UNIVERSIDADE FEDERAL DA PARAIBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMATICA

UNIX: Conceitos Avançados e Programação do Shell

ALVARO FRANCISCO DE CASTRO MEDEIROS

Campina Grande, Dezembro de 1991.

UNIVERSIDADE FEDERAL DA PARAIBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMATICA

ALVARO FRANCISCO DE CASTRO MEDEIROS

UNIX: Conceitos Avançados e Programação do Shell

Dissertação apresentada ao curso de MESTRADO EM INFORMATICA da Universidade Federal da Paraíba, em cumprimento às normas para obtenção do Grau de Mestre.

JACQUES PHILIPPE SAUVÉ

Orientador

M488u Medeiros, Alvaro Francisco de Castro
UNIX : conceitos avancados e programacao do Shell / Alvaro Francisco de Castro Medeiros. - Campina Grande, 1991.
274 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. UNIX - 2. Filosofia e Uso do UNIX 3. Ambiente Operacional UNIX 4. Software - 5. Dissertacao I. Sauve, Jacques Phillippe, Dr. II. Universidade Federal da Paraiba - Campina Grande (PB)

CDU 004.4(043)

UNIX: Conceitos Avançados e Programação do Shell

Álvaro Francisco de Castro Medeiros

DISSERTAÇÃO APROVADA EM 23/12/91

JACQUES PHILIPPE SAUVÉ - Dr.

Orientador

JOSÉ ANTÃO BELTRÃO MOURA - Dr.

Componente da Banca

MARIA DE FÁTIMA CAMÊLO - M.Sc.

Componente da Banca

Campina Grande, 23 de Dezembro de 1991

À minha familia. Em especial à minha esposa, filha e a meus irmãos. Pelo amor que lhes tenho e pelos meses que deixei de estar com eles para me dedicar a este trabalho.

AGRADECIMENTO

Ao meu orientador Dr. Jacques Philippe Sauvé pela orientação, pelo encorajamento nas horas difíceis e, acima de tudo, pelo conhecimento que me transmitiu, contribuindo significativamente para o meu desenvolvimento acadêmico.

ABSTRACT

We have witnessed, in the last two decades, great improvements in hardware technology. Powerful computers have been realeased into the marketplace with increased frequency. However, despite the enhacements in technological features, the prices go down. It means that the cost of time spent to use computers diminishes every day. Now, the problems come with software professionals. Software produced to run in sophisticated machines are full of complexity and, as a matter of fact, very expensive.

The main purpose of this work is to sketch some concepts on which lies the UNIX Operational Environment Philosophy. This philosophy allows software designers to produce more and more in less time. In others words, it enhances productivity.

UNIX: Conceitos Avançados e Programação do Shell

RESUMO

Testemunhamos nas últimas décadas um avanço tecnológico muito grande na área de hardware. Novos computadores, cada vez mais potentes, são lançados no mercado com frequência. Os preços destas máquinas diminuem assustadoramente. Isto significa que o custo do tempo de uso do computador fica mais barato a cada dia. O problema agora é o profissional de software. Os programas para usar todos os recursos dos novos equipamentos ficam mais complexos e, consequentemente, mais caros.

O objetivo deste trabalho é traçar os conceitos sobre os quais está montada a Filosofia de uso do ambiente operacional UNIX. Através desta filosofia, o projetista de software poderá fazer mais trabalho em menos tempo, ou seja, ele terá mais produtividade.

ORGANIZAÇÃO DA TESE

I) INTRODUÇÃO DA TESE

Esta é uma tese de mestrado diferente das demais apresentadas neste departamento até o momento. É uma dissertação que tem como objetivo se transformar em um livro. O tema escolhido foi a Filosofia de uso do ambiente operacional UNIX. Como o próprio titulo da tese já indica "UNIX: Conceitos Avançados e Programação do Shell", estamos interessados em transmitir conhecimentos que possibilite ao leitor aumentar sua produtividade. Os conceitos aqui apresentados, apesar de terem como máquina alvo o UNIX, poderão ser adaptados também a outros sistemas operacionais.

O UNIX introduziu vários comandos e novas técnicas em Sistemas Operacionais. Nenhum programa ou idéia isoladamente funciona bem. A força do UNIX está na sua capacidade de programação e na FILOSOFIA de uso do computador. Esta Filosofia não pode ser expressa em uma simples sentença, na sua essência está a idéia que a potência do UNIX vem mais do relacionamente entre os programas que dos programas propriamente ditos. Muitos programas fazem tarefas triviais isoladamente, mas, quando combinados com outros programas, se tornam ferramentas gerais e poderosas.

A produtividade dos usuários do UNIX está diretamente relacionada com o nível de conhecimento dos conceitos fundamentais da Filosofia de uso deste Sistema Operacional. Usuários mais experientes, os chamados "gurus", conseguem taxas

de produtividade altissimas.

Apesar de existir uma bibliografia razoável sobre os comandos do UNIX, o treinamento de novos gurus é lento. Textos como [KORN 88] e [ANDE 86] são ricos em detalhes sobre o aspecto de programação do sistema, mas não falam sobre os Conceitos da Filosofia do UNIX. Já [SWAR 90], [KOGH 90], [MANI 86], e vários outros textos usam os conceitos em seus exemplos, sem tecer nenhum comentário sequer sobre os conceitos da filosofia do UNIX. [KERN 84] foi o único autor pesquisado a abordar levemente a questão filosófica do UNIX.

Este é um livro sobre a Filosofia de Uso do Sistema Operacional UNIX. Explicaremos o **como** e o **porquê** dos conceitos do UNIX.

II) IMPORTÂNCIA DA TESE

Uma dissertação de mestrado não objetiva, necessariamente, abordar tópicos de pesquisa básica ou aplicada. Não precisa consistir de implementações. Algumas teses nestes moldes foram ultrapassadas pelo avanço da tecnologia mesmo antes de concluidas. Outras abordaram assuntos tão marginais que serviram apenas como cumprimento parcial das exigências do programa do curso e ficaram arquivadas em prateleiras.

Algumas instituições estrangeiras acolheram o mérito de "surveys" aprofundadas que fazem apanhado abrangente e critico de uma determinada área de conhecimento e as endossam como dissertações de mestrado pelo valor didático, elucidativo e compreensivo que oferecem.

Aqui, fomos além de uma survey. Ofereceremos contribuição para o ensino e utilização do sistema UNIX, apontado como um dos três mais significativos na década de 90. Produzimos um texto com aproveitamento imediato para publicação como livro que servirá a várias instituições de P&D nacionais em Informática.

Este trabalho, além de ser uma tarefa intelectual não trivial, vem preencher uma lacuna na bibliografia sobre o ambiente operacional UNIX. Durante a pesquisa bibliográfica não foi encontrado texto que abordasse profundamente o aspecto filosófico, com ênfase para o aumento da taxa de produtividade, deste sistema operacional.

III) METODOLOGIA

A metodologia empregrada neste trabalho é fruto de vários anos de pesquisas dos orientadores. Alguns programas e vários manuais foram escritos baseados na metodologia que descreveremos a seguir.

A metodologia é baseada nas técnicas **topdown**, onde uma tarefa é refinada sucessivamente em operacões de menor complexidade. Na realidade, construimos um roteiro para a elaboracão desta dissertacão. Este roteiro é composto das seguintes etapas: 1. aquisição do conhecimento; 2. Despejo de memória em vários niveis; 3. ordenação das idéias; 4. catalogação e montagem dos exemplos; 5. desenho das figuras; 6. redação final dos capitulos.

Cada etapa do roteiro, para elaboração da dissertação, tem que obedecer um conjunto de diretrizes. Para saber se uma fase chegou ao seu final é só consultar a **lista de diretrizes** daquela etapa.

É de fundamental importância que a concentração em cada fase seja total. Não deve haver a preocupação em ordenar as idéias durante a fase de despejo de memória, por exemplo.

1. Aquisição de conhecimento

A aquisição do conhecimento necessário aos despejos de memória tem sua origem na experiência dos orientadores, em nosso trabalho de: pesquisa bibliográfica; uso do ambiente operacional

UNIX nos últimos oito anos; consulta ao código fonte do Sistema Operacional UNIX; definir, executar e gerenciar vários projetos de desenvolvimento de software na linguagem C.

2. Despejo de memória

Esta é a fase na qual devem ser anotadas todas as idéias sobre as quais se deseja dissertar. Não deve haver preocupação com ordem com que as idéias surgem. Esta é a fase que exige muita organização e disciplina do autor. Neste momento só existe o autor, a caneta e o papel em branco.

As idéias despejadas devem pertencer, inicialmente, ao nivel principal. No nosso caso, escolhemos dividir o livro em partes. Nosso despejo inicial foi decidir quantas partes teriamos e qual o conteúdo de cada uma destas partes do livro.

O processo de despejo de memória deve ser dividido em vários niveis de detalhamento ou refinamentos sucessivos. Um refinamento do despeja incial de nosso livro foi decidir quantos capitulos iriam compor cada parte.

3. Ordenação das idéias

Naturamente as idéias despejadas ficam desordenadas. Esta etapa tem como objetivo fazer um encadeiamento lógico das idéias despejadas em cada nivel.

É possivel que nesta fase o autor volte à fases anteriores.

Isto porque quando ordenamos os despejos de memória notamos a falta de informações para ajudar na fluência da apresentação dos conceitos.

4. Catalogação e montagem dos exemplos

Esta fase é marcada pela decisão de quais exemplos usar ao longo de todo o livro. Estes exemplos devem ser catalogados, implementados e submetidos a testes de execução antes de serem incorporados ao texto.

5. Desenho das figuras

Nesta fase devemos folhear nossas anotações e descobrir no texto os pontos nos exemplos que precisam ser enfatizados. Nestes locais devemos colocar figuras ou relatórios impressos pelos exemplos.

6. Redação final

Nesta fase final o autor deverá "costurar" o texto. Costurar no sentido de conseguir uma redação final que se apresente suavemente aos olhos do leitor. Este texto final deve englobar todo o conteúdo despejado e o sequenciamento lógico da idéias deve ser mantido.

O processo para escrever a redação final é simples. O autor deve ter a visão geral do trabalho apresentado no despejo inicial ou índice central. Para cada capítulo a ser escrito, os despejos

de memória ordenados e as figuras deverão ser consultados.

Vejamos, a titulo de ilustração, um exemplo parcial de uso da metodologia para a construção do livro que compõe esta tese.

O resultado da fase **Despejo de memória - nivel 1:**

1. Escolha das partes da dissertação
    - I. Filosofia do UNIX
    - II. Questionando conceitos familiares
    - III. Programação no Shell
    - IV. Aumentando sua produtividade
    - V. Um exemplo real
    - VI. Automatizando as tarefas do programador
    - VII. Os vários shells do UNIX
    - A. Exercicios propostos
    - B. Micro-manual do UNIX
    - C. Manual de referência do Shell
    - D. Variáveis pré-definidas

O nivel 2 de despejo de memória deve detalhar mais cada item acima. Não deve haver preocupação nesta fase com a ordem lógica das idéias, o importante é que as elas sejam **despejadas no papel**. A ordenação se dará em uma etapa posterior.

Vejamos a seguir o resultado parcial da fase **Despejo de memória - nivel 2** que, na realidade, é um refinamento da fase 1. Nesta etapa, escolhemos um tópico, no caso "PARTE I: FILOSOFIA DO UNIX". A partir deste tema,

escolhemos os capitulos que vão compor esta parte do livro:

PARTE I: FILOSOFIA DO UNIX

01. Introdução

02. Pedras angulares do UNIX

(...)

A seguir mostraremos o resultado parcial de um terceiro refinamento de um Capitulo da Parte I do livro. Esta fase é o que chamamos de **Despejo de memória - nivel 3:**

PARTE I: FILOSOFIA DO UNIX

01. Introdução

1 O que queremos dizer com Filosofia?

2 UNIX tem seu próprio estilo. Devemos conhecer as razões deste estilo para usar o UNIX melhor;

3 Se não conhecer a filosofia, pode usar UNIX, mas tem a produtividade comprometida;

4 Filosofia do UNIX tem como objetivo melhorar a produtividade do usuário;

5 Mostrar exemplo:

solução comum versus solução UNIX

DESAFIO PARA A SOLUÇÃO DE UM PROBLEMA;

6 FERRAMENTA o meio de se conseguir produ-

tividade usando a filosofia UNIX (explicar superficialmente);

7 ORTOGONALIDADE: Simplicidade é desejável. Ortogonalidade faz com que as coisas simples sejam poderosas;

8 FERRAMENTA é por definição o resultado de se aplicar ORTOGONALIDADE + SIMPLICIDADE ao mundo dos Sistemas Operacionais;

9 FERRAMENTA + ORTOGONALIDADE + "COLA" = PRODUTIVIDADE;

10 Originalmente, o UNIX era o melhor exemplo da frase SMALL IS BEAUTIFUL;

11 Mostrar exemplos de ortogonalidade no UNIX;

12 Aprendizagem lenta. Dizer porquê (tipo de interface);

13 Benefícios da interface UNIX;

14 Necessidades de um "GURU";

15 Livro dirigido para quem não tem Guru;

16 Não é livro de comandos porque queremos mostrar outras coisas;

17 UNIX não é perfeito. Mostrar os pontos negativos:

- Usuário leigos não gostam;
- Livro dirigido a programadores.

18 Ler livro mais de uma vez para enxergar mais;

19 UNIX deu certo no mundo não pelos moti-

vos técnicos aqui expostos, mas por razões mercadológicas;

20 A interface do UNIX não foi projetada para interagir com o Homem e sim com a máquina. O Shell do UNIX foi projetado para programadores;

21 Por que um histórico?

- Dar crédito às pessoas corretas;
- Nós (autores) achamos interessante.

22 Evolução do UNIX (padrão);

23 Contribuições importantes que vão perdurar quando o UNIX morrer:

- Transportabilidade;
- Sistemas abertos;
- Ferramentas (como usar os conceitos do UNIX em outros Sistemas Operacionais).

24 Definição de leve de ORTOGONALIDADE

25 Conceitos introdutórios

- Multiusuário;
- Multitarefa.

27 Popularidade do UNIX;

03 Um pequeno problema;

04 Mostrar de superficialmente como os conceitos da Filosofia do UNIX pode

ajudar a melhorar a produtividade:

- Ferramenta de software;
- Filtros;
- Interface seca;
- Unicidade / simplicidade.

Estes mesmos passos foram repetidos para todos os capítulos desta dissertação. O número de nivéis ou refinamentos foi três. Após três refinamentos o gráu de detalhamento foi condiderado satisfatório para se montar os exemplos.

Depois de ordenar o texto despejado acima, os próximos passos foram: catalogação dos exemplos, através da nossa experiência e pesquisa bibliográfica; desenhos das figuras usadas em conjunto com os exemplos; redação do texto final a partir das anotações despejadas no papel.

O objetivo maior deste trabalho é explicar o **como** e o **porquê** da existência dos conceitos do UNIX. Mostraremos que é possivel aumentar sua produtividade com taxas acima de **mil por cento. Esta** não é uma dissertação introdutória expondo os comandos do UNIX. A filosofia do UNIX, fartamente discutida neste trabalho, poderá **ser** aplicada a outros sistemas operacionais com o objetivo de melhorar sua taxa de produtividade.

Este livro se destina a usuários tecnicamente sofisticados: programadores, analistas, engenheiros de softwares, adminstradores de máquinas UNIX e todas as pessoas envolvidas com tarefas de projetar, desenvolver e manter software. É recomendável que o leitor tenha prévios conhecimentos sobre o sistema operacional UNIX ou outro que seja compatível. É muito interessante que o usuário tenha também noções sobre a linguagem de programação Shell. Recomendamos a leitura de [KERN 84] para os usuários iniciantes no UNIX.

O que você vai ganhar lendo este livro?

A resposta é **produtividade.** Você poderá automatizar suas tarefas diárias fazendo com que o computador trabalhe pesado. Você vai se libertar das atividades repetitivas para fazer apenas as tarefas nobres, como pensar em novas soluções para seu sistema atual. O conhecimento da filosofia do UNIX também é de grande

valia para analistas de pré-venda e pós-venda no seu dia-a-dia.

Este trabalho foi dividido em quatro partes: na primeira parte discutimos os aspectos filosóficos que apoiam o UNIX; na segunda parte questionamos os conceitos familiares deste sistema operacional, mostrando alguns detalhes internos de implementação; na terceira parte analisamos o interpretador de comandos do UNIX como uma linguagem de programação; finalmente, na quarta parte nós avaliamos a importância de algumas ferramentas para aumentar sua produtividade.

Este livro pode ser usado em um curso sobre o UNIX com melhor destaque para o interpretador de comandos shell. O instrutor poderá usar a primeira parte deste livro com o objetivo de motivar o aluno a investir seu tempo em pesquisa sobre os recursos do UNIX. O retorno ao aluno deve ser prometido em termos de melhores taxas de produtividade. Na segunda parte, poderemos nivelar a turma com relação aos conceitos básicos sobre o sistema operacional UNIX. Na terceira parte, o instrutor deve explorar os recursos do interpretador de comandos do UNIX do ponto de vista de uma linguagem de programação. Finalmente, na quarta e última parte, o mestre poderá escolher algumas ferramentas do UNIX, dentre as apresentadas e fazer um estudo mais detalhado sobre as mesmas. A critério do professor, novas ferramentas, pertinentes às atividades dos alunos, podem ser apresentadas.

Este texto se aplica a qualquer UNIX ou UNIX-like disponivel

no mercado.

Adotamos as seguintes convenções quanto ao tipo de letra usado neste livro:

1. Letras normais usadas no texto;

2. *Letras em italico que indicam saída de um programa na tela ou dados que são digitados do teclado.*

3. Saídas de programas ou dados digitados pelo usuário são apresentados dentro de uma moldura. Por exemplo:

$ comando

```
linha um impressa pelo comando
linha dois impressa pelo comando
          ( ... )
```

$ comando

```
dados digitados para comando
          ( ... )
```

V) TABELA DAS FIGURAS DO LIVRO

VI) SUMARIO DO LIVRO

PARTE I: FILOSOFIA DO UNIX

PARTE I: FILOSOFIA DO UNIX

1. INTRODUÇÃO

Neste capitulo introdutório queremos estudar, através de exemplos concretos, como o conhecimento da Filosofia do UNIX resulta diretamente em expressivos ganhos de produtividade para o usuário tecnicamente sofisticado. Como veremos, o conceito de **ferramenta de software** ocupa uma posição de destaque nesta filosofia.

1. Introdução

1.1 A Importância da Filosofia UNIX

O que queremos dizer com **Filosofia UNIX** é a forma como os utilitários do UNIX são usados para resolver um determinado problema. O UNIX, como qualquer outro sistema, tem seu próprio estilo. Devemos conhecer as razões deste estilo para melhor utilizar este sistema operacional. O usuário que não conhecer a Filosofia UNIX certamente conseguirá usar o sistema, mas, provavelmente, terá sua taxa de produtividade muito baixa em relação às pessoas que conhecem o jeito UNIX de usar seus comandos.

Aumentar sua produtividade é, portanto, nosso objetivo maior. Através de exemplos mostraremos os conceitos fundamentais que sustentam a Filosofia do UNIX.

1.2 O desafio

No sentido de pôr o estilo UNIX em evidência, adotemos a seguinte estratégia. O trabalho do dia-a-dia de um programador varia muito. Vamos descrever uma tarefa típica que um programador teria que resolver. Desafiamos você a resolvê-la no menor espaço de tempo possível, usando seu sistema operacional predileto.

1.2.1 O problema

Num certo diretório do seu computador estão uma centena de arquivos compondo o código fonte de um programa escrito na linguagem de programação C. Uma das variáveis do programa chama-se *tempo*. O nome desta variável deve ser mudado para *tempo_de_resposta*. Faça as modificações necessárias nos arquivos.

1.2.2 O tempo

Levando em consideração os vários arquivos abaixo, em quanto tempo você resolve o problema acima? É uma questão de horas, minutos ou segundos?

alug.idx	*im_desoc.c*	*im_lcntr.c*	*im_prop.h*
cntr.dat	*im_etiq.c*	*im_lhist.c*	*im_rec.c*
cntr.idx	*im_falug.c*	*im_limov.c*	*im_todos.c*
esp_im.doc	*im_fhist.c*	*im_lprop.c*	*im_varia.c*
estados.cnf	*im_fimov.c*	*im_ltxds.c*	*im_venc.c*
hist.dat	*im_finq.c*	*im_macro.h*	*im_venc.mod*
hist.idx	*im_fprop.c*	*im_malug.c*	*im_versao.h*
iatraso.cnf	*im_frec.c*	*im_mcntr.c*	*imd.cnf*
im.c	*im_ftxds.c*	*im_mhist.c*	*imov.dat*

(...)

repetitiva de grifar padrões; 2) em um trabalho manual, onde não há motivação criativa, a tendência é a ocorrência de falha humana; 3) os círculos feitos ao redor do padrão *tempo* não vão ajudar na correção. **O computador não lê marcas de lápis.**

Supondo que o usuário já sublinhou todas as ocorrências do padrão *tempo* na listagem, ele terá agora que editar os arquivos. Para modificar o texto dos programas fontes, o usuário deverá chamar um editor de texto passando o nome de cada arquivo como parâmetro. Vamos admitir que o usuário tenha escolhido usar o editor de texto *ed* do UNIX. Para saber o nome correto dos arquivos ele deverá usar um comando que liste o diretório, como o comando *ls* do UNIX, por exemplo:

$ ls -C

alug.idx	*im_desoc.c*	*im_lcntr.c*	*im_prop.h*
cntr.dat	*im_etiq.c*	*im_lhist.c*	*im_rec.c*
cntr.idx	*im_falug.c*	*im_limov.c*	*im_todos.c*
esp_im.doc	*im_fhist.c*	*im_lprop.c*	*im_varia.c*
estados.cnf	*im_fimov.c*	*im_ltxds.c*	*im_venc.c*
hist.dat	*im_finq.c*	*im_macro.h*	*im_venc.mod*
hist.idx	*im_fprop.c*	*im_malug.c*	*im_versao.h*
iatraso.cnf	*im_frec.c*	*im_mcntr.c*	*imd.cnf*
im.c	*im_ftxds.c*	*im_mhist.c*	*imov.dat*
im.cnf	*im_gera.c*	*im_mimov.c*	*imov.idx*

(...)

O exemplo acima mostra a saída do comando *ls -C* que imprime todos os arquivos do diretório corrente. A opção *-C* é para imprimir o relatório em colunas. O símbolo *$* representa o caractere de prontidão do UNIX.

Agora tem início o trabalho manual. Através de uma pesquisa visual no relatório impresso pelo comando *ls*, o usuário localiza o primeiro arquivo cujo nome termina em *.c* ou *.h*. O arquivo *im.c* é o primeiro a ser editado (todos os programas escritos na linguagem C usam estes sufixos). Para modificar o arquivo *im.c*, temos que chamar o editor de texto. O usuário pode escolher o que melhor lhe convier. Para editar o arquivo *im.c* o usuário teria que chamar o *ed*, que é um dos editores de textos disponíveis no UNIX, da seguinte forma:

$ ed im.c

Aqui, mais um vez, a produtividade do usuário está ligada ao seu conhecimento em relação às ferramentas usadas. Ele precisa localizar a linha, dentro do arquivo *im.c*, que contém o padrão *tempo*. Esta operação poderá ser instantânea, se o usuário conhecer o comando de posicionamento de linha do editor *ed*, ou levar alguns minutos se a busca for visual na tela. Vamos supor que o usuário conhece o comando de posicionamento absoluto do *ed*, o que é uma forma intermediária entre a pesquisa visual na tela, e a localização instantânea. Saber chegar diretamente às linhas do arquivo correspondentes às linhas grifadas na listagem do

lápis vermelho é muito útil para o usuário.

Uma vez que a linha foi localizada dentro do arquivo, precisamos fazer a troca do *padrão-velho* pelo *padrão-novo*, no caso *tempo* por *tempo_de_resposta*. O comando interno ao editor *ed* para fazer a substituição de um *padrão-velho* por um *padrão-novo* é o seguinte:

s/padrão-velho/padrão-novo/

A sequência completa de comandos para editar um arquivo, *im.c* por exemplo, é a seguinte:

```
$ ed im.c
10
s/tempo/tempo_de_resposta/
25
s/tempo/tempo_de_resposta/
     ( ... )
w
q
```

Observe que os números *10* e *25* são comandos internos para o *ed* se posicionar, respectivamente, nas linhas de mesmo número dentro do arquivo *im.c*. Estamos supondo que houve ocorrência do padrão *tempo* nestas linhas. Na realidade, estas são as linhas grifadas pelo lápis vermelho. As reticências (...) no exemplo acima significam que o usuário deverá digitar as duas linhas de comandos para o editor *ed* repetidas vezes:

```
10
s/tempo/tempo_de_resposta/
```

para poder localizar e substituir cada ocorrência do padrão *tempo* no arquivo *im.c*. Claro que os números *10*, *25* e ... são particulares ao arquivo *im.c* refletindo as linhas onde o padrão pesquisado foi encontrado neste arquivo. O importante aqui é que você terá que digitar dois comandos para o *ed*. Um número para localizar a linha que contém o padrão e um comando para fazer a substituição propriamente dita. É muito trabalho manual, não é? Finalmente, o comando interno do *ed* para gravar em disco as modificações feitas é *w* (write). O comando *q* (quit) é usado para sair do editor de texto *ed* e voltar ao sistema operacional.

Pronto! Completamos a edição das modificações para um arquivo cujo nome é *im.c*. Você se lembra qual o próximo arquivo a editar?

Para você leitor é fácil, basta voltar à página anterior e olhar a saída do relatório do comando *ls*. No caso do usuário, a saída do comando *ls* já rolou tela acima, desaparecendo. Ele terá que executar o comando *ls -C* novamente, para listar os nomes dos arquivos:

```
$ ls -C

alug.idx        im_desoc.c      im_lcntr.c      im_prop.h
cntr.dat        im_etiq.c       im_lhist.c      im_rec.c
cntr.idx        im_falug.c      im_limov.c      im_todos.c
esp_im.doc      im_fhist.c      im_lprop.c      im_varia.c
estados.cnf     im_fimov.c      im_ltxds.c      im_venc.c
hist.dat        im_finq.c       im_macro.h      im_venc.mod
hist.idx        im_fprop.c      im_malug.c      im_versao.h
iatraso.cnf     im_frec.c       im_mcntr.c      imd.cnf
im.c            im_ftxds.c      im_mhist.c      imov.dat
im.cnf          im_gera.c       im_mimov.c      imov.idx
                        ( ... )
```

O mesmo processo usado para editar o arquivo *im.c* será repetido para modificar o arquivo *im_desoc.c*. O nome *im_desoc.c* foi obtido através de pesquisa visual no relatório impresso pelo comando *ls*. O usuário deverá digitar os mesmos comandos usados para alterar o arquivo *im.c*, só que agora usados no arquivo *im_desoc.c*, é claro. O primeiro passo para fazer a substituição é a chamada do editor que é feita com o seguinte comando:

```
$ ed im_desoc.c
```

Agora que o editor de texto está executando, temos que digitar vários comandos internos (comandos interpretados pelo *ed*) para o editor fazer a substituição de *tempo* por *tempo_de_resposta*:

```
35
s/tempo/tempo_de_resposta/
47
s/tempo/tempo_de_resposta/
     ( ... )
w
q
```

Estamos supondo que no arquivo *im_desoc.c* o padrão *tempo* ocorreu nas linhas 35, 47, (...).

Ufa! Acabamos de editar o segundo arquivo. O mesmo processo acima deve se repetir pelo menos noventa e oito vezes, levando em consideração que temos uma centena de arquivos a editar.

Na solução apresentada acima o esforço manual foi muito grande. Nada foi executado de forma automática. A Filosofia do UNIX não foi empregada. O tempo total gasto para listar, pesquisar no papel, fazer círculos com lápis vermelho, listar arquivos, ativar editor para cada arquivo, ..., é de aproximadamente cinco horas. Este tempo poderá variar para mais ou para menos, dependendo da agilidade na digitação dos comandos e do conhecimento dos recursos do editor de texto que for usado.

O que? Você está achando muito? Você resolveria em menos tempo? Pode até ser verdade que você resolva mais rapidamente o problema da pesquisa e troca de um padrão por outro em uma cente-

nas de arquivos, usando o método do lápis vermelho, mas nós achamos muito difícil que você consiga, pelas seguintes razões. Primeiro, você não vai aguentar editar todos os arquivos sem dar uma paradinha para fumar um cigarro e tomar um café. Você tem que levar em consideração aquele tempo gasto depois do lanche com discussões inúteis, que não levam a lugar nenhum, como futebol, por exemplo, que tem que existir em qualquer empresa com mais de um funcionário. Você não concorda?

Precisamos de algum recurso do UNIX que nos permitam reduzir o tempo gasto no desafio. Éste recurso se chama ferramenta.

1.3.2 Ferramenta

Aqui vamos introduzir o conceito de **ferramenta de software**, através da automação da pesquisa dos padrões nos arquivos. Vamos escrever uma **ferramenta especifica** para procurar as linhas dos arquivos da linguagem C que contenham o padrão *tempo* e as imprima. Discutiremos também a importância das ferramentas gerais.

A solução anterior penalizou o usuário com uma carga intensa de trabalho manual, repetitivo, sem nenhuma criatividade. A Filosofia do UNIX preza justamente o contrário: a máquina deve ser responsável pelo trabalho repetitivo e o homem deve ocupar seu tempo com tarefas nobres, que exijam sua capacidade de raciocinio.

Nesta solução, o usuário constrói um programa *prog* que pesquisa e lista as linhas dos arquivos fontes, escritos em C, que contenham o padrão *tempo*. O programa *prog* é ativado da seguinte forma:

```
$ prog

    im.c:10:        tempo = 130;
     ( ... )
    im.c:25:        horas = tempo / 60;
     ( ... )
```

Observe que no relatório do programa acima é impresso o nome do arquivo *im.c*, o número da linha no arquivo *10* e o conteúdo da linha *tempo = 130;*. Esta é a forma como o programa *prog* localiza, de forma automática, as linhas contendo o padrão *tempo* em todos os arquivos. Como o programa *prog* é especifico, ele sabe que tem que abrir todos os arquivos cujo nomes terminam em *.c* e *.h*. As linhas dos arquivos (*im.c, im_desoc.c, ...*) onde aparecem o padrão *tempo* são impressas no vídeo. O trabalho manual para resolver o desafio foi reduzido pela metade. A parte da pesquisa ficou automática através de *prog*, mas a edição ainda tem que ser manual:

```
$ ed im.c
      10
      s/tempo/tempo_de_resposta/
      25
      s/tempo/tempo_de_resposta/
           ( ... )
      w
      q
```

onde o usuário tem que ativar o editor *ed* para cada arquivo que contém o padrão e digitar os comandos internos para o editor poder fazer a substituição de *tempo* por *tempo_de_resposta* no arquivo *im.c*. O mesmo processo se repete para cada novo arquivo a editar.

A solução acima ainda não é a ideal, pois ainda tem muito trabalho manual sendo feito, mas é melhor que usar o lápis vermelho e uma listagem. O trabalho manual de grifar os padrões foi automatizado através do programa *prog*, que imprime os nomes dos arquivos e os números das respectivas linhas onde estão localizados os padrões *tempo*.

O programa *prog* é muito especifico: só funciona para pesquisar um determinado padrão *tempo* em arquivos terminados em *.c* ou *.h*. Você usa uma vez e, provavelmente, o jogará na lata de lixo depois. Utilitários restritos como *prog* não ajudam outros programadores. Cada programador terá que escrever suas próprias ferramentas especificas, o que afeta a sua taxa de produtividade de forma negativa.

O leitor mais atento já devè ter desconfiado qual o caminho para alcançar melhor produtividade e usar a Filosofia do UNIX: a palavra chave é **AUTOMAÇÃO das tarefas repetitivas**. Por ter automatizado a pesquisa acima, não é mais preciso usar o lápis vermelho e uma listagem, o que reduz o tempo gasto para resolver o problema de **cinco horas** para **quatro horas**.

Escrever **ferramentas especificas**, como o programa *prog*, que atendam nossas necessidades imediatas e eliminem o trabalho braçal, ajuda bastante a melhorar nossa produtividade. Uma ferramenta especifica resolve o seu problema particular. Pode existir outro usuário no sistema que precise pesquisar outro padrão onde o programa *prog* não se encaixa por ser muito expecifico: ele só procura o padrão *tempo*. O jeito é escrever uma **ferramenta geral** que possa ser utilizada por outros usuários, para pesquisar qualquer padrão em qualquer arquivo. A ferramenta geral recebe os padrões e os nomes dos arquivos como **parâmetros**. O usuário que quiser pesquisar o padrão *tempo* em todos os arquivos escritos na linguagem C, deverá ativar a ferramenta geral *pesquisa*, por exemplo, passando os parâmetros *tempo *.c *.h* da seguinte forma:

```
$ pesquisa tempo *.c *.h

     im.c:10:        tempo = 10;
          (...)
     im_desoc.c:35: horas = tempo / 60;
          (...)
```

Outro usuário que desejar procurar o padrão *alvaro* em todos os arquivos terminados em *.doc* deverá executar *pesquisa* com os seguintes parâmetros:

```
$ pesquisa alvaro *.doc

     cap01.doc:      ... padrão alvaro em todos os
          (...)
```

ao invés de fazer sua própria ferramenta para pesquisar o padrão *alvaro*.

Felizmente não precisaremos escrever uma ferramenta para pesquisar padrões em arquivos pois o UNIX já fornece um utilitário com esta finalidade que se chama *grep*.

Ao contrário do programa *prog*, que só funcionava pesquisando um padrão especifico *tempo*, o utilitário *grep* procura qualquer padrão em qualquer arquivo. Os padrões e os nomes dos arquivos a serem pesquisados são passados ao comando *grep* como argumentos. A parametrização do comando *grep* é muito importante. Outros programadores não precisarão desenvolver programas para fazer pesquisas de padrões em arquivos, é só usar a ferramenta *grep* que já está pronta. O que significa economia de tempo, e, consequentemente, maior produtividade para você.

Os comandos *grep*, *ls*, *ed* e centenas de outros são distribuidos junto com o sistema operacional UNIX e devem estar disponiveis para todos os usuários do sistema.

Ferramenta é um dos quatros conceitos fundamentais sobre os quais a **Filosofia do UNIX** se apoia. Os outros conceitos são: **filtros, unicidade/simplicidade** e a **linguagem shell**. Todos estes conceitos serão discutidos ao longo deste livro.

1.3.3 Ferramentas *grep* e *ed*

Nesta seção, veremos como o conhecimento dos recursos de cada ferramenta usada, no caso os comandos *ed* e *grep*, contribuem para o aumento de sua produtividade.

Até este ponto, vimos duas soluções ao desafio. A primeira, uma solução não inteligente que usava um lápis vermelho + listagem para grifar os padrões. A segunda, utilizou uma ferramenta especifica para automatizar parte da solução ao desafio.

O fato de você conhecer melhor os comandos *grep* e *ed* ajudam

a diminuir o trabalho manual na solução do desafio. A opção *-l* do comando *grep*, por exemplo, lista apenas o nome dos arquivos que contêm o padrão pesquisado, o que resolve o problema de saber quais arquivos devem ser editados. O comando *g* do editor de texto *ed*, por outro lado, serve para pesquisar e substituir um padrão em todo arquivo, evitando que você digite repetidas vezes os dois comandos:

```
numero_da_linha
s/tempo/tempo_de_resposta/
   (...)
```

para localizar e substituir cada linha que contenha o padrão *tempo*. Usando o comando global *g* do editor de texto *ed*, há uma redução drástica na digitação. Todos os padrões de um arquivo são substituídos com um só comando:

```
g/tempo/s/tempo/tempo_de_resposta/gp
```

O comando interno do *ed* acima é composto de duas partes. A primeira *g/tempo/* é usada para localizar todas as linhas dentro do arquivo que contém o padrão *tempo*. A segunda parte, *s/tempo/tempo_de_resposta/*, é usada para fazer a substituição nas linhas selecionadas. As letras ***gp*** no final da linha do comando global significam que se houver mais de um padrão *tempo* na linha selecionada, a substiuição deve ser feita em toda a linha e o resultado da linha substituída deve ser impresso na saída padrão, no caso, o monitor de vídeo. Como estamos falando em produtividade, podemos usar uma forma resumida do comando acima:

```
g/tempo/s//tempo_de_resposta/gp
```

onde aproveitamos o fato de que o padrão a ser substituído é o mesmo padrão de pesquisa usado para localizar as linhas dentro do arquivo. No comando global g tanto faz escrever *g/tempo/s/tempo/tempo_de_resposta/gp* como usar uma forma mais resumida *g/tempo/s//tempo_de_resposta/gp*, para o editor *ed* o significado é o mesmo. Para o programador há uma redução na digitação.

Usando melhor as ferramentas *grep* e *ed* a solução ao desafio fica assim:

```
$ grep -l tempo *.c *.h

im.c
im_desoc.c
 ( ... )
```

No exemplo acima você seleciona de forma automática, através do comando *grep*, apenas os nomes dos arquivos que contém o padrão *tempo*. A opção *-l* modifica o formato de impressão do relatório de saída do comando *grep* imprimindo apenas os nomes dos arquivos. Observe que o simples fato de conhecer esta opção evita que você tenha que usar o comando *ls* para listar os nomes dos arquivos de

seu diretório e fazer pesquisa visual para saber qual o nome do próximo arquivo a editar. O que antes era chato e cansativo com o comando *ls* agora é simples e automático com o *grep -l*. Estamos evoluindo em direção à Filosofia UNIX.

Outro pilar sobre o qual a Filosofia UNIX está montada é o shell. O shell é ao mesmo tempo um interpretador de comandos (representado pelo símbolo de prontidão *$*) e uma Linguagem de Programação. Existem caracteres que são tratados de forma especial pelo shell. O caractere * (asterisco), por exemplo, é um deles. Quando você digitou o comando:

*$ grep -l tempo *.c *.h*

o shell interpretou o comando acima o expandindo para:

$ grep -l tempo im.c im_desoc.c (...)

Ou seja, o comando *grep* não recebe como parâmetro **.c* e **.h*. As expressões **.c* e **.h* serviram para o shell montar uma lista de argumentos *im.c im_desoc.c (...)* que são passadas para o comando *grep*. A expansão de caracteres especiais é muito cômoda para o programador evitar digitação desnecessária, o que significa mais conforto e melhor produtividade.

Apesar de você conhecer um pouco o interpretador de comandos shell, saber como ele faz expansão de caracteres especiais *(*.c e *.h)*, de saber que o *grep* aceita a opção *-l* e que o editor de texto *ed* do UNIX tem comandos globais, não automatizou completamente a solução ao desafio. A sua produtividade aumentou bastante em relação às soluções anteriores, isto é verdade, mas, você ainda continua fazendo muito trabalho repetitivo de digitação. O tempo total para esta solução deve ter caído de cinco horas para duas horas, em relação à solução do lápis vermelho, o que convenhamos, é um ganho significativo de produtividade.

1.3.4 Usuário leu o manual do shell

Nesta solução daremos uma visão introdutória da linguagem de programação shell. Mostraremos o comando de controle de fluxo *for* que possibilita a contrução de laços para a execução repetida de comandos.

1.3.4.1 Variáveis do shell

Aqui queremos explicar variáveis pois a sintaxe do comando *for* envolve o uso das mesmas.

Ao ler o manual do shell, o usuário descobriu que é possível criar variáveis. O mecanismo de criação é simples, basta fazer uma atribuição que a variável é automaticamente criada. Por exemplo, o comando abaixo:

var=alvaro

cria uma variável no shell cujo nome é *var*. Para usarmos o conteúdo de uma variável basta precedermos seu nome com o símbolo *$*, como mostra o exemplo abaixo:

```
$ echo o valor da variavel é $var
```

o comando acima imprimirá a seguinte mensagem na tela:

*o valor da variavel e **alvaro***

1.3.4.2 Exemplo de uso do comando *for*

O comando *for*, que é uma construção interna ao shell, é composto das palavras reservadas: *for*, *in*, *do* e *done*. Vejamos um simples exemplo de uso do comando *for*, para termos uma idéia superficial do seu funcionamento:

```
$ for var in 1 2 alvaro jacques
> do
>         echo $var
> done
```

O que será impresso na tela como resultado da execução do comando acima é o seguinte:

```
1
2
alvaro
jacques
```

Ou seja, a variável *var* assume o valor de cada palavra na lista que segue à palavra reservada *in*. Inicialmente a variável *var* assume o primeiro valor na lista que é *1*. O primeiro ciclo é processado, isto é, os comandos entre as palavras reservadas *do* e *done* são executados. No exemplo acima temos apenas o comando *echo* dentro do laço. Antes de executar o segundo ciclo, a variável de controle do laço *var* assume o próximo valor na lista, que no caso é *2*, o corpo do laço é executado novamente e assim sucessivamente.

Uma novidade surgiu com o exemplo anterior: o caractere de prontidão secundário >. Como o comando *for* se estende por mais de uma linha, o shell usa um segundo símbolo de prontidão para avisar ao usuário que está interpretando um comando composto de mais de uma linha física. Quando o usuário digita a palavra reservada *for* e seus parâmetros e em seguida tecla *<Enter>* (a tecla de fim-de-linha), o shell já coloca o caractere > esperando que você complete o comando em mais de uma linha física. Por exemplo:

```
$ for var in 1 2 alvaro jacques
> _
```

no comando acima o shell coloca o símbolo de prontidão secundário e fica esperando que você complete o comando digitando *do*:

```
$ for var in 1 2 alvaro jacques
> do
> _
```

nas próximas linhas, o shell fica com o cursor esperando que você digite os comandos que farão parte do corpo do laço *for*, no caso é só o comando *echo*:

```
$ for var in 1 2 alvaro jacques
> do
>         echo $var
> _
```

observe que o símbolo de prontidão secundário > está sendo mostrado o tempo todo. Isto significa que a digitação do comando *for* ainda não chegou ao final. O comando *for* só termina com a palavra reservada *done*:

```
$ for var in 1 2 alvaro jacques
> do
>         echo $var
> done
```

A partir da digitação da palavra *done*, o comando *for* começa a ser executado mostrando na tela a seguinte saída:

```
1
2
alvaro
jacques
```

Agora que já conhecemos superficialmente o *for* do shell, um comando que permite a repetição da execução de outros comandos, podemos automatizar a solução ao desafio da substituição de um padrão por outro em vário arquivos. Verifique se sua solução se parece com a solução apresentada abaixo:

```
$ for i in *.c *.h
> do
>     echo editando arquivo $i
>     ed $i
> done
```

O próprio shell se encarrega de expandir **.c* e **.h* para uma lista de argumentos que a variável *i* vai assumir. Esta relação é composta de todos os nomes dos arquivos do diretório corrente terminadas em *.c* ou *.h*:

```
$ for i in  im.c im_desoc.c im_etiq.c ...
> do
>     echo editando arquivo $i
>     ed $i
> done
```

O corpo do laço *for* acima é composto de dois comandos. O primeiro, o comando *echo*, é apenas para informar a você qual o arquivo que está sendo editado pelo *ed*. O segundo comando é o próprio editor de texto *ed* que recebe o nome do arquivo a editar como parâmetro. A execução do *for* acima resulta na impressão da seguinte mensagem na tela:

editando arquivo im.c

A mensagem acima indica que *im.c* é o primeiro arquivo na lista de valores da variável *i* do comando *for*. O próximo comando a ser executado é o *ed* que recebe como parâmetro *$i* que tem valor igual a *im.c* no primeiro ciclo do laço. Ou seja, o shell executa o seguinte comando:

ed im.c

O comando *ed*, apesar de estar rodando dentro de um laço, lê dados do terminal. Agora é moleza. A gente já conhece o comando global *g* do *ed* para, com um único comando, fazer a substituição do padrão *tempo* por um novo padrão *tempo_de_resposta* em todo o arquivo. O comando que vai fazer a substituição em todo o arquivo *im.c* é o seguinte:

```
g/tempo/s//tempo_de_resposta/gp
w
q
```

Na realidade, o comando global *g/.../gp* faz a substituição na memória temporária do editor, o comando *w* (write) faz a gravação das modificações, efetivamente, em disco e o comando *q* serve para finalizar a execução do editor *ed*. Quando o comando *ed* termina sua execução, o shell volta a executar no ponto imediatamente abaixo do comando *ed*. O próximo nome que o shell encontra no seu laço é a palavra reservada *done*. Ao encontrar *done* o shell faz com que a variável de controle *i* do *for* receba o valor seguinte na lista de variáveis. No nosso caso, *i* recebe o nome de arquivo *im_desoc.c*, e um novo ciclo do laço tem início com a execução do comando *echo*, que imprime a mensagem abaixo:

editando arquivo im_desoc.c

A mensagem acima indica que a variável *i* no laço *for* assumiu outro valor, *im_desoc.c*. O comando seguinte ao *echo* a ser executado é um novo *ed* que recebe como parâmetro *$i*. Como *$i* tem valor *im_desoc.c*, o comando que será executado é o seguinte:

ed im_desoc.c

Mais uma vez, com um simples comando para o editor *ed* você pode fazer a substituição do padrão *tempo* por *tempo_de_resposta* em todo o arquivo *im_desoc.c*:

```
g/tempo/s//tempo_de_resposta/gp
w
q
```

Ao digitar *q* (quit no inglês) o comando *ed* é finalizado, o shell encontra a palavra *done* e um novo ciclo é iniciado com a variável *i* valendo *im_etiq.c*. Os comandos a seguir:

```
echo arquivo $i
ed $i
```

são executados tendo como parâmetro o valor atual da variável *i* usada para controlar o *for*. A cada ciclo uma nova cópia dos comandos *echo* e *ed* são executados. Você terá que reponder apenas digitando os comandos *g/.../gp*, *w* e *q* para cada arquivo editado automaticamente pelo shell, através do *for*.

Resumindo, o que o usuário vai ter que digitar está impresso em destaque e o que o shell faz de forma automática está escrito em letras normais no relatório abaixo:

```
editando arquivo im.c
ed im.c
g/tempo/s//tempo_de_resposta/gp
w
q
editando arquivo im_desoc.c
ed im_desoc.c
g/tempo/s//tempo_de_resposta/gp
w
q
editando arquivo im_etiq.c
ed im_etiq.c
g/tempo/s//tempo_de_resposta/gp
w
q
        (...)
```

É importante relembrar que o shell é ao mesmo tempo um interpretador de comandos e uma linguagem de programação. É ele quem mostra a cadeia de caracteres de prontidão "$ " para você no terminal, lê e interpreta seus comandos. O utilitário */bin/sh*, o próprio shell, é também uma Linguagem de Programação com construções de controle de fluxo tipo *if*, *for*, *while*, etc.

No exemplo acima, o shell foi o responsável pela automação das várias ativações do editor de texto *ed* para todos os arquivos terminados em *.c* e *.h* do sistema. O usuário teve que digitar apenas os comandos que são usados pelo editor de texto *ed* para cada arquivo:

```
g/int tempo/s//float tempo/gp
w
q
      (...)
```

Conhecer a Filosofia do UNIX (o shell é apenas uma parte dela) ajudou o usuário a **automatizar** tarefas manuais e, consequentemente, aumentar sua taxa de **produtividade.** O nível de automação melhorou bastante, mas o usuário ainda precisa digitar comandos repetitivos *g/.../gp* para o editor de textos *ed.* Acreditamos que o *tempo* total gasto nessa solução não seja superior a **uma hora.**

1.3.5 Usuário preguiçoso

Mostraremos aqui como o simples fato de o *ed* poder ler comandos de um arquivo pode ajudar na solução. Ilustraremos também como a preguiça de digitar acaba ajudando a encontrar soluções que usam a Filosofia do UNIX (deixar o computador fazer o trabalho pesado).

A preguiça de digitar acaba melhorando a solução e forçando o aprendizado dos conceitos fundamentais do UNIX. O usuário pesquisa o manual à procura de uma forma mais confortável ou menos cansativa de fazer as coisas no UNIX. Queremos evitar a digitação desnecessária de qualquer jeito.

Na solução anterior tivemos a automação da edição de todos os arquivos através do comando *for* :

```
$ for i in *.c *.h
> do
>     echo editando arquivo $i
>     ed $i
> done
```

O único problema é que tínhamos que digitar, várias vezes, três comandos para o editor de texto *ed* fazer a alteração do padrão *tempo* pelo novo padrão *tempo_de_resposta*. Os comandos digitados repetidamente eram:

```
g/tempo/s//tempo_de_resposta/gp
w
q
```

O bom no usuário preguiçoso é que ele vai atrás de uma forma mais fácil de fazer as coisas e acaba descobrindo, nos manuais do sistema, como evitar esta digitação dos comandos *g/.../gp* repetidas vezes: é simples, basta editar um arquivo contendo os comandos para o editor *ed.* Por exemplo, o arquivo *comandos* poderia conter os comandos do editor *ed* para fazer a substituição requerida no desafio:

```
$ cat /tmp/comandos

     g/tempo/s//tempo_de_resposta/gp
     w
     q
```

Através do comando do UNIX *cat*, que lista o conteúdo de arquivos cujos nomes são passados como parâmetros, visualizamos o conteúdo do arquivo */tmp/comandos* que é o texto abaixo:

```
     g/tempo/s//tempo_de_resposta/gp
     w
     q
```

o arquivo */tmp/comandos* é temporário, ou seja, usado uma vez é jogado fora. Por isso ele foi criado no diretório */tmp* que é o local no UNIX para arquivos temporários. Você pode ativar o editor *ed* redirecionando sua entrada padrão para o arquivo *comandos*. Em outras palavras, o *ed* vai ler os comandos para fazer a troca do padrão *tempo* por *tempo_de_resposta* a partir do arquivo */tmp/comandos*. A forma como o shell implementa redirecionamento é através do símbolo <. Por exemplo, o comando abaixo:

```
$ ed im.c < /tmp/comandos
```

O comando acima faz a substituição dos padrões desejados no arquivo *im.c*. No exemplo acima, o *ed* leu os comandos *g/.../gp*, *w* e *q* a partir de um arquivo de comandos */tmp/comandos*.

Já deu para perceber que esta solução vai automatizar as duas partes da solução: a geração de nomes e substituição propriamente dita. Pela solução anterior, onde usamos o laço *for*, somos levados a crer que o comando seguinte resolverá o desafio de forma automática:

```
$ for i in *.c *.h
> do
>     ed $i < /tmp/comandos
> done
```

O programador não precisa mais digitar dados repetidos sem necessidade. Ele criou um arquivo de comandos */tmp/comandos* contendo os comandos:

```
     g/tempo/s//tempo_de_resposta/gp
     w
     q
```

que são usados pelo *ed* para editar todos os arquivos, um a cada ciclo do laço *for*.

Com certeza esta será a sua solução, não é? Não que você seja, necessariamente, um usuário preguiçoso, pelo contrário, você pesquisou os manuais do sistema. Você descobriu o laço *for* e também como fazer redirecionamento. Uma pergunta que você pode

estar fazendo a si próprio é a seguinte: será que isto funciona mesmo?

Vamos responder a pergunta acima mostrando como o shell resolve o *for* e o redirecionamento do *ed* passo-a-passo. Assim como existem opções para o comando *grep* (-l), existem opções para o shell (utilitário */bin/sh*). Uma opção que faz com que o shell mostre o que está fazendo é -x. A forma de ligar esta opção no shell é através do comando interno *set*:

```
$ set -x
```

O comando acima informa ao shell que ele deve mostrar as linhas expandidas e todos os comandos antes de executá-los.

Para ver a execução da solução é só digitar o comando *for* contendo o *ed* para cada arquivo com sua entrada redirecionada para o arquivo */tmp/comandos*. O comando *for* seria o seguinte:

```
$ for i in *.c *.h
> do
>     ed $i < /tmp/comandos
> done
```

Após a digitação do comando acima, as ações do shell podem ser visualizadas na tela. O shell usa o caractere + para indicar a linha que ele vai executar. Entretanto, existem ações, como a expansão da lista de valores para a variável *i* e o redirecionamento da entrada padrão do comando *ed* que é feita internamente pelo shell e que, consequentemente, você não verá no vídeo. A saída do comando acima na tela é a seguinte:

```
+ ed im.c
1234
tempo_de_resposta = 10;
horas = tempo_de_resposta / 60;
      ( ... )
1432
+ ed  im_desoc.c
      ( ... )
+ ed im_etiq.c

      ( ... )
```

Onde a mensagem que é impressa pelo shell *+ ed im.c* é o resultado da opção -x estar ligada. O número *1234* é impresso pelo *ed* e indica a quantidade de caracteres que o arquivo *im.c* tem no início de sua edição. A linha *tempo_de_resposta = 10;* é impressa pelo *ed* por causa do comando *gp* que é usado no arquivo de comandos */tmp/comandos*. *gp* quer dizer faça substituição globalmente e imprima a linha substituída. As reticências representam a impressão de todas as linhas substituídas no arquivo. O número *1432* indica a quantidade de caracteres na hora de fazer a gravação do arquivo *im.c* editado, depois da alteração (comando *w* no arquivo de comandos */tmp/comandos*). O mesmo processo acima se

repete para todos os arquivos da lista do *for* (*im.c im_desoc.c im_etiq.c ...*).

Aqui o usuário começa a colher os frutos pelo seus conhecimentos dos utilitários */bin/sh* e */bin/ed*. Novamente a filosofia do UNIX aumentou a produtividade do usuário, que reduziu o **tempo** de uma tarefa que inicialmente era executada em **cinco horas** para **quinze minutos.**

A justificativa para o tempo de quinze minutos é que todos os arquivos são editados (**.c* e **.h*), mesmo os arquivos que não contenham o padrão *tempo* são processados pelo *ed*. Quer dizer, estamos computando o tempo de carga de cada nova cópia do *ed* em cada ciclo do *for*, além da pesquisa desnecessária do padrão pelo *ed*. Outro fator que contribui para termos este tempo é o comando *g/.../gp* que imprime na tela as linhas substituidas, o que atrasa o processo global de substituição.

Você deve estar se questionando: será que os autores conseguirão realmente reduzir o tempo da tarefa acima, de **cinco horas** para **cinco segundos**? A solução acima já foi completamente automatizada, não foi? Tanto a pesquisa do padrão *tempo* em todos os arquivos quanto o fornecimento dos comando internos para o editor *ed* foram automatizados. Qual a mágica que os autores estão guardando na cartola para conseguir resolver o desafio da substituição de um padrão por outro em vários arquivos em apenas **cinco segundos**?

1.3.6 O instrutor do UNIX

Nesta seção, mostraremos duas características fundamentais para se obter maior produtividade no UNIX: processos em retaguarda e dados imediatos.

Não se intimide com a solução apresentada a seguir. Todos os conceitos e funcionalidades, abaixo apresentados, serão discutidos detalhadamente em outros capítulos ao longo deste livro. Estes detalhes avançados do ambiente UNIX aparecem aqui para justificar o ganho magnífico de produtividade que é a redução do tempo de uma tarefa de **cinco horas** para **cinco segundos.**

Se você chegou até aqui é porque já tem condições de se tornar o administrador de sua instalação UNIX. Todo administrador do UNIX deve conhecer a filosofia de uso dos comandos e algumas opções de ferramentas deste sistema operacional.

A seguir vamos apresentar uma solução digna de um instrutor do UNIX para o desafio da substituição do padrão *tempo* por outro padrão *tempo_de_resposta* em mais de uma centena de arquivos:

```
$ for i in *.c *.h
> do
>     ed - $i <<FIM_DOS_DADOS
> g/tempo/s//tempo_de_resposta/g
> w
> q
> FIM_DOS_DADOS
> done &
```

Nossa pretensão, após uma breve explicação do funcionamento da solução apresentada acima, é que você tenha apenas uma visão geral de como os comandos no UNIX podem ser conectados e como a entrada padrão de um comando pode ser facilmente manipulada. Esta explicação, aparentemente prematura, não visa afugentar noviços ou aprendizes de instrutores, pelo contrário, tem como objetivo mostrar, na prática, a Filosofia do UNIX.

O tempo total gasto para solucionar o desafio da substituição do padrão *tempo* por *tempo_de_resposta* em todos os arquivos será 20 segundos. Isso mesmo 20 segundos! Após os 20 segundos, tempo necessário para digitar o texto do comando *for* acima, a troca ainda não foi feita mas temos o terminal livre para fazer outras tarefas, por causa do & no final do comando. O tempo para o usuário é realmente 20 segundos.

Não fique nervoso se você não entendeu a solução acima. Temos vários capítulos para explicar os detalhes deste exemplo. Uma coisa você deve concordar conosco, o tempo necessário para digitar o texto abaixo:

```
$ for i in *.c *.h
> do
>     ed - $i <<FIM_DOS_DADOS
> g/tempo/s//tempo_de_resposta/g
> w
> q
> FIM_DOS_DADOS
> done &
```

não deve ser superior a 20 segundos, você não concorda? O que apareceu de novo nesta solução para termos um ganho de produtividade extraordinário?

Se formos analisar o corpo do *for* acima com mais atenção, veremos que apareceram duas novidades na ativação do comando *ed* e no uso do shell. A primeira é o surgimento da opção - que avisa a *ed* para não imprimir diagnósticos (não mostrar a quantidade de caracteres que o arquivo tem). A segunda modificação está na forma de ativar o comando interno do *ed* ***g/.../g*** (note que não tem o ***p*** no final) que faz a substituição globalmente em todo o arquivo e não imprime a linha modificada. As outras duas novidades na solução apresentadas acima são contruções internas ao shell. A primeira novidade apresentada pelo shell é o operador << que permite entrar com dados para um comando imediatamente, evitando assim a criação de arquivos temporários como */tmp/coman-*

dos, por exemplo. A segunda novidade é o operador *&* que permite que uma tarefa seja executada em retaguarda, liberando o terminal para outras tarefas.

Vamos analisar a solução acima em duas partes. A primeira trata o *for* como um todo. A segunda parte revela detalhes de funcionamento do corpo ou comandos dentro do laço *for*.

Já conhecemos o funcionamento do comando *for*. O shell monta uma lista de valores, a partir da expressão **.c *.h*, para a variável *i*. Em outras palavras, a variável *i* vai assumir os valores *im.c*, *im_desoc.c*, ..., levando em consideração que estes são os nomes dos arquivos terminados em *.c* e *.h* do diretório corrente. O & no final do comando *for* (lembre-se que o comando *for* termina depois da palavra *done*) significa que todo o laço será executado em retaguarda, deixando o terminal livre para você executar outros comandos. Esta funcionalidade do shell poder executar mais de uma tarefa simultaneamente é preciosa para aumentar nossa produtividade. Todos os comandos internos ao laço *for* serão executados em retaguarda.

A parte do comando que compõe o corpo do *for*, talvez a que mais nos interesse neste instante, é a seguinte:

```
    ed - $i <<FIM_DOS_DADOS
g/tempo/s//tempo_de_resposta/g
w
q
FIM_DOS_DADOS
```

Falamos anteriormente que o comando *ed*, por estar dentro de um laço *for*, é executado várias vezes, dependendo do número de ciclos que o laço apresentar. A cada ciclo, a variável de controle *i* do *for* assume um nome de arquivo que termina em *.c* ou *.h* e uma nova cópia do editor *ed* é executada. Se formos visualizar a execução do laço *for*, teremos o seguinte. No primeiro ciclo do *for*, a variável *i* tem o nome do arquivo *im.c* e o comando do corpo do laço *for* a ser executado é o seguinte:

```
   ed - im.c <<FIM_DOS_DADOS
g/tempo/s//tempo_de_resposta/g
w
q
FIM_DOS_DADOS
```

No segundo laço do comando *for*, a variável *i* assume o segundo valor na lista de variáveis que no nosso exemplo é *im_desoc.c*. O comando do corpo do laço a ser executado no segundo ciclo é o seguinte:

```
  ed - im_desoc.c <<FIM_DOS_DADOS
g/tempo/s//tempo_de_resposta/g
w
q
FIM_DOS_DADOS
```

O número de ciclos do laço *for* está diretamente ligado à quantidade de nomes de arquivos da lista de valores que a variável *i* vai assumir.

Observe que o laço *for* automatizou a ativação do editor *ed* para cada arquivo do diretório corrente terminado em *.c* ou *.h*. Note também que você não precisou digitar:

```
/tempo/s//tempo_de_resposta/g
w
q
```

como resposta a cada arquivo editado, como nas soluções anteriores. A construção << permitiu que dados fossem fornecidos ao comando *ed* imediatamente a cada nova ativação deste comando dentro do laço *for*. O nome deste operador (<<) é Documento imediato (do inglês **here document**). A palavra que segue o operador << é usada como delimitador para o shell saber quando os dados imediatos, para o comando *ed*, terminaram. No caso acima, *FIM_DOS_DADOS* foi a palavra escolhida para indicar fim dos dados de entrada para o comando *ed*. Podia ser outra palavra ou simbolo qualquer. Todo o texto que vier entre as linhas que contém << *FIM_DOS_DADOS* e *FIM_DOS_DADOS* será interpretado como dado de entrada para o comando *ed*, a cada nova execução do mesmo. Por causa do documento imediato (<<) o texto abaixo é usado pela entrada padrão do comando *ed* a cada novo ciclo do *for*:

```
g/tempo/s//tempo_de_resposta/g
w
q
```

Agora sim, toda a solução do problema foi automatizada. O computador fica fazendo o trabalho pesado, em retaguarda, por causa do &, e nós não precisamos mais digitar dados repetidos para o comando *ed*, graças a outra facilidade do shell <<. Isto é que é produtividade. Isto é Filosofia UNIX. Se sua solução foi parecida com esta, parabéns!, você pode se orgulhar, você será um ótimo instrutor do UNIX.

1.3.7 A força do UNIX

Na próxima solução ao desafio, queremos mostrar a flexibilidade do UNIX na criação de novas ferramentas gerais e explicar detalhes da ferramenta *troca*, por nós construída. Explicaremos alguns detalhes do shell como **sub-shell**, **variáveis** e **substituição**.

Se sua solução não ficou parecida com a anterior não há motivo para desespero. Como o shell é uma linguagem de programação, existem milhões de formas de fazer a mesma coisa. O importante é que você nunca use o **lápis vermelho** nas suas soluções. Falaremos disto depois.

Você está pronto para a solução final?

Zere o cronômetro de seu relógio e vamos ver em quantos segundos você digita o texto que aparece em destaque abaixo:

$ *troca tempo tempo_de_resposta *.c *.h &*

Se você conseguiu digitar a linha de comandos acima em um tempo menor ou igual a 5 **segundos**, nada mal, você está digitando melhor ao igual a nós. Se o seu tempo foi superior a isto, você está precisando urgentemente de um curso de datilografia, porque somos péssimos digitadores (fomos reprovados no curso de datilografia e aí tentamos informática) e mesmo assim, conseguimos uma marca de 5 segundos.

Veja a seguir a solução final ao desafio da pesquisa e troca do padrão *tempo* por *tempo_de_resposta* em mais de uma centena de arquivos:

$ *troca tempo tempo_de_resposta *.c *.h &*

1234
$

Pronto! Resolvemos o desafio em apenas 5 **segundos**. A argumentação anterior também se aplica neste caso. Vimos que 5 segundos é o tempo necessário para digitar o comando acima, a troca ainda está sendo executada em retaguarda pelo UNIX, através do processo número *1234*. Explicaremos melhor o mecanismo de processos adiante. No momento o que importa é que o tempo para o usuário é realmente 5 segundos. Depois deste tempo o terminal está livre para outras tarefas.

O comando *troca* com & no final da linha siginifica que todo o comando, depois de feita a expansão dos nomes **.c *.h*, deve ser executado em retaguarda deixando o terminal livre para outras tarefas. Quem interpreta o & é o shell corrente que tambem faz a expansão de **.c *.h* para *im.c im_desoc.c im_etic.c ....* O comando que realmente é executado em retaguarda é o seguinte:

troca tempo tempo_de_resposta im.c im_desoc.c im_etiq.c ...

Você pergunta se o comando *troca* pertence ao UNIX? Não, o comando *troca* foi criado e instalado no sistema por um usuário que conhece a Filosofia do UNIX. Nós simplesmente o usamos. Na realidade, nós o criamos porque ele é muito útil. Observamos que estávamos fazendo a mesma tarefa mais de uma vez e empacotamos a solução. O usuário pode escrever ferramentas semelhantes a *troca* e incorporá-las aos comandos do UNIX. O comando *troca*, assim como outras ferramentas que você venha a construir, podem ser colocadas em uma área pública do UNIX e ser utilizado por todos os usuários do sistema.

Mostraremos, superficialmente, alguns detalhes da ferramenta *troca*. Não nos abandone se o ***script*** ou roteiro abaixo parecer

complicado. A explicação seguinte tem como objetivo principal te dar um gostinho da Filosofia do UNIX. Não é nossa intenção assustá-lo com exemplos complexos. A Parte III deste livro traz uma visão mais detalhada dos recursos da linguagem shell usados no script *troca*. Vejamos o conteúdo da ferramenta troca:

```
$ cat troca

     PADRAO1=$1
     PADRAO2=$2
     shift 2
     for i in `grep -l $PADRAO1 $*`
     do
          ed - $i <<FIM_COMANDOS_ED
     g/$PADRAO1/s//$PADRAO2/g
     w
     FIM_COMANDOS_ED
     done
```

A ferramenta *troca* é simplesmente um arquivo texto contendo comandos na linguagem shell. Chamamos tais arquivos de scripts. Observe que *troca* recebe vários parâmetros que são passados por intermédio da linha de comandos:

```
$ troca tempo tempo_de_resposta im.c im_desoc.c im_etiq.c ...
   $0   $1          $2             $3      $4         $5
```

O comando *troca* deve ter uma sintaxe de uso, para poder ser uma ferramenta geral. Neste caso, os primeiros dois nomes passados como parâmetros são usados como os padrões de pesquisa. O primeiro é o padrão antigo a pesquisar. O segundo é o padrão que será usado para fazer a substituição. A passagem de parâmetros da linha de comandos para dentro do script *troca* é feita através de parâmetros posicionais. Internamente ao script, os parâmetros são manipulados através das variáveis especiais do shell *$* (cifrão) seguida de um número. O significado destas variáveis está descrito a seguir:

```
$0   -    nome do próprio arquivo, no caso, troca
$1   -    nome do primeiro argumento, no caso tempo
$2   -      "     segundo      "        tempo_de_resposta
$3   -      "     terceiro     "             im.c
     ...
```

Vamos explicar o funcionamento da ferramenta *troca*. Os comandos dentro do arquivo *troca* são executados como se eles tivessem sido digitados do teclado. As primeiras duas linhas:

```
PADRAO1=$1
PADRAO2=$2
```

criam duas variáveis chamadas *PADRAO1* e *PADRAO2* contendo, respectivamente *tempo* e *tempo_de_resposta*, que são os valores correntes de *$1* e *$2* associados à linha de comando *$ troca ....* Variáveis na linguagem shell são todas do tipo string (composta de

caracteres) e criadas automaticamente após uma atribuição. Existem outras variáveis especiais pré-definidas no shell, como *$** por exemplo, que representa todas as variáveis juntas que foram passadas como parâmetros para o script. O valor corrente de *$** no exemplo acima é o seguinte:

$ - tempo tempo_de_resposta im.c im_desoc.c im_etiq.c ...*

Ou seja, *$** é uma cadeia composta das palavras *$1*, *$2*, *$3*, etc. O comando *shift 2* faz com que dois argumentos originais *$1* e *$2*, já processados, desapareçam da lista de argumentos. Em outras palavras, o comando *shift 2* eliminou *tempo* e *tempo_de_resposta* da lista de parâmetros *$**. Depois do comando *shift 2* a variável *$** ficou com o seguinte conteúdo:

$ - im.c im_desoc.c im_etiq.c ...*

Outra construção que surgiu no script *troca* usa crases. Crases são usadas na linha de comandos do shell para fazer substituição na linha de comandos. Um comando é colocado entre crases. Este comando é executado previamente pelo shell. Antes de executar a linha de comandos como um todo, o shell substitui o nome do comando entre crases pelo resultado de sua execução. Só então é que toda a linha (agora já substituída) é executada. Por exemplo, o comando a seguir:

```
for i in `grep -l $PADRAO1 $*`
do
     ...
done
```

é expandido pelo shell antes de laço *for* ter início. A construção ` ... ` (leia-se crases) é usada pelo shell para fazer uma substituição. O comando entre crases ` ... ` é executado e seu resultado é substituído, no contexto, pelas crases ` ... `. Vamos acompanhar o que o shell faz antes de executar o *for* acima. O shell vai executar os comandos que estão entre crases:

`grep -l $PADRAO1 $*`

Antes de executar o comando acima, o shell processa a substituição de variáveis. A expressão *$PADRAO1* é substituída por seu valor corrente *tempo* e a variável especial *$** é substituída por seu valor corrente *im.c im_desoc.c im_etiq.c ....* O comando entre crases que realmente vai ser executado é o seguinte:

`grep -l tempo im.c im_desoc.c im_etiq.c ... `

O resultado do comando *grep* com a opção *-l* mostra apenas o nome dos arquivos que contém o padrão *tempo*. Vamos supor que apenas quatro arquivos *im.c*, *im_rel.c*, *im_estr.h* e *im_cons.h* contenham este padrão. O resultado do comando *grep* entre crases seria substituído pelos quatro nomes de arquivos:

im.c im_rel.c im_estr.h im_const.h

O comando que o shell corrente vai executar não é o *for* originalmente digitado, como mostramos abaixo:

```
for i in `grep -l $PADRAO1 $*`
do
     ...
done
```

Antes do shell, efetivamente, executar o *for* é feita a substituição das variáveis *$PADRAO1* e *$**. Depois da substituição, o comando *grep -l* é ativado e seu resultado é usado no lugar das crases ` ... `. Vejamos o *for* que o shell vai realmente executar depois das substituições:

```
for i in  im.c im_rel.c im_estr.h im_const.h
do
     ...
done
```

Observe que o mecanismo de crases no shell faz com que apenas os arquivos que realmente contenham o padrão *tempo* sejam editados. É uma funcionalidade do shell muito poderosa que otimiza nossa solução, fazendo apenas as edições dos arquivos estritamente necessários.

O corpo do laço *for* onde encontramos o comando *ed* é o mesmo da solução anterior:

```
    ed - $i <<FIM_COMANDOS_ED
g/$PADRAO1/s//$PADRAO2/g
w
FIM_COMANDOS_ED
```

Só que no lugar do padrão *tempo* nós usamos o nome de uma variável, *PADRAO1*, que contém o padrão *tempo* que foi digitado pelo usuário na ativação do comando *troca*. No lugar do padrão *tempo_de_resposta* usamos a variável *PADRAO2* que contém o padrão *tempo_de_resposta*. Este valor foi obtido a partir da linha de comandos. Antes da execução de cada comando *ed*, a cada ciclo do *for*, o shell faz a substituição das variáveis *PADRAO1* e *PADRAO2* por seus respectivos valores. O resto do funcionamento do corpo do *for* é idêntico ao laço da solução anterior. A cada ciclo do comando *for* a variável de controle *i* recebe um nome da lista de valores do *for im.c im_rel.h im_estr.h*. Por causa do documento imediato (operador <<) e após a substituição das variáveis, o *ed*, a cada execução, lê de sua entrada padrão os seguintes comandos:

```
g/tempo/s//tempo_de_resposta/g
w
```

Note que o comando *q* (quit) não é mais necessário. O próprio comando *ed* sabe que deve terminar a edição do arquivo porque chegou ao *FIM_DOS_DADOS*.

Alguns autores como [ARTH 90] atribuem a força do UNIX ao shell: "O shell é a chave para melhorar a produtividade e a qualidade em um ambiente UNIX. O shell pode automatizar tarefas repetitivas, descobrir onde você deixou as coisas, trabalhar enquanto você almoça ou dorme, além de uma série de outras atividades que lhe farão ganhar tempo. O uso do shell pode duplicar, triplicar ou quadruplicar sua produtividade, tornando-o mais rápido e mais eficiente". Concordamos com ele.

O shell é tão importante para a Filosofia do UNIX quanto os outros conceitos de ferramentas, unicidade/simplicidade e filtros, como veremos ao longo dos próximos capítulos.

PARTE I: FILOSOFIA DO UNIX

2. PEDRAS ANGULARES

Queremos discutir, neste capítulo, as principais características do UNIX. O que é um sistema multiusuário, multitarefa e o conceito de processos. Falaremos também sobre a Filosofia ou jeito de ser do UNIX.

2.1 Multiusuário

Nesta seção queremos estudar como o UNIX faz para atender mais de um usuário por vez. No UNIX, é possível que mais de um usuário por vez use a máquina. Isto é, uma máquina com uma única Unidade Central de Processamento (UCP) pode atender mais de um usuário ao mesmo tempo, como mostra a *Figura 2.1* abaixo:

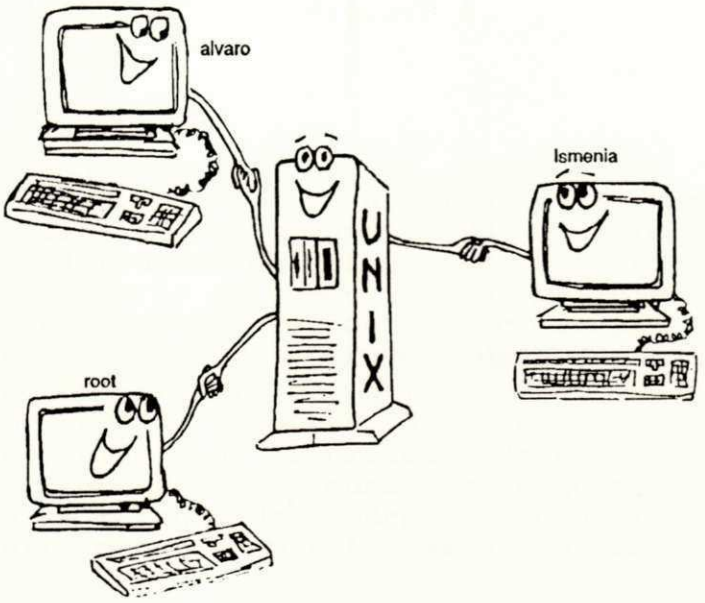


Figura 2.1: Sistema multiusuário

O UNIX é um sistema operacional **multiusuário**. Isto significa que mais de um usuário pode usar o sistema simultaneamente. O comando *who*, por exemplo, mostra quem está usando a máquina no momento, como vimos na introdução:

```
$ who

    root        tty02        Jul 20 05:59
    alvaro      tty01        Jul 20 05:48
    ismenia     tty10        jul 20 10:15
```

A saída do comando *who* indica que os usuários *alvaro*, *ismenia* e *root* estão usando a máquina neste momento.

2.2 Multitarefa

Como estamos interessados em ganhar produtividade, vamos estudar agora o que significa um sistema operacional ser multitarefa. Nestes tipos de sistema, um mesmo usuário pode executar várias atividades simultaneamente em um mesmo terminal.

O UNIX é **multitarefa** permitindo que um mesmo usuário execute mais de uma tarefa por vez. Isto é exemplificado mais facilmente através do operador **&** do shell. Esta funcionalidade do shell permite que um programa seja executado em retaguarda, liberando o terminal para outras atividades. Por exemplo, um mesmo usuário pode fazer uma compilação e procurar um determinado arquivo no sistema de forma concorrente. Por exemplo:

```
$ find / -name "lixo" -print > arquivo &

    1234
    $
```

O comando *find* acima pesquisa em todo o sistema, a partir do inicio do sistema de arquivos /, arquivos cujos nomes são iguais a *lixo* e imprime seus nomes. O número *1234* é a identificação do processo que está rodando em retaguarda. Discutiremos processos em detalhes mais adiante; no momento é o suficiente saber que existe um número sequencial, único e crescente que identifica cada atividade no computador. Através deste número poderíamos cancelar o programa *find*, se quiséssemos. O shell mostrou o número *1234* e em seguida mostrou *$* que é o caractere de prontidão. Este caractere significa que o interpretador shell está pronto para executar novos comandos. Enquanto o *find* está sendo executado em retaguarda, podemos executar qualquer comando, inclusive outro comando em retaguarda. Vejamos um exemplo:

```
$ cc prog.c &

    1236
    $
```

No exemplo acima temos dois comandos rodando em retaguarda, o *find* e o *cc*, e o terminal está livre. Como saber se os comandos cuja identificação é *1234* e *1236* já terminaram?

O UNIX possui um comando que mostra os processos em atividade. É o comando *ps*. Por exemplo, para saber minhas atividades neste instante é só digitar o comando *ps*:

```
$ ps

    PID TTY  TIME COMMAND
     44  01  0:02 sh
   1234  01  0:17 find
   1236  01  0:00 cc
   1291  01  0:01 ps
```

O comando *ps*, (ps é uma abreviação de process status), mostra quais os processos que estão rodando. No momento que o *ps* foi executado tinhamos os seguintes programas rodando associados ao usuário *alvaro* que está no terminal *tty01*: o interpretador de comandos shell (*sh*) cuja identificação é *44*, o comando *find*, a compilação *cc* e o próprio comando *ps* que emitiu o relatório acima.

A capacidade de você executar mais de uma tarefa por vez aumenta sua produtividade. Voltamos à argumentação inicial deste trabalho: conhecer os recursos do UNIX significa ter maior produtividade.

2.3 Processos

Nesta seção queremos mostrar, superficialmente, como as atividades são representadas no UNIX. Como uma única Unidade Central de Processamento (UCP) pode atender vários usuários, cada um podendo executar mais de uma tarefa por vez?

O UNIX identifica cada **processo** através de um número que é único na máquina, como foi dito anteriormente. Conforme afirma [SWART 90], muitos computadores têm apenas uma UCP (Unidade Central de Processamento), outros têm várias. Uma única UCP só pode executar apenas um processo por vez. Um computador com quatro UCPs pode rodar quatro processos de forma concorrente. O número de UCPs não interessa para o UNIX. Isto porque o UNIX é um sistema operacional de tempo compartilhado. Isto é, o UNIX guarda informações de todos os processos prontos para rodar e executa cada um por uma fração de segundos, compartilhando tempo de UCP disponível entre os processos sendo executados. Se existirem mais UCPs, existirá mais tempo de UCPs para ser compartilhado.

A saída do comando *ps* do exemplo anterior, listada novamente abaixo, é um exemplo de como as coisas são feitas no UNIX:

```
$ ps

   PID TTY  TIME COMMAND
    44  01  0:02 sh
  1234  01  0:17 find
  1236  01  0:00 cc
  1291  01  0:01 ps
```

O relatório acima mostra os processos *44*, *1234*, *1236* e *1291*, que representam, respectivamente:

- Interpretar comandos (*sh*);
- Pesquisar nomes de arquivos (*find*);
- Compilador C (*cc*);
- Mostrar o status dos processos que estão rodando (*ps*).

Toda e qualquer tarefa no UNIX é representada por um processo. Os processos compartilham a UCP. A coluna *TIME* do

relatório acima indica a quantidade de tempo de uso de **UCP** acumulado desde o início da execução de cada comando.

2.4 O lápis vermelho virtual

O **lápis vermelho** será usado aqui para exemplificar conceitos novos e revolucionários. Na realidade, queremos enfatizar o jeito UNIX de fazer as coisas.

Por que o lápis vermelho?

É uma forma simbólica de chamar sua atenção para detalhes aparentemente sem importância. Usamos o lápis vermelho como sinônimo de lentidão. Você lembra do desafio onde uma tarefa é realizada em **cinco horas** com o lápis vermelho e conseguimos reduzir este tempo para **cinco segundos**, usando a filosofia UNIX?

Voltemos ao desafio, onde tínhamos que pesquisar o padrão *tempo* e substituí-lo por *tempo_de_resposta*. A primeira solução que vem à sua cabeça, com certeza, agora, não será esta:

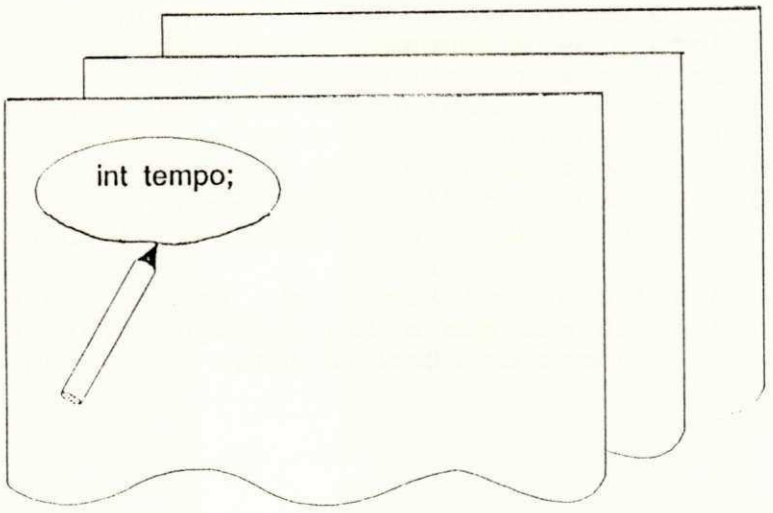


Figura 1.1: Listagem + lápis vermelho

usar o lápis vermelho para grifar os padrões em uma listagem. Mas, provavelmente, você escolherá uma solução que usa o Lápis Vermelho Virtual, onde você fará algum tipo de trabalho manual, repetitivo que poderia ser automatizado. O Lápis vermelho virtual está em nossas cabeças, isto é uma realidade. O Lápis Vermelho Virtual representa a forma manual de fazer as coisas. Às vezes fazemos as coisas automaticamente sem pensar. Por exemplo:

Quantos processos estão rodando neste instante?

Sabendo que o comando para mostrar os processos é *ps* e que para ver todos os processos temos que passar a opção *-e* como parâmetro. Poderiamos digitar o comando *ps -e* para obter o seguinte relatório abaixo:

```
$ ps -e

  PID TTY  TIME COMMAND
    0   ?  0:00 swapper
    1   ?  0:01 init
   44  01  0:02 sh
   45  02  0:01 sh
   23   ?  0:00 logger
   22   ?  0:01 update
   46  03  0:01 getty
   25   ?  0:00 cron
   41   ?  0:00 lpsched
   47  04  0:01 getty
      ( ... )
   63  5h  0:01 getty
   77  01  1:36 iwe.ovl
   94  01  0:01 ps
```

Para responder à pergunta "quantos processos estão rodando neste instante?" é só digitar o comando *ps -e* e contar as linhas que aparecerem na tela com o dedo. Cada linha representa um processo que está rodando. No caso acima temos 30 processos rodando. Você está vendo? Acabamos de usar o Lápis Vermelho Virtual de novo. Fizemos uma tarefa que não é nobre: contar. A máquina conta melhor do nós. Como poderiamos evitar o uso do Lápis Vermelho para saber quantos processos estão rodando agora?

Se você se lembra de como foi construida a ferramenta *qtd_usu*, na introdução, a outra maneira de responder à pergunta de "Quantos processos estão rodando no momento?" é a seguinte:

```
$ ps -e | wc -l

   30
```

Note que no primeiro caso, no qual o usuário usa o dedo para contar cada linha, a quantidade de linhas pode ser superior ao número máximo de linhas do vídeo, o que aumenta a possibilidade de erro na contagem. Certamente contar linhas não é uma atividade nobre. Neste exemplo o dedo representa o lápis vermelho virtual que o usuário tem na cabeça.

A segunda forma *ps -e | wc -l* usa o jeito UNIX de resolver os problemas. Desta forma não usamos o dedo ou o lápis vermelho. A máquina fez o trabalho pesado por nós. Usamos um recurso do shell | (chamado de duto ou pipe) para conectar dois comandos. O utilitário *ps -e* gera informações sobre todos os processos da máquina e o *wc -l* os conta e imprime a quantidade de linhas.

Agora que você está atento para o aspecto filosófico do ambiente UNIX, responda a seguinte pergunta: "Alvaro está usando a máquina agora? e Jacques?"

Espero que voçê associe os dois comandos que devem ser usados. O comando *who* que vai produzir os nomes de todos os usuários que estão usando a máquina, e o comando *grep* que pesquisa padrões. Enquanto você não estiver habituado à cultura ou jeito UNIX de fazer as coisas, a solução seguinte pode não fluir naturalmente em sua mente:

```
$ who | grep alvaro

    alvaro      tty01       Jul 20 05:48
```

O relatório do comando acima nos dá a resposta. O usuário *alvaro* está usando a máquina no terminal 01 desde as 5 horas e 48 minutos da manhã de 20 de Julho. Para saber se *jacques* está no ar, o comando seria o seguinte:

```
$ who | grep jacques

    $
```

A saida do comando composto *who | grep jacques* é nula, o que significa que o usuário *jacques* não está usando o sistema neste momento.

Se fizermos a mesma pergunta a outro usuário que não conheça o aspecto filosófico do UNIX, "Alvaro está usando a máquina agora?", ele provavelmente responderá digitando o comando *who* e usando o lápis vermelho virtual para encontrar a linha que contenha o nome *alvaro*.

```
$ who

    root        tty02       Jul 20 05:59
    alvaro      tty01       Jul 20 05:48
    (...)
```

Aumentar sua produtividade e quebrar o lápis vermelho virtual que existe na cabeça de alguns usuários UNIX é nosso objetivo principal.

2.5 Filosofia do UNIX

Nesta seção, apresentaremos os conceitos fundamentais do UNIX: Ferramenta de Software, Filtros e Dutos, Unicidade, Simplicidade, Interface Seca e Ortogonalidade.

Será que eu tenho um lápis vermelho virtual na minha cabeça? Será que conheço o jeito UNIX de fazer as coisas? Como andará minha taxa de produtividade?

A resposta às perguntas acima será dada por você mesmo. Você terá a resposta estudando os conceitos sobre os quais se fundamenta a filosofia do UNIX e comparando com seu jeito atual de fazer as coisas. Podemos dividir, os conceitos do UNIX da seguinte forma:

1. **Ferramenta de Software**
2. **Filtros e Dutos**
3. **Ortogonalidade**
4. **Unicidade e Simplicidade**
5. Interface Seca

Alguns programadores já utilizam estes conceitos intuitivamente. A seguir vamos discutir cada um dos conceitos acima:

2.5.1 Ferramenta de software

Aqui a palavra de ordem é produtividade. Precisamos de ferramentas para aumentar nossa taxa de produtividade. Podemos fazer uma analogia de ferramenta de software com ferramenta em uma oficina comum onde os problemas são resolvidos através da reutilização de ferramentas.

Numa oficina comum uma mesma **ferramenta** serve para apertar ou arrochar vários tipos de parafuso. Uma mesma ferramenta é usada em várias situações. Existem ferramentas **reguláveis** para se tornarem mais **genéricas**. Não tem que criar uma nova ferramenta cada vez que precisar manipular um novo parafuso, é só abrir a caixa de ferramentas, escolher a ferramenta correta, ajustar a regulagem se for o caso, e pronto. É só usar a ferramenta para solucionar o problema.

O mesmo conceito pode ser aplicado para ferramenta de software. Ao invés de construir a solução do problema a partir do nada, abra a caixa de ferramenta (a biblioteca) e utilize as ferramentas que já estão disponíveis. As opções dos comandos servem para regular a ferramenta.

Concordamos com o depoimento de [MANI 86]: "é interessante compararmos ferramenta de software com ferramenta de hardware. A construção de circuitos eletrônicos é feita de forma espetacular, enquanto o desenvolvimento de software é um problema. Hardware teve um enorme incremento na produtividade e decremento no preço, enquanto o software se torna mais ilegível e produzido com custos altos".

Reforçando o que já dissemos na introdução sobre ferramentas, vale salientar alguns requisitos que todo comando deve ter. Uma ferramenta de software deve:

1.1. usar a máquina
1.2. solucionar um problema geral
1.3. ser fácil de usar
1.4. ser usada de fato

2.5.1.1. Ferramenta de software deve usar a máquina

Por que é importante que a ferramenta de software use exaustivamente os recursos computacionais do sistema?

Existem várias razões. A seguir, destacaremos algumas:

1. libera o programador do trabalho repetitivo. As atividades manuais devem ser automatizadas. O tempo do programdor é precioso. Sua atenção deve se voltar para tarefas nobres, que exijam criatividade, tais como construção de algoritmos, estruturas de dados, codificação, etc. Um exemplo de uma ferramenta que libera o programador de tarefas repetitivas é o comando *make* do UNIX.

   O comando *make*, por exemplo, automatiza o processo de compilação de um sistema. Depois de criar uma árvore de dependência para compilação dos módulos do sistema, o programador não precisa se preocupar em saber quais arquivos devem ser compilados para gerar uma cópia executável do programa, é só digitar *make* e pronto, a geração é automática;

2. outra razão para usarmos bastante a máquina é que ela não cansa, não erra, não reclama, não cobra horas extras, não tira férias e nem entra em greve.

Em resumo, **queremos produtividade.** Para conseguirmos produtividade, como já dissemos no inicio, dividimos as tarefas em duas partes: as **tarefas nobres,** feitas por seres humanos e as **repetitivas** ou automatizáveis executadas pela máquina.

2.5.1.2. Ferramenta de software soluciona um problema geral

Queremos mostrar que as ferramentas que resolvem um problema genérico podem ser usadas por todos os usuários do sistema. O que reduz o investimento feito em software e treinamento e possibilita reaproveitamento de software.

Usar ferramentas evita trabalho desnecessário para os outros usuários. Ao instalar uma ferramenta no UNIX todos os usuários poderão usá-la. Usuários que precisem de uma ferramenta para pesquisar padrão, por exemplo, não terão que escrevê-la, é só usar o comando *grep* que já está disponivel no sistema.

Menos ferramentas são criadas do que se fizesse muitos comandos especificos, isto implica, consequentemente, em manual menor e aprendizagem mais rápida.

2.5.1.3. Ferramenta de software deve ser fácil de usar

Por que é importante que a ferramenta seja fácil de usar?

Se for complexa na interface ou na concepção ou se precisar consultar o manual a cada instante, ninguém vai usar a ferramenta.

Existe um padrão para a interface de todos os comandos do UNIX: as opções devem ser precedidas de um caractere de menos - e devem aparecer, na linha de ativação, entre o nome dos comandos e seus argumentos. Um exemplo típico de comando com opções é o seguinte:

$ ls -l arquivo

Onde *ls* é o nome do comando, *-l* é uma opção para o comando *ls* listar o arquivo *arquivo* no formato longo; *arquivo* é o parâmetro que o comando *ls* recebe para listar. O padrão acima, ou seja:

comando opções argumentos

deve ser seguido na construção de uma nova ferramenta.

Outro aspecto importante é como a nova ferramenta usa a entrada padrão e a saída padrão: será que sua ferramenta se encaixa direitinho com outros utilitários do sistema?

De uma forma geral, a nova ferramenta deve usar a entrada padrão e a saída padrão se não foi passado nenhum parâmetro na linha de comando.

2.5.1.4. Ferramenta de software deve ser usada de fato

Talvez este seja o teste mais abrangente. Depois que tudo foi dito uma única coisa realmente importa: é o teste final. A ferramenta é ou não utilizada. Às vezes, poderíamos avaliar os outros itens e achar que a ferramenta é boa, mas por algum motivo a combinação das coisas não ficou simpática e a ferramenta não é usada.

Será que o problema é o preço da ferramenta? Ou o manual que não está claro? Ou o marketing que não foi bem feito pelo autor? Ou a velocidade da ferramenta deixa a desejar? Ou a ferramenta não é necessária? ...

2.5.2 Filtros e dutos

Nesta seção, queremos mostrar que podemos fazer uma analogia entre os conceitos de filtros e dutos no UNIX com os mesmos conceitos na vida real.

2.5.2.1 Filtros

É bastante intuitivo o conceito de filtro. Todo o filtro tem uma entrada, uma saída, e faz algum processamento. Vejamos a Figura 2.2 que representa um filtro na vida real:

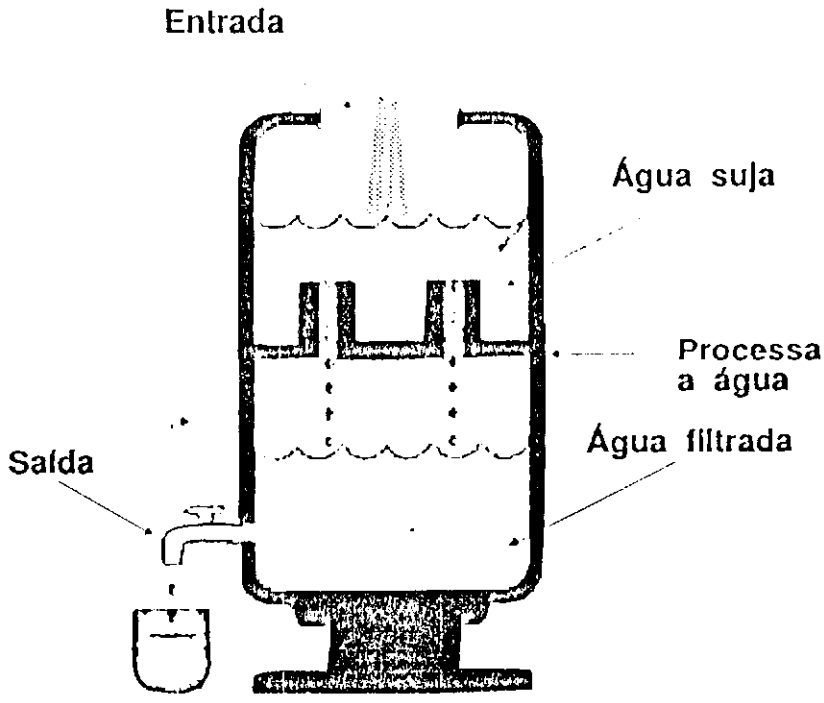


Figura 2.2: Filtro de água / filtro no UNIX

Analizando a figura acima constatamos que existe uma entrada e uma saída. O conceito de filtro no UNIX é similar. Todos os programas têm uma entrada e uma saída, que são chamadas respectivamente de Entrada Padrão (EP) e Saída Padrão (SP).

O mais simples dos filtros é o comando *cat*, que copia a Entrada Padrão à Saída Padrão. No exemplo a seguir, representado na Figura 2.3, o comando *cat* lê dados de sua entrada padrão e os grava na sua saída padrão:

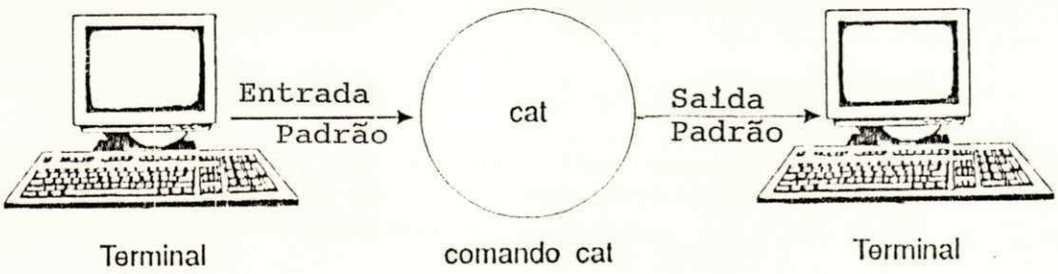


Figura 2.3: Entrada e saída do processo *cat*

Não é verdade que os valores default da Entrada e Saída Padrão são sempre iguais ao terminal. Na realidade, existe um mecanismo onde o processo a ser executado **herda** a entrada padrão e a saída padrão, conforme estudaremos mais adiante.

Todos os comandos possuem uma entrada e uma saída padrão. Um comando é caracterizado como filtro se ele usa **apenas** a entrada e a saída padrão. Como descobrir se um determinado comando é um filtro?

Uma regra geral seria digitar o comando e esperar. Se o cursor ficar parado aguardando que dados sejam digitados, significa que o comando está lendo dados da entrada padrão. Se ao invés disto o comando imprimir uma mensagem de erro significa que ele não usa a entrada padrão.

2.5.2.1.1 Exemplos de filtros

Para ser filtro, um comando deve usar a entrada e a saída padrão. O comando *cat*, por exemplo, é um filtro:

```
$ cat
_
```

Quando digitamos *cat* sem parâmetros o cursor fica esperando que dados sejam digitados do teclado. Neste caso, a entrada padrão do comando *cat* está amarrada ao teclado. Em outras palavras, o *cat* herdou a entrada padrão do seu pai, o processo shell que de alguma forma tem sua entrada padrão associada ao terminal. Aqui usamos o termo pai para designar o shell corrente. Voltaremos a falar de **processo pai** e **herança** mais adiante. O comando *cat* utiliza a entrada padrão. Da mesma forma o comando *wc* e *grep* são filtros, pois eles usam suas entradas e as saídas padrão também. Por exemplo:

$ wc

_

$ grep alvaro

_

No exemplo acima, o cursor fica esperando que os dados sejam digitados da entrada padrão. Isto significa que estes comandos utilizam a entrada padrão.

2.5.2.1.2 Exemplos de NÃO filtros

Geralmente os comandos que não são filtros ou que não usam tua entrada padrão imprimem uma mensagem de erro ou de sintaxe quando são ativados sem parâmetros. Por exemplo, o comando que ativa o compilador da linguagem C emite a seguinte mensagem, se não for passado o nome do arquivo fonte a ser compilado:

$ cc

cc: nome do programa deve ser fornecido

O comando *who*, apesar de ter uma entrada padrão não a usa. Para gerar o relatório que mostra os usuários que estão usando a máquina no momento, o *who* lê dados a partir do arquivo */etc/wtmp*. Ao contrário da maioria dos comandos que não são filtros, o comando *who* não imprime mensagem de sintaxe se não for passado nenhum parâmetro. Ao invés disto, ele mostra a relação de usuários que estão no ar.

2.5.2.1.3 Programas produtores de informações

Existem programas no UNIX que produzem informações a partir de arquivos de controle do sistema e as imprimem na saída padrão. Chamamos este tipo de programa de fonte, porque são fonte de informação. Exemplos destes programas:

$ who

```
ismenia     tty03        Jul 20 08:12
root        tty02        Jul 20 05:59
infosol     tty05        Jul 20 08:13
sandra      tty04        Jul 20 08:12
alvaro      tty01        Jul 20 05:48
```

$ ps

```
  PID TTY  TIME COMMAND
   44  01  0:02 sh
  144  01  0:01 ps
```

Ao digitar *who* ou *ps* os dados são gerados na saída padrão. Comandos geradores de informação, geralmente, não esperam que dados sejam digitados do teclado. No caso do exemplo acima, os comandos *who* e *ps* usam os arquivos */etc/wtmp* (no caso de *who*), */UNIX* e */dev/kmem* (no caso de *ps*) como entrada para produzir seus relatórios na saída padrão.

2.5.2.1.4 Programas sorvedouros

Ao contrário dos programas geradores de dados, os programas sorvedouros ou consumidores lêem dados da entrada padrão e escrevem em um arquivo pré-estabelecido. Exemplos de programas sorvedouros:

```
$ lp

_

$ mail alvaro

_
```

No exemplo acima, os dois comandos *lp* e *mail* aguardam que dados sejam digitados da entrada padrão. O comando *lp* vai gravar dados em arquivos que serão usados para imprimir na impressora. O utilitário *mail* vai escrever os dados lidos no arquivo */usr/spool/mail/alvaro*.

2.5.2.2. Dutos

Queremos discutir aqui, o conceito de duto ou pipe no UNIX.

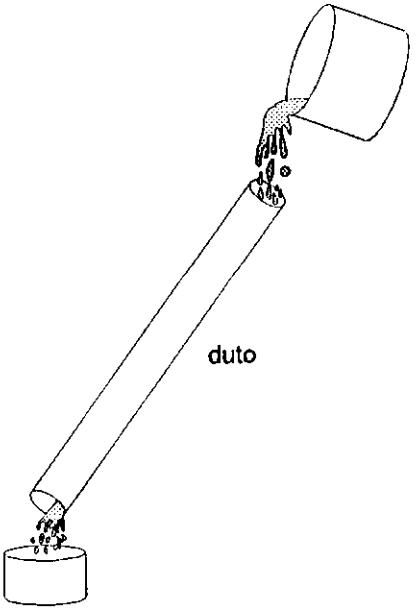


Figura 2.4: Representação gráfica de um *duto*

Podemos também fazer uma analogia com a vida real para explicar dutos no UNIX. O conceito de duto na vida real é muito intuitivo. É o cano onde passa água entre dois reservatórios na figura acima.

No UNIX o duto pode ser visto como um cano mesmo cuja finalidade é canalizar dados entre processos ao invés de liquidos entre reservatórios. O duto é representado pelo símbolo **|**. O interpretador de comandos shell interpreta a linha de comando *who | wc -l* e sabe que tem que pedir ao núcleo do UNIX para criar o duto. Antes de executar os comandos *who* e *wc* o shell conecta a saída padrão do *who* ao duto **|** e a entrada padrão do *wc -l* também ao duto **|**.

Filtros facilitam a implementação do conceito de ortogonalidade. Os comandos continuam lendo da sua entrada padrão e escrevendo na sua saída padrão, como antes. Para usar dutos não precisamos redefinir a entrada ou a saída padrão.

Mais adiante veremos outras formas como o shell usa dutos para implementar novas funcionalidades.

2.5.3. Unicidade e simplicidade

Aqui queremos mostrar que é fundamental que cada ferramenta faça apenas **uma** coisa. Ferramentas que fazem apenas uma coisa tendem a ser mais simples de implementar e manter, terem manuais claros e uma interface de fácil manuseio.

Ferramentas que não têm unicidade provavelmente não poderão se encaixar com outros comandos, porquê?

Vamos ilustrar a resposta mostrando um exemplo de comandos que fazem apenas uma coisa e de utilitários que fazem mais de uma tarefa. Os comandos *ps*, *who* e *wc* fazem apenas uma coisa, mostram os processos, listam os usuários no ar, e conta coisas, respectivamente. As três combinações abaixo:

1. *ps -e | wc -l*
2. *who | wc -l*
3. *who | grep alvaro*

fazem o seguinte: (1) conta o número de processos; (2) mostra o número de usuários no ar; e (3) verifica se o usuário *alvaro* está usando o sistema.

Os comandos acima *ps*, *who*, *grep* e *wc* puderam ser combinados livremente, porque cada um deles, isoladamente, faz apenas uma coisa e utilizam a entrada e (ou) a saída padrão.

Vamos supor, só para ilustrar, que exista um comando que faça simultaneamente duas coisas, por exemplo o comando *who2*:

$ who2

```
        RELAÇÃO DE USUARIOS NO AR
-----------------------------------
sandra      tty03        Jul 21 07:35
ismenia     tty02        Jul 21 07:35
alvaro      tty01        Jul 21 07:10
jacques     tty04        Jul 21 07:36
root        tty05        Jul 21 07:35
-----------------------------------
Menor horário ..............: 07:10
Maior horário ..............: 07:36
```

No exemplo acima o comando *who2* faz duas coisas. Mostra usuários e imprime uma estatistica. Sua saida também é mais bonitinha que a saida do comando *who*. Agora, será que *who2* é mais útil que o *who*? Se precisássemos saber o número de usuários que estão no ar a combinação *who2 | wc -l* imprimiria *10* usuários, o que não reflete a verdade. As duas linhas do cabeçalho:

```
        RELAÇÃO DE USUARIOS NO AR
-----------------------------------
```

e as três linhas da estatistica:

```
-----------------------------------
Menor horário ..............: 07:10
Maior horário ..............: 07:36
```

foram contadas como se fossem usuários, o que não está correto.

Fatalmente, se uma ferramenta resolve dois problemas **X** e **Y**, e se num determinado momento precisarmos solucionar o problema Y, provavelmente não conseguiremos usar a ferramenta que faz as duas coisas **X** e **Y**.

Se uma ferramenta faz **X** e **Y**, é muito provável que tenhamos **outras** ferramentas fazendo também **X** ou **Y**. Isto duplica a funcionalidade, confunde o usuário, que tem que aprender como fazer Y de várias formas, complica o manual, etc.

O fato de ter unicidade ou de forçar ou desejar unicidade, reforça a simplicidade. Uma ferramenta que só faz uma coisa tende, naturalmente, a ser mais simples do que outra que faz várias coisas.

De forma geral no UNIX, cada ferramenta faz apenas **uma** coisa **bem feita**. Por exemplo, o compilador só gera código, mas gera código de boa qualidade (otimizado).

Vamos analisar um comando do UNIX que faz apenas uma coisa bem feita. O comando *grep* que só faz pesquisa de padrões, por exemplo:

$ grep alvaro /etc/passwd

alvaro::100:51::/u/alvaro:/bin/sh

No exemplo acima, o comando *grep* pesquisa o padrão *alvaro* no arquivo */etc/passwd*, imprimindo as **linhas** onde ocorre o padrão *alvaro*.

Outro exemplo:

*$ grep -l alvaro /etc/**

/etc/group
/etc/passwd
/etc/wtmp

No exemplo acima, o comando *grep* pesquisa o padrão *alvaro* em todos os arquivos do diretório */etc*, imprimindo apenas os **nomes dos arquivos** onde ocorreu o padrão *alvaro*. Finalmente, mais um exemplo:

$ grep -v alvaro /etc/passwd

root::0:0::/:/bin/sh
sysadm::0:0::/usr/sysadm:/bin/sh
...

No exemplo acima, o comando *grep* pesquisa o padrão *alvaro* no arquivo */etc/passwd*, imprimindo as linhas onde **não** ocorre o padrão *alvaro*.

O comando *grep* faz apenas uma coisa: **pesquisa** padrões em arquivos. As opções *-l* e *-v* são uma forma de **regular** a ferramenta geral. A opção *-v* imprime na saída as linhas onde **não** houve casamento do padrão. O comando *grep* ativado com a opção *-l* imprime apenas o **nome do arquivo** que contém o padrão pesquisado.

Concluindo, *grep* tem apenas uma função: pesquisar padrão. Agora ele faz este serviço bem feito, permitindo uma variação no relatório resultante da pesquisa.

2.5.4. Interface seca e ortogonalidade

A capacidade de programação do shell e a forma bem definida com que os conceitos do UNIX são construídos elevam muito sua produtividade.

A interface seca do UNIX é proveniente do shell e dos utilitários do UNIX. Seca porque ela não é nada amigável, não é gráfica nem baseada em menus. A interface do UNIX foi projetada para programas e não para o homem. A produtividade é o objetivo maior desta interface.

Ortogonalidade é a propriedade que os conceitos do UNIX

possuem de serem perpendiculares, bem definidos, retos. A Ortogonalidade do UNIX tem suas vantagens. Para fazer redirecionamento de um comando, por exemplo, a forma é uma só, independente de querermos gravar em um arquivo ou escrever em uma impressora, ou unidade de fita, ou disco, etc.

Simplicidade mais ortogonalidade no UNIX implica em produtividade através de ferramentas. Para o programador a ortogonalidade é desejável acima de tudo.

Qual é a interface mais apropriada para permitir sua implementação? Uma interface seca ou baseada em menus?

Reflita um pouco e responda a pergunta: qual o tipo ideal de interface que você escolhe? Se uma interface amigável, baseada em menus ou se um interface menos confortável com maior capacida-de produtiva. A resposta é: qualquer interface que permita a **junção** de ferramentas. Interface de utilitários (ferramentas) deve ser projetada para programas e não para seres humanos.

Como queremos produtividade somos forçados a esquecer o seguinte:

1. Interface totalmente baseada em menus;
2. Interfaces interativas;
3. Interfaces com saídas bonitinhas (cabeçalhos, etc).

Se usassemos qualquer uma das três alternativas acima, poderíamos esquecer da combinação de ferramentas !!. Na interface tipo (1) não há possibilidade de automação de tarefas; em interpretadores tipo (2) a interação prejudica a combinação de comandos; e em (3) os cabeçalhos, etc, não permitem a combinação de comandos. Você se lembra do comando *who2* que não pôde ser combinado?

Resumindo, multiprogramação não combina com essas interfaces. O **programador** quer multiprogramação por causa da **produtividade.**

É claro que existem outros tipos de usuários, não programadores, que precisam de outro tipo de interface. Não negamos isso. Porém, este trabalho é **dedicado ao programador** e não ao usuário leigo. UNIX precisa dos dois tipos de interfaces (e tem).

PARTE II: FILOSOFIA DO UNIX

3. EXECUÇÃO DE COMANDOS

Neste capitulo, queremos mostrar alguns aspectos do shell como interpretador de comandos. Veremos como é feita a expansão e a execução da linha de comandos. Veremos também exemplos do uso dos caracteres * (asterisco), aspas e contra-barra.

O shell original do UNIX se chama Bourne Shell ou *sh*, cujo autor é o Sr. Bourne. Existem outros shells que já são distribuidos em novas versões do UNIX, como o C shell (*csh*) e o Korn shell (*ksh*). A letra C aqui lembra a linguagem C.

Este capitulo se aplica a todos os shells do UNIX.

3.1 O interpretador shell

É importante observar que é o shell quem interage com o núcleo do UNIX. O shell é, ao mesmo tempo, um interpretador de comandos e também uma linguagem de programação. O shell é o responsável pela interação homem-computador.

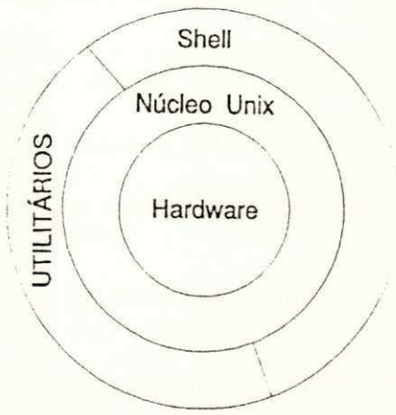


Figura 3.1: O interpretador shell

Diferente de outros sistemas operacionais, o UNIX trata o interpretador de comandos shell sem privilégios. O shell tem o mesmo status para o núcleo do UNIX que qualquer outro comando (*ls*, *who*, ...).

Supondo o shell em uso interativo em um terminal suas funções básicas são:

1. expansão da linha de comandos; e
2. "parsing" da linha de comandos.

3.2. Expansão da linha de comandos

Aqui vamos mostrar que o comando digitado por você não é diretamente executado. O shell verifica se na linha tem algum caractere especial e o processa antes que o comando seja, efetivamente, executado.

Os argumentos são expandidos pelo shell antes de serem executados. Existem alguns caracteres com significado especial para o shell. O "*" é um destes caracteres. Por exemplo, o comando:

$ ls arq.c*

é expandido pelo shell, antes que o comando *ls* seja executado, para:

$ ls arquivo1.c arq_tela.c arqtexto.c arqa.c arq1.c

supondo que existem estes arquivos *arquivo1.c, arq_tela.c, arqtexto.c, arqa.c* e *arq1.c* no seu diretório de trabalho.

3.3. Parsing da linha de comandos

É o shell quem faz o reconhecimento da linha de comandos. Os caracteres usados como final de linha são:

'\n' - caractere nova-linha.

A linha é separada em palavras. Os caracteres usados como separador de palavras são:

' ' - caractere branco ou espaço;
'\t' - caractere de tabulação.

A primeira palavra é interpretada pelo shell como sendo o comando a ser executado. As demais palavras são usadas como argumentos que o comando recebe.

$ comando arg1 arg2 arg3

Argumentos

Nome do comando

O UNIX distingue letras maiúsculas de minúsculas. O comando *LS* maiúsculo, por exemplo, é diferente do comando *ls* minúsculo, que é diferente dos comandos *Ls* e *lS*.

3.4 Aspas e contra-barra

Mostraremos a seguir uma forma de proteger caracteres especiais contra sua expansão natural pelo shell.

Os caracteres (") e (\) são usados para desfazer o efeito especial de alguns caracteres do UNIX. Por exemplo:

$ echo alo mundo > arq

$

o comando *echo alo mundo > arq* faz com que o shell redirecione a mensagem (*alo mundo*) criando um arquivo *arq*. A mensagem *alo mundo* foi armazenada em *arq*, conforme mostra o comando *cat* abaixo:

$ cat arq

alo mundo

Se colocarmos o caractere especial (>) entre aspas ">" ou precedido de contra-barra \>, o caractere > perderá seu efeito especial. Vejamos um exemplo:

$ echo alo mundo ">" arq

o comando (*echo alo mundo ">" arq*) ou (*echo alo mundo \> arq*) não cria o arquivo *arq*. O comando *echo* acima simplesmente imprime a seguinte mensagem:

alo mundo > arq

na saída padrão que é o vídeo. O caractere > perdeu seu efeito especial por causa da contra-barra ou das aspas. As aspas ou a contra-barra eliminam o efeito especial de todos os metacaracteres do shell. Metacaractere é o nome dado aos caracteres com significado especial para o shell.

Os caracteres (") e o (\) são interpretados pelo shell. Falaremos mais adiante de outras aplicações destes metacaracteres.

3.5 Símbolo de prontidão do shell

O shell interpreta comandos do usuário. O shell imprime um caractere no terminal para informar o usuário que está pronto para executar outro comando. O símbolo de prontidão do shell é normalmente o *$* (cifrão).

O shell permite que comandos sejam digitados em mais de uma linha física. Vamos analisar o seguinte exemplo:

```
$ echo "alo
> mundo
> novo"
```

onde o comando *echo* imprimirá a seguinte mensagem na tela de seu terminal:

alo
mundo
novo

O shell possue um **caractere secundário de prontidão** (>), que serve para avisar o usuário que seu comando é composto de mais de uma linha física. O caractere de nova-linha dentro das aspas "..." perdeu seu efeito especial de fim-de-comando. O caractere de fim-de-linha entre as aspas serviu para imprimir a mensagem em várias linhas.

3.6 Execução de Comandos

Como resultado do parsing, o shell executa a primeira palavra como comando passando as demais palavras como argumento.

O símbolo (-) não é um caractere especial para o shell. É apenas uma convenção que foi adotada por todos os comandos do UNIX, para indicar que o argumento é uma opção.

```
$ ls -l          (listagem longa)

$ wc -c          (conta caracteres)

$ rm -i          (remoção interativa)
```

Figura 3.2: Exemplos de opções para comandos

A convenção de que palavras precedidas do caractere menos (-) é uma opção foi a forma encontrada para regular uma ferramenta geral. A ferramenta continua fazendo apenas uma coisa; o que muda são detalhes tais como o formato do relatório de saída, por exemplo. Um exemplo concreto é o comando *ls*. Ele mostra os nomes dos arquivos. Já o comando *ls -l* mostra o nome dos arquivos no formato longo, com tamanho dos arquivos em bytes, data, e outras informações.

PARTE II: QUESTIONANDO CONCEITOS FAMILIARES

4. ARQUIVOS E DIRETORIOS

Neste capitulo vamos discutir, superficialmente, alguns conceitos familiares do UNIX: arquivos, diretórios, nomes de arquivos relativos ou absolutos e os principais diretório de um sistema UNIX. Apesar de não serem conceitos avançados, preferimos expor a matéria para nivelar os conhecimentos dos leitores. Fique à vontade para pular este capitulo.

4. Arquivos e Diretórios no UNIX

4.1 Arquivos

Apesar de concordarmos que os arquivos têm um papel fundamental no UNIX, não queremos nos aprofundar na discussão sobre arquivos aqui nesta seção. Falaremos superficialmente de arquivos e diretórios. Citaremos os principais diretórios do UNIX e seus respectivos conteúdos.

Os arquivos são representados no UNIX por nomes. Podemos criar um arquivo simplesmente redirecionando a saída de um comando com o caractere (>). Por exemplo, para criarmos um arquivo chamado *arq_saida*, basta digitar o comando:

```
$ who > arq_saida

$
```

para observarmos o conteúdo deste arquivo *arq_saida*, temos que usar o comando:

```
$ cat arq_saida
```

e veremos na tela o conteúdo do arquivo *arq_saida*:

```
alvaro     tty01        Jul 21 14:43
   ( ... )
```

Os arquivos no UNIX são formados por uma parte simbólica, que é o nome, alguns atributos (tamanho, dono, permissões, etc) e o conteúdo do arquivo propriamente dito. Geralmente os nomes dos arquivos podem ter até quatorze caracteres em versões do UNIX derivada da AT&T. Já nos sistemas UNIX baseados na versão Berkeley os nomes podem chegar a ter o tamanho de 256 caracteres. O tamanho máximo do conteúdo do arquivo está normalmente limitado à capacidade do dispositivo físico utilizado tal como um disco, por exemplo. Por outro lado, havendo um dispositivo adequado, o limite máximo do tamanho do arquivo é de 4 Gigabytes.

4.2. Diretórios

Queremos mostrar que através de diretórios é possível classificar a informação introduzida no computador de modo a facilitar sua manipulação.

Sistemas operacionais antigos mantinham todos os nomes dos arquivos em um mesmo local, o que tornava lento o processo de pesquisa de um determinado nome no diretório.

O ser humano precisa classificar e organizar as informações para diminuir sua complexidade. O UNIX resolve este problema

criando uma ferramenta organizacional agrupadora chamada **diretório**.

4.2.1 Nomes de diretórios

Os diretórios são representados, externamente, por nomes. Assim como os arquivos, os diretórios têm uma parte simbólica, atributos (tamanho, dono, permissões, etc) e um conteúdo. Através da parte simbólica, que são os nomes, você consegue organizar suas informações.

Os diretórios podem conter sub-diretórios que são agrupados em uma estrutura hierárquica semelhante a uma árvore.

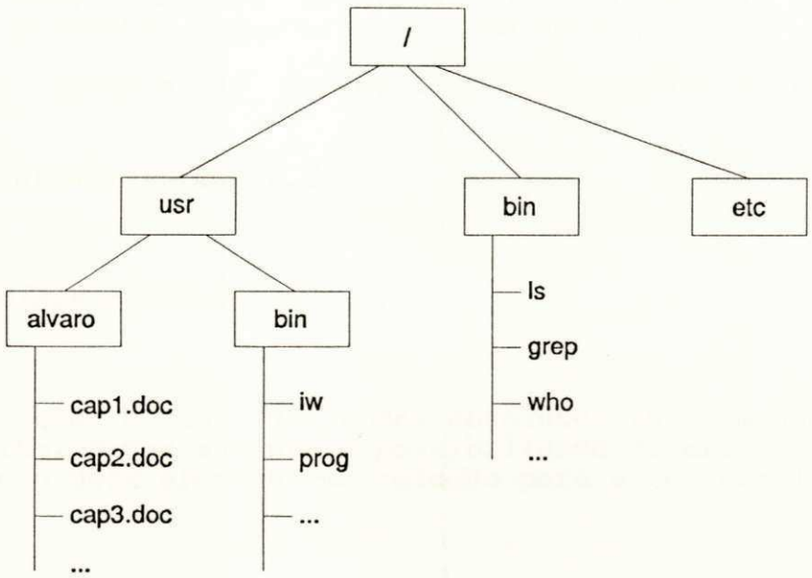


Figura 4.1: Sistema de arquivos UNIX

Como mostra a figura acima, os arquivos no UNIX podem ser organizados conforme uma árvore invertida, onde a raiz seria o diretório (/). Não há limite horizontal nem vertical para o crescimento da árvore de diretórios, embora haja uma penalidade de desempenho para o crescimento horizontal.

Na Figura 4.1 o que estiver dentro das caixas são nomes de diretórios e os outros nome são arquivos. O diretório */usr* é reservado para informações do usuário; o diretório */bin* é o local dos comandos e o diretório */etc* é o diretório dos arquivos de configuração do UNIX.

4.2.1.1. Nomes absolutos

Podemos referenciar os arquivos dentro dos diretórios de

duas formas. A primeira é através de nomes absolutos, onde temos que escrever o nome do percurso completo de todos os diretórios, desde o diretório raiz até o nome do arquivo desejado. Por exemplo, para editar o arquivo que contém este texto temos que digitar o comando:

$ vi /usr/alvaro/cap1.doc

ou seja, tivemos que digitar o percurso completo. O comando *vi* é um dos editores de texto do UNIX.

Observe que o nome simbólico / pode ser usado com dois significados distintos:

1. Como diretório raiz
2. Como caractere separador de diretórios (serve para delimitar o percurso do nome do arquivo)

Outro exemplo de manipulação dos arquivos usando os nomes absolutos:

$ ls -l /usr/alvaro/cap06.doc

...

$ cat /usr/alvaro/cap01.doc

...

Note que o uso de nomes absolutos faz com que o usuário digite muito, o que aumenta a possibilidade de erro de digitação. Felizmente o UNIX oferece uma solução para este problema: nomes relativos.

4.2.1.2. Nomes relativos

Nomes relativos levam em consideração o **diretório corrente** do usuário. O diretório corrente é aquela área onde estamos trabalhando no momento. Para mudar o diretório corrente o usuário deve usar o comando *cd*, uma abreviação do inglês (change directory) muda de diretório.

O esquema relativo evita digitação desnecessária aumentando a produtividade do usuário. Por exemplo, para trabalhar no diretório */usr/alvaro*, digitamos o comando a seguir:

$ cd /usr/alvaro
$ vi cap01.doc
...

$ ls -l cap06.doc
...
$ cat cap01.doc
...

Observe no exemplo acima que não precisamos usar o nome absoluto para ver o conteúdo do arquivo */usr/alvaro/cap01.doc*, como seu diretório corrente já é */usr/alvaro*, você pode simplesmente usar o comando *cat cap01.doc*.

A forma como o UNIX implementa a árvore de diretórios é interessante. O arquivo que o UNIX utiliza para armazenar o diretório tem o seguinte formato:

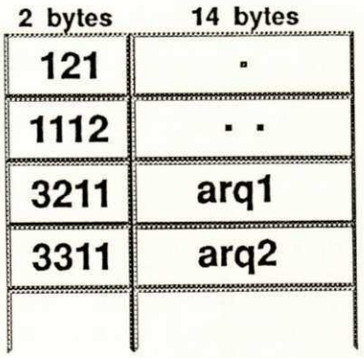


Figura 4.2: Layout de diretórios

São entradas de tamanho fixo de dezesseis bytes, onde os dois primeiros bytes armazenam um número que é usado para manipular o conteúdo do arquivo. Os quatorze bytes restantes armazenam o nome simbólico do arquivo.

4.2.1.3. Diretórios "." e ".."

Existem dois nomes simbólicos de arquivos com significado especial para o UNIX. São os arquivos "." e o arquivo "..", representando respectivamente, o diretório corrente e o diretório imediatamente anterior ao diretório corrente, o diretório "pai". A implementação da árvore de diretório no UNIX só é possível graças a estes dois nomes simbólicos "." e "..".

Usuários ou programas quaisquer podem usar nomes relativos ou absolutos. Os nomes relativos podem conter os nomes de diretórios especiais "." e "..". Por exemplo, para ir trabalhar no diretório */usr/alvaro* basta digitar o comando abaixo:

$ cd /usr/alvaro

Para ver o conteúdo do arquivo *cap01.doc*, por já estar no diretório */usr/alvaro*, é só digitar o seguinte comando:

$ cat cap01.doc
...

Ou seja, o comando *cat cap01.doc* usa o arquivo *cap01.doc* do diretório corrente que é */usr/alvaro* de forma implícita. Se quisermos explicitar o diretório corrente usamos o ".". Por exemplo, o comando abaixo tem o mesmo significado que o comando anterior:

```
$ cat ./cap01.doc
          ...
```

No exemplo acima temos o nome relativo do arquivo *cap01.doc* envolvendo o diretório corrente ".".

Outro exemplo de nomes relativos é quando queremos usar um arquivo que está em outro diretório e não queremos usar o nome ou percurso absoluto do arquivo. Por exemplo, vamos supor que você está trabalhando no diretório */usr/alvaro/trabalho/fontes/c*:

```
$ cd /usr/alvaro/trabalho/fontes/c
```

Se você quer ver o conteúdo de um arquivo que está no diretório de */usr/alvaro/trabalho/doc*. Uma forma é usar o nome de diretório especial ".." que quer dizer diretório anterior. O comando relativo seria: volte para o diretório anterior "..", entre no diretório *doc* e use o arquivo *cap03.doc*, por exemplo:

```
$ cat ../doc/cap03.doc
          ...
```

Podemos usar o nome do diretório anterior ".." várias vezes em um mesmo percurso. Supondo que estamos no diretório corrente: */usr/alvaro/trabalho/fontes/c* e queremos ver o conteúdo de um arquivo no diretório */usr/alvaro* através de nomes relativos:

```
$ cat ../../../arquivo
```

Podemos citar algumas vantagens do diretório corrente:

1. Diminui o tamanho dos nomes de arquivos facilitando sua digitação;

2. Focaliza a atenção do usuário para as informações relevantes. A seleção do diretório corrente já elimina do visual do usuário outras informações desnecessárias para o momento.

A mudança de diretório é feita através do comando *cd*. O comando *cd* é interno ao próprio shell.

4.2.2. Diretório HOME

Como vimos na seção anterior, sempre estamos trabalhando em um diretório corrente.

Ao iniciar uma sessão no UNIX temos que ter um diretório

inicial de trabalho. Este diretório é criado pelo administrador do sistema, quando somos catalogados na máquina, e se chama diretório HOME.

O interpretador shell sabe que deve abrir um arquivo especial de configuração de usuários */etc/passwd* e descobrir, para cada usuário, qual é seu diretório HOME. Isto porque todos os programas no UNIX devem ter um diretório corrente.

Todos os programas que são executados **herdam** o diretório corrente do shell. Programas criados a partir do shell são chamados **filhos** do shell. Os programas **filhos** do shell podem mudar o seu diretório corrente. A mudança de diretório no processo filho não altera o diretório corrente do processo pai, ou seja, o shell corrente.

Falaremos sobre o mecanismo de execução de processos e dos seus ambientes mais adiante.

4.3. O Sistema de arquivos UNIX

Queremos mostrar nesta seção os principais diretórios que compõem um sistema de arquivo típico do UNIX.

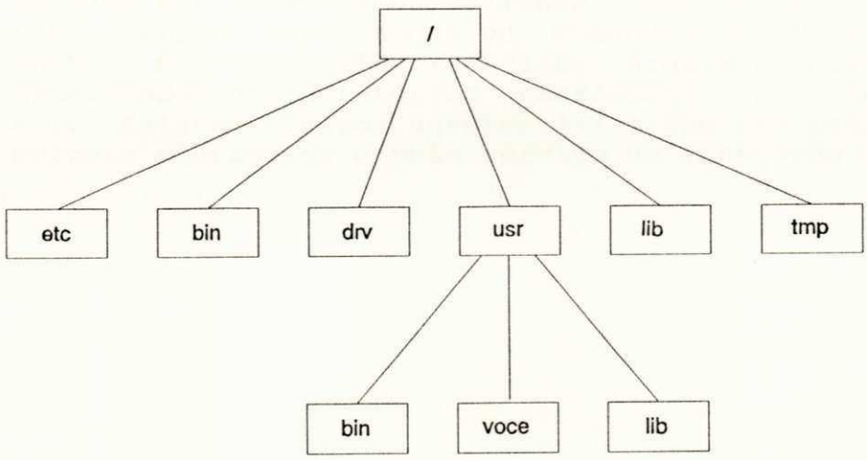


Figura 4.3: Arvore típica

Toda máquina que roda o sistema operacional UNIX tem uma árvore de arquivos com disposição semelhante à figura acima. A seguir listamos os principais diretórios com uma descrição breve de seu conteúdo:

/bin O diretório */bin* é o local do sistema onde se alojam os

utilitários que são distribuídos junto com o UNIX;

/dev O UNIX trata arquivos e dispositivos de forma idêntica, o que facilita a vida do programador. Para cada dispositivo existe um nome. O nome de um dispositivo pode residir em qualquer diretório. Por uma questão de padronização, o UNIX convencionou que todos os nomes de dispositivos deveriam ficar no diretório */dev*. O nome */dev* é uma abreviação do inglês (device), que quer dizer dispositivo em nossa língua.

/etc Comandos usados pelo administrador do sistema e os arquivos de configuração do sistema se localizam neste diretório;

/lib Algumas linguagens são distribuídas junto com o sistema operacional UNIX. A linguagem C é um exemplo. O diretório */lib* serve para armazenar as bibliotecas destas linguagens;

/tmp O diretório */tmp* é uma área livre usada por todos os usuários e comandos que precisam de espaço temporário em disco;

/usr Este é o diretório do usuário. Aqui são arquivadas informações diversas sobre usuários. Novos comandos criados por usuários normalmente ficam em um subdiretório */usr/bin*. Bibliotecas ou comandos auxiliares são postos no diretório */usr/lib*. Arquivos pessoais são armazenados nos diretórios */usr/fulano*, */usr/beltrano*, etc. Existem outros subdiretórios que são usados pelo correio eletrônico e pelo serviço de impressão do UNIX.

PARTE II: QUESTIONANDO CONCEITOS FAMILIARES

5. ENTRADA PADRÃO, SAIDA PADRÃO E REDIRECIONAMENTO

Vamos estudar neste capitulo como é implementado o conceito de arquivos padrão e redirecionamento. Estes são, seguramente, os conceitos-chave mais importantes que permitem implementar a ortogonalidade do UNIX. Falaremos também sobre a comunicação entre o núcleo do UNIX e o terminal, já que o terminal é o arquivo padrão mais utilizado.

5. Entrada padrão, saída padrão e redirecionamento

5.1. Arquivos padrão

Para descrevermos arquivos padrão temos que falar dos descritores padrão. Todo processo do UNIX tem uma tabela de descritores de arquivos. Por enquanto, podemos considerar um processo como sendo um programa em execução. Processos sempre tem os descritores de arquivos abertos para leitura ou gravação, conforme a figura abaixo:

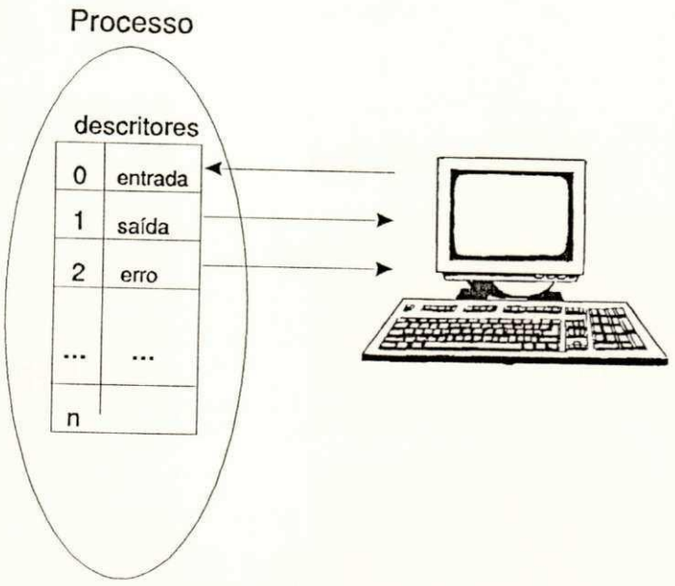


Figura 5.1: Descritores padrão

Alguns descritores de arquivos estão sempre abertos (alguns processos formam uma exceção a essa regra. Eles serão mencionados oportunamente). É o caso, por exemplo, dos descritores 0, 1 e 2, usados respectivamente para a entrada padrão, saída padrão de mensagens normais e a saída padrão de mensagens de erro. Você já descobriu o porquê da existência destes descritores padrão estarem sempre abertos?

Os arquivos padrão são de fundamental importância para a filosofia do UNIX. É através deste mecanismo que o UNIX consegue combinar utilitários, criando novas ferramentas. O programa não se preocupa de onde lê ou para onde grava informações; ele simplesmente usa os arquivos padrão para fazer entrada e saída. A amarração destes descritores a arquivos reais é feita em tempo de execução. Os arquivos padrão são ortogonais e versáteis. Um exemplo disto é o comando *cat* que pode ler e gravar nos mais variados dispositivos.

5.2 Abertura dos arquivos padrão

Aqui queremos dar uma visão geral da hierarquia de execução dos processos do UNIX, desde a ativação (ligação física do computador) do sistema até que o shell seja acionado.

5.2.1 O Gerenciamento de processos

Voltaremos a falar de processos de forma mais detalhada adiante. Aqui usaremos o conceito de processos apresentado na introdução onde cada processo representa um programa em execução no UNIX. A Figura 5.2 nos mostra a sequência de execução de processos para você abrir sua sessão:

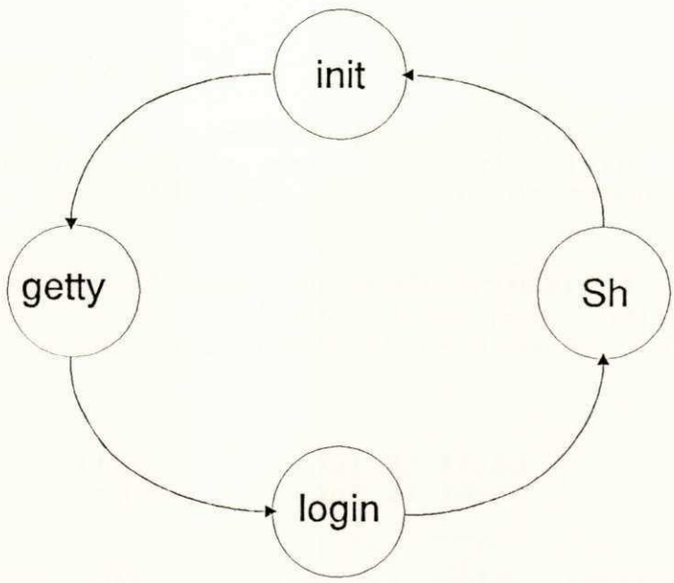


Figura 5.2: O shell de login

Existe um processo que é responsável pela criação dos demais processos. Ele é chamado *init* ou processo 1. O processo 1 consulta um arquivo de configuração, o */etc/inittab*, e para cada terminal configurado, cria um processo filho que executa o programa */etc/getty*. Este programa imprime a mensagem *Identificação:* ou *Login:* no terminal, esperando então que o usuário entre com seu nome. O papel do *getty* é acertar os parâmetros de comunicação do terminal (velocidade, paridade, login.

O comando *getty*, de posse do nome do usuário, ativa outro comando, o */etc/login*, passando como parâmetro o nome que o usuário digitou na identificação. A finalidade do programa *login* é ler a senha e verificar se o nome do usuário confere com a senha digitada. Se tudo correr bem, ou seja, se a senha estiver correta o programa */bin/sh* é executado tendo como diretório

corrente o nome que está cadastrado no arquivo */etc/passwd* no sexto campo. O arquivo */etc/passwd* tem linhas divididas em campos separados pelo caractere **:** (dois pontos). Veja um exemplo de uma linha do arquivo de senhas do UNIX */etc/passwd*, onde o usuário *alvaro* está cadastrado no diretório */usr/alvaro*:

```
$ cat /etc/passwd

      ...
alvaro::0:51::/usr/alvaro:/bin/sh
      ...
```

Quando o shell (*sh*) começa a executar, os arquivos padrão já foram abertos. A partir deste instante, todos os processos filhos do shell herdam todos os decritores do shell, em particular os descritores 0, 1 e 2, os três correspondendo ao terminal.

5.3 Redirecionamento

Normalmente os comandos que utilizam a saída padrão emitem suas mensagens na tela de seu terminal. Isto porque a saída padrão está associada ao terminal, no momento da execução do comando.

Uma forma de alterar ou redirecionar a saída padrão é através do caractere > interpretado pelo shell. O comando *who*, por exemplo, emite sua listagem na tela da seguinte forma:

```
$ who

alvaro     tty01        Jul 27 14:53
jacques    tty5c        Jul 27 15:07
        ...
```

O mesmo comando acima pode ter sua saída redirecionada para o arquivo *arq* se usarmos o caractere especial > (interpretado pelo shell):

```
$ who  > arq
```

Nada é impresso na tela como resultado do comando acima pois a saída padrão foi redirecionada para o arquivo *arq*. Se formos olhar o conteúdo do arquivo *arq*, veremos:

```
alvaro     tty01        Jul 27 14:53
jacques    tty5c        Jul 27 15:07
             ...
```

De forma análoga à saída padrão, podemos redirecionar a entrada padrão. Vejamos um exemplo de um comando que usa a entrada padrão:

```
$ cat
—
```

No comando acima, o cursor fica esperando que os dados sejam digitados no terminal, já que não houve redirecionamento da entrada padrão. Isto significa que o comando *cat* utiliza a entrada padrão se não for passado nenhum argumento. Um exemplo de redirecionamento da entrada padrão do comando *cat* é o seguinte:

$ cat < arq

No exemplo acima o comando *cat* lê dados de sua entrada padrão que foi redirecionada pelo shell para o arquivo *arq*. As informações são lidas da entrada padrão e o relatório de saída que é impresso pelo *cat* é o seguinte:

```
alvaro      tty01       Jul 27 14:53
jacques     tty5c       Jul 27 15:07
              ...
```

5.4 Saída padrão de erro

Por quê existe uma saída padrão de erro?

Para separar mensagens normais das mensagens de erro. Talvez esta resposta não tenha lhe sensibilizado o bastante para você sentir a importância da saída padrão de erro. Vamos imaginar uma situação onde não haja esta separação. Neste ambiente as mensagens de erro são misturadas com a saída normal. A ocorrência de erro é verificada visualmente nos relatórios impressos. Estes relatórios podem ser enormes (centenas ou milhares de linhas). Seguindo nesta linha vamos fazer muito trabalho repetitivo - com o lápis vermelho virtual - apenas para selecionar as mensagens de erro das mensagens normais.

Voltando à nossa realidade no UNIX, temos que usar corretamente as saídas padrão para não termos mensagens de erro misturadas com mensagens normais. Vejamos na prática exemplo de um comando que gera um texto como relatório na saída padrão normal:

$ comando

```
     ...
este é um texto para demonstrar  a
importância da saída padrão de
ERRO NO UNIX
     ...
Erro no núcleo do UNIX
     ...
```

Será que a mensagem *ERRO NO UNIX* faz parte do texto ou é uma mensagem de erro?

Do jeito que foi colocado não dá para responder. A única forma de saber é redirecionando a saída padrão de *comando* para um

arquivo temporario *temp*:

```
$ comando > temp

     Erro no núcleo do UNIX
     $
```

Ao redirecionar a saída padrão de *comando*, verificamos que a mensagem *ERRO NO UNIX* não é mensagem de erro. Por outro lado, descobrimos que existia uma mensagem de erro (*Erro no núcleo do UNIX*) misturada com o texto da saída padrão normal.

A saída padrão de erro é muito útil também para depuração de programas. Nesta fase de desenvolvimento precisamos dar uma atenção especial às mensagens de erro.

Para separar as mensagens de erro das mensagens normais, os programas poderiam ser executados da seguinte forma:

```
$ programa 2> erros

          ...
     este é um texto para demonstrar  a
     importância da saída padrão de
     ERRO NO UNIX
...
```

Observe que a mensagem de erro *Erro no núcleo do UNIX* não apareceu na tela. O shell, ao interpretar a construção: *2> erros*, criou o arquivo de nome *erros* associando-o à saída padrão de erro do comando *comando*. A implicação disso é que a mensagem de erro *Erro no núcleo do UNIX* for impressa no arquivo de nome *erros*:

```
$ cat erros

     Erro no núcleo do UNIX
```

Note que a mudança de valores default da saída padrão e da saída padrão de erro é feita de forma ortogonal. Para redirecionar qualquer descritor é só usar o caractere >, já visto. Para redirecionar a saída padrão de erro, por exemplo, usamos uma construção nova do shell:

```
2> erros
```

que quer dizer: o descritor 2 (saída padrão de erros) é redirecionado para o arquivo *erros*. De forma análoga o comando:

```
$ prog > relatório
```

pode ser escrito usando uma notação mais extensa, para que o shell não perca sua ortogonalidade e consistência:

```
$ prog 1> relatório
```

Em geral, um descritor ***m*** pode ser redirecionado usando ***m>***, onde *m* é um número na tabela de descritores que cada processo possue.

Vejamos um exemplo de um comando que redireciona a saída padrão e a saída padrão de erro para o mesmo arquivo *arq*:

$ comando > arq 2> arq

Ou, usando a outra forma mais extensa, podemos re-escrever o mesmo comando acima da seguinte forma:

$ comando 1> arq 2> arq

Ou ainda, usando uma nova forma sintática mais reduzida do shell, podemos re-escrever o comando acima:

$ comando > arq 2>&1

No exemplo acima a construção *2>&1* quer dizer: o descritor padrão de erro (*2*) deve ser associado (*>&*) ao mesmo arquivo que está ligado ao descritor da saída padrão (*1*). Em geral, podemos usar *m>&n*. Onde *m n* são indices na tabela de descritores que representam os arquivos padrão. O operador do shell *>&* faz com que o descritor *m* fique associado ao mesmo arquivo que está ligado ao descritor *n*.

5.5 Desprezando a saída padrão e a saída padrão de erro

As vezes, durante os testes de um determinado programa, queremos desprezar tanto a saída normal quanto a saída padrão de erro. Neste caso, usamos a seguinte construção do shell:

$ prog 2> /dev/null

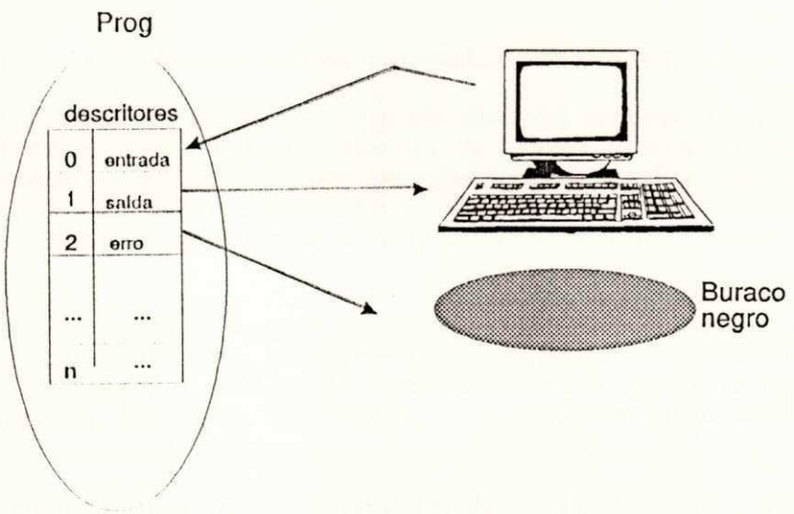


Figura 5.3: O buraco negro do UNIX

Note a diferença entre as figuras 5.5 e 5.6: a seta que representa o descritor *2* não é igual nos dois casos. A seguir, apresentaremos, através de comandos na prática, a ocorrência do fato relatado acima. Vamos escolher o comando *ls* do UNIX e executá-lo com parâmetro inexistente para termos mensagens de erro sendo gerada:

```
$ ls /ufal

     /ufal not found
```

a mensagem de erro */ufal not found* é impressa na saída padrão de erro, como era de se esperar já que o diretório */ufal* não existe. O passo seguinte, para demostrarmos a importância da ordem dos operadores de redirecionamento (>) e (>&), é executarmos o comando *ls /ufal* com os operadores de redirecionamento em posições diferentes. Vamos observar o conteúdo do arquivo *arquivo* para o comando *ls* executado da seguinte forma:

```
$ ls /ufal > arquivo 2>&1
```

O comando acima não imprime nenhuma mensagem na tela, porque tanto a saída padrão normal quanto a saída padrão de erro foram redirecionadas para *arquivo*. Se quisermos ver o conteúdo de *arquivo*:

```
$ cat arquivo

     ...
     /ufal not found
```

O arquivo *arquivo* contém as mensagens de erro impressa pelo comando *ls*.

Agora, se invertermos a ordem dos operadores de redirecionamento do comando *ls /ufal > arquivo 2>&1*, teremos um resultado diferente:

```
$ ls /ufal   2>&1  > arquivo

     /ufal not found
```

O comando acima, apesar de ter redirecionado a saída padrão de erro (*2>&1*), imprime a mensagem de erro */ufal not found* na tela. Este fato comprova que a saída padrão de erro ficou associada ao terminal.

5.6 O operador >>

A forma de redirecionamento vista até agora destruia o conteúdo anterior do arquivo. A seguir vamos apresentar uma forma alternativa para fazer o redirecionamento da saída padrão.

O novo formato de redirecionamento usa >> e tem a mesma sintaxe que o redirecionamento >> visto até agora. A única diferença é que o operador >> faz com que os dados sejam acrescentados no final do arquivo. Por exemplo, para criar um arquivo de *histórico* contendo as datas em que determinadas tarefas são executadas, podemos usar o comando *date* a seguir:

```
$ date >> historico
```

O comando *date* mostra a data corrente. Podemos ver o conteúdo do arquivo *historico* na tela:

```
$ cat historico

    Fri Feb  7 18:02:11 GMT 1992
```

Se, depois de esperar alguns minutos, executarmos novamente o comando *date* redirecionando sua saída para o arquivo *historico*:

```
$ date >> historico
```

Ao olharmos o conteúdo do arquivo *historico*, veremos que o seu conteúdo anterior não foi destruído, indicando que uma tarefa foi executada às *18* horas e *2* minutos e a outra tarefa foi processada às *18* horas e *20* minutos:

```
$ cat historico

    Fri Feb  7 18:02:11 ...
    Fri Feb  7 18:20:07 ...
```

O operador >> é ideal para se manter um histórico ou arquivo de "log" (ocorrências) de alguma atividade (backup, etc).

5.7 Como é feito o redirecionamento?

Chegou a hora de falarmos um pouco de como as coisas acontecem internamente no shell. Discutiremos a seguir o mecanismo de execução de processos e o uso de algumas funções básicas do núcleo do UNIX. Veremos também, como estas funções são usadas pelo shell para implementar redirecionamento. A título de ilustração, representaremos graficamente as atividades do shell, passo-a-passo, para fazer o redirecionamento de um simples comando.

O comando escolhido foi o comando *who* velho conhecido nosso. A simples execução do comando *who* faz com que o seguinte relatório apareça na sua tela:

```
$ who

    alvaro  tty ...
      ...
```

Nosso objetivo é ver o que o shell faz para executar o

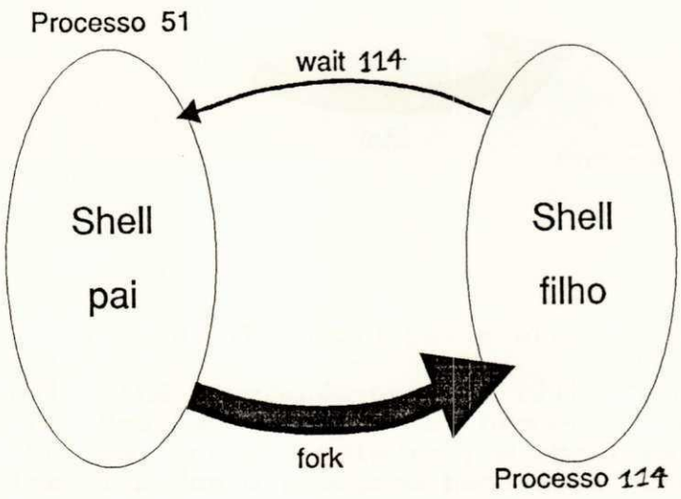
comando *who > arq*. Entretanto, antes, precisamos tecer alguns comentários sobre processos.

Tudo no UNIX é feito através de processos. O próprio interpretador de comandos de seu terminal é um processo. Quer ver? Digite o comando ***ps*** e teremos listados os processos de sua sessão:

```
$ ps

   PID TTY  TIME COMMAND
    51  01  0:02 sh
   114  01  0:00 ps
```

A saída impressa acima indica que temos dois processos rodando nesta sessão: o processo *51* (Process IDentification), que é o *sh* e o processo *114* que é o próprio *ps*. No instante em que digitamos comandos para o shell, estamos sob o controle do processo *51*. É importante observar que o shell cria um novo processo para executar cada comando. No caso acima, o processo *114* foi criado pelo shell e é chamado de **processo filho** do shell, ou seja, o shell é o **processo pai** do *ps*. O número de cada processo é fornecido pelo próprio núcleo do UNIX. É um número sequencial, crescente e único que identifica cada processo. Por outro lado, é o shell quem coordena a execução de comandos. O shell, na realidade, usa um serviço do núcleo do UNIX, a chamada ao sistema ***fork***, para poder criar o filho fisicamente na memória. Em determinado momento temos a seguinte figura:



(Figura 5.7: Criação de processos)

Na figura acima o processo *114* é criado através da chamada ao sistema *fork*. Através de outro serviço do UNIX este processo é substituido pelo programa *who*. Enquanto o processo *114* (*who*) está rodando, o shell fica aguardando. Para que hajá sincronismo entre os processos *51* e *114*, o shell utiliza outro serviço do núcleo

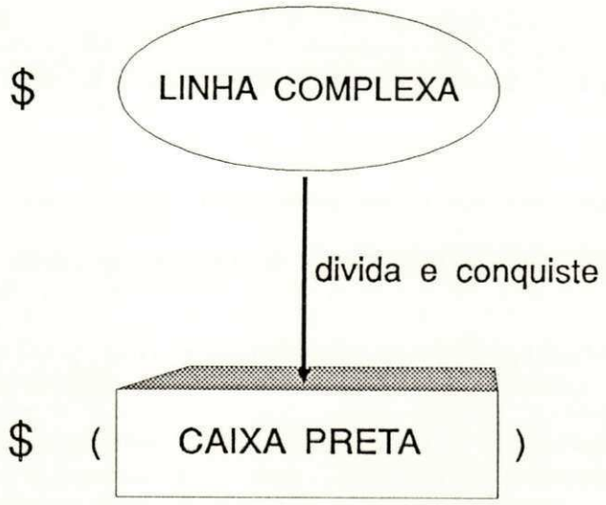


Figura 8.3: Divisão da complexidade de um comando

O significado do sub-shell é bastante intuitivo, devido a escolha dos operadores *(e)* para sua representação *(...)*. Sub-shell usa a estratégia de **divisão e conquista**, onde um problema complexo é dividido em outros menores. Por exemplo, o comando abaixo:

$ (who ; date) > arq

pode ser visto da seguinte forma:

1. *(...) > arq*
2. execução de comandos " ... "

onde em (1) há preocupação apenas com o que está fora dos parênteses, ou seja, com o redirecionamento >. Em (2) temos que executar a **caixa preta** " ... ", como se estes comandos dentro da caixa preta fossem digitados a partir do teclado.

Sub-shell é uma construção interna do shell onde os comandos podem ser agrupados usando *(...)*. Um shell (1.) trata os operadores que podem ser aplicados externamente ao *(...)*, como redirecionamento (<,>,<<,>>,|, etc) ou retaguarda (&). Um outro shell (o sub-shell) é ativado para executar os comandos da caixa preta "...", como se estes comandos, dentro da caixa preta, fossem digitados a partir do teclado.

Como será que o shell implementa internamente sub-shells? Lembre-se que entre os parênteses pode vir qualquer comando e que toda a linha pode ser redirecionada ou posta em retaguarda. Desenhe a figura de execução para os processo do comando abaixo. Ao tentar desenhar a figura, você vai verificar as dificuldades de o shell implementar internamente, o mecanismo para resolver o

comando abaixo:

$ (who; ps) &

Realmente não é fácil se você não usar a estratégia divida e conquiste.

8.4.1. Dificuldades para implementação do sub-shell

* Dentro dos parenteses pode vir qualquer combinação de comandos

* No exemplo *(who ; date) &* a dificuldade é controlar o sequenciamento de tarefas em retaguarda.

* No exemplo *(who ; date) > arq* a dificuldade é controlar o redirecionamente sem que haja sobreposição das saídas dos comandos *who* e *date* no arquivo *arq*.

8.4.2. Representação gráfica

Para entender como o shell implementa internamente Sub-shells, vejamos a representação gráfica da execução do comando abaixo:

$ (who ; date) > arq

FASE I:

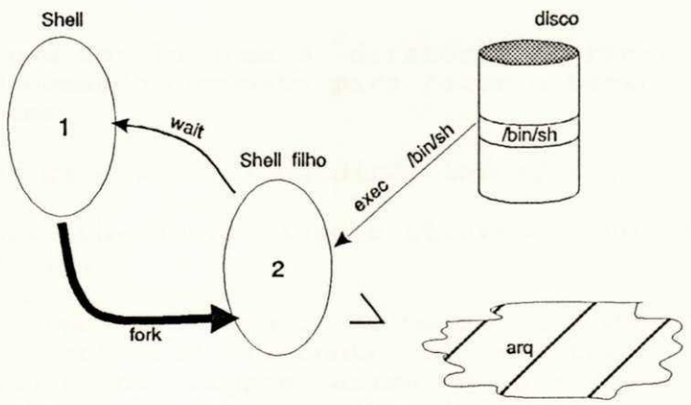


Figura 8.4: Redirecionamento da *caixa preta*

A figura acima foi dividida em duas fases. A primeira fase é executada pelo shell corrente que interpreta o redirecionamento de toda a caixa preta, a segunda fase é interpretada pelo sub-shell que executa o que estiver dentro da caixa preta.

Na primeira fase, o shell corrente (1) cria um filho, o

processo 2, e espera que ele morra, através das chamadas ao núcleo do UNIX *fork* e *wait*. O shell filho (2) reconhece que a linha de comandos *(...) > arq* se trata de um sub-shell com redirecionamento devido aos parenteses e ao caractere (>). O shell filho (2) redireciona a sua saída padrão para o arquivo *arq*, através das rotinas do UNIX *open*, *dup* e *close* e em seguida ativa o sub-shell. A ativação do sub-shell consiste na execução do programa */bin/sh*, passando os argumentos *who ; date* como parâmetros. Este mecanismo, na realidade, ativa um shell novinho para interpretar os comandos *who ; date*, como se eles tivessem sidos digitados a partir do teclado para um shell interativo.

A segunda fase é executada pelo sub-shell, veja na Figura 8.5. Ele já tem sua saída padrão redirecionada para o arquivo *arq*. O detalhe importante nesta segunda fase é que a saída do segundo comando, o comando *date*, não sobrepôs a saída do primeiro comando, o *who*. Por quê?

A resposta é que *who* e *date* imprimiram seus relatórios nas suas saídas padrões, que estavam redirecionadas para o arquivo *arq*. O redirecionamento foi feito pelo processo 2 da primeira fase, antes de ativar o sub-shell. Os comandos *who* e *date* herdaram a saída padrão do sub-shell por terem sido executados a partir dele.

O conceito de sub-shell pode ajudar a construir comandos bastantes poderosos. Por exemplo, conhecendo o comando *tar*, usado para fazer backup de arquivos, podemos copiar o diretório *dir1* para o diretório *dir2*, com o seguinte comando:

$ pwd

/usr/alvaro

O comando *pwd* nos informa o diretório corrente, no caso, */usr/alvaro*. O comando composto para fazer o backup de *dir1* para *dir2* é o seguinte:

$ (cd dir1; tar cf - .) | (cd dir2; tar xf -)

onde temos dois sub-shells ligados através de um duto, conforme mostra a Figura 8.5.

O comando *pwd* foi usado acima só para mostrar que vamos fazer backup de diretórios diferentes do diretório corrente. O detalhe importante na figura acima é que podemos mudar o diretório corrente do sub-shell-1 (cd dir1) e do sub-shell-2 (cd dir2), sem alterar o diretório corrente do shell pai (/usr/alvaro).

O comando *(cd dir1; tar cf - .) | (cd dir2; tar xf -)* faz com que o shell corrente crie dois sub-shells ligados através de um duto *(...) | (...)*. Os comandos do primeiro sub-shell não interferem no ambiente do shell corrente nem no outro sub-shell.

Antes da execução do comando *tar* há uma mudança de diretório corrente nos dois sub-shells.

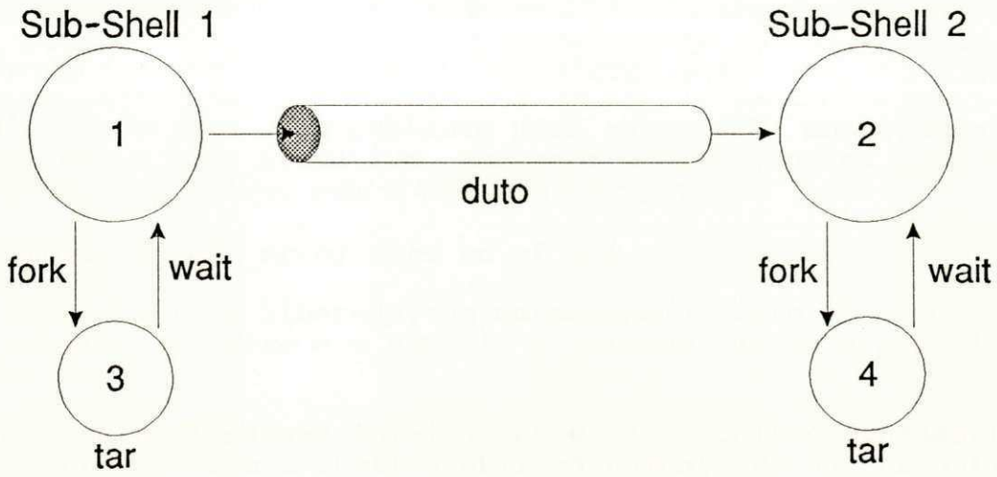


Figura 8.5: Ligação dos sub-shells

O comando ***tar cf - .*** formata e imprime na sua saída padrão todos os dados relativos aos arquivos e sub-diretórios a partir do diretório corrente ".", que é *dir1*. Já o comando *tar xf -* faz exatamente o contrário. Ele transforma os dados do formato *tar*, lidos da entrada padrão, para arquivos e sub-diretórios do UNIX, a partir do diretório corrente que é *dir2*. Como o sub-shell-1 está no diretório *dir1* e o sub-shell-2 está no diretório *dir2*, o diretório *dir1* foi integralmente copiado para o diretório *dir2*.

Podemos usar sub-shells para aumentar nossa produtividade. Por exemplo, o comando:

1. *cc prog.c*

faz com que o usuário fique esperando que a compilação termine para poder proseguir seu trabalho. Se o usuário conhecer um pouco o shell ele poderá compilar em retaguarda através do seguinte comando:

2. *cc prog.c &*

com o comando acima o terminal fica livre, mas se houver mensagens de erro o usuário pode perdê-las. Este problema seria facilmente resolvido com o comando:

3. *cc prog.c 2> erro &*

com o comando acima, o terminal fica livre e temos a garantia que as mensagens de erro não serão perdidas. O problema agora é saber quando a compilação terminou:

$ ps

...

$ ps

...

Ficar digitando *ps* a cada instante para saber se a compilação já terminou não é nada produtivo. Queremos produtividade. Sub-shell vai resolver o problema com o comando seguinte:

$ (cc prog.c 2> arq_erro; echo cc ok) &

onde o terminal fica liberado, as mensagens de erro são armazenadas no arquivo *arq_erro* e o usuário é avisado quando a compilação terminar.

Voltamos a discussão inicial sobre os conceitos fundamentais do UNIX. Conhecer bem o shell significa aumento na sua produtividade.

8.5. Outra forma de agrupar comandos

A seguir vamos mostrar uma forma de agrupar comandos onde estes serão executados pelo próprio shell corrente.

A forma como o UNIX implementa a execução de comandos, onde cada processo é executado pelo shell filho, ajuda a resolver muitos problemas. Às vezes precisamos que o próprio shell corrente execute comandos agrupados. Por exemplo, necessitamos mudar o diretório corrente do shell ou qualquer outro atributo de seu ambiente, como veremos mais adiante.

A forma de agrupar comandos sem criar um sub-shell é através da construção interna do shell "*{ ... }*". Os comando " ... " são agrupados da mesma forma que em "*(...)*", mas são executados pelo shell corrente ao invés do sub-shell.

A expressão entre "*{ ... }*" será tratada por um sub-shell "*(...)*" se houver redirecionamento da entrada ou da saída padrão. Por exemplo, o comando:

$ { ls ; who } > arq

é por definição a mesma coisa que:

$ (ls ; who) > arq

Você poderia perguntar: Qual a utilidade de "*{ ... }*" se já temos "*(...)*"?

A resposta já foi citada acima. Às vezes precisamos manipu-

lar o ambiente do shell corrente e com "(...)" não seria possivel. Esta diferença deve ficar mais clara quando estivermos estudando as funções do shell nos capitulos seguintes.

PARTE II: QUESTIONANDO CONCEITOS FAMILIARES

9. ARQUIVOS E SUAS ESTRUTURAS

A seguir vamos fazer um comparativo entre o UNIX e outro sistema operacional com relação ao tratamento de arquivos. Vamos discutir vantagens e desvantagens do jeito UNIX de manipular arquivos e dispositivos.

Alguns sistemas operacionais antigos implementam vários tipos de arquivos. Era responsabilidade destes sistemas suportar os vários tipos de arquivos diferentes:

* sequencial
* direto
* indexado
* sequencial-indexado

...

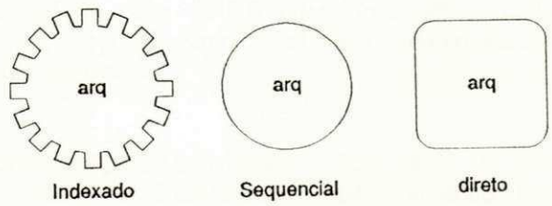


Figura 9.1: Tipos de arquivos

A Figura 9.1 nos mostra que os arquivos diferentes são tratados de forma diferente por estes sistemas operacionais mais antigos.

Já no sistema operacional UNIX todos os arquivos são tratados de uma mesma forma. Para o UNIX arquivos são sequências de bytes:

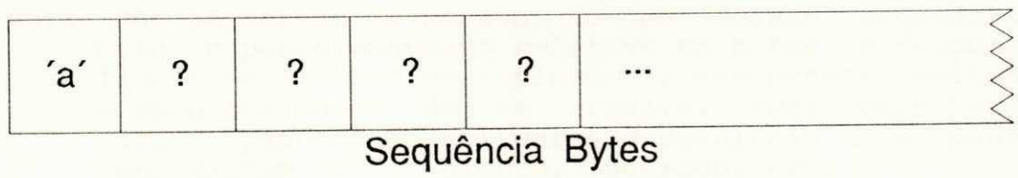


Figura 9.2: Estrutura de um arquivo no UNIX

O MSDOS é um exemplo de sistema operacional que trata os arquivos de maneira parecida com o UNIX (do ponto de vista do usuário).

O comparativo abaixo é fornecido para ajudar novos usuários do UNIX. Estamos supondo que estes usuários conhecem outro sistema operacional. Supomos ainda que o sistema antigo tratava arquivos diferentes de forma diferente.

9.2. Comparativo entre sistemas operacionais

Vamos fazer um comparativo entre o **UNIX** e **outro sistema operacional**, para ver as principais diferenças. O objetivo aqui é levantar os principais pontos onde há divergência.

1. Outro sistema operacional: Arquivos de formatos diferentes para armazenar informações diferentes. Aqui o próprio sistema operacional armazena informações diferentes de forma diferente. Por exemplo, arquivos sequenciais não são tratados pelos sistema da mesma forma que os arquivos indexados.

   UNIX: Arquivos são iguais e podem conter qualquer tipo de informação. No UNIX que tem que interpretar a informação é o próprio aplicativo. Arquivos são sempre sequências de bytes.

2. Outro sistema operacional: Existe o conceito de registro, campo, chave, tamanho. Isto é uma consequência diretada do tratamento diferenciado aos arquivos.

   UNIX: Arquivos são sequência de bytes. O acesso é estabelecido pela própria aplicação. O aplicativo pode criar o conceito de registro, campo, etc.

3. Outro sistema operacional: Existe um método de acesso diferente para cada tipo de arquivo. Podemos recuperar um registro de várias formas, dependento do tipo de arquivo sendo pesquisado: acesso direto, sequencial, relativo, indexado sequencial.

   UNIX: Não há método de acesso. Só tem acesso sequencial a byte e posicionamento relativo em bytes. É responsabilidade do aplicativo implementar seu próprio sistema de acesso aos dados de um arquivo. Felismente existem vários pacotes de softwares disponiveis no mercado que implementam acesso direto, indexado, etc.

4. Outro sistema operacional: Os sistemas mais antigos não permitiam o compartilhamento dos arquivos.

   UNIX: O Compartilhamento total. Muito permissivo. Não tem lock (travamento) por default. Historicamente, UNIX é muito permissivo.

9.3. Vantagens e desvantagens do jeito UNIX

A seguir vamos discutir as principais vantagens e desvantagens do jeito UNIX de fazer a manipulação de arquivos.

A vantagem do UNIX tratar arquivos sempre como sequência de bytes é: simplicidade. Se torna mais simples para o projeto do sistema operacional; é mais maleável para o programador que pode fazer um mapeamento do que o aplicativo quer para o que o UNIX tem; é um denominador comum de qualquer modelo de arquivo; os arquivos podem conter qualquer informação; não precisa informar tamanho na criação do arquivo na sua criaçao. O crescimento de arquivo é dinâmico; permite compartilhamento total da informação entre os vários usuários.

A desvantagem é que o programador não tem ferramentas nativas no UNIX para aplicações comerciais. Ele precisa comprar bibliotecas para simular acesso indexado, etc. Estas bibliotecas não tem padronização e o seu desempenho as vezes deixa a desejar; o núcleo do UNIX precisa de mais esforço computacional para administrar o crescimento dinâmico de arquivos; o compartilhamento total das informações pode acarretar corrupção da informação.

9.4. Diretórios

Nesta seção vamos apresentar alguns detalhes internos de implementação do sistema de arquivos e diretórios do UNIX.

Já falamos nos capítulos anteriores que diretório é uma ferramenta organizacional para o homem poder classificar suas informações no UNIX.

A forma como o UNIX implementa diretório não é comum em outros sistemas operacionais. Diretórios no UNIX não são locais escondidos. Eles podem ser acessados como arquivos. A Figura 9.3 nos mostra que diretórios no UNIX são arquivos onde seu formato é interpretado como entradas de dezesseis bytes. Os dois primeiros bytes armazenam o número interno para controle do UNIX, e os quatorze bytes restantes guardam o nome simbólico do arquivo. O nome simbólico é aquele que aparece como saída do comando *ls*.

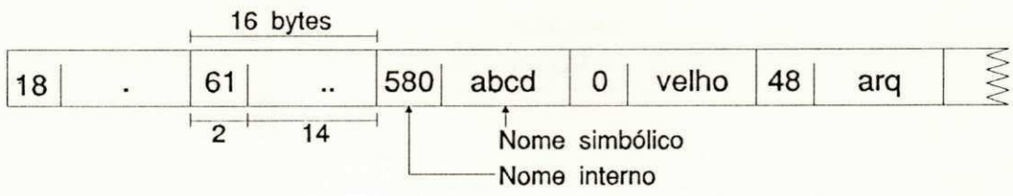


Figura 9.3: Layout de um diretório

Apesar de o usuário poder ler o arquivo que contém o diretório, só quem pode escrever no diretório é o núcleo do UNIX. Por exemplo, se o usuário quiser ler o diretório */bin* é só usar o

seguinte comando:

```
$ od -c /bin

000  \0   .  \0  \0  \0  \0 ...
016  \0   .   .  \0  \0  \0 ...
032  \0   p   s  \0  \0  \0 ...
048  \0   l   s  \0  \0  \0 ...
          ...
```

A saída do comando *od*, que imprime o conteúdo do diretório */bin*, deve ser interpretada como caracteres (devido a opção *-c*). Os dois primeiros bytes, em negrito, representam o nome interno do sistema UNIX (**inode**). Os quatorze bytes restantes significam o nome visível para o usuário. Na saída do comando *od* acima, temos os seguintes nomes simbólicos:

```
.
..
ps
ls
```

Através do nome simbólico, o UNIX encontra informações adicionais armazenadas no **inode** tais como: Os dois primeiros nomes simbólicos "." e ".." servem para implememtar o Sistema de Arquivos em Arvore do UNIX. Os nomes de arquivos *ps* e *ls* representam os comandos para mostrar o status dos processos que estão rodando e para listar os arquivos do diretório, respectivamente. A seguir mostramos uma figura da estrutura de como o UNIX acessa o conteúdo de *arquivo* a partir de seu nome simbólico.

A Figura 9.4 nos dá uma visão geral do que é feito para se ver o conteúdo de um arquivo através do comando *cat*. Quando digitamos *cat arquivo* o diretório corrente é pesquisado a procura do nome simbólico *arquivo*. O que interessa para o UNIX é o número *125*, batizado de inode, associado com o nome simbólico *arquivo*. No inode *125* é onde se encontra as informações para a manipulação efetiva do arquivo tais como *dono* e *grupo*:

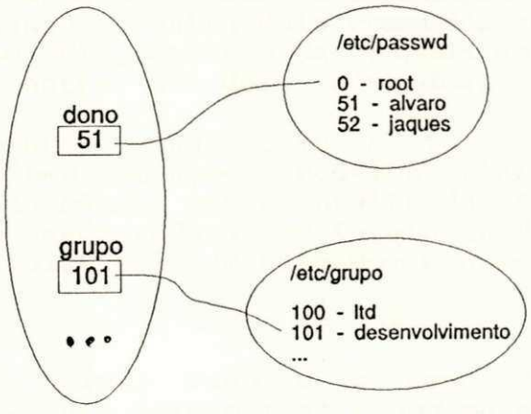


Figura 9.4: Informações de um inode

Além das informações *dono* e *grupo*, que representam, respectivamente, o dono e o grupo a que pertence o arquivo, existem outras informações dentro do *inode*:

permissões - Usado para verificar se um usuário, ao executar um comando qualquer, pode ter acesso ao arquivo. Voltaremos a falar deste atributo mais adiante.

dono - Contém o código que representa o dono do arquivo no sistema. Esta informação é usada para checar as permissões.

grupo - Usuários são agrupados no UNIX. Grupos podem ser criados por departamento, por projeto ou qualquer outro critério. Este campo contém o código que representa o grupo no sistema.

tamanho - É o tamanho do arquivo em bytes.

blocos dados - Indica quais são os blocos onde se encontram os dados do arquivo *arquivo*.

data acesso - Ultima data que o arquivo foi acessado.

data modificação - Ultima data que o arquivo foi modificado.

data criação - Data que o arquivo foi criado.

9.5. Arquivos FIFO

Aqui queremos mostrar dutos com nomes e como fazer comunição entre processos em terminais diferentes.

Um dos conceitos responsáveis pela alta taxa de produtividade no UNIX é o duto ou pipe.

Os pipes existem para fazer a comunição entre dois processos. O shell quando encontra dois comandos conectados através do símbolo | pede ao UNIX, através da chamada ao sistema **pipe**, para criar um duto entre os dois comandos.

Vamos relembrar ainda como é feita a criação de dutos permanentes ou *pipes* com nomes. Podemos criar um duto permanente, através do comando *mknod*, com o intuito de fazer a comunicação entre dois processos rodando em terminais diferentes. Por exemplo, o comando abaixo cria um duto com o nome cano:

$ mknod cano p

onde, no comando acima, cano é o nome do arquivo de pipe permanente e *p* é para informar ao *mknod* para criar um arquivo de pipe. O comando *mknod* pode ser usado para criar outros tipos de

arquivos especiais como nome de terminais, discos, impressoras, etc.

Um exemplo envolvendo o duto com nome cano é a comunicação entre dois comandos rodando em terminais diferentes, por exemplo:

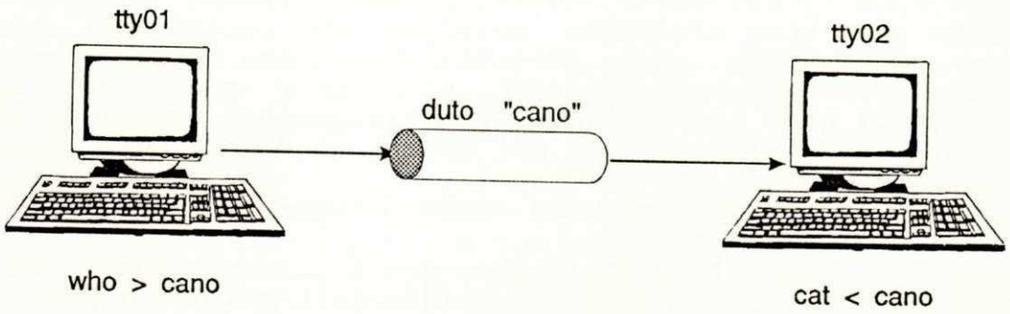


Figura 9.5: Comunicação usando pipe com nome

Uma das diferenças entre arquivos normais e dutos com nome ou arquivos fifo é que a **leitura** dos dados de um duto é **destrutiva**, ou seja, a leitura remove os dados do duto. A consequência deste fato é que o tamanho do duto não precisa variar, ele é sempre constante, independente da quantidade de dados que trafegar no cano.

Normalmente o duto com nome é usado como canal de comunicação unidirecional entre dois processos concorrentes: um consumidor e o outro produtor de informações.

O núcleo providencia o sincronismo de leitura e gravação entre consumidor e produtor. Se um processo for ler de duto vazio ou for gravar em duto cheio, o núcleo do UNIX coloca este processo para dormir (processo fica esperando que o duto seja liberado).

O duto é implementado internamente como arquivo. Por questões de eficiência, normalmente, o tamanho do duto esta limitado a dez blocos, o que em algumas máquina representa 10 kilo-bytes.

9.6. Arquivos especiais

Aqui queremos mostrar que a manipulação de dispositivos externos é simples como manipular arquivos, devido a existência do conceito de arquivos especiais.

Os recursos externos ou os dispositivos têm nomes de arquivos no UNIX. Para ler um disquete, basta digitar o comando:

$ cat < /dev/disquete

que os dados serão lidos do disquete e apresentados na tela. Obviamente que o nome */dev/disquete* varia de instalação para instalação.

O nome */dev/disquete* é na realidade um arquivo especial, que representa a unidade de disquete. O nome */dev* (de device) representa o diretório default onde devem ser criados todos os arquivos especiais. Os arquivos especiais podem ser usados em qualquer situação onde for possível usar um arquivo normal. Algumas operações normais a arquivos podem parecer estranhas quando aplicadas a dispositivos: *open* (abrir), *read* (ler), *write* (gravar), *close* (fechar), *lseek* (se posicionar), etc.

O que significa *open* e *close* para um dispositivo?

Em linhas gerais, ligar e desligar o dispositivo ou o que é feito para iniciar o dispositivo e o que é feito para encerrar o mesmo.

Uma coisa foge um pouco ao modelo de arquivo: não podemos travar (*lock*) pacialmente dispositivos nem rebobinar (*ioctl*) arquivos:

arquivo	dispositivo
lock	*ioctl* - rebobinar - formatar - velocidade

O resultado da junção dos modelos de arquivos e dispositivos padroniza a entrada e saída dos comandos, o que facilita nossa vida. Para cada dispositivo deve existir um arquivo especial: */dev/tty01, ..., /dev/hd0, ...,/dev/laser, ...*, etc.

9.7. Acionamento de drivers

A discussão a seguir consiste em mostrar que a execução de drivers ou programas específicos que tratam dispositivos é feita através de uma tabela usando dois números como índices: os números maiores e menores da tabela de dispositivos.

O nome */dev/disquete* tem dois números associados a ele: o número **maior** e o número **menor.** É através destes códigos que o UNIX ativa o dispositivo físico.

O nome simbólico */dev/disquete* serve para o UNIX fazer o mapeamento e acessar os dados no dispositivo fisicamente.

Mecanismo de acionamento de drivers é feito da seguinte forma: existe uma tabela de dispositivos. O número maior é usado para acionar o dispositivo, que normalmente é um programa que

reside em uma placa fornecida pelo seu fabricante. O dispositivo é acionado e recebe como parâmetro o número menor para tratar internamente ao dispositivo.

Geralmente o número menor é usado para identificar que unidade do dispositivo esta sendo usado. Por exemplo, o comando *ls -l /dev/tty** mostra o seguinte relatório de saida:

```
...      0,  0 Jul 31 11:35 /dev/tty01
...      0,  1 Jul 31 11:35 /dev/tty02
...      0,  2 Jul 31 08:09 /dev/tty03
          ...
```

onde o número maior é *0* (zero) e os números menores são *0, 1 e 2* para os terminais *tt00, tt01* e *tty02*, respectivamente. O dispositivo que controla os terminais é o primeiro na tabela de dispositivos (o número maior é o índice nesta tabela *0*). O número menor é usado pelo dispositivo para saber qual o terminal que deve ser tratado.

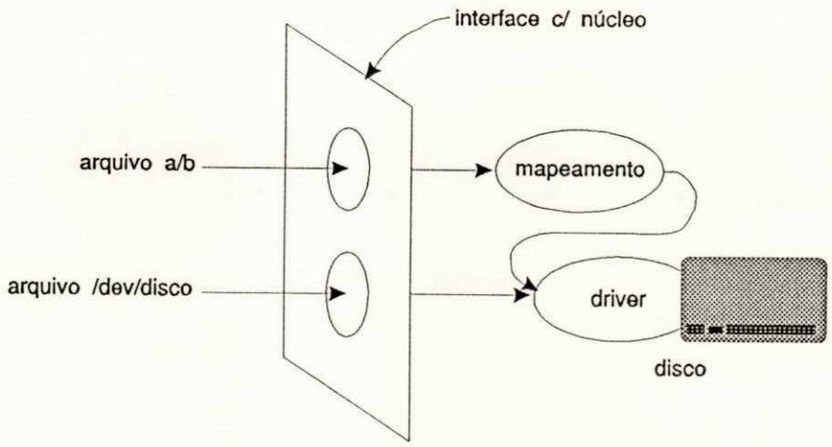


Figura 9.6: Mapeamento do nome simbólico

Em geral o mapeamento de um nome simbólico é usado para se encontrar o número menor e o número maior, para o UNIX poder acionar o driver associado ao dispositivo. A Figura 9.6 nos mostra que é feito um mapeamento, pelo núcleo do UNIX, do nome simbólico *a/b* para se localizar o conteúdo do arquivo *b* no diretório *a* que reside no dispositivo */dev/disco*. O mapeamento transforma o nome *a/b* em endereços de blocos que estão no dispositivo */dev/disco*.

A manipulação do dispositivo pode ser feita diretamente, através do arquivo especial */dev/disco*, sem usar o mapeamento. Neste caso, se tem acesso a todo o dispositivo.

9.8. Vantagens dos arquivos especiais

A sintaxe é igual para todos os comandos, independente de que dispositivos estejam sendo acessados. É ortogonal com outros conceitos, implicando em flexibilidade, uniformidade e, consequentemente, aumentando sua **produtividade**. Como resultado temos o sistema de proteção idênticos para arquivos normais e dispositivos.

9.9. Tipos de arquivos especiais

Existem dois tipos de arquivos: arquivos especiais a bloco e arquivos especiais a caractere.

Em dispositivos que operam a **bloco**, tais como: discos, fitas, etc, as operações são realizadas através da transferência de um bloco a cada vez:

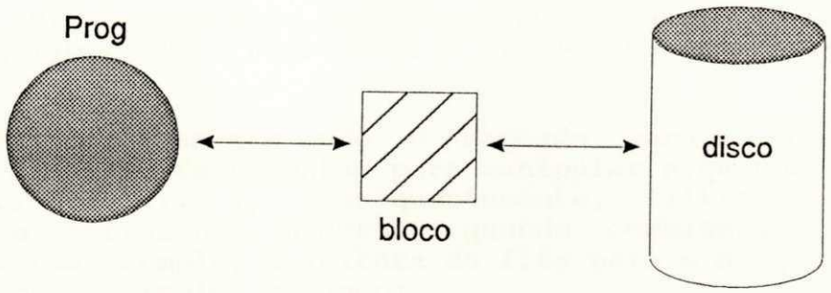


Figura 9.7: Dispositivos que operam a bloco

Dispositivos que operam a caractere, tais como terminal, impressora paralela, etc, as operações são realizadas através da transferência de **linhas** ou de **caracteres**, dependendo da disciplina de linha de comunicação entre o processo e o dispositivo.

Um exemplo concreto de arquivo especial a caractere são os terminais. Estes arquivos */dev/tty01*, */dev/tty02*, */dev/tty03*, ... representam os terminais *01*, *02* , *03*, ..., respectivamente. Existe um arquivo especial */dev/tty* que representa o terminal correntemente em uso. A Figura 9.8 ilustra um processo *prog* sendo executado e lendo caracteres do terminal. Observe que o terminal e o processo manipulam caracteres.

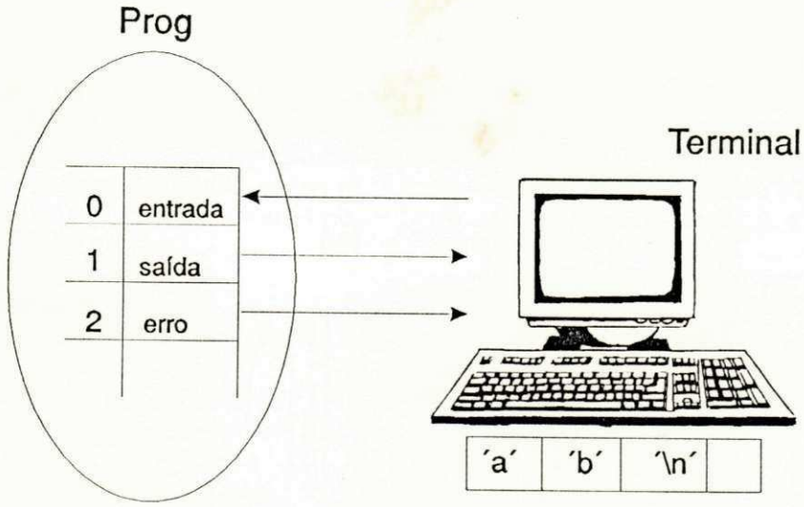


Figura 9.8: Dispositivos que operam a caractere

9.10. Importância dos arquivos no UNIX

A manipulação de arquivos no UNIX é muito importante. Saber usar corretamente arquivos no UNIX pode significar um aumento enorme de produtividade. Usando arquivos, desenvolver software básico no UNIX parece mais software comercial onde temos que abrir arquivos, ler registros e formatar relatórios, não é simples?

No UNIX praticamente tudo é mapeado para o conceito de arquivo. Existem ferramentas para manipular arquivos (*sh*, *grep*, *sort*, *find*, *cat* ...), consequentemente, filtros simples se tornam uma ferramenta poderosa quando combinados com outros programas. Por exemplo, a leitura da fita para o disquete poderia ser feita com o simples comando:

$ cat < /dev/fita > /dev/disquete

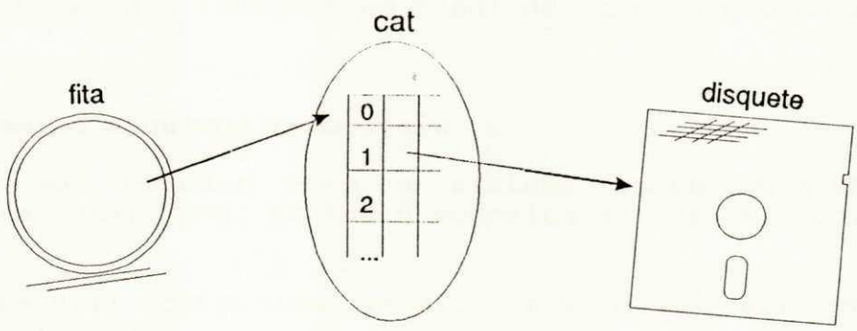


Figura 9.9: Manipulação de dispositivos

Outro exemplo de leitura do winchester para uma impressora

a laser seria feita com o comando seguinte:

```
$ cat < /dev/disco > /dev/laser
```

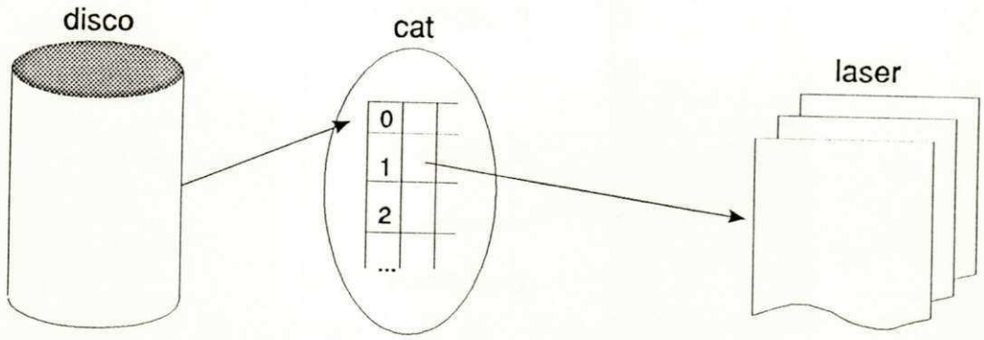


Figura 9.10: Manipulação de dispositivos

A ortogonalidade no UNIX permite que a maior parte do processamento do UNIX seja simples manipulação de arquivos. Vejamos alguns exemplos de arquivos que existem no UNIX: arquivos de dados; arquivos executáveis ou módulos de carga dos comandos do UNIX (*/bin/sh*, */bin/ls*, etc); alguns arquivos de controle usados pelo sistema operacional (/etc/passwd, /etc/group, etc); arquivos usados para implementar o conceito de diretório. O comando *ls*, na realidade, lê arquivos para poder mostrar os nomes dos arquivos; os dispositivos são arquivos. Os comandos *backup*, *fsck* e *mkfs*, manipulam arquivos de controle do sistema; arquivos especiais. O próprio núcleo do UNIX é representado por um arquivo especial. O comando *ps*, para saber quais os processos em execução, manipula simplesmente o arquivo */unix* e */dev/kmem*; arquivos de pipe para fornecer um canal de comunicação entre dois processos;

9.11. Sistema de arquivos removíveis

Aqui vamos estudar como o sistema operacional UNIX e o sistema operacional MSDOS tratam o conceito de disco lógico ou partição.

A Figura 9.11 mostra como um disco físico pode ser separado em partições lógicas.

Como será que os sistemas operacionais juntam os pedaços ou partições do disco?

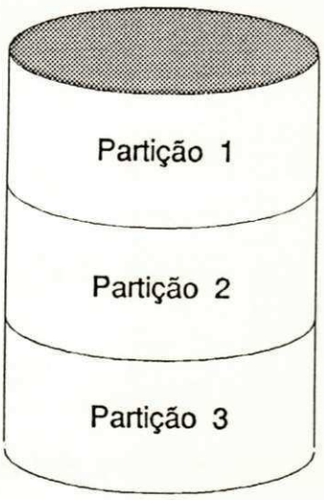


Figura 9.11: Disco particionado

O MSDOS usa nome de partição como componente do nome do arquivo. Por exemplo, o arquivo *d:teste.dat* está armazenado na partição *d:*. Já o arquivo *a:teste1.dat* está na partição removível *a:*:

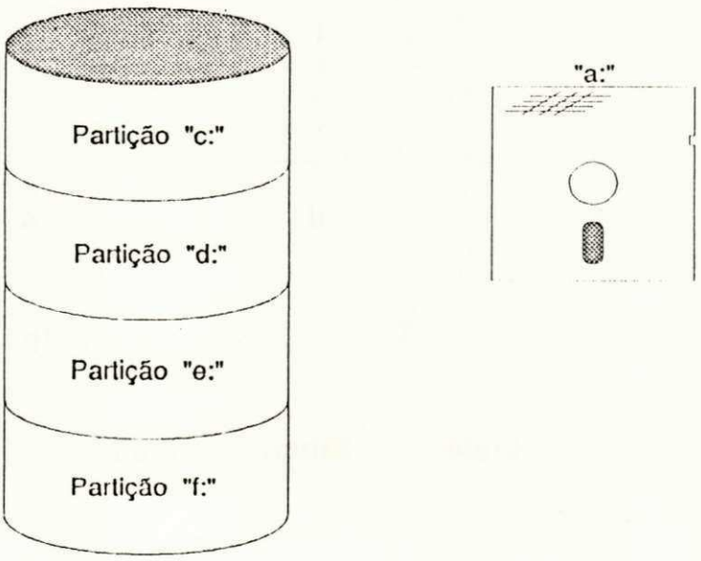


Figura 9.12: Particionamento no MSDOS

O UNIX junta as partições através dos comandos *mount* e *umount*. Vamos supor duas partições, uma no disco rígido e outra no disquete. Antes da montagem a situação é a seguinte:

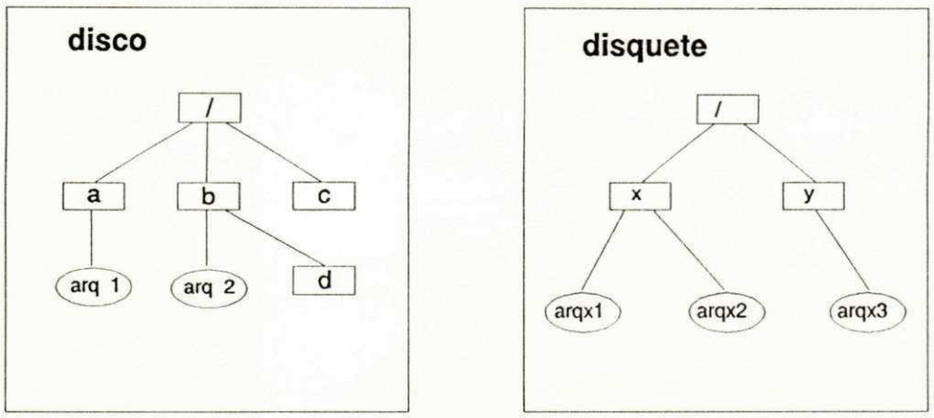


Figura 9.13: Particionamento no UNIX

na Figura 9.13 temos duas partições separadas com seus sistemas de arquivos independentes. Após a montagem do *disquete* no diretório */b* temos a seguinte figura:

 # mount /dev/disquete /b

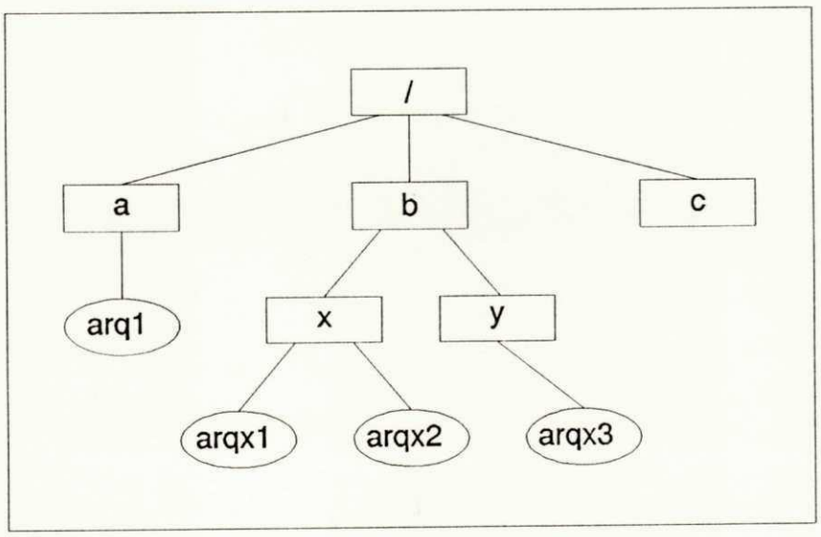


Figura 9.14: Partições UNIX após montagem

Note que durante a montagem do disquete, as informações (diretórios e arquivos) que estavam no diretório */b* ficam escondidas durante a montagem. O arquivo */b/arq2* e o subdiretorio */b/d*

continuam lá na partição do disco rigido, mas ficam invisiveis temporariamente durante a montagem do disquete.

PARTE II: QUESTIONANDO CONCEITOS FAMILIARES

10. PROTEÇÃO DE ARQUIVOS

Neste capitulo vamos apresentar o sistema de proteção de acesso aos dados no UNIX.

10.1. Dono, grupos e outros

Queremos discutir aqui que tanto usuários quanto programas são representados por uma identificação nos arquivos de configuração do sistema.

Todo arquivo no UNIX tem vários atributos: permissões, nome, grupo, tamanho e outras informações.

Existem dois arquivos de configuração do UNIX: o arquivo */etc/passwd* que mantém as identificações dos usuários e o arquivo */etc/group* que guarda as identificações dos grupos.

Cada usuário tem que ser cadastrado no sistema e pertencer, obrigatoriamente, a pelo menos um grupo.

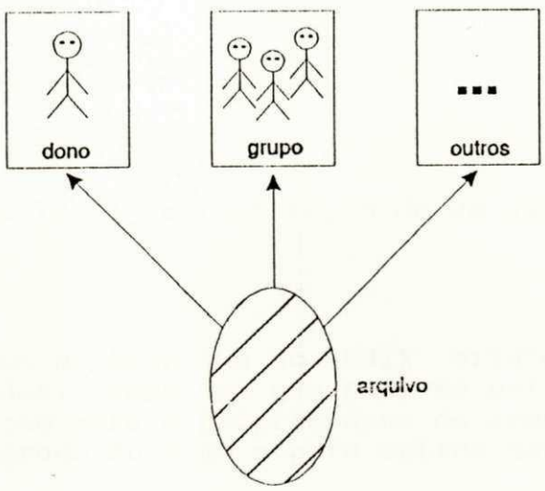


Figura 10.1: Conceito de dono, grupo e outros

Todos os arquivos do UNIX guardam informações sobre quem é o seu **dono**, qual o **grupo** do dono e informações sobre o que os **outros** usuários podem fazer com este arquivo.

Cada *programa* representa um *usuário* e um *grupo*. Por exemplo, quando um usuário entra no ar, ele recebe um interpretador de comandos representado na tela pelo caractere de prontidão, normalmente *$*.

Vejamos a representação gráfica, através da Figura 10.2, de um processo que representa você quando entra no ar. O interpretador de comandos */bin/sh*, que é o processo na memória, roda representando o usuário *51* (alvaro) e grupo *101* (desenvolvimento). Isto porque *alvaro* foi cadastrado no sistema com *id* (identificação) *51* e gid *101* (identificação do grupo).

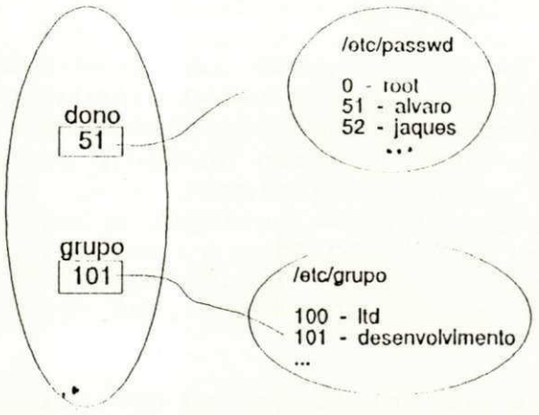


Figura 10.2: Representação de um usuário

10.2. Permissões

A seguir vamos mostrar que no UNIX existe um esquema de proteção aos dados. Vamos ver que pode-se estabelecer níveis de permissões de acesso para o próprio dono do arquivo, para pessoas que pertençam ao grupo do dono e para outros usuários quaisquer.

Quando um programa usa um arquivo, as informações de identificação do processo corrente e do arquivo são cruzadas para ver se o processo pode manipular o arquivo.

Existem três vias de acesso ao arquivo *arquivo*. Uma via para o próprio **dono** do arquivo, outra para pessoas que pertençam ao **grupo do dono** e uma terceira via para outras pessoas. É o núcleo do UNIX, através de suas rotinas ou chamadas ao sistema, que verifica se o processo tem ou não permissão para manipular o arquivo.

Através do comando *ls -l* listamos as permissões do arquivo *arquivo*:

```
$ ls -l arquivo

    -rw-r----x ... arquivo
     (1)(2)(3)
```

onde *r*, *w* e *x*, significam respectivamente, read, write e

Permissões especiais na execução (décimo bit de permissão ou **set user id**). Com o bit **suid** ligado no módulo executável, o programa *com* executa com as permissões do dono do arquivo. Veja figura abaixo:

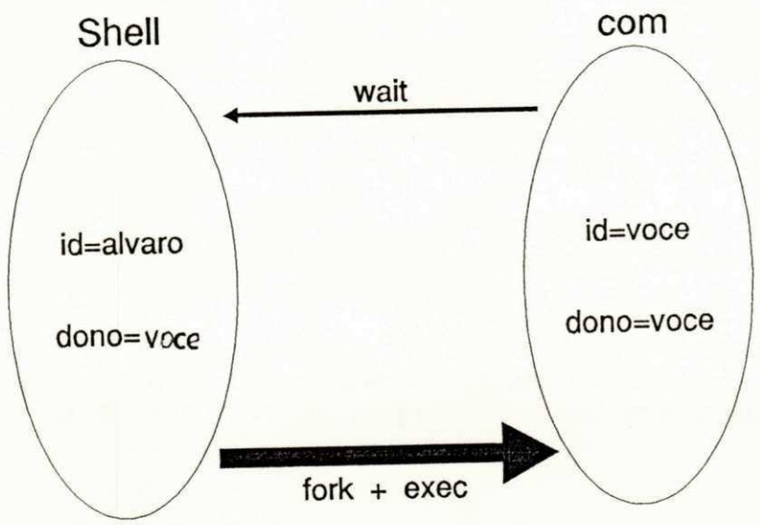


Figura 10.4: Permissões durante execução (bit s)

Como o shell não tem o bit s ligado, o processo executa com a permissão *id=alvaro* (informação lida do arquivo */etc/passwd*).

Ao executar o comando *com* o UNIX percebeu o bit s ligado e mudou as permissões da execução do comando *com* para:

id=você
dono=você

Se o comando *com* não tivesse o bit s ligado, as pemissões do comando *com* executado por *alvaro* seria:

id=alvaro
dono=você

Qual a necessidade de se ter comandos com o bit **s** ligado?

Acesso controlado a informações confidenciais, seria a resposta. Um exemplo da necessidade do bit s pode ser demonstrado através do comando *passwd*. O comando */bin/passwd* é executado por todos os usuário para mudar a senha de acesso ao sistema, apesar de o usuário não ter permissão de gravação no arquivo de senhas.

Exemplo de comando que acessam informações confidenciais por todos os usuários:

```
$ ls -l /bin/df /bin/passwd /bin/mkdir /bin/rmdir

-rws--x--x  1 sysinfo  bin  13756 May 13 1988 /bin/df
-rws--x--x  1 root     bin  18036 Dec  5 1988 /bin/passwd
-rws--x--x  1 root     bin  18036 Dec  5 1988 /bin/mkdir
-rws--x--x  1 root     bin  18036 Dec  5 1988 /bin/rmdir
```

Para ligar o bit s de um comando executável usamos o *chmod* da seguinte forma:

```
$ chmod u+s comando
```

onde ***u+s*** é uma forma sintática para o comando *chmod* entender que queremos ligar (+) o bit s do usuário (***u***).

De forma semelhante para a permissão ao "bit user id", existe o "bit group id", que é ligado para o processo ter as permissões do grupo do dono do arquivo (*chmod g+s comando*).

PARTE II: QUESTIONANDO CONCEITOS FAMILIARES

11. METACARACTERES DO SHELL

Neste capitulo vamos discutir alguns caracteres com significado especial para o shell chamados de metacaracteres.

11.1. Visão do usuário

A visão do usuário novato em um sistema operacional qualquer é monolítica. Para o iniciante quem faz tudo é o sistema operacional. No UNIX não podia ser diferente.

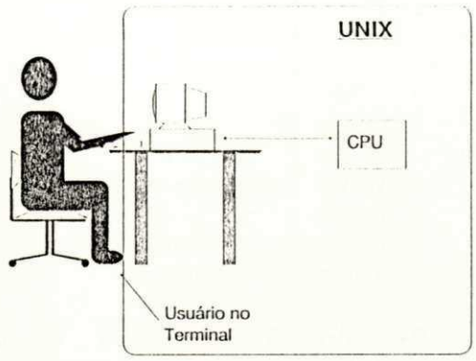


Figura 11.1: Visão do usuário UNIX

Ao contrário do que a maioria dos usuários iniciantes imaginam, o UNIX é modular. O UNIX pode ser dividido em serviços do núcleo e utilitários da seguinte forma:

Núcleo: abrir arquivo, fechar arquivo, ler disco, gravar disco, etc.

Utilitários: Usam os serviços do núcleo.

Um exemplo da modularidade do UNIX é o próprio interpretador de comandos. O shell é um utilitário como outro qualquer. O núcleo não o trata de maneira diferenciada em relação aos demais comandos do sistema, por ele interpretar comandos do usuário.

A finalidade do shell é interagir com o usuário. Ele aceita, na digitação de comandos, caracteres normais e caracteres com significado especial (metacaracteres).

Os metacaracteres são interpretados pelo shell para facilitar a geração de nomes, minimizar a digitação, diminuir a taxa de erro e, mais uma vez, aumentar sua produtividade. Veja a seguir os seguintes caracteres especiais *, ? e [ ] que o shell interpreta:

'*'	Casa com qualquer cadeia;
'?'	Casa com um caractere;
'[...]'	Casa com um caractere pertencente à classe ...

A seguir vamos ilustrar o uso dos metacaracteres *, ? e [ ] através de exemplos.

11.2. O metacaractere '?'

Para demonstrar o uso do metacaractere *?* vamos explicar como funciona a execução do comando ***$ echo a?***, assumindo que o diretório corrente tem os seguintes arquivos:

$ ls -C

TEMP	*a*	*aA*	*aB*	*aC*
c10	*c11*	*c12*	*c13*	*c2*
c3	*c4*	*c5*	*c6*	*c7*
c8	*c9*	*conteudo*	*f1-1*	*f10-1*
f11-1	*f11-2*	*f2-1*	*f2-2*	*f2-3*
f7-1	*f8-1*	*f8-2*	*f3-1*	*f3-2*
t6-1	*t8-1*	*tmp*	*Zebu8*	

O resultado para o comando *$ echo a?* é mostrado mais adiante indicado em três passos (1), (2) e (3): o que é digitado por você(1), o que é feito pelo shell (2), e pelo o que é feito pelo comando *echo*(3)

```
$ echo a?                              (1)

    + echo aA aB aC                    (2)
    aA aB aC                           (3)
```

Antes de explicar a expansão acima, vamos abrir um parenteses e falar do comando ***set***. O shell permite a visualização da expansão de metacaracteres através de uma opção interna:

```
$ set -x        (mostra a expansão dos metacaracteres)
$ set +x        (volta ao tratamento normal: não mostra expansão)
```

onde *set* é um comando interpretado internamente ao shell e *-x* informa ao shell para mostrar a linha de comando que foi expandida colocando o caractere + na frente para melhor visualização.

Agora vamos explicar o que é feito pelo núcleo do UNIX, pelo shell propriamente dito e pelo comando *echo*.

O shell é responsável pela expansão dos metacaracteres. A linha (1) acima é digitada pelo usuário. O shell interpreta a linha (1) e **monta** a linha (2) pesquisando no diretório corrente nomes de arquivos com duas letras começando por *a*. Dizemos que o shell **expande** a linha (1) para a linha (2). O metacaractere *?* não é visto pelo comando *echo*. O caractere + foi usado pelo shell para imprimir a linha de comando expandida. A linha *+echo aA aB aC* foi impressa pelo shell antes de executar o comando *echo* devido a opção *set -x*, usada para configurar o shell, estar ligada.

A expressão *a?* casa com todos os nomes de arquivos com dois

caracteres que começe com a letra *a*. É mais produtivo digitar *a?* do que digitar *aA aB aC*, você não concorda?

Depois de fazer a expansão dos metacaracteres o shell chama uma rotina do núcleo *exec* para executar o comando (2) acima. O resultado da execução do comando é mostrado na linha (3).

Exercicio 11-1

De acordo com os arquivos no diretório corrente:

$ ls -C

```
TEMP      a         aA        aB        aC
c10       c11       c12       c13       c2
c3        c4        c5        c6        c7
c8        c9        conteudo  f1-1      f10-1
f11-1     f11-2     f2-1      f2-2      f2-3
f7-1      f8-1      f8-2      f3-1      f3-2
t6-1      t8-1      tmp       Zebu8
```

Qual o resultado para os comandos abaixo levando em consideração a saida do comando *ls -C* acima?

1.) *$ echo c?*

2.) *$ echo ?*

3.) *$ echo f?*

4.) *$ echo c??*

11.3. O metacaractere *

O metacaractere * representa **todos**. A expressão *a** quer dizer todos os nomes de arquivos começando pela letra *a*.

O resultado para o comando *$ echo a**, tomando como base o mesmo diretório da listagem *ls -c* acima é o seguinte:

```
$ echo a*                          (1)

+ echo a aA aB aC                  (2)
a aA aB aC                         (3)
```

A expressão *a** faz com que o shell pesquise todos os nomes de arquivos começando por *a*. A expansão da linha (1) para a linha (2) mostrou que *a** foi expandida para *a aA aB aC*. A expressão *a** casa com nomes começando com a letra *a*, independente do tamanho do nome.

Exercício 11-2

$ ls -C

TEMP	*a*	*aA*	*aB*	*aC*
c10	*c11*	*c12*	*c13*	*c2*
c3	*c4*	*c5*	*c6*	*c7*
c8	*c9*	*conteudo*	*f1-1*	*f10-1*
f11-1	*f11-2*	*f2-1*	*f2-2*	*f2-3*
f7-1	*f8-1*	*f8-2*	*f3-1*	*f3-2*
t6-1	*t8-1*	*tmp*	*Zebu8*	

Qual o resultado para os comandos abaixo levando em consideração a saída do comando *ls -C* acima:

1.) *$ echo c**

2.) *$ echo ?8**

3.) *$ echo ?11**

4.) *$ echo *8**

5.) *$ echo c[12684xyz]*

6.) *$ echo [tf]**

7.) *$ echo c1[0-3]*

11.4. O perigo

Aqui queremos alertar para o poder e o perigo representados pelo matacaractere *. O poder no sentido de se evitar digitação desnecessária. O perigo está representado no exemplo a seguir.

O UNIX é um sistema operacional que supõe que você sabe o que esta fazendo. Você deve ter muito cuidado quando estiver usando metacaracteres em operações traumáticas onde não é possível voltar atrás. Por exemplo: você queria digitar o comando:

*$ rm c**

para remover todos os arquivos cujos nomes começassem com a letraco entre o *c* e o *)

Observe que o shell vai interpretar esta última linha de comando *$ rm c ** de forma diferente da primeira. Ou seja, ele entende que você quer remover o arquivo cujo nome é *c* e **todos** os arquivos do diretório corrente (lá se foram meus arquivos... e dependendo do sistema de backup, alguns finais-de-semana ficaram comprometidos para recuperar as informações jogadas fora). Tome cuidado! Isto pode acontecer com você um dia.

11.5. O metacaractere *[]*

O metacaratere *[ ]* indica uma **classe** de letras. A expressão *a[A-Z]* significa nome de arquivos com **duas** letras. A primeira letra deve ser *a* e a segunda letra pertence à classe *A, B, C, (...) ,Z,* das letras maiúsculas.

Note que os metacaracteres são interpretados pelo shell (*/bin/sh*). O programa não enxerga os metacaracteres. O shell os trata antes de executar o programa. Tomando como base o diretório do exercicio 11-2, o resultado do comando:

rm a[A-Z] (1)

é expandido para:

rm aA aB aC (2)

Note que o comando (2) já recebe o resultado da expansão antes de começar a executar. O utilitário *rm* não tem acesso aos parâmetros no formato (1).

Quem expande o * é o shell e não o núcleo do UNIX ou o programa *rm*. Isto significa maior produtividade para você que não tem que se preocupar com estes detalhes.

11.6. Aspas, Escape e Apóstrofos

A seguir vamos mostrar alternativas para neutralizar o efeito especial dos metacaracteres em algumas situações.

O shell é responsável pela expansão de metacaracteres. Como fazer para evitar que o shell **não expanda** metacaracteres?

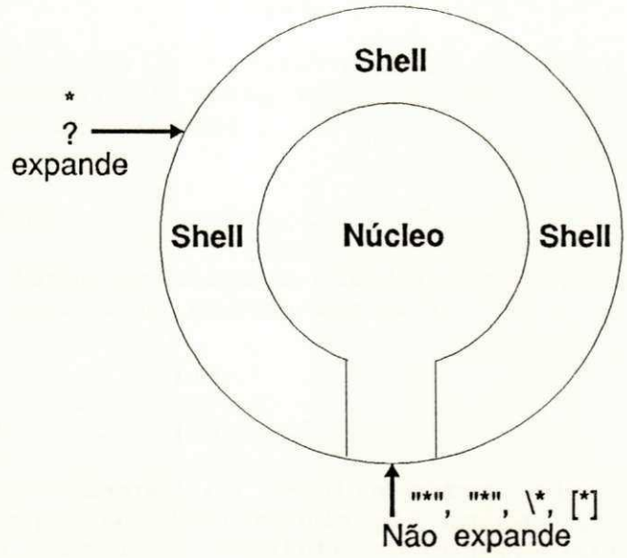


Figura 11.2: A organização do UNIX

A figura acima reflete a organização do UNIX. O shell é uma camada de software que permite a interface entre homem e o núcleo do UNIX. O fato de o shell ser um interpretador não impede que você use, diretamente, as rotinas do núcleo do UNIX. Observe também na figura acima que o núcleo do UNIX pode criar arquivos cujos nomes usam os caracteres *, ?, etc.

É possível que você precise criar arquivos cujos nomes contenham os caracteres especiais ou metacaracteres. Para evitar que o shell trate estes caracteres de forma especial e só usar aspas. Para exemplificar vamos supor que um determinado usuário quer criar um arquivo chamado *. Isto mesmo, pode parecer esquisito mas o nome do arquivo é asterisco. Como fazer?

```
$ cat > "*"
```

O comando acima cria o arquivo *. Este exemplo demonstra o quão permissivo é o UNIX.

Atenção: tome muito cuidado ao usar * como nome de arquivo. Um passo em falso e **todos** os seus arquivos do diretório corrente são afetados. Uma boa maneira de se proteger contra desastres é nunca criar arquivos cujos nomes que contenham *.

11.6.1 Aspas

Uma forma de evitar que o shell faça a expansão do metacaractere é colocá-lo entre aspas. As aspas tiram o efeito especial dos metacaracteres:

```
$ rm "*"                    (1)

    + rm *                  (2)
```

Neste caso, como o * estava entre aspas, o comando *rm* recebe o * como parâmetro. O shell sabe que os caracteres entre aspas não devem ser interpretados.

11.6.2. Escape

Outra forma de tirar o efeito especial dos metacaracteres é através do escape ou contra-barra \.

```
$ rm \*                     (1)

    + rm *                  (2)
```

O caractere **escape** ou **contra-barra** tira o efeito especial do caractere seguinte. No comando (1) acima, * informa ao shell que o caractere asterisco seguinte ao contra-barra deve ser tratado de forma simples. Já no comando expandido (2), observe que o *rm* recebe como argumento o caractere *.

11.6.3. Apóstrofos

O apóstrofo é semelhante às aspas. Tudo que estiver entre apóstrofos é protegido contra expansão de metaracteres pelo shell. Há uma pequena diferença entre apóstrofos e aspas que explicaremos quando falarmos de variáveis do shell.

Vejamos um exemplo de proteção de metacaracteres usando apostrofos. Vamos analisar como o shell trata o comando *rm '*'*:

```
$ rm '*'                    (1)

    + rm *                  (2)
```

O * do comando (1) não foi interpretado pelo shell. O resultado da expansão do comando (1), mostrado na linha (2), não expandiu o *. Neste caso, o * é passado como parâmetro para o comando *rm*.

11.6.4. Classes

Outra forma de tirar o efeito especial de metacaracteres é através de *[ ]*. Vejamos outro exemplo envolvendo um caractere especial na linha de comandos:

```
$ rm [*]                    (1)

+ rm *                     (2)
```

O comando (1) *rm [*]* foi digitado. O shell não expandiu o asterisco porque ele estava protegido por *[ ]*. O comando que vai ser executado está representado na linha (2).

Exercício 11-3

Qual o significado do comando *rm "a b"*?

PARTE III: PROGRAMAÇÃO DO SHELL

12. O SHELL COMO LINGUAGEM DE PROGRAMAÇÃO

Até este ponto estudamos caracteristicas do interpretador de comandos shell. Vimos explorando os aspectos interativos do shell.

Neste capitulo queremos ressaltar as principais necessidades de uma linguagem de programação e que o shell preenche perfeitamente estes requisitos.

12.1 Necessidades de um linguagem de programação

Todas as linguagens de programação necessitam dos seguintes recursos básicos:

* Empacotamento - uma forma de criar e executar programas;
* Entrada e saida - um meio de comunicação do programa com o mundo externo;
* Variáveis - usadas para armazenar valores durante a execução do programa;
* Laços e decisões - uma maneira de controlar o fluxo de execução dos comandos;
* Subrotinas - um jeito útil e estruturado de agrupar comandos.

O shell tem tudo isso! O shell é um interpretador de comandos e uma Linguagem de Programação. Veja como ele resolve cada item acima:

* Empacotamento - o empacotamento no shell é feito simplesmente colocando os comandos do UNIX em um arquivo texto. O arquivo texto, chamado de **script**, tem suas permissões alteradas com o comando *chmod +x script*, para se tornar executável;
* Entrada e saida - pode usar redirecionamento dos comandos com >, < ou outros operadores de redirecionamento ou ainda usar os comandos *read* e *readonly*;
* Variáveis - são todas do tipo string;
* Laços e decisões - existem várias formas *if*, *case*, *while*, *for* e *until*;
* Subrotinas - *scripts* e funções podem servir como subrotina no shell.

12.2. Filosofia da linguagem shell

O shell pode ser visto como um processador de strings, segundo a visão de [KORN 88]. É uma linguagem que resolve aplicações de manipulação de strings com facilidade.

Não existe diferença entre o interpretador de comandos shell e a linguagem de programação shell. O módulo executável é o mesmo */bin/sh*.

12.2.1 Scripts

Toda linguagem de programação permite que comandos sejam empacotados para formar programas. A forma como o shell empacotas seus comandos é chamado de **script**.

Os scripts são, na verdade, arquivos textos normais contendo os nomes dos comandos executáveis do sistema.

Vimos que o shell está permanentemente lendo comandos da sua entrada padrão e não do terminal. A titulo de exemplo, podemos empacotar comandos dentro de um arquivo, de nome *arq_comandos*, e executá-los da seguinte forma:

$ sh < arq_comandos

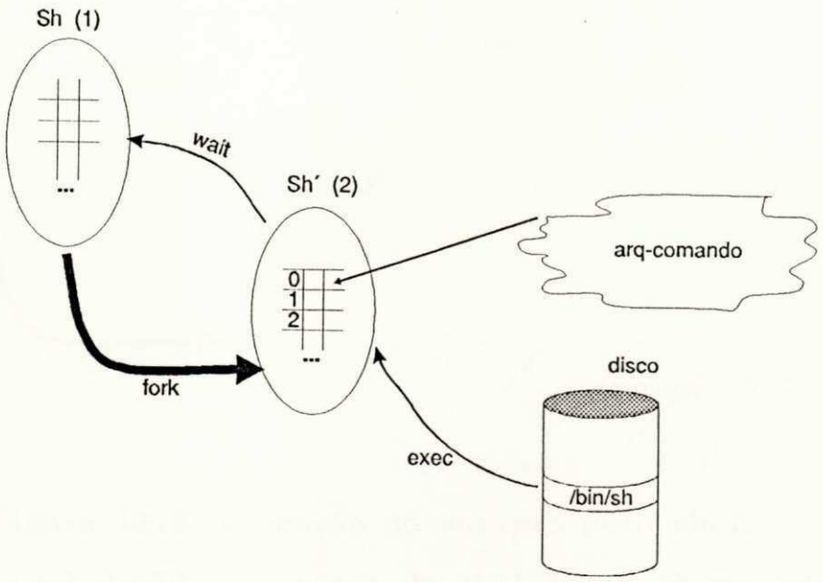


Figura 12.1: Execução de *scripts* pelo shell

Na figura acima, o shell (1) mostra o caractere de prontidão e lê o comando *sh < arq_comandos*. O shell (1) cria o shell (2) com a chamada *fork*. O shell (1) se sincroniza com o shell (2) através da chamada à rotina do núcleo *wait*.

O shell (2) agora está executando. Ele redireciona sua entrada padrão para o arquivo *arq_comandos* por causa do caractere *< arq_comandos*. O passo seguinte do shell (2) é carregar o utilitário */bin/sh* do disco para a memória através da rotina do núcleo *exec*.

O utilitário *sh* herdou a entrada padrão do shell (2). Agora quem está executando é o utilitário *sh* lendo comandos de sua

entrada padrão que é o arquivo *arq_comandos*.

Quem interpreta *arq_comandos* é um segundo shell. Este fato é importante, por causa dos efeitos colaterais de comandos no arquivo *arq_comandos* do tipo troca diretório corrente *cd* e outros. Voltaremos à esta discussão mais adiante.

O arquivo *arq_comandos* é chamado de *script* por conter comandos do UNIX. O shell lê e executa cada comando deste arquivo. Outra forma de executar *arq_comandos* é a seguinte:

$ sh arq_comandos

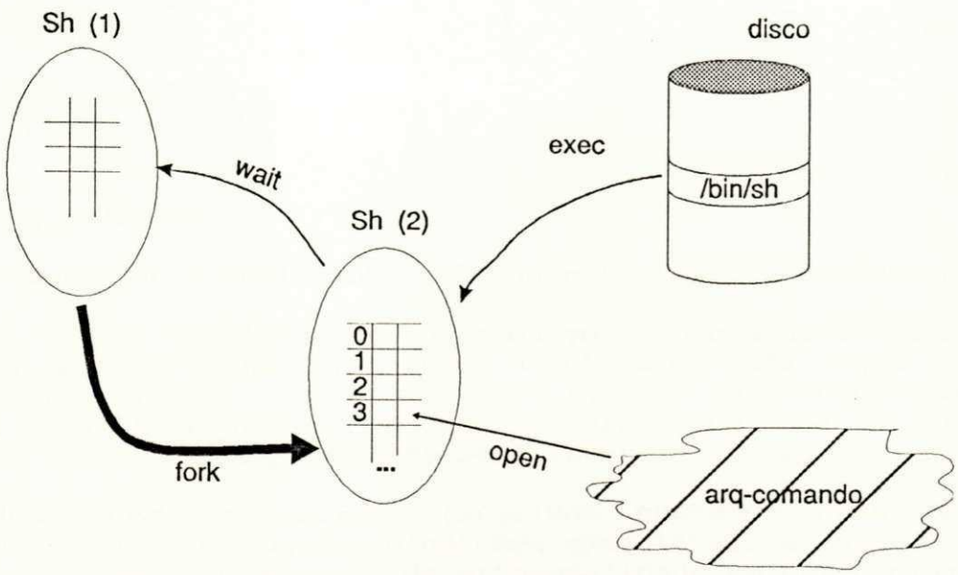


Figura 12.2: Execução de *scripts* pelo shell

isto é possível devido à sintaxe do utilitário *sh* que verifica se foi passado algum argumento na linha de comandos.

Note que na figura acima o utilitário *sh* não está lendo de sua entrada padrão (descritor 0). O arquivo *arq_comandos* foi passado como parâmetro e aberto usando o serviço do núcleo do UNIX *open* (provavelmente, usou o descritor 3).

É chato ter que digitar *sh* antes dos nomes dos scripts. Toda linguagem interpretada tem este incoveniente: ter que ativar o interpretador para executar os comandos desejados.

Digitar *sh comando* antes do nome do comando quando este é um script, e digitar simplesmente *comando* quando se trata de um arquivo executável normal, confunde o usuário e não é ortogonal. Como resolver este problema?

É simples, o shell faz o parsing da linha de comandos e o núcleo do UNIX através do serviço *exec* é quem, efetivamente,

ativa o comando.

Como tornar scripts executáveis?

Simplesmente liga o **bit x** do arquivo que contém os comandos através do comando *chmod* da seguinte forma:

$ chmod +x arq_comandos

Agora o script *arq_comandos* pode ser ativado digitando-se apenas seu nome, como os demais comandos do sistema:

$ arq_comandos

ao invés de:

$ sh < arq_comandos

ou

$ sh arq_comandos

Como o shell implementa internamente scripts executáveis?

O shell verifica se o arquivo *arq_comandos* tem o **bit x** ligado. Em caso afirmativo, chama a rotina do UNIX *exec* supondo que é um arquivo executável puro (gerado pelo compilador C ou Cobol, etc). Se o serviço do núcleo *exec* que faz a execução falhar, o arquivo *arq_comandos* será interpretado por um shell.

Resultado: os scripts (arquivos contendo comandos) são tratados como os outros programas executáveis do sistema. Mais uma vez temos o conceito de ortogonalidade em evidência: tanto comandos normais quanto *scripts* são executados da mesma forma.

Veja exemplo de alguns comandos que são scripts em algumas implementações do UNIX:

1) O comando *whodo* para mostrar o que cada usuário está fazendo:

$ whodo

```
Sun Aug  4 13:56:17 PDT 1991
01       alvaro   12:40
  01      119      0:00 ps
  01      121      0:00 who
02       ismenia  13:56
```

no exemplo acima *alvaro* e *ismenia* estão usando a máquina.

2) O comando *dirname* para imprimir o diretório base:

$ dirname /usr/alvaro/trab/tese/arq_comandos

/usr/alvaro/trab/tese

o resultado do comando *dirname* é imprimir o diretório base na saída padrão do percurso passado como parâmetro.

3) O comando *shutdown* para desligar ou encerrar as operações no sistema antes de desligar a máquina (normalmente usado pelo superusuário *root* ou administrador).

PARTE III: PROGRAMAÇÃO DO SHELL

13. VARIAVEIS

Neste capitulo vamos ver alguns aspectos envolvendo variáveis e o ambiente onde elas se encontram. Falaremos de parâmetros posicionais, variáveis pré_definidas do shell, o shell de login, o comando ".", os comandos *read* e *readonly*.

13.1. Variáveis

Algumas linguagens implementam vários tipos de variáveis. A Linguagem C, por exemplo, possue variáveis inteiras, reais, arranjos e apontadores.

13.1.1. Tipos de variáveis

Na Linguagem shell as variáveis são sempre do mesmo tipo: **string**. Não existem inteiros, arrays, apontadores ou reais, tudo é armazenado como cadeia de caracteres.

Por quê algumas linguagens exigem declarações prévias das variáveis?

* Para alocar espaço na memória (na definição);
* Para fornecer tipo, tamanho e escopo.

No shell o tipo das variáveis é sempre igual: cadeia de caracteres, as variáveis não precisam ser declaradas previamente:

* O tamanho da variável se ajusta dinamicamente;
* O escopo é sempre global ao script.

13.1.2. Atribuição e uso de variáveis

Variáveis shell não precisam ser declaradas previamente. A criação é automática. É só usar que ela passa a existir.

Programas ficam mais simples de escrever, por exemplo:

```
$ var=alvaro
$ echo Conteudo var=$var

    Conteudo var=alvaro
```

No exemplo acima a variável *var* foi criada contendo a string *alvaro*. A expressão *$var* representa o conteúdo da variável *var*.

13.1.3. Variáveis como macros

Variáveis podem ser vistas como macros que são expandidas pelo shell na linha de comandos. Por exemplo, o comando abaixo:

```
$ var=alvaro
```

cria a variável *var* no ambiente do shell. Já o comando a seguir:

```
$ echo Conteudo var=$var
```

faz com que a macro *$var* seja substituida pelo seu valor que é

alvaro. Visualizando a linha expandida pelo shell através da opção *-x* do comando *set*, temos o seguinte:

```
+ echo Conteudo var=alvaro
```

depois de feita a substituição (note que colocamos o caractere + para indicar que a linha foi expandida pelo shell) o comando *echo* é executado.

Podemos usar *${var}* no caso de a variável *var* ser seguida de um caractere diferente de branco:

```
$ echo X${var}Y

    XalvaroY
```

13.2. Parâmetros posicionais

A seguir vamos mostrar como é feita a comunicação entre scripts ou módulos shell com a linha de comandos.

Por quê os parâmetros posicionais são declarados na maioria das linguagens?

A resposta é para saber como recuperar a informação que foi passada como parâmetro. Se o programa chamador passar uma variável do tipo caractere, por exemplo, a rotina receptora saberá que deve recuperar um byte. Por outro lado, se o programa passar como parâmetro um valor real, a rotina receptora saberá que deve recuperar quatro bytes, se os números reais forem representados em quatro bytes.

Por quê o shell não precisa da declaração de parâmetros posicionais?

Já dissemos a resposta anteriormente: porque todas as variáveis são do tipo string e a forma de como recuperar é uma só. Vejamos, por exemplo, o script *mos_parametros* que mostram os parâmetros posicionais:

```
$ cat mos_parametros

    echo parametro1=$1
    echo parametro2=$2
    echo numero de parametros=$#
```

Vamos ativar o script *mos_parametros* passando os seguintes argumentos *alo* e *mundo* como parâmetros:

```
$ mos_parametros alo mundo

    parametro1=alo
    parametro2=mundo
    numero de parametros=2
```

Se o script *mos_parametros* fosse ativado com os argumentos *alvaro, jacques* e *anna*:

```
$ mos_parametros alvaro jacques anna

     parametro1=alvaro
     parametro2=jacques
     numero de parametros=3
```

o terceiro parâmetro *anna* não seria impresso, porque dentro do script *mos_argumentos* o *$3* não é usado.

Os parâmetros posicionais são referenciados de *$1* a *$9*. Falaremos mais adiante como tratar do décimo parâmetro posicional em diante.

Existem outros parâmetros posicionais. A seguir listamos alguns:

- *$#* - indica o número de argumentos que foram passados para o script;
- *$0* - indica o nome do arquivo que contém o script;
- *$** - todos os parâmetros *$1 $2 ...*;
- *$@* - todos os parâmetros *$1 $2 ...* (existe uma diferença entre *$** e *$@* que explicaremos com exemplos mais adiante);
- *$-* - contém as opções correntes do shell;
- *$?* - contém o código de retorno do último comando executado;
- *$$* - indica o número do processo corrente;
- *$!* - indica o número do último processo executado em retaguarda.

13.2.1. Atribuição de parâmetros posicionais

Queremos mostrar que os parâmetros posicionais ou recebem valor através da ativação da linha de comandos ou através do comando *set*.

A atribuição de parâmetros posicionais é feita com o comando interno do shell *set*. O comando *set* também é usado para manipular opções para o funcionamento do shell: *set -x* ativa a impressão da expansão dos comandos antes de serem executados.

Veja a seguir um exemplo do uso do comando *set* para fazer atribuição de valores aos parâmetros posicionais:

```
$ set Alo Brasil
$ echo $1

     Alo

$ echo $2

     Brasil

$ echo $#

     2
```

Os caracteres aspas, apóstrofos e escape podem ser usados com variáveis e parâmetros:

```
$ mos_parametros "alo mundo"

     parametro1=alo mundo
     parametro2=
     numero de parametros=1
```

o caractere branco ' ' entre *alo* e *mundo* não foi interpretado como separador de argumento porque estava entre aspas.

```
$ var="alo mundo"
$ mos_parametros "$var"

     parametro1=alo mundo
     parametro2=
     numero de parametros=1
```

a macro *$var* foi substituida pelo seu valor *alo mundo* apesar de estar entre aspas. Em outras palavras, o shell faz substituição de variáveis entre aspas mas não faz expansão de caracteres especiais.

```
$ var="alo mundo"
$ mos_parametros '$var'

     parametro1=$var
     parametro2=
     numero de parametros=1
```

o shell não faz nem expansão nem substituição de variáveis do que estiver entre apóstrofos ' ... '.No exemplo acima "*$var*".

13.3. O ambiente

Aqui queremos mostrar que todos os processos possuem um ambiente composto entre outras informações, de variáveis locais e globais ao processo.

Todos os processos no shell têm seu próprio ambiente. O ambiente é uma área de memória onde ficam armazenadas as variáveis, o diretório corrente e outras informações sobre o processo. Podemos dividir o ambiente em duas áreas distintas:

* área global
* área local

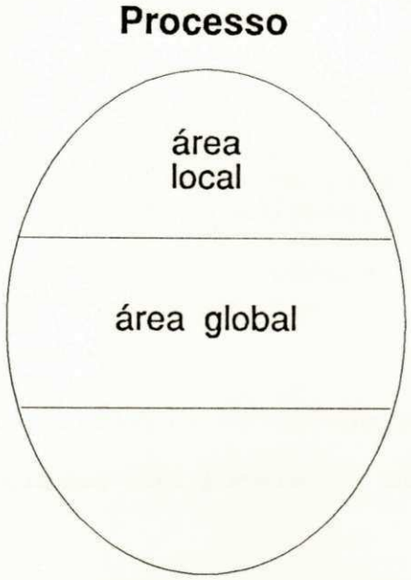


Figura 13.1: Variáveis globais e locais

O processo de execução de comandos no UNIX é sempre o mesmo: o processo pai cria um filho. O processo filho herda apenas o ambiente global do pai.

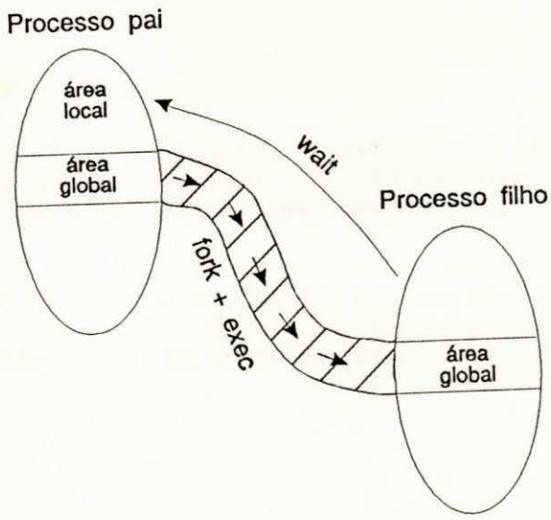


Figura 13.2: Herança das variáveis globais

Para saber as variáveis globais do ambiente, usamos o comando interno do shell *env*:

```
$ env

     HOME=/u/alvaro
     PATH=/bin:/usr/bin:.
     PS1=$
     SHELL=/bin/sh
     TERM=ansi
```

No exemplo acima temos uma relação das variáveis globais que são passadas aos processos filhos. As variáveis listada são usadas pelo sistema. Falaremos delas mais adiante.

Para listar todas as variáveis usamos o comando *set* sem parâmetros. Por exemplo:

```
$ set

     ...       (todas as variáveis incluindo as globais)
```

Variáveis recém-criadas são locais ao ambiente. Por exemplo, o comando abaixo:

```
$ a=alo
```

cria, no ambiente uma variável cujo nome é *a*. Na realidade, *a* é uma variável local. Para tornar *a* uma variável global, temos que exporta-la através do comando interno do shell *export*:

```
$ export a
```

Agora a variável *a* é global porque ela foi exportada através do comando *export*.

13.4. Diretório HOME

Queremos mostrar aqui que o diretório HOME é o diretório inicial a partir do qual começamos a trabalhar quando fazemos *login*. Veremos também como é sua implementação.

O diretório HOME é, na realidade, uma variável global no ambiente de cada processo. O conteúdo da variável *HOME* reflete o diretório inicial de cada usuário ou processo. No exemplo anterior, quando falavamos do comando *env*, tinhamos a seguinte saída:

```
$ env

     ...
HOME=/usr/alvaro
     ...
```

A saída do comando *env* nos mostra a linha *HOME=/usr/alvaro*. Esta linha indica que o diretório HOME do usuário é */usr/alvaro*.

O diretório HOME é escolhido pelo administrador do sistema no momento do cadastro do mesmo no sistema. Geralmente é */usr/nome-usuário*. O arquivo que contém os dados cadastrais, inclusive o diretório HOME, é */etc/passwd*.

13.4.1. Diretório corrente

Uma vez que começamos a trabalhar no diretório HOME, podemos mudar de diretório através do comando *cd*. O nome *cd* é uma abreviação do inglês "change directory". Este comando recebe o nome do diretório para onde queremos ir. Por exemplo, para mudar para o diretório de trabalho de *jacques* usamos o seguinte comando:

```
$ cd /usr/jacques/src/doc/livro
```

Se nós tivermos permissão para mudar para o diretório de jacques, o diretório corrente passará a ser */usr/jacques/src/doc/livro*.

Por outro lado, se o comando *cd* for usado sem parâmetros a mudança de diretório será feita levando em consideração o valor da variável *HOME*. Por exemplo, supondo que o diretório corrente é */usr/jacques/src/doc/livro* e que a variável *HOME* tem valor igual a */usr/alvaro* (*HOME=/usr/alvaro*), o comando *cd* sozinho faz a mudança para um diretório pré-definido:

```
$ cd
```

O comando *cd* sozinho faz com que o diretório corrente passe a ser o diretório HOME, ou seja, neste caso, */usr/alvaro*.

A mudança de diretório consiste na atualização de uma variável interna ao processo. A seguir citaremos algumas das principais variáveis pré-definidas do UNIX.

13.5. Variáveis do sistema

Existem variáveis pré-definidas para uso do sistema. Vamos mostrar as principais com uma breve descrição para que elas servem:

PATH informa ao shell qual o nome dos subdiretórios que devem ser pesquisados à procura do comando a ser executado;

HOME indica o nome do subdiretório a partir do qual o usuário começa a trabalhar;

PS1 contém a cadeia de caracteres que é usada para

indicar ao usuário que o shell está pronto para executar seus comandos. É a cadeia de caracteres do símbolo de prontidão primário (geralmente é $);

PS2 contém a cadeia de caracteres que é usada para o símbolo de prontidão secundário (geralmente é >). Ele é usado em comandos do shell que se estende por mais de uma linha física;

TERM informa ao processo qual o nome do terminal correntemente em uso. Este nome é usado para identificar as características do terminal em um banco de dados, chamado terminfo, que descreve os terminais do UNIX.

13.6. O shell de login

Ao iniciar sua seção no sistema o usuário recebe um caractere de prontidão ($) indicando que existe um interpretador shell disponível. Este é o chamado shell de login por ter sido executado a partir do comando *login*.

O shell de login processa o script do sistema chamado */etc/profile* antes de executar o script de personalização de cada usuário *.profile*.

Normalmente o administrador do UNIX utiliza o */etc/profile* para criar variáveis que são comuns a todos os usuários. Veja um exemplo:

```
# cat /etc/profile

     ...
IWPATH=/usr/lib/iw
export IWPATH
     ...
```

no caso acima temos a criação de uma variável usada pelo Processador de texto *Infoword* da Infocon. Todos os usuários terão esta variável definida no seu ambiente global (*export IWPATH*, fez com que a variável *IWPATH* se tornasse global).

O passo seguinte, executado pelo shell de login, é processar o arquivo *.profile*. Se você quiser fazer alguma personalização na sua sessão de trabalho, é só editar o arquivo *.profile* e colocar os comandos desejados. Por exemplo, para alterar o valor da variável *IWPATH* apenas para a sua sessão é só colocar a linha de comando abaixo no arquivo *.profile*:

```
IWPATH=/usr/novo_diretorio
export IWPATH
```

o resultado da atribuição do comando acima é:

```
x1 <-     alvaro
x2 <-     olhai
x3 <-
```

13.10. O comando *readonly*

O objetivo do comando *readonly* é para proteger variáveis contra modificações indesejáveis. O formato do comando é o seguinte:

```
readonly var1 var2 var3 ...
```

A partir do instante que o comando *readonly* foi aplicado a uma variável, esta não poderá mais ser alterada. A variável poderá ser usada apenas para leitura. Por exemplo, o comando:

```
$ TERM=vt100
```

indica que o terminal correntemente sendo usado é da DEC e segue o padrão *vt100*. A variável *TERM* pode ser alterada:

```
$ TERM=ansi
```

no comando acima, o usuário mudou o padrão para *ansi*. Se quisermos travar a variável *TERM* contra mudanças podemos digitar o comando:

```
$ readonly TERM
```

A partir deste instante, a variável *TERM* não poderá mais ser usada para alteração. Vejamos uma tentativa de alteração frustrada:

```
$ TERM=agix

    TERM: is read only
```

o shell mostrará a mensagem *TERM: is read only* indicando que a variável só poderá ser usada para leitura.

13.11. Comandos internos ao shell

A seguir faremos uma relação dos comandos internos do shell.

Uma pergunta nos ocorreu agora: o que significa comando interno ao shell?

Para o usuário é transparente, ele não vê difença, do ponto de vista de execução do comando, entre comandos internos e comandos normais. O mecanismo normal de execução de comandos no shell utiliza alocação de memória e o disco.

O significado de comandos internos do shell são aqueles comandos que já vêm dentro do próprio shell (*/bin/sh*). Comandos internos do shell são executados internamente pelo shell. Não há necessidade de acesso ao disco para pesquisar o comando a ser executado.

Antes de executar um comando, o shell verifica se o mesmo é o nome de um dos seus comandos internos. Isto é, quando o usuário digita um comando em resposta ao caractere de prontidão *$*, o shell verifica se este é um comando interno através do algoritmo abaixo:

```
se comando_digitado for igual a cd
então
     mude diretório

se comando_digitado for igual a read
então
     leia lista de variáveis

se comando_digitado for igual a set
então
     manipule ambiente do processo
     ...
```

A seguir listaremos os principais comandos internos do shell com uma breve descrição do que ele faz:

cd	para mudança de diretório corrente;
continue	usado pelo shell para alterar o fluxo de execução dos comandos de controle de fluxo *for*, *while* e *until*;
eval	comando para incrementar o número de vezes que o shell faz o parsing da linha de comandos;
exec	faz a carga e execução de um comando do disco sobrepondo o processo corrente;
exit	termina um processo;
export	torna uma variável global;
for	comando para fazer um tarefa um número constante de vezes;
if	verifica se uma tarefa foi executada com sucesso. Permite tomar uma decisão em relação ao resultado da tarefa;
newgrp	o processo passa a pertencer a um novo grupo de usuários;

read	lê uma linha da entrada padrão e a armazena em uma lista de variáveis;
readonly	muda o atributo de uma variável para permitir apenas leitura;
set	manipula variáveis e opções do ambiente shell;
shift	desloca os parâmetros posicionais *$1, $2, ...*;
test	avalia expressões condicionais;
time	mostra o tempo de execução de um comando;
trap	faz o tratamento de sinais durante a execução do comando;
ulimit	limita o tamanho de arquivos gravados por processos filhos;
umask	define permissões default para a criação de arquivos e diretórios;
until	uma forma alternativa do comando de controle de fluxo *for*. Serve para executar uma tarefa um número variável de vezes, dependendo de uma condição ser falsa;
wait	espera que um processo em retaguarda seja completado. Este comando servee para o interpretador de comandos shell se re-sincronizar com programas rodando em retaguarda;
while	uma forma alternativa do comando de controle de fluxo *for*. Serve para executar uma tarefa um número variável de vezes, dependendo de uma condição ser verdadeira.

PARTE III: PROGRAMAÇÃO DO SHELL

14. ENTRADA E SAIDA

Neste capitulo queremos relembrar alguns pontos sobre Entrada e Saida padrão que são importantes para criação de scripts. Discutiremos, superficialmente, uma forma para emissão de relatórios mais elaborados através do comando *awk*.

14.1. O que já foi dito sobre Entrada e Saída

No capítulo cinco vimos o quão importante é o conceito de entrada e saída padrão para a filosofia UNIX. Filtros simples como o comando *cat* se tornam ferramentas poderosas devido a flexibilidade do UNIX redirecionar a entrada e a saída dos comandos.

Comandos que lêem a entrada padrão e ou escrevem na saída padrão podem ser combinados livremente.

O shell fornece operadores para redirecionamento >, < e outros símbolos. Dispositivos são tratados como arquivos. O shell é quem trata o redirecionamento para todos os comandos do UNIX. Os comandos não precisam saber de onde estão lendo ou onde estão gravando, o que significa maior produtividade para você que não precisa se preocupar com redirecionamento.

14.2. O comando *read*

A linguagem de programação Shell tem um comando interno que permite a leitura de uma linha da entrada padrão: o comando *read*, conforme falamos no Capítulo anterior.

A sintaxe do comando *read* permite que toda uma linha seja armazenada em uma lista de variáveis:

```
$ read var1 var2 ...

_
```

ao chamar o comando *read* o cursor _ fica esperando que algo seja digitado da entrada padrão.

Vejamos o exemplo abaixo para entendermos melhor como o comando *read* funciona:

```
$ read X Y

    Alvaro Francisco de Castro Medeiros


$ echo $X

    Alvaro


$ echo $Y

    Francisco de Castro Medeiros
```

como já falamos no capítulo anterior, a primeira variável recebe

a primeira palavra:

X <- Alvaro

e a segunda variável, como é a última da lista, recebe o restante do texto:

Y <- Francisco de Castro Medeiros

Segundo [MORG 86], em versões do UNIX mais recentes, da versão V.2 em diante, é possível redirecionar o comando *read*.

14.3. As variáveis *OFS* e *IFS*

Aqui queremos mostrar que o caractere usado para separar palavras pode ser configurado.

Os caracteres defaults que servem como separador de palavra são: o caractere gerado pela tecla *Tab*, o caractere gerado pela tecla *ESPACO* e o caractere gerado pela tecla *Enter*. Eles ficam armazendados na variável pré-definida do sistema *IFS* ("Input Field Separator", traduzindo: caracteres separadores de campos na entrada).

Se quisermos usar outros caracteres como separador de campo na entrada, é só mudar o valor da variável *IFS* e pronto. Por exemplo, para se usar o caractere + como separador de campo faríamos o seguinte:

```
$ TMP=$IFS
```

salvaríamos o valor original da variável *IFS* para poder voltar ao normal no futuro. Mudaríamos a variável *IFS* para o separador desejado, no caso +, por exemplo:

```
$ IFS="+"
```

Pronto, agora o shell vai saber que + indica separador de palavra na entrada. Se um comando *read* for executado para ler valores e armazenar nas variáveis *X*, *Y*, *Z* e *W*:

```
$ read X Y Z W

_
```

o shell vai ficar esperando que o usuário digite dados. Vamos supor que o usuário digitou o seguinte:

Alvaro Francisco+de+Castro+Medeiros

Qual seria o valor das variáveis *X* e *Y*?

Para saber a resposta é só fazer o casamento das palavras digitadas na entrada com as variáveis *X*, *Y*, *Z* e *W* respectivamen-

te. O resultado é o seguinte:

X <- Alvaro Francisco
Y <- de
Z <- Castro
W <- Medeiros

Onde a variável *X* recebeu o valor *Alvaro Francisco* porque o separador de campo (o caractere +) apareceu logo após *Alvaro Francisco+*.

De forma análoga à variável *IFS*, existe a variável *OFS* (Output Field Separator) que controla o separdor de campos na saída.

14.4. O comando *echo*

Aqui queremos mostrar o comando default do UNIX para imprimir mensagens na saída padrão.

O comando *echo* é usado para imprimir uma mensagem na saída padrão. O comando abaixo:

$ echo mensagem

imprime a mensagem *mensagem* na saída padrão. O que significa simplicidade.

Praticamente todos os comandos do UNIX usam a saída padrão, o que permite a combinação entre eles: ortogonalidade levando à produtividade.

Em implementações mais recentes do UNIX, existem duas versões do comando *echo*: uma interna (embutida) e outra em disco. A versão interna é usada em construções simples, onde não há redirecionamento da saída. A versão que está no disco */bin/echo* é usada em expressões que envolvam redirecionamento da saída. Já a versão embutida do comandos *echo* possibilita melhor performance de execução, não tendo que criar um novo processo.

14.5. O comando *awk*

Aqui queremos mostrar que é possível emitir relatórios mais elaborados através do comando *awk*. Ou em outras palavras, responder à pergunta: Como formatar relatórios para ter cabeçalho bonitinhos?

Reservamos o Capítulo vinte e um para falar com mais detalhes do filtro *awk* pela sua riqueza em recursos.

Relatórios grosseiros podem ser construidos usando o comando *echo*. Usuários que necessitem de saídas mais elaboradas podem usar o comando *awk*. Esta é uma ferramenta muito poderosa para a

geração de relatório e manipulação de arquivos de uma forma geral.

Um dos fatores positivos na ferramenta *awk* é que podemos gerar relatórios a partir da linha de comandos do shell. É só ativar o *awk* passando alguns parâmetros e pronto: o relatório será gerado. Vamos mostrar um exemplo do comando *awk* para emitir o seguinte relatório:

```
Nome            Dir HOME
--------        -------------
root            /
alvaro          /usr/alvaro
jacques         /usr/jacques

          ...
```

Observe que o arquivo de dados de entrada para o *awk* deverá ser o arquivo do sistema */etc/passwd*. A maneira de ativar o comando *awk* é igual aos outros comandos:

```
$ awk -F: -f prog.k < /etc/passwd
```

Onde *awk* é o nome do comando *-F:* e *-f prog.k* são parâmetros passados ao comando *awk*. Onde -F: significa que os campos na entrada serão separados por *:*. A opção *-f prog.k* indica que o programa *prog.k* contém comandos *awk*. A terminação *.k* é uma convenção, por nós adotada, para arquivos contendo programas em *awk*. No comando acima, o *awk* lê dados da sua entrada padrão que foi redirecionada pelo shell para o arquivo */etc/passwd*.

awk é uma linguagem de programação voltada para a geração de relatórios e manipulação de arquivos. A linguagem *awk* possue uma sintaxe própria. Veja o programa a seguir:

```
$ cat prog.k

BEGIN { printf ( "%-20s\t%-20s\n", "Nome",      "Dir HOME" )
        printf ( "%-20s\t%-20s\n", "--------", "----------" )
      }

      { printf "%-20s\t%-20s\n", $1, $6 }
```

A primeira vista é meio estranha a sintaxe da linguagem *awk*. Você deve estar se perguntando: Como funciona o comando acima?

É simples, a sintaxe de todo programa na linguagem *awk* tem o seguinte formato:

```
condição1 { ação1 }
condição2 { ação2 }
```

O programa é composto de linhas. Cada linha é dividida em duas partes: uma **condição** e uma **ação**. Durante a execução do programa, para cada linha de dados lidas na entrada, as condições são

avaliadas. As ações serão executadas para as condições verdadeiras.

Acompanhando a execução para o exemplo anterior: a palavra reservada *BEGIN* é interpretada pelo *awk* como uma condição especial que significa: a ação associada a esta condição que deve ser executada antes de qualquer leitura da entrada. A ação associada ao *BEGIN* é imprimir o cabeçalho formatado. O trecho de programa que faz a impressão é o seguinte:

```
BEGIN { printf ( "%-20s\t%-20s\n","Nome",      "Dir HOME" )
        printf ( "%-20s\t%-20s\n","--------", "----------")
}
```

A ação do trecho de programa acima usa a função pré-definida da linguagem *awk*: *printf*. O resultado desta ação é impresso na saída padrão:

```
Nome        Dir HOME
--------    ----------
```

Onde *"%-20s\t%-20s\n"* informa ao *printf* que duas cadeias de 20 caracteres devem ser impressas separadas por um caractere de tabulação. Por outro lado, a ação:

```
{ printf "%-20s\t%-20s\n", $1, $6 }
```

é executada para cada linha de dados lida pelo *awk* da entrada padrão, imprimindo na saída padrão o nome do usuário (*$1*) e o diretório HOME (*$6*):

```
alvaro      /usr/alvaro
jacques     /usr/jacques
        ...
```

Lembre-se que a entrada padrão foi redirecionada para */etc/passwd* pelo shell e que o formato deste arquivo:

```
$ cat /etc/passwd

        ...
alvaro::100:51::/usr/alvaro:/bin/sh
jacques::101:51::/usr/jacques:/bin/sh
        ...
```

tem o primeiro campo (*$1*) os nomes dos usuários e o sexto campo (*$6*) os nomes dos diretórios HOME.

14.6. Saída padrão de erro

Aqui queremos relembrar que o fato de as mensagens de erro poderem ser separadas das mensagens normais aumenta sua produtividade.

O UNIX tem um descritor de arquivo especial para mensagens de erro. Esta funcionalidade permite que mensagens normais sejam separadas das mensagens de erro automaticamente, aumentando a produtividade do usuário que não precisa procurar as mensagens de erro manualmente.

A sintaxe usada para fazer o redirecionamento da saída padrão de erro é a seguinte:

$ comando 2> arquivo

No comando acima, a expressão *2>* informa ao shell para redirecionar a saída padrão de erro para o arquivo *arquivo*.

Outra forma de redirecionar a saída padrão de erro é a seguinte:

$ comando > arquivo 2>&1

Onde *2>&1* tem o seguinte significado para o shell: redirecione a saída padrão de erro para o mesmo local da saída padrão normal.

PARTE III: PROGRAMAÇÃO DO SHELL

15. TOMADA DE DECISÃO

Neste capitulo vamos apresentar alguns comandos da linguagem shell para tomada de decisão. Estes comandos alteram o fluxo natural de execução de comandos. A seguir, estudaremos os comandos *if*, *exit*, *test* ou *[...]*, *&&*, *||*, *shift*, e *case*. Veremos também outras formas de comandos condicionais envolvendo variáveis.

15.1. Execução de um comando

Com relação a execução de um comando, como saber se um comando foi executado com ou sem sucesso?

A primeira resposta, provavelmente, seria a observação visual da execução do comando. Se não aparecer mensagens de erro no vídeo, significa que o programa foi executado com sucesso. Outra pergunta vem à tona: como distinguir mensagens normais das mensagens de erro?

Voltamos à questão inicial: O que significa um programa executar com sucesso?

15.2. Sucesso de um programa

Aqui queremos definir o significado de sucesso na execução de um programa. O significado de sucesso de um programa depende de cada utilitário. Vamos tomar como exemplo alguns comandos do UNIX e ver o que significa sucesso para cada um deles:

grep se o padrão pesquisado for encontrado significa sucesso;

ls se o diretório for lido significa sucesso;

cat se a entrada for lida significa sucesso.

Observe que cada utilitário tem uma interpretação particular para sucesso. É praticamente impossível decorar o significado para o sucesso de todos os comandos do UNIX. Temos um problema: como descobrir genericamente se um comando obteve sucesso?

Existe uma forma automática para se testar se o programa foi executado com sucesso. Trata-se de uma convenção adotada por todos os programas do UNIX: é um código de retorno do processo filho para o processo pai. A convenção é muito simples: se o programa terminar com sucesso o código de retorno deve ser 0. Qualquer outro valor indica que o programa não obteve sucesso.

O shell reservou uma variável especial *$?* para armazenar o código de retorno dos comandos executados. Por exemplo, o comando para pesquisar o nome *alvaro* no arquivo */etc/passwd* é o seguinte:

```
$ grep alvaro /etc/passwd
```

o comando acima mostrou na saída padrão:

```
alvaro::100:51::/u/alvaro:/bin/sh
```

indicando que o padrão *alvaro* realmente estava cadastrado. Se quisermos ver qual foi o código retornado, podemos digitar o comando *echo* abaixo passando como parâmetro o nome da variável do

shell *$?*. Esta variável armazena o status de execução do último comando:

```
$ echo $?

     0
     $
```

O comando acima mostra que o conteúdo da variável *$?* é *0* indicando que o último comando foi executado com sucesso.

Se usarmos o *grep* para pesquisar um usuário inexistente, por exemplo *aloysius*, não teremos nada impresso na saída:

```
$ grep aloysius /etc/passwd

     $
```

O comando acima não imprimiu nada na saída padrão. Vejamos o valor de *$?*:

```
$ echo $?

     1
     $
```

O valor da variável *$?* acima é diferente de *0* o que significa que o comando *grep aloysius /etc/passwd* não obteve sucesso.

15.3. O comando *if*

Aqui queremos mostrar uma forma automática para testar se um comando foi executado com sucesso.

O código de retorno de um comando poder ser interpretado automaticamente através do comando *if*. Por exemplo, o comando *grep* para pesquisar um usuário poderia ser re-escrito:

```
$ if grep alvaro /etc/passwd
> then
>     echo GREP COM SUCESSO
> fi
```

No comando acima temos dois sinais de prontidão. O símbolo de prontidão normal "*$* " e o secundário "> ". O símbolo de prontidão secundário é apresentado para o usuário saber que o comando se estende por mais de uma linha física.

O comando *if* é um comando interno do shell, assim como o *cd*. Ele é composto das palavras chaves *if*, *then* e *fi*.

No exemplo acima, se o comando *grep* retornar *0* (sucesso) o comando (*echo*) que está entre as palavras reservadas *then* e *fi* será executado. No exemplo acima, como alvaro está cadastrado, o

resultado impresso na tela é o seguinte:

```
alvaro::100:51::/u/alvaro:/bin/sh
GREP COM SUCESSO
```

15.3.1. Sintaxe do comando *if*

A forma geral do comando *if* é:

```
if condição
then
     comandos-sucesso     (1)
else
     comandos-insucesso   (2)
fi
```

Onde o comando *condição* é executado. Se o *código de retorno* do comando *condição* for **0**, os comandos *comandos-sucesso* (1) serão executados. Caso contrário, os comandos *comandos-insucesso* (2) serão executados.

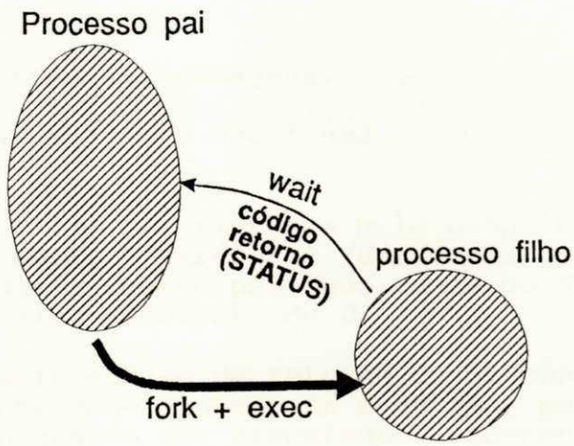


Figura 15.1: Código de retorno entre processo pai e filho

A figura acima representa o mecanismo de execução de processo no UNIX. Este esquema de execução de processos permite que um processo filho entregue ao processo pai um STATUS. Este STATUS representa o motivo do término do filho ou de sua morte. No caso do comando *if*, o shell é o processo pai. O processo filho está executando um comando. O STATUS é um código numérico que o *if* testa. Se for igual a zero, então o comando foi bem sucedido e a *condição* é considerada verdadeira.

O STATUS é subdividido em duas partes: uma parte é atualizada pelos script shell, através do comando *exit*, e a outra parte é preenchida pelo próprio núcleo do UNIX:

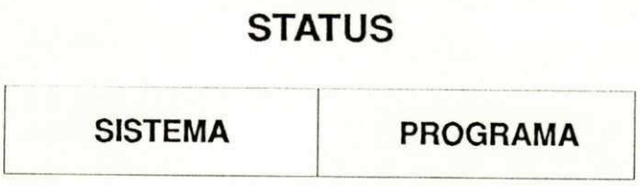


Figura 15.2: O STATUS de um processo

Vejamos um exemplo de um comando que não obtém sucesso:

```
$ if ls /dir/inexistente
> then
>     echo SUCESSO
> else
>     echo INSUCESSO
> fi
```

imprime na sua tela as mensagens:

```
ls: /dir/inexistente not found          (1)
INSUCESSO                               (2)
```

A mensagem (1) foi impressa pelo próprio comando *ls* por não ter conseguido abrir o diretório */dir/inexistente*. A mensagem (2) foi impressa pelo processo pai, por causa do STATUS do processo filho (*ls*) ter sido diferente de *0*.

Não confundir código de retorno ou STATUS com saída padrão. Alguns comandos, como o *grep*, *ls* e outros, por exemplo, imprimem na saída padrão quando são executados. Independende da impressão na saída padrão, o STATUS de um comando será sempre atribuido à variável *$?*.

Podemos usar um comando *if* dentro do outro repetidamente. Por exemplo:

```
if condição-1
then
     ação-1
else
     if condição-2
     then
          ação-2
     else
          if condição-3
          then
               ação-3
          else
               ação-4
          fi
     fi
fi
```

O comando *if* possue uma formato *elif* alternativo quando só queremos usar a parte *else* de *ifs* aninhados. O comando acima poderia ser re-escrito de maneira mais legivel com *elif* da seguinte forma:

```
if condição-1
then
     ação-1
elif condição-2
then
     ação-2
elif condição-3
then
     ação-3
else
     ação-4
fi
```

Veja exemplo do uso do comando *if* na Seção 15.5.

15.4. O comando *exit*

O comando *exit* é usado para finalizar um *script*. Um parâmetro pode ser passado para o comando *exit* indicando o STATUS ou causa do término do processo.

O comando *exit 0* deve ser usado para terminar um processo com sucesso. Já o comando *exit 1*, *exit 2*, ..., deve ser usado quando o *script* não conseguir completar seu objetivo, indicando insucesso.

15.5. O comando *test* ou *[...]*

O comando *test* avalia uma expressão e seu STATUS indica se a expressão é verdadeira ou falsa. O exemplo abaixo mostra o uso tipico do comando *test* e do comando *exit*:

```
$ cat script

     var=10
     if test  $var -gt 2
     then
          exit 1
     fi
       ...
     exit 0
```

Onde a expressão acima *$var -gt 2* testa se o conteúdo da variável *var* é maior que *2*. Note que o comando *exit 1* termina o programa e retorna um STATUS de insucesso para o processo chamador do script *script*.

Usuário que não gosta de escrever *test expressão* poderá digitar *[expressão]* O efeito computacional é exatamente o mesmo. Por exemplo, o comando:

```
test  $var -gt 2
```

faz a mesma coisa que:

```
[ $var -gt 2 ]
```

A escolha fica por conta da preferência estética de cada programador.

Veja a seguir os operadores que podem ser usados para fazer comparação usando o comando *test*.

15.5.1 Expressões do comando *test*

O comando *test* usa a expressão *-gt*, para indicar *maior que*, ao invés do símbolo >. O símbolo > não pode ser usado nas expressões de *test* porque ele tem efeito especial para o shell (redirecionamento).

Os operadores que podem ser usados para construir expressões para o comando **test** (ou [ ]) são os seguintes:

OPERADOR	PARÂMETRO	VERDADE SE ...
-r	*arquivo*	pode ler *arquivo*
-w	*arquivo*	pode gravar em *arquivo*
-x	*arquivorquivo*	pode ler *arquivo*
-w	*arquivo*	pode gravar em *arquivo*
-x	*arquivo*	pode executar *arquivo*
-d	*nome*	*nome* é um diretório
-s	*arquivo*	*arquivo* existir e tiver dados
-z	*cadeia*	*cadeia* for nula ("")
-n	*cadeia*	*cadeia* for diferente de nula
cad1 = cad2		*cad1* for igual a *cad2*

cad1 != cad2	*cad1* for diferente de *cad2*
val1 -eq val2	*val1* é igual a *val2*
val1 -gt val2	*val1* é maior que *val2*
val1 -ne val2	*val1* é diferente de *val2*
val1 -ge val2	*val1* é maior ou igual a *val2*
val1 -lt val2	*val1* é menor que *val2*
val1 -le val2	*val1* é menor ou igual a *val2*

Outros operadores usados pelo comando *test*:

OPERADOR	DESCRIÇÃO
!	nega uma expressão
-a	faz o **E** lógico de duas expressões
-o	faz o **OU** lógico de duas expressões
(...)	agrupa expressões para avaliação da precedência dos operadores

15.6. Retorno de scripts

É importante que os novos scripts, ao serem construidos, devam seguir a filosofia do UNIX com relação ao código de retorno para indicar sucesso. No UNIX o código de retorno ou STATUS do processo filho tem o seguinte significado:

STATUS	Significado
0	Processo terminou com sucesso
diferente de 0	Processo **não** terminou com sucesso

O programador deve sempre usar o comando *exit* de forma disciplinada, pois um script shell pode ser usado em um comando *if*. Por exemplo, o programa shell *script* abaixo:

```
$ cat script

     var=10
     if test  $var -gt 2
     then
          exit 1
     fi
      ...
     exit 0
```

pode ser usado junto com o comando *if*:

```
$ if script
> then
>    ...
> fi
```

porque usa *exit* de forma disciplinada, ou seja, ao terminar usa *exit 0* e na ocorrência de erro usa *exit 1*.

15.7. O comando &&

No UNIX há uma preocupação constante em aumentar a produtividade. O comando *&&* é uma forma resumida do comando *if*. O exemplo abaixo demonstra isto:

```
if [ $# -ne 2 ]
then
     exit 1
fi
 ...
```

para testar se o número de argumentos passados ao script acima é diferente de 2. O mesmo exemplo acima pode ser re-escrito da seguinte forma:

```
[ $# -ne 2 ] && exit 1
```

De uma forma geral, o operador *&&* pode ser usado com vários comandos:

```
com1 && com2 && com3
```

A ordem de execução dos comandos acima é a seguinte: o shell executa *com1* e verifica seu STATUS de término. Se *com1* terminou com sucesso, o shell executa *com2* e verifica seu STATUS. Se *com2* terminou com sucesso, o shell executa *com3*, e assim sucessivamente.

15.8. O comando :

O comando *:* (dois pontos) é um comando nulo que não faz nada. Você pode perguntar: de que vai me servir um comando que não faz nada?

Responderemos à pergunta acima através do exemplo da seção seguinte.

15.9. O comando || (*OU* exclusivo)

O comando || é uma forma resumida do comando *if*. Vamos usar o comando nulo *:* para explicar o comando ||. Por exemplo, queremos dar uma mensagem apenas se o usuário não estiver cadastrado no sistema:

```
if grep usuário /etc/passwd > /dev/null
then
     :
else
     echo mensagem
fi
```

o mesmo comando acima poderia ser re-escrito em função de || como mostra a linha de comandos abaixo:

```
$ grep usuário /etc/passwd > /dev/null || echo mensagem
```

Note que no exemplo acima, a saída padrão do comando *grep* foi redirecionada para o lixo do UNIX (/dev/null), pois só o que nos interessa é seu STATUS (sucesso ou não) de término.

Na versão mais extensa do *if*, se o *grep* terminar com sucesso o comando nulo *:* é executado. Se o comando *grep* não conseguir encontrar o usuário o *echo* deve ser executado.

De uma forma geral, o operador || pode ser usado com vários comandos:

```
com1 || com2 || com3
```

A ordem de execução dos comandos acima é a seguinte: o shell executa *com1* e verifica seu STATUS de término. Se *com1* terminou sem sucesso, o shell executa *com2* e verifica seu STATUS. Se *com2* terminou sem sucesso, o shell executa *com3*.

15.10. Uso de *$var* versus *"$var"*

O programador shell deve tomar cuidado quando usa variáveis em expressões no comando *test*. Principalmente se o valor da variável *var*, por exemplo, pode assumir o valor nulo ("").

Vamos mostrar um exemplo, aparentemente correto, mas com erro de sintaxe detectado pelo shell:

```
$ var=""

$ if [ $var = alvaro ]
> then
>       echo "nome é igual"
> fi

     test: argument expected
```

Note que no instante que o comando acima foi completamente digitado o shell fez a macro substituição da variável *$var* para nada (cadeia nula) ficando com a seguinte expressão:

```
$ if [  = alvaro ]
```

Na hora de executar o comando expandido, faltou um argumento do lado esquerdo do caractere (=) e o *test* imprimiu a mensagem de erro *test:argument expected*.

Para solucionar o problema acima é só colocar a variável *$var* entre aspas:

```
$ if [ "$var" = alvaro ]
> then
>         echo "nome é igual"
> fi
```

Com a variável entre aspas o shell sabe que, mesmo se o conteúdo da variável *var* for nulo, tem um argumento ("") do lado esquerdo do caractere (=).

15.11. Outras formas de *if*

Neste ponto mostraremos outras formas de *if*'s envolvendo a manipulação de variáveis.

Mais uma vez voltamos a argumentação de que o UNIX é um ambiente produtivo. Mostraremos algumas formas disfarçadas de *if*'s. Segundo [KERN 84] podemos fazer atribuição de variáveis com testes embutidos na operação. Veja a tabela a seguir com a sintaxe do comando e a descrição do seu funcionamento:

x=${var - valor}	*x* recebe o valor de *var* se esta variável estiver definida, caso contrário, *x* recebe *valor*. O conteúdo de *var* não muda;
x=${var = valor}	*x* recebe o valor de *var* se esta variável estiver definida, caso contrário, *x* recebe *valor*. Se a variável *var* estiver indefinida seu conteúdo recebe *valor*;
x=${var ? msg}	*x* recebe o valor de *var* se esta variável estiver definida, caso contrário, o shell imprime a mensagem *msg* e encerra o script. Se não existir *msg* o shell imprime *var: parameter not set*.
x=${var + valor}	*x* recebe *valor* se a variável *var* estiver definida.

15.12. O comando *shift*

Aqui mostraremos como manipular a janela de parâmetros posicionais.

Os parâmetros posicionais são acessados através de uma janela, como mostra a Figura 15.3. Estamos supondo que um comando foi digitado passando dez parâmetros:

```
$ comando P1 P2 P3 P4 P5 P6 P7 P8 P9 P10
```

Onde o os parâmetros posicionais *$1*, *$2*, ..., *$9* correspondem, respectivamente, aos argumentos *P1*, *P2*, ..., *P9*. Note que o

argumento *P10* não pode ser acessado via parâmetro posicional. Não existe o parâmetro posicional *$10*.

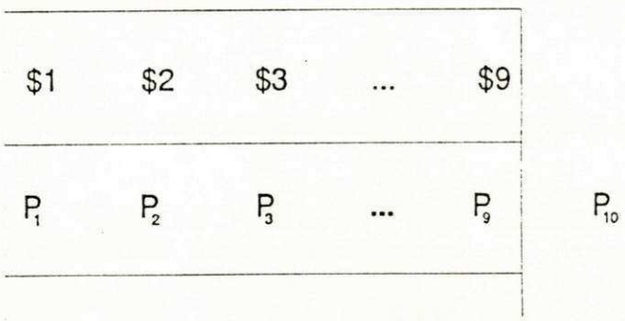


Figura 15.3: Janela de rolamento do comando *shift*

O jeito de acessar o décimo parâmetro é fazendo com que a janela se desloque uma posição a direita. O comando para deslocar a janela de parâmetros posicionais é *shift*:

$ shift

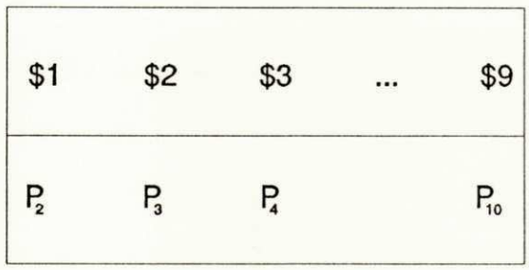


Figura 15.4: Janela de rolamento do comando *shift*

Na figura acima há um deslocamento de todos os parâmetros posicionais: *$2* passa a ser *$1*, *$3* passa a ser *$2*, *$4* passa a ser *$3* e assim por diante. O valor de *$1* é perdido. A rotação da janela acima é ireversível, ou seja, não há como fazer um *shift* para esquerda recuperando o parâmetro *P1*.

15.13. O comando *case*

O comando *case* tem duas funções principais. É um *if* com múltiplas escolha e também uma ferramenta poderosa para **casamento de padrões.**

A sintaxe do comando *case* é a seguinte:

```
case palavra in
     padrão) comandos;;
     padrão) comandos;;
          ...
esac
```

Onde o texto em destaque representa os elementos componentes do comando *case*.

O comando *case* compara *palavra* com *padrão* de cima para baixo, e executa os *comandos* associados como o primeiro *padrão* que casou com *palavra*. Os padrões são escritos usando as regras de expansão de nomes do shell (*,?,[]). Cada ação é terminada por um par de ponto-e-virgula (;;).

Vejamos um *script* onde queremos verificar se o usuário digitou uma das três opções: *-l*, *-f* ou *-i* em uma linha de comando. É uma solução tipica para fazer casamento de padrões. Vamos analisar uma solução inicial usando o comando *if*. Na solução final usaremos o comando *case*:

15.13.1 Versão usando *if*

```
if [ $1 = -l ]
then
   OPC_L=ok
elif [ $1 = -f ]
then
   OPC_F=ok
elif [ $1 = -i ]
then
   OPC_I=ok
fi
```

No script acima, *$1* é o parâmetro posicional que foi passado na linha de comandos. A ação desejada é atribuir o valor *ok* às variáveis *OPC_L*, *OPC_F* ou *OPC_I* dependendo, respectivamente, se o valor digitado *$1* é igual a *-l*, *-f* ou *-i*.

15.13.2 Versão usando *case*

```
case $OPC in
   -l) OPC_L=ok;;
   -f) OPC_F=ok;;
   -i) OPC_I=ok;;
esac
```

Observe que a versão usando o comando *case* é mais legível e compacta. Apesar de poder substituir *if*'s, casamento de padrões é a caracteristica mais forte do comando *case*. Vamos demonstrar este recurso do *case* através de exemplo: queremos criar um script

nome que descubra o tipo de arquivo que foi passado como parâmetro:

```
$ cat nome

     case $1 in
          *.c) echo arquivo fonte na linguagem C;;
          *.o) echo arquivo objeto;;
          /[a-z].*) echo nome arquivo absoluto;;
          -[0-9]+)  echo é uma opção numérica;;
          "")       echo faltou nome;;
     esac
```

No exemplo acima *$1* representa o nome que foi digitado na linha de comandos. O conteúdo de *$1* é comparado com o padrão **.c* pelo comando *case*. Se houver casamento o comando *echo* é executado e o *case* termina. Se não houver casamento, *$1* é comparado com o segundo padrão **.o*, e assim por diante. A possibilidade de escrever expressões como **.c* para fazer casamento com todos os nomes terminados em *.c* e *.o* ou /[a-z]*, para casar padrões com nomes de arquivos começando com / seguido de uma letra minuscula, torna o comando *case* muito potente.

A execução do script *nome* recebendo *teste.c* e *teste.o* como argumentos emitiria o seguinte relatório:

```
$ nome teste.c

     arquivo fonte na linguagem C


$ nome teste.o

     arquivo objeto


$ nome /usr/alvaro/nome

     nome arquivo absoluto
```

PARTE III: PROGRAMAÇÃO DO SHELL

16. LAÇOS

Neste capitulo queremos apresentar os comandos do shell *for*, *while*, *until* e *break* para controle de fluxo de execução. Veremos também algumas dicas úteis para estruturar programas escritos na linguagem shell.

16.1. Controle de Fluxo

Os laços existem para controlar o fluxo de execução dos comandos. O shell fornece várias construções sintáticas para laços de controle de fluxo de execução de comandos, tais como: *for*, *while* e *until*. A seguir vamos demonstrar cada um deles através de exemplos.

16.2. O Laço *for*

```
for var in lista de argumentos
do
     comandos
done
```

O comando *for* é composto das palavras reservadas *for*, *in*, *do* e *done*. Observe o exemplo de uso do comando *for* abaixo:

```
$ for var in alvaro jacques anna
> do
>    echo nome=$var
> done
```

O comando acima tem como saída o seguinte relatório:

```
nome=alvaro
nome=jacques
nome=anna
```

O funcionamento do comando *for* será descrito a seguir: inicialmente, a variável *var* assume o valor *alvaro*. O comando *echo* entre as palavras *do* e *done* é executado e um ciclo do laço é completado. No segundo ciclo a variável *var* assume o valor *jacques* e o comando *echo* é novamente executado e, finalmente no terceiro ciclo *var* assume o valor *anna* e o comando *echo* é mais uma vez executado.

Exemplo 1 (*for*) - Cria arquivos vazios

Vamos mostrar um script *cria_arq* que cria arquivos vazios passados como parâmetros:

```
$ cat cria_arq

     for i in $*
     do
          echo > $i
     done
```

O script *cria_arq* acima usa a variável *$** que representa todos os parâmetros (*$1 $2 $3 ...*) passados ao script. A variável *i* vai assumir os valores *$1, $2, ...* que forem passados como argumentos.

Para criar os arquivos *arq1, arq2* e *cadastro* poderíamos executar o script *cria_arq* da seguinte forma:

```
$ cria_arq arq1 arq2 cadastro
```

Para verificar se os arquivos foram realmente criados, podemos listar o diretório corrente com o comando seguinte:

```
$ ls -C

     arq1      arq2      cadastro
```

Exemplo 2 (for) - Processa arquivos em ordem cronológica

```
$ for i in `ls -lt`
> do
>          ...
> done
```

No exemplo acima o comando *ls -lt* é executado tendo como saída os nomes de arquivos do diretório corrente em ordem cronológica de criação. A construção `...` (crase) do shell é substituida pela saída do comando *ls*. Resumindo: a variável *i* assume como valor a cada ciclo os nomes de arquivos ordenados cronologicamente pelo comando *ls -lt*.

Exemplo 3 (*for*) - Processa sub-árvore de arquivos modificados nos últimos dois dias

```
$ for i in `find . -mtime -2 -print`
> do
>          ...
> done
```

Aqui vale a mesma argumentação anterior sobre o crases `...`. A saída do comando *find* será usada pela variável *i* a cada ciclo. O

comando *find* pesquisa, a partir do diretório corrente (.), todos os arquivos que foram modificados nos últimos dois dias (*-mtime -2*) e os imprime (*-print*).

16.3. O laço *while*

```
while condição
do
     comandos
done
```

O comando *while* é composto das palavras reservadas *while*, *do* e *done*. Enquanto *condição* for executada com sucesso, os comandos entre as palavras *do* e *done* são executados.

Observe os exemplos de uso do comando *while* a seguir.

Exemplo 4 (*while*) - Processa todos os argumentos passados como parâmetros

$ cat com

```
while [ -z "$1" ]
do
     echo parametro1=$1
     shift
done
```

O comando *while* será executado enquanto a condição *[-z "$1"]* for verdadeira. Lembre-se que o flag *-z* quer dizer: enquanto existir o conteúdo da variável posicional *$1*. O comando *shift* faz com que haja um deslocamento dos parâmetros posicionais já processados, ou seja, *$1* vai receber os argumentos da linha de comandos, um a cada ciclo do *while*. Veja um exemplo de ativação do script *com*:

$ com alvaro jacques ismenia

O comando acima tem como saída no vídeo o seguinte relatório:

```
parametro1=alvaro
parametro1=jacques
parametro1=ismenia
```

Exemplo 5 (*while*) - Monitora a permanência de um usuário no ar

O script *avisa_saída* monitora a permanência de um usuário no sistema.

```
$ cat avisa_saída

     NO_AR=nao
     while `who | grep $1 >/dev/null`
     do
          NO_AR=sim
          sleep 120       # espera dois minutos
     done
     if [ $NO_AR = sim ]
     then
          echo "Usuário $1 encerrou sua sessão"
     else
          echo "Usuário $1 nao esta usando a maquina"
     fi
```

A execução do script *avisa_saída* para monitorar *alvaro* é ativado através do seguinte comando:

```
$ avisa_saída alvaro
```

O comando acima imprimirá a mensagem *Usuário alvaro encerrou sua sessão*, logo após *alvaro* encerrar sua sessão, ou a mensagem *Usuário alvaro nao esta usando a maquina*, se *alvaro* estiver fora do sistema. O comando *sleep* faz com que o processo durma 120 segundos.

O uso de # no script acima indica linha de comentário. O texto após # não é interpretado pelo shell serve apenas de comentário para o usuário.

16.4. O laço *until*

```
until condição
do
     comandos
done
```

O comando *until* é composto das palavras reservadas *until*, *do* e *done*. Até que *condição* seja verdadeira, os comandos entre as palavras *do* e *done* são executados. Em outras palavras, enquanto *condição* for falsa, execute *comandos*.

Observe o exemplo de uso do comando *until* a seguir:

Exemplo 6 (*until*) - Monitora a entrada de um usuário no ar:

```
$ cat avisa_entrada

     until `who | grep $1 >/dev/null`
     do
          sleep 60       # espera um minuto
     done
     echo "Usuário $1 esta usando a maquina"
```

A execução de *avisa_entrada* para saber quando *alvaro* entrar no ar:

```
$ avisa_entrada alvaro
```

A mensagem *Usuário alvaro esta usando a maquina* será impressa no vídeo tão logo o usuário *alvaro* entre no ar.

16.5. Quem executa os laços?

O próprio shell corrente se o laço não for redirecionado, ou um sub-shell se houver redirecionamento da entrada ou saída padrão do laço.

Vamos usar como exemplo, para demonstrar redirecionamento de um laço, uma variável que é alterada dentro de um laço. Na versão sem redirecionamento do laço, a variável permanece com último valor depois do laço. Por exemplo, o laço seguinte incrementa uma variável começando de 1 até 4:

```
$ var=1
$ while [ $var -lt 4 ]
> do
>     echo var=$var
>     var=`expr $var + 1`
> done
```

O laço acima imprime no video as seguintes linhas:

```
var=1
var=2
var=3
```

Para nos certificarmos que o laço foi executado pelo shell corrente basta imprimir o valor da variável *var* (deve ser 4):

```
$ echo $var

     4
```

No exemplo acima usamos um novo comando para *somar*: *expr*. Como as variáveis no shell são sempre do tipo cadeia de caracteres as operações aritméticas são efetuadas usando crases, a saída padrão e o comando expr da seguinte forma:

```
num=`expr $mem = 1`
```

Onde *expr* recebe os parâmetros através da linha de comandos, faz a operação imprimindo o resultado na saída padrão.

Se o mesmo exemplo anterior for redirecionado para um arquivo, quem executará o laço será um sub-shell:

```
$ var=1
$ while [ $var -lt 4 ]
> do
>     echo var=$var
>     var=`expr $var + 1`
> done > arq
```

O arquivo **arq** terá o seguinte conteúdo:

```
var=1
var=2
var=3
```

Para termos certeza que o laço **não** foi executado pelo shell corrente basta imprimir o valor da variável *var*:

```
$ echo $var

    1
```

O fato de o valor de *var* ser igual a *1* significa que o laço foi executado por um sub-shell. A variável *var* foi alterada em um processo filho. O valor original *1* da variável *var*, no processo pai, não foi afetado.

16.6. O comando *exec*

O comando *exec* é um dos comandos embutidos do shell. Ele pode ser usado com dois objetivos:

* carregar um programa do disco sobrepondo o processo corrente

* manipular a entrada padrão e a saída padrão do processo corrente

O processo normal de execução de comandos no UNIX é mostrado na figura seguinte:

$ prog

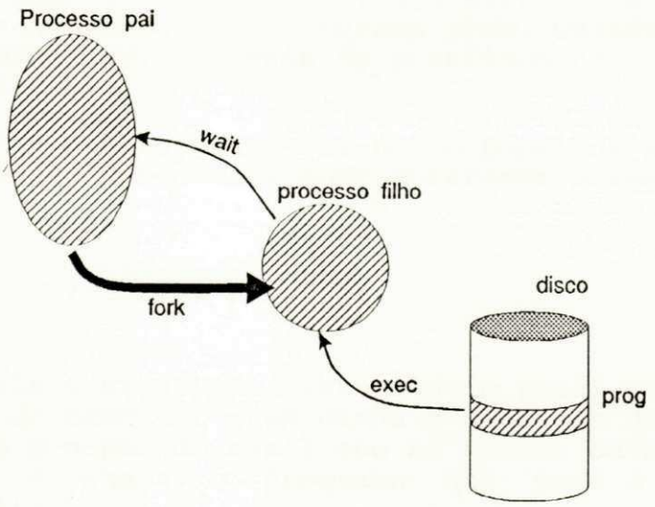


Figura 16.1: Mecanismo de execução de comandos no UNIX

Onde o shell cria um filho (fork) e aguarda (wait) seu término. O processo filho chama um serviço do núcleo do UNIX (exec) para fazer a carga do programa *prog* do disco para a memória ocupada pelo processo filho.

O comando *exec* do shell pode ser usado para carregar o programa *prog* sobre o shell:

$ exec prog

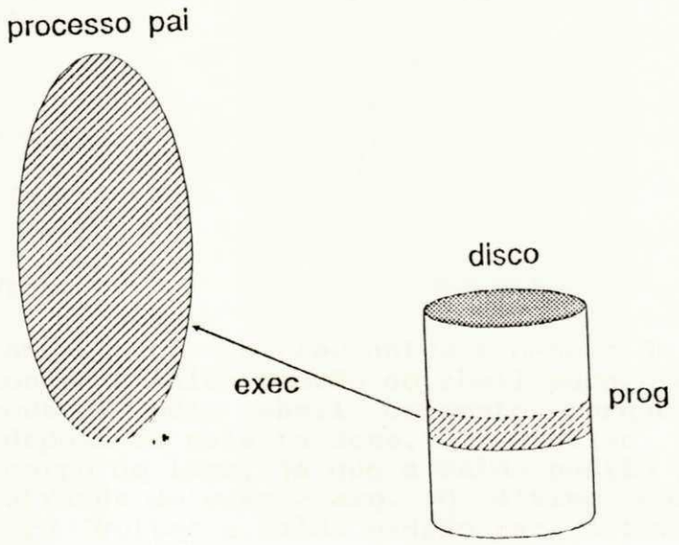


Figura 16.2: Execução com sobreposição do processo original

Note que a figura acima não tem (fork+wait). A área de memória do shell foi sobreposta pelo programa *prog*. Quando o programa *prog* terminar de executar, a cadeia de prontidão não será mais "*$* " e sim "*Login:*".

Suponha o comando *date* fazendo o papel de *prog*. Experimente digitar o comando *exec date*, como mostramos a seguir:

```
$ exec date

    Wed Aug  7 10:20:09 PDT 1991
 n:
```

O comando *date* é executado sobrepondo o shell corrente. Quando o *date* termina de executar quem ganha o controle de execução da UCP é o *init* (que é o pai do shell que se tornou *date*). O *init* cria outro filho e chama o programa que pede a identificação do usuário (*getty*).

O comando *exec* pode ser usado também com a finalidade de redirecionar os arquivos padrões do shell corrente. Por exemplo, se quisermos redirecionar o laço:

```
$ var=1
$ while [ $var -lt 4 ]
> do
>     echo var=$var
>     var=`expr $var + 1`
> done > arq
```

para o arquivo *arq* sem que o laço seja processado por um sub-shell, usaremos o comando *exec* para enganar o shell corrente da seguinte forma:

```
$ var=1
$ exec > arq
$ while [ $var -lt 4 ]
> do
>     echo var=$var
>     var=`expr $var + 1`
> done
$ exec > /dev/tty
```

Observe que usamos o comando *exec* antes e depois do laço. Antes para redirecionar a saída padrão do shell para o arquivo *arq*. O laço foi executado pelo shell corrente porque não tinha o caractere (>) depois da palavra *done*. O arquivo *arq* contém o resultado do corpo do laço, já que a saída padrão foi redirecionada para *arq* através de *exec > arq*. O último comando *exec > /dev/tty* é para voltar a saída padrão para o terminal, que é o seu valor default.

16.7. Programação baseada em tabelas

Aqui queremos mostrar que é possível construir programas estruturados na linguagem shell.

Os comandos *read* e *while* podem ser usados na Linguagem shell para construir programas baseados em tabelas. Programação baseada em tabelas consiste em dividir um problema em duas partes:

* estrutura de dados (tabelas)
* código shell

Uma vez que as tabelas foram escolhidas, podemos escrever o código para processar estas tabelas usado os comandos *read* e *while*.

Programas baseados em tabelas são mais legíveis e de fácil manutenção, como demonstraremos a seguir.

Vamos supor um sistema de backup em um microcomputador que precisa oferecer vários dispositivos como opções.

Por exemplo, a tabela *tab* abaixo representa os dispositivos nesta instalação fictícia:

```
DISQUETE  DD   360  5(1/4) /dev/df048ds9
DISQUETE  HD  1200  5(1/4) /dev/df096ds15
DISQUETE  DD   720  3(1/2) /dev/df136ds9
DISQUETE  HD  1440  3(1/2) /dev/df136ds18
```

para processar estas informações em um script *prog*, por exemplo, teríamos o seguinte código:

```
$ cat prog

  ...
exec < tab
while read NOME DENS CAP MOD DEV
do
     echo "NOME=$NOME DENSIDADE=$DENS"
done
  ...
```

Para incluir, alterar ou excluir qualquer dispositivo é só alterar a tabela. O trecho de código que manipula a tabela permanece inalterado. É o que chamamos programação baseada em tabela. Outro fator positivo neste tipo de programação é que éfácil de depurar.

16.8 "$*" versus "$@"

Vimos anteriormente que os argumentos são passados aos scripts através dos parâmetros posicionais *$1 $2 $3 ...*

A variável especial do shell *$** representa todos os parâme-

tros *$1 $2 $3 ...* juntos. Há outra variável do shell *$@* que também representa a união dos parâmetros *$1 $2 $3 ...*

Qual a diferença entre *$** e *$@*?

Segundo [KERN 84], a diferença só é notada quando precisamos interpretar os argumentos posicionais exatamente do mesmo jeito que eles foram digitados pelo usuário.

Vamos fazer uma analogia entre as variáveis *$** e *$@*:

"$" <- "$1 $2 $3 ..."*

"$@" <- "$1" "$2" "$3" ...

Resumindo, usamos "$@" quando queremos processar cada argumento da linha de comandos exatamente do mesmo jeito que ele foi digitado.

16.9. O comando *break*

O comando *break* sai do corpo do laço mais interno. Por exemplo:

```
$ cat prog

     ...
     for i in lista-de-variáveis
     do
          ...
          if [ $i = fim ]
          then
               break
          fi
          ...
     done
      ...
```

O exemplo acima usa *break* para sair do corpo do laço antes de seu término normal.

Podemos sair de mais de um laço com um só comando *break*. A forma *break n* é usada para sair de *n* níveis dos laços mais internos:

```
for i in lista1
do
     while condição-2
     do
          until condição-3
          do
               for j in lista-4
               do
                    ...
                    if condição-saída
                    then
                         break 3
                    fi
                    ... (comandos laco-4)
               done
               ... (comandos laço-3)
          done
          ... (comandos laço-2)
     done
     ... (comandos for-1)
done
```

Se *condição-saída* for verdadeira, o comando *break 3* será executado provocando a saída dos três corpos dos laços mais internos. Depois do *break* o comando a ser processado é *comandos for-1*.

PARTE III: PROGRAMAÇÃO DO SHELL

17. SUBPROGRAMAS OU SUBROTINAS

Neste capitulo mostraremos quais os recursos que a linguagem shell oferece para juntar ou agrupar comandos. Veremos funções e scripts, suas diferenças e igualdades. Veremos ainda como ter funções já definidas no inicio de nossa sessão no terminal.

17.1. Empacotamento

Toda Linguagem de Programação precisa de um mecanismo para empacotar seus comandos. O shell implementa subprogramas via script como vimos anteriormente ou através de funções que serão discutidas neste capitulo. Já falamos também que há uma ligação entre pacotes. Eles trocam informações entre si. Pacotes recebem informações (parâmetros) e devolvem informações (status de término).

A seguir vamos discutir as duas formas de empacotamento na linguagem shell: scripts e funções.

17.2. Scripts

Uma das formas de empacotamento de comandos no shell é através de *scripts*, que consiste simplesmente em criar um arquivo texto contendo comandos do shell. Os scripts gozam dos mesmos privilégios dos programas executáveis puros sendo executados exatamente da mesma forma dos demais comandos do UNIX.

A forma de comunicação de um script com seu pai é através do comando interno do shell *exit n*, onde *n* é o STATUS de término.

17.3. Funções

Outra forma de empacotar comandos no UNIX é através de um recurso do shell chamado funções. Funções do shell podem ser vistas como apelido para um conjunto de comandos.

Para criar uma função, digitamos seu *nome* seguido de *()* e depois teclamos os comandos que vão compor seu corpo. Os comandos do corpo da função devem estar entre *{}* (chaves). Por exemplo, para criar a função *dir* que mostra o diretório fazemos o seguinte:

```
$ dir()
> {
>         ls -l
> }
```

Neste instante, a função *dir* já existe no ambiente corrente do shell. Para executa-la é só usar seu nome:

```
$ dir

 -rw-rw-rw-   1 root     dsvmto      33879 Jul 30 22:13 cap01.doc
 -rw-rw-rw-   1 root     dsvmto       6790 Jul 31 08:10 cap02.doc
 -rw-rw-rw-   1 root     dsvmto      28756 Jul 30 23:12 cap03.doc
                                ...
```

Podemos terminar uma função informando qual foi o seu estado

de término (STATUS) através do comando *return n*, onde *n* é o código de retorno da função.

A visualização das funções já definidas é feitas por intermédio do comando *set*:

```
$ set

     ...
    dir()
    {
    ls -l
    }
     ...
```

As reticências "..." indicam que as funções são mostradas junto com as variáveis do ambiente.

Funções são locais ao processo, o que significa que os processos filhos não herdam as funções que foram definidas no processo pai.

Vejam a seguir as principais diferenças entre scripts e funções.

17.4. Diferenças entre scripts e funções

1) As funções são sempre executadas pelo shell corrente. Os scripts são executados por um shell filho.

2) A forma de criação de scripts e funções é diferente, como exemplificado acima

3) O retorno do status de execução: no script deve ser *exit n* e na função deve ser *return n*. O que acontece se em uma função usarmos *exit n* para sair?

   Por exemplo, a função *dir* é definida de forma errada:

```
$ dir()
> {
>     if ls -l
>     then
>          exit 0
>     fi
>     exit 1
> }
```

Ao executar a função acima *dir*, o programador verá na tela a saída do comando *ls -l* e a mensagem de *login*:

```
$ cat .func

  ...
dir()
{
    if ls -l
    then
         return 0
    fi
    return 1
}
type()
{
     if /bin/cat $*
     then
          return 0
     fi
return 1
}
  ...
```

O arquivo *.func* deve conter todas as definições de funções que você quer ter definidas logo após o login.

O passo seguinte é editar o arquivo de personalização do usuário *.profile* e colocar lá dentro a seguinte linha de comando:

```
$ .  .func                    (Note o ponto no início da linha)
```

(isso mesmo que você está vendo, um ponto seguido de um espaço e depois o nome do arquivo *.func*)

O ponto no início da linha vai instruir o shell para não criar um processo filho para executar o arquivo *.func* definindo as funções no ambiente corrente do próprio shell do seu terminal.

Outra forma de termos funções definidas logo após o shell de login é usar o próprio arquivo de personalização do usuário *.profile*, ou o arquivo de personalização do sistema */etc/profile*. Para colocar a definição das funções desejadas nós particularmente não gostamos desta solução porque a instalação de novas versões do sistema pode sobrepor estes arquivos, o que implica em ter que repetir todo o trabalho manual para digitar novamente a definição das funções, o que não seria nada produtivo. Você não concorda?

17.6. O que existe de igual entre funções e scripts?

1. A passagem de parâmetros é feita da mesma forma.

2. A forma de ser ativada (executada)

17.7. Necessidade de funções

As funções são necessárias em situações que precisamos manipular o ambiente do shell corrente: um exemplo é a alteração do diretório corrente e das variáveis.

Vejamos o exemplo da função que faz uma pergunta e retorna *0* se a resposta for igual a *sim*, retorna *1* se a resposta for *não*:

```
pergunta()
{
     while true
     do
          echo "$* \c" > /dev/tty
          read RESP    < /dev/tty

          case $RESP in
               [sS]*)    return 0;;      # sim
               [nN]*)    return 1;;      # nao
          esac
     done
}
```

true é uma expressão que sempre retorna status verdadeiro. */dev/tty* é o nome de um arquivo especial que representa o terminal corrente. A função acima fica em "loop" (laço infinito *while true*): mostra a mensagem que foi passada como parâmetro ($*) no terminal (/dev/tty) e lê a resposta (read RESP) também do terminal. A resposta é cosiderada correta se a primeira letra for um *s* ou um *n*, retornando o valor *0* ou *1* para quem chamou a função *pergunta*.

A função *pergunta* pode ser testada de forma interativa:

```
$ if pergunta "Responda Sim ou Nao"
> then
>     echo respondeu Sim
> else
>     echo respondeu Nao
> fi
```

Logo após o comando *if* ter sido completamente digitado a função *pergunta* começa a executar imprimindo a seguinte mensagem na tela:

Responda Sim ou Nao?

Vamos supor que digitamos *s*, a função *procura* iria retornar *0* e o comando *if* executaria o *echo* imprimindo a mensagem abaixo:

respondeu Sim

PARTE III: PROGRAMAÇÃO DO SHELL

18. TRATAMENTO DE SINAIS

Neste capitulo queremos mostrar como manipular os eventos ou sinais que possam afetar os processos.

Como é gerado um sinal no UNIX?

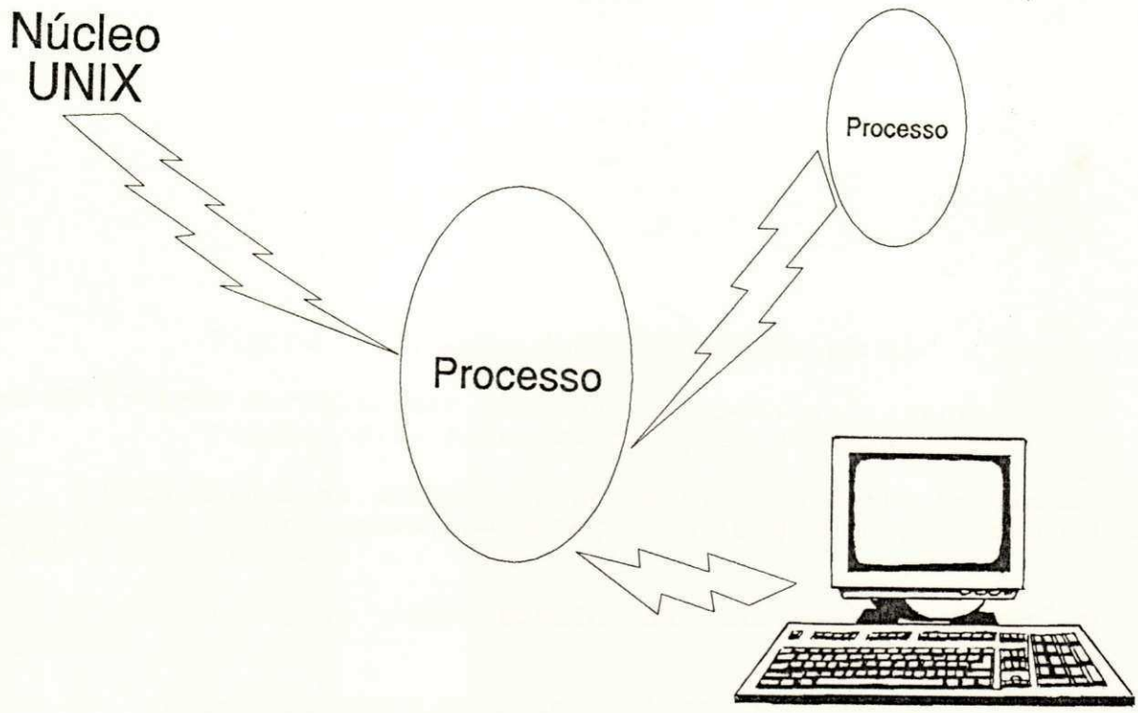


Figura 18.1: Geração de sinal no UNIX

Um sinal pode ser gerado de três formas:

* pelo núcleo do UNIX
* pelo teclado
* por outro processo

O **núcleo** do UNIX gera um sinal, por exemplo, quando o terminal é desligado pelo usuário ou quando há uma chamada às rotinas do núcleo com parâmetros errados, ou quando um processo grava em um duto sem consumidor, ou ainda quando há erro de barramento, etc.

Um sinal é gerado pelo **teclado** quando o usuário aperta algumas teclas prédefinidas, tais como *break* ou *quit*. (tecla *Del* ou *Ctrl* e \).

Outra forma alternativa de geração de sinal é através do comando *kill* ou da rotina do núcleo do UNIX de mesmo nome *kill*.

Assim como existe uma tabela de descritores de arquivos por processo, existe também uma tabela de sinais por processo. A tabela de sinais é composta de um **número que identifica cada**

sinal e uma ação que deve ser tomada na ocorrência do sinal:

sinal	ação
1	morte
2	morte
...	...

Figura 18.2: Ação default para um sinal

ura 18.2: Ação default para um sinal.....#
Figura 18.2: Ação default para um sinal

A ação normal na ocorrência de um sinal é **morte imediata** do processo. Em alguns casos, há geração da imagem do processo em um arquivo chamado *core*.

O significado de cada número na tabela de sinais é o seguinte:

1: Significa que o usuário desligou a chave de alimentação elétrica do terminal. Este sinal é conhecido como *SIGHUP*;
2: Este sinal é gerado quando o usuário digitar a tecla de interrupção *Break* ou *Del*. Um sinônimo para este sinal é *SIGINT*;
3: O usuário apertou as teclas *Ctrl* + \ (tecla de controle e contra barra) simultaneamente. Outro nome para este sinal é *SIGQUIT*. Este sinal gera *core*;
4: Sinal *SIGILL* gerado quando o núcleo executa uma instrução ilegal. Este sinal gera *core*;
5: Sinal *SIGTRAP* gerado por uma rotina do núcleo do UNIX para fazer depuração da execução de processos. Este sinal gera *core*;
6: Sinal *SIGIOT* usado nas primeiras versões do UNIX para manipular a entrada e saída;
7: Sinal *SIGEMT*;
8: Sinal *SIGFPE* gerado pelo núcleo quando ocorre exceção de ponto flutuante. Este sinal gera *core*;
9: Sinal *SIGKILL*. Processos morrem ao receber este sinal. Não há como evitar a morte;
10: *SIGBUS* - Erro de barramento. Este sinal gera *core*;
11: *SIGSEGV* - Violação de endereço. Este sinal gera *core*;
12: *SIGSYS* - Erro na chamada de uma rotina do núcleo. Este sinal gera *core*;
13: *SIGPPE* - Escrever em duto sem consumidor;
14: *SIGALARM* usado para acordar processos no tempo futuro;

15: *SIGTERM* - Sinal para término do programa por software;
16: *SIGUSR1* - Sinal livre para ser definido pelo usuário;
17: *SIGUSR2* - Sinal livre para ser definido pelo usuário;
18: *SIGCLD* - Sinal que indica morte de um processo filho;
19: *SIGPWR* - Sinal que indica falta de energia (usado em sistemas que possuem baterias).

18.2. Como capturar sinais?

A ação default para um determinado sinal pode ser alterada através do comando *trap*.

18.2.1. O comando *trap*

A sintaxe do comando *trap* é a seguinte:

trap comandos sinais

Vejamos um exemplo:

trap "rm arq_temp; exit 1" 1 2

o comando *trap* acima vai remover o arquivo temporário *arq_temp* e encerrar (*exit*) o script na ocorrência do sinal *1* (usuário desliga o terminal) ou *2* (usuário tecla *Break*) ou *Del*). No exemplo acima os sinais *SIGHUP* e *SIGINT* foram capturados e tratados.

O shell aproveitou o esquema de tratamento de sinais do UNIX e criou um novo sinal para indicar *fim de script*: o sinal *0*. O núcleo do UNIX não conhece este sinal. Este sinal é uma implementação útil a nivel de shell.

O comando *trap* é usado normalmente no inicio de cada script, mas pode ser usado interativamente sem nenhum problema. O comando *trap* sem argumentos mostra os sinais que estão sendo capturados e suas respectivas ações.

18.2.2. Um exemplo de uso do comando *trap*

Vejamos um exemplo de um script completo que use *trap*:

```
$ cat menu

     trap 'echo Interrupção; date' 2 3
     while true
     do
          echo "1 - incluir"
          echo "2 - alterar"
          echo "3 - excluir"
          echo "0 - finalizar"
          echo "\c"
          read RESP
          case $RESP in
               1|2|3)    echo "opção $RESP ok";;
               0)        exit 0;;
          esac
     done
```

O comando menu acima usa *'comandos'* como argumento para *trap*. Ao executarmos o comando *menu* temos as seguintes informações na tela:

```
1 - incluir
2 - alterar
3 - excluir
0 - finalizar
```

Se teclar *Del* ou *Ctrl* + \ (as teclas de controle e contra-barra apertadas simultaneamente) a seguinte mensagem aparecerá no vídeo:

```
Interrupção
Thu Aug  8 10:17:41 PDT 1991
 ...
```

Se esperarmos um pouco e teclarmos de novo *Del* a mesagem será a seguinte:

```
Interrupção
Thu Aug  8 10:17:50 PDT 1991
 ...
```

O fato de o argumento do comando *trap* estar entre apostrófes *'...: date'* implica em toda vez que ocorrer o sinal *2* ou *3* o comando *date* será executado, mostrando uma nova data a cada interrupção.

18.3. Como ignorar os sinais?

Para ignorar sinais devemos usar o comando *trap* da seguinte forma:

```
trap '' sinais a ignorar
```

Por exemplo, o comando *trap* abaixo faz com que os sinais

SIGHUP, SIGINT e SIGQUIT sejam ignorados:

```
trap '' 1 2 3
```

18.4. Como voltar ao tratamento default de sinais?

[SWAR 90] nos mostra como voltarmos ao tratamento default (padrão) para um determinado sinal. Simplesmente usamos o comando *trap* apenas especificando os sinais:

```
trap 1 2 3
```

O comando *trap* acima faz com que o shell volte a ação default (morte) para o processo corrente no caso da ocorrência de SIGHUP (1), SIGINT(2) ou SIGQUIT (3).

18.5. '...' ou "..."?

Quando usar *'comandos'* ou *"comandos"*, como argumento para *trap*?

[SWAR 90] é quem levanta esta questão. O shell interpreta todo o script antes de executá-lo. Se *comandos* depender do momento (data e hora) da execução, então *"comandos"* terá um resultado diferente de *'comandos'*. A seguir vamos ver dois exemplos que ilustram o uso de *'...'* e *"..."*.

18.6. Exemplo de uso de "..." no comando *trap*

Vamos modificar o exemplo anterior, *menu*. A nova versão do script *menu1* é a seguinte:

```
$ cat menu1

     trap "echo removendo arquivo {$TMP}; rm $TMP; exit 1" 2 3
     TMP=/tmp/menu$$
     echo > $TMP
     while true
     do
          echo "1 - incluir"
          echo "2 - alterar"
          echo "3 - excluir"
          echo "0 - finalizar"
          echo "\c"
          read RESP
          case $RESP in
               1|2|3)    echo "opção $RESP ok";;
               0)        exit 0;;
          esac
     done
```

O comando *TMP=/tmp/menu$$* acima cria a variável *TMP* com o

valor */tmp/menu657*. A variável *$$* representa o número do processo corrente (*657* é o valor de *$$* neste exemplo).

O comando *echo > $TMP* cria o arquivo */tmp/menu657* só para termos alguma coisa para remover quando houver uma interrupção.

Ao executar *menu1* se teclarmos *Del* ou *Ctrl* + \ (as teclas de controle e contra-barra apertadas simultaneamente) a seguinte mensagem aparecerá no vídeo:

```
removendo arquivo {}
usage: rm [-fir] file ...
```

indicando que o comando:

```
trap "echo removendo arquivo {$TMP}; rm $TMP; exit 1" 2 3
```

foi executado e a variável *TMP* não tinha valor, demonstrado pela mensagem do *echo "{}"* e pela mensagem de erro do comando *rm*.

Por quê o conteúdo da variável *$TMP* apareceu nulo *{}*?

A resposta está no fato de que o shell lê todo o script antes de executar. Como o comando *trap* foi interpretado antes da definição da variável, o conteúdo da mesma foi tratado como nulo e, consequentemente, não foi passado nome de arquivo para o comando *rm*.

18.7. Exemplo de uso de '...' no comando *trap*

A solução do problema anterior é fazer com que o shell só interprete a linha de comandos, que serve para o *trap* quando ocorrer a interrupção. A forma de dizer ao shell para não fazer substituição de variáveis é através de apóstrofos *'...'*.

A nova versão do script *menu1* é o *menu2* seguinte:

```
$ cat menu2

     trap 'echo removendo arquivo {$TMP}; rm $TMP; exit 1' 2 3
     TMP=/tmp/menu$$
     echo > $TMP
     while true
     do
          echo "1 - incluir"
          echo "2 - alterar"
          echo "3 - excluir"
          echo "0 - finalizar"
          echo "\c"
          read RESP
          case $RESP in
               1|2|3)    echo "opção $RESP ok";;
               0)        exit 0;;
          esac
```

done

Ao executar *menu2* se teclarmos *Del* ou *Ctrl* + \ (as teclas de controle e contra-barra apertadas simultaneamente) a seguinte mensagem aparecerá no vídeo:

removendo arquivo {/tmp/menu677}

e o comando *rm* funcionará removendo o arquivo */tmp/menu677*.

No caso de usarmos *trap 'comandos' sinais*, o shell não interpretará o que está entre apóstrofos. Os comandos entre apóstrofes *comandos* só serão interpretados na ocorrência de *sinais*.

PARTE III: PROGRAMAÇÃO DO SHELL

19. O SHELL COMO PROCESSADOR DE CADEIAS DE CARACTERES

Neste capitulo faremos uma análise do shell visto como um Processador de Strings.

19.1 O shell visto de outro ângulo

[KORN 88] foi quem nos alertou para o fato de que o shell pode ser visto como um processador de cadeia de caracteres.

O que o shell faz realmente?

Se formos enumerar as atividades do shell iremos verificar, no final das contas, que o shell é um processador de cadeias de caracteres: ele faz o parsing da linha de comandos; interpreta caracteres especiais; expande metacaracteres; faz manipulação de strings através de substituição ($var, `...`) e concatenação. Discutiremos mais adiante este aspecto do shell.

19.2 Parsing da linha de comandos

Normalmente o shell faz a interpretação da linha de comandos uma só vez. Quando você digita um comando:

$ comando

O shell monta cadeia de caracteres e depois a interpreta para execução, onde a primeira palavra da cadeia é o nome do comando e as demais palavras servem como argumentos para o mesmo, isto nós já sabemos. Vamos analisar a seguir como uma cadeia de caracteres pode ser obtida:

1.) A partir de uma constante

```
$ date
 ...

$ who
 ...
```

(O usuário digita a cadeia de caracteres constante *date* e *who*, por exemplo)

2.) a partir de uma substituição de variáveis

```
$ var=alvaro
$ echo $var
+ echo alvaro
```

(O shell, antes de executar o comando:
$ echo $var, faz a substituição da variável *var* e o comando efetivamente executado é
+ echo alvaro. O caractere *+*, foi usado para indicar que a linha foi substituida pelo shell)

3.) por meio de uma concatenação

$ var=alvaro
$ x=$var" "$var
$ echo $x
+ echo alvaro alvaro

(O shell faz a concatenação *$var" "$var* antes da atribuição). O conteúdo da variável *x* é *alvaro alvaro*)

4.) A partir de metacaracteres

*$ echo arq**
+ echo arq1 arq2 arq3.c

(O shell pesquisa o diretório corrente a procura de nomes de arquivos que *case* com a expressão *arq**, expandindo a linha para
+ echo arq1 arq2 arq3.c)

5.) A partir de um duto (`comando`)

$ echo data `date` agora
*+ echo data **Thu Aug 8 13:40:33 PDT 1991** agora*

(Uma cadeia de caracteres pode ser obtida por intermédio de uma construção interna do shell `...`. Quando o shell encontra este comando, ele cria um *duto* para ler o resultado do comando que está entre crases `...`. Se o usuário digitar:

$ echo data `date` agora

O shell expande a linha para:

*+ echo data **Thu Aug 8 13:40:33 PDT 1991** agora*

o comando *echo* já recebe a cadeia de caracteres expandida pelo shell)

19.3 Parsing mais de uma vez

Mostraremos aqui outro recurso do shell, o comando embutido eval, para facilitar a manipulação da expansão da linha de comandos.

Às vezes é preciso avaliar a linha (fazer o parsing) mais de uma vez. Vamos supor o exemplo abaixo:

```
$ echo Mensagem > arquivo

     + echo Mensagem
     $
```

No comando acima a mensagem *Mensagem* foi armazenada no arquivo *arquivo*. O shell interpretou o redirecionamento *> arquivo* redirecionando a saida padrão do comando *+ echo Mensagem* para *arquivo*. Vejamos o conteúdo de *arquivo*:

```
$ cat arquivo
Mensagem
```

Através do exemplo abaixo, vamos mostrar que às vezes é preciso que o shell interprete a linha mais de uma vez:

```
$ MAIOR=">"
$ echo Mensagem $MAIOR arquivo

     + echo Mensagem > arquivo
     Mensagem > arquivo
```

No exemplo acima o shell só interpretou a linha de comandos uma única vez: No comando *echo Mensagem $MAIOR arquivo* a variável *$MAIOR* foi substituida resultando: *echo Mensagem > arquivo*, mas o caractere de redirecionamento > não foi interpretado pelo shell, uma segunda vez. Uma prova disto é que a mensagem:

```
     "mensagem  > arquivo"
```

foi impressa no video.

19.3.1 O COMANDO *eval*

O comando *eval* faz com que o shell interprete a linha de comandos mais uma vez. Tomemos o exemplo anterior:

```
$ MAIOR=">"
$ echo Mensagem $MAIOR arquivo

     + echo Mensagem > arquivo
     Mensagem > arquivo
```

O arquivo *arquivo* não foi criado porque o shell só interpretou a linha de comando uma vez. Podemos usar *eval* para avisar ao shell que o parsing da linha deve ser feito duas vezes: uma normalmente e a outra por causa de *eval*:

```
$ MAIOR=">"
$ eval echo Mensagem $MAIOR arquivo

     + eval echo Mensagem > arquivo
     + echo Mensagem
     Mensagem
```

Observe que temos duas linhas começando com +, indicando que o shell fez o parsing duas vezes. Na primeira linha houve a substituição da variável *$MAIOR* pelo seu respectivo valor >. Na segunda passagem, por causa de *eval*, o shell interpretou o redirecionamento da saida padrão do comando *echo* para *arquivo*. A mensagem *mensagem* foi armazenada em *arquivo* por causa do comando *eval*.

19.4 Ortogonalidade do pocessador de strings shell

O shell visto como um processador de cadeia de caracteres não é completamente ortogonal. Se o shell fosse completamente ortogonal ele executaria o laço *for* abaixo:

```
$ cat f

     echo for


$ cat d

     echo do


$ `f` i in a b c d e
> `d`
>          echo $i
> `d`ne
```

19.5 Aplicação

O exemplo acima não funciona. O que significa que o shell, visto como um processador de strings, não é completamente ortogonal.

O shell é uma linguagem de programação voltada para a manipulação de cadeias de caracteres. O shell junto com outros utilitários do UNIX possibilitam um rápido desenvolvimento de protótipos de aplicações baseadas na manipulação de arquivos e cadeia de caracteres.

[ARTH 90] levanta uma questão interessante com relação ao shell e ao UNIX: a transportabilidade das aplicações. Programas escritos em shell são transportáveis. Outro ponto discutido é a questão de se poder extrair informações a partir de dados brutos simplesmente usando o conceito de filtro do UNIX e o shell. Aplicações ou consultas imediatas aos dados podem ser construídas através de uma linha de comando shell.

VIII) LISTA DE DESAFIOS DO LIVRO

Os exercicios abaixo estão ordenados de forma crescente, de acordo com o nivel de dificuldade. Os exercicios mais fáceis aparecem primeiro.

1. Construir um programa na linguagem Shell para fazer uma cópia ***backup*** de um diretório.

2. Modifique o programa de backup acima para:

   i) perguntar se o meio magnético foi inserido antes usar o dispositivo;

   ii) criar um arquivo de log contendo o nome dos arquivos que foram copiados.

   iii) Usar tabelas para novas unidades que venham a ser adquiridas ou removidas da máquina;

   iv) Configurar opções default.

3. Escreva um script de nome ***cade*** que mostre a localização (o percurso) do comando na máquina. Por exemplo:

   $ cade ls
   /bin/ls
   $ cade cade
   /u/alvaro/tese/cade

4. É comum ter nomes de arquivos no diretório corrente com sequência de caracteres invisiveis. Escreva um comando que permita visualizar estes caracteres não imprimiveis. Veja um exemplo de ativação deste novo comando chamado ***ils***:

   $ ils
   arq[02]1.c (onde 02 é o caracter ^B)

   Note que o caractere invisivel ^B (Control+B), que faz parte do nome do arquivo *arq^B.c*, foi impresso no formato *[02]*.

5 Escreva um programa produtor de informações, que imprima números na saida padrão. A sintaxe de uso do comando está descrita a seguir:

   Sintax: from [inicio] [fim] [incremento]
   Valores default inicio -> 1

fim -> *10*
incremento -> *1*

Veja um exemplo de uso deste comando de nome *from*:

```
$ from 10 30 5
10
15
20
25
30
```

6. Escreva uma ferramenta que permita setar a variável TERM quando o usuário entrar no ar.

7. Crie um comando para copiar um diretório completo em outro com a seguinte sintaxe:

cpdir fonte destino

O comando ***cpdir*** deve copiar a sub-árvore *fonte* para o diretório *destino*. Este último diretório já deve existir.

O comando deve ser construído em três versões. A primeira deve usar, internamente, o comando *cpio* do UNIX. A segunda versão do comando *cpdir* pode ser feito em função do comando *tar* do UNIX. Finalmente, a última versão do comando *cpdir* a ser construído deve usar apenas os recursos da linguagem de programação Shell.

8. Verifique se a ferramenta ***troca***, construída ao longo do primeiro capítulo deste livro, já está com sua sintaxe de uso no seguinte formato:

troca padrão1 padrão2 [arq ...]

9. Escreva um program ***inter*** que permita a execução de qualquer programa interativamente, aplicado a cada um dos arquivos do diretório corrente. Por exemplo, se o diretório contiver os arquivos ***a***, ***b***, e ***c***, então a execução do comando

inter rm

aplicaria o comando ***rm*** a cada um dos arquivos ***a***, ***b*** e ***c***, um por vez, pedindo confirmação antes de executar o comando. A saída seria, por exemplo:

rm a ?

Você responde apenas **s** ou **n** para executar ou não o comando

rm. Como o diretório corrente tem mais de um arquivo, a próxima pergunta feita pelo interpretador de comandos *inter* seria:

rm b ?

e assim sucessivamente. Observe que no exemplo acima o comando *rm* é apenas um exemplo de argumento para o comando *inter*. O comando *inter* deve funcionar para interpretar, como argumento, qualquer comando do UNIX que aceite um arquivo como parâmetro. Veja a seguir exemplos de uso do comando *inter*:

```
inter cat
inter ls -l
inter chmod +x
```

e assim por diante.

10. Modifique o comando *inter* para que qualquer ocorrência do string {} seja trocado pelo nome do arquivo corretamente sendo tratado. Em outras palavras, queremos que os símbolos *{}* sirvam como metacaractere para o comando *inter* representar o nome do arquivo corrente na linha de comandos. Vejamos um exemplo de como isto vai funcionar na prática. Vamos supor que estamos no mesmo diretório anterior onde existem os arquivos *a*, *b* e *c*. Se voce digitar o novo interpretador de comandos, passando como argumento *cp {} dir*

novo_inter cp {} dir

O comando *novo_inter* acima copiaria os arquivos *a*, *b* e *c* para o diretório *dir*, de forma interativa. Ou seja, o usuário seria questionado com a pergunta

cp a dir ?

feita pelo comando *novo_inter* ao encontrar o arquivo *a* no diretório corrente. O usuário decide, apertando a tecla *s* ou *n*, se quer ou não fazer a cópia. Dando sequência ao processamento dos arquivos do diretório corrente, o comando *novo_inter* faria a seguinte pergunta

cp b dir ?

Finalmente, no processamento do último arquivo do diretório corrente, o comando *novo_inter* perguntaria

cp c dir ?

Observe que o metacaractere *{}* do comando *novo_inter* foi substituido pelos nomes do arquivos do diretório corrente (um nome a cada ciclo de execução do comando *novo_inter*).

11. Será que sua solução para o exercício 10 foi geral ou especifica? Modifique (se necessário) seu script anterior *novo_inter* para que o seguinte exemplo funcione:

inter3 mv {} {}.bak

Assumindo que os arquivos *arq1*, *texto1* e *texto3* sejam os únicos arquivos do diretório corrente e que o comando *inter3* seja a nova versão do comando *novo_inter*, as três peguntas deveriam ser impressas na tela pelo programa *inter3*:

```
mv arq1 arq1.bak ?
mv texto1 texto1.bak ?
mv texto3 texto3.bak ?
```

12. Agora estamos precisando de uma ferramenta que crie subdiretórios vazios. Faça um script com a seguinte sintaxe:

criadir dir [dir ...]

que crie um diretório passando como argumento. O comando ***criadir*** deve funcionar mesmo que vários pedaços do percuros estejam faltando. Vejamos um exemplo prático de ativação deste novo comando. A execução do comando ***criadir*** abaixo:

$ criadir /tmp/dir1/dir2/dir3/dir4

criaria os seguintes diretórios:

```
/tmp                    (se for preciso)
/tmp/dir1
/tmp/dir1/dir2
/tmp/dir1/dir2/dir3
/tmp/dir1/dir2/dir3/dir4
```

13. Crie dois comandos ***ativa*** e ***desativa*** que habilitem e desabilitem, respectivamente, o login em terminais no sistema operacional UNIX. A sintaxe dos comandos está descrita a seguir:

```
ativa terminal01 [ terminal02 ...]
desativa terminal01 [ terminal02 ...]
```

Vejamos como exemplo prático a ativação dos terminais 5, 2 e 6:

ativa tty05 tty02 tty06

Note que para destivar um terminal é o mesmo procedimento:

desativa tty03

14. Escreva um programa denominado ***quemeh*** com a seguinte

sintaxe:

quemeh uid [uid ...]

Este programa deve imprimir na sua saída padrão o nome do usuário cuja identificação foi passada como argumento. Veja um exemplo para saber qual o nome do usuário que tem identificação igual a zero:

```
$ quemeh 0
root
```

15. A maioria dos ambientes operacionais UNIX permitem troca de arquivos como o MSDOS. Porém, há um incoveniente. Como o sistema operacional MSDOS não faz diferença entre letras minúsculas e maiúsculas, os arquivos copiados a partir do MSDOS chegam no UNIX sempre em letras MAIUSCULAS. Você quer nos ajudar a solucionar este problema? Então escreva um programa que mude os nomes de arquivos com letras MAIUSCULAS em nomes com apenas letras minúsculas. Fique a vontade para decidir o nome e a interface da nova ferramenta.

16. Fazer com que o comando ***rm*** peça confirmação para remover mais que *RMMAX* arquivos de uma só vêz. *RMMAX* é uma variável do ambiente.

```
$ RMMAX=3
$ rm a b
$ rm c d e f
Confirma remoção de "c" ? _
Confirma remoção de "d" ? _
Confirma remoção de "e" ? _
Confirma remoção de "f" ? _
```

17. Re-escreva o comando ***rm*** incluindo uma nova opção ***-a*** ("a" de arrependido) para recuperar (**fazer aparecer**) arquivos removidos enquanto a máquina estiver ligada.

18. Em um ambiente onde há uma demanda de usuários maior do que o número de terminais disponíveis, o que fazer para escalonar o pessoal no uso do computador?

    Sugestão: Implemente COTA DE TEMPO POR USUARIO. Cada conta só pode usar o computar por um intervalo de tempo. Ao término deste período, o usuário deverá ser notificado que sua fatia de tempo terminou. Após o aviso o usuário terá dois minutos para encerrar a sessão, findo os quais todos os programas associados ao seu terminal deverão ser cancelados.

19. Implementar o comando ***hist*** que mostra o histórico dos últimos comandos digitados pelo usuário. Algumas opções devem estar disponíveis:

hist	-	Mostra os últimos comandos
!	-	Executa o último comando
!n	-	Executa o comando de número "*n*"
!e	-	Edita último comando

20. Muitas vezes queremos alterar o conteúdo de uma variável do ambiente e não temos nenhum recurso no Bourne Shell para isto. Ajude a solucionar este problema. Implemente um **editor de variáveis**. Crie o comando de nome ***edv*** com este objetivo. Agora para **editar** uma variável é só chamar ***edv*** passando o nome da variável a editar como parâmetro. Vejamos um exemplo prático onde queremos alterar o conteúdo da variável ***TERMCAP***:

```
$ edv TERMCAP
          ... (recursos do editor do vi)
```

Uma otimização do comando ***edv*** poderia ser feita para tornar este comando mais flexível. Ao invés de sempre usar o editor *vi*, o comando *edv* poderia usar o editor de texto que estivesse configurado através da variável *EDITOR*. Assim, o usuário que usar outro editor é só atribuir seu nome à variável *EDITOR*:

```
$ EDITOR=meu_editor_predileto
$ edv TERMCAP
          ... (recursos do meu_editor_predileto)
```

21. Fazer um utilitário que ***trave*** o terminal durante a ausência do usuário.

```
$ trave
Senha: _
```

O terminal só deve ser destravado se for digitada a mesma senha de travamento.

22. Implementar uma agenda de compromissos. Ao entrar no ar, o computador deverá mostrar os compromissos de hoje até uma data previamente configurada. O rolamento de página na tela deverá ser tratado.

23. Fazer um comando que informe quais são os usuários que fazem "login" e quais usuários que fazem "logoff".

24. Fazer um programa que ***mate*** os processos pelo seu nome.

25. Fazer o comando ***tgrep*** que pesquisa um padrão em uma sub-árvore.

26. Como resolver o problema de editores de texto que não fazem o travamento durante a edição de um arquivo? Este problema possibilita mais de um usuário gravando informações ao mesmo tempo em um mesmo arquivo, o que é uma catastrofe. O editor *vi* é um exemplo concreto de um editor de texto muito permissivo. Crie um novo editor de texto, em função do *vi*, que faça o travamento a nível de arquivo. Um arquivo não poderá ser editado simultaneamente por mais de uma pessoa.

27. Fazer com que o calendário do mês corrente seja apresentado ao usuário no início de sua sessão. O dia corrente deve ser impresso em letras destacadas das demais. Todo o calendário deve ser impresso em uma moldura de caracteres semigráficos.

IX) CONCLUSÃO DA TESE

A conclusão a que chegamos pode ser dividida em duas partes. A primeira com relação ao livro propriamente dito. A segunda trata da nosso experiência pessoal a ser passada para outras pessoas que desejem escrever um livro.

Sobre o livro, identificamos as principais caracteristicas que um bom livro deve ter: um bom índice; um bom índice remissivo; uma boa bibliografia; uma apresentação dos assuntos de forma didática; bons resumos no final dos capitulos; figuras significativas e coerentes com os exemplos; um aspecto gráfico de qualidade comparada aos livros de última geração; muitos exemplos e exercícios; um conteúdo que venha preencher uma lacuna no mercado.

Acreditamos que este trabalho atende a quase totalidade dos requisitos acima. Isto porque o escopo deste trabalho foi muito grande.

Finalmente, a experiência que queremos passar para outras pessoas que desejem escrever um livro é que elas adotem, desde cedo, uma metodologia e sejam disciplinadas. A metodologia de despejo de memória funcionou muito bem conosco.

X) INDICE REMISSIVO

D

E

F

G

H

I

S

T

U

V

W

X

XI) BIBLIOGRAFIA

[ANDE 86], ANDERSON, Gail & ANDERSON Paul - **The UNIX C Shell Field Guide** - Prentice-Hall, 1986;

[ANDL 90], ANDLEIGH, Prabhat K. - **UNIX System Architecture** - Prentice-Hall, 1990;

[AHO 87], AHO, V. Alfred & KERNIGHAN, W. Brian & WEINBERGER, Peter - **The AWK Programming Language** - Addison Wesley, 1987;

[ARTH 90], ARTHUR, Lowell Gay - **UNIX Programação do Shell** - Livros Técnicos e Cientificos, 1990;

[BACH 86], BACH, Maurice J. - **The Design of the UNIX Operating System** - Prentice-Hall, 1986;

[BOLS 89], BOLSKY, Morris I. & KORN, David G. - **The Korn Shell Command and Programming Language** - Prentice-Hall, 1989;

[COFF 90], COFFIN, Stephen - **UNIX System V Release 3 the Complete** - McGraw-Hill, 1990;

[COFF 88], COFFIN, Stephen - **UNIX System V Release 3 the Complete** - McGraw-Hill, 1988;

[EGAN 88], EGAN, Janet I. & TEIXEIRA, Thomas J. - **Writing a UNIX Device Driver**, Wiley, 1988;

[FIEL 91], FIELDER, David - **SCO Hot, BYTE**, janeiro de 1991, David Fielder, p. 101-104;

[GRAC 87], GRACE, Todino & O'REILLY, Tim - **A Nutshell Handbook "Managing UUCP and Usenet"** - Nutshell, 1987.

[KERN 84], KERNIGHAN, Brian W. & PIKE, Rob - **The UNIX Programming Environment**, Prentice-Hall, 1984;

[KOGH 90], KOGHAN, Stephen G. & WOOD, Patrick H. - **UNIX Shell Programming**, Hayden Books, 1990;

[KORN 88], David G. - **The UNIX Software Readings** "The Shell-Past, Present, and Future", , p. 159-181, Prentece-Hall, 1988;

[MANI 86], MANIS, Rod & MEYER, Marc H. - **The UNIX Shell Program ming Language** - Sams, 1986;

[MORG 86], MORGAN, Christopher L. Morgam - **Inside Xenix** - Sams, 1986;

[ROSE 90], ROSEN, Kenneth H., ROSINSKY, Richard R., & FARBER, James - **UNIX System V Release 4 An Introdution** - McGraw-Hill, 1990;

[SWAR 90], SWARTZ, Ray - **UNIX Applications Programming Mastering the Shell** - Sams, 1990.

ANEXO A

(Tabela-1: Os Sinais do UNIX)

Número	Descrição
1	Usuário desliga o terminal;
2	Usuário digita a tecla de interrupção *<DEL>*;
3	Usuário aperta as teclas *<CONTROL>+<\>* simultaneamente;
4	Sinal gerado quando o núcleo executa uma instrução ilegal;
5	Sinal *TRACE* gerado por uma rotina do núcleo do UNIX;
6	Instrução *IOT*;
7	Instrução *EMT*;
8	Sinal gerado pelo núcleo quando ocorre exceção de ponto flutuante;
9	Sinal *KILL* (morte certa);
10	Erro de barramento;
11	Violação de endereço;
12	Erro na chamada de uma rotina do núcleo;
13	Erro no duto ou pipe;
14	Sinal *ALARM* usado para acordar processos no futuro;
15	Sinal para término do programa por software (comando *kill* do UNIX ativado sem parâmetros);
16	Sinal livre para ser definido pelo usuário;
17	Sinal livre para ser definido pelo usuário;
18	Sinal que indica morte de um processo filho;
19	Falta de energia (usado em sistemas que possuam baterias);