



Sistemas Operacionais

Marcos Aurélio Pchek Laureano

É professor e coordenador do curso Técnico na área de Informática do Instituto Federal do Paraná (IFPR). Doutorando e Mestre em Informática Aplicada pela Pontifícia Universidade Católica do Paraná (PUCPR); Técnico em Processamento de Dados pela Universidade Federal do Paraná (ET/UFPR); Tecnólogo em Processamento de Dados pela Escola Superior de Estudos Empresariais e Informática (ESEEI).

Diogo Roberto Olsen

É professor e coordenador do curso Técnico em Informática para Internet no Instituto Federal do Paraná (IFPR) – Câmpus de Londrina. Mestrando em Informática e graduado em Ciência da Computação pela Pontifícia Universidade Católica do Paraná (PUCPR).

Direção Geral	Jean Franco Sagrillo
Direção Editorial	Jeanine Grivot
Edição	Leonel Francisco Martins Filho
Assistente Editorial	Melissa Harumi Inoue Pieczarka
Gerência de Editoração	Marcia Tomeleri
Revisão	Jeferson Turbay Braga e Miriam Raquel Moro Conforto
Projeto Gráfico e Capa	Adriana de Oliveira
Editoração	Zilceano Fonseca

Olsen, Diogo Roberto.
 Sistemas operacionais / Diogo Roberto Olsen; Marcos Aurelio Pchek Laureano. – Curitiba: Editora do Livro Técnico, 2010.

160 p.
 ISBN: 978-85-63687-15-9

1. Sistemas operacionais (Computadores). I. Marcos Aurelio Pchek Laureano. II. Título.

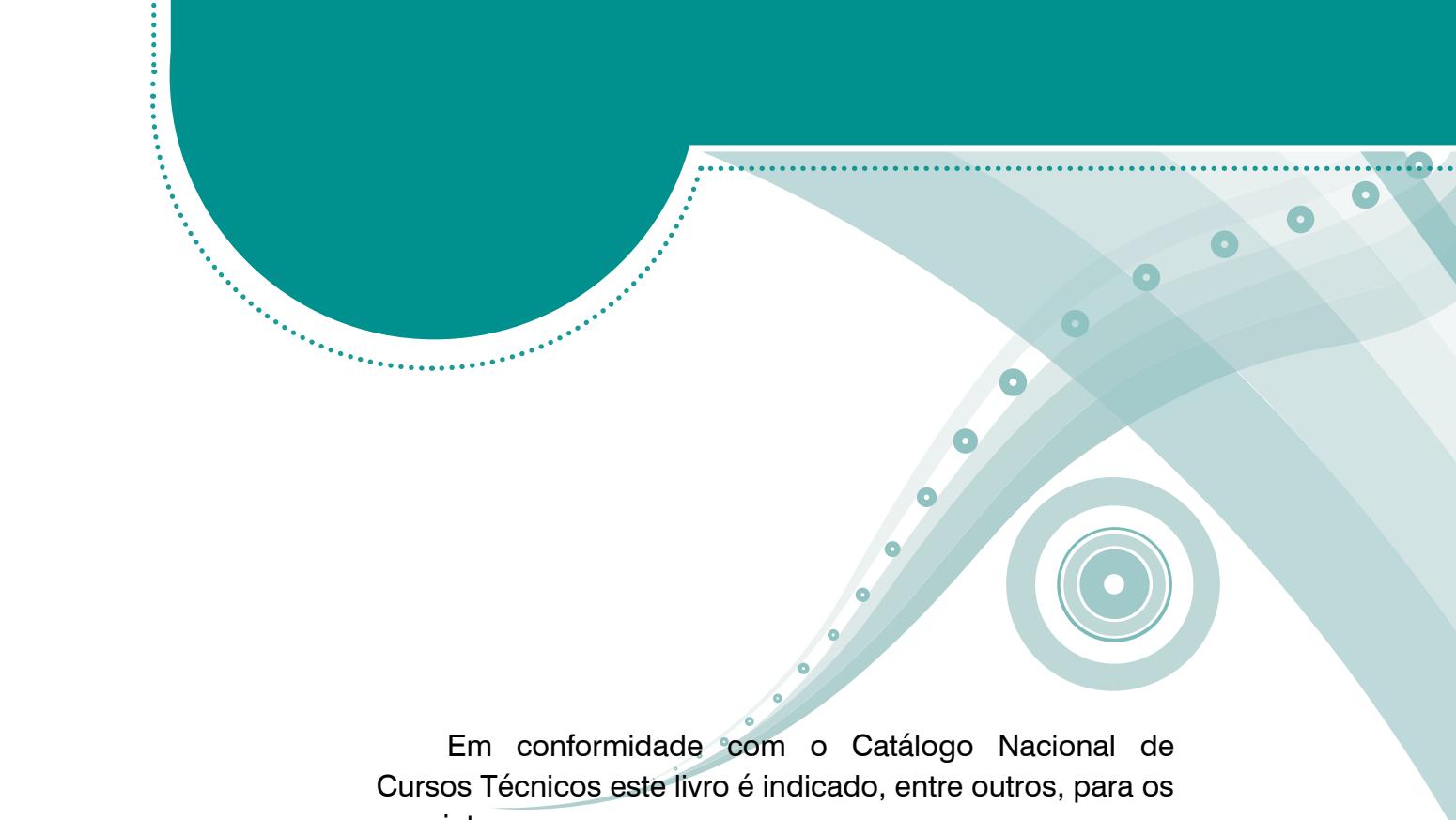
CDD 005.43

Eutália Cristina do Nascimento Moreto CRB-9/947

Telas padrão Microsoft reproduzida com permissão da Microsoft Corporation.

2010

Todos os direitos reservados pela Editora do Livro Técnico
 Edifício Comercial Sobral Pinto
 Avenida Cândido de Abreu, 469 – 2º andar, conj. n°s 203-205
 Centro Cívico – Cep: 80530-000
 Tel.: (41) 3027-5952/Fax: (41) 3076-8783
www.editoralt.com.br
 Curitiba – Pr



Em conformidade com o Catálogo Nacional de Cursos Técnicos este livro é indicado, entre outros, para os seguintes cursos:

Eixo Tecnológico: Informação e Comunicação

- Técnico em Informática
- Técnico em Informática para Internet
- Técnico em Manutenção e Suporte em Informática
- Técnico em Redes de Computadores
- Técnico em Telecomunicações
- Técnico em Programação de Jogos Digitais

Eixo Tecnológico: Controle e Processos Industriais

- Técnico em Mecatrônica
- Técnico em Automação Industrial

Apresentação

O crescimento dos cursos técnicos específicos e com curta duração (2 a 3 anos) gerou uma demanda de livros que tratam diretamente do assunto de maneira clara e eficiente.

Assim, o mercado de trabalho precisa de profissionais que saibam aliar teoria com prática. Neste cenário, não é mais possível aplicar o ensino tradicional de sistemas operacionais (como em cursos de longa duração). É necessário uma abordagem que permita ao futuro profissional conhecer a teoria e, ao mesmo tempo, dominar o ambiente de trabalho.

Logo, este livro é indicado aos cursos técnicos e aos estudantes e profissionais que precisem dominar os conceitos de sistemas operacionais de forma rápida e precisa.

A obra aborda, de forma objetiva, os principais conceitos de sistemas operacionais e, ao mesmo tempo, incentiva a prática para entendimento dos conceitos em um ambiente Linux. Esse ambiente foi escolhido por estar em amplo crescimento e utilização nos meios acadêmicos e, principalmente, no espaço profissional.

Praticamente todos os exemplos e exercícios sugeridos, além de exemplificar a teoria, poderão ser utilizados no dia a dia do futuro profissional.

Sumário

CAPÍTULO 1 – Introdução	9
Arquitetura de Sistemas Operacionais.....	12
Tipos de Sistemas Operacionais	14
Chamada de Sistema (<i>System Call</i>)	16
Atividades.....	17
CAPÍTULO 2 – Sistemas Operacionais Atuais	18
Windows 7	18
Linux 2.6	19
Mac OS x Snow Leopard	20
Google Chrome OS	22
Iphone OS 4	22
Android 2.2.....	23
Symbian^3	24
Windows Mobile 6.5.....	24
Atividades.....	25
CAPÍTULO 3 – Estudo de Caso de Sistemas Operacionais	26
Utilizando o Linux	26
O que é Linux	26
História	27
Características do Sistema Linux	27
Atividades.....	28
Atividade.....	29
Atividade.....	31
CAPÍTULO 4 – Comandos Básicos	33
Iniciando uma Sessão.....	33
Comando exit.....	34
Atividades.....	34
Obtendo Help no Sistema – Páginas de Manual	34
Atividades.....	37
Comando passwd	37
Atividade.....	37
Comando expr.....	38
Atividades.....	38

CAPÍTULO 5 – Sistema de Arquivos

39

Partições	40
Arquivos	41
Diretórios.....	42
Comandos Básicos para Trabalhar com Diretórios.....	43
Comandos Básicos para Trabalhar com Arquivos.....	46
Atividade.....	49
Atividade.....	53
Comandos Avançados.....	59

CAPÍTULO 6 – Sistemas de Arquivos e Segurança

62

Permissão em Arquivos.....	62
Integridade de Arquivos	64
Alterando Permissão dos Arquivos	65
Atividades.....	69

CAPÍTULO 7 – Máquinas Virtuais

70

Por que Máquinas Virtuais Existem?	71
Tipos de Máquinas Virtuais	73
Estratégias de Virtualização	74
Uso de Máquinas Virtuais	76
A Máquina Virtual User-Mode Linux	78
Atividades.....	80

CAPÍTULO 8 – Administração de Usuários

81

Verificando Informações do Usuário	81
Arquivo passwd e group	84
Adicionando Grupos – Comando groupadd	85
Eliminando Grupos – Comando groupdel.....	86
Adicionando Usuários – Comando useradd	86
Alterando a Senha do Usuário – Comando passwd.....	87
Eliminando Usuários – Comando userdel	88
Atividades.....	89

CAPÍTULO 9 – O Sistema de Arquivo /procfs	90
Atividade.....	90
Atividade.....	90
Atividade.....	91
Atividade.....	91
Estrutura do /proc	92
Atividade.....	93
Utilizando o /proc	94
Atividade.....	94
Atividade.....	96
Atividade.....	97
Atividade.....	98
CAPÍTULO 10 – Gerência de Processos	99
Programas e Processos.....	99
Atividade.....	101
Obtendo Informações sobre Processos no /proc.....	102
Multiplexação do Processador.....	106
Atividade.....	108
Atividade.....	119
Usuários e Grupos	119
Atividade.....	120
Entradas e Saídas Padrão de um Processo	120
Atividade.....	126
Prioridades	127
Escalonamento das Tarefas no Linux	128
O Ciclo de Vida de um Processo no Linux.....	132
Atividade.....	151
Atividade.....	159

Referências Bibliográficas

160

Introdução

O sistema operacional é um *software* situado entre o *hardware* e as aplicações para gerenciar todos os recursos do sistema (memória, processador, discos, impressoras e outros dispositivos) de forma organizada e otimizada. É um *software* que habilita as aplicações a interagirem com o *hardware* de um computador.

É o primeiro programa que a máquina executa no momento em que é ligada (em um processo chamado de *bootstrapping* ou inicializar o computador) e, a partir de então, não deixa de funcionar até que o computador seja desligado. Ele reveza sua execução com a de outros programas, como se estivesse vigiando, controlando e orquestrando todo o processo computacional.

O *software* que contém os componentes centrais do sistema operacional é denominado núcleo. Esse núcleo (*kernel*) tem a responsabilidade de gerenciar os diversos recursos presentes no sistema operacional.

Os sistemas operacionais podem ser encontrados em dispositivos que vão de telefones celulares a automóveis e de computadores pessoais a computadores de grande porte (*mainframe*). Na maioria desses sistemas, um usuário requisita ao computador para realizar uma ação (por exemplo, executar uma aplicação ou imprimir um documento). Nestes casos, o sistema operacional gerencia o *software* e o *hardware* para produzir o resultado desejado.

Para grande parte dos usuários, esse sistema é uma “caixa-preta” entre as aplicações e o *hardware* sobre o qual funcionam, e que assegura o resultado correto, dadas as entradas adequadas. Pode-se dizer que os sistemas operacionais são gerenciadores de recursos – administram *hardwares* como: processadores, memória, dispositivos de entrada/saída e dispositivos de comunicação. Também devem gerenciar aplicações e outras abstrações de *software* que, diferentemente do *hardware*, não são objetos físicos.

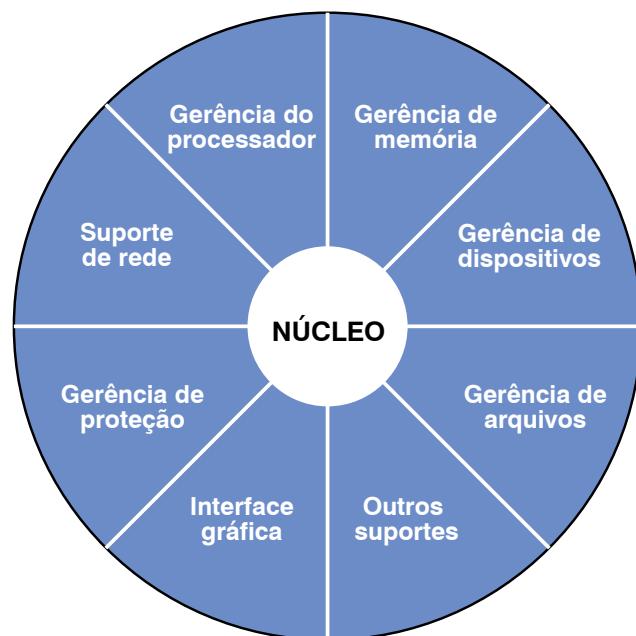
A figura a seguir demonstra esta relação.



O objetivo de um sistema operacional é fornecer uma plataforma operacional para que os usuários possam executar programas. Seu objetivo secundário é utilizar o *hardware* de forma eficaz e eficiente. Ele funciona, então, como gerenciador de recursos e como máquina virtual (compatibilizando aplicações escritas para uma plataforma e que não funcionam em outra), integrando os componentes de *hardware* e criando uma única máquina abstrata.

O que o sistema operacional faz, então, é gerenciar as particularidades de cada *hardware* e criar uma interface homogênea (genérica) para as aplicações a serem utilizadas.

Como gerenciador de recursos, um sistema operacional é composto de vários módulos com funcionalidades distintas. Cada módulo é responsável por gerenciar uma particularidade do sistema. É o que pode ser visto nesta figura:



- **Gerência do processador** – visa a distribuir a capacidade de processamento (uso de CPU) de forma justa. Deve-se lembrar que algumas aplicações demandam mais processamentos que outras (navegador de Internet *versus* processamento de vídeo, por exemplo).
- **Gerência de memória** – tem como função fornecer, a cada aplicação, um espaço próprio de memória, independente e isolado das demais aplicações. Responsável também pelo uso do disco como memória complementar (*swap*), neste caso, a aplicação desconhece o tipo da memória em uso.
- **Gerência de dispositivos** – cada periférico do computador possui suas peculiaridades; logo, temos vários dispositivos diferentes, mas com problemas comuns. *Pen-drives*, discos IDE e SCSI são dispositivos diferentes, em essência iguais, já que basta um endereço ou área de locação para que um determinado dado seja buscado. Logo é possível criar uma abstração única de acesso.
- **Gerência de arquivos** – construída sobre a gerência de dispositivos, possibilita criar abstrações de arquivos e diretórios.
- **Gerência de proteção** – políticas de acesso e uso do sistema operacional. Está disponível no maior dos sistemas operacionais. Permite a definição de usuários, grupos de usuários e registro de recursos por usuários.
- **Interface gráfica** – a interação com o usuário se faz necessária, assim a maioria dos sistemas operacionais apresentam “telas”, nas quais pode-se informar ao sistema operacional qual a operação que ele deverá fazer.
- **Suporte de rede** – a comunicação em rede é, atualmente, essencial ao mundo dos computadores. Assim, o gerenciamento dessas comunicações se faz necessário e é realizado sob uma abstração do sistema operacional sobre os dispositivos físicos, como placas de redes ou *modems*.
- **Outros suportes** – há sistemas operacionais para os mais diversos usos. Sistemas de uso geral (que permitem ao usuário ouvir músicas, navegar na Internet, editar textos) normalmente têm mais recursos para gerência de multimídia. Sistemas de uso específico (que possibilitam o controle de uma usina nuclear, por exemplo) possuem outras características específicas, tais como tempo de resposta ou suporte a um *hardware* especial. Nestes casos, o sistema operacional é mais enxuto e tem menos módulos de gerência (por exemplo, não faria sentido um sistema operacional para controle de uma usina nuclear ter suporte multimídia).

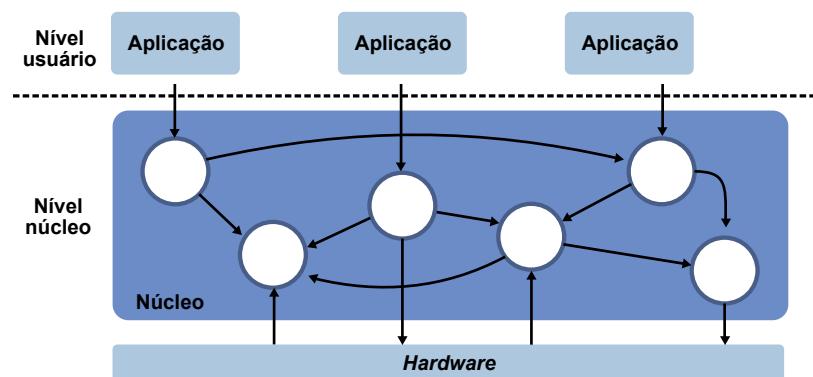
Cada sistema operacional tem suas características, ou seja, alguns sistemas podem oferecer mais recursos que outros.

Neste livro, demonstraremos e exporemos conceitos relativos a sistema de arquivos, gerência e comunicação de processos; gerência de memória e recursos e gerência de usuários, utilizando o sistema operacional Linux como base para testes. Ao término da obra, esperamos que, além de você entender os conceitos mais simples sobre um sistema operacional, tenha domínio de um sistema moderno que conta com ampla utilização na indústria.

Arquitetura de Sistemas Operacionais

Em função de sua arquitetura os sistemas operacionais modernos podem ser classificados em três tipos:

- **Núcleo monolítico ou monobloco** – é um núcleo que implementa uma interface de alto nível, para possibilitar chamadas de sistema específicas para gestão de processos, concorrência e gestão de memória por parte de módulos dedicados, que são executados com privilégios especiais. Mesmo que cada módulo de manutenção dessas operações seja separado, de uma forma geral, é muito difícil fazer o código de integração entre todos esses módulos. E, uma vez que todos eles executam em um mesmo espaço de endereçamento, um erro em um módulo pode derrubar todo o sistema.

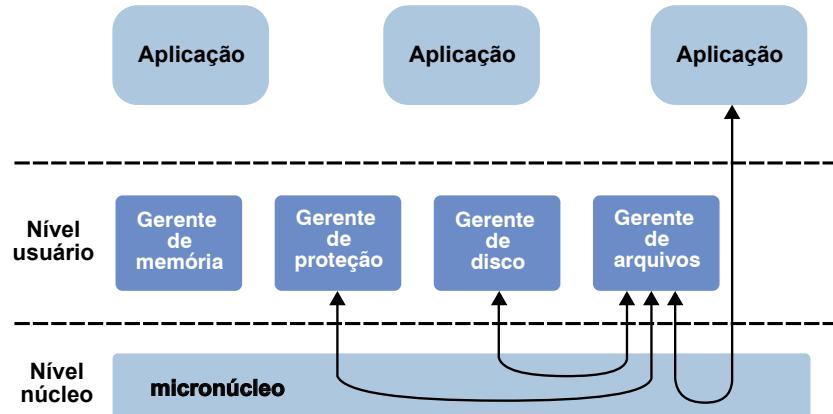


12

Você Sabia

Que os maiores exemplos de sistemas operacionais monolíticos são o Windows e o Linux?

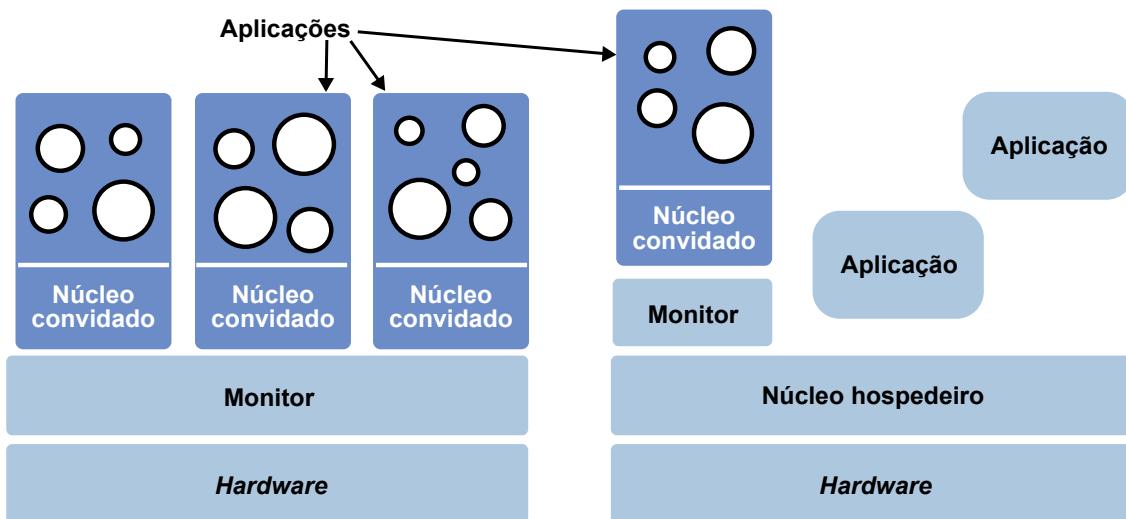
- Introdução
- **Micronúcleo ou *microkernel*** – é um termo usado para caracterizar o sistema cujas funcionalidades saíram do núcleo e foram para servidores, que se comunicam com um núcleo mínimo, usando o mínimo possível do “espaço do sistema” (nesse local, o programa tem acesso a todas as instruções e a todo o *hardware*) e deixando o máximo de recursos rodando no “espaço do usuário” (nesse espaço, o *software* sofre algumas restrições, não podendo acessar alguns *hardwares* e não tendo acesso a todas as instruções).



Você Sabia

Que sistemas de micronúcleo são considerados os mais seguros e funcionais? Mas também são considerados os mais difíceis de se manter? O criador do projeto GNU (projeto com objetivo de criar um sistema operacional totalmente livre), Doutor Richard Stallman, propôs a criação e começou a trabalhar no sistema operacional GNU HURD em 1990, mas até a atualidade o sistema não alcançou a maturidade adequada para uso em sistemas de produção.

- **Máquina virtual** – é uma arquitetura que permite que um sistema operacional suporte aplicações de outro sistema (Linux executando sobre o Windows, por exemplo) ou mesmo outro sistema operacional completo. Possibilita que sobre um mesmo *hardware* possam ser executados dois ou mais sistemas operacionais diferentes.



No primeiro caso, o monitor executa diretamente sobre o *hardware* e gerencia um ou mais sistemas operacionais convidados. No segundo caso, o monitor possibilita a execução de um sistema operacional sobre o outro.

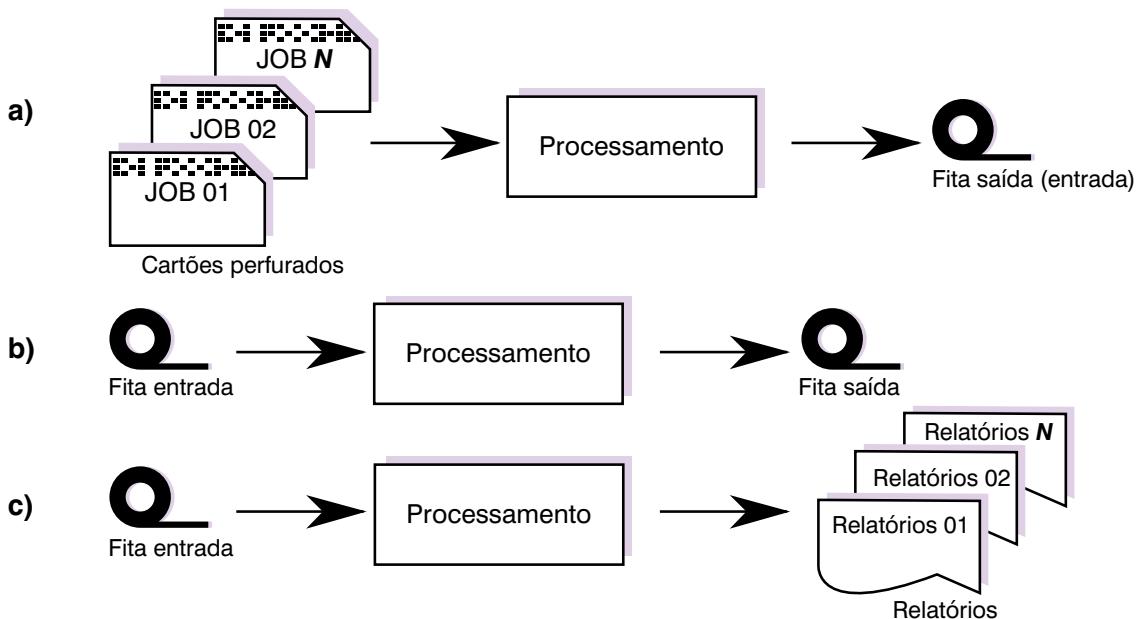
Você Sabia

Que os maiores exemplos de aplicações de máquinas virtuais são bem diferentes entre si? O XEN é livre para uso e com seu código fonte disponível para estudo e o VMware é proprietário e, portanto, deve ser comprado para ser utilizado.

Tipos de Sistemas Operacionais

Os sistemas operacionais podem ser dos mais diversos tipos ou utilizados para os mais diversos fins. Alguns podem ser do mesmo tipo (Windows, por exemplo, pode ser considerado um sistema operacional de rede, servidor, multi-usuário e *desktop*). Os tipos de sistemas operacionais são:

- **Batch ou lote** – neste tipo todos os programas são colocados em uma fila para execução pelo processador. Este recebe um programa após o outro (em sequência). Atualmente o termo **batch** é utilizado para definir um conjunto de comandos que rodam sem interferência do usuário.



- **Rede** – a característica deste sistema é possibilitar o compartilhamento dos recursos de computadores e disponibilizá-los para uso. A maioria dos sistemas operacionais modernos implementam recursos de rede.

Converse com o professor de redes de computadores sobre quais sistemas operacionais serão utilizados para entender os conceitos de redes que serão vistos no curso.

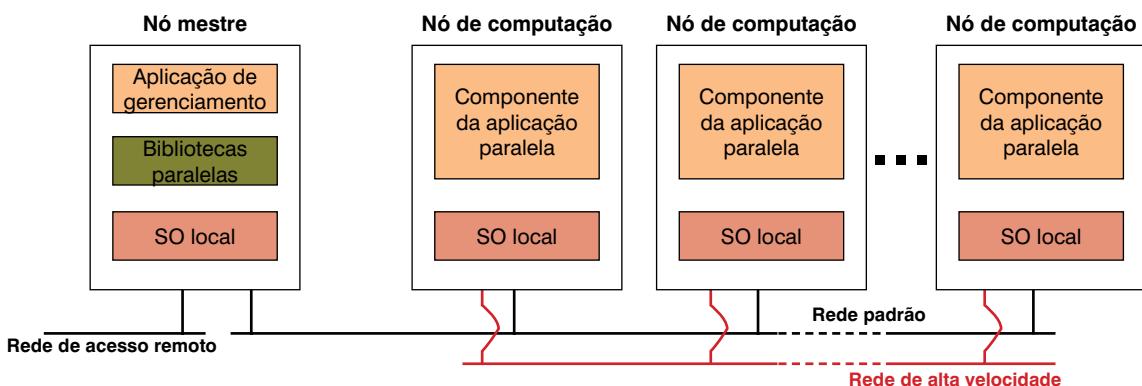
- **Distribuído** – os recursos de cada máquina estão disponíveis globalmente de forma transparente para o usuário. Do ponto de vista das aplicações é como se não houvesse um conjunto de vários computadores, mas apenas um único sistema centralizado. Portanto o usuário **desconhece** qual o computador que atendeu a sua solicitação. Infelizmente, estes sistemas operacionais não são uma realidade de mercado, um exemplo é o Amoeba.

Você Sabia

Que o Amoeba, além de ser um sistema operacional distribuído, implementa o conceito de micronúcleo?

Que no dia a dia utilizamos vários serviços, como o do buscador Google, que aparentemente funcionam como se fosse apenas uma única máquina, mas na prática são formados por conjuntos de grandes computadores servidores espalhados pelo mundo?

- **Cluster** – um *cluster*, ou aglomerado de computadores, é formado por um conjunto de computadores que utiliza um tipo especial de sistema operacional classificado como sistema distribuído. Muitas vezes, é construído a partir de computadores convencionais, os quais são ligados em rede e comunicam-se por meio do sistema, trabalhando como se fossem uma única máquina de grande porte.



15

Introdução

Você Sabia

Que uma das tecnologias de *cluster* mais utilizadas foi criada pela NASA? Ela se chama Beowulf e está disponível para ser usada por qualquer organização que deseje montar um *cluster*.

- **Multusuário** – sistema operacional que possibilita vários usuários simultâneos. Esse sistema deve suportar a identificação do **dono** de cada recurso, tais como arquivos, processos ou conexões de rede, e impor regras de controle de acesso para impedir o uso desses recursos por usuários não autorizados. Os principais sistemas operacionais modernos são considerados multusuários, tais como Linux, Unix e Windows.

- **Desktop** – sistema operacional para uso doméstico ou corporativo. Utilizado para atividades corriqueiras como escutar música, editar textos ou navegar na Internet. Normalmente possui um ambiente gráfico e suporte de rede. Exemplos: Windows e Linux.
- **Servidor** – sistema operacional que permite gerir grandes quantidade de recursos, tais como discos, memórias ou processadores. Possui suporte para multiusuários e controla prioridades e quotas de uso do sistema. Exemplos: Windows e Linux.
- **Embutido ou embedded** – também conhecido como embarcado. É um sistema operacional com função específica e bem definida. Normalmente utilizado em *hardwares* com pouca capacidade de processamento, como celulares, calculadoras ou tocadores de MP3.
- **Tempo real** – sistema operacional com tempos de respostas conhecido no melhor e no pior caso. Pode ser do tipo *soft*, em que a perda de prazo implica degradação do serviço prestado, ou *hard*, em que a perda de prazo pode causar grandes prejuízos econômicos ou ambientais. Exemplo: sistemas embutidos em caças F-16.

Você Sabia

Que sistemas de tempo real são utilizados em aplicações de missão crítica, como hidroelétricas, plataformas de petróleo, caldeiras industriais e guindastes submarinos? O principal sistema operacional de tempo real em uso é o QNX Neutrino RTOS (<http://www.qnx.com>) que é um sistema de micronúcleo.

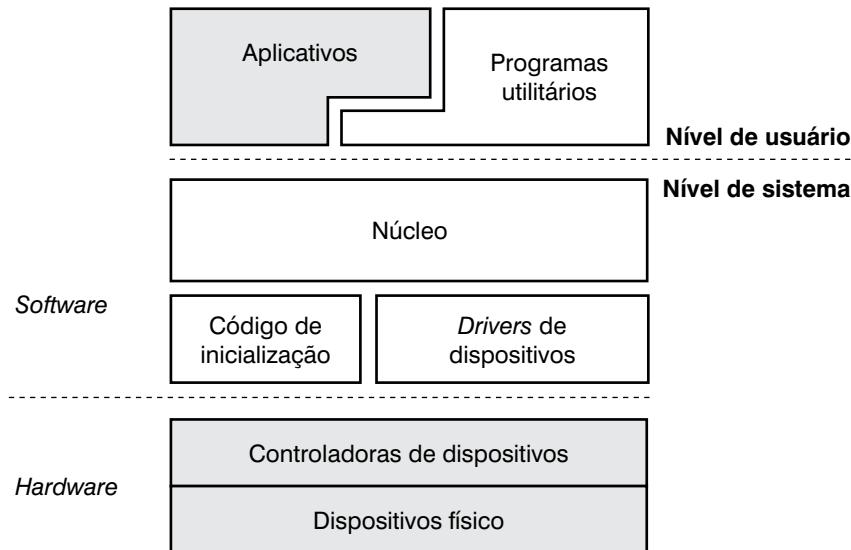
16

Chamada de Sistema (*System Call*)

Chamadas de sistema constituem o método utilizado pelos programas para solicitar serviços ao sistema operacional. Os processadores trabalham em dois modos e, portanto, o sistema operacional opera em dois modos: usuário e privilegiado. Isso para obter proteção, já que o modo usuário não pode executar tarefas que possam comprometer o sistema operacional.

No modo usuário (ou *userspace*), apenas algumas instruções do processador estão disponíveis para a aplicação utilizar. Instruções como HALT (paralisar o processador) ou RESET (reiniciar o processador) somente estão disponíveis no modo privilegiado (supervisor ou monitor). Neste, o sistema tem acesso irrestrito a todas as instruções do processador, inclusive as instruções “perigosas” (como HALT ou RESET).

Uma aplicação de usuário (um editor de texto, por exemplo) não pode ter acesso irrestrito ao processador e, portanto, deve ser executado no modo usuário. Não seria interessante o editor de textos afetar o funcionamento do *player* de música. Mas, muitas vezes, a aplicação precisa ter acesso a recursos do sistema operacional (como ler o conteúdo de um arquivo, por exemplo).



Para que as aplicações possam acessar de forma segura os recursos de *hardware*, o sistema operacional disponibiliza um conjunto de chamadas de sistemas (*system call*). Uma chamada de sistema nada mais é do que uma instrução específica disponibilizada pelo sistema operacional para a aplicação. Sempre que a aplicação necessita de um serviço específico do sistema operacional, ela se utiliza de uma chamada de sistema para isto. Por exemplo, a chamada de sistema **open** indica que o aplicativo deseja abrir um arquivo, a chamada **read** indica que se deseja ler o conteúdo de um arquivo ou dispositivo de entrada (teclado, por exemplo).

A chamada de sistema possibilita, então, que a aplicação tenha acesso ao modo privilegiado, mas de forma controlada pelo sistema operacional. As chamadas de sistemas também são conhecidas como “portas de entrada para acesso ao núcleo do sistema operacional”.

O conjunto de chamadas de sistema oferecidas por um núcleo define a API (*Application Programming Interface*) desse sistema operacional. O sistema Windows oferece um conjunto de instruções conhecido como Win32. No Linux, é utilizado a API POSIX, que também é a interface padrão dos sistemas operacionais Unix disponíveis no mercado (AIX, OpenBSD, FreeBSD, etc.).



Atividades

- 1)** Faça uma pesquisa e tente identificar os principais sistemas operacionais em uso. Classifique-os de acordo com sua arquitetura (monolítico, micronúcleo ou máquina virtual) e tipo (rede, multiusuário, etc.).
- 2)** Assista aos filmes *Revolution OS* e *Piratas do Vale do Silício* para conhecer melhor a história dos sistemas operacionais.
- 3)** Veja junto com o professor o documentário *Triunfo dos Nerds*.

Sistemas Operacionais Atuais

Para pessoas envolvidas com tecnologia, além de conhecimentos técnicos e teóricos, é muito importante estar atualizado em relação aos *softwares* mais modernos e usados no mercado da tecnologia. O objetivo deste capítulo é apresentar alguns dos sistemas operacionais mais modernos usados atualmente.

Além de apresentar os sistemas operacionais dos tradicionais computadores de mesa, este capítulo irá mostrar alguns sistemas usados em dispositivos portáteis, como os *smartphones*. Além de serem tecnicamente muito interessantes, esses sistemas são muito usados, visto que em 2010 o Brasil já conta com mais de 191 472 142 assinaturas de telefones móveis, segundo dados da Anatel (<http://www.anatel.gov.br>).

Primeiro serão apresentados os sistemas operacionais para computadores de mesa, Windows 7, Linux 2.6, Mac OS X, Google Chrome OS. Depois, serão apresentados os Sistemas Operacionais para dispositivos portáteis Iphone OS 4, Android 2.2, Symbian ^3 e Windows Mobile.

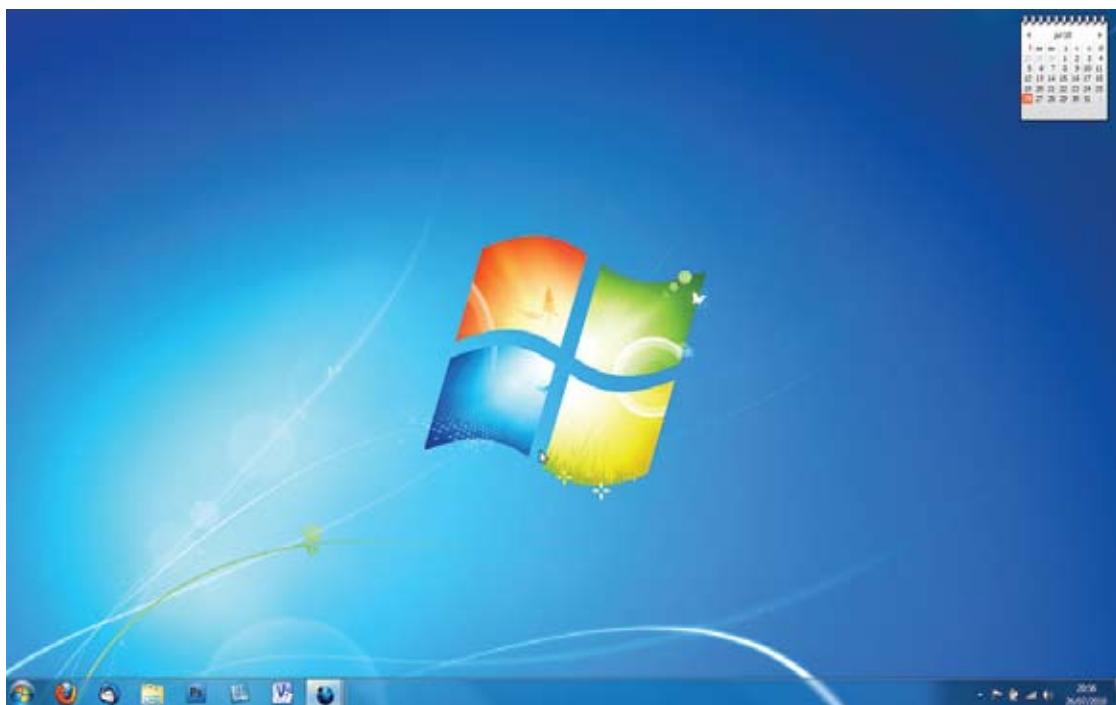
Windows 7

Lançado em outubro de 2009, o Windows 7 é a ultima versão dos sistemas operacionais da Microsoft. O Windows 7 possui seis versões diferentes: Starter, Home Basic, Home Premium, Professional, Ultimate e Enterprise. As primeiras versões apresentam menos recursos e são mais baratas que as últimas.

Entre os recursos presentes no sistema podem ser citados: a reformulação do navegador de Internet, que se tornou o Internet Explorer 8; a reformulação da interface gráfica; as melhorias de desempenho; o menor tempo de *boot* (processo de inicialização do computador que carrega o sistema operacional) e o aperfeiçoamento no uso de placas de vídeo e de memória RAM.

Os requisitos mínimos para que o Windows 7 funcione, em uma máquina 64 bits, são: um processador de 1 GHz, 2 GB de memória RAM, 20 GB de espaço em disco, além de uma placa de vídeo com suporte ao DirectX 9, para que a interface gráfica funcione com os recursos gráficos que o Windows 7 oferece.

Atualmente, os sistemas operacionais da família Windows são os mais usados em computadores de mesa e a família Windows é a que conta com o maior número de programas desenvolvidos para ele. Além de possuir uma grande variedade de *hardwares* compatíveis.



Tela do Windows 7

Windows 7	
Desenvolvedor	Microsoft
Página	http://www.microsoft.com/brasil/windows7/
Lançamento	Outubro de 2009

19

Linux 2.6

O Linux foi anunciado oficialmente pela primeira vez em outubro de 1991, por Linus Torvalds, na versão 0.02, e evoluiu muito desde então. Atualmente, o núcleo Linux encontra-se na versão 2.6.35, segundo o <http://kernel.org/>. Porém como o Linux possui um desenvolvimento muito rápido é provável que, ao ler este livro, a versão do núcleo Linux seja outra mais nova.

O Linux é apenas o núcleo do sistema operacional. Há diversas distribuições que o usam como núcleo. A distribuição mais utilizada atualmente é a Ubuntu (<http://distrowatch.com/>), porém há várias outras distribuições, como: Red Hat, Suse, Debian, Linux Mint, Fedora, Mandriva, OpenSuse, entre várias outras.

Além de possuir diversas distribuições que utilizam o núcleo Linux, há diversas interfaces gráficas que podem ser usadas com Linux. Cada interface apresenta características próprias, algumas tendem a gastar menos recursos enquanto outras primam mais por efeitos visuais e facilidade de uso. Alguns exemplos dessas interfaces são o Gnome, KDE, XFCE e OpenBox.

Atualmente, o Linux possui um bom suporte para *hardware* e apresenta uma grande quantidade de programas. São raros os dispositivos que não funcionam no Linux. Este é muito usado em servidores, instituições de ensino e órgãos do governo brasileiro, porém ainda não foi muito aceito pelos usuários finais.



20

Linux 2.6

Desenvolvedor	Comunidade (Software Livre)
Página	http://kernel.org/
Lançamento	Agosto de 2010 (Versão 2.6.35)

Mac OS X Snow Leopard

O Mac OS X Snow Leopard, ou Mac OS v10.6, é a versão mais recente do sistema operacional da Apple Computer. Essa versão do Mac OS foi lançada em agosto de 2009 e não incluiu grandes novidades se comparada à versão 10.5, chamada de Leopard. As alterações realizadas nessa versão foram a melhoria de desempenho e eficiência no uso de recursos computacionais.

Diferentemente do Windows e do Linux, o Mac OS não pode ser instalado em qualquer computador. Como a Apple, além de desenvolver o Sistema Operacional Mac OS, fabrica computadores, os sistemas operacionais da Apple necessitam de computadores Apple para rodar.

O fato de a Apple produzir o *hardware* e o *software* de seus computadores faz com que o sistema não dependa de *drivers* de terceiros, nem necessite ser compatível com muitos dispositivos diferentes. Estas características acabam melhorando a robustez e a segurança do sistema, além de diminuir o tamanho deste.

O núcleo do Mac OS X, conhecido como Darwin, é certificado UNIX, possui seu código fonte aberto e é baseado em um BSD. O Mac OS X usa uma interface gráfica chamada de Aqua, que possui o foco do seu desenvolvimento, principalmente, na usabilidade e beleza da Interface.



Tela do Mac OS X

21

Mac OS X Snow Leopard

Desenvolvedor	Apple Computer
Página	http://www.apple.com/br/macosx/
Lançamento	Agosto de 2009

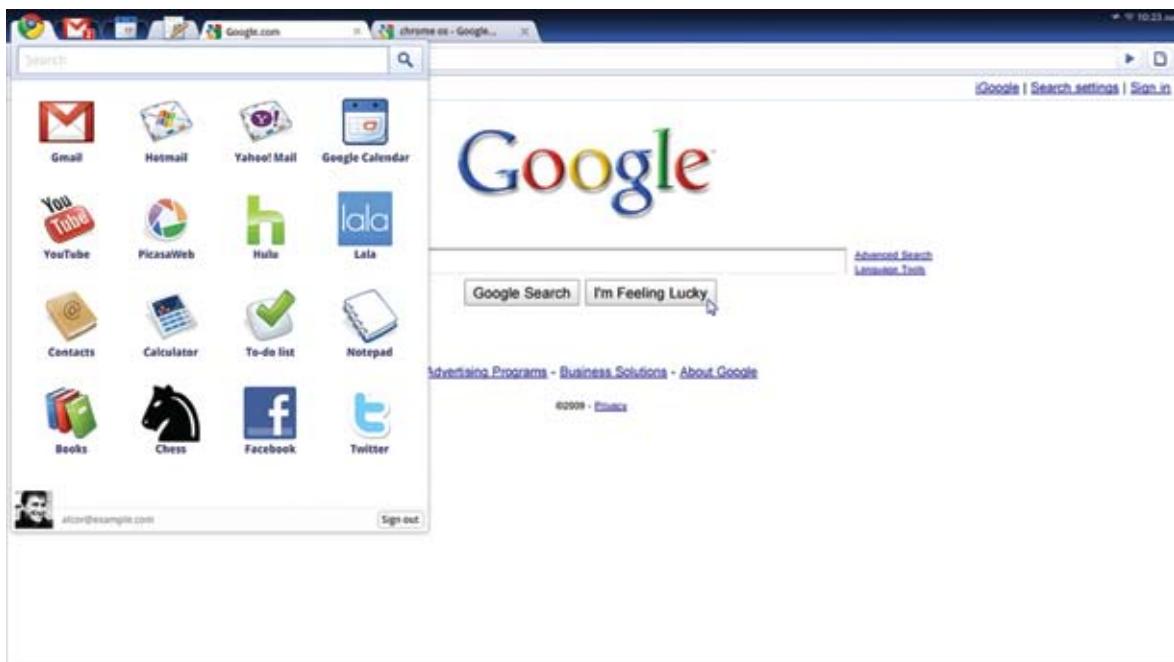
Você Sabia?

Em maio de 2010 a Apple Computer ultrapassou o valor de mercado da sua maior concorrente, a Microsoft. Isto tornou-a a maior empresa de tecnologia do mundo e segunda maior empresa do mundo, perdendo apenas para a Exxon Mobil. A Microsoft foi a maior empresa de tecnologia durante as últimas duas décadas.

Google Chrome OS

A Google está desenvolvendo um sistema operacional chamado de Chrome OS. O núcleo desse sistema é baseado no núcleo Linux, porém o objetivo do sistema é executar apenas um aplicativo, o navegador Google Chrome. Segundo a Google, esse sistema visa atingir usuários que ficam a maior parte do tempo conectados à Internet e usam aplicativos hospedados na Internet.

Como ocorre com o sistema operacional da Apple, o Chrome OS só poderá ser instalado em um *hardware* específico que será construído por parceiros da Google. Além disso, o sistema foi desenhado, principalmente, para rodar em *netbooks*. Como o sistema ainda não foi concluído, pouco se sabe sobre o Chrome OS, há previsão de que seja lançado até o final de 2010.



Tela do Google Chrome OS

Google Chrome OS	
Desenvolvedor	Google Inc.
Página	http://googleblo.blogspot.com/2009/07/introducing-google-chrome-os.html
Lançamento	Final de 2010

Iphone OS 4

O IPhone OS 4 ou IOS 4 é um sistema operacional desenvolvido pela Apple para executar em três famílias de dispositivos da Apple: o *smartphone* IPhone e os dispositivos IPod Touch e IPad. O sistema, que está na sua quarta versão, é baseado no Mac OS X e segue o padrão UNIX. A versão 4 do sistema foi lançado em junho de 2010 e só pode ser instalado nos dispositivos da Apple.

Usando a loja de aplicativos do IPhone OS é possível baixar e instalar vários aplicativos no IOS. Alguns desses aplicativos são desenvolvidos pela própria Apple, outros por outras empresas/usuários.

Na loja chamada de Itunes Store é possível comprar aplicativos ou baixar outros que são gratuitos.

http://commons.wikimedia.org



Aparelho que utiliza o Sistema OS 4

iPhone OS 4

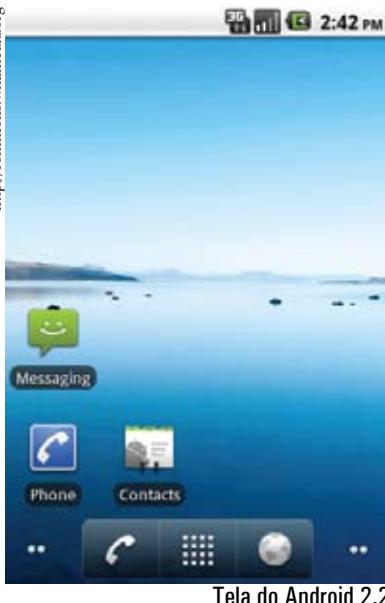
Desenvolvedor	Apple
Página	http://www.apple.com/br/iphone
Lançamento	Junho de 2010

Android 2.2

Mais um sistema operacional baseado no núcleo Linux, o Android é voltado para *smartphones* e dispositivos portáteis como *netbooks* e PDAs, trata-se de um sistema de código fonte aberto que atualmente encontra-se na versão 2.2. Diferente do IPhone OS, o Android pode ser instalado em *hardwares* de diversos fabricantes e, por possuir código aberto, pode ser livremente alterado.

Há diversas empresas envolvidas no projeto do Android, entre elas a Google e a Nokia, além da comunidade que trabalha com códigos *open source* (código aberto). Assim como ocorre com o IPhone OS, o Android também conta com uma loja própria de aplicativos, chamada de Android Market, e possui aplicativos pagos e gratuitos.

http://commons.wikimedia.org



Tela do Android 2.2

Android 2.2

Desenvolvedor	Diversos
Página	http://www.android.com/
Lançamento	Maio de 2010

Symbian^{^3}

O Symbian^{^3}, sim o nome está correto, é circunflexo 3 mesmo, significa Symbian elevado ao cubo. Trata-se de um sistema operacional de código aberto desenvolvido para ser executado em telefones móveis. Há diversos fabricantes de telefones que usam este sistema em seus aparelhos, pois é gratuito.

Apesar de possuir uma interface mais simples que a do Android e do Iphone OS, o Symbian possui as vantagens de necessitar poucos recursos computacionais, como processamento e memória para funcionar. Esta característica faz do Symbian uma alternativa para telefones com menos recursos, ou seja, é mais barato que os concorrentes com Android e IOS.



Aparelho que utiliza o sistema Symbian^{^3}

Symbian ^{^3}	
Desenvolvedor	Symbian Foundation
Página	http://www.symbian.org
Lançamento	Maio de 2002

Windows Mobile 6.5

O Windows Mobile é a versão do Windows para dispositivos móveis como *smartphones* e PDAs. Atualmente, encontra-se na versão 6.5 e possui um conjunto de aplicativos comuns aos do Windows para computadores, tais como Word, Excel, PowerPoint e Windows Media Player.

O Windows Mobile não possui uma grande presença no mercado de dispositivos móveis. Está presente em aparelhos de empresas como HTC, LG e Samsung.

http://commons.wikimedia.org



Dispositivo móvel que utiliza o Windows Mobile 6.5

Windows Mobile 6.5

Desenvolvedor	Microsoft
Página	http://www.microsoft.com
Lançamento	Fevereiro de 2009



Atividades

- 1) Faça uma pesquisa e levante os requisitos de *hardware* necessários para instalar os sistemas operacionais que você conheceu neste capítulo.
- 2) Verifique, junto com o professor a possibilidade de utilizar os sistemas operacionais dos celulares por meio de emuladores (*softwares* que reproduzem as funções de um determinado ambiente). Há vários tutoriais na Internet que mostram como testar o Android, por exemplo.
- 3) No Linux é possível utilizar e configurar vários ambientes gráficos diferentes. Verifique quais os principais ambientes utilizados. Use, para auxiliar na pesquisa, o termo “gerenciador de janelas”.

Estudo de Caso de Sistemas Operacionais

Utilizando o Linux

Como explicado no primeiro capítulo, utilizaremos como estudo de caso o sistema operacional Linux para compreendermos os conceitos relativos a sistemas operacionais. Essa escolha se deve por ser um sistema aberto, livre, disponível para uso e amplamente utilizado no governo, nas indústrias e em várias outras organizações.

26

O que é Linux

Linux é o termo geralmente usado para designar qualquer sistema operacional que utilize o núcleo Linux. Foi desenvolvido pelo finlandês Linus Torvalds, inspirado no sistema Minix. O seu código fonte está disponível sob licença GPL (*General Public License* – Licença Pública Geral), para qualquer pessoa que o utilizar, estudar, modificar e distribuir, de acordo com os termos da licença.

Inicialmente desenvolvido e utilizado por grupos de entusiastas em computadores pessoais, o sistema Linux passou a ter a colaboração de grandes empresas, como a IBM, a Sun Microsystems, a Hewlett-Packard, a Red Hat, a Novell, a Google e a Canonical.

Você Sabia?

GPL – *General Public License* (Licença Pública Geral), GNU GPL ou simplesmente GPL, é a designação da licença para software livre idealizada por Richard Stallman no final da década de 80 do século vinte, no âmbito do projecto GNU da *Free Software Foundation* (FSF).

A GPL é a licença com maior utilização por parte de projetos de software livre, em grande parte devido à sua adoção para o projeto GNU e o sistema operacional GNU/Linux.

História

O *kernel* Linux foi, originalmente, escrito por Linus Torvalds, do Departamento de Ciência da Computação da Universidade de Helsinki, Finlândia, com a ajuda de vários programadores voluntários, por meio da Usenet (uma espécie de sistema de listas de discussão existente desde os primórdios da Internet).

Linus Torvalds começou o desenvolvimento do *kernel* como um projeto particular, inspirado pelo seu interesse no Minix, um pequeno sistema UNIX, desenvolvido por Andrew S. Tanenbaum. Ele limitou-se a criar, segundo suas próprias palavras, “um Minix melhor que o Minix” (*a better Minix than Minix*). E, depois de algum tempo de trabalho no projeto, sozinho, anunciou a primeira versão “oficial” do *kernel* Linux, versão 0.02 no dia 5 de outubro de 1991.

Desde então, muitos programadores têm respondido ao seu chamado e têm ajudado a fazer do Linux o sistema operacional que é atualmente. No início, era utilizado por programadores ou só por quem tinha conhecimentos avançados em informática (configuração manual de dispositivos), que usava linhas de comando. Na atualidade, isso mudou. Há diversas empresas que criam os ambientes gráficos, com distribuições cada vez mais amigáveis, de forma que mesmo uma pessoa com poucos conhecimentos consegue usar o Linux. Este é, atualmente, um sistema estável e consegue reconhecer todos os periféricos de forma automatizada (em raros casos é necessária a interferência do usuário para configuração personalizada).

Curiosamente, o nome Linux foi criado por Ari Lemmke, administrador do site <http://ftp.funet.fi/> que deu esse nome ao diretório FTP, local onde o *kernel* Linux estava disponível. Linus tinha-o batizado como “Freak” inicialmente.



Linus Torvalds

http://pt.wikipedia.org

27

Características do Sistema Linux

O sistema operacional Linux apresenta as seguintes características:

- **Interativo** – o usuário requisita os comandos e obtém os resultados de sua execução por meio do terminal.
- **Multitarefa** – um único usuário pode requisitar que sejam efetuados vários comandos ao mesmo tempo em seu terminal. É responsabilidade do sistema operacional controlar estas execuções paralelas. Quando um usuário executa mais de um comando ao mesmo tempo, somente um necessita a interação com o usuário. Os demais comandos executados são, em sua maioria, comandos que não exigem atenção, sendo tarefas demoradas. Quando isto ocorre, dizemos que os programas que o usuário está executando, sem interação, ficam em *background*. Aquele que está sendo executado com interação fica em *foreground*.

- **Multusuário** – o sistema operacional pode controlar o acesso ao sistema de vários terminais, virtuais ou reais, cada um pertencendo a um usuário. O Linux aceita as requisições de comandos de cada um dos usuários e gera as filas de controle e prioridades para que haja uma distribuição correta dos recursos de *hardware* necessários a cada um. Devido à característica de sistema multusuário, o sistema operacional implementa um método de segurança, visando a impedir o acesso aos arquivos e diretórios de um usuário por outro.

Atividades

- 1) Para verificar como funciona a execução de programas em *background* e *foreground*, com a ajuda do professor utilize os seguintes comandos no terminal do Linux:
 - man python – Exibe a ajuda do interpretador python.
 - pressione Ctrl + X – Isto coloca o processo man em segundo plano.
 - bg – Este comando exibe os processos que estão em segundo plano.
 - fg 1 – coloca novamente o processo man em primeiro plano.
 - q <Enter> – fecha o processo man.
- 2) Para demonstrar como o Linux suporta vários usuários ao mesmo tempo, execute os seguintes comandos:
 - Control + Alt + F1 – Inicia uma nova seção em modo texto.
 - Nesta seção efetue *login* com um usuário.
 - Control + Alt + F2 – Inicia outra nova seção em modo texto.
 - Nesta seção efetue *login* com outro usuário.
 - Neste momento, há dois usuários no sistema.

28

Alguns termos e conceitos, que serão muito utilizados nas explicações e textos, devem ser conhecidos, pois toda a bibliografia e documentação os utiliza. Esses termos fazem parte do jargão do Linux, são eles:

- **Shell** – é o termo para definir o interpretador de comandos. O shell nada mais é que um programa que recebe os comandos do usuário e ativa o sistema operacional. Ele faz o controle do terminal, tanto na entrada como na saída. Há várias opções de shell para o usuário, cada um com determinadas características e facilidades. Entre os mais conhecidos podemos citar: **bash**, **sh**, **rsh**, **csh** e **ksh**. No Linux, o shell padrão é o **bash**.
- **Kernel** – é como é chamado o núcleo do sistema Linux. Esse núcleo faz o gerenciamento direto dos dispositivos de E/S (*device drivers*), gerenciamento de memória e controle do uso da CPU pelos vários processos do sistema.
- **Comando** – um comando Linux nada mais é que um arquivo (programa executável) guardado em um diretório específico do sistema. Portanto, quando o usuário executa um comando, ele simplesmente está rodando um programa como qualquer outro do sistema.



- **Processo** – é um conceito básico do sistema. Toda vez que se executa um programa/comando é gerado um processo no sistema. Todo gerenciamento é feito sobre ele. Os processos são, portanto, comandos/programas em execução. Todo processo é identificado por um número chamado *Process ID* (PID). Esse ID é único no sistema durante a execução do processo, portanto pode e deve ser usado para identificá-lo em caso de necessidade, como em caso de remoção desse processo da memória.

Usuários

Dentro do sistema há dois tipos de usuários: comum e superusuário. O usuário comum é aquele que tem acesso limitado somente a seus dados e arquivos. Se tentar acessar dados de outro usuário, o sistema, dependendo das permissões configuradas, não permitirá, emitindo uma mensagem de erro.

O superusuário ou conta *root* é uma conta com poderes supremos sobre toda a máquina. Com ela pode-se acessar qualquer arquivo que se encontra na máquina, removê-lo, mudá-lo de lugar, etc. O esquema de permissão e segurança do Linux não se aplica ao superusuário.

Há também, certos comandos que só podem ser executados quando o usuário tem permissão de superusuário. Esses comandos geralmente servem para a manutenção do sistema e não devem ser deixados à disposição de usuários comuns devido à complexidade e perigo do mau uso desses comandos.



Atividade

29

Para demonstrar as diferenças entre usuários comuns e superusuários tente executar o comando **`shutdown -h now`**, que desliga o computador como um usuário normal. Depois, se possível e com a ajuda do seu professor, execute o mesmo comando como superusuário.

Comandos

No Linux, as aplicações são também conhecidas como comandos. Todo comando está armazenado em um formato binário (executável) que é interpretado pelo sistema operacional.

O formato geral de um comando: **comando [opções] [argumentos]** (o que está entre colchetes não é obrigatório).

- **Comando** – comando ou programa a ser executado.
- **Opções** – modificadores do comando (opcional).
- **Argumentos** – define o objeto a ser afetado pelo comando (opcional).

A maioria dos comandos Linux apresentam a sintaxe compatível ao formato acima. Temos o nome do comando, seguido de opções e argumentos. As opções, quando colocadas, devem sempre preceder os argumentos.

Veja o exemplo a seguir:

```
$ wc /etc/passwd  
32    53 1570 /etc/passwd  
  
$ wc -l /etc/passwd  
32 /etc/passwd
```

O comando **wc** é utilizado para contar quantas linhas, palavras e caracteres tem um arquivo. O argumento **/etc/passwd** é o nome do arquivo que está sendo analisado pelo comando **wc**. O comportamento padrão do comando pode ser observado no primeiro exemplo. No segundo exemplo, ao se informar a opção **-l** estamos indicando que o comando **wc** deve apenas contar a quantidade de linhas do arquivo **/etc/passwd**.

Observe que os caracteres separadores dos campos da linha de comando são o espaço em branco e o <TAB>. Um outro detalhe, muito importante, é o fato de que o Linux faz distinção entre os caracteres maiúsculos e minúsculos. Portanto, para o Linux, **Wc** é diferente de **wc**.

Quase sempre as opções dos comandos são precedidas pelos caracteres “-” (menos) ou “+” (mais) e podem entrar em qualquer ordem e posição na linha de comando, mas sempre antes dos argumentos (há poucas exceções). Na maioria das vezes, as opções são representadas por letras, podendo-se agrupar uma série delas em uma única opção. Por exemplo, as opções **“-w -l -c”** do comando **wc** podem ser escritas como **“-wlc”**. Há, também, opções que são mutuamente exclusivas, não podendo aparecer ao mesmo tempo em um comando.

O terceiro tipo de opção que pode existir em um comando é a opção que exige, logo após, um argumento específico. Neste caso, quase sempre essa opção é colocada separada, precedida por “-” ou “+” e seguida de seu argumento. Caso ela seja colocada juntamente com as demais opções, ela deve ser a última da lista. Veja o exemplo abaixo:

```
$ cut -d : -f 1 /etc/passwd  
root  
daemon  
bin  
sys  
sync  
----- resultado suprimido propositalmente ----->
```

Em que:

cut – comando;

-d – opção do comando;

: – argumento da opção **-d**;

-f – opção do comando;

1 – argumento da opção **-f**;

/etc/passwd – argumento do comando **cut**.

Um detalhe que gera muita confusão para o iniciante do sistema Linux é o fato de que as opções variam de comando para comando, tornando difícil uma memorização delas. Se isso acontecer com você, não se preocupe, pois poucas pessoas sabem todas as opções de todos os comandos. Os argumentos definem os objetos sobre os quais o comando será aplicado. Temos como exemplos de argumentos: arquivos, periféricos, etc.



Atividade

No shell execute o seguinte comando:

- `cd ~`
- `ls`

Depois, usando a interface gráfica, entre no seu diretório home e compare os arquivos exibidos na interface gráfica, com os exibidos no shell. Eles são os mesmos?

Principais Diretórios

Os diretórios de um sistema de arquivos têm uma estrutura predefinida, com poucas variações. A seguir, ilustramos os principais:

- **/home**: raiz dos diretórios home dos usuários;
- **/boot**: arquivos de boot (*kernel* do sistema, etc.);
- **/var**: arquivos variáveis, áreas de spool (impressão, e-mail, *news*), arquivos de log;
- **/etc**: arquivos de configuração dos serviços;
- **/usr**: aplicações voltadas aos usuários;
- **/tmp**: arquivos temporários;
- **/mnt**: montagem de diretórios compartilhados temporários;
- **/bin**: aplicações de base para o sistema;
- **/dev**: arquivos de acesso aos dispositivos físicos e conexões de rede;
- **/lib**: bibliotecas básicas do sistema.

31

O Diretório Home

Cada usuário possui um diretório especial, chamado “diretório home” (casa), onde são armazenados:

- arquivos e diretórios pessoais de trabalho;
- e-mails já lidos (pastas pessoais);
- arquivos de configuração individuais;
- configuração das aplicações usadas.

O diretório home do usuário é o seu local de início de sessão de trabalho (via shell ou gráfica). O usuário possui plenos poderes de acesso ao seu diretório home (e seus subdiretórios) e, normalmente, não pode criar arquivos fora dele. O diretório home de cada usuário é, comumente, inacessível aos outros usuários, mas isso pode ser controlado pelo administrador do sistema (root).

Interpretador de Comandos

Também conhecido como shell. É a principal ligação entre o usuário, os programas e o *kernel*. O Linux apresenta diversos tipos de interpretadores de comandos, entre eles podemos destacar o **bash**, **ash**, **csh**, **tcs**, **sh**, etc. Destes, o mais usado é o **bash** (no Linux). O interpretador de comandos do DOS, por exemplo, é o **command.com**.

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

- **Interativa** – os comandos são digitados no aviso de comando e passados ao interpretador de comandos, um a um. Neste modo, o computador depende do usuário para executar uma tarefa ou o próximo comando.
- **Não interativa** – são usados arquivos de comandos criados pelo usuário (*scripts*) para o computador executar os comandos na ordem encontrada no arquivo. Neste modo, o computador executa os comandos do arquivo um por um e, dependendo do término do comando, o *script* pode checar qual será o próximo comando que será executado e dar continuidade ao processamento. Este sistema é útil quando temos que digitar por várias vezes seguidas um mesmo comando ou para compilar algum programa complexo.

O **shell bash** possui, ainda, outra característica interessante: a complementação dos nomes. Isto é feito pressionando-se a tecla <TAB>. Por exemplo, se digitar “ls tes” e pressionar <TAB>, o **bash** localizará todos os arquivos que iniciam com “tes” e completará o restante do nome. Caso a complementação de nomes encontre mais do que uma expressão que satisfaça a pesquisa, ou nenhuma, é emitido um beep. A complementação de nomes funciona sem problemas para comandos internos.

Exemplos:

- **ech** (pressione <TAB>) (resultado esperado “echo”)
- **ls /vm** (pressione <TAB>) (resultado esperado “ls /vmlinuz”)



Comandos Básicos

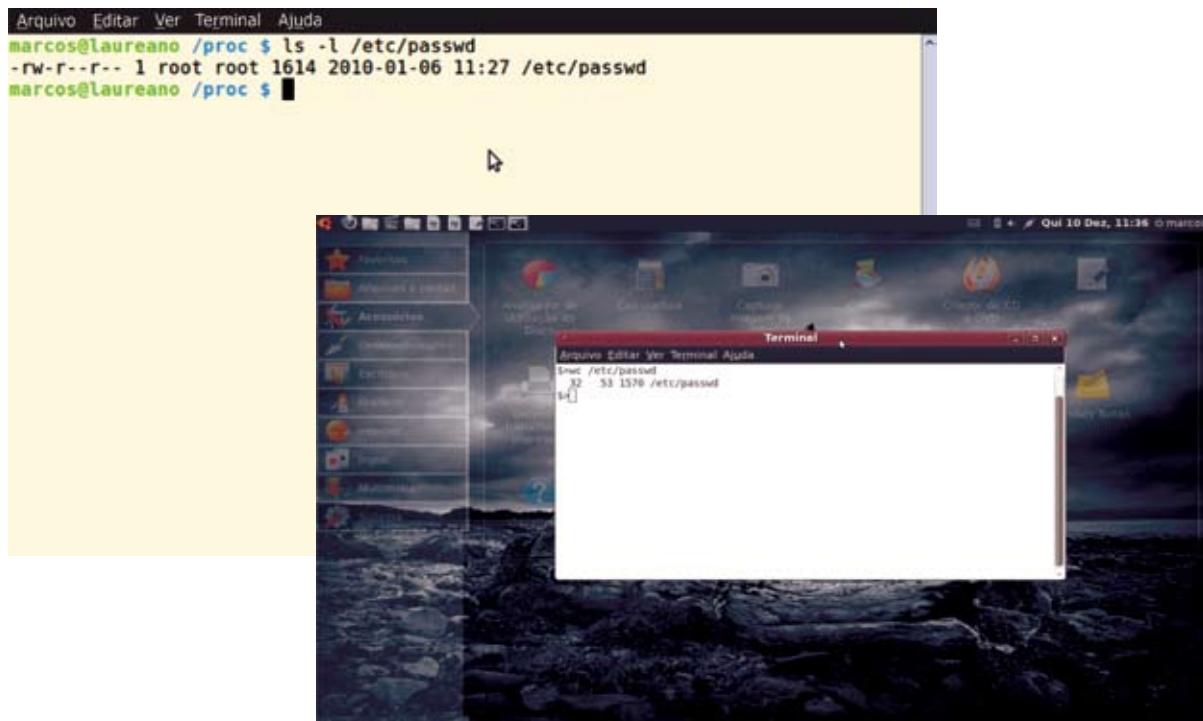
Agora, iremos aprender alguns comandos básicos que, além de possibilitar o domínio do Linux, irá ajudá-lo no aprendizado dos conceitos de sistema operacional.

Iniciando uma Sessão

O acesso ao sistema é feito por meio de uma conta previamente cadastrada no sistema pelo administrador deste. Sem essa conta, chamada **user name** ou **login name**, não é possível o acesso. O nome da conta deve ser uma *string* – sequência de caracteres – de até oito caracteres e deve ser única em cada máquina.

Junto com a conta é atribuída uma senha de acesso. A senha garante que não se faça nenhum acesso indevido aos dados de determinado usuário. A manutenção dessa senha e de sua validade é de responsabilidade única do usuário.

Uma vez acertadas a conta e a senha, o sistema apresentará um **prompt** (veja a figura de exemplo), indicando que ele está apto a receber comandos do usuário. Neste ponto dizemos que estamos com uma sessão aberta.



Comando exit

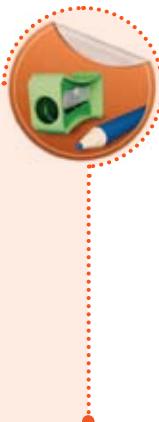
O comando **exit** deve ser utilizado para encerrar uma sessão de trabalho, quando é utilizado o modo texto. Terminando a conexão com o usuário, o sistema volta a pedir uma nova conta.

Quando se está utilizando terminal no modo gráfico, o comando **exit** fecha o terminal, mas não finaliza a conexão. Entenda por ambiente gráfico os gerenciadores de tela disponíveis nos sistemas Unix/Linux, tais como: Gnome, KDE, XFCE, entre outros.

Um fato importante a relatar é que a sessão (dependendo do shell) pode terminar se for digitado um <CTRL> D, pois este caractere significa fim de arquivo ou fim da entrada de dados.

Atividades

- 1) Verifique se o *login* de acesso e a senha, fornecidos pelo professor, estão corretos.
- 2) Teste o comando **exit** e verifique o que acontece.



Obtendo Help no Sistema – Páginas de Manual

34

As páginas de manual acompanham todos os sistemas Linux. Elas trazem uma descrição básica do comando/programa e detalhes sobre o funcionamento de uma opção. Uma página de manual é visualizada na forma de texto único com rolagem vertical. Também documenta parâmetros usados em alguns arquivos de configuração.

A utilização da página de manual é simples, digite:

```
$ man [opções] [seção] [título]
```



A seção, aqui, indica ao comando qual a seção do manual que será aberta; se omitida, mostra a primeira seção encontrada sobre o comando (em ordem crescente). O título é o nome do comando ou arquivo que se está buscando.

Cada seção da página de manual contém explicações sobre uma determinada parte do sistema. As seções são organizadas em diretórios separados e localizadas no diretório **/usr/man**.

Os programas e arquivos são classificados nas seguintes seções:

1. Programas executáveis ou comandos internos (comandos do shell).
2. Chamadas do sistema (funções oferecidas pelo *kernel*).
3. Chamadas de bibliotecas (funções dentro de bibliotecas do sistema).
4. Arquivos especiais (normalmente encontrados no diretório **/dev**).
5. Formatos de arquivos e convenções (**/etc/inittab**, por exemplo).
6. Jogos.
7. Pacotes de macros e convenções (por exemplo, **man**).
8. Comandos de administração do sistema (normalmente usados pelo *root*).
9. Rotinas do *kernel* (não padrões).

Na figura abaixo é visto a execução do comando **man ls**

```
Arquivo Editar Ver Terminal Ajuda
LS(1)                                         User Commands                                         LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILEs (the current directory by default). Sort entries alphabetically
    if none of -cftuvSUX nor --sort.

    Mandatory arguments to long options are mandatory for short options too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        do not list implied . and ..

    --author
        with -l, print the author of each file

    -b, --escape
        print octal escapes for nongraphic characters

    --block-size=SIZE
        use SIZE-byte blocks
Manual page ls(1) line 1
```

35

Um detalhe deve ser observado, o fato de que a página de manual é dividida em algumas partes comuns:

- NAME – nome do comando com breve descrição.
- SYNOPSIS – sintaxe do comando com opções e argumentos.
- DESCRIPTION – descrição detalhada do comando.
- SEE ALSO – na grande maioria dos casos um comando está ligado à execução de diversos outros. Nesta parte do manual são colocados todos os comandos citados no texto anterior ou que têm alguma relação com o comando.

O manual é composto de outras partes menos importantes. Outro detalhe: o número da seção sempre aparece entre parênteses “()” logo após o nome do comando. No nosso exemplo, aparece LS(1).

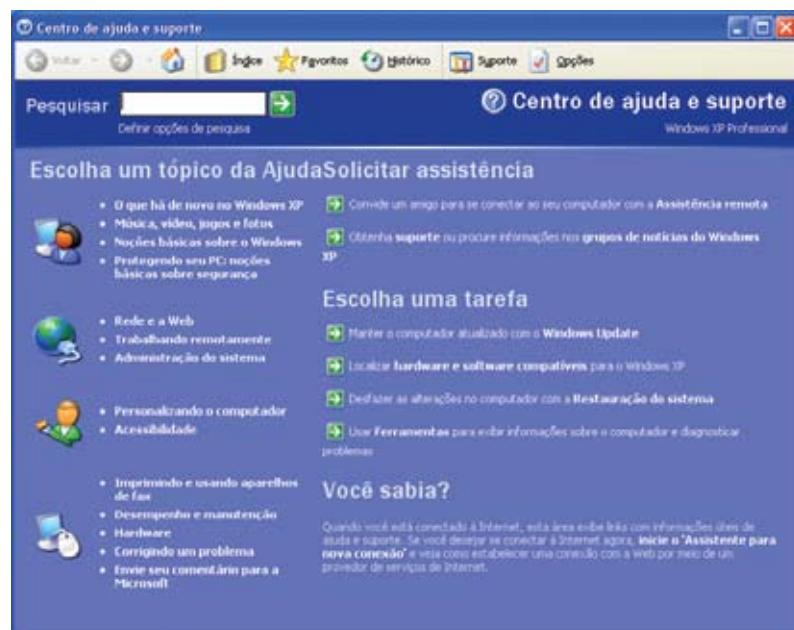
A documentação de um programa também pode ser encontrada em duas ou mais categorias, como é o caso do comando **printf** que é documentado na seção 1 (comando do shell) e 3 (função de biblioteca). Por este motivo é necessário digitar **man 3 printf** para ler a página sobre o formato do arquivo, porque o comando **man** procura a página de manual nas seções em ordem crescente e a digitação do comando **man printf** abriria a seção 1.

A opção **-k** faz com que o comando **man** apresente diversas opções de busca em função da palavra informada. Por exemplo, o comando **man -k bash** produz o seguinte resultado:

```
$ man -k bash
bash (1)           - GNU Bourne-Again SHell
bash-builtins (7)   - bash built-in commands, see bash(1)
bashbug (1)         - report a bug in bash
builtins (7)        - bash built-in commands, see bash(1)
dh_bash-completion (1) - install bash completions for package
rbash (1)           - restricted bash, see bash(1)
```

Você Sabia?

O sistema de ajuda (*help*) é diferente em cada sistema operacional. No sistema Windows o acesso à ajuda ocorre pressionando a tecla F1 e, depois, digitando uma palavra chave para a pesquisa.





Atividades

- 1) Qual a função do comando **ls**? Quais são suas opções e o que representa cada uma delas? Quais argumentos este comando aceita?
- 2) Qual a função da opção **-u** do comando **date**?
- 3) Quais são as informações constantes do arquivo **passwd**?
- 4) Qual é o comando correlato ao comando **mesg**?

Comando passwd

Para efetuar a troca de senha, inicialmente o comando pede, por medida de segurança, que seja digitada a senha atual da conta. Em seguida, pede que seja digitada a nova senha. Por último, pede que seja digitada novamente a nova senha para verificação. Em todas estas etapas as senhas digitadas não são apresentadas na tela por medida de segurança. A senha só será trocada se a atual conferir com a cadastrada no sistema e a nova for digitada igual na confirmação.

Algumas regras para criação de senhas:

- A senha deve ter, no mínimo, seis caracteres.
- A senha deve ter, no mínimo, duas letras maiúsculas e/ou duas letras minúsculas e, pelo menos, um dígito ou caractere especial.
- São aceitos somente os caracteres ASCII padrão de códigos 0 a 127.
- A senha deve ser diferente do nome da conta.
- A nova senha deve ser diferente da senha velha em pelo menos três caracteres.

37

Atividade

Mude a senha do seu *login* de acesso. Não a divulgue para ninguém. Não se preocupe no caso de esquecer a senha, o professor poderá modificá-la novamente para você.

Comando expr

O uso comum do **expr** no terminal é utilizado para se fazer cálculos simples de adição, subtração, divisão inteira e multiplicação. Ele apresenta a restrição de que se deve sempre colocar uma barra invertida antes de algumas operações ou parênteses (exceto + e -) para que o shell não interprete estes caracteres. Veja o exemplo:

```
$ expr 14 + \(\ 10 \ * \ 4 \ \)
```

```
54
```

```
$ expr 5 \> 1
```

```
1
```

```
$ expr 5 \< 1
```

```
0
```

38

Atividades

- 1) Qual é o resultado de $(30 - 5) * 789$?
- 2) Qual é o resultado de $(20 / 4) * 15 + 4$?
- 3) Qual o resultado para $(5 * 10) + 10 * 7$?
- 4) Qual o resultado para $5 > 6$?
- 5) Qual o resultado para $5 < 6$?
- 6) Qual o resultado para $5 = 5$?
- 7) Qual o resultado para $10 + 20 + (30 * 5) - 50$?

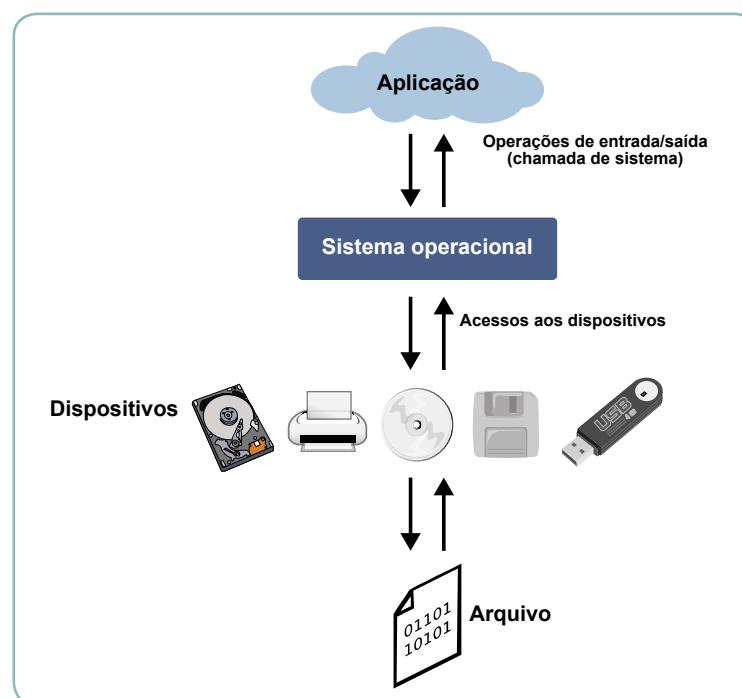


Sistema de Arquivos

O sistema de arquivos é o modo como o sistema operacional organiza o acesso aos arquivos e pastas no computador. Para ser criado, o disco deve passar por uma formatação, que cria a forma e o prepara para receber leitura/gravação. Normalmente, este passo é feito durante a instalação do sistema operacional. Cada sistema de arquivos tem uma característica em particular, mas, em geral, seu propósito é criar a abstração necessária para dar acesso físico aos dispositivos e permitir que sejam gravadas/lidas informações nele.

O sistema de arquivos é criado durante a formatação da partição de disco. Após essa formatação, toda a estrutura para leitura/gravação de arquivos e diretórios pelo sistema operacional estará pronta para ser usada. Normalmente, este passo é feito durante a instalação do sistema operacional. Cada sistema de arquivos tem uma característica em particular, mas seu propósito é o mesmo: oferecer ao sistema operacional a estrutura necessária para ler/gravar os arquivos/diretórios.

O sistema de arquivos é o responsável pela gerência dos arquivos. Os arquivos são armazenados de maneira uniforme e independente dos diferentes dispositivos de armazenamento (veja a figura a seguir).



Partições

São divisões existentes no disco rígido que marcam onde começa e onde termina um sistema de arquivos. Por causa dessas divisões, pode-se usar mais de um sistema operacional no mesmo computador (como o Linux ou Windows), ou dividir o disco rígido em uma ou mais partes para ser usado por um único sistema operacional.

Para gravar os dados, o disco rígido deve ser primeiro particionado (usando o comando **fdisk** ou similar), depois deve ser escolhido o tipo da partição (Linux Native, Linux Swap, etc.) e, por fim, a partição deve ser formatada (preparada para receber os arquivos e diretórios).

Para partitionar (dividir) o disco rígido em uma ou mais partes é necessário o uso de um programa de particionamento. Os programas mais conhecidos para particionamento de discos no Linux são os comandos **fdisk**, **cfdisk** ou o utilitário GParted (<http://gparted.sourceforge.net/>).

Identificação de Discos e Partições em Sistemas Linux

Depois de criada e formatada, a partição será identificada como um dispositivo no diretório **/dev** e deverá ser montada para permitir seu uso no sistema. Uma partição de disco não interfere em outras partições existentes, por este motivo é possível usar o Windows, Linux e qualquer outro sistema operacional no mesmo disco.

No Linux, os dispositivos existentes no computador (como discos rígidos, disquetes, monitor, portas de impressora, *modem*, etc.) são identificados por um arquivo referente a esses dispositivos no diretório **/dev**.

A identificação de discos rígidos no Linux é feita da seguinte forma:

```
/dev/hda1
|   |   |
|   |   |_Número que identifica o número da partição no disco rígido.
|   |
|   |_Letra que identifica o disco rígido (a=primeiro,
|       b=segundo, etc...).
|
|   |_ Sigla que identifica o tipo do disco rígido (hd=ide,
|       sd=SCSI, xt=XT).
|
_|_Diretório onde são armazenados os dispositivos existentes no sistema.
```



A partição EXT3 (*Third Extended Filesystem*) é usada para criar o sistema de arquivos Linux Native que serve para armazenar o sistema de arquivos EXT3 (após a formatação) e permitir o armazenamento de dados.

Logo que foi inventado, o Linux utilizava o sistema de arquivos Minix (e consequentemente uma partição Minix) para o armazenamento de arquivos. Com a evolução, foi criado o padrão EXT (*Extended Filesystem*) e logo o EXT2 (*Second Extended Filesystem*), que ainda é usado atualmente.

O sistema EXT3 implementa o conceito de *journaling*. O sistema de *journaling* grava qualquer operação que seja feita no disco em uma área especial chamada *journal*, assim, se acontecer algum problema durante a operação de disco, ele pode voltar ao estado anterior do arquivo, ou finalizar a operação.

Desta forma, o *journal* acrescenta ao sistema de arquivos o suporte da alta disponibilidade e da maior tolerância a falhas. Após uma falta de energia, por exemplo, o *journal* é analisado durante a montagem do sistema de arquivos e todas as operações que estavam sendo feitas no disco são verificadas. Dependendo do estado das operações, elas podem ser desfeitas ou finalizadas. O retorno do servidor é praticamente imediato, garantindo a rápida retomada dos serviços da máquina.

Outra situação que pode ser evitada é a inconsistência no sistema de arquivos do servidor após a situação acima, fazendo-o ficar em estado *single user* (usuário único), esperando pela intervenção do administrador.

Arquivos

Os arquivos são gerenciados pelo sistema operacional e é mediante a implementação deles que o sistema operacional estrutura e organiza as informações. Um arquivo é constituído de informações logicamente relacionadas, podendo representar programas ou dados.

Os arquivos são identificados por meio de um nome formado por uma sequência de caracteres. A identificação de um arquivo é composta por duas partes separadas por um ponto. A parte após o ponto é chamada extensão do arquivo e “serve” para identificar o conteúdo. Exemplos:

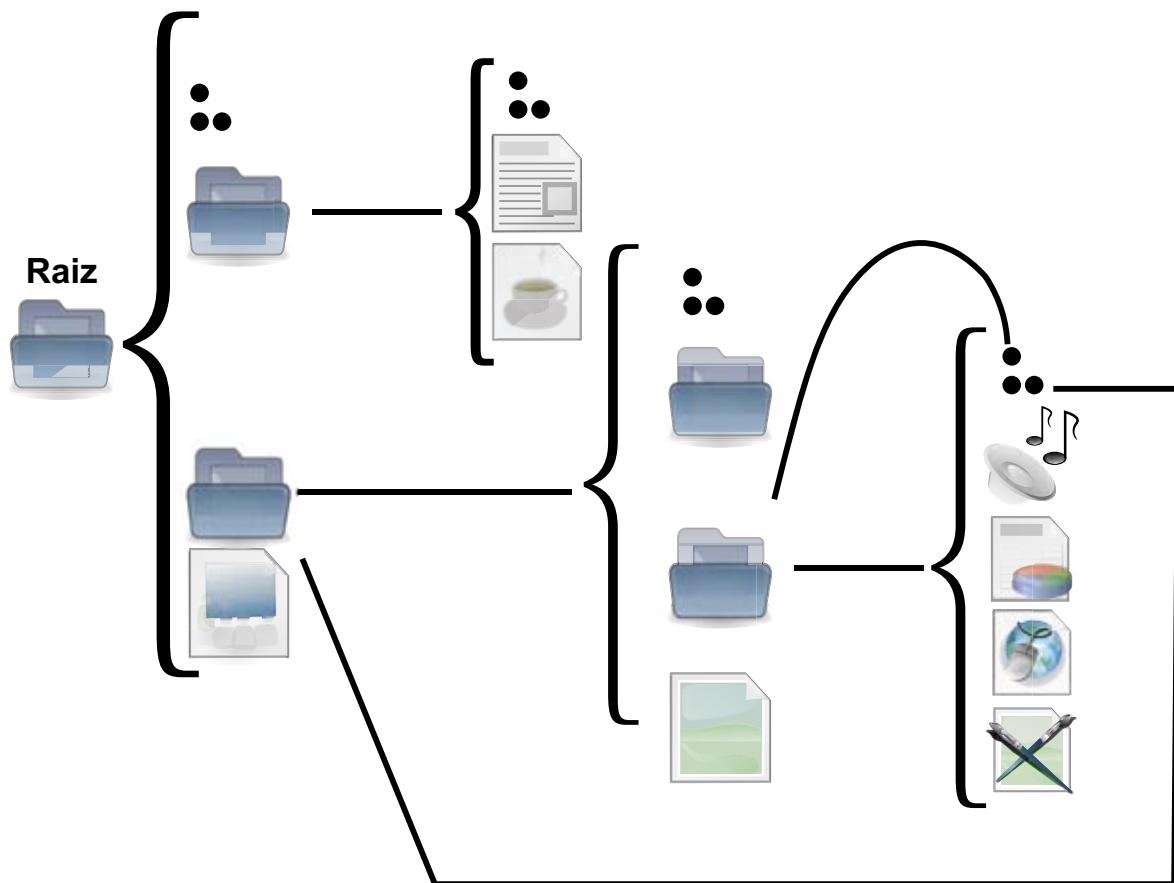
- **arquivo.c** – Arquivo fonte em C;
- **arquivo.mp3** – Arquivo de música;
- **arquivo.dll** – Biblioteca dinâmica.

Diretórios

É o modo como o sistema organiza os diferentes arquivos contidos em um disco. Utilizando um método relativamente simples, o diretório é implementado como um arquivo estruturado, cujo conteúdo é uma relação de entradas na forma de uma estrutura de dados. Cada estrutura contém entradas associadas aos arquivos onde estão informações, como localização física, nome, organização e demais atributos.

Os tipos de entradas normalmente consideradas nessa relação são arquivos normais, diretórios, atalhos e entradas associadas a arquivos especiais. Cada entrada contém ao menos o nome do arquivo (ou do diretório), seu tipo e a localização física dele na partição.

O Linux utiliza algumas entradas padronizadas. A entrada “.” (ponto), que representa o próprio diretório, e a entrada “..” (ponto-ponto), que representa seu diretório pai (o diretório imediatamente acima dele na hierarquia de diretórios). No caso do diretório raiz, ambas as entradas apontam para ele próprio. Veja a figura a seguir.



Diretórios Especiais “.” e “..”

Toda vez que um diretório é criado, sempre são criadas duas entradas nele. Uma entrada, com o nome de “.”, referencia-se ao próprio diretório criado e a outra entrada, com o nome de “..”, uma referência ao diretório anterior, ou diretório pai, na estrutura do sistema de arquivos.

Estes dois arquivos podem ser usados para compor qualquer caminho, relativo ou absoluto, dentro dos comandos do Linux, e visam a facilitar a digitação de comandos.

Normalmente, eles não aparecem na relação de arquivos, pois o sistema esconde todos os arquivos que começam com um ponto em seu nome. Para que se listem estas duas entradas, devemos usar a opção **-a** do comando **ls**.

Caminho de Arquivos

O sistema de arquivos é apresentado como uma única hierarquia unificada que se inicia no diretório / e prossegue abaixo até um número arbitrário de subdiretórios. O diretório / também é chamado de diretório-raiz.

Sempre que precisamos localizar um arquivo para qualquer operação (ler, gravar, remover, criar, etc.) o sistema operacional deve conhecer em que ponto do sistema de arquivos ele se encontra. Isto é feito por meio da especificação de um caminho antes do nome do arquivo. Esse caminho, chamado de **path**, pode ser indicado de duas maneiras:

- **Absoluto** – o caminho absoluto sempre começa com uma barra /. Esse caminho dá a localização do arquivo desde o diretório-raiz do sistema. O sistema operacional começa pela raiz e vai seguindo os diretórios indicados até o último.
- **Relativo** – a procura de um arquivo por meio de um caminho relativo começa no próprio diretório atual da sessão.

Comandos Básicos para Trabalhar com Diretórios

Os comandos usados para navegação na árvore de diretórios são similares aos usados em outros sistemas operacionais:

- **pwd**: indica qual o diretório corrente do shell.
- **cd**: troca de diretório.
- **cd dir**: muda para o diretório dir.
- **cd ..**: muda para o diretório pai imediatamente superior.
- **cd -**: volta para o último diretório visitado.
- **cd**: volta ao diretório HOME.

Imprime o nome do diretório corrente.

```
$ pwd  
/home/marcos
```

Acessa o diretório /etc (utilizando caminho absoluto, ou seja, utiliza o /).

```
$ cd /etc
```

Imprime o nome do diretório corrente.

```
$ pwd  
/etc
```

Acessa o diretório apt e imprime o nome do diretório corrente.

```
cd apt  
$ pwd  
/etc/apt
```

Acessa o diretório home (comando **cd** sem argumentos) e imprime o nome do diretório corrente.

```
$ cd  
$ pwd  
/home/marcos
```

Acessa o diretório anterior (comando **cd** com argumento -).

```
$ cd -  
/etc/apt
```

Acessa o diretório anterior utilizando caminho relativo.

```
$ cd ..  
$ pwd  
/home
```

Acessa o diretório/etc (utilizando caminho relativo, ou seja, informando o caminho a partir da posição atual).

```
$ cd ../etc  
$ pwd  
/etc
```

Caminho relativo ou absoluto, qual a melhor opção?

Considere a árvore de diretório abaixo.

```
\  
|-- bin  
|-- boot  
|  |-- boot  
|  |  `-- grub  
|  '-- grub  
|-- etc  
|  |-- X11  
|  |-- acpi  
|  |-- alternatives  
|  '-- apm  
|-- home  
|  '-- marcos  
|    |-- Documentos  
|    |-- Downloads  
|    |-- Imagens  
|    |-- Modelos  
|    '-- uml  
|-- lib  
|-- mnt  
`-- opt
```

Suponha que você está posicionado no diretório **Documentos** e deseja acessar o diretório **Downloads**. Utilizando caminho relativo, você usaria o seguinte comando:

cd .../Downloads (indica que você quer descer um nível e subir para o diretório Downloads.)

Utilizando o caminho absoluto (ou seja, informar o caminho completo a partir do raiz).

cd /home/marcos/Downloads

Ambos os comandos fazem a mesma coisa, só que no primeiro caso você digitou menos palavras. Agora considere que você está no diretório **Downloads** e deseja acessar o diretório **etc**. Utilizando caminho relativo:

cd ../../etc (desce três níveis para depois subir para o diretório **etc**).

Utilizando o caminho absoluto (ou seja, informar o caminho completo a partir do raiz):

cd /etc

Fica evidente que, neste caso, é mais vantajoso utilizar o caminho absoluto.

Ao trabalhar com arquivos e diretórios, o uso dos caminhos absolutos ou relativos fica a critério de cada usuário.

- **mkdir dir**: criação do diretório dir.
- **rmdir dir**: remoção do diretório dir.

Cria o diretório teste_aula e acessa o diretório teste_aula.

```
$ mkdir teste_aula  
$ cd teste_aula  
$ pwd  
/home/marcos/teste_aula
```

Volta ao diretório anterior e apaga o diretório.

```
$ cd ..  
$ rmdir teste_aula
```

Tenta acessar o diretório que foi apagado anteriormente.

```
$ cd teste_aula  
bash: cd: teste_aula: Arquivo ou diretório não encontrado
```

46

Comandos Básicos para Trabalhar com Arquivos

Os comandos a seguir implementam operações básicas em arquivos:

- **ls** – listar o conteúdo do diretório corrente (ou de um diretório dado).

```
$ ls -l /etc  
total 1488  
-rw-r--r-- 1 root      root        149 2009-07-13 22:25 00-header  
drwxr-xr-x  4 root      root      4096 2009-10-27 16:10 acpi  
-rw-r--r--  1 root      root     2986 2009-10-27 15:57 adduser.conf  
drwxr-xr-x  2 root      root      4096 2009-11-23 20:54 alternatives  
-rw-r--r--  1 root      root       395 2009-09-17 16:33 anacrontab
```

<----- resultado suprimido propositalmente ----->

- **rm** – este comando serve para eliminar um arquivo do sistema de arquivos. Ele também fará a remoção de diretórios se for especificada a opção **-r**. Com essa opção todo o diretório é excluído, inclusive todos os seus subdiretórios, indiferente se ele possui ou não arquivos. A opção **-f** (**método forçado**) indica que nenhuma confirmação deve ser solicitada ao usuário, ou seja, o comando deverá excluir arquivos e diretórios que estejam com permissão apenas de leitura ou sem permissão alguma.

Apagando um arquivo.

```
$ rm arquivo
```

Apagando um diretório.

```
$ rm -rf diretorio
```

Note que: O comando **rm -rf** deve ser usado com cuidado, pois após apagado um diretório e seus arquivos, não é possível recuperá-los.

- **cp** – O comando **cp** permite a cópia de arquivos e diretórios. Existem três formas básicas do comando: cópia de arquivos para arquivos, cópia de arquivos para diretórios e cópia de diretórios.

Copiando o arquivo /etc/passwd para o diretório corrente.

```
$ cp /etc/passwd
```

Copiando o diretório /etc para o diretório corrente com o nome de conf.

```
$ cp -R /etc ./conf
```

47

- **mv** – O comando **mv** permite a movimentação de um arquivo ou diretório de um local, no sistema, para outro. Apresenta três formas básicas: a primeira forma permite que se mude o nome do arquivo origem para o nome do arquivo destino (mesmo diretório); a segunda forma do comando faz a movimentação de todos os arquivos especificados para o diretório informado como destino e a terceira forma é similar à primeira, mas em vez de arquivo são utilizados diretórios.

Renomeando o arquivo passwd para senhas.txt.

```
$ mv passwd senhas.txt
```

Movendo o arquivo senhas.txt para o diretório conf.

```
$ mv senhas.txt conf
```

- **cat** – apresenta o conteúdo de arquivos.
- **more** – visualiza o conteúdo de arquivos (paginado).

```
$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/bin/sh  
  
<----- resultado suprimido propositalmente ----->
```

```
$ more /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/bin/sh  
  
<----- resultado suprimido propositalmente ----->
```

48

- **head** – o comando **head** mostra na tela as primeiras linhas ou caracteres dos arquivos especificados como argumento.

Mostrando as primeiras 5 linhas do arquivo /etc/passwd.

```
$ head -n 5 /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync
```

- **tail** – este comando mostra a última parte de um arquivo, contado em blocos, caracteres ou linhas. Quando não se coloca opção alguma, o comando assumirá que se está pedindo contado em linhas.

Mostrando as últimas 5 linhas do arquivo /etc/passwd.

```
$ tail -n 5 /etc/passwd
kernooops:x:109:65534:Kernel Oops Tracking Daemon,,,:/bin/false
saned:x:110:116::/home/saned:/bin/false
pulse:x:111:117:PulseAudio daemon,,,:/var/run/pulse:/bin/false
gdm:x:112:119:Gnome Display Manager:/var/lib/gdm:/bin/false
marcos:x:1000:1000:Marcos,,,:/home/marcos:/bin/bash
```



Atividade

Posicione-se no diretório /tmp.

- O que acontecerá se for digitado neste diretório o comando **ls -l ..**? Explique.
- Quais as duas possíveis formas de se posicionar no diretório /etc a partir do diretório /tmp ?

49

Racionalização do Disco – Como Alocar Arquivos

O sistema de arquivos é organizado em blocos de alocação. Cada bloco tem um tamanho fixo e pode variar de um sistema operacional para outro. Tomando como exemplo um sistema operacional que trabalha com blocos de 4 Kb, neste sistema, um arquivo de 6 Kb utilizaria dois blocos para alocação.

Alocação de Espaço em Disco – normalmente, durante o uso do sistema operacional, criar arquivos é uma operação comum, logo, exige que o sistema operacional tenha controle de quais áreas ou blocos no disco estão livres. Esse controle é realizado por meio de uma estrutura (geralmente lista ou tabela) de dados que armazena informações e possibilita ao sistema de arquivos gerenciar o espaço livre.

Mas o mais importante é **controlar a alocação de arquivos em disco** assim, é possível que o sistema operacional controle a alocação de três formas: alocação contígua, alocação encadeada e alocação indexada.

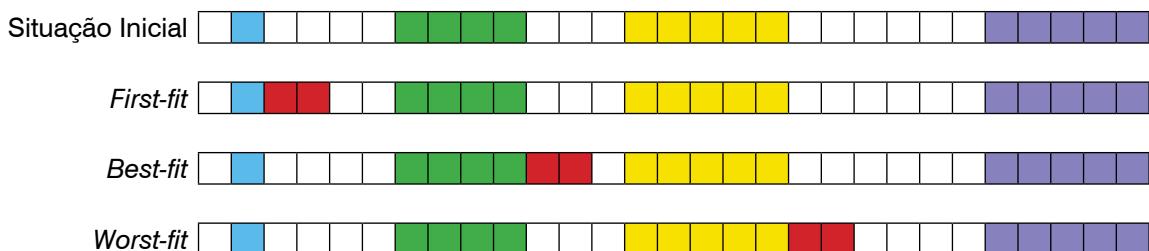
Alocação Contígua

Os blocos são sequencialmente dispostos, o sistema localiza um arquivo por meio do endereço do primeiro bloco e da sua extensão em blocos. O acesso é simples, tanto para a forma sequencial como para a direta. Mas há um problema na alocação de novos arquivos nos espaços livres, pois para colocar n blocos é necessário que se tenha uma cadeia com n blocos dispostos sequencialmente no disco.

Arquivos	Início	#blocos
acorde.txt	002	003
sei_que_aula_cansa.doc	010	002
ainda_mais_depois_de_passar_a_noite_namorando.gif	022	007
aguanta_firme.pdf	006	010

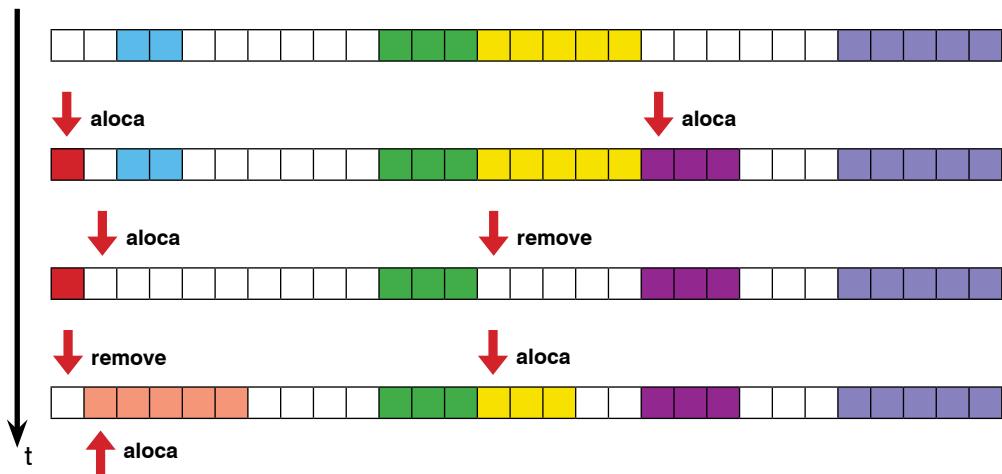
Como determinar o espaço necessário a um arquivo criado e que, depois, pode necessitar de extensão? Alterar o tamanho do arquivo é uma operação complexa. A pré-alocação seria uma solução, mas pode ocorrer que parte do espaço alocado permaneça ocioso por um longo período de tempo. Quando o sistema operacional deseja alocar espaço para um novo arquivo, pode existir mais de um segmento livre disponível com o tamanho exigido e é necessário que alguma estratégia de alocação seja adotada para selecionar qual segmento deve ser escolhido. Algumas estratégias podem ser utilizadas:

- **First-fit:** o primeiro segmento livre com tamanho suficiente para alocar o arquivo é selecionado. A busca na lista é sequencial, sendo interrompida tão logo se encontre um segmento adequado.
- **Best-fit:** seleciona o menor segmento livre disponível com tamanho suficiente para armazenar o arquivo. A busca em toda a lista se faz necessária para a seleção do segmento, a não ser que a lista esteja ordenada por tamanho.
- **Worst-fit:** o maior segmento é alocado e a busca por toda a lista se faz necessária, a menos que haja uma ordenação por tamanho.

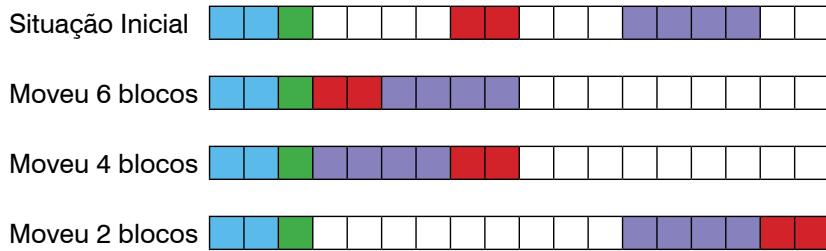


Não importa a estratégia, a alocação apresenta um problema chamado fragmentação de espaços livres. O problema pode se tornar crítico quando um disco possuir blocos livres disponíveis, porém sem um segmento contíguo onde o arquivo possa ser alocado, pois a medida que o sistema evolui arquivos são criados e removidos, mais espaços vazios aparecem; os espaços vazios ficam menores e alocar novos arquivos fica difícil.

Como alocar um arquivo com 5 blocos?



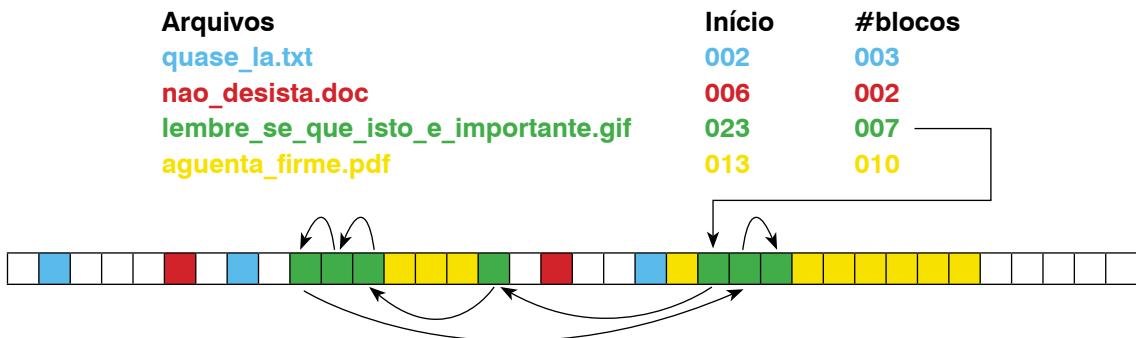
Deve ser feita uma desfragmentação periodicamente (para que este problema seja resolvido) com o objetivo de reorganizar os arquivos no disco para que haja um único segmento de blocos livres. Isto exige um grande consumo de tempo e tem efeito temporário. Uma solução é o uso de algoritmos para minimizar a movimentação de arquivos (rapidez) (observe a figura):



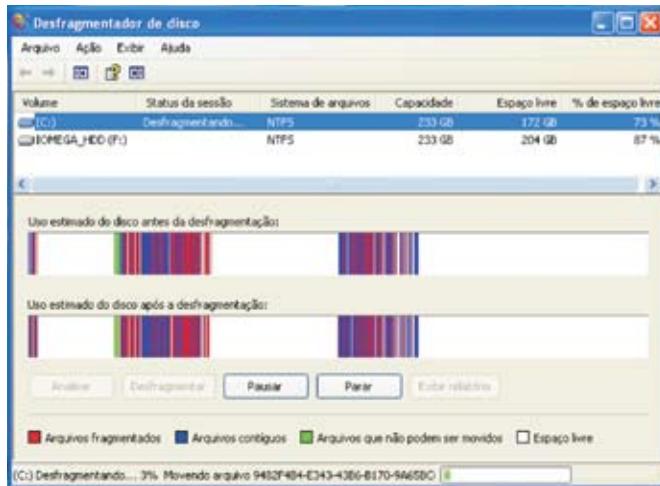
51

Alocação Encadeada

O arquivo é organizado como um conjunto de blocos ligados no disco. Independente de sua localização física cada bloco deve possuir um ponteiro para o bloco seguinte.

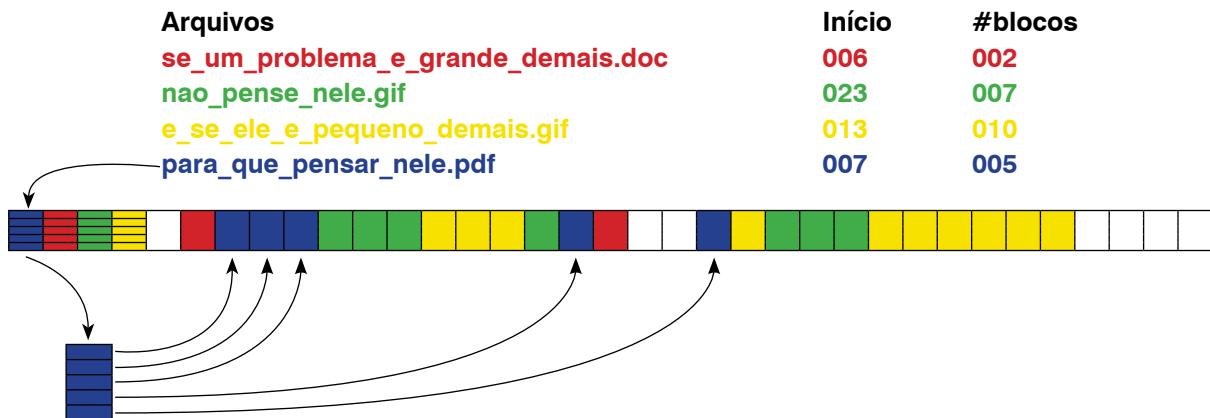


Esta alocação só permite acesso sequencial e desperdiça espaço nos blocos com armazenamento de ponteiros. Ocorre a fragmentação de arquivos, ou seja, a quebra do arquivo em diversos pedaços denominados *extents*. A fragmentação de arquivos causa aumento do tempo de acesso ao arquivo, pois o disco (cabeçotes de leitura) deve deslocar-se diversas vezes para acessar todas as *extents*. Logo, o tempo de acesso aos arquivos é elevado e há desgaste do meio de armazenamento. Como solução é necessário que o disco seja desfragmentado periodicamente.



Alocação Indexada

Neste tipo de alocação ocorre a existência de ponteiros indexando todos os blocos de arquivos em uma única estrutura denominada bloco de índice ou nó-índice (inode). Permite o acesso direto aos blocos do arquivo e não utiliza informações de controle nos blocos de dados como na alocação encadeada.



Conceito de Inode

Cada arquivo armazenado no sistema de arquivo do Linux ocupa uma determinada área do disco. O Linux coloca um endereço nessa área, para determinar facilmente a posição do arquivo no disco. Este endereço é chamado de inode, ou nó-índice. O inode, portanto, serve para identificar os dados de um arquivo e alguns atributos (datas, tamanho, permissões, etc.).

Para achar os dados de um arquivo, usando o nome, primeiramente será aberto o arquivo diretório onde o arquivo está. A seguir, procura-se, dentro desse arquivo, a entrada (registro) com o nome do arquivo requisitado. Portanto, o diretório nada mais é que um arquivo com registros, contendo o nome dos arquivos ali residentes e os respectivos inodes. O sistema pega o inode e acha os dados e o restante dos atributos.

Observar que o diretório também é um arquivo para o Linux. Portanto, ele possui um inode próprio. Sendo assim, quando temos um subdiretório, temos o nome desse subdiretório armazenado como um registro no arquivo de diretório junto com o respectivo inode.



Atividade

Por que os diretórios possuem o atributo de número de *links* no comando **ls -l** com valores maiores de 1? Para responder, digite a sequência de comandos abaixo, isso irá ajudar na análise.

```
$ cd  
$ mkdir teste_dir  
$ ls -l -di teste_dir  
$ cd teste_dir  
$ mkdir teste_outro_dir  
$ ls -l -di . teste_outro_dir  
$ cd teste_outro_dir  
$ ls -di . ..
```

Atributos de Arquivos

Qualquer arquivo no sistema operacional nada mais é do que uma sequência de bytes, armazenados em um dispositivo de acesso direto (disco), que possui certos atributos. Esses atributos são manipulados pelo sistema operacional sempre que for realizado um acesso sobre o arquivo. Os atributos são:

- **Tipo** – indica se um arquivo armazenado no sistema de arquivos é um arquivo propriamente dito ou um diretório. Também é utilizado para diferenciar outros tipos de arquivos dentro desse sistema. Estes tipos podem ser:
 - d:** Um diretório.
 - :** Arquivo normal de dados.
 - l:** Arquivo ligado simbolicamente (*symbolic link*).
 - c:** Arquivo especial de controle de dispositivos orientados a caractere (*device file*). No sistema estes arquivos ficam debaixo do diretório **/dev**.
 - b:** Arquivo especial de controle de dispositivos orientados a bloco.
 - p:** Arquivo pipe com nome.
- **Permissões** – o atributo permissão, também conhecido como **mode**, ou modo de arquivo, indica qual o tipo de acesso um usuário pode fazer sobre o arquivo.
- **Links** – indica de quantas maneiras (nomes) este arquivo está sendo referenciado.

- **Dono (owner)** – neste campo é colocada a identificação do usuário (*user ID*) a que pertence o arquivo.
- **Grupo (group)** – neste campo é colocada a identificação do grupo (*group ID*) a que pertence o arquivo. Esse campo e o anterior, junto com as permissões, vão liberar ou impedir o acesso do arquivo por outra pessoa que não seja o dono.
- **Tamanho** – tamanho do arquivo.
- **Nome** – nome do arquivo. Quando se usa o sistema de arquivos longo, o nome de um arquivo pode ter até um máximo de 255 caracteres quaisquer.

Atributos de Arquivos.

```
$ ls -li /etc/passwd
330 -rw-r--r-- 1 root root 1570 2009-11-15 17:23 /etc/passwd
|   |   |   |   |   |   |
|   |   |   |   |   |   |_ nome do arquivo
|   |   |   |   |   |   |_ data e hora de última alteração no arquivo
|   |   |   |   |   |
|   |   |   |   |   |_ tamanho do arquivo
|   |   |   |   |_ grupo do arquivo (group)
|   |   |   |_ dono do arquivo (owner)
|   |   |_ quantidade de ligações
|   |_ permissões do arquivo
|_ Nº índice dentro do sistema de arquivos (inode)
```

54

Na prática, todo arquivo ou diretório possui vários outros atributos, o comando **stat** nos permite verificar quais são esses atributos:

Verificando os atributos do arquivo /etc/passwd.

```
$ stat /etc/passwd
  File: `/etc/passwd'
  Size: 1570          Blocks: 8           IO Block: 4096  arquivo comum
Device: 806h/2054d  Inode: 330          Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/      root)  Gid: (    0/      root)
Access: 2009-12-10 10:55:27.452540652 -0200
Modify: 2009-11-15 17:23:04.233610010 -0200
Change: 2009-11-15 17:23:04.319234721 -0200
```

Verificando os atributos do diretório /etc.

```
$ stat /etc
  File: `/etc'
  Size: 12288          Blocks: 24          IO Block: 4096   diretório
Device: 806h/2054d      Inode: 24481        Links: 141
Access: (0755/drwxr-xr-x) Uid: (      0/    root)  Gid: (      0/    root)
Access: 2009-12-10 14:16:14.525020717 -0200
Modify: 2009-12-10 14:09:22.722515647 -0200
Change: 2009-12-10 14:09:22.722515647 -0200
```

- **ln:** o comando **ln** faz uma ligação (*link*) de um arquivo existente com um novo nome. Podemos dizer que com esse comando são criados dois nomes para se referenciar o mesmo conjunto de dados. O mais correto é trabalhar com *links* simbólicos, por meio do comando **ln -s**. A funcionalidade principal do *link*, ou seja, de se ter mais de um nome para o mesmo conjunto de dados continua a mesma, e o *link* simbólico difere somente na implementação interna. Em vez de as duas entradas no diretório possuírem o mesmo inode, no simbólico um arquivo aponta para o outro e este aponta para os dados. Essa implementação permite que se façam *links* simbólicos para arquivos que residem em sistemas de arquivos diferentes (discos diferentes).

Cria o arquivo texto.txt com a mensagem “Teste de link”.

```
$ echo "Teste de link" > texto.txt
```

Cria um link simbólico para o arquivo texto.txt.

```
$ ln -s texto.txt link_simbolico
```

Cria um link físico para o arquivo texto.txt.

```
$ ln texto.txt link_fisico
```

Lista os arquivos criados, repare que o nó-indice do arquivo texto.txt e link_fisico são os mesmos (na prática são o mesmo arquivo).

```
$ ls -li texto.txt link_simbolico link_fisico
51698 -rw-r--r-- 2 marcos marcos 14 2009-12-10 15:39 link_fisico
51706 lrwxrwxrwx 1 marcos marcos 9 2009-12-10 15:39 link_simbolico -> texto.txt
51698 -rw-r--r-- 2 marcos marcos 14 2009-12-10 15:39 texto.txt
```

Adiciona uma linha no arquivo com link físico e verifica o conteúdo do arquivo texto.txt.

```
$ echo "adicionando linhas no arquivo" >> link_fisico
$ cat texto.txt
Teste de link
adicionando linhas no arquivo
```

Mais Comandos para Manipular Arquivos

- **file**: identifica o conteúdo de um arquivo, analisando-o.

```
$ file /etc/*
/etc/00-header:          POSIX shell script text executable
/etc/acpi:                directory
/etc/adduser.conf:        ASCII English text
/etc/alternatives:       directory
/etc/anacrontab:         ASCII text
/etc/apm:                 directory

<----- resultado suprimido propositalmente ----->
```

- **whereis**: indica onde estão armazenados os arquivos binários, fontes e páginas de manual de um comando dado.

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

- **du**: indica o espaço usado em disco pelos arquivos ou diretórios informados.

```
$ du -h /etc
104K   /etc/pam.d
212K   /etc/apparmor.d/abstractions
4,0K    /etc/apparmor.d/disable
16K    /etc/apparmor.d/tunables
956K   /etc/apparmor.d/cache
4,0K    /etc/apparmor.d/force-complain
1,2M    /etc/apparmor.d
8,0K    /etc/bonobo-activation
```

```
<----- resultado suprimido propositalmente ----->
```

- **df**: informa sobre os sistemas de arquivos disponíveis na máquina e sua taxa de ocupação.

```
$ df -h
Sist. Arq.          Tam Usad Disp Uso% Montado em
/dev/sda6           15G  5,3G  8,2G  40% /
udev                2,0G 312K  2,0G  1% /dev
none                2,0G 464K  2,0G  1% /dev/shm
none                2,0G  88K  2,0G  1% /var/run
none                2,0G     0  2,0G  0% /var/lock
none                2,0G     0  2,0G  0% /lib/init/rw
none                15G  5,3G  8,2G  40% /var/lib/ureadahead/debugfs
/dev/sda7           23G   12G  9,5G  56% /home
/dev/sdb1           7,5G  1,9G  5,7G  25% /media/02F0-6539
/dev/sda2           86G  34G  53G  40% /media/Marcos
```

- **tree**: apresenta na tela uma estrutura de diretórios, com ou sem os arquivos.

```
$ tree -d -L 2 /etc
/etc
|-- ConsoleKit
|   |-- run-session.d
|   `-- seats.d
|-- NetworkManager
|   |-- dispatcher.d
|   `-- system-connections
|-- ODBCDataSources
|-- X11
|   |-- Xresources
|   |-- Xsession.d
|   |-- app-defaults
|   |-- cursors
|   |-- fonts
|   |-- xinit
|   `-- xkb
|-- acpi
----- resultado suprimido propositalmente ----->
```

Wildcards (Caracteres Curingas ou Metacaracteres)

Wildcards são caracteres que permitem designar nomes genéricos para arquivos e diretórios na linha de comando. O bash permite o uso das seguintes possibilidades de *wildcards*:

- * designa qualquer *string* com 0 ou mais caracteres.

Para listar todos os arquivos que começam com a letra "c" no diretório /usr/bin.

```
$ ls -l /usr/bin/c*
```

Idem, arquivos cujo nome contém a *string* "coa" em qualquer posição.

```
$ ls -l /usr/bin/*coa*
```

Idem, arquivos cujo nome contém as letras "c", "e" e "s" em sequência.

```
$ ls -l /usr/bin/*c*e*s*
```

- ? designa qualquer caractere único.

Para listar todos os arquivos cujo nome contém três letras no diretório /etc.

```
$ ls -l /etc/**?
```

Idem, arquivos cujo nome contém "a" como segunda letra.

```
$ ls -l /etc/?a*
```

- [] designa um conjunto de caracteres.

Para listar todos os arquivos que começam com uma vogal minúscula no diretório /etc.

```
$ ls -l /etc/[aeiou]*
```

Idem, arquivos contendo dois dígitos consecutivos.

```
$ ls -l /etc/*[0-9][0-9]*
```

Comandos Avançados

Os comandos a seguir são muito úteis na manipulação de arquivos.

- **grep**: permite procurar *strings* dentro de arquivos de texto.

Procurar todas as linhas contendo 'bash' em /etc/passwd.

```
$ grep bash /etc/passwd
root:x:0:0:root:/root:/bin/bash
couchdb:x:106:113:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
marcos:x:1000:1000:Marcos,,,:/home/marcos:/bin/bash
```

Procurar todas as linhas que não contenham bash em /etc/passwd.

```
$ grep -v bash /etc/passwd
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
<----- resultado suprimido propositavelmente ----->
```

59

- **find**: permite encontrar arquivos que satisfaçam certas características. É possível fazer-se conjunções com os comandos de seleção por meio de argumentos **-a** para conjunção “e” e **-o** para a conjunção “ou”. Pode-se, também, agrupar mais de um critério por meio da colocação de parênteses. Estes devem ser precedidos por uma barra invertida e devem ser colocados precedidos e sucedidos por um ou mais caracteres em branco.

Procurar todas as entradas *txt dentro do diretório /usr.

```
$ find /usr -name "*txt" -print
/usr/src/bcmwl-5.10.91.9+bdcom/lib/LICENSE.txt
/usr/share/apps/ksgmltools2/docbook/xml-dtd-4.1.2/41chg.txt
/usr/share/apps/ksgmltools2/docbook/xml-dtd-4.1.2/readme.txt
/usr/share/apps/ksgmltools2/docbook/xml-dtd-4.1.2/40chg.txt
<----- resultado suprimido propositavelmente ----->
```

Procurar todas as entradas *ab* ou *cd* (maiúsculas ou minúsculas) presentes em /opt.

```
$ find /opt -iname "*ab*" -or -iname "*cd*"  
/opt/Adobe/Reader9/Resource/TypeSupport/Unicode/mappings/ARABIC.TXT  
/opt/Adobe/Reader9/Resource/Shell/acroread_tab
```

Procurar todas as entradas acessadas a mais de 3 dias em /etc.

```
$ find /etc -atime +3  
/etc/gshadow-  
/etc/ssl/private  
<----- resultado suprimido propositalmente ----->
```

Procurar todas as entradas modificadas a menos de 2 dias em /etc que tenham mais de 3 Kbytes de tamanho.

```
$ find /etc -mtime -2 -and -size +3k  
/etc  
/etc/pam.d  
/etc/network/if-up.d  
<----- resultado suprimido propositalmente ----->
```

Procurar em todos os arquivos regulares do diretório /etc os arquivos que contém a palavra marcos.

```
$ find /etc -type f -exec grep -H marcos {} \;  
/etc/passwd:marcos:x:1000:1000:Marcos,,,,:/home/marcos:/bin/bash  
/etc/group:adm:x:4:marcos  
/etc/group:dialout:x:20:marcos  
/etc/group:cdrom:x:24:marcos  
/etc/group:plugdev:x:46:marcos  
/etc/group:lpadmin:x:104:marcos  
/etc/group:admin:x:115:marcos  
/etc/group:marcos:x:1000:  
/etc/group:sambashare:x:120:marcos  
<----- resultado suprimido propositalmente ----->
```

- **touch**: atualiza a data de um arquivo.

Lista os arquivos com suas horas e datas.

```
$ ls -l  
total 0  
-rwx----- 1 marcos marcos 0 2009-12-10 16:49 diogo.c  
-rwxr--r-- 1 marcos marcos 0 2009-12-10 17:01 instala.txt  
-rw-r--r-- 1 marcos marcos 0 2009-12-10 17:01 leiamme.txt  
-rwx----- 1 marcos marcos 0 2009-12-10 16:49 marcos.c
```

Cria o arquivo arquivo.txt.

```
$ touch arquivo.txt
```

Atualiza a data/hora do arquivo leiamme.txt para o horário local.

```
$ touch leiamme.txt
```

Atualiza a data/hora do arquivo marcos.c para 27/12/2009 as 23:15.

```
$ touch -t 200912272315 marcos.c
```

Lista os arquivos com suas horas e datas.

```
$ ls -l  
total 0  
-rw-r--r-- 1 marcos marcos 0 2009-12-10 17:35 arquivo.txt  
-rwx----- 1 marcos marcos 0 2009-12-10 16:49 diogo.c  
-rwxr--r-- 1 marcos marcos 0 2009-12-10 17:01 instala.txt  
-rw-r--r-- 1 marcos marcos 0 2009-12-10 17:35 leiamme.txt  
-rwx----- 1 marcos marcos 0 2009-12-27 23:15 marcos.c
```

Sistemas de Arquivos e Segurança

É inútil dotar o sistema operacional de ferramentas de **criptografia** e autenticação, se ele grava seus dados de forma não segura, em arquivos que possam ser indevidamente acessados. Neste contexto, o sistema de arquivos tem especial importância, pois é ele que responde pelo acesso e os direitos de acesso que os usuários têm sobre os arquivos.



Criptografia:

Transformação de informações de sua forma original para outra forma ilegível, conhecida apenas por seu destinatário.

Permissão em Arquivos

O Linux possui um sistema de controle de acesso ao sistema de arquivos, seguindo o paradigma de Listas de Controle de Acesso (ACL – *Access Control Lists*). A cada arquivo ou diretório são associados:

- Um usuário proprietário (*owner*). Normalmente, é quem criou o arquivo.
- Um grupo proprietário. Normalmente, é o grupo primário de quem criou o arquivo, mas este pode mudá-lo para outro grupo do qual ele também faça parte.
- Permissões de acesso definidas para o usuário, o grupo e outros usuários (terceiros).

As permissões definidas para os arquivos são:

- **Leitura**: permitindo acesso ao conteúdo do arquivo.
- **Escrita**: permitindo modificar o conteúdo do arquivo.
- **Execução**: permitindo executar o arquivo (caso seja um executável ou script).

As permissões definidas para os diretórios são similares:

- **Leitura**: permitindo acesso ao conteúdo do diretório (listar os arquivos presentes).
- **Escrita**: permitindo modificar o conteúdo do diretório (criar ou apagar arquivos).
- **Execução**: permitindo entrar no diretório, ou atravessá-lo.

Pode-se afirmar que um arquivo é protegido contra leituras ou modificações por suas próprias permissões, e contra apagamentos ou renomeações pelas permissões do diretório onde ele se encontra.

As permissões de acesso a arquivos e diretórios podem ser consultadas por meio de uma listagem de diretório longa, usando o comando **ls -l**.

```
$ ls -l /etc
total 1488
-rw-r--r-- 1 root      root      149 2009-07-13 22:25 00-header
drwxr-xr-x  4 root      root     4096 2009-10-27 16:10 acpi
-rw-r--r-- 1 root      root    2986 2009-10-27 15:57 adduser.conf
drwxr-xr-x  2 root      root     4096 2009-11-23 20:54 alternatives
-rw-r--r-- 1 root      root     395 2009-09-17 16:33 anacrontab

<----- resultado suprimido propositalmente ----->
```

Vamos analisar melhor os caracteres das colunas iniciais da listagem de diretório apresentadas acima. As entradas de diretório em um sistema Linux têm seu tipo indicado pelo primeiro caractere da listagem de diretório longa.

Os demais caracteres representam os direitos de acesso do usuário (*user*), do grupo (*group*) e de terceiros (*others*), em grupos de três caracteres:

r: permissão de leitura (*read*).

w: permissão de escrita (*write*).

x: permissão de execução (*execute*).

-: indica que o respectivo direito está negado.

S: bits SUID e SGID setados (veremos mais adiante).

Vejamos um exemplo:

```
-rw-r---- 1 laureano  users  4956 mar 26 20:34 descricao.html
```

A linha de listagem acima indica que:

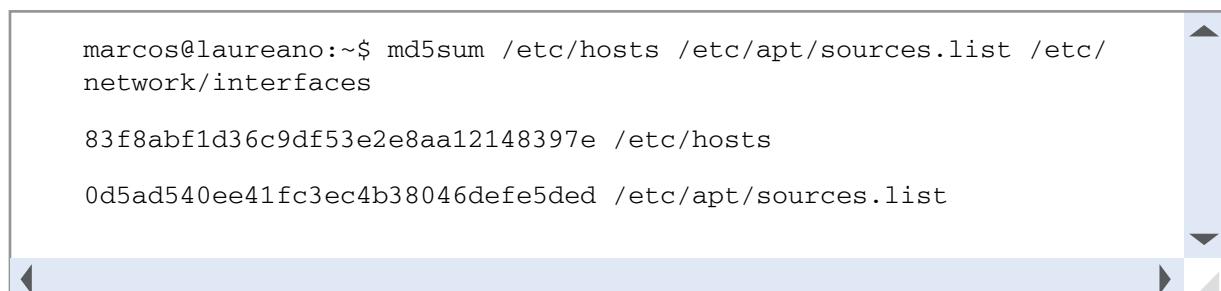
- A entrada corresponde a um arquivo normal (o primeiro caractere é “**-**”).
- O proprietário do arquivo **descricao.html** é o usuário laureano.
- O proprietário possui direito de leitura e escrita sobre o arquivo, mas não de execução.
- O arquivo também está ligado ao grupo *users*.
- O grupo possui apenas direito de leitura sobre o arquivo.
- Outros usuários (terceiros) não possuem nenhum direito de acesso ao arquivo.

Além dos atributos básicos que um arquivo pode ter (*read*, *write* e *execute*), há dois outros atributos de extrema relevância para a segurança, que são:

- **SUID**: este atributo, quando definido em um arquivo, indica que este, caso seja um executável ou um script, quando do momento de sua execução, terá o seu *userid* atribuído de acordo com o *userid* de quem é o proprietário do arquivo, ao invés de quem o executa. Este tipo de privilégio é especialmente perigoso, pois um arquivo cujo dono seja o *root*, com SUID setado e que possa ser executado por qualquer usuário, independente de quem o execute, terá os privilégios de um processo do próprio *root*.
- **SGID**: este atributo, quando setado em um arquivo, similarmente ao SUID, fará com que o processo herde os privilégios do grupo do proprietário do arquivo.

Integridade de Arquivos

Um dilema ao qual vivem submetidos os administradores de sistema é o de saber se os arquivos não foram indevidamente alterados. Uma forma de tentar detectar a possível ação de um intruso é por meio da utilização de ferramentas como **md5sum**.



```
marcos@laureano:~$ md5sum /etc/hosts /etc/apt/sources.list /etc/
network/interfaces
83f8abf1d36c9df53e2e8aa12148397e /etc/hosts
0d5ad540ee41fc3ec4b38046defe5ded /etc/apt/sources.list
```

64

Para cada arquivo especificado, é calculado um *checksum* e, com isto, verifica-se se o arquivo sofreu qualquer tipo de alteração.

Sistemas de Arquivo Criptografados

Em um caso extremo de segurança, em que o próprio acesso ao disco rígido não é garantido, pode ser interessante criptografar os dados contidos no sistema de arquivos. Para tanto há, para o Linux, o *Cryptographic File System* (CFS), que usa um servidor NFS rodando na máquina local para armazenar os arquivos criptografados.

Uma outra versão mais sofisticada do CFS é o TCFS (*Transparent Cryptographic File System*) faz o mesmo que o CFS, mas de forma transparente, sendo integrado ao próprio sistema de arquivos do Linux.

Alterando Permissão dos Arquivos

O comando **chmod** permite alterar as permissões dos arquivos e diretórios. Somente o proprietário de um arquivo pode alterar suas permissões, mesmo que o grupo ou outros possuam direitos de escrita sobre o arquivo. O comando **chmod** tem a seguinte sintaxe:

chmod [Opções] Permissões

A definição das permissões pode ser feita de forma simbólica ou octal (utiliza oito símbolos). A forma simbólica é a mais simples e, por isso, a mais usada por iniciantes. A forma octal é, no entanto, mais empregada, sobretudo em scripts antigos.

Modo Simbólico

As permissões na forma simbólica têm a seguinte sintaxe:

[u g o a] [+ - =] [r w x u g o X]

As letras do primeiro grupo indicam de quem as permissões devem ser alteradas:

- u**: o usuário, proprietário do arquivo.
- g**: o grupo proprietário do arquivo.
- o**: outros (terceiros).
- a**: todos (*all*).

Os símbolos do segundo grupo indicam como os direitos devem ser alterados:

- +**: os direitos indicados devem ser adicionados.
- : os direitos indicados devem ser suprimidos.
- =**: os direitos devem ser ajustados ao valor indicado.
- r**: permissão de leitura.
- w**: permissão de escrita.
- x**: permissão de execução (ou acesso ao diretório).
- u**: usar as permissões atribuídas ao usuário proprietário.
- g**: usar as permissões atribuídas ao grupo proprietário.
- o**: usar as permissões atribuídas a outros.

Veja o exemplo a seguir:

Mostra os atributos iniciais.

```
$ ls -l  
total 0  
-rwxrw-rw- 1 marcos marcos 0 2009-12-10 16:49 diogo.c  
-rwxrw-rw- 1 marcos marcos 0 2009-12-10 16:49 marcos.c
```

Retira de terceiros a permissão de escrita sobre todos os arquivos C no diretório corrente .

```
$ chmod o-w *.c  
$ ls -l  
total 0  
-rwxrw-r-- 1 marcos marcos 0 2009-12-10 16:49 diogo.c  
-rwxrw-r-- 1 marcos marcos 0 2009-12-10 16:49 marcos.c
```

Retira do grupo e de terceiros todas as permissões (leitura, escrita, execução) sobre todos os arquivos C no diretório corrente .

```
$ chmod go-rwx *.c  
$ ls -l  
total 0  
-rwx----- 1 marcos marcos 0 2009-12-10 16:49 diogo.c  
-rwx----- 1 marcos marcos 0 2009-12-10 16:49 marcos.c
```

Outro exemplo:

Mostra os atributos iniciais.

```
$ ls -l *.txt  
-rwxrwxrwx 1 marcos marcos 0 2009-12-10 17:01 instala.txt  
-r--rw-rw- 1 marcos marcos 0 2009-12-10 17:01 leiamme.txt
```

Concede ao usuário permissão de escrita e ajusta ao grupo e outros somente permissão de leitura sobre os arquivos *.txt do diretório corrente. Observe que as permissões podem ser agrupadas, usando vírgulas.

```
$ chmod u+w,go=r *.txt  
$ ls -l *.txt  
-rwxr--r-- 1 marcos marcos 0 2009-12-10 17:01 instala.txt  
-rw-r--r-- 1 marcos marcos 0 2009-12-10 17:01 leiamme.txt
```

O comando **chmod** possui uma opção interessante **-R**, que permite atribuir permissões de maneira recursiva, ou seja, nos conteúdos dos subdiretórios. Assim, a melhor maneira de proteger seu diretório home dos olhares indiscretos de membros do seu grupo e de terceiros é executar o seguinte comando: **chmod -R go-rwx ~**

Modo Octal

O uso do comando **chmod** em modo octal é similar ao modo simbólico, ou seja, são utilizados números ao invés de caracteres. As expressões de permissão são substituídas por valores octais representando as permissões desejadas. Assim, se desejarmos atribuir as permissões **rwxr-x---** a um arquivo **teste.c**, devemos converter o modo simbólico para o modo octal conforme segue:

r w x	r - x	- - -	expressão simbólica
1 1 1	1 0 1	0 0 0	em valores binários
7	5	0	em octal

Desta forma, deve ser executado o comando **chmod 750 teste.c** para obter as permissões desejadas.

A definição binária leva aos seguintes valores:

r = 4

w = 2

x = 1

Isto quer dizer que o valor 7 (rwx) é obtido pela soma de r + w + x (4 + 2 + 1), o valor 5 (r - x) é obtido pela soma de r + x (4 + 1) e finalmente o valor 0 (-) indica que não houve nenhuma soma.

A definição de permissões em modo octal é bem menos flexível que a notação simbólica, mas ainda muito usada, por ser aceita em todos os sistemas Linux. Além disso, sua compreensão é importante para o uso do comando **umask**.



Cada número do conjunto octal indica que o perfil de um dos atributos (usuário, grupo ou terceiros) do arquivo está sendo modificado. O uso do formato octal é similar ao uso do formato simbólico ajustado.

Exemplo: **chmod 750 teste.c** é idêntico à

chmod u=rwx,g=rx,o= teste.c.

O Comando umask

O comando **umask** permite definir uma máscara padrão de permissões para a criação de novos arquivos e diretórios. A sintaxe desse comando usa a notação octal para definir as permissões a suprimir nos novos arquivos e diretórios, a partir das permissões máximas. Vejamos um exemplo:

```
r w x    r - x    - - - permissões desejadas para os novos arquivos  
- - -    - w -    r w x permissões a suprimir  
0 0 0    0 1 0    1 1 1 permissões a suprimir em binário  
0        2        7    máscara em octal
```

Assim, o comando **umask 027** permite definir a máscara desejada (**rwxr-x---**). Normalmente, esse comando é usado nos arquivos de configuração do shell, e nos scripts de instalações de aplicações. Uma forma de definir a máscara de criação de arquivos é seguir o seguinte raciocínio:

- Para que um arquivo fique com a permissão 750:

7 7 7
- 7 5 0

0 2 7

- Para que um arquivo fique com a permissão 640:

7 7 7
- 6 4 0

1 3 7

- Para que um arquivo fique com a permissão 666:

7 7 7
- 6 6 6

1 1 1

Ou seja, pega-se o valor que seria usado no comando **chmod**, subtrai-se este valor de 777 e o resultado é utilizado no comando **umask**.

- O valor do **umask** será utilizado somente na criação de novos arquivos. Arquivos já existentes não terão sua permissão modificada.
- Por mais que defina no comando **umask** a permissão de execução, arquivos regulares jamais serão criados com essa permissão ativa, a qual será habilitada somente para a criação de novos diretórios. Esta é uma medida de precaução do sistema operacional, pois, caso contrário, todos os novos arquivos criados seriam executáveis.



Atividades

- 1) Você conseguiria mudar o nome de seu próprio diretório **home**? Por quê?
- 2) Crie um arquivo teste com os direitos de acesso **rw-rw-rw-** e indique como usar o comando **chmod** para alterar seus direitos de acesso para:
 - **rw-rw-r--**
 - **r-xr-xr-x**
 - **rw-r--r--**
 - **r-----**
- 3) Execute o comando **umask** para que novos arquivos criados no sistema tenham as permissões de acesso definidas a seguir, e teste as máscaras, criando novos arquivos (comando **touch**) e diretórios (comando **mkdir**). Finalmente, explique por que razão as permissões dos arquivos não coincidem com as esperadas, mas as dos diretórios sim.
 - **rw-rw-r--**
 - **r-xr-xr-x**
 - **rw-r--r--**
 - **r-----**
- 4) Crie dois diretórios **d1** e **d2**, com permissões respectivas **r--r--r--** e **r-xr-xr-x**, e compare as possibilidades de acesso em ambos. É possível listar o conteúdo dos dois diretórios, estando fora deles? É possível entrar em ambos?

Máquinas Virtuais

O termo máquina virtual foi descrito na década de 60, do século XX, utilizando um termo de sistema operacional: uma abstração de *software* que enxerga um sistema físico (máquina real). Com o passar dos anos, o termo englobou um grande número de abstrações – por exemplo, *Java Virtual Machine* – JVM que não virtualiza um sistema real.

Ao invés de ser uma máquina real, isto é, um computador real feito de *hardware* e executando um sistema operacional específico, **uma máquina virtual é um computador fictício criado por um programa de simulação**. Sua memória, processador e outros recursos são virtualizados. A virtualização é a interposição do *software* (máquina virtual) em várias camadas do sistema. É uma forma de dividir os recursos de um computador em múltiplos ambientes de execução.

Na ciência da computação, máquina virtual é o nome dado a uma máquina implementada por meio de *software*, que executa programas como um computador real.

Uma máquina virtual é um ambiente criado por um monitor de máquina virtual (*Virtual Machine Monitor* – VMM), também conhecido como *hypervisor*. O monitor pode criar uma ou mais máquinas virtuais sobre uma única máquina real.

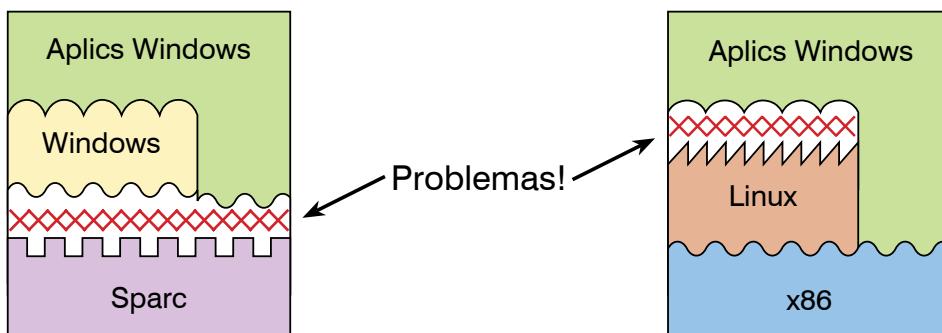
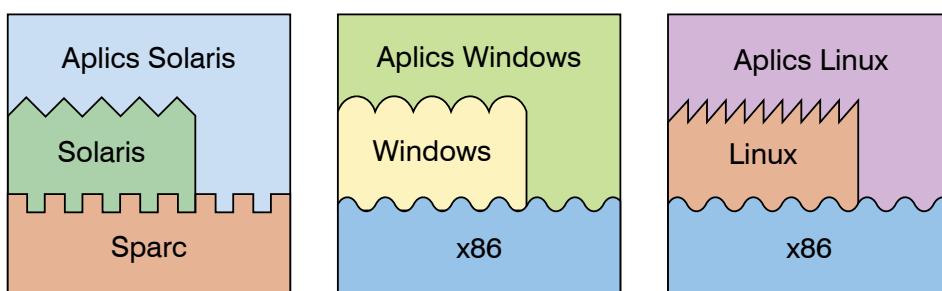
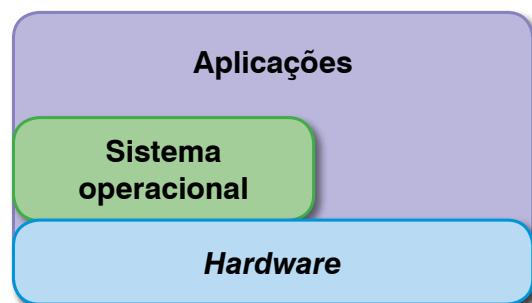
As finalidades primárias de um sistema operacional são habilitar aplicações a interagir com um *hardware* de computador e gerenciar os recursos de *hardware* e *software* de um sistema. Por tal motivo, o monitor de máquinas virtuais pode ser definido como “sistema operacional para sistemas operacionais”.

Por que Máquinas Virtuais Existem?

Os sistemas de computadores são projetados com basicamente três componentes: *hardware*, sistema operacional e aplicações.

O papel do *hardware* é executar as operações solicitadas pelas aplicações. O sistema operacional recebe as solicitações das operações (por meio das chamadas de sistemas) e controla o acesso ao *hardware* – principalmente nos casos em que os componentes são compartilhados, como sistema de memória e entrada e saída.

Os sistemas operacionais, assim como as aplicações, são projetados para aproveitar o máximo dos recursos que o *hardware* fornece. Normalmente os projetistas de *hardware*, sistema operacional e aplicações, trabalham de forma independente (em empresas e tempos diferentes). Estes trabalhos independentes geraram, ao longo dos anos, várias plataformas operacionais diferentes (e não compatíveis entre si). Assim, aplicações escritas para uma plataforma operacional não funcionam em outras plataformas.



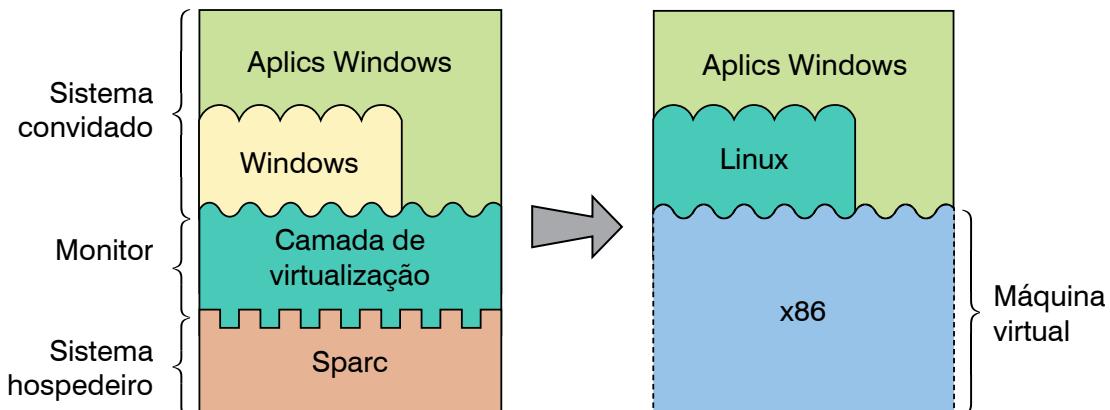
A utilização de máquinas virtuais possibilita a resolução desse problema, pois a máquina virtual cria uma “camada” para compatibilizar diferentes plataformas. Essa “camada” – softwares que podem ser utilizados para fazer os recursos parecerem diferentes do que realmente são – é chamada de virtualização.



Definições

Usando os serviços oferecidos por uma determinada interface de sistema, a camada de virtualização constrói outra interface de mesmo nível, de acordo com as necessidades dos componentes de sistema que farão uso dela. A nova interface de sistema, vista por meio dessa camada de virtualização, é denominada máquina virtual.

A figura a seguir apresenta um exemplo de máquina virtual, em que uma camada de virtualização permite executar um sistema operacional Windows e suas aplicações sobre uma plataforma de *hardware* Sparc, distinta daquela para a qual esse sistema operacional foi projetado (Intel/AMD).



Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na figura acima:

- O sistema real, ou sistema hospedeiro (*host system*), que contém os recursos reais de *hardware* e *software* do sistema.
- O sistema virtual, também denominado sistema convidado (*guest system*), que executa sobre o sistema virtualizado. Em alguns casos, vários sistemas virtuais podem coexistir, executando sobre o mesmo sistema real.
- A camada de virtualização, ou monitor de virtualização, que constrói as interfaces virtuais a partir da interface real.

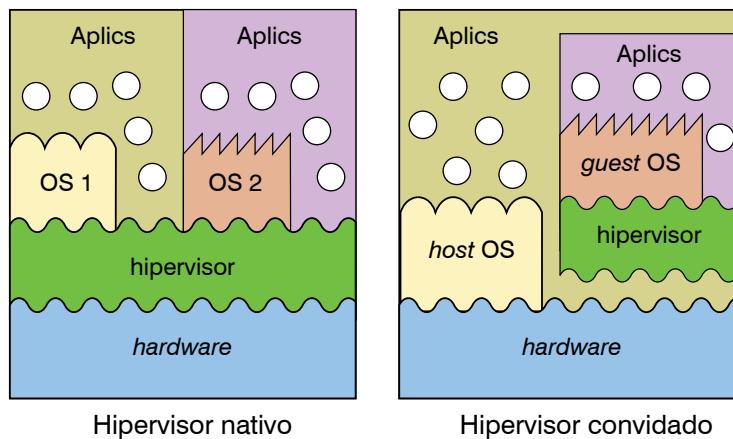
A construção de máquinas virtuais é bem mais complexa do que pode parecer à primeira vista. Caso os conjuntos de instruções do sistema real e do sistema virtual sejam diferentes, será necessário usar as instruções da máquina real para simular as instruções da máquina virtual. Além disso, é necessário mapear os recursos de *hardware* virtuais (periféricos oferecidos ao sistema convidado) sobre os recursos existentes na máquina real (os periféricos reais). Por último, pode ser necessário mapear as chamadas de sistema emitidas pelas aplicações do sistema convidado em chamadas equivalentes no sistema real, quando os sistemas operacionais virtual e real forem distintos.

Tipos de Máquinas Virtuais

Há basicamente duas abordagens para a construção de sistemas de máquinas virtuais:

- **Tipo I ou nativa:** sistema em que o monitor é implementado entre o *hardware* e os sistemas convidados (*guest system*).
- **Tipo II ou convidada:** nele o monitor é implementado como um processo do sistema operacional real subjacente, denominado sistema anfitrião (*host system*).

A figura a seguir ilustra a organização tradicional de um sistema de máquinas virtuais. Para maximizar o desempenho, o monitor, sempre que possível, permite que a máquina virtual execute diretamente sobre o *hardware*, em modo usuário. O monitor retoma o controle sempre que a máquina virtual tenta executar uma operação que possa afetar o correto funcionamento do sistema, o conjunto de operações de outras máquinas virtuais ou do próprio *hardware*. O monitor simula com segurança a operação antes de retornar o controle à máquina virtual.



Máquinas Virtuais de Tipo I ou Nativas

O monitor tem o controle do *hardware* e cria um ambiente de máquinas virtuais. Cada máquina virtual se comporta como uma máquina física completa que pode executar o seu próprio sistema operacional, semelhante a um sistema operacional tradicional que está no controle da máquina. O resultado da completa virtualização da máquina é um conjunto de computadores virtuais executando sobre o mesmo sistema físico.

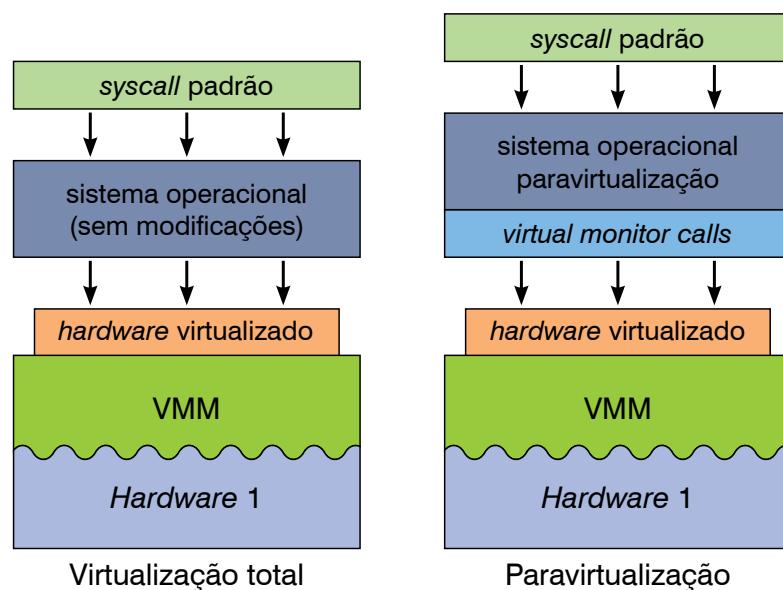
Máquinas Virtuais de Tipo II ou Convidada

O monitor executa sobre um sistema anfitrião. O monitor de tipo II funciona de forma análoga ao de tipo I, sendo a sua maior diferença a existência de um sistema abaixo deste. Neste modelo, o monitor simula todas as operações que o sistema anfitrião controlaria.

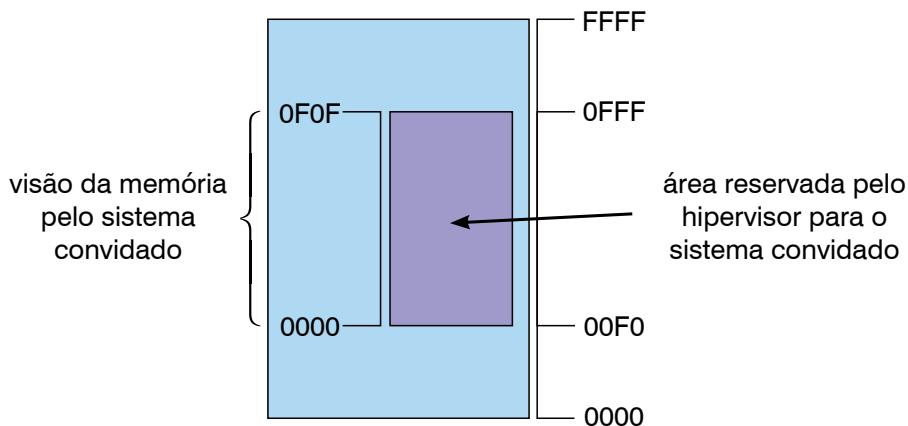
Estratégias de Virtualização

As estratégias mais utilizadas para virtualização, atualmente, são a virtualização total (*full virtualization*) a paravirtualização (*paravirtualization*) e a recompilação dinâmica (*dynamic recompilation*).

- **Virtualização Total** – na virtualização total, uma estrutura completa de *hardware* é virtualizada, portanto o sistema que vai ser virtualizado (sistema convidado) não precisa sofrer qualquer tipo de alteração. Como toda a interface do *hardware* é virtualizada, incluindo todas as instruções do processador e os dispositivos de *hardware*, o principal benefício da virtualização total é justamente o fato de que o sistema que será virtualizado não sofre qualquer tipo de alteração; em compensação sua execução é mais lenta e o monitor precisa implementar alternativas para que as operações privilegiadas possam ser executadas em processadores que não suportem a virtualização nativamente, tais como os processadores Intel 32 bits disponíveis atualmente.
- **Paravirtualização** – na paravirtualização o sistema que vai ser virtualizado (sistema convidado) sofre modificações para que a interação com o monitor de máquinas virtuais seja mais eficiente. A paravirtualização, embora exija que o sistema a ser virtualizado precise ser modificado, o que diminui a portabilidade do sistema, permite que o sistema convidado consiga acessar recursos do *hardware* diretamente. O acesso é monitorado pelo monitor de máquinas virtuais que fornece ao sistema convidado todos os “limites” do sistema, tais como endereços de memória que pode ser utilizado e endereçamento em disco, por exemplo. A paravirtualização reduz a complexidade do desenvolvimento das máquinas virtuais, pois, historicamente, os processadores não suportam a virtualização nativa. A performance obtida, a principal razão para utilizar a paravirtualização, compensa as modificações que serão implementadas nos sistemas convidados.



A virtualização total exige do monitor de máquinas virtuais um trabalho adicional para gerenciamento da memória. Ao ser instanciado (criado virtualmente), o monitor reserva um espaço da memória para trabalho, mas o sistema convidado acredita estar utilizando o início da memória. A figura a seguir demonstra exatamente esta dificuldade: a máquina real tem endereçamento de memória iniciando em 0000 até FFFF.



O monitor reserva o espaço de memória compreendido entre 000F e 00FF, mas o sistema convidado enxerga este espaço reservado como sua memória total e, portanto, iniciando em 0000 e finalizando em 0F0F. Sempre que o sistema convidado acessa a memória virtual, o monitor faz a tradução dos endereços para acessar a memória real. Isto não ocorre na paravirtualização, pois o monitor informa ao sistema convidado qual o espaço de memória que pode ser utilizado. Dessa forma, o sistema convidado acessa diretamente a memória fornecida sem a interferência do monitor de máquinas virtuais.

O mesmo processo ocorre para acesso ao disco: enquanto em um sistema com virtualização total, o sistema convidado enxerga o disco como sendo seu, para uso exclusivo, na paravirtualização o sistema convidado sabe que o disco é compartilhado. Ainda, na virtualização total, sempre que o sistema realiza uma chamada de sistema para acesso ao disco, o monitor de máquinas virtuais deve capturar esta chamada, interpretar e repassar para que o sistema anfitrião execute a operação (em casos de máquinas virtuais convidadas), em máquinas nativas, o monitor de máquinas virtuais realiza a chamada de sistema a disco. O resultado da chamada de sistema então é repassado para o sistema convidado. Na paravirtualização o sistema convidado sabe que o disco é compartilhado e realiza diretamente o acesso a ele (sem a interferência do monitor de máquinas virtuais).

- **Recompilação dinâmica ou Tradução dinâmica** – Outra técnica bastante utilizada em construção de máquinas virtuais é a recompilação dinâmica (*dynamic recompilation*) ou tradução dinâmica (*dynamic translation*) de partes do código. Nessa técnica, o monitor irá traduzir o código original para um código que possa ser entendido nativamente pelo *hardware*. Com a recompilação, durante a execução, o sistema pode adequar o código gerado de forma a refletir o ambiente original do programa, explorando informações que normalmente não estão disponíveis, como uma instrução de chamada para o processador, e um compilador estático tradicional, para que o código gerado seja mais eficiente. Em outros casos, um sistema pode empregar a recompilação dinâmica como parte de uma estratégia de otimização adaptável para executar uma representação portável do programa, tal como bytecodes Java.

Uso de Máquinas Virtuais

Ao longo dos anos, as máquinas virtuais vêm sendo utilizadas com várias finalidades, entre elas processamento distribuído e segurança. Um uso frequente de sistemas baseados em máquinas virtuais é a chamada “consolidação de servidores”. Em vez da utilização de vários equipamentos com seus respectivos sistemas operacionais, utiliza-se somente um computador com máquinas virtuais, abrigando os vários sistemas operacionais e suas respectivas aplicações e serviços. Muitos dos benefícios das máquinas virtuais utilizadas no ambiente dos *mainframes* também podem ser obtidos nos computadores pessoais.

Há várias vantagens para a utilização de máquinas virtuais em sistemas de computação:

- Facilitar o aperfeiçoamento e testes de novos sistemas operacionais.
- Auxiliar no ensino prático de sistemas operacionais e programação ao permitir a execução de vários sistemas para comparação no mesmo equipamento.
- Executar diferentes sistemas operacionais sobre o mesmo *hardware*, simultaneamente.
- Simular configurações e situações diferentes do mundo real, como mais memória disponível ou a presença de outros dispositivos de E/S.
- Simular alterações e falhas no *hardware* para testes ou reconfiguração de um sistema operacional, provendo confiabilidade e escalabilidade para as aplicações.
- Garantir a portabilidade das aplicações legadas (que executariam sobre uma máquina virtual simulando o sistema operacional original).
- Desenvolver novas aplicações para diversas plataformas, garantindo a portabilidade destas aplicações.
- Diminuir custos com *hardware*, utilizando a consolidação de servidores.
- Facilitar o gerenciamento, a migração e a replicação de computadores, de aplicações ou de sistemas operacionais.
- Prover um serviço dedicado para um cliente específico com segurança e confiabilidade.

Uso de Máquinas Virtuais no Processo de Ensino

O uso de máquinas virtuais tem tornado o processo de ensino extremamente versátil e principalmente, gerado um custo menor. Tomando como exemplo, normalmente o treinamento para administração de sistemas é um processo delicado. A exigência de equipamentos dedicados para este tipo de ensino acaba por tornar o curso caro (equipamento), sem contar os problemas que ocorrem durante o curso, tais como: reinstalação de sistema operacional e aplicações por falha do aluno (muito comum e totalmente compreensível), perda de configurações, perda do acesso ao sistema (esquecimento da senha do administrador), entre outros.

Nestas situações, o uso de máquinas virtuais facilita bastante o processo. O maior investimento seria em equipamentos de maior capacidade (servidor) e com os alunos utilizando máquinas menos potentes (*thin clients* ou clientes magros); além do tempo dedicado pelo professor para adaptar as suas aulas à nova tecnologia.

Outro problema que pode ser contornado é a utilização do laboratório de informática por várias turmas (ou disciplinas) diferentes. O compartilhamento de recursos causa um esforço maior por parte da administração de informática e do próprio professor (ao ter que contornar eventuais ocorrências em sala de aula). Com o uso de máquinas virtuais, cada aluno poderá ter o seu computador virtual (ou computadores, para formar uma rede) para trabalhar da forma que desejar. O professor (ou professores) poderá ter vários laboratórios (imagem, programação, redes, sistemas operacionais, etc.) montados sobre um único laboratório físico.

As máquinas virtuais podem ser utilizadas no ensino de:

- Administração de sistemas: Instalação, configuração.
- Redes: protocolos, planejamento e configuração (redes virtuais podem ser montadas para cada aluno).
- Programação de computadores: Testes de aplicações em várias plataformas (ANSI C, Perl e Java, por exemplo).
- Sistemas operacionais: conceitos de construção, entendimento de processos; testes, implementações ou alteração no *kernel* de sistemas existentes.

Enfim, as máquinas virtuais podem ser utilizadas da mesma forma que um sistema real é utilizado.

Algumas Máquinas Virtuais Disponíveis

- **VMware** – o VMware é, atualmente, a máquina virtual para a plataforma x86 de uso mais difundido. Provendo uma implementação completa da interface x86 ao sistema convidado, o VMware é uma ferramenta útil em diversas aplicações. Embora essa interface seja extremamente genérica para o sistema convidado, acaba conduzindo a um monitor mais complexo. Como podem existir vários sistemas operacionais em execução no mesmo *hardware*, o monitor tem que **emular** certas instruções para representar corretamente um processador virtual em cada máquina virtual; as instruções que devem ser emuladas são chamadas de instruções sensíveis. Atualmente, o VMware possui versões nativas e convidadas de seus monitores de máquinas virtuais. O VMware Workstation (convidada), e seus derivados, utiliza a estratégia de virtualização total e recompilação dinâmica. O VMware ESX Server (nativa) implementa, ainda, a paravirtualização.
- **Xen** – o ambiente de máquinas virtuais Xen permite executar sistemas operacionais convencionais como Linux e Windows, modificados para executar sobre o monitor Xen. A degradação média de desempenho observada em sistemas virtualizados sobre a plataforma Xen não excede 5%. O ambiente Xen é um monitor nativo para a plataforma x86. Suporta múltiplos sistemas convidados simultaneamente, com bom desempenho e isolamento. Sua arquitetura está descrita e disponível para consulta no site do projeto – <http://www.xen.org/>, sendo o componente principal de um projeto mais amplo chamado XenoServers que consiste em construir uma infraestrutura para computação distribuída. A proposta original do ambiente Xen é suportar aplicações sem a necessidade de alterações, múltiplos sistemas operacionais convidados e a cooperação entre estes sistemas, mas com o máximo de desempenho possível.

Emular:
Procurar imitar.

A Máquina Virtual User-Mode Linux

O User-Mode Linux foi proposto por Jeff Dike, em 2000, como uma alternativa para o uso de máquinas virtuais no ambiente Linux. O *kernel* do Linux foi portado de forma a poder executar sobre si mesmo, como um conjunto de processos do próprio Linux. O resultado é um *user space* separado e isolado, na forma de uma máquina virtual, que utiliza a simulação de *hardware*, construída a partir dos serviços providos pelo sistema anfitrião. Essa máquina virtual é capaz de executar todos os serviços e aplicações disponíveis para o sistema anfitrião.

O User-Mode Linux é uma máquina virtual convidada, ou seja, executa na forma de um processo no sistema anfitrião, e os processos em execução na máquina virtual não têm acesso aos recursos do sistema anfitrião diretamente. O monitor é um processo único que controla a execução de todas as máquinas virtuais. Há um processo, no anfitrião, para cada instância da máquina virtual, e um único processo no anfitrião para o monitor.

No UML, cada máquina virtual é vista como um processo do sistema Linux nativo subjacente. Esse processo contém internamente o monitor de máquina virtual, o núcleo (*kernel*) convidado e os processos da máquina virtual. Todos os dispositivos físicos usados dentro da máquina virtual são virtualizados a partir de recursos do sistema nativo:

- Os discos da máquina virtual correspondem a arquivos no sistema nativo.
- As interfaces de rede virtuais são providas por **daemons**, executando no sistema nativo.

Os usuários e processos da máquina virtual são distintos dos usuários e processos do sistema nativo; por isso, cada usuário pode ser *root* (superusuário) de suas máquinas virtuais, sem prejuízo para a segurança do sistema.

Com o auxílio do professor, vamos preparar uma imagem do User-Mode Linux para que você possa realizar as mais variadas atividades de forma segura.

Preparação do Kernel

Primeiro é necessário baixar o último *kernel* do Linux (disponível em: <http://www.kernel.org/>). Supondo que a última versão seja o *kernel* 2.6.32, basta executar o seguinte comando:

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.tar.bz2
```

Após ter baixado o *kernel* do Linux, descompacte-o com o seguinte comando:

```
$ tar -jxf linux-2.6.32.tar.bz2
```

Agora, prepare o *kernel* com configurações padrões:

```
$ cd linux-2.6.32  
$ make ARCH=um defconfig
```

Entre no *kernel* e desabilite algumas coisas (suporte a módulo, som, sistemas de arquivos não utilizados, etc.). Caso você não se sinta confortável em alterar alguma configuração, este passo pode ser pulado:

```
$ make ARCH=um menuconfig
```

Agora é só compilar (esta é a parte mais demorada):

```
$ make ARCH=um linux
```

Parabéns, você acabou de compilar o seu primeiro *kernel* no Linux. Agora é só copiar o executável gerado (um arquivo com o nome Linux) para o mesmo diretório da imagem Linux que irá executar.

Pronto, inicie a máquina virtual. Para isto, você precisará de uma imagem de um sistema operacional Linux configurada. Peça a indicação ao professor de qual imagem utilizar ou baixe uma de <http://fs.devloop.org.uk/>.

Aqui vamos utilizar a imagem do Debian 4.0 (uma das distribuições Linux mais famosa e utilizada). Basta realizar o *download* da imagem com o comando:

```
$ wget http://fs.devloop.org.uk/filesystems/Debian-4.0/  
Debian-4.0-x86-root_fs.bz2
```

E depois descompactar com o comando:

```
$ bunzip2 Debian-4.0-x86-root_fs.bz2
```

Para executar a sua imagem do Linux:

```
$ ./linux ubd0=vm0,Debian-4.0-x86-root_fs
```

Todas as alterações que você fizer na imagem estarão salvas no arquivo **vm0** (que será criado na primeira execução). Você verá um processo de inicialização no seu terminal, depois serão pedidos um usuário e uma senha, só informar o usuário *root* e teclar <ENTER> na senha (senha em branco).

Agora você, com o auxílio do professor, poderá testar comandos de administração, alterar arquivos de configuração, entre outras atividades. Não se preocupe se errar algum comando e sua imagem parar de funcionar, basta apagar o arquivo **vm0** e recomeçar novamente. Para finalizar o uso da sua imagem, digite o comando **halt** (comando para desligar o sistema operacional Linux).

80

Atividades

- 1) Verifique se há diferenças entre os comandos executados no terminal da máquina virtual e os comandos que você vinha utilizando em aula até o momento.
- 2) Faça uma pesquisa e verifique quais empresas da sua região utilizam a virtualização em sistemas de produção. Identifique quais as máquinas virtuais utilizadas.

Administração de Usuários

O Linux é um sistema operacional multiusuário, portanto é necessário que todos os usuários sejam cadastrados e tenham permissões de acesso diferenciadas. É possível, também, cadastrá-los em grupos, para facilitar o gerenciamento.

A criação e a administração de contas de usuários no sistema são exclusivas do superusuário (*root*). É uma tarefa que demanda responsabilidade e deve ser acompanhada com muita atenção.

Verificando Informações do Usuário

Todo usuário possui um número chamado *user ID* com o qual o sistema Linux o identifica. Além do *user ID*, os usuários possuem o *group ID*. Toda vez que um processo for ativado será atribuído a este um *User ID* e um *Group ID*. Os ID's são chamados de identificação efetiva do processo.

Sintaxe: **id [opções][nome]**

Exemplo de uso:

```
Sem parâmetros, pegando as informações do usuário atual .
```

```
$ id  
uid=1000(marcos) gid=1000(marcos)  
grupos=4(adm),20(dialout),24(cdrom),46(plugdev),104(lpadmin),115()
```

Com parâmetros, pegando as informações do usuário bin.

```
$ id bin  
uid=2(bin) gid=2(bin) grupos=2(bin)
```

Pegando somente o user id do usuário atual.

```
$ id -u  
1000
```

Tornando-se Outro Usuário – Comando su

Tal comando permite ao usuário mudar sua identidade, sem fazer o *logout*. É útil para executar um programa ou comando como superusuário sem ter que abandonar a seção atual.

Sintaxe: **su [-] [usuário]**

Em que:

- Quando informado, indica para iniciar as configurações do usuário que está sendo acessada.
- Usuário é o nome que a pessoa usa para acessar o sistema. Se não digitado, é assumido o usuário *root*.

Será pedida a senha do superusuário para autenticação. Digite **exit** quando desejar retornar à identificação do usuário anterior.

Exemplos de uso:

Acessando o usuário *root* sem carregar suas configurações.

Pegando as informações do usuário atual.

```
$ id  
uid=1000(marcos) gid=1000(marcos)  
grupos=4(adm),20(dialout),24(cdrom),46(plugdev),104(lpadmin),115()
```

Virando o usuário *root*.

```
$ su  
Senha:
```

Pegando as informações do usuário atual.

```
$ id  
uid=0(root) gid=0(root) grupos=0(root)
```

Verificando que não houve mudança de usuário.

```
$ pwd  
/home/marcos
```

Saindo da conta root.

```
$ exit  
exit
```

Pegando as informações do usuário atual.

```
$ id  
uid=1000(marcos) gid=1000(marcos)  
grupos=4(adm),20(dialout),24(cdrom),46(plugdev),104(lpadmin),115()
```

Acessando o usuário *root*, carregando suas configurações.

Verificando o usuário atual.

```
$ id  
uid=1000(marcos) gid=1000(marcos)  
grupos=4(adm),20(dialout),24(cdrom),46(plugdev),104(lpadmin),115()
```

Tornando-se o usuário root e carregando suas configurações.

```
$ su -  
Senha:
```

Imprimindo o diretório atual e pegando as informações da nova conta.

```
$ pwd  
/root  
$ id  
uid=0(root) gid=0(root) grupos=0(root)
```

Finalizando a conta root.

```
$ exit  
sair
```

Imprimindo o diretório atual e pegando as informações conta.

```
$ pwd  
/home/marcos  
$ id  
uid=1000(marcos) gid=1000(marcos)  
grupos=4(adm),20(dialout),24(cdrom),46(plugdev),104(lpadmin),115()
```

Arquivo passwd e group

O arquivo **/etc/passwd** é o banco de dados dos usuários que podem logar no sistema. Tem formato de vários campos, os quais são separados pelo caractere : (dois pontos) e sempre na mesma ordem:

- Nome de *login* do usuário.
- Senha (criptografada).
- Id do usuário (identificação única, semelhante a um número de carteira de identidade).
- Grupo primário desse usuário (o usuário poderá participar de vários grupos).
- Nome completo (nome normal, sem ser o de *login*).
- Diretório home deste usuário.
- Shell inicial.

O exemplo a seguir mostra como são esses valores na prática:

```
marcos:x:1000:1000:Marcos Laureano,,,:/home/marcos:/bin/bash
```

Quando estamos utilizando **shadow password** (um pacote que evita o acesso de *hackers* ao conteúdo das senhas, mesmo criptografadas, dificultando, assim, a tentativa de quebra de senha), o segundo campo é substituído por um * e a senha é armazenada em outro arquivo (**/etc/shadow**), normalmente inacessível.

O arquivo **/etc/group** define os grupos aos quais os usuários pertencem. Seu conteúdo são linhas da forma: **group_name:passwd:GID:user_list**, em que: **group_name** – é o nome do grupo.

- *passwd* – um grupo pode opcionalmente ter uma senha.
- *GID (group id)* – é um código, como o *user id* (no arquivo **/etc/passwd**), mas relativo ao grupo.
- *user list* é a lista (separada por vírgulas) de todos os usuários que pertencem a este grupo.

Um usuário pode pertencer a qualquer número de grupos e herdará todas as permissões de acesso aos arquivos desses grupos.

Adicionando Grupos – Comando groupadd

Para facilitar a administração do sistema, pode-se usar o conceito de grupos de usuários com perfis semelhantes. Por exemplo, definir grupos conforme os departamentos de uma empresa.

Sintaxe: **groupadd [opções] grupo**

Este comando irá alterar os arquivos:

- **/etc/group** – informações de grupos.
- **/etc/gshadow** – informações de grupos armazenadas de forma segura (senhas de grupo).

Criando o grupo de vendas e verificando o final do arquivo /etc/group.

```
$ groupadd vendas  
$ tail -3 /etc/group  
nogroup:x:65534:  
crontab:x:101:  
vendas:x:102:
```

Criando o grupo alunos com o Group ID = 2424.

```
$ groupadd -g 2424 alunos  
$ tail -3 /etc/group  
crontab:x:101:  
vendas:x:102:  
alunos:x:2424:
```

Criando o grupo teste sem informar o GID (observe que o sistema pega o último GID utilizado e soma 1).

```
$ groupadd teste  
$ tail -3 /etc/group  
vendas:x:102:  
alunos:x:2424:  
teste:x:2425:
```

Eliminando Grupos – Comando groupdel

O comando **groupdel** permite que se eliminem grupos do sistema. Somente o superusuário poderá utilizá-lo.

Sintaxe: **groupdel grupo**

Exemplo de utilização:

Apagando os grupos criados anteriormente.

```
$ groupdel vendas  
$ groupdel alunos  
$ tail -3 /etc/group  
nogroup:x:65534:  
crontab:x:101:  
teste:x:2425:  
$ groupdel teste  
$ tail -3 /etc/group  
users:x:100:  
nogroup:x:65534:  
crontab:x:101:
```

86

Adicionando Usuários – Comando useradd

O comando **useradd** permite que se criem usuários conforme especificado em opções. Somente o superusuário poderá utilizá-lo.

Sintaxe: **useradd [opções] usuário**

Este comando irá alterar os arquivos:

- **/etc/passwd** – informações de contas de usuários e senhas criptografadas.
- **/etc/shadow** – informações de contas de usuários e senhas criptografadas.
- **/etc/group** – informações de grupos.

Exemplos de utilização:

Criando o usuário Diogo, como não foi informado o grupo, o comando cria automaticamente um grupo com o mesmo nome do usuário.

```
$ useradd diogo  
$ id diogo  
uid=1000(diogo) gid=1000(diogo) groups=1000(diogo)  
$ tail -3 /etc/passwd  
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh  
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh  
diogo:x:1000:1000::/home/diogo:/bin/sh
```

Cria o usuário Rosa, especificando o seu grupo e o diretório home.

```
$ useradd rosa -d /home/rosa -g root  
$ id rosa  
uid=1001(rosa) gid=0(root) groups=0(root)  
$ tail -3 /etc/passwd  
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh  
diogo:x:1000:1000::/home/diogo:/bin/sh  
rosa:x:1001:0::/home/rosa:/bin/sh
```

Alterando a Senha do Usuário – Comando passwd

O comando **passwd** permite que se troque a senha de determinado usuário. O superusuário pode trocar a senha de qualquer outro. O usuário comum, porém, pode trocar somente a sua senha. As senhas são armazenadas no arquivo **/etc/passwd** ou **/etc/shadow**. No arquivo **/etc/passwd** também são armazenadas as informações relativas aos usuários.

Após a criação do usuário será necessário criar uma senha para ele, caso contrário, não será permitido que o usuário faça *login* no sistema.

Sintaxe: **passwd [usuário]**

Eliminando Usuários – Comando userdel

O comando **userdel** permite que se eliminem usuários do sistema. Somente o superusuário poderá utilizá-lo.

Sintaxe: **userdel [opções] usuário**

Exemplos de utilização:

Posiciona-se no diretório /home e lista os diretórios dos usuários.

```
$ cd /home  
$ ls -l  
total 8  
drwxr-xr-x 2 diogo diogo 4096 Dec 14 17:20 diogo  
drwxr-xr-x 2 rosa root 4096 Dec 14 17:20 rosa
```

Apaga o usuário Diogo, mas o seu diretório de usuário é mantido e fica órfão (sem dono).

```
$ userdel diogo  
$ ls -l  
total 8  
drwxr-xr-x 2 1000 1000 4096 Dec 14 17:20 diogo  
drwxr-xr-x 2 rosa root 4096 Dec 14 17:20 rosa
```

Apaga o usuário Rosa juntamente com o seu diretório de usuário.

```
$ userdel -r rosa  
$ ls -l  
total 4  
drwxr-xr-x 2 1000 1000 4096 Dec 14 17:20 diogo
```

Atenção: ao eliminar um usuário do sistema, verifique se em seu diretório home não há arquivos importantes. Na dúvida, elimine o usuário do sistema, mas mantenha o seu diretório home e peça para alguém analisar o conteúdo.

Assim que tiver eliminado um usuário, talvez seja interessante verificar se o UID antigo deste não tem propriedade sobre outros arquivos no sistema. Para localizar os caminhos de arquivos órfãos pode-se usar o comando **find** com o argumento **-nouser**.

Você Sabia?

- A administração de usuários é muito importante em sistemas operacionais. Por questões de segurança a pessoa não pode utilizar senhas que possam ser facilmente descobertas e deve-se tomar cuidado para que um usuário não possa acessar os arquivos pessoais de outros usuários.
- Cada sistema operacional possui sua particularidade para o gerenciamento de usuário. Alguns sistemas operacionais possuem telas gráficas para facilitar o processo. No entanto, o princípio de gerência é o mesmo em todos os sistemas operacionais. São usuários ligados a grupos e permissões para usuários ou grupos de usuários (em segurança chamamos este tipo de controle de discricionário, pois ocorre a descrição das permissões). O sistema operacional Solaris (ex-Sun e atualmente pertencente a Oracle) implementa o conceito de administração de usuários baseado em papéis (*role based access control*) no qual, em vez de grupos se tem atribuições do que pode ser realizado no sistema operacional. Um usuário pode receber uma ou mais atribuições (papéis).



89

Atividades

- 1) Crie os grupos aluno_mesa01, aluno_mesa02 e aluno_mesa03.
- 2) Crie os usuários aluno_01 e aluno_02 com o grupo aluno_mesa01.
- 3) Crie os usuários aluno_03 e aluno_04 com o grupo aluno_mesa02.
- 4) Crie os usuários aluno_05 e aluno_06 com o grupo aluno_mesa03.
- 5) Crie o grupo alunos_geral.
- 6) Ligue todos os alunos_0X ao grupo alunos_geral.
- 7) Crie o usuário admina.
- 8) Ligue o usuário admina com os grupos aluno_mesa01, aluno_mesa02, aluno_mesa03, root e alunos_geral.
- 9) Mude o shell dos usuários ligados aos grupos aluno_mesa01 e aluno_mesa03 para /bin/sh.

O Sistema de Arquivo /procfs

O **procfs** (*Proc File System – Sistema de Arquivos Proc*), ou simplesmente **/proc**, é um sistema de arquivos diferente, que funciona como uma janela para o núcleo do sistema operacional. O **procfs** é chamado de sistema de arquivos “virtual” porque, na realidade, não referencia arquivos reais, os arquivos encontrados no **/proc** não existem em dispositivo físico algum, nem possuem seu conteúdo fixado em blocos de dados, como os arquivos comuns.

O entendimento do sistema de arquivo é importante para compreendermos aspectos de gerência de memória e processos do sistema operacional.

Por exemplo, o arquivo **/proc/uptime** apresenta duas informações sobre o sistema: o tempo de execução, em segundos, do sistema desde que foi ligado pela última vez, e, deste total, quantos segundos o processador ficou desocupado.

O conteúdo do arquivo pode ser lido com o comando **cat <arquivo>**:

```
$ cat /proc/uptime
54040.81 46805.64
```

90

Atividade

1. Use o comando **cat** para ler o arquivo **/proc/uptime**, como no exemplo acima.

Com o comando **cat**, percebe-se que o arquivo apresenta algum conteúdo (o tempo de execução e o tempo ocioso do sistema), ou seja, se o arquivo estivesse armazenado em um dispositivo, como um disco rígido por exemplo, seu tamanho não poderia ser zero. Porém ao usar o comando **ls -l <arquivo>** para verificar seu tamanho (quinto campo da saída, após a segunda palavra *root*), o comando **ls** apresenta como resposta zero:

```
$ ls -l /proc/uptime
-r--r--r-- 1 root root 0 2009-11-29 22:28 uptime
```

Atividade

2. Verifique se o arquivo **uptime** do seu Linux também possui tamanho 0, como no exemplo acima.

Isto ocorre porque estes arquivos são na realidade ligações para informações como: variáveis do núcleo, parâmetros internos do núcleo, estruturas de dados do núcleo e dados calculados com informações do núcleo. Como esses arquivos não são arquivos comuns gravados em um disco, eles têm tamanho zero.

Estes dados são informações internas do núcleo e seus conteúdos não são estáticos, como arquivos comuns, e podem variar com o tempo. Ao ler duas vezes o arquivo **/proc/uptime**, o seu conteúdo não será o mesmo que o lido na primeira vez, pois o tempo de execução do sistema aumentou:

```
$ cat /proc/uptime  
55484.10 48084.44  
$ cat /proc/uptime  
55484.67 48084.99
```



Atividade

Repita o procedimento apresentado acima.

Como as informações apresentadas no **/proc** são dados internos do núcleo, estes podem ser usados para verificar o estado da execução do núcleo e do sistema como um todo. Algumas das informações que podem ser acessadas usando o **/proc** são:

- informações sobre os processos existentes no sistema;
- informações sobre o uso de processador;
- informações sobre o uso de memória;
- informações sobre os sistemas de arquivos.

Os **procfs** existem em vários sistemas operacionais Unix como: Linux, BSD e Solaris. E podem ser montados em qualquer endereço, porém comumente são acessados pelo endereço **/proc/**. Seu conteúdo pode variar de acordo com a versão do núcleo do sistema e a versão do **/proc** instalado. Alguns módulos de núcleo também podem alterar o conteúdo do **/proc**.

Para descobrir onde o **/proc** está montado, usa-se o comando **mount** sem utilizar parâmetros, desta forma o comando exibe todas os sistemas de arquivos montados no sistema, dentre eles o **/proc**.

```
$ mount  
/dev/sda7 on / type reiserfs (rw,relatime,notail)  
proc on /proc type proc (rw,noexec,nosuid,nodev)  
/sys on /sys type sysfs (rw,noexec,nosuid,nodev)  
varrun on /var/run type tmpfs (rw,noexec,nosuid,nodev,mode=0755)  
varlock on /var/lock type tmpfs (rw,noexec,nosuid,nodev,mode=1777)  
udev on /dev type tmpfs (rw,mode=0755)  
devshm on /dev/shm type tmpfs (rw)  
devpts on /dev/pts type devpts (rw,gid=5,mode=620)  
1rm on /lib/modules/2.6.24-25-generic/volatile type tmpfs (rw)
```

Atividade

E no seu computador, onde está montado o **/proc**? Descubra usando o comando **mount**.

Estrutura do /proc

Após descobrir onde o `/proc` está montado usa-se o comando `cd <diretório>` para entrar no `/proc`. Para listar todo o seu conteúdo usa-se o comando `ls`:

\$ cd /proc	
\$ ls	
1 2186 3134 35 3943 5	devices mounts
10 2187 3156 3546 3945 581	diskstats mtrr
10557 22 3159 3574 3947 6	dma net
11 2257 3160 36 3948 6572	driver pagetypeinfo
11098 23 32 3601 3950 6586	execdomains partitions
11111 2302 3223 3661 3952 6587	fb sched_debug
11126 2325 3253 3682 3954 6588	filesystems schedstat
11127 2327 33 37 3955 6780	fs scsi
11128 2350 3303 3774 3963 6782	interrupts self
11161 2374 3304 38 3969 7	iomem slabinfo
12 24 3306 3831 3972 7117	ioports stat
13 2434 3308 3834 3975 7118	irq swaps
14 25 3324 3835 3979 755	kallsyms sys
15 26 3359 3840 3982 8	kcore sysrq-trigger
16 29 3362 3841 3985 889	key-users sysvipc
17 2952 336 6 3843 3988 8920	kmsg timer_list
18 2956 3388 3858 3990 9	kpagecount timer_stats
19 2962 3393 3861 3999 acpi	kpageflags tty
1959 2964 3396 3867 4 asound	latency_stats uptime
2 2968 34 3874 4009	buddyinfo loadavg version
20 3 3412 3876 4015 bus	locks version_signature
21 30 3413 3881 4016 cgroups	mdstat vmallocinfo
2177 31 3436 39 4018 cmdline	meminfo vmnet
2178 3105 3470 3940 4021 cpufreq	misc vmstat
2185 3130 348 3941 4095 crypto	modules zoneinfo

O `/proc` é composto por diretórios (aqui representados em negrito) e arquivos, também há vários diretórios que possuem seus nomes representados apenas por números. Cada arquivo/diretório possui alguma informação do sistema, como o arquivo `/proc/uptime` que apresenta o tempo em segundos de execução do sistema. Outro exemplo é o arquivo `/proc/cmdline` que contém as informações de como o *kernel* do Linux foi inicializado. Para descobrir a lista completa de todas as informações que o `/proc` pode apresentar, usa-se o comando `man proc`.

Como algumas das informações apresentadas no **/proc** podem comprometer a segurança do sistema e alguns arquivos do **/proc** podem receber dados para alterar configurações do sistema, nem todos os arquivos podem ser acessados por qualquer usuário. Por este motivo, para acessar completamente o conteúdo do **/proc** é necessário fazer isso como **superusuário**.

Alguns arquivos encontrados no **/proc**, principalmente no **/proc/sys**, podem também receber dados, por exemplo: o arquivo **/proc/sys/hostname**, que informa o nome de *host* do sistema, pode ser lido com o comando **cat** e editado com os comandos **echo <texto> <arquivo>**. Como esta operação altera parâmetros do núcleo só pode ser realizada pelo superusuário e com cuidado, pois operações erradas de escrita no núcleo podem parar todo o sistema.

Superusuário, *root* ou administrador é um usuário especial de um sistema operacional, que possui permissão para acessar todos os recursos, desta forma o *root* pode configurar o sistema, alterar usuários e grupos e alterar propriedades de processos, entre outras coisas.

```
$ cat /proc/sys/kernel/hostname  
oldname  
# echo newname > /proc/sys/kernel/hostname  
$ cat /proc/sys/kernel/hostname  
newname
```



Atividade

Qual o nome do seu computador? Use o comando **cat /proc/sys/kernel/hostname** para descobrir.

Como mencionado anteriormente, os arquivos e diretórios do **/proc** apresentam dados do núcleo do sistema. Como esses dados não estão armazenados em dispositivos físicos como um disco rígido, eles devem ser lidos no núcleo antes de serem apresentados ao processo que os solicitou, é necessário efetuar cálculos durante este processo.

Para evitar o uso desnecessário de recursos como processador e memória, os dados são gerados apenas quando os arquivos do **/proc** são usados. Por exemplo, os dados do arquivo **/proc/uptime** não existem até o momento que o comando **cat** tente ler seu conteúdo. Neste momento, o tempo de execução do sistema é calculado e entregue ao comando **cat** para então ser impresso na tela.

Os diretórios nomeados apenas com números representam os processos existentes no sistema, cada processo do sistema possui uma pasta nomeada com seu PID (*Process IDentifier* – Identificador de Processo). O PID é um número usado para identificar unicamente um processo dentro do sistema. Todos os processos possuem um PID e um diretório no **/proc** com informações sobre o processo.

As pastas que não são nomeadas com números apresentam informações sobre o uso específico de um recurso do sistema. Por exemplo, o diretório **/proc/net** contém informações sobre o uso das interfaces de rede disponíveis e a pasta **/proc/fs** possui informações sobre os sistemas de arquivos.

Os arquivos que não estão em pastas também apresentam informações gerais do sistema. Por exemplo, o arquivo **/proc/uptime**, mencionado anteriormente, o arquivo **/proc/cpuinfo** contém informações sobre o processador usado pelo sistema; e o arquivo **/proc/meminfo** possui informações sobre a memória do sistema.

Utilizando o /proc

O **/proc** pode ser usado para obter informações gerais sobre o sistema, essas informações podem ser sobre o *hardware*, o *software* ou sobre o estado do sistema. A seguir serão vistos alguns dos arquivos que contém tais informações, e o significado de alguns dos seus campos.

Para obter dados sobre o processador usado pelo sistema, existe o arquivo **/proc/cpuinfo** que apresenta uma descrição do processador. Esse arquivo pode ser lido com o comando **cat** e seu conteúdo varia de acordo com o processador.

```
$ cat /proc/cpuinfo
Processor       : 0
vendor_id      : AuthenticAMD
Cpu family    : 6
model          : 10
model name     : AMD Athlon(tm) XP 2600+
Stepping       : 0
cpu Mhz        : 1920.500
cache size     : 512 KB
fdiv_bug       : no
hlt_bug        : no
f00f_bug       : no
coma_bug       : no
fpu            : yes
fpu_exception  : yes
cpuid level   : 1
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 mmx fxsr sse syscall mmxext 3dnowext
3dnow up
Bogomips       : 3849.19
clflush size   : 32
power management : ts
```

94

Atividade

Use o comando **cat /proc/cpuinfo** para descobrir qual processador seu computador está usando.

O primeiro campo deste arquivo identifica o processador, caso existam mais de um processador na máquina. Neste exemplo, o processador instalado é um AMD Athlon XP 2.6 Ghz (Família 6, Modelo 10), que opera em uma frequência de aproximadamente 1.9 Ghz e possui 512 KB de memória cache. No campo Flags percebe-se que esse processador suporta algumas tecnologias para tratamento de gráficos 3d, como a mmx e a 3dnow.



Para obter informações sobre a memória do sistema existe o arquivo **/proc/meminfo** que apresenta um panorama geral sobre a memória. Quando instanciados, os comandos **free** e **top** usam os dados contidos neste arquivo para calcular e exibir as informações sobre o uso de memória.

```
$ free
      total        used        free      shared      buffers      cached
Mem:  2579624   2452940   126684          0      233112   1239020
      -/+ buffers/cache:   980808   1598816
Swap:      979956          0   979956
$ cat /proc/meminfo
MemTotal:           2579624 kB
MemFree:            231116 kB
Buffers:              232376 kB
Cached:                1238712 kB
SwapCached:            0 kB
Active:                1332776 kB
Inactive:              919932 kB
Active(anon):          794396 kB
Inactive(anon):         0 kB
Active(file):          538380 kB
Inactive(file):         919932 kB
Unevictable:            0 kB
Mlocked:                0 kB
HighTotal:              1716168 kB
HighFree:                15128 kB
LowTotal:               863456 kB
LowFree:                215988 kB
SwapTotal:           979956 kB
SwapFree:            979956 kB
Dirty:                  100 kB
Writeback:                0 kB
AnonPages:              781616 kB
Mapped:                 164372 kB
```

```
Slab:           64732 kB
SReclaimable:   54664 kB
SUnreclaim:     10068 kB
PageTables:     4992 kB
NFS_Unstable:   0 kB
Bounce:         0 kB
WritebackTmp:   0 kB
CommitLimit:    2269768 kB
Committed_AS:   1857424 kB
VmallocTotal:   122880 kB
VmallocUsed:    45436 kB
VmallocChunk:   69108 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize:   4096 kB
DirectMap4k:    28664 kB
DirectMap4M:    876544 kB
```

Atividade

Use o comando `cat/proc/meminfo` para descobrir quanta memória há livre, no seu computador.

Os principais campos deste arquivo são:

- **MemTotal** – informa a quantidade total de memória disponível.
- **MemFree** – significa a quantidade total de memória livre no sistema.
- **SwapTotal** – é a quantidade total de swap disponível.
- **SwapFree** – a quantidade de swap livre.

Usando as informações desses campos, é possível calcular a quantidade de memória usada pelo sistema. Por exemplo, para saber quanta memória residente na RAM foi usada, subtrai-se da memória total a memória livre.

```
MemóriaUsada = MemTotal - MemFree
MemóriaUsada = 2579624kB - 231116kB = 2348508kB
```

O arquivo **/proc/partitions** apresenta todas as partições (volumes) do sistema, os identificadores, os números de blocos e o nome das partições. Um exemplo de uso pode ser visto a seguir. O arquivo mostra que há uma unidade de disco chamada **sda**, a qual está dividida em 7 partições (**sda1**, **sda2**, ...**sda7**)

```
$ cat /proc/partitions
major minor   #blocks  name
    8        0   156290904  sda
    8        1   40000826  sda1
    8        2   20000925  sda2
    8        3           1  sda3
    8        4   48203032  sda4
    8        5   30716280  sda5
    8        6   15317946  sda6
    8        7   2048224  sda7
```



Atividade

Descubra quantas partições há em seu disco usando o arquivo **/proc/**.

O arquivo **/proc/filesystems** apresenta uma lista de todos os sistemas de arquivos suportados pelo núcleo. Os sistemas de arquivos marcados como **nodev** são partições que não necessitam de dispositivos de bloco para serem montados, como o **/proc**. O comando **mount** pode usar este arquivo para montar partições quando o sistema de arquivos não é especificado.

```
$ cat /proc/filesystems
nodev      sysfs
nodev      rootfs
nodev      bdev
nodev      proc
nodev      cgroup
nodev      cpuset
nodev      debugfs
nodev      securityfs
nodev      sockfs
nodev      usbfs
nodev      pipefs
nodev      anon_inodefs
nodev      tmpfs
```

```
nodev         inotifyfs
nodev         devpts
              ext3
              ext4
              ext2
              cramfs
nodev         ramfs
nodev         hugetlbfs
nodev         ecryptfs
nodev         fuse
              fuseblk
nodev         fusectl
nodev         mqueue
nodev         binfmt_misc
```

Atividade

Descubra quantos sistemas de arquivo há em seu sistema.



98

O arquivo **/proc/version** apresenta informações sobre a versão do núcleo que está em execução. Um exemplo da saída deste arquivo pode ser visto abaixo.

```
$ cat /proc/version
Linux version 2.6.28-15-generic (buildd@rothera) (gcc version
4.3.3 (Ubuntu 4.3.3-5ubuntu4) ) #52-Ubuntu SMP Wed Sep 9 10:49:34
UTC 2009
```

Segundo o arquivo, a versão do núcleo do sistema é 2.6.28-15-generic foi compilada pelo usuário buildd da máquina rothera, por meio do compilador gcc com a versão 4.3.3 no dia 09 de setembro de 2009, às 10:49.

Gerência de Processos

Um **programa** é um conjunto de uma ou mais sequências de instruções escritas para resolver um problema específico, constituindo, assim, uma aplicação ou utilitário. O programa representa um **conceito estático**, sem um estado interno definido (que represente uma situação específica da execução) e sem interações com outras entidades (o usuário ou outros programas).

Uma **tarefa** é a execução, pelo processador, das sequências de instruções definidas em um programa para realizar seu objetivo. Trata-se de um **conceito dinâmico**, que possui um estado interno bem definido a cada instante (os valores das variáveis internas e a posição atual da execução) e interage com outras entidades: o usuário, os periféricos e/ou outras tarefas. Tarefas podem ser implementadas de várias formas, como processos ou **threads**.

Fazendo uma analogia, pode-se dizer que um programa é o equivalente de uma “receita de torta” dentro de um livro de receitas (um diretório) guardado em uma estante (um disco) na cozinha (o computador). Essa receita de torta define os ingredientes necessários e o modo de preparo da torta. Por sua vez, a ação de “executar” a receita, providenciando os ingredientes e seguindo os passos definidos na receita, é a tarefa propriamente dita. A cada momento, a cozinheira (o processador) está seguindo um passo da receita (posição da execução) e tem a sua disposição certos ingredientes e utensílios em uso (as variáveis internas da tarefa).

Programas e Processos

Como vimos anteriormente, **programas** são arquivos em disco, contendo instruções para execução pelo processador, enquanto **processos** são as execuções em andamento. Processo também poderia ser definido como um programa em execução.

No Linux, cada processo executado no sistema em um determinado momento é identificado por um número único, o **PID – Process IDentifier**. Além disso, cada processo possui outras informações que o caracterizam, como:

- Usuário proprietário (aquele que lançou o processo).
- Sessão de shell de onde foi lançado (se foi lançado por meio de um shell).
- Estado atual (Running, Suspended, Swapped, ...).
- Linha de comando usada para lançá-lo.
- Uso de memória e CPU.

Comandos ps e pstree

Podemos visualizar os processos em execução no sistema por meio do comando **ps**, cuja execução sem parâmetros gera uma listagem como a seguinte:

```
$ ps
  PID TTY      TIME CMD
 3134 pts/0    00:00:00 bash
 3139 pts/0    00:00:00 ps
```

O comando **ps** aceita uma série de parâmetros, entre os quais os mais importantes são:

- a** – mostra processos de outros usuários também (*all*);
- u** – mostra listagem mais detalhada dos processos, com uso de memória e de CPU;
- x** – mostra processos não conectados a terminais;
- w** – mostra mais detalhes sobre as linhas de comando dos processos.

Para obter uma listagem completa dos processos em execução no sistema operacional, usam-se as opções **auxw**, que geram uma listagem como a que segue:

100

Gerência de Processos

```
$ ps auxw
USER  PID %CPU %MEM   VSZ   RSS TTY STAT START   TIME COMMAND
root   1  0.0  0.0 19440 1752 ? Ss 14:35 0:00 /sbin/init
root   2  0.0  0.0     0     0 ? S< 14:35 0:00 [kthreadd]
root   3  0.0  0.0     0     0 ? S< 14:35 0:00 [migration/0]
root   4  0.0  0.0     0     0 ? S< 14:35 0:00 [ksoftirqd/0]
root   5  0.0  0.0     0     0 ? S< 14:35 0:00 [watchdog/0]
root   6  0.0  0.0     0     0 ? S< 14:35 0:00 [migration/1]
root   7  0.0  0.0     0     0 ? S< 14:35 0:00 [ksoftirqd/1]
root   8  0.0  0.0     0     0 ? S< 14:35 0:00 [watchdog/1]

----- resultado suprimido propositalmente ----->
```

Os principais campos dessa listagem são:

- **USER** – o proprietário do processo, que pode ser quem o lançou ou, no caso de executáveis com o bit SUID habilitado, o proprietário do arquivo executável.
- **PID** – número do processo.

- **%CPU** – porcentagem da CPU usada pelo processo.
- **%MEM** – porcentagem da memória usada pelo processo.
- **SIZE** – memória total usada pelo processo.
- **RSS** – memória física (RAM) usada pelo processo.
- **TTY** – terminal ao qual o processo está ligado.
- **STAT** – status do processo (rodando, suspenso, ...).
- **START** – data de lançamento do processo.
- **TIME** – tempo total de CPU usado pelo processo.
- **COMMAND** – comando usado para lançar o processo.

O comando **pstree** é útil por mostrar a hierarquia existente entre os processos ativos no sistema:

```
$init--NetworkManager---dhclient
|           `-{NetworkManager}
|-acpid
|-atd
|-avahi-daemon---avahi-daemon
|-bonobo-activati---{bonobo-activati}
|-console-kit-dae---63*[{console-kit-dae}]
|-couchdb---couchdb---beam.smp---heart
|           `--5*[{beam.smp}]
|-cron
|-cupsd
|-2*[dbus-daemon]
|-2*[dbus-launch]
|-dd
|-devkit-disks-da---devkit-disks-da
|-devkit-power-da
|-evolution-data---2*[{evolution-data-}]
|-firefox---npviewer.bin---2*[{npviewer.bin}]
|   |-npviewer.bin---acoread---{acoread}
|   `--8*[{firefox}]
|-gconfd-2

<----- resultado suprimido propositalmente ----->
```

101

Gerência de Processos



Atividade

Use o comando **pstree** para ver a árvore de processos.

Obtendo Informações sobre Processos no /proc

O **/proc** disponibiliza diversas informações sobre processos, como o número total de processos e os arquivos abertos por cada processo. Como mencionado anteriormente, cada processo do sistema possui um diretório nomeado com o seu PID onde estão as informações individuais de cada processo. No diretório há vários arquivos e pastas, como pode ser visto com o comando **ls**.

```
$ ls /proc/4205/
attr          environ   maps      pagemap    statm
auxv          exe       mem       personality status
cgroup        fd        mountinfo root      syscall
clear_refs   fdinfo   mounts    sched      task
cmdline      io        mountstats schedstat wchan
coredump_filter latency  net       sessionid
cpuset        limits   oom_adj   smaps
cwd           loginuid oom_score stat
```

102

Como ocorre na raiz do **/proc**, há diretórios (em negrito) e arquivos sobre o uso de recursos, porém as informações aqui apresentadas são sobre o uso de um certo recurso por um certo processo. Por exemplo, o arquivo **/proc/4205/mem** apresenta informações sobre o uso de memória pelo processo com o PID 4205. Para descobrir qual é este processo, pode-se ler o arquivo **/proc/4205/cmdline**.

```
$ cat /proc/4205/cmdline
bash
```

O arquivo **/proc/PID/cmdline** exibe o comando usado para criar esse processo, identificando-o. Para descobrir o PID de um certo processo pode-se usar os comandos **ps a**, **|** e **grep**.

Será usada a notação [PID] para indicar que qualquer PID existente no sistema pode ser usado neste endereço. Por exemplo, para acessar o arquivo **/proc/[PID]/cmdline** do processo com PID 4205 deve-se substituir [PID] pelo PID do processo desejado da seguinte forma: **/proc/4205/cmdline**.

- O comando **ps a** serve para listar todos os processos do sistema.
- O comando **|** (pipe) redireciona a saída de um programa da tela para outro programa.
- E o comando **grep** encontra ocorrências de uma palavra dentro de um texto.

Desta forma, usa-se o comando **ps a | grep [processo]** para listar todos os processos com um determinado nome. A saída deste comando são uma ou mais linhas, sendo cada linha um processo com o nome passado ao comando **grep**. Essas linhas são divididas em cinco campos, sendo que o primeiro campo é o PID do processo e o último campo o comando usado para criá-lo.

Por exemplo, para descobrir qual é o PID do processo bash, usa-se o seguinte comando: **ps a | grep bash**. Nota-se que o comando **grep bash** também é listado.

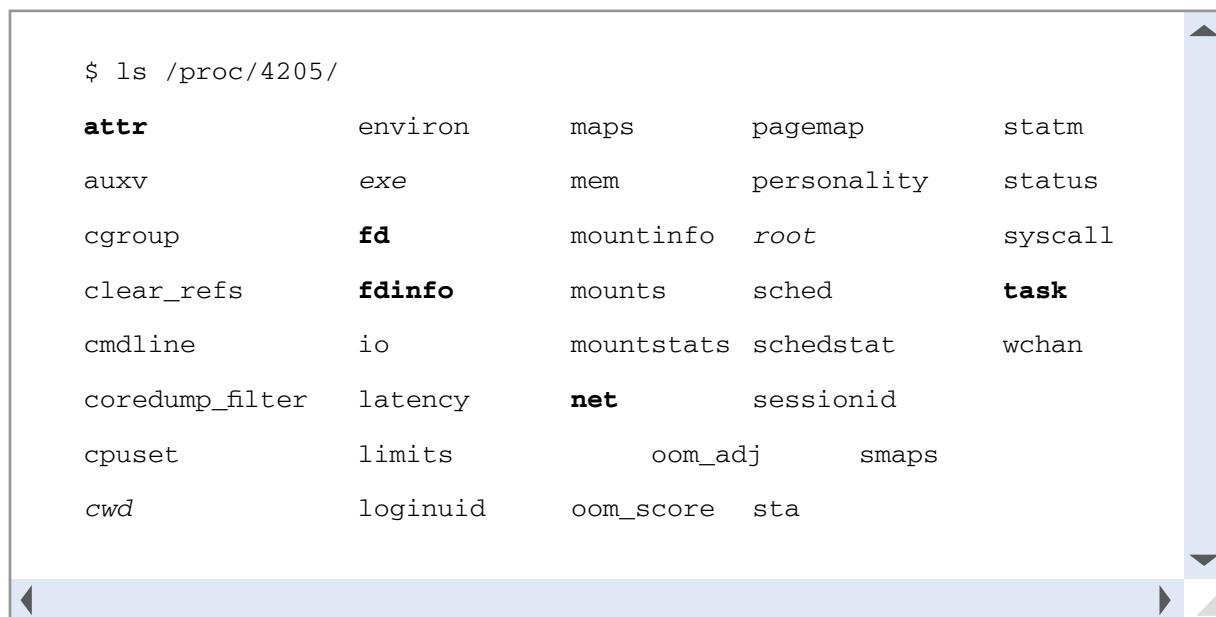
```
$ ps a | grep bash
4205 pts/0      Ss    0:00  bash
4185 pts/1      Ss    0:00  bash
5295 pts/0      S+    0:00  grep  bash
```

Segundo a saída do comando, há dois processos chamados de bash, um com o PID 4205 e outro com o PID 4185. Além desses dois processos, o comando **grep bash** também foi listado, pois possui a palavra bash como parte do seu nome. Porém ao tentar acessar a pasta com o PID do processo grep ocorrerá um erro:

```
$ cd /proc/5295
bash: cd: /proc/5295: Arquivo ou diretório inexistente
```

Isso ocorre porque o processo grep com o PID 5295 já foi encerrado, portanto não há uma pasta para ele no **/proc**. Comandos que possuem uma duração muito curta, como o **grep** que dura menos de um segundo, dificilmente podem ser acessados por um usuário, pois não há tempo de digitar o comando para acessá-lo no terminal. Para acessar os arquivos, é necessário um programa que realize o acesso.

Na pasta de um processo podem ser encontradas várias informações sobre ele. Com o comando **ls /proc/PID/** é listado o conteúdo da pasta, e seus arquivos/diretórios mais importantes são:



```
$ ls /proc/4205/
attr      environ   maps    pagemap   statm
auxv      exe       mem    personality   status
cgroup    fd        mountinfo  root    syscall
clear_refs fdinfo   mounts   sched    task
cmdline   io        mountstats  schedstat  wchan
coredump_filter latency  net    sessionid
cpuset    limits   oom_adj   smaps
cwd       loginuid  oom_score sta
```

- **cmdline**: exibe o comando usado para instanciar o processo;
- **exe**: trata-se de um *link* para o arquivo executável do processo;
- **fd**: diretório contendo um *link* para cada arquivo aberto pelo processo;
- **fdinfo**: diretório contendo um arquivo com informações sobre cada um dos arquivos;
- **sched**: arquivo contendo informações sobre o escalonamento do processo;
- **status**: apresenta uma visão geral sobre o estado do processo;
- **task**: diretório que contém outros diretórios, um para cada **thread** do processo.

Para obter mais informações sobre outros arquivos, ou sobre os conteúdos dos arquivos, usa-se o comando **man proc**.

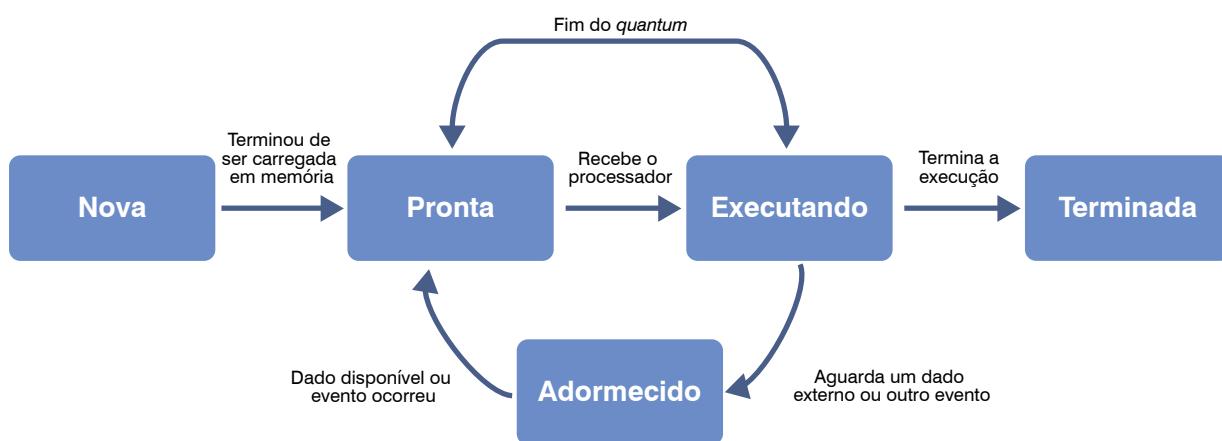
Estados de Processos no Linux

Um processo não é automaticamente elegível para receber tempo de CPU simplesmente porque ele existe. Há, basicamente, quatro estados de execução sobre os quais deve-se estar ciente:

- **executável ou pronto** – processo pode ser executado;
- **dormente ou bloqueado** – o processo está aguardando algum recurso;
- **zumbi** – o processo está tentando se destruir (finalizando e saindo da fila);
- **parado** – o processo é suspenso (não há permissão para ser executado).

Um processo executável é aquele que está pronto para ser executado toda vez que houver tempo de CPU disponível. Ele obteve todos os recursos necessários e está apenas esperando o tempo de CPU (*quantum*) para processar seus dados. Assim que o processo fizer uma chamada de sistema que não pode ser completada imediatamente (como uma solicitação de leitura de parte de um arquivo), o Linux o colocará no estado de dormência.

Os processos em estado de dormência estão aguardando a ocorrência de um evento específico. Daemons e sistemas e shells interativos passam grande parte de seu tempo no estado de dormência, aguardando a entrada via terminal (teclado) ou conexões de rede. Já que um processo em estado de dormência é efetivamente bloqueado até que sua solicitação seja atendida, ele não obterá nenhum tempo de CPU, a menos que receba um sinal. Zumbis são processos que terminaram a execução, porém ainda não tiveram seus estados coletados.



Processos parados são administrativamente proibidos de serem executados. Os processos são interrompidos no recebimento de um sinal STOP e são reiniciados com um sinal CONT. Ser interrompido é similar a adormecer, porém não há nenhum jeito de sair do estado interrompido a não ser com algum outro processo que os faça acordar ou os mate. Os sinais STOP e CONT, juntamente com outros sinais, serão abordados mais adiante no tópico **Sinais no Linux**.

Voltando ao Comando ps

Utilizando o comando `ps -O stat` é obtida uma lista como se segue:

```
$ ps -O stat
PID STAT S TTY          TIME COMMAND
3134 Ss   S pts/0        00:00:00 bash
3260 R+   R pts/0        00:00:00 ps -O stat
```

As colunas **STAT** e **S** demonstram os estados que um processo no Linux pode ter:

Coluna Stat

- D**: suspenso (*sleep*) e não pode ser interrompido (normalmente, realizando operações de **E/S**);
- R**: em execução (*running*);
- S**: suspenso (*sleep*), aguardando um evento para continuar;
- T**: parado (*stopped*), por meio da intervenção do usuário ou de outro processo;
- X**: morto (*dead*), já saiu da fila de execução dos processos (esta informação, embora suportada pelo comando **\verb|ps|**, nunca deveria ser vista);
- Z**: processo zumbi (*defunct*), já terminou, mas seu *status* de término ainda está disponível no sistema.

Colunas

- <**: processo com alta prioridade (não pode ser modificado por outros usuários);
- N**: processo com baixa prioridade (pode ser modificado por outros usuários);
- L**: processo com páginas de memória presas (*locked*);
- s**: processo líder da sessão;
- I**: processo **multi-thread**;
- +**: dentro do grupo de processos está em primeiro plano (*foreground*);

106

Multiplexação do Processador

Listando o conteúdo do **/proc**, nota-se que apesar de um computador comum possuir um ou alguns processadores, há muito mais processos do que processadores no sistema. Para resolver esse problema, o sistema operacional deve multiplexar (compartilhar) o processador, de forma que todos os processos consigam usá-lo. Esta multiplexação do processador pode ser notada acessando o arquivo **/proc/stat**.

```
$ cat /proc/stat
cpu 385679 27990 58865 970180 8808 1932 185 0 0
cpu0 192769 9781 35292 468571 6945 1932 156 0 0
cpu1 192910 18208 23572 501608 1862 0 28 0 0
intr 5183797 1214311 12617 0 0 0 0 ...
...
ctxt 8962097
btime 1259666387
processes 6396
procs_running 4
procs_blocked 0
```

Segundo o arquivo, existem dois processadores (cpu0 e cpu1, sendo que o cpu é a soma do cpu0 e do cpu1) instalados no sistema. Durante a execução do sistema, já foram criados 6396 processos (*processes*), sendo que no momento da leitura do arquivo quatro processos (**procs_running**) estavam concorrendo pelo processador. O sistema já realizou 8962097 trocas de contexto (ctxt) desde o momento em que a máquina foi ligada para dividir os dois processadores entre os 6396 processos.

Além de descobrir quantas trocas de contexto o sistema já realizou, também é possível descobrir quantas trocas de contexto um processo já realizou, para isso pode ser usado o arquivo **/proc/PID/status**. Usando como exemplo o bash com PID 4205:

```
$ cat status
Name: bash
State: S (sleeping)
Tgid: 4205
Pid: 4205
PPid: 4180
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 4 20 24 46 106 121 122 1000
VmPeak: 6440 kB
VmSize: 6440 kB
VmLck: 0 kB
VmHWM: 3668 kB
VmRSS: 3668 kB
VmData: 2356 kB
VmStk: 84 kB
VmExe: 692 kB
VmLib: 1888 kB
VmPTE: 16 kB
Threads: 1
SigQ: 0/16382
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
```

```

SigBlk:      0000000000010000
SigIgn:      0000000000384004
SigCgt:      000000004b813efb
CapInh:      0000000000000000
CapPrm:      0000000000000000
CapEff:      0000000000000000
CapBnd:      ffffffff
Cpus_allowed:    00000000,00000003
Cpus_allowed_list: 0-1
Mems_allowed:   1
Mems_allowed_list: 0
Voluntary_ctxt_switches:      113
Nonvoluntary_ctxt_switches: 68

```

Atividade

Descubra quantas trocas de contexto o seu sistema já realizou.



108

Segundo o arquivo, o processo já realizou 113 trocas de contexto voluntárias (**Voluntary_ctxt_switches**) e 68 trocas involuntárias (**Nonvoluntary_ctxt_switches**), totalizando 181 trocas de contexto. Como visto, as trocas de contexto podem ocorrer de duas formas:

- **Trocas Voluntárias** – ocorrem quando o processo abdica voluntariamente do processador. Isso ocorre principalmente quando um processo está realizando operações de E/S (Entrada e Saída), como a leitura de um arquivo do disco.
- **Trocas Involuntárias** – ocorrem quando um processo está usando o processador, porém seu *quantum* de tempo acabou e o scheduler retira o processo do processador, para que este possa ser usado em outra tarefa.

Processos Orientados a E/S e a Processamento

Com base nos dados sobre as trocas de contexto voluntárias ou não, é possível classificar os processos em duas categorias: processos orientados a processamento e processos orientados a E/S.

- **Processos Orientados a Processamento** são processos que realizam muitos cálculos e poucas operações de E/S, desta forma, o número de trocas involuntárias de contexto será muito maior que o número de trocas voluntárias.
- **Processos Orientados a Entrada e Saída** por outro lado são processos que realizam mais operações de E/S do que operações de cálculo. Um processo interativo normalmente assume este perfil, pois fica a maior parte do seu ciclo de vida no estado suspenso, aguardando a interação do usuário.

Para ilustrar esta classificação, é possível criar dois programas, um que execute diversos cálculos e outro que fique aguardando dados do teclado, depois serão analisadas as trocas de contexto realizadas na execução de cada um deles.

Um possível código orientado a processamento pode ser construído com diversas estruturas de repetição aninhadas que realizem cálculos, por exemplo:

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

int get_pid ()
{
    char target[32];
    int pid;

    readlink("/proc/self", target, sizeof(target));
    sscanf (target, "%d", &pid);

    return (pid);
}

int main (int argc, char* argv)
{
    unsigned long a;
    unsigned long b;
    unsigned long i;
    unsigned long j;
    unsigned long k;
    unsigned long l;

    printf ("Meu PID é %d\n", get_pid ());

    a = b = i = j = k = l = 0;

    while (i < 2000000000) {
        i++;
        while (j < 2000000000) {
            j++;
            while (k < 2000000000) {
```

```

        k++;
        while (l < 2000000000) {
            l++;
            a = (int)(k % l);
            b = (int)(i % j);
        }
    }

    return (0);
}

```

Este código é dividido em duas funções, a primeira função, **get_pid**, lê no **/proc** o seu próprio PID, para que fique mais fácil coletar os dados sobre o processo. A função **main** executa quatro laços de repetição e realiza operações de divisão com os índices das estruturas de repetição. Este programa é voltado para processamento e só irá realizar uma operação de entrada de dados, quando ler o seu próprio PID do **/proc**, e uma operação de saída, quando imprimir seu PID na tela.

Para criar um código orientado à entrada e saída, pode-se criar um código que receba uma frase do teclado e, depois, imprima-a na tela, como no exemplo a seguir:

110

```

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

int get_pid ()
{
    char target[32];
    int pid;

    readlink("/proc/self", target, sizeof(target));
    sscanf (target, "%d", &pid);

    return (pid);
}

int main (int argc, char* argv)
{

```

```

char c;

printf ("Meu PID é %d\n", get_pid ());

puts ("Digite . para sair");

do {
    c=getchar();
    putchar (c);
} while (c != '.');

return (0);
}

```

Este segundo código também é dividido em duas funções, sendo que a função **get_pid** é idêntica a função **get_pid** do primeiro exemplo. Porém a função **main** realiza sempre a leitura de um caractere do teclado e depois o imprime na tela, caso o caractere seja um ponto final (.) o programa é encerrado.

Para que estes códigos sejam compilados, eles devem, primeiro, ser copiados em arquivos, aqui chamados de **prog1** (voltado a processamento) e **prog2** (voltado a entrada/saída) e, depois, compilados com o seguinte comando:

```

$ gcc prog1 -o prog1
$ gcc prog2 -o prog2
$ 

```

Em seguida, devem ser executados cada um dos programas em um terminal separado, com os seguintes comandos:

```

$ ./prog1
Meu PID é 12309

$ 

```

```
$ ./prog2  
Meu PID é 12068  
Digite . para sair  
a  
a  
b  
b  
c  
c  
.  
.  
$
```

112

Desta forma, o **prog1** irá iniciar a realização dos cálculos e o **prog2** irá aguardar as entradas do teclado. O **prog1** executa diversos cálculos assim sua execução demora tempo suficiente para que seja possível ler seus arquivos no **/proc**. O **prog2** por sua vez só encerrará quando for digitado um ponto na sua entrada.

Enquanto os códigos estão sendo executados, eles irão imprimir na tela os seus PIDs, estes devem ser usados para acessar seus diretórios no **/proc** da seguinte forma:

```
$ cat /proc/12309/sched  
prog1 (12309, #threads: 1)  
-----  
nr_switches : 166  
nr_voluntary_switches : 1  
nr_involuntary_switches : 165  
policy : 0  
prio : 120  
clock-delta : 70
```

```
$ cat /proc/12068/sched
prog2 (12068, #threads: 1)

-----
nr_switches : 5
nr_voluntary_switches : 5
nr_involuntary_switches : 0
policy : 0
prio : 120
clock-delta : 171
```

O processo originado pelo **prog1**, com pid 12309, realizou durante a sua execução 166 trocas de contexto, sendo que apenas uma das trocas foi voluntária. Isso ocorreu porque o processo ficou a maior parte do tempo realizando cálculos, ou seja, usando o processador, e quando cada *quantum* do processo acabava o *scheduler* (agendador) o retirava do processador.

Nota-se que o processo originado pelo **prog2**, com PID 12068, realizou apenas 5 trocas de contexto durante a sua execução, e todas elas voluntárias. Isso por que o processo ficou a maior parte do tempo aguardando dados do teclado ou imprimindo os dados na tela. Quando processos estão aguardando dados, eles assumem o estado suspenso e não usam o processador até que o dado que ele está esperando esteja disponível.

Como visto, o **prog1** é orientado a processamento e realizou mais trocas de contexto involuntárias do que trocas de contexto voluntárias, ficou a maior parte do seu tempo de execução no processador ou aguardando o processador na fila de tarefas prontas. O **prog2** por sua vez realizou mais trocas de contexto voluntárias e ficou a maior parte do tempo na fila de tarefas suspensas.

Threads

Atualmente, vários programas precisam executar várias tarefas ao mesmo tempo, por exemplo, um sistema de segurança de um prédio precisa receber imagens de várias câmeras, gravá-las em um disco e ainda exibi-las em uma tela para que possam ser monitoradas.

Para que um único processo possa realizar mais que uma tarefa ao mesmo tempo, há as **threads**. Uma **thread** pode ser vista como um fluxo de execução de uma tarefa, como receber imagens de câmeras ou gravá-las no disco. Usando **threads** é possível que um único processo consiga executar vários fluxos de execução ao mesmo tempo.

Por exemplo, em um sistema de monitoramento de câmeras poderia existir uma **thread** para receber as imagens das câmeras, outra **thread** para gravá-las em um disco e outra para exibi-las na tela.

O `/proc` oferece diversas informações sobre as **threads** de cada processo. No arquivo `/proc/PID/status` há um campo que apresenta o número de **threads** de um processo, como mostrado a seguir:

```
$ cat /proc/13448/status
Name:    prog3
State:   S (sleeping)
Tgid: 13448
Pid: 13448
PPid: 11651
TracerPid: 0
Uid: 1001 1001 1001 1001
Gid: 1001 1001 1001 1001
FDSize: 256
Groups: 4 20 21 24 25 26 29 30 44 46 105 107 109 115 124 1001
VmPeak: 18188 kB
VmSize: 18188 kB
VmLck: 0 kB
VmHWM: 536 kB
VmRSS: 536 kB
VmData: 16568 kB
VmStk: 84 kB
VmExe: 4 kB
VmLib: 1500 kB
VmPTE: 16 kB
Threads: 3
SigQ: 0/16307
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000001800000000
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
Cpus_allowed: 03
Mems_allowed: 1
voluntary_ctxt_switches: 2
nonvoluntary_ctxt_switches: 2
```

Vale ressaltar que em processos com mais de uma **thread**, como no exemplo anterior, as informações apresentadas no arquivo **/proc/PID/status** são válidas apenas para a **thread** principal, a **thread** que lançou as outras. Por exemplo, o campo **voluntary_ctxt_switches** indica que a **thread** principal realizou duas trocas de contexto voluntárias, as **threads** deste processo podem ter realizado outra quantidade de trocas de contexto, como se fossem processos separados.

Para ler os dados sobre cada **thread** de um processo, há o diretório **/proc/PID/task/TID** (como ocorre com o PID o TID – *Thread ID ou IDentificador de Thread* – é um número que identifica unicamente uma **thread** dentro do sistema) para cada **thread** do processo. Nesse diretório, há os mesmos arquivos que há no diretório **/proc/PID**, porém com dados sobre cada uma das **threads**. Por exemplo, para descobrir quantas trocas de contexto voluntárias uma **thread** executou é possível ler o arquivo **/proc/PID/task/TID/status**.

Para exemplificar esse funcionamento, podemos criar um código que lance, além da **thread** principal, mais duas. No exemplo abaixo a função **main** (**thread** principal) cria mais duas **threads** (função **func**) e espera que elas terminem sua execução. Cada **thread** por sua vez irá dormir por 180 segundos para que haja tempo de acessar os conteúdos dos arquivos **/proc/PID/task/TID**. O código do exemplo pode ser visto a seguir:

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

int get_pid (){
    char target[32];
    int pid;

    readlink("/proc/self", target, sizeof(target));
    sscanf (target, "%d", &pid);

    return (pid);
}

int func (void *threaddid){
    int i;
    for (i=0; i<180; i++){
        sleep(1);
    }
}
```

```

int main (int argc, char *argv[]){
    pthread_t thread[2];
    int status, i;

    printf ("Meu PID é %d\n", get_pid ());

    //cria mais duas threads
    for(i=0; i<2; i++){
        printf("Criando thread %d\n", i);
        status = pthread_create(&thread[i], NULL,
(void*) func, (void*) i);
        if (status){
            exit(-1);
        }
    }

    // Aguarda a execução das outras threads
    for(i=0; i<2; i++){
        printf("Esperando execução da thread %d\n", i);
        status = pthread_join(thread[i], NULL);
        if (status){
            exit(-1);
        }
    }

    pthread_exit(NULL);
}

```

116

Para compilar o código, é necessário incluir a biblioteca **pthread**, que é responsável pela criação de **threads** em sistemas UNIX, com o seguinte comando:

```

$ gcc prog3.c -lpthread -o prog3
$ ./prog3
Meu PID é 13448
Criando thread 0
Criando thread 1
Esperando execução da thread 0
Esperando execução da thread 1
$ 

```

Ao executar o código, percebe-se que foram criados três diretórios no diretório **/proc/12068/task**:

```
$ ls /proc/13448/task/  
13448 13449 13450
```

Nesses diretórios, há as mesmas informações que no diretório de um processo, exceto as informações que não fazem sentido para as **threads**, como informações sobre **threads** (pasta **task**), pois essas informações estão associadas ao processo e não a cada uma.

```
$ ls /proc/13448/  
attr cpuset io mountinfo pagemap smaps wchan  
auxv cwd latency mounts personality stat  
cgroup environ limits mountstats root statm  
clear_refs exe loginuid net sched status  
cmdline fd maps oom_adj schedstat syscall  
coredump_filter fdinfo mem oom_score sessionid task  
  
$ ls /proc/13448/task/13449/  
attr cpuset fdinfo maps oom_score schedstat status  
auxv cwd io mem pagemap sessionid syscall  
cgroup environ latency mountinfo personality smaps wchan  
clear_refs exe limits mounts root stat  
cmdline fd loginuid oom_adj sched statm
```

Ao ler os arquivos **/proc/13448/sched**, que apresentam dados sobre o escalonamento da **thread** principal, e os arquivos **/proc/13448/task/13449/sched**, que apresentam dados sobre o escalonamento da primeira que foi lançada, percebe-se que a execução de cada **thread** é independente, ou seja, a **thread main** pode realizar apenas o lançamento das outras, enquanto que cada uma pode realizar cálculos ou simplesmente dormir, sem precisar interagir com a principal.

A única exceção a esta independência é o fato de a principal precisar esperar a execução das **threads** que lançou antes de ser encerrada, e as lançadas devem avisar o seu encerramento para a principal. A seguir os arquivos **sched** da **thread** principal e das outras duas:

```
$ cat /proc/13448/sched
prog3 (13448, #threads: 3)

-----
nr_switches : 4
nr_voluntary_switches : 2
nr_involuntary_switches : 2
policy : 0
prio : 120
clock-delta : 773
```

```
$ cat /proc/13448/task/13449/sched
prog3 (13449, #threads: 3)

-----
nr_switches : 58
nr_voluntary_switches : 57
nr_involuntary_switches : 1
policy : 0
prio : 120
clock-delta : 923
```

```
$ cat /proc/13448/task/13450/sched
prog3 (13450, #threads: 3)

-----
nr_switches : 59
nr_voluntary_switches : 59
nr_involuntary_switches : 0
policy : 0
prio : 120
clock-delta : 825
```

Estados das Tarefas

State (para obter mais informações sobre os possíveis estados de um processo: **man top** ou **man ps.**)

R (*running*) – executando.

S (*sleeping*) – dormindo.

D (*disk sleep*) – dormindo no disco (*swap*).

T (*stooped*) – parado.

Z (*zombie*) – quando o processo termina, mas não avisa o processo pai.

X (*dead*) – finalizado.



Atividade

Use o arquivo **sched** para verificar o estado de alguns processos no seu Linux. Lembre-se que o arquivo **sched** está armazenado no diretório “**/proc/PID do processo**” que você deseja observar.

Usuários e Grupos

Para cada usuário e grupo de usuários, há um único identificador dentro do sistema.

No Linux, cada processo está associado a pelo menos um usuário e um grupo e esta relação pode ser constatada usando o arquivo **/proc/PID/status**, o qual apresenta o usuário (UID) e o grupo (GID) de dado processo. Além desses, a cada processo estão associados outros identificadores especiais de usuários e grupos. A seguir um exemplo do arquivo **/proc/PID/status** e os significados dos seus campos seguindo a ordem dos campos de cada linha:

```
$ cat /proc/12019/status
Name: bash
State: S (sleeping)
Tgid: 12019
Pid: 12019
PPid: 6621
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
...
...
```

UID e GID – identificadores de usuário (UID) e grupo (GID), são de quatro tipos, segundo a ordem das colunas do arquivo **/proc/PID/status**:

- **UID/GID** – usuário/grupo real do processo.
- **EUID/EGID** – usuário/grupo efetivo (este usuário é usado em algumas operações especiais como trocas de senha) do processo.
- **SUID/SIGID** – usuário/grupo que iniciou o processo.
- **FSUID/FSGID** – usuário/grupo para permissões do sistema de arquivos.

Atividade

Use o arquivo **status** para descobrir os usuários e grupos de alguns processos do **/proc**. Lembre-se que o arquivo **status** está armazenado no diretório “**/proc/PID** do processo” que você deseja observar.



Entradas e Saídas Padrão de um Processo

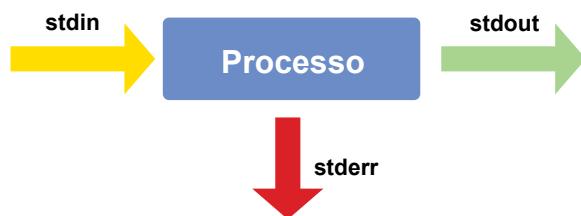
120

Gerência de Processos

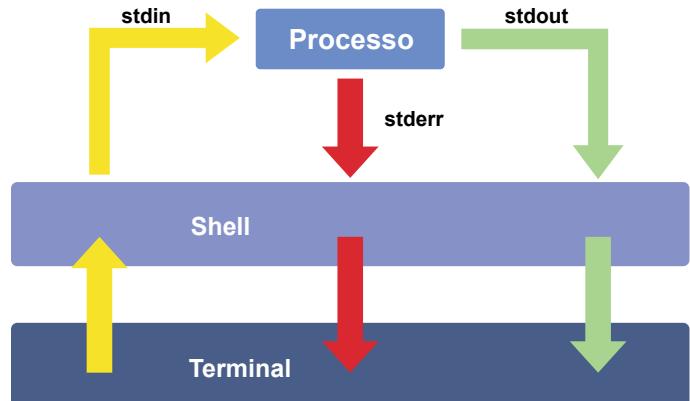
A maioria dos comandos pode comunicar-se com o sistema por meio de descritores de arquivos especiais, conhecidos como entradas e saídas padrão. São eles:

- Entrada padrão (**stdin** – *standard input*): onde o comando vai ler os dados de entrada.
- Saída padrão (**stdout** – *standard output*): onde o comando vai escrever os dados de saída.
- Saída de erro (**stderr** – *standard error*): onde o comando vai enviar mensagens de erro.

A figura a seguir mostra a relação entre um processo e os arquivos padrão.



Quando um comando é lançado sem indicar seu arquivo de trabalho, ele busca seus dados da entrada padrão. Por *default*, o shell onde o comando foi lançado associa o processo ao seu terminal, ou seja: a entrada padrão do processo é associada ao teclado e as saídas padrão e de erros à tela da sessão corrente.



Vejamos um exemplo de uso da entrada e da saída padrão com o comando **rev**, o qual escreve em sua saída padrão as linhas de texto lidas em sua entrada padrão, invertendo-as:

```
$ rev  
Teste de entrada e saida padrao  
oardap adias e adartne ed etseT  
mussum  
mussum
```

Para indicar o fim da entrada padrão (fim de arquivo), basta pressionar <CTRL> D. Com esse caractere o comando **rev** encerra sua execução, pois chegou ao final do arquivo de entrada (que neste caso é o teclado). Vejamos outro exemplo com o comando **sort**:

```
$ sort  
Marcos  
Jose  
Diogo  
Abreu  
Maria <---- pressionado CTRL D  
Abreu  
Diogo  
Jose  
Marcos  
Maria
```

Normalmente o shell direciona a entrada padrão para o teclado e a saída padrão para a tela da sessão do usuário, mas pode ser instruído a redirecioná-las para arquivos ou mesmo para outros programas, como veremos na sequência.

Redirecionamento para Arquivos

O shell pode redirecionar as entradas e as saídas padrão, de comandos, para arquivos normais, usando operadores de redireção. Os principais operadores de redireção para arquivos são:

- **Saída em arquivo:** a saída padrão (**stdout**) do comando é desviada para um arquivo usando o operador **>**.

Lista os arquivos atuais.

```
$ ls  
arquivo.txt diogo.c instala.txt leiamme.txt marcos.c
```

Redireciona a saída do comando ls para o arquivo redireciona.txt.

```
$ ls > redireciona.txt
```

Verifica o conteúdo do novo arquivo criado.

```
$ cat redireciona.txt  
arquivo.txt  
diogo.c  
instala.txt  
leiamme.txt  
marcos.c  
redireciona.txt
```

122

- **Entrada de arquivo:** a entrada padrão (**stdin**) pode ser obtida a partir de um arquivo usando o operador **<**.

Redireciona a entrada padrão do comando rev, isto é, em vez de ler do teclado, estamos lendo o conteúdo do arquivo.

```
$ rev < redireciona.txt  
txt.oviuqra  
c.ogoid  
txt.alatsni  
txt.emaiel  
c.socram  
txt.anoicerider
```

- **Uso combinado:** os dois operadores podem ser usados simultaneamente.

Redireciona a entrada padrão do comando rev, isto é, em vez de ler do teclado, estamos lendo o conteúdo do arquivo e redireciona o conteúdo para o arquivo rev.txt.

```
$ rev < redireciona.txt > rev.txt
```

Verifica o conteúdo do novo arquivo.

```
$ cat rev.txt  
txt.oviuqra  
c.ogoid  
txt.alatsni  
txt.emaiel  
c.socram  
txt.anoicerider
```

- **Concatenação:** a saída padrão pode ser concatenada a um arquivo existente, usando-se o operador **>>**.

Redireciona a saída do comando ls anexo ao final do arquivo já existente.

```
$ ls >> redireciona.txt
```

Verifica o conteúdo do arquivo.

```
$ cat redireciona.txt  
arquivo.txt  
diogo.c  
instala.txt  
leiame.txt  
marcos.c  
redireciona.txt  
arquivo.txt  
diogo.c  
instala.txt  
leiame.txt  
marcos.c  
redireciona.txt  
rev.txt
```

- **Saída de erros:** a saída de erros (**stderr**) pode ser redirecionada juntamente com a saída *standard*. Para isso basta usar o modificador **&1** em conjunção com **2>** ou **2>>**. Vejamos um exemplo:

Executa o comando ls, forçando que seja listado um arquivo que não existe.

```
$ ls * tata  
ls: impossível acessar tata: Arquivo ou diretório não encontrado  
arquivo.txt  instala.txt  marcos.c      rev.txt  
diogo.c     leiamme.txt  redireciona.txt
```

Redireciona a saída do comando para o arquivo, só o erro aparece na tela.

```
$ ls * tata > saida_comando.txt  
ls: impossível acessar tata: Arquivo ou diretório não encontrado
```

Redireciona a saída de erro para o arquivo, só o resultado do comando aparece na tela.

```
$ ls * tata 2> saida_erro.txt  
arquivo.txt  instala.txt  marcos.c      rev.txt  
diogo.c     leiamme.txt  redireciona.txt  saida_comando.txt
```

Redireciona a saída do comando para um arquivo e de erro para outro arquivo.

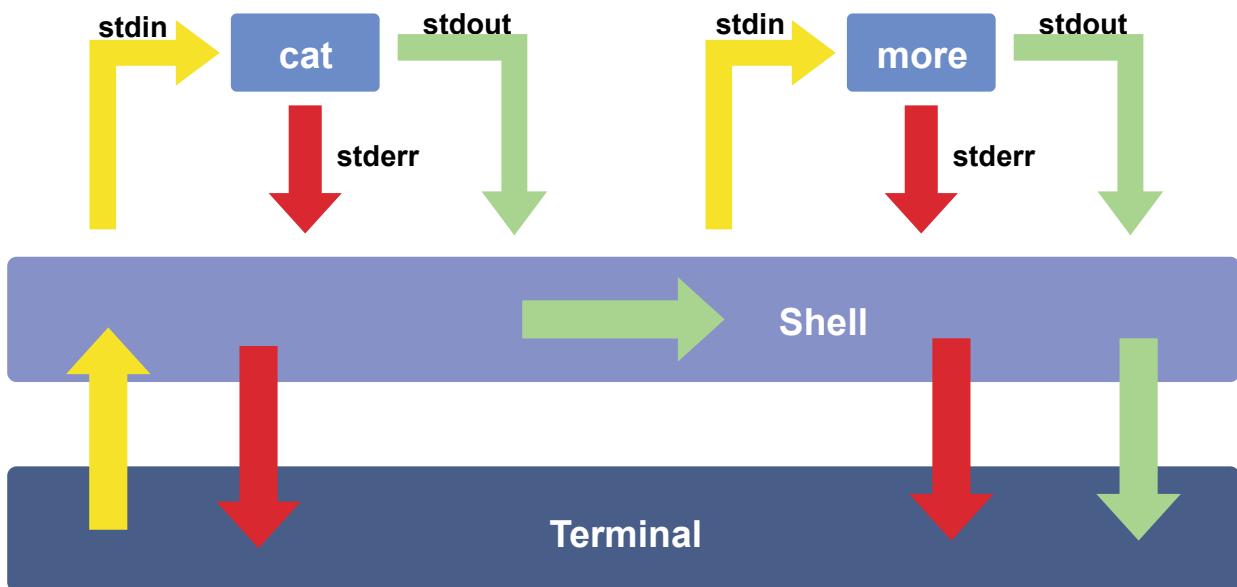
```
$ ls * tata > saida_comando.txt 2> saida_erro.txt
```

Redireciona a saída do comando e saída de erro para o mesmo arquivo.

```
$ ls * tata > saida.txt 2>&1  
$ cat saida.txt  
ls: impossivel acessar tata: Arquivo ou diretorio nao encontrado.  
arquivo.txt  
diogo.c  
instala.txt  
leiamme.txt  
marcos.c  
redireciona.txt  
rev.txt  
saida_comando.txt  
saida_erro.txt
```

Redirecionamento Usando Pipes (Comunicação entre Processos)

O shell permite a construção de comandos complexos por meio da combinação de vários comandos simples. O operador `|`, conhecido como **pipe**, ou tubo, permite conectar à saída *standard* de um comando à entrada *standard* de outro. Com isso, um mesmo fluxo de dados pode ser tratado por diversos comandos consecutivamente, como mostra a figura abaixo.



É importante ressaltar que os comandos conectados são lançados simultaneamente pelo shell e são executados ao mesmo tempo. O shell controla a execução de cada um para que não haja acúmulo de dados entre os comandos (a cada **pipe** é associado um buffer de tamanho limitado).

Vejamos alguns exemplos:

```
Lista todo o conteúdo do diretório /etc de forma paginada.
```

```
$ ls /etc | more
```

```
Visualiza o conteúdo do arquivo /etc/passwd, separando o primeiro campo do arquivo e ordenando.
```

```
cat /etc/passwd | cut -d: -f1 | sort
```

Atividade

Execute os passos a seguir e explique o que acontece. Observe o tamanho do arquivo **listagem.txt**:



```
$ mkdir novo  
$ cd novo  
$ touch a b c  
$ ls -l  
total 0  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 a  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 b  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 c  
$ ls -l > listagem.txt  
$ cat listagem.txt  
total 0  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 a  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 b  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 c  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 listagem.txt  
$ ls -l  
total 4  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 a  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 b  
-rw-r--r-- 1 marcos marcos 0 2009-12-11 16:48 c  
-rw-r--r-- 1 marcos marcos 211 2009-12-11 16:48 listagem.txt
```

126

Observe que o arquivo **listagem.txt** não existia, mas foi relacionado (com tamanho 0). Isto significa que antes do comando **ls -l** ser executado, o arquivo **listagem.txt** foi criado.

Comandos de Filtros

Há um grande número de comandos bastante simples, cujo uso direto é pouco útil, mas que podem ser de grande valia quando associados entre si por meio de **pipes**. Esses comandos são chamados filtros, porque funcionam como filtros para o fluxo de dados. Vejamos os filtros de uso mais corrente (alguns vocês já conhecem):

- cat** – concatena diversos arquivos na saída padrão.
 - tac** – igual ao **cat**, mas inverte a ordem das linhas.
 - more** – permite a paginação do fluxo de dados.
 - tr** – troca de caracteres entre dois conjuntos.
 - head** – seleciona as **n** linhas iniciais do fluxo de dados.
 - tail** – seleciona as **n** linhas finais do fluxo de dados.
 - wc** – conta o número de linhas, palavras e bytes do fluxo.
 - sort** – ordena as linhas, segundo critérios ajustáveis.
 - uniq** – remove linhas repetidas, deixando uma só linha.
 - sed** – para operações complexas de strings (trocas, etc.).
 - grep** – seleciona linhas contendo uma determinada expressão.
 - cut** – seleciona colunas do fluxo de entrada.
 - rev** – reverte a ordem dos caracteres de cada linha do fluxo de entrada.
 - tee** – duplica o fluxo de entrada (para um arquivo e para a saída *standard*).
- ...e muitos outros (são mais de 300).

Prioridades

O Linux implementa o conceito de prioridade, ou seja, alguns processos prontos para execução da CPU podem ceder a sua vez para outros processos mais importantes. Imagine uma fila de banco na qual as pessoas idosas têm prioridade no atendimento pelo caixa bancário. É a mesma ideia.

Mudando Prioridades de Processos – Comandos **nice** e **renice**

O comando **nice** configura a prioridade da execução de um comando/programa. Sua sintaxe: **nice [opções] [comando/programa]**

O comando **renice** configura a prioridade de um processo que já esteja em execução (somente o dono do processo ou o superusuário podem mudar a prioridade de um processo). Sua sintaxe: **renice [opções] [processos ou usuários]**

Se um programa for executado com maior prioridade, ele usará mais recursos do sistema para seu processamento; caso tenha uma prioridade baixa, ele permitirá que outros programas tenham preferência. A prioridade de execução de um programa/comando ou processo pode ser ajustada de -19 (a mais alta) até 19 (a mais baixa). Somente o superusuário pode aumentar a prioridade de um programa ou processo.

Executa o comando `find` com prioridade 10.

```
$ nice -10 find / -name apropos 2>/dev/null &  
[1] 3928
```

Lista os processos em execução.

```
$ ps -l  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 1000 3134 3132 0 80 0 - 4952 wait pts/0 00:00:00 bash  
0 R 1000 3928 3134 8 90 10 - 3113 - pts/0 00:00:00 find  
0 R 1000 3929 3134 0 80 0 - 1684 - pts/0 00:00:00 ps
```

Muda a prioridade do processo em execução.

```
$ renice 19 -p 3928  
3928: prioridade antiga = 10; prioridade nova = 19
```

Lista os processos em execução.

```
$ ps -l  
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD  
0 S 1000 3134 3132 0 80 0 - 4952 wait pts/0 00:00:00 bash  
0 D 1000 3928 3134 9 99 19 - 3138 sync_b pts/0 00:00:02 find  
0 R 1000 3931 3134 0 80 0 - 1684 - pts/0 00:00:00 ps
```

Escalonamento das Tarefas no Linux

O Linux implementa duas políticas de escalonamento, a política FCFS (*First-Come First Served* – Primeira a Chegar Primeira a ser Servida), na qual a tarefa que assume o processador é a primeira que o solicita. No Linux, tarefas que usam a política FCFS não sofrem **preempção** por tempo, nem são analisadas suas prioridades, apenas a ordem de chegada.

Preempção:

Direito de preferência.

A outra política implementada no Linux é a política *Round-Robin*. Ela usa preempção por tempo e baseia-se na prioridade de cada tarefa para escolher qual delas irá assumir o processador. Além das políticas de escalonamento, o Linux divide as tarefas do sistema em classes de escalonamento.

O Linux possui três classes de escalonamento de tarefas, duas classes para tarefas que exijam políticas de escalonamento de tempo real, sendo que uma delas usa politicas FCFS (SCHED_FIFO) e a outra políticas *Round-Robin* (SCHED_RR), e uma terceira classe de escalonamento para as tarefas que não precisam de políticas de tempo real (SCHED_OTHER).

A principal diferença entre as classes de escalonamento de tempo real é que a classe SCHED_FIFO não possui preempção por tempo nem analisa a prioridade das tarefas que serão executadas, enquanto que a classe SCHED_RR possui preempção por tempo e usa a prioridade de cada tarefa para escolher qual tarefa irá assumir o processador.

A classe SCHED_OTHER também usa prioridades na escolha da tarefa que irá assumir o processador e usa preempção por tempo para dividir o processador entre as tarefas do sistema. Vale ressaltar que tarefas de tempo real só podem ser lançadas pelo administrador do sistema e os outros usuários só podem lançar tarefas da classe SCHED_OTHER.

Tarefas que são escalonadas por políticas *Round-Robin* usam a prioridade de cada tarefa para selecionar qual tarefa irá assumir o processador. As prioridades podem variar de 0 (mais prioritário) até 140 (menos prioritário), sendo que as tarefas com prioridades de 0 até 99 são tarefas de tempo real e as tarefas com prioridades entre 100 e 140 são tarefas sem políticas de tempo real.

Apesar de o núcleo representar as prioridades no intervalo de 0 até 140, fora do núcleo, como no comando **nice**, estes valores são apresentados de outra forma. Para tarefas que não usam políticas de tempo real o valor das prioridades varia de -20 até 19, ou seja, para descobrir a representação da prioridade de uma tarefa fora do núcleo deve-se subtrair 120 do valor representado no núcleo.

A seguir um exemplo do arquivo **/proc/PID/sched**, o qual apresenta os dados **policy** (política de escalonamento) e **prio** (prioridade do processo):

```
$ cat /proc/12068/sched
prog2 (12068, #threads: 1)

-----
nr_switches : 5
nr_voluntary_switches : 5
nr_involuntary_switches : 0
policy : 0
prio : 120
clock-delta : 171
```

Execução de Comandos em *Background* e *Foreground*

- **Caractere &** – É possível que se queira executar um comando demorado e, enquanto se espera o resultado dele, fazer outras coisas no terminal. Este processo de se rodar um programa ou comando desvinculado do terminal chama-se rodar em ***background***. Se for colocado o caractere **&** na linha de comando, a mesma será executada em ***background***, liberando assim o terminal para outras tarefas. Se por acaso as saídas padrão e de erro do comando submetido em ***background*** não forem redirecionadas, elas continuarão saindo na tela, apesar da execução do comando não estar mais vinculada a ele. Portanto sempre que for usar comandos em ***background***, redirecione tanto a saída padrão como a saída de erro. O caractere **&** deve ser sempre o último na linha de comando.
- **jobs** – O comando interno **jobs** mostra a situação de todos os processos que estão em ***background*** e que foram submetidos debaixo da sessão corrente. Um processo pode estar em situação suspensa ou rodando. Os processos estão numerados, conforme o número indicado na submissão em ***background***. Estas informações estão armazenadas em uma tabela interna do processo, que é perdida quando se termina a sessão e o processo.

Considere o programa a seguir para testar os comandos.

```
#include <stdio.h>

int main(void)
{
    int i;
    for( i=1;i<100;i++)
    {
        sleep(1);
    }
}
```

130

Compila o programa teste.c.
\$ gcc teste.c -oteste

Executa em background.
\$./teste &
[1] 4101

Lista os processos.
\$ jobs
[1]+ Executando . /teste &
\$

- **Suspendendo processos** – Pode-se suspender a execução de um processo, rodando em ***foreground*** por meio da digitação do caractere <CTRL> + <Z>. Ao se digitar o caractere, o processo é interrompido e fica esperando um comando para que volte a rodar. Este processo fica no sistema ocupando os recursos, mas não utiliza a CPU da máquina. O processo suspenso pode ser cancelado, pode voltar a executar em ***foreground*** ou pode colocar o mesmo para rodar em ***background***.

Executa o programa

```
$ ./teste
^Z      ----- pressionado CTRL Z
[1]+  Parado          ./teste
```

Verificando o processo com o comando jobs

```
$ jobs
[1]+  Parado          ./teste
```

131

Gerência de Processos

- **bg** – Para ativar um processo parado, podemos usar os comandos internos **bg** ou **fg**, bastando fornecer o número do processo indicado pelo comando **jobs** ou alguma string de caracteres que identifique o processo. Com o comando interno **bg** indica-se que o processo deve voltar a executar em ***background***.
- **fg** – Podemos também chamar um processo que está suspenso ou em ***background*** para que o mesmo volte a rodar em ***foreground***. Usamos para isto o comando interno **fg**, fornecendo o número, ou uma **string**, obtido pelo comando interno **jobs**.

Verifica o estado do processo.

```
$ jobs
[1]+  Parado          ./teste
```

Coloca em execução em ***background***.

```
$ bg %1
[1]+  ./teste &
```

Verifica o estado.

```
$ jobs  
[1]+ Executando      ./teste &
```

Traz o processo que está rodando em background para foreground (prende o terminal).

```
$ fg %1  
./teste  
  
^Z <----- pressionado CTRL Z  
[1]+ Parado          ./teste
```

Verifica o estado.

```
$ jobs  
[1]+ Parado          ./teste
```

O Ciclo de Vida de um Processo no Linux

Ao ser criado, o processo faz uma cópia de si mesmo por meio da chamada de sistema **fork**. Uma chamada **fork** cria uma cópia do processo original que é em grande parte idêntica ao pai. O novo processo possui um PID distinto e possui suas próprias informações contábeis.

A chamada de sistema **fork** possui a propriedade exclusiva de retornar dois valores diferentes. Sob o ponto de vista do filho, ele retorna zero. Para o pai, por outro lado, é retornado o PID do filho recém-criado. Já que os dois processos são, sob outros aspectos, idênticos, ambos precisam examinar o valor retornado para descobrir que papel eles devem supostamente assumir.

Após uma **fork**, o processo-filho normalmente usará uma das chamadas de sistema da família **exec** para começar a execução de um novo programa. Essas chamadas mudam o texto do programa que o processo está executando e reinicializam os segmentos de dados e de pilha para um estado inicial predefinido. As várias formas de **exec** diferem somente na maneira pela qual elas especificam o ambiente e os argumentos de linha de comando a serem dados ao novo programa.

Quando o sistema é inicializado, o *kernel* cria e instala de maneira autônoma vários processos. O mais notável desses é o processo **init**, que sempre é o processo número 1. O processo **init** é o responsável pela execução da maioria dos scripts de inicialização do sistema. Todos os processos, exceto aqueles que o *kernel* cria, são descendentes (filhos, netos, bisnetos, etc.) do processo **init**.

O processo **init** também desempenha um importante papel no gerenciamento de processos. Quando um processo é completado, ele chama uma rotina denominada **_exit** para notificar o *kernel* de que está pronto para expirar. Ele fornece um código de saída (um inteiro) que informa o porquê de ele estar saindo. Por convenção, é usado para indicar um término normal ou “bem-sucedido”.

Antes de um processo ter a permissão para desaparecer completamente, o Linux requer que sua expiração seja reconhecida pelo processo-pai, o que o pai faz por meio de uma chamada a **wait**. O pai recebe uma cópia do código de saída do filho (ou uma indicação do porquê o filho foi extinto, caso este não tenha abandonado voluntariamente) e, também, pode obter uma síntese do uso de recursos do filho rodando.

O programa a seguir demonstra como isto ocorre:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void)
{
    pid_t iPid;
    int iStatus;

    if( (iPid = fork())<0) /* cria um processo filho */
    {
        perror("Erro no fork");
        return 0;
    }

    if( iPid != 0) /* no processo pai*/
    {
        printf("\nCriado o processo %d", iPid);
        while(1)
```

```
{  
    printf("\nEsperando o status do filho.");  
    wait(&iStatus);  
    printf("\nStatus do filho pego.");  
  
    if( WIFEXITED(iStatus) )  
    {  
        printf("\nFilho terminou normalmente");  
        printf("\nO status de término foi %d.", WEXITSTATUS(iStatus));  
        break;  
    }  
    if( WIFSIGNALED(iStatus) )  
    {  
        printf("\nFilho recebeu sinal e terminou");  
        printf("\nO sinal que terminou o filho foi %d.", WTERMSIG(iStatus));  
        break;  
    }  
  
    if( WIFSTOPPED(iStatus) )  
    {  
        printf("\nFilho recebeu sinal stop");  
        printf("\nO sinal que parou o filho foi %d.", WSTOPSIG(iStatus));  
    }  
}  
else /* no processo filho */  
{  
    /* fica em loop esperando um sinal via comando kill ou ate terminar */  
    int i;  
    printf("\nFilho %d em execucao", getpid());  
    for(i=0;i<20;i++)  
    {  
        printf("\nFilho faltam %d segundos para terminar.", 20-i);  
        sleep(1);  
    }  
}  
return 0;  
}
```

Compila e executa o programa.

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

```
$ ./teste
```

```
Criado o processo 3501
```

```
Filho 3501 em execucao
```

```
Filho faltam 20 segundos para terminar.
```

```
Filho faltam 19 segundos para terminar.
```

```
Filho faltam 18 segundos para terminar.
```

```
Filho faltam 17 segundos para terminar.
```

```
Filho faltam 16 segundos para terminar.
```

```
Filho faltam 15 segundos para terminar.
```

```
Filho faltam 14 segundos para terminar.
```

```
Filho faltam 13 segundos para terminar.
```

```
<----- resultado suprimido propositalmente ----->
```

Em outro terminal, vamos verificar os processos criados (repare nos campos PID e PPID de cada processo).

Compila e executa o programa.

```
$ ps -ef | grep teste
```

```
marcos 3500 3238 0 09:49 pts/0 00:00:00 ./teste
```

```
marcos 3501 3500 0 09:49 pts/0 00:00:00 ./teste
```

Esse esquema funciona bem se os pais sobreviverem aos filhos e se estiverem cientes da necessidade de chamarem **wait** para que os processos extintos possam ser eliminados. Entretanto, se o pai for extinto primeiro, o *kernel* reconhecerá que nenhum **wait** virá no futuro e ajustará o processo para que o órfão se torne um filho do processo **init**. O processo **init** aceita os processos-órfãos e executa o **wait** necessário para se livrar deles quando forem extintos.

Comando nohup – Entregando o Processo Criado para o Processo Init

Todos os processos que o usuário roda estão ligados ao processo shell de sessão. O Linux possui a característica de que se o processo principal terminar, todos os seus filhos serão encerrados. Isto torna-se um problema quando temos que rodar um comando ou programa demorado e não podemos manter a sessão aberta. O comando **nohup** serve para desligar um processo do processo shell da sessão, permitindo, assim, que se encerre a sessão sem prejuízo da execução do programa ou comando. Na prática, o comando informa ao sistema operacional que, se a sessão for encerrada (pai do processo), o processo filho será entregue ao processo **init** da máquina (PPID 1).

Deve-se sempre colocar o comando para rodar em **background**, pois o comando **nohup** não faz isso automaticamente.

Considere o programa a seguir para testar o comando **nohup**.

```
#include <stdio.h>

int main(void)
{
    int i;
    for( i=1;i<100;i++)
    {
        sleep(1);
    }
}
```

136

Compila e coloca o programa em execução em background.

```
$ gcc teste.c -o teste
$ nohup ./teste >/dev/null 2>&1 &
[1] 4184
```

Lista os processos, repare no campo PPID do processo teste.

```
$ ps -l
F S     UID      PID  PPID   C PRI  NI ADDR SZ WCHAN  TTY TIME      CMD
0 S    1000    4170  3132   0  80    0 - 4922 wait    pts/1  00:00:00 bash
0 S    1000    4184  4170   0  80    0 -  942 hrtime pts/1  00:00:00 teste
0 R    1000    4187  4170   0  80    0 - 1684 -       pts/1  00:00:00 ps
```

Feche o terminal anterior, abra outro e digite o comando abaixo, repare que o campo PPID agora é 1.

```
$ ps -l -u marcos | grep teste  
0 S 1000 4184 1 0 80 0 - 942 hrtme ? 00:00:00 teste
```

Conceito de Daemon

Em várias situações, é necessário que um processo fique em execução continuamente (eternamente) em uma máquina. A esses processos dá-se o nome de **daemons** ou serviços.

Geralmente, são programas iniciados assim que o sistema operacional é inicializado. Coloca-se a chamada dos **daemons** nos arquivos de configuração para que os mesmos sejam ativados automaticamente durante o processo de *boot* do sistema. Um **daemon** só deve ser cancelado quando o sistema operacional estiver encerrando o seu processamento. **Daemons** são processos que rodam em **background** e não devem ter um terminal associado a eles.

O programa a seguir implementa um **daemon** simples. Este programa será executado automaticamente em **background**, seu PPID (número do identificador do processo pai) será 1 (processo **init**) e ele enviará mensagens para o **daemon syslog** (responsável por receber e armazenar as mensagens de log do sistema operacional).

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <signal.h>  
#include <syslog.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
  
int daemon_init (void)  
{  
    pid_t iPid;  
    long iMaxFd;  
    int i, fd0, fd1, fd2;  
    struct sigaction sa;  
  
    /* 1. passo - duplicar o processo usando o fork */  
    if((iPid = fork()) < 0)
```

```
{  
    perror("fork:");  
    exit(1);  
  
}  
  
if(iPid != 0) /* finaliza o pai */  
{  
    exit(0);  
}  
  
/* 2. passo - chamar a função setsid para criar uma nova sessão  
de processo, ficando o processo filho como líder da sessão e sem um  
terminal de controle associado ao processo */  
setsid();  
  
/* 3. passo - é necessário realizar 2 forks para evitar que o  
shell ligue um terminal ao processo. Também é necessário ignorar o  
sinal de término do pai */  
  
sa.sa_handler = SIG_IGN;  
sigemptyset(&sa.sa_mask);  
sa.sa_flags = 0;  
if(sigaction(SIGHUP, &sa, NULL) < 0)  
{  
    perror("sigaction:");  
    exit(1);  
}  
  
if((iPid = fork()) < 0)  
{  
    perror("fork:");  
    exit(1);  
}  
  
if(iPid != 0) /* finaliza o pai */  
{  
    exit(0);  
}
```

```
/* 4. passo - troca-se o diretório atual para a raiz do sistema
ou para um diretório do próprio daemon */
chdir("/");

/* 5. passo - inicializa a máscara padrão de criação de arquivos */
umask(0);

/* 6. passo - fechando todos os descritores de arquivos existentes no sistema. Utiliza-se a informação de número máximo de descritores configurado no sistema e obtido com a função sysconf. */
iMaxFd = sysconf (_SC_OPEN_MAX);
for (i=0; i < iMaxFd; i++)
{
    close (i);
}

/* 7. passo - abrindo arquivos de entrada padrão, saída padrão e saída de erro para /dev/null */
fd0 = open("/dev/null", O_RDWR);
fd1 = dup(0);
fd2 = dup(0);

/* abrindo a log */
openlog("daemon_init", LOG_CONS, LOG_DAEMON);

if( fd0 != 0 || fd1 != 1 || fd2 != 2 )
{
    syslog(LOG_ERR, "não foi aberto um dos arquivos %d %d %d",
fd0, fd1, fd2);
}

return 0;
}

int main(void)
{
    int i;
    daemon_init(); /* chama o daemon */
}
```

```

for(i=0;i<20;i++)
{
    syslog(LOG_NOTICE, "faltam %d segundos para terminar o
daemon", 20-i);
    sleep(1);
}

return 0;
}

```

Abra um terminal e faça como no quadro a seguir:

Compila e coloca o programa em teste.

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

Executa o daemon.

```
$ ./teste
```

Verifica os dados do processo teste (repare que o número 1 é o número do processo pai (PPID), que sempre é o processo init).

```
$ ps -e -f | grep teste
marcos      3404      1  0 09:40 ?          00:00:00 ./teste
```

140

Abra outro terminal e observe o arquivo de **logs** do sistema (gerado e mantido pelo **daemon syslog**):

Visualiza as alterações no final do arquivo, à medida que o mesmo vai ocorrendo.

```
$ tail -f /var/log/syslog
<----- resultado suprimido propositalmente ----->
Dec 14 09:40:36 laureano daemon_init: faltam 3 segundos para
terminar o daemon

Dec 14 09:40:37 laureano daemon_init: faltam 2 segundos para
terminar o daemon

Dec 14 09:40:38 laureano daemon_init: faltam 1 segundos para
terminar o daemon
```

Sinais no Linux (Outra Forma de Comunicação com Processos)

Sinais são solicitações de interrupção em nível de processo. São definidos cerca de 30 tipos distintos e eles são usados de várias maneiras:

- podem ser enviados entre processos como um meio de comunicação;
- podem ser enviados pelo driver de terminal para extinguir, interromper ou suspender processos quando teclas especiais como <CONTROL C> e <CONTROL Z> forem digitadas;
- podem ser enviados pelo administrador (via comando **kill**) para obter vários resultados;
- podem ser enviados pelo *kernel* quando um processo comete uma infração como uma divisão por zero.

Quando se quer matar um processo, seja em **background** seja em **foreground** (de outro terminal), deve-se usar o comando **kill**. Deve-se fornecer para esse comando o número do PID do processo que se quer eliminar. Somente o proprietário do processo e o usuário *root* podem enviar sinais.

Opcionalmente, podemos estabelecer o tipo de sinal a ser mandado para o processo. Para isto basta colocar o número ou o mnemônico (abreviatura de códigos de operações que representam funções e tarefas desempenhadas pelo processador) do mesmo. Caso não se coloque nenhum sinal, será enviado o sinal SIGTERM para o processo.

Lista todos os sinais disponíveis no sistema.

```
$ kill -1
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT    7) SIGBUS      8) SIGFPE      9) SIGKILL    10) SIGUSR1
 11) SIGSEGV   12) SIGUSR2    13) SIGPIPE    14) SIGALRM   15) SIGTERM
 16) SIGSTKFLT 17) SIGCHLD   18) SIGCONT    19) SIGSTOP   20) SIGTSTP

----- resultado suprimido propositalmente ----->
```

141

Gerência de Processos

Os sinais mais importantes são:

- **SIGTERM** – Interrupção amena do processo. Este sinal pode ser interceptado e ignorado pelo processo.
- **SIGKILL** – Interrupção forçada do processo. Este sinal não pode ser interceptado e nem ignorado pelo processo.
- **SIGSTOP** – Suspende a execução de um processo. Este sinal também não pode ser interceptado e ignorado pelo processo.
- **SIGCONT** – Ativa a execução de um processo suspenso.

Considere o script abaixo para testar o comando **kill**.

```
#!/bin/bash  
trap "echo Não vou terminar!!!" 15 #intercepta o sinal 15  
while true  
do  
    sleep 1  
done
```

Atributo de execução para o script e coloca para executar em background.

```
$ chmod u+x tempo.sh  
$ ./tempo.sh &  
[1] 4248
```

Convida o processo a morrer.

```
$ kill -TERM 4248  
$ Não vou terminar!!!
```

142

Verifica o estado do processo.

```
$ jobs  
[1]+ Executando          ./tempo.sh &
```

Para o processo (retira da fila de execução).

```
$ kill -STOP 4248
```

Verifica o estado do processo.

```
$ jobs  
[1]+ Parado              ./tempo.sh
```

Coloca o processo para executar novamente (coloca na fila de execução).

```
$ kill -CONT 4248
```

Verifica o estado do processo.

```
$ jobs  
[1]+ Executando ./tempo.sh &
```

Mata o processo.

```
$ kill -KILL 4248
```

Comunicação entre Processos com FIFO

O Linux implementa um arquivo especial para a comunicação entre os processos. Este arquivo é especial para o sistema operacional porque ele implementa uma estrutura FIFO (*first-in first-out* – o primeiro a entrar é o primeiro a sair).

Como o arquivo tem um nome e ocupa uma posição dentro do sistema de arquivo, pertencendo a um diretório, os processos podem abri-lo como se fosse um arquivo comum e realizar as operações de entrada/saída para a troca de mensagens entre si.

O arquivo FIFO pode ser criado com o comando **mkfifo**. Para haver a comunicação entre os processos, é necessário que um processo esteja escrevendo no arquivo e outro processo esteja lendo (escutando este arquivo). O próprio sistema operacional sincroniza a comunicação entre os processos.

Cria o arquivo de comunicação.

```
$ mkfifo teste_fifo
```

Verifica os atributos do novo arquivo criado (repare na letra p no início, isto indica que é um arquivo de pipe).

```
$ ls -l teste_fifo  
prw-r--r-- 1 marcos marcos 0 2009-12-14 09:55 teste_fifo
```

Executa processo em background (só coloca uma mensagem no arquivo FIFO). O processo fica “preso” até que algum outro processo leia o arquivo de FIFO.

```
$ echo "Teste da mensagem" > teste_fifo &  
[1] 3541  
$ jobs  
[1]+ Executando echo "Teste da mensagem" > teste_fifo &
```

```
Lê o arquivo FIFO e neste momento o processo que gerou a mensagem é concluído.
```

```
$ cat teste_fifo
```

Teste da mensagem.

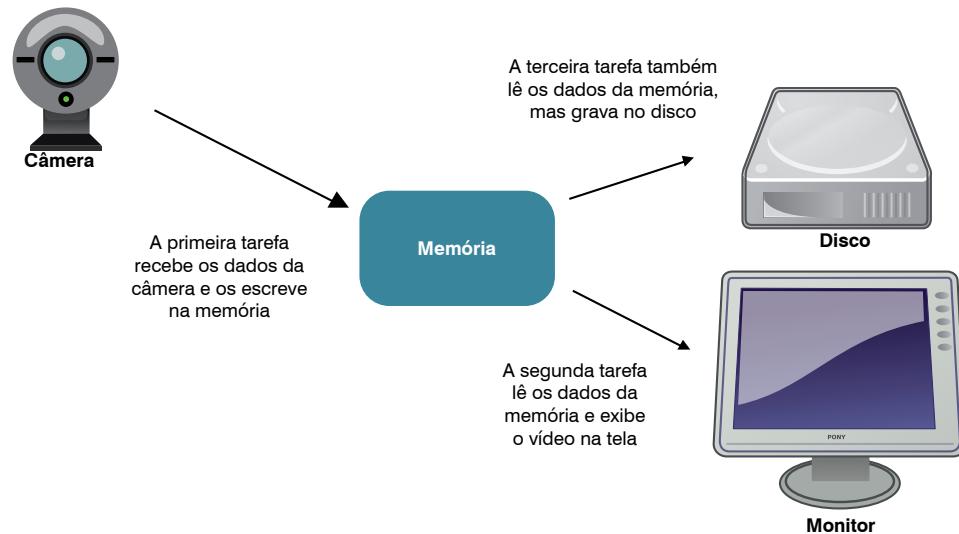
```
[1]+ Concluído
```

```
echo "Teste da mensagem" > teste_fifo
```

```
$ jobs
```

Condições de Disputa

Há algumas situações em que tarefas diferentes precisam cooperar para que seja possível resolver algum problema. Um exemplo seria um *software* para usar uma webcam capaz de receber os dados da câmera, imprimi-los na tela e ainda armazená-los no disco rígido. Fica claro, neste exemplo, que em um único processo há três tarefas distintas acessando uma mesma região de memória, como pode ser visto no desenho a seguir.



144

Há ainda vários outros exemplos em que processos ou tarefas precisam cooperar ou, então, competem por algum recurso. O que aconteceria se dois processos tentassem imprimir coisas distintas, em uma mesma impressora, ao mesmo tempo? Certamente papel e tinta seriam desperdiçados. Casos assim, nos quais tarefas competem por algum recurso comum, são chamados de condições de corrida ou condições de disputa.

Para solucionar estes problemas deve existir algum mecanismo capaz de gerenciar tais condições, evitando, assim, que algum processo atrapalhe a execução de outro processo. A forma mais usada para impedir que um processo/tarefa atrapalhe outro que está competindo por um mesmo recurso é garantir o acesso mutuamente exclusivo, ou seja, garantir que apenas um processo/tarefa acesse o recurso de cada vez.

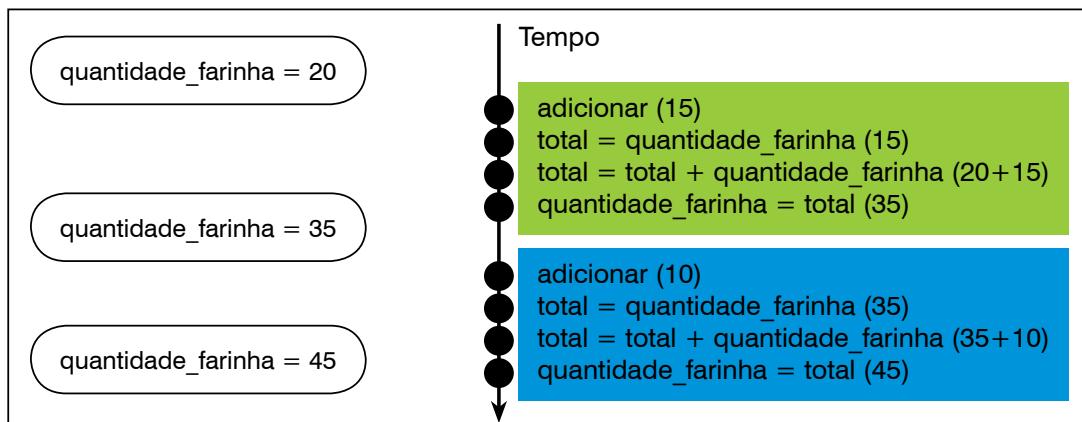
Para garantir o acesso mutuamente exclusivo a um determinado recurso é necessário primeiramente determinar quais são as regiões críticas (ou seção crítica) do código. Uma seção crítica é um conjunto de instruções cuja execução pode resultar na ocorrência de uma condição de disputa, ou seja, a parte do código que acessa o recurso que deve ser compartilhado.

O código a seguir ilustra esse problema. Este código poderia ser usado para controlar o estoque de uma empresa, como uma padaria. No código há uma variável global (`quantidade_farinha`, que representa a quantidade de farinha no estoque), essa variável é, portanto, compartilhada entre todas as funções do programa, além disso há a função `adicionar` que altera o valor da variável global.

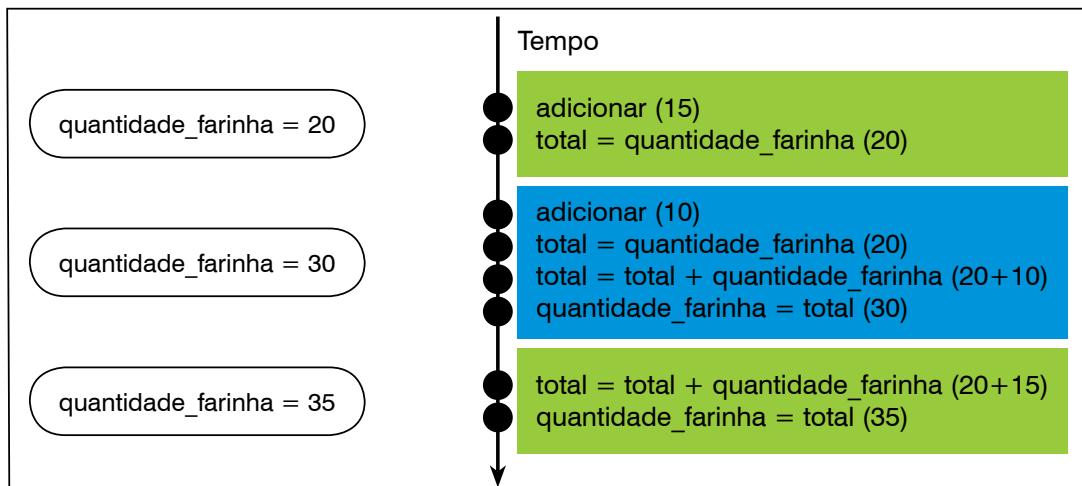
```
float quantidade_farinha

...
float adicionar(float quantidade)
{
    total = quantidade_farinha
    total = total + quantidade
    quantidade_farinha = total
}
```

Caso a função `adicionar` seja executada duas vezes, uma adicionando 15 a variável `quantidade_farinha` e outra adicionando 10, em tempos distintos, nada incomum acontecerá. O valor final desta variável será 45 ($20 + 15 + 10 = 45$) como pode ser visto no diagrama:



Porém, se essas duas execuções ocorrem paralelamente podem acontecer situações como a demonstrada no diagrama:



Como pode ser visto pelo diagrama anterior, o resultado final da execução em série ficou diferente do resultado da execução em paralelo. Isso ocorreu porque na execução das duas funções em paralelo, ambas leram como valor de `quantidade_farinha` 20 e somaram respectivamente 15 e 10 nesta variável, porém quando a função adicionar (15) escreveu seu resultado na variável ela apagou a soma que a primeira função havia feito.

Situações assim podem ocorrer várias vezes, quando há dois ou mais processadores na máquina, ou quando um fluxo de execução é interrompido por algum motivo, como o término de um *quantum*, e outro processo assume o processador. Nestes casos, é importante que existam mecanismos capazes de impedir erros.

Se ocorrerem condições de disputa dentro do núcleo, podem ocorrer falhas graves que possibilitam, até mesmo, travar o sistema operacional. Portanto, cabe ao sistema operacional prover mecanismos para evitar a ocorrência de problemas causados pela ordem de execução de instruções, ou pelo instante em que um certo conjunto de instruções é executado.

Semáforos

O mecanismo mais usado para sincronizar condições de disputa são os semáforos, os quais, como o próprio nome sugere, controlam a execução de determinados pontos do código. Estes são as regiões críticas, pois se forem executados de forma descontrolada poderão causar erros no sistema.

146

Voce Sabia?

Edsger Wybe Dijkstra (1930-2002) foi um cientista da computação que nasceu em Roterdã, nos Países Baixos, em 1930 e deixou significativas contribuições para a computação, dentre elas os semáforos e um algoritmo para encontrar o caminho mínimo em grafos (diagramas compostos de pontos usados para representar conjuntos), também conhecido como “Algoritmo de Dijkstra”.

Semáforos são estruturas de dados sobre as quais é possível executar basicamente duas operações: P(s) e V(s). A operação P(s) ocupa um semáforo, ou seja, tenta executar uma região crítica, se o semáforo estiver livre a região crítica é executada, porém se o semáforo estiver ocupado isto significa que outra região crítica está executando e um determinado recurso está ocupado. Neste caso, o processo aguarda até que seja sua vez de executar.

A operação V(s), por sua vez, libera o semáforo para que outras tarefas possam executar normalmente. As operações P(s) e V(s) são normalmente usadas no inicio (P) e fim (V) de uma região crítica. Para demonstrar o uso de semáforos será usado o código do exemplo anterior.

```

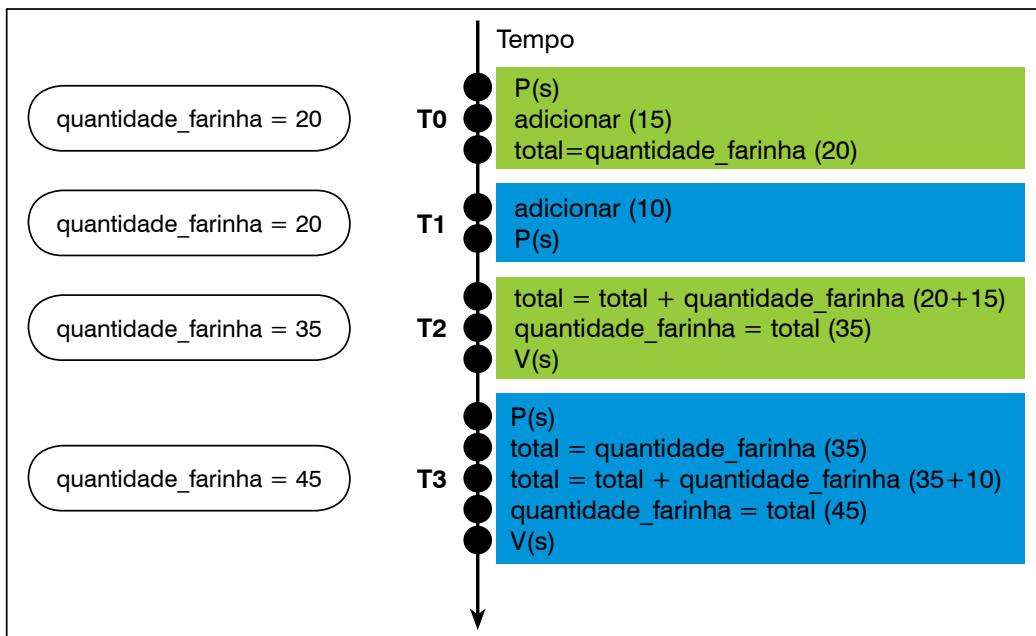
float quantidade_farinha

. . .

float adicionar(float quantidade)
{
    P(s)
    total = quantidade_farinha
    total = total + quantidade
    quantidade_farinha = total
    V(s)
}

```

Usado desta forma, o semáforo controlaria o acesso a variável `quantidade_farinha` evitando o erro que ocorreu quando os códigos foram executados em paralelo. A execução de duas funções `adicionar` em paralelo (em uma máquina com dois processadores, por exemplo) usando semáforos ficaria como demonstrado no diagrama:

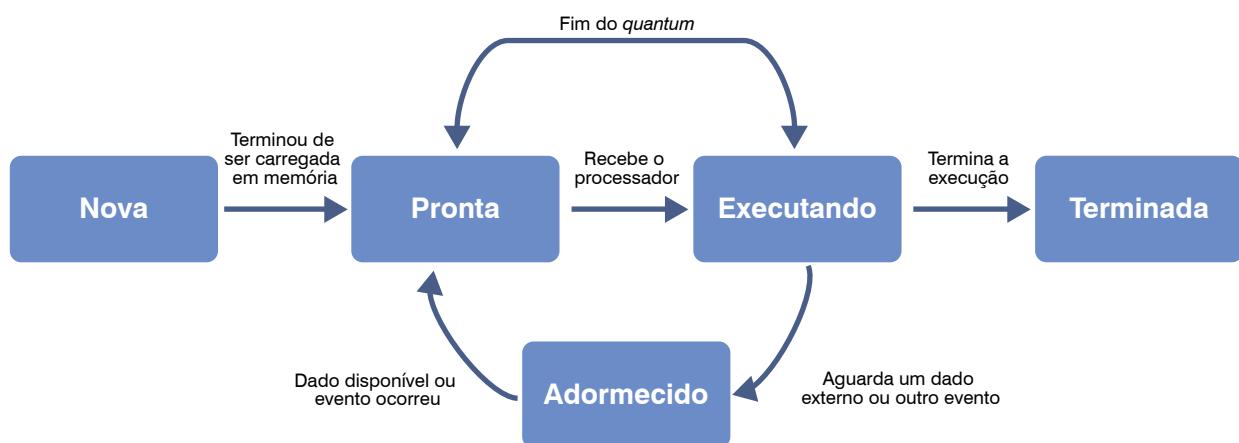


A execução do código anterior poderia ser dividido em quatro instantes:

- Em T0 a função `adicionar (15)` começa e executa `P(s)`, ou seja, bloqueia o acesso a região crítica para que outras funções não possam acessar a variável `quantidade_farinha`, a qual neste instante vale 20.
- No instante T1 a função `adicionar (10)` inicia em outro processador e tenta executar a operação `P(s)`. Neste instante a função `adicionar (10)` fica bloqueada, ou seja, não consegue acessar a região crítica, pois a execução de `adicionar (15)` havia bloqueado o acesso a esta região crítica. Repare que o valor da variável `quantidade_farinha` não foi alterado.

- Em T2 a execução de adicionar (10) termina e executa a operação V(s) liberando assim o acesso exclusivo à região crítica. A partir deste instante a função adicionar (15) pode executar. O valor de quantidade_farinha no final de T2 é de 35.
- Em T3, após a liberação da região crítica, a função adicionar (10) pode retomar sua execução normal. Neste instante, a variável quantidade_farinha vale 35 e a função adicionar (10) termina com a variável quantidade_farinha valendo 45. Antes de terminar, a função libera a região crítica para que outras funções possam executá-la.

Mas como uma tarefa fica bloqueada ao tentar executar uma região crítica que não está livre? Vamos relembrar o ciclo de vida das tarefas.



148

Ao tentar executar uma região crítica que está bloqueada o sistema operacional retira a tarefa do estado executando e a coloca na fila dos processos suspensos. A tarefa permanece nesta fila até que o semáforo seja liberado. Quando a região crítica é liberada a tarefa volta para a fila de processos prontos e continua seu ciclo de vida normal.

Mutex

Quando semáforos são usados para garantir exclusão mútua, eles também podem ser chamados de mutex, porém semáforos também podem ser usados para garantir que apenas um certo número de regiões críticas sejam executadas de uma vez. Por exemplo: um semáforo pode ser usado para garantir que apenas três regiões críticas sejam executadas em um certo momento.

Este uso de semáforos não garante a exclusão mútua, porém possui outras aplicações, sendo muito usado em simulações. A maioria dos semáforos utilizados em sistemas operacionais são usados para garantir a exclusão mútua, ou seja, a maioria dos semáforos são mutex.

Programando o mutex

A API POSIX oferece uma biblioteca que contém semáforo e mutex que podem ser usados nos programas que criamos. O uso dos semáforos POSIX é bastante simples, a primeira coisa que deve ser feita é a inclusão da biblioteca **pthread** no código. Isso pode ser feito da seguinte forma:

- `#include <pthread.h>`

O próximo passo é a criação e inicialização de um mutex. Um mutex POSIX é uma variável do tipo **pthread_mutex_t**, esta pode ser inicializada com uma macro definida na biblioteca **pthread** que se chama **PTHREAD_MUTEX_INITIALIZER**. Em outras palavras, um **mutex** pode ser criado e inicializado com a seguinte linha:

- `pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER`

No exemplo anterior foi criado e inicializado um **mutex** chamado **mtx**. Depois que o **mutex** foi inicializado podem ser feitas as operações de P(s) e V(s) usando as seguintes funções, respectivamente:

- `pthread_mutex_lock(&mtx);`
- `pthread_mutex_unlock(&mtx).`

O código a seguir apresenta um exemplo de como os **mutex** POSIX podem ser usados. Para este exemplo foram criadas quatro **threads** que incrementam uma variável chamada `quantidade_farinha`, como no exemplo anterior.

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

void* adicionar();

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
float quantidade_farinha;

int main()
{
    int rc1, rc2, rc3, rc4;
    pthread_t thread1, thread2, thread3, thread4;

    // Cria quatro threads.

    if( (rc1=pthread_create( &thread1, NULL, &adicionar, NULL)) )
    {
```

```
    printf("Thread creation failed: %d\n", rc1);

}

if( (rc2(pthread_create( &thread2, NULL, &adicionar, NULL)) ) )
{
    printf("Thread creation failed: %d\n", rc2);
}

if( (rc3(pthread_create( &thread3, NULL, &adicionar, NULL)) ) )
{
    printf("Thread creation failed: %d\n", rc3);
}

if( (rc4(pthread_create( &thread4, NULL, &adicionar, NULL)) ) )
{
    printf("Thread creation failed: %d\n", rc4);
}

// Espera as execuções das threads para terminar o programa.

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);
pthread_join( thread3, NULL);
pthread_join( thread4, NULL);

exit(0);
}

void* adicionar()
{
    int x;
    float total;
    x = 0;
```

```

pthread_mutex_lock( &mutex1 );

for (x = 0; x < 100; x++)

{
    total = quantidade_farinha;

    sleep (0,1);

    total = total + 1;

    sleep (0,1);

    quantidade_farinha = total;

    sleep (0,1);

}

printf("Quantidade de farinha: %f\n", quantidade_farinha);

pthread_mutex_unlock( &mutex1 );

}

```



Atividade

Com a ajuda do professor compile e execute o código anterior para verificar o funcionamento dos **mutex**. Depois comente as linhas que realizam as operações P(s) e V(s) para verificar se ocorrem erros no código, caso não sejam usados semáforos.

151

Vantagens e desvantagens

O uso de semáforos implica em bloquear algumas tarefas enquanto outras acessam algum recurso que é compartilhado. A grande vantagem dos semáforos é que eles garantem que não ocorrerão erros de consistência causados pela concorrência de tarefas por recursos, como os mostrados nos códigos acima.

Porém os semáforos possuem algumas desvantagens, uma delas é que como algumas tarefas ficam bloqueadas esperando a liberação de algum recurso o desempenho dos programas que usam os semáforos pode diminuir. Isto ocorre, principalmente, se os semáforos não forem usados de forma correta, mas, sim, de forma desnecessária.

Mas o principal problema do uso de semáforos é que eles podem causar impasses (ou *Deadlock*). Um impasse é uma situação em que duas ou mais tarefas ficam bloqueadas esperando a liberação de algum semáforo e nada mais acontece, ou seja, todas as tarefas ficam bloqueadas.

Um exemplo de impasse no mundo real seria a situação em que um vendedor só entrega dada mercadoria após o pagamento e o cliente só paga após a entrega da mercadoria. Neste caso, a venda nunca será realizada. A mesma situação pode ocorrer com duas ou mais tarefas que bloqueiam recursos.

Há algumas técnicas que podem evitar ou desfazer situações de impasse, porém possuem um custo computacional muito alto, por isso a maioria dos sistemas operacionais não as aplica e assume o risco de que impasses ocorram.

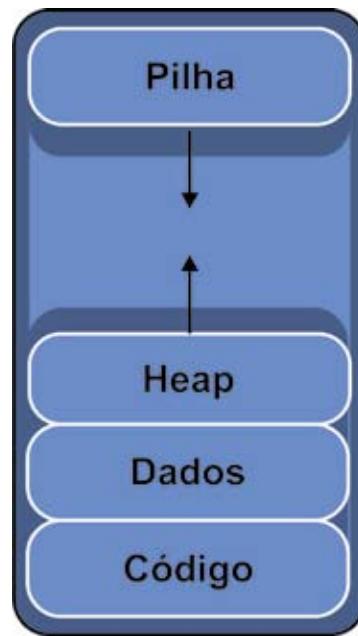
Memória de um Processo

Para o sistema operacional, um processo é visto como uma região de memória isolada dos outros processos, a qual apenas o sistema operacional e o próprio processo podem acessar. Nesta região de memória estão todos os dados do processo, como suas instruções para o processador, suas variáveis internas, seus buffers, etc.

Como há dados de diversas naturezas, a memória de um processo é dividida em quatro segmentos bem definidos, de forma que cada região de memória possui características próprias. Os quatro segmentos de memória são:

- **Código** ou *Text*: Neste segmento de memória estão as instruções que o processo usará e o processador executará. Seu tamanho e conteúdo são fixos, ou seja, não são alterados durante a execução e são gerados durante o processo de compilação. Este segmento possui permissão para ser executado e lido, não podendo ser alterado.
- **Dados** ou *Data*: Segmento de memória que armazena as variáveis globais e estáticas de um processo, também possui um tamanho fixo visto que as variáveis desta região são definidas durante o processo de compilação. Este segmento pode ser lido e alterado, porém não pode ser executado como o segmento de código.
- **Pilha** ou *stack*: Neste segmento estão as regiões de memória locadas durante a execução do processo. Na linguagem C, estas regiões são alocadas com a função **malloc** e liberadas com a função **free**. Como esse segmento possui dados gerados durante a execução, seu tamanho varia durante esta, a execução e o segmento possui permissões de leitura e escrita, porém não possui permissão de execução.
- **Heap**: Este segmento de memória armazena a pilha de execução do processo, ou seja, armazena informações referentes às chamadas de funções, aos valores de retorno destas, às variáveis locais de cada função e aos parâmetros das chamadas de funções. Como no segmento de pilha seu conteúdo pode ser lido e alterado, porém não pode ser executado e seu tamanho é alterado durante a execução do processo.

Estes quatro segmentos de memória são organizados como na figura a seguir:



Os segmentos de pilha e **heap** crescem para uma região em comum, isso ocorre para garantir uma melhor utilização da memória disponível ao processo, porém os segmentos podem acabar se chocando, de forma que um deles acaba escrevendo dados no outro. Esta situação é conhecida como *Stack Overflow* e nos sistemas que não a verificam um processo pode assumir um comportamento inesperado.

Informações sobre Uso de Memória pelos Processos no /proc

Além do arquivo **/proc/meminfo**, que apresenta um panorama geral sobre o uso da memória pelo sistema, o **/proc** oferece muitas informações sobre o uso da memória pelos processos. A forma mais simples de obter esses dados é lendo o arquivo **/proc/status**, no qual há as seguintes informações sobre o uso da memória:

VmPeak:	3980 kB
VmSize:	3584 kB
VmLck:	0 kB
VmHWM:	1560 kB
VmRSS:	1312 kB
VmData:	416 kB
VmStk:	84 kB
VmExe:	148 kB
VmLib:	1604 kB
VmPTE:	20 kB

Os significados destes campos são:

- **VmPeak** – Área total de memória acessível pelo processo, incluindo bibliotecas compartilhadas.
- **VmSize** – Tamanho total de memória virtual alocado pelo processo.
- **VmLck** – Total de memória travada na RAM, memória que não pode sofrer swap para o disco.
- **VmHWM** – Total de memória acessível pelo processo, incluindo bibliotecas compartilhadas, residente na RAM.
- **VmRSS** – Memória residente na RAM.
- **VmData** – Memória no segmento de dados.
- **VmStk** – Memória no segmento de pilha.
- **VmExe** – Memória marcada como executável ou do segmento de código.
- **VmLib** – Memória acessível como bibliotecas.
- **VmPTE** – Tamanho da tabela de páginas.

O arquivo **/proc/[PID]/statm** também apresenta informações sobre memória, porém os dados são apresentados em uma linha dividida em 7 colunas, seus campos são:

```
$ cat statm
896 292 175 37 0 125 0
```

154

- Tamanho total do programa.
- Tamanho total residente na RAM.
- Tamanho total de memória compartilhada.
- Tamanho total do segmento de texto.
- Tamanho total de bibliotecas acessíveis.
- Soma do segmento de dados e pilha.
- Tamanho total de páginas alteradas recentemente (esta informação é usada em algoritmos de seleção de páginas de memória que podem ser movidas para o disco – swap – de forma a liberar memória RAM para outros processos).

Memória do Sistema

O arquivo **/proc/meminfo** apresenta dados sobre o uso geral de memória pelo sistema. Nos campos desse arquivo há as quantidades de memória alocadas na RAM e a quantidade de memória que sofreu swap. Além disso deles, é possível obter dados sobre a memória virtual do sistema e a quantidade de memória alocada no nível dos usuários e do núcleo. Um exemplo desse arquivo e os significados dos seus campos podem ser vistos a seguir:

```
$ cat /proc/meminfo

MemTotal:          1016304 kB
MemFree:           143352 kB
Buffers:              32432 kB
Cached:               381416 kB
SwapCached:           0 kB
Active:                377204 kB
Inactive:             450616 kB
Active(anon):        260168 kB
Inactive(anon):      179880 kB
Active(file):         117036 kB
Inactive(file):       270736 kB
Unevictable:          8 kB
Mlocked:              8 kB
HighTotal:          132816 kB
HighFree:            232 kB
LowTotal:           883488 kB
LowFree:            143120 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:                36 kB
Writeback:              0 kB
AnonPages:            4 13956 kB
Mapped:                 70128 kB
Slab:                  16308 kB
SReclaimable:          7720 kB
SUnreclaim:             8588 kB
PageTables:             3196 kB
```

```

NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:           508152 kB
Committed_AS:          931708 kB
VmallocTotal:        122880 kB
VmallocUsed:         4796 kB
VmallocChunk:        80244 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:           4096 kB
DirectMap4k:             40952 kB
DirectMap4M:             864256 kB

```

Arquivos Abertos de cada Processo

O **/proc** apresenta informações sobre cada arquivo aberto por um processo. Dentro do diretório de cada processo há dois diretórios sobre arquivos abertos, são eles: **/proc/PID/fd/** e **/proc/PID/fdinfo**.

Além destes diretórios, o arquivo **/proc/PID/status** contém o campo **FDSsize**, que apresenta o tamanho do vetor de descritores de arquivos. O tamanho deste vetor corresponde ao número máximo de arquivos que um processo pode manter aberto ao mesmo tempo. Para descobrir quantos estão abertos e quais são eles, há o diretório **/proc/PID/fd**.

```
$ ls /proc/18772/fd/  
0 1 10 2 3 4 5
```

157

Gerência de Processos

```
$ ls -l /proc/18772/fd/  
total 0  
lrwx----- 1 olsen olsen 64 2009-12-08 16:02 0 -> /dev/pts/0  
lrwx----- 1 olsen olsen 64 2009-12-08 16:02 1 -> /dev/pts/0  
l-wx----- 1 olsen olsen 64 2009-12-08 16:02 10 -> pipe:[123400]  
lrwx----- 1 olsen olsen 64 2009-12-08 15:56 2 -> /dev/pts/0  
lr-x----- 1 olsen olsen 64 2009-12-08 16:02 3 -> /usr/local/share/man  
lr-x----- 1 olsen olsen 64 2009-12-08 16:02 4 -> /usr/local/share/man  
lr-x----- 1 olsen olsen 64 2009-12-08 16:02 5 -> /usr/share/man
```

Há, também, o diretório **/proc/PID/fdinfo**, o qual apresenta informações sobre os arquivos abertos pelo processo. Um exemplo desse arquivo e seus significados pode ser visto a seguir:

```
$ cat /proc/18772/fdinfo/  
0 1 10 2 3 4 5  
olsen@netbook man $ cat /proc/18772/fdinfo/10  
pos: 0  
flags: 04001
```

- **pos**: posição de leitura dentro do arquivo, em bytes.
- **flags**: flags de controle da leitura do arquivo.

O Comando top

Já que o comando **ps** oferece apenas uma fotografia instantânea do seu sistema em um dado momento, normalmente fica difícil ter uma compreensão do que ocorre no todo. O comando **top** fornece um sumário atualizado regularmente dos processos ativos e o emprego dos recursos do sistema operacional.

Como padrão, a tela é atualizada a cada 10 segundos. Os processos mais ativos aparecem no topo da relação. O comando **top** também aceita entrada pelo teclado e permite que enviamos sinais e alteremos os valores de *nice* (prioridade) dos processos, para que possamos observar como nossas ações afetam a condição geral da máquina.

O comando **top** tem de consumir uma porção dos recursos da CPU para mostrar uma atualização a cada 10 segundos. Ele deve ser utilizado apenas para fins de monitoração e diagnóstico.

```
Executa o comando  
$ top  
  
Tasks: 172 total, 3 running, 169 sleeping, 0 stopped, 0 zombie  
Cpu(s): 33.8%us, 8.1%sy, 0.0%ni, 57.8%id, 0.3%wa, 0.0%hi, 0.0%si,  
0.0%st  
Mem: 2052036k total, 1530420k used, 521616k free, 194368k buffers  
Swap: 2000052k total, 0k used, 2000052k free, 858140k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2558	marcos	20	0	188m	43m	19m	S	57	2.2	29:00.24	totem
1733	marcos	20	0	220m	5532	4344	R	13	0.3	6:15.50	pulseaudio
2206	marcos	20	0	449m	12m	31m	S	9	5.6	15:18.48	firefox
977	root	20	0	57204	21m	10m	S	4	1.1	10:00.32	Xorg
2161	marcos	20	0	53380	16m	11m	S	2	0.8	2:31.16	transmission
3593	marcos	20	0	2468	1192	884	R	1	0.1	0:00.08	top
1840	marcos	20	0	77212	25m	7900	S	1	1.3	2:04.53	compiz.real
3149	marcos	20	0	265m	134m	60m	S	1	6.7	9:28.30	soffice.bin
59	root	15	-5	0	0	0	S	0	0.0	0:01.63	kondemand/1
2042	marcos	20	0	32928	19m	4024	S	0	1.0	0:21.98	ubuntuone-syncd
3236	marcos	20	0	41624	16m	10m	S	0	0.8	0:06.24	gnome-terminal
1	root	20	0	2644	1520	1120	S	0	0.1	0:01.46	init
2	root	15	-5	0	0	0	S	0	0.0	0:00.00	kthreadd
3	root	RT	-5	0	0	0	S	0	0.0	0:00.00	migration/0
4	root	15	-5	0	0	0	S	0	0.0	0:02.35	ksoftirqd/0
5	root	RT	-5	0	0	0	S	0	0.0	0:00.00	watchdog/0
6	root	RT	-5	0	0	0	S	0	0.0	0:00.00	migration/1



Atividade

159

Você sabe o que é um **thread**? É uma forma de um processo dividir a si mesmo em duas ou mais tarefas, que podem ser executadas concorrentemente. Nós utilizamos a biblioteca **pthread**, do Linux, para demonstrar algumas situações. Mas existem outras técnicas para a implementação de **threads**. Cada sistema operacional adota sua metodologia. As **threads** também são utilizadas em várias linguagens de programação, principalmente na linguagem Java. Então, faça o seguinte:

- Pesquise os modelos de **threads** e monte uma tabela comparativa com as informações encontradas. **Dica:** verifique os sistemas operacionais Windows, Linux, MAC OS e Solaris.
- Combine com o seu professor de sistemas operacionais um trabalho em conjunto com o seu professor de programação. Monte um programa para troca de mensagens utilizando **threads**. Tente implementar o mesmo programa em várias linguagens de programação e, depois, acompanhe a execução dos programas utilizando os comandos aprendidos por você e analisando os arquivos do **/proc** correspondentes a cada processo.

Referências Bibliográficas

- BIRNBAUM, J. *The Linux /proc file system as a programmers' tool*. Disponível em: <<http://www.linuxjournal.com/article/8381>>. Acesso em: 6 out. 2010.
- JONES, M. T. *Access the linux kernel using the /proc file system*. Disponível em: <<http://www.ibm.com/developerworks/linux/library/l-proc.html>>. Acesso em: 6 out. 2010.
- LAUREANO, M. A. P. *Máquinas virtuais e emuladores*. Brasil: Editora Novatec, 2006.
- LAUREANO, M. A. P.; MAZIERO, C. A. Virtualização: conceitos e aplicações em segurança. In: MAZIERO, C.; GASPARY, L. P.; WEBER, R. F. (Org.). *VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*. 1. ed. Porto Alegre: SBC, 2008. v. 1, p. 139-187.
- LINUX MAN-PAGES. *Linux programmer's manual*. Disponível em: <<http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html>>. Acesso em: 6 out. 2010.
- MAZIERO, C. A. *Sistemas operacionais*. Disponível em: <http://www.ppgia.pucpr.br/~maziero/doku.php/so:livro_de_sistemas_operacionais>. Acesso em: 6 out. 2010.
- MITCHELL, M.; SAMUEL, A. *Advanced linux programming*. Thousand Oaks, CA, USA: New Riders Publishing, 2001.
- RED HAT LINUX 9. *Red hat linux reference guide*. Disponível em: <<http://docs.redhat.com/docs/pt-BR/index.html>>. Acesso em: 6 out. 2010.
- TANENBAUM, A. S. *Modern operating systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.