*

[13] *www.vivaolinux.com.br*

[14] *www.scriptbrasil.com.br*