

**PROF. SÉRGIO NERY SIMÕES**

**SISTEMAS OPERACIONAIS II**

**SERRA**

**2009**

**Governo Federal**

**Ministro de Educação**

Fernando Haddad

**Ifes – Instituto Federal do Espírito Santo**

**Reitor**

Dênio Rebello Arantes

**Pró-Reitora de Ensino**

Cristiane Tenan Schlittler dos Santos

**Coordenadora do CEAD – Centro de Educação a Distância**

Yvina Pavan Baldo

**Coordenadoras da UAB – Universidade Aberta do Brasil**

Yvina Pavan Baldo

Maria das Graças Zamborlini

**Curso de Tecnologia em Análise e Desenvolvimento de Sistemas**

**Coordenação de Curso**

Isaura Nobre

**Designer Instrucional**

Danielli Veiga Carneiro / José Mário Costa Júnior

**Professor Especialista/Autor**

Sérgio Nery Simões

Catalogação da fonte: Rogéria Gomes Belchior - CRB 12/417

S593s Simões, Sérgio Nery

Sistemas operacionais II. / Sérgio Nery Simões. – Vitória: Ifes, 2009. 88p. : il.

1. Sistemas operacionais (Computadores). 2. Linux (Sistema operacional de computador). I. Instituto Federal do Espírito Santo. II. Título.

CDD 005.43

**DIREITOS RESERVADOS**

**Ifes – Instituto Federal do Espírito Santo**

Av. Vitória – Jucutuquara – Vitória – ES - CEP - (27) 3331.2139

**Créditos de autoria da editoração**

**Capa:** Juliana Cristina da Silva

**Projeto gráfico:** Juliana Cristina da Silva / Nelson Torres

**Iconografia:** Nelson Torres

**Editoração eletrônica:** Duo Translations

**Revisão de texto:**

Ilioni Augusta da Costa

Maria Madalena Covre da Silva

**COPYRIGHT** – É proibida a reprodução, mesmo que parcial, por qualquer meio, sem autorização escrita dos autores e do detentor dos direitos autorais.

*Olá, Aluno(a)!*

*É um prazer tê-lo conosco.*

*O Ifes oferece a você, em parceria com as Prefeituras e com o Governo Federal, o Curso Tecnologia em Análise e Desenvolvimento de Sistemas, na modalidade a distância. Apesar de este curso ser oferecido a distância, esperamos que haja proximidade entre nós, pois, hoje, graças aos recursos da tecnologia da informação (e-mails, chat, videoconferência, etc.) podemos manter uma comunicação efetiva.*

*É importante que você conheça toda a equipe envolvida neste curso: coordenadores, professores especialistas, tutores a distância e tutores presenciais porque, quando precisar de algum tipo de ajuda, saberá a quem recorrer.*

*Na EaD – Educação a Distância, você é o grande responsável pelo sucesso da aprendizagem. Por isso, é necessário que se organize para os estudos e para a realização de todas as atividades, nos prazos estabelecidos, conforme orientação dos Professores Especialistas e Tutores.*

*Fique atento às orientações de estudo que se encontram no Manual do Aluno!*

*A EaD, pela sua característica de amplitude e pelo uso de tecnologias modernas, representa uma nova forma de aprender, respeitando, sempre, o seu tempo.*

*Desejamos-lhe sucesso!*

*Equipe do Ifes*

## ICONOGRAFIA

Veja, abaixo, alguns símbolos utilizados neste material para guiá-lo em seus estudos.

### Fala Professor



Fala do professor.

### Conceitos



Conceitos importantes. Fique atento!

### Atividades



Atividades que devem ser elaboradas por você, após a leitura dos textos.

### Indicações



Indicação de leituras complementares, referentes ao conteúdo estudado.

### Atenção



Destaque de algo importante, referente ao conteúdo apresentado. Atenção!

### Reflexão

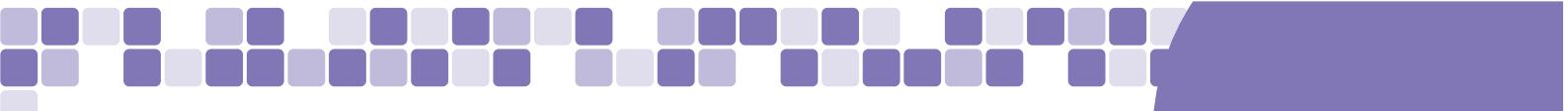


Reflexão/questionamento sobre algo importante, referente ao conteúdo apresentado.

### Anotações



Espaço reservado para as anotações que você julgar necessárias.



# Sumário

## SISTEMAS OPERACIONAIS II

### Cap. 1 - Introdução ao GNU/Linux 9

- 1.1 Visão Geral do GNU/Linux 9
- 1.2 Instalação de Aplicativos no GNU/Linux (Ubuntu) 10
  - 1.2.1 Ferramenta “Adicionar/Remover Programas” 10
  - 1.2.2 Gerenciador de Pacotes Synaptic 14

### CAP. 2 - ÁRVORE DE DIRETÓRIOS DO GNU/LINUX 19

- 2.1 Estrutura básica da árvore de diretórios do GNU/Linux 19

### CAP. 3 - COMANDOS BÁSICOS DO GNU/LINUX 25

- 3.1 Introdução 25
- 3.2 Formato Geral dos Comandos 26
- 3.3 Comandos de Manipulação de Arquivos e Diretórios 26
  - 3.3.1 Comando ls – (list) 27
  - 3.3.2 Comando pwd – (print work directory) 28
  - 3.3.3 Comando cd – (change directory) 28
  - 3.3.4 Comando mkdir – (make directory) 30
  - 3.3.5 Comando rmdir – (remove directory) 31
  - 3.3.6 Comando cat – (concatenate) 31
  - 3.3.7 Comando cp – (copy) 32
  - 3.3.8 Comando mv – (move) 32
  - 3.3.9 Comando rm – (remove) 33
  - 3.3.10 Comando touch 34
  - 3.3.11 Referenciando vários arquivos através de “curingas” 35
  - 3.3.12 Comando grep (Get Regular Expression) 36
  - 3.3.13 Comando sort 37
  - 3.3.14 Comando head 37
  - 3.3.15 Comando tail 37
  - 3.3.16 Comandos more e less 38

## CAP. 4 - EDIÇÃO E COMPILAÇÃO DE PROGRAMAS NO GNU/LINUX 41

- 4.1 Introdução 41
- 4.2 Edição de Programas 41
- 4.3 Compilação e Execução de Programas no GNU/Linux 45
  - 4.3.1 Exibindo Warnings com a opção “-Wall” 46
  - 4.3.2 Exceções 48
- 4.4 Pipes e Redirecionamentos 50
  - 4.4.1 Pipes 50
  - 4.4.2 Redirecionamento de Entrada 51
  - 4.4.3 Redirecionamento de Saída 52
- 4.5 Gerenciamento de Processos no GNU/Linux 53
  - 4.5.1 Comando ps 53
  - 4.5.2 Comando top 54
  - 4.5.3 Interrompendo a execução de um processo 56
  - 4.5.4 Parando momentaneamente a execução de um processo 56
  - 4.5.5 Enviando sinais com o comando kill 56
  - 4.5.6 Programando a rotina de tratamento de sinais 59

## CAP. 5 - MANIPULAÇÃO DE PROCESSOS NO GNU/LINUX 61

- 5.1 Introdução 61
- 5.2 Identificação de um processo 62
- 5.3 Criação de processos 64

## CAP. 6 - MANIPULAÇÃO DE THREADS NO GNU/LINUX 71

- 6.1 Introdução 71
- 6.2 Implementação de Threads 74
  - 6.2.1 A primitiva pthread\_create() 75
  - 6.2.2 A primitiva pthread\_join() 75
  - 6.2.3 Implementação de threads em C 76
  - 6.2.4 Utilização de threads para aumentar o desempenho 80
  - 6.2.5 O problema do compartilhamento de recursos 83

## REFERÊNCIAS BIBLIOGRÁFICAS 87

# APRESENTAÇÃO

*Olá!*

*Meu nome é Sérgio Nery Simões, responsável pela disciplina Sistemas Operacionais II. Atuo como professor do Ifes há cinco anos e já lecionei em outras instituições de ensino superior (UFES, FAVI e Salesiano). Sou graduado em Engenharia de Computação (2000) e Mestre em Informática (2004), ambos pela UFES. Minhas áreas de interesse são: Processamento de Alto Desempenho, Arquiteturas Avançadas de Computador, Arquitetura de Sistemas Operacionais e Sistemas Distribuídos.*

*Nesta disciplina você conhecerá um pouco da aplicação prática dos conceitos de Sistemas Operacionais. O sistema operacional escolhido para isso foi o GNU/Linux, principalmente por ser compatível com o padrão POSIX, o que possibilita aplicar os conhecimentos obtidos nesta disciplina em vários sistemas operacionais.*

*O objetivo deste material é prover uma visão geral da utilização prática de sistemas operacionais, começando pela linha de comando com comandos básicos, passando por edição e compilação de programas, e chegando à utilização de processos e threads em C e entender como o sistema operacional realiza o escalonamento desses processos e threads. Lembre-se, sempre, de que tudo que aprender aqui poderá ser utilizado em diversos outros sistemas operacionais Unix-like.*

*Em geral, para ser bem sucedido neste curso, é importante que se façam os exercícios e se estude regularmente, evitando-se, dessa forma, o acúmulo de conteúdo.*

*Assim, desejo-lhe bastante sucesso!!!*

*Prof. Sérgio Nery Simões*





# INTRODUÇÃO AO GNU/LINUX

Prezado aluno,

Começaremos o primeiro capítulo com uma visão geral do Sistema Operacional GNU/Linux. Em seguida, você verá duas ferramentas para gerenciar a instalação e remoção de aplicativos ao sistema. A primeira, mais simples e intuitiva e a segunda, mais sofisticada e flexível. Como atividades, você deverá instalar os pacotes que lhe permitirão ouvir músicas, ver seus vídeos prediletos, gravar CDs/DVDs e permitir navegar melhor na Internet – através de páginas que necessitem de plugins como o Flash e Java. Também aprenderá a instalar o compilador gcc, juntamente com suas bibliotecas e manuais de documentação do sistema.

Bom estudo!

angue risus ac  
ne velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

## 1.1. Visão Geral do GNU/Linux

O GNU/Linux é um sistema semelhante ao Unix, composto pelo Kernel Linux e pelos utilitários e aplicativos do projeto GNU que o acompanham. O kernel foi iniciado em 1991 por Linus Torvalds, que se baseou no Minix (uma versão reduzida do Unix criada por Andrew S. Tanenbaum para fins didáticos) e posteriormente contou com a colaboração de vários outros programadores. Quando a primeira versão do kernel ficou pronta, Torvalds juntou o kernel a um conjunto de utilitários e aplicativos do projeto GNU, criado por Richard Stallman, em 1984, que contou com a participação de diversos colaboradores. Stallman fundou a *Free Software Foundation*, uma fundação cujo objetivo principal é promover o desenvolvimento e a utilização de software livre.

Posteriormente, várias organizações passaram a organizar, juntamente com o Linux, os utilitários e aplicativos, personalizá-los e distribuir cópias para as pessoas. Essas cópias personalizadas passaram a se chamar de distribuição do GNU/Linux. Dentro as distribuições mais conhecidas e utilizadas encontram-se: Ubuntu, Debian, Suse, Red Hat, Slackware, Mandriva, Fedora, Kurumin e Knoppix.

**Fala Professor**

Nigrae risus at  
ne velit at tellus.  
massa portitor  
insectetur magna.

É importante lembrar que, atualmente, o GNU/Linux está cada vez mais sendo adotado por grandes empresas em ambientes de produção. Com isso, o conhecimento desse sistema operacional torna-se imprescindível aos profissionais modernos. Além disso, no GNU/Linux é mais fácil demonstrar a aplicação de diversos conceitos de sistemas operacionais, por ser um sistema aberto. Por último, nunca é demais lembrar que a maior parte do conhecimento adquirido na utilização de sistemas GNU/Linux é aplicável também em sistemas Unix-like: BSD (e seus derivados FreeBSD, NetBSD e OpenBSD), Solaris, AIX, HP-UX, etc.

## 1.2. Instalação de Aplicativos no GNU/Linux (Ubuntu)

Nesta disciplina, adotaremos a distribuição Ubuntu Desktop 8.04 (a mesma vista em sistemas operacionais I) como distribuição oficial, devido a sua facilidade de uso. Recomendo fortemente que utilize essa distribuição para facilitar o trabalho dos tutores no esclarecimento de dúvidas, pois todo o material foi desenvolvido baseado nessa distribuição. Na disciplina Sistemas Operacionais I, você aprendeu a instalar o Ubuntu Desktop 8.04. Assim, assumindo que você já sabe instalar o Ubuntu e que você está conectado na Internet, começaremos mostrando como instalar algumas aplicações.

Há diversas maneiras de instalar ou remover programas no Ubuntu. Vamos conhecer duas delas. Uma mais simples, através da ferramenta “**Adicionar/Remover Programas**” e outra, através de uma ferramenta um pouco mais sofisticada, o “**Gerenciador de pacotes Synaptics**”.

### 1.2.1. Ferramenta “Adicionar/Remover Programas”

Primeiramente clique no menu “**Aplicações**” e em seguida “**Adicionar/Remover programas**”. Será aberta uma janela semelhante à da Figura 1-1.

Para obter acesso a todos os aplicativos disponíveis e não somente às aplicações suportadas pela Canonical (distribuidora do Ubuntu), no item Exibir, substitua “**Aplicações Suportadas**” por “**Todos os aplicativos disponíveis**”, conforme a Figura 1-2.

## Introdução ao GNU/Linux

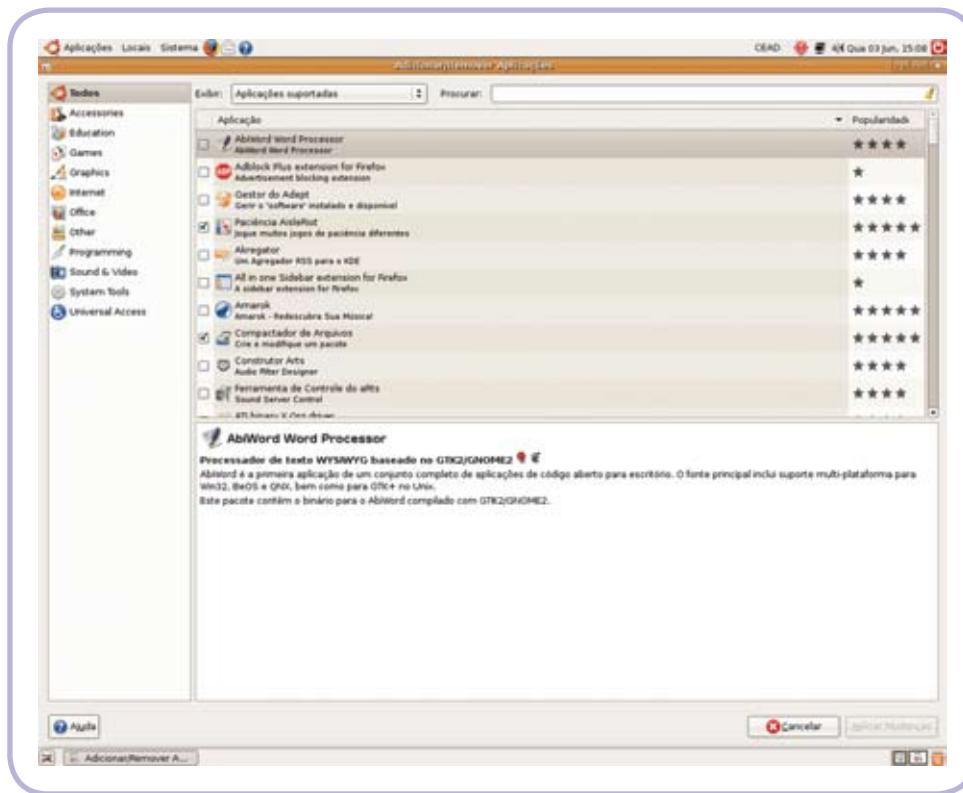


Figura 1-1: Um utilitário para adicionar e remover programas

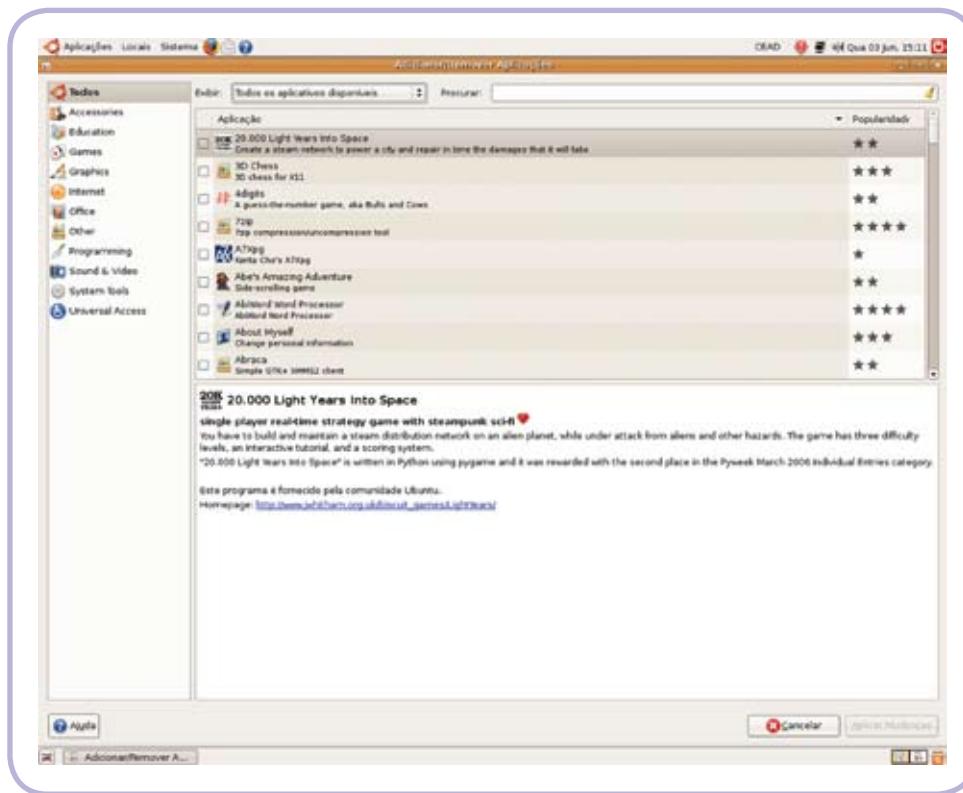


Figura 1-2: Adicionar e remover programas – exibindo todos aplicativos disponíveis

Agora vamos instalar um aplicativo como exemplo. O aplicativo escondido é o “**Gerenciador de Pacotes Synaptic**” – que também permite adicionar ou remover aplicações, porém de forma mais avançada. Para instalá-lo, clique na caixa ao lado de Procurar e digite **Synaptic** ou simplesmente procure Ferramentas de Sistema no painel à esquerda e lo-

calize o Synaptic no painel à direita, rolando a barra de rolagem. Após localizá-lo, selecione-o, marcando a caixa de seleção ao lado do nome do aplicativo, e clique em **aplicar mudanças**. O programa solicita sua senha e em seguida inicia o download do arquivo e consequentemente a instalação do aplicativo. Após estar concluída a instalação, você verá uma janela semelhante à da Figura 1-3.

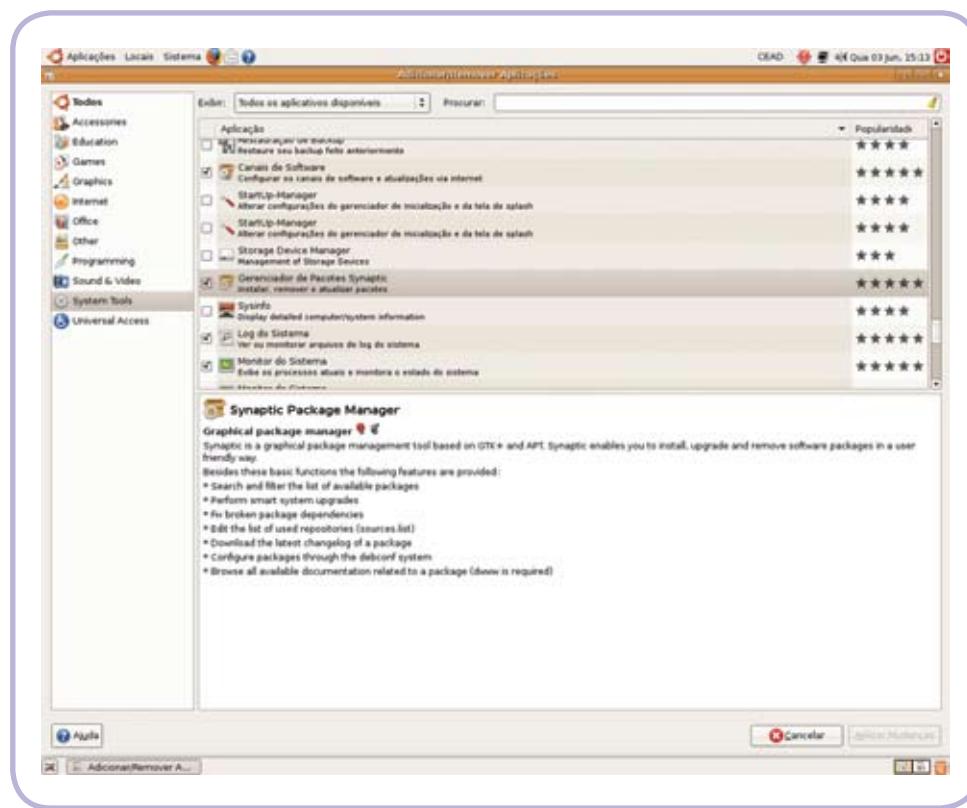


Figura 1-3: Instalando um aplicativo

O seu sistema vem instalado apenas com o básico. Assim, como atividade, indicarei alguns utilitários e aplicativos para tornar a utilização do sistema mais confortável e mais ampla. Algumas tarefas como, por exemplo, a navegação na internet ou a visualização de vídeos, só poderão ser realizadas plenamente após a instalação dos aplicativos sugeridos abaixo. Após estas as atividades, veremos como utilizar o Gerenciador de Pacotes Synaptic.

## Atividades



### Atividades

1. Utilizando o utilitário de adicionar e remover programas visto anteriormente, instale os aplicativos/utilitários abaixo. Lembre-se de que você pode selecionar diversos aplicativos antes de clicar em aplicar.

- a. **Ubuntu restricted extras** – aplicações com restrição de copyright (mp3, avi, mpeg, TrueType, Java, Flash)
- b. **7Zip** – ferramenta de compatação/descompactação
- c. **Adobe Flash Plugin 10** – Plugin Flash para navegadores web (Ex: Firefox, Seamonkey).
- d. **GIMP** – Editor de Imagens
- e. **Inkscape** – Editor de Imagens vetoriais
- f. **Gstreamer Plugins** – Plugins de vídeo (instale todos)
- g. **K3b** – Programa para Gravação de CDs/DVDs
- h. **Macromedia Flash Plugin** – plugin para navegadores web
- i. **Mplayer Movie Player** – Reprodutor Multimídia
- j. **OpenOffice.org Office Suite** – Suíte completa de produtividade openOffice
- k. **SAMBA** – Integração em rede com máquinas Windows (instale apenas se estiver conectado em rede)
- l. **Sun Java 6.0 Plugin** – Plugin Java para navegadores web
- m. **Sun Java 6 Runtime** – plataforma padrão para o ambiente Java (JRE)
- n. **Suporte a idiomas** – configurar suporte a idiomas em seu sistema
- o. **Wine Windows Emulator** – Camada para rodar aplicações Windows no Linux



## Atividades

Após a instalação desses aplicativos você poderá: ouvir músicas em diversos formatos, assistir aos vídeos em formatos proprietários, navegar na Web visualizando melhor as páginas que necessitem da instalação de plugins e fontes especiais, gravar CDs/DVDs, etc. Se quiser, instale outras aplicações e também alguns jogos, pois o sistema possui uma variedade deles – mas não gaste seu tempo todo jogando, afinal você precisa estudar SO2!

### 1.2.2. Gerenciador de Pacotes Synaptic

Outra forma de instalar aplicativos em seu sistema é através da ferramenta **Gerenciador de Pacotes Synaptic**. Apesar de mais complexa, esta ferramenta é mais sofisticada e mais flexível que a ferramenta “Adiciona/Remover Programas”. Por exemplo, existem alguns programas (como o compilador GCC, por exemplo) que não podem ser instalados pela ferramenta apresentada anteriormente. Nesses casos, será necessária a utilização do gerenciador de pacotes Synaptic – ou de alguma outra ferramenta mais sofisticada.

#### Fala Professor

*Angus risus ac  
e velit at tellus.  
massa porttitor  
insectetur magna.*

Nas distribuições Debian e seus derivados (Ubuntu é derivado do Debian), pacotes são programas em um formato especial, disponíveis para instalação. A distribuição Ubuntu 8 possui repositórios com cerca de 25 mil pacotes, mas esse número pode aumentar ou diminuir de acordo com a quantidade de repositórios habilitados no sistema.

Para facilitar a localização, a lista de pacotes é dividida em seções em que os pacotes são agrupados apropriadamente. Para abrir o Synaptic, clique em **Sistema=>Administração=>Gerenciador de Pacotes Synaptic**. A aplicação solicita sua senha e, após digitá-la, uma janela semelhante à Figura 1-4 se abrirá.

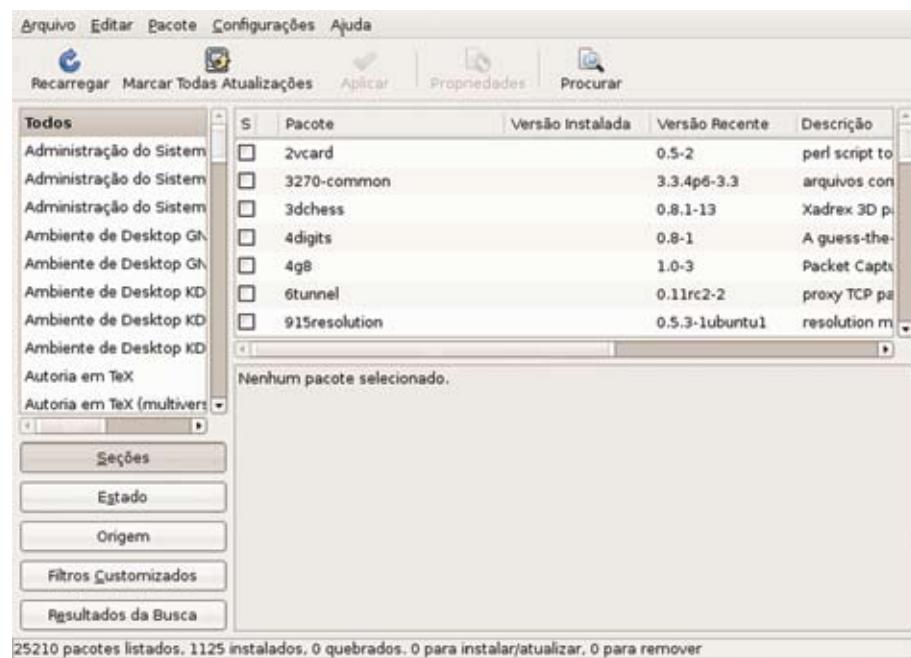


Figura 1-4: Gerenciador de Pacotes Synaptic

A interface exibe, além do menu, 5 botões disponíveis (que são as operações realizadas mais frequentemente na ferramenta), são eles: (i) **Recarregar**, (ii) **Marcar todas as atualizações**, (iii) **Aplicar**, (iv) **Propriedades** e (v) **Procurar**.

- i. O botão “**Recarregar**” atualiza a lista de pacotes disponíveis, para saber se há novas versões. Isso ocorre porque os pacotes estão em constante alterações, devido a melhorias e a correções de bugs ou de segurança. Após clicar em recarregar, o sistema baixará a lista de pacotes disponíveis para instalação – não confunda isso com baixar os pacotes. É recomendável que você pressione esse botão antes de atualizar seu sistema ou instalar algum pacote.
- ii. Após recarregar a nova lista de pacotes, o sistema poderá informar, dentre os pacotes instalados, quais podem ser atualizados. Com isso, você pode atualizar seu sistema através do botão **Marcar todas as atualizações**.
- iii. O botão **Aplicar** serve para efetivamente efetuar as alterações – instalação, remoção ou atualização de pacotes. Por exemplo, após pressionar o botão **Marcar todas as atualizações**, o botão **Aplicar** deve ser pressionado para que os pacotes sejam atualizados.
- iv. Caso você queira saber mais informações sobre um pacote individualmente, selecione-o e clique no botão **Propriedades**.
- v. Por último, o botão **Procurar**, como o próprio nome indica, procura pacotes pra você, bastando para isso apenas digitar o nome do pacote, ou parte dele, ou simplesmente algum termo relacionado ao pacote.

Agora vamos ao que interessa, ou seja, instalar alguns utilitários e aplicativos que utilizaremos no desenrolar desta disciplina. Neste caso, instalaremos o compilador gcc e alguns manuais de ajuda do sistema. Para isso pressione o botão **Procurar**, digite “gcc” e presione <enter>. Você obterá um resultado semelhante ao da Figura 1-5. Todos os pacotes relacionados ao termo “gcc” são listados, neste caso, 121 pacotes listados. Inicialmente, selecione para instalação os pacotes: **gcc** (compilador) e **gcc-doc** (documentação do gcc). Em seguida, clique com o botão direito em cima de **gcc**, e verá que este pacote recomenda a instalação de um outro pacote: **libc6-dev** (que são as bibliotecas), que também deve ser marcado para instalação. Clique novamente com o botão direito em **gcc** e verá que ele sugere a instalação do pacote **manpages-dev** (documentação de desenvolvimento),

marque-o também. Somente após selecionar esses quatro pacotes (**gcc**, **gcc-doc**, **libc6-dev** e **manpages-dev**) clique no botão **Aplicar**. O programa avisa que será necessário fazer o download, confirme e aguarde a instalação.

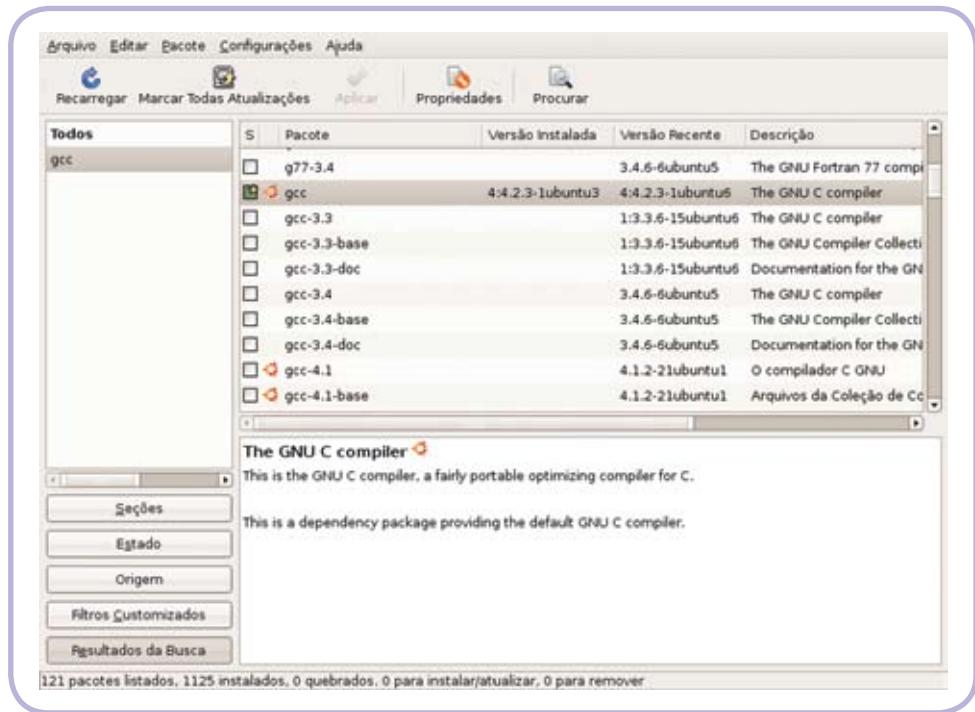


Figura 1-5: Instalação de pacotes através do Synaptic

### Fala Professor

Angue risus at  
ne velit at tellus  
massa porttitor  
issectetur magna.

*Pronto! Você tem o compilador gcc instalado. Mas você deve estar se perguntando: como saber os pacotes que devem ser selecionados para instalação? Parece complexo à primeira vista. Porém, todas essas informações podem ser encontradas facilmente na Internet e em livros específicos de programação. Ou simplesmente com a prática e o dia-a-dia. O que você fez na verdade foi simples: em vez de instalar apenas o compilador gcc, você instalou também as bibliotecas necessárias para compilação, bem como os manuais de documentação para consultas futuras.*

Agora faça as atividades a seguir para verificar seus conhecimentos.

### Atividades



#### Atividades

2. Utilizando o gerenciador de pacotes Synaptic, instale os aplicativos/utilitários abaixo:

- a. **glibc-doc** – documentação para a biblioteca de desenvolvimento C
- b. **manpages** – (*manual pages*) documentação dos sistemas GNU/Linux
- c. **manpages-dev** – documentação para desenvolvimento nos sistemas GNU/Linux
- d. **manpages-posix** – documentação do padrão POSIX
- e. **manpages-posix-dev** – documentação para desenvolvimento no padrão POSIX
- f. **manpages-pt** – documentação do padrão POSIX em português
- g. **manpages-pt** – documentação para desenvolvimento no padrão POSIX em português

**Atividades**

Para obter mais informações, consulte:

Mazioli, Gleydson. **Guia Foca Linux.** ([www.guiafoca.org](http://www.guiafoca.org)). 2007.

Site da distribuição Ubuntu no Brasil (<http://www.ubuntu-br.org/>)

Documentação do Ubuntu no Brasil (<http://wiki.ubuntu-br.org/Documentacao>)

Suporte do Ubuntu no Brasil (<http://www.ubuntu-br.org/suporte>)

Site do projeto GNU (<http://www.gnu.org/>)

Informações sobre sistemas Linux (<http://pt.wikipedia.org/wiki/Linux>)

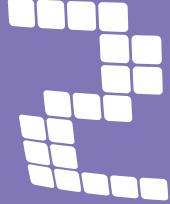
Dicas, notícias e tutoriais sobre Linux (<http://br-linux.org/>)

Comunidade Linux no Brasil (<http://www.vivaolinux.com.br/>)

Apostilas gratuitas e documentação sobre o Linux (<http://www.dicas-l.com.br/>)

**Indicações**





# ÁRVORE DE DIRETÓRIOS DO GNU/LINUX

Prezado aluno,

Agora que você já sabe como instalar aplicativos em seu sistema utilizando a interface gráfica, veremos um pouco da utilização da interface linha de comando. Começaremos aprendendo a estrutura básica da árvore de diretórios no GNU/Linux.

Bom estudo!

ngue risus at  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

## 2.1. Estrutura básica da árvore de diretórios do GNU/Linux

Nos sistemas GNU/Linux os diretórios estão organizados seguindo uma hierarquia denominada de árvore de diretórios. Esta árvore de diretórios possui um diretório principal, a partir do qual é possível acessar todos os diretórios. Este é chamado de **diretório raiz** (ou root, em inglês) e é representado por uma “/” (barra).

No MS-Windows cada dispositivo, ou partição, é representado por uma letra, por exemplo: o drive de disquete é “A:\”, o disco-rígido é “C:\” e, se tiver mais uma partição, será chamada de “D:\”. Neste caso, o drive de CD/DVD poderia ser representado por “E:\” e, se for conectado um pendrive, esta será representado por “F:/”. É como se cada dispositivo tivesse seu próprio diretório “raiz”. Assim, no Windows não há um diretório raiz único, a partir do qual é possível acessar qualquer diretório no sistema.

ngue risus at  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

No GNU/Linux, todos os dispositivos, como drive de disquetes, CDs/DVDs e pendrives, são montados (ou mapeados) em diretórios (chamados de **pontos de montagens**) abaixo do diretório raiz “/”. Por exemplo, o ponto de montagem do disquete normalmente é o diretório “/media/floppy”, o do CD/DVD é “/media/cdrom”. Ou seja, para acessar o CD

você deve entrar no diretório “**/media/cdrom**”. Assim, no GNU/Linux todos os dispositivos podem ser acessados a partir do diretório raiz “**/**”, como se fosse uma única e grande árvore hierárquica de diretórios.

Abaixo do diretório raiz estão alguns sub-diretórios do sistema, organizados para facilitar a utilização, segundo a FHS (Filesystem Hierarchy Standard) – padrão em ambientes Unix-like.

<b>/bin</b>	Contém arquivos programas do sistema que são usados com frequência pelos usuários.
<b>/boot</b>	Contém arquivos necessários para a inicialização do sistema.
<b>/media</b>	Ponto de montagem de dispositivos diversos do sistema (rede, pen-drives, CD-ROM em distribuições mais novas).
<b>/dev</b>	Contém arquivos usados para acessar dispositivos (periféricos) existentes no computador.
<b>/etc</b>	Arquivos de configuração de seu computador local.
<b>/home</b>	Diretórios contendo os diretórios das contas (“homes”) dos usuários.
<b>/lib</b>	Bibliotecas compartilhadas pelos programas do sistema e módulos do kernel.
<b>/mnt</b>	Ponto de montagem temporário.
<b>/proc</b>	Sistema de pseudo-arquivos do kernel. Este diretório não existe fisicamente no disco rígido, ele é colocado neste diretório pelo kernel e usado por diversos programas que fazem sua leitura, que verificam configurações do sistema ou que modificam o funcionamento de dispositivos do sistema através da alteração em seus arquivos.
<b>/root</b>	Diretório do usuário root.
<b>/sbin</b>	Diretório de programas usados pelo superusuário (root) para administração e controle do funcionamento do sistema.
<b>/tmp</b>	Diretório para armazenamento de arquivos temporários criados por programas.
<b>/usr</b>	Contém maior parte de seus programas. Normalmente acessível somente como leitura.
<b>/var</b>	Contém a maior parte dos arquivos que são gravados com frequência pelos programas do sistema, e-mails, spool de impressora, cache, etc.

Existem três tipos de usuários em sistemas GNU/Linux: os usuários comuns, o superusuário e os usuários de sistema. Os usuários comuns são as pessoas com login e senha cadastradas para usarem o sistema. Superusuário ou usuário root é como é chamado o administrador do sistema. E os usuários de sistema não são pessoas com login/senha, apenas usuários cadastrados pelo próprio sistema para a execução de alguns serviços.

As contas dos usuários, ou seja, seus arquivos e subdiretórios, são localizadas no diretório **/home**. Por exemplo, se tivermos dois usuários comuns no sistema, com os logins **joao** e **maria**, provavelmente teremos subdiretórios de mesmo nome dentro do diretório **/home**. Em outras palavras, teremos o

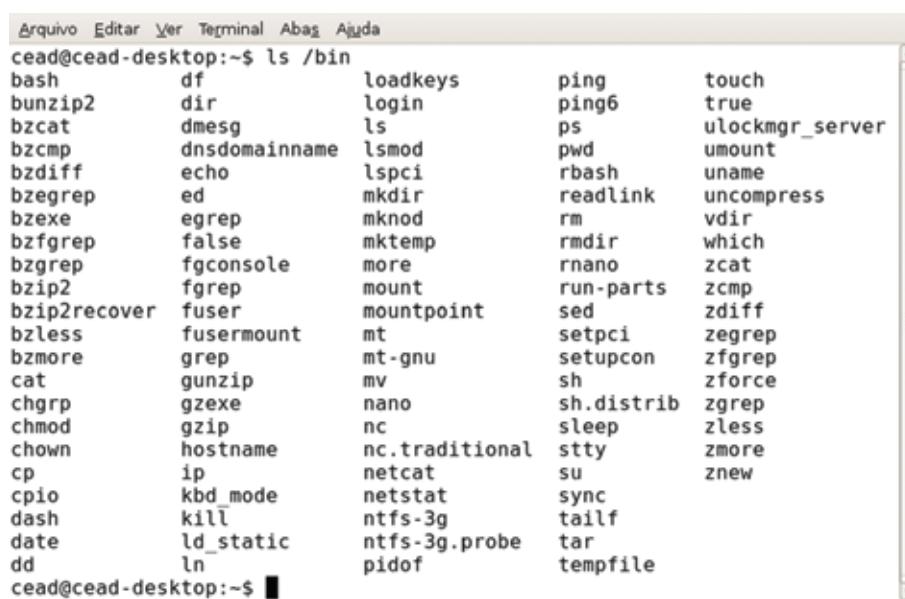
diretório **/home/joao** (também chamado de “*home* do João”) para armazenar os dados pessoais do usuário João, e **/home/maria** (também chamado de “*home* da Maria”) para armazenar os dados da usuária Maria.

*Nigra risus at  
e volit at tellus.  
massa portitor  
nsectetur magna.*

### Fala Professor

Normalmente, os sistemas são configurados para que um usuário tenha acesso apenas ao seu próprio “diretório *home*”, não tendo acesso ao diretório de outro – por questões de segurança. Há também o diretório de uso temporário **/tmp**, ao qual todos os usuários possuem acesso. Mas, em geral, os usuários estão confinados apenas a esses dois diretórios, ficando os diretórios de sistema sem permissão de acesso ou com permissão de uso restrito (somente leitura). Não estou considerando aqui, como diretórios de sistema, os diretórios de montagem de dispositivos como disquete, CD/DVD ou pendrive aos quais, obviamente, o usuário que os montou possuirá acesso.

O diretório **/bin** é interessante, pois ele possui os comandos básicos que todo usuário pode executar. Assim, caso esqueça algum comando básico, basta listar o conteúdo do diretório **/bin** através do comando “**ls /bin**” (veremos o comando “**ls**” mais a frente. Para executar este comando, vá em **Aplicações=>Acessório=>Terminal** ou **Aplicações=>Acessório=>Consola** em algumas traduções. Uma janela se abrirá com o prompt de comando, aguardando você digitar um comando. Digite o comando “**ls /bin**” e você obterá uma saída semelhante à da Figura 2-1.



```
Arquivo Editar Ver Terminal Abas Ajuda
cead@cedad-desktop:~$ ls /bin
bash      df      loadkeys    ping     touch
bunzip2   dir      login      ping6    true
bzcat     dmesg   ls         ps      unlockmgr_server
bzcmp     dnsdomainname lsmod    pwd     umount
bzdiff   echo     lspci     rbash   uname
bzegrep  ed      mkdir     readlink  uncompress
bzexe    egrep   mknod    rm      vdir
bzfgrep  false   mktemp   rmdir   which
bzgrep   fgconsole more    rnano   zcat
bzip2    fgrep   mount   run-parts  zcmp
bzip2recover fuser  mountpoint sed    zdiff
bzless   fusermount mt     setpci  zegrep
bzmore   grep    mt-gnu   setupcon zfgrep
cat     gunzip  mv      sh      zforce
chgrp   gzexe   nano    sh.distrib zgrep
chmod   gzip    nc      sleep   zless
chown   hostname nc.traditional stty   zmore
cp     ip      netcat  su      znew
cpio   kbd_mode netstat sync
dash   kill    ntfs-3g tailf
date   ld_static ntfs-3g.probe tar
dd     ln      pidof   tempfile
cead@cedad-desktop:~$
```

Figura 2 6: Conteúdo do diretório **/bin**

Outro diretório interessante é o /proc, que na verdade é um pseudo-diretório, pois seu conteúdo não está fisicamente no disco-rígido e sim na memória e é nele que são armazenadas as informações sobre os processos. Além disso, neste diretório também se encontram informações sobre o Hardware de seu computador. Por exemplo, para saber informações a respeito de seu processador – ou até mesmo se certificar sobre a configuração de seu processador –, você pode consultar o conteúdo do pseudo-arquivo “**cpuinfo**” dentro do diretório /proc. Para exibir na tela o conteúdo deste arquivo, digite “**cat /proc/cpuinfo**”. O resultado será algo semelhante à Figura 2-2. Note que se você tiver rodando GNU/Linux a partir de uma máquina virtual, o resultado exibido não será o seu processador real e sim o processador virtual da máquina virtual.

```

Arquivo Editar Ver Terminal Abas Ajuda
cead@cead-desktop:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 15
model name     : Genuine Intel(R) CPU           2140 @ 1.60GHz
stepping        : 2
cpu MHz        : 1200.000
cache size     : 1024 KB
physical id    : 0
siblings        : 2
core id         : 0
cpu cores      : 2
fdt_bug        : no
hlt_bug        : no
f00f_bug       : no
coma_bug       : no
fpu             : yes
fpu_exception  : yes
cpuid level   : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
ov ov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_t
sc arch_perfmon pebs bts dni monitor ds_cpl est tm2 ssse3 cx16 xtr lahf_lm
bogomips       : 3194.73
clflush size   : 64

```

Figura 2-7: Exibindo informações sobre o seu processador

Outro pseudo-arquivo que pode ser útil na verificação do hardware de seu computador é o **meminfo** que também se encontra no diretório /proc. Para exibir seu conteúdo na tela basta digitar “**cat /proc/meminfo**”, e você obterá um resultado semelhante ao da Figura 2-3. Com esse comando, é possível verificar a quantidade real de memória de seu computador, além de uma série de outras, como memória utilizadas por buffers, por swap, etc. Lembre-se novamente de que, caso esteja rodando em uma máquina virtual, o resultado será da memória reservada para a máquina virtual e não da memória real.

## Árvore de Diretórios do GNU/Linux

```

Arquivo Editar Ver Terminal Abas Ajuda
cead@cead-desktop:~$ cat /proc/meminfo
MemTotal:      2066180 kB
MemFree:       1189888 kB
Buffers:        31192 kB
Cached:        415820 kB
SwapCached:      0 kB
Active:        481992 kB
Inactive:      240884 kB
HighTotal:     1169984 kB
HighFree:       470372 kB
LowTotal:      896196 kB
LowFree:        719516 kB
SwapTotal:    1052248 kB
SwapFree:      1052248 kB
Dirty:          780 kB
Writeback:       0 kB
AnonPages:     276000 kB
Mapped:         88164 kB
Slab:          26604 kB
SReclaimable:   18004 kB
SUnreclaim:     8600 kB
PageTables:     2224 kB
NFS_Unstable:    0 kB
Bounce:          0 kB
CommitLimit:   2085336 kB
Committed_AS:  592292 kB

```

Figura 2-8: Exibindo informações sobre a memória de seu computador

Agora, consulte o Guia Foca ([guiafoca.org](http://guiafoca.org)) para obter mais informações sobre a estrutura de diretórios do GNU/Linux. Em seguida faça as atividades ao lado.

### Atividades

1. Digite os comandos abaixo e informe o resultado.

a. **ls /boot**

b. **ls /dev**

c. **ls /etc**

d. **ls /home**

e. **ls /media**

f. **ls /proc**

g. **ls /root**

h. **ls /tmp**



### Atividades

**Atividades**

i. ls /usr

j. ls /var

k. Houve algum diretório cujo conteúdo não foi exibido? Se sim, o que você acha que foi a causa?

2. Sabendo que geralmente o diretório **/usr** é o diretório padrão no GNU/Linux para armazenar os aplicativos instalados, faça uma analogia desse diretório com um diretório equivalente no Windows XP.

3. Sabendo que o diretório **/home** é o diretório padrão no GNU/Linux para armazenar as contas dos usuários, qual o diretório no Windows XP que possui a mesma finalidade?

Para obter mais informações, consulte:

**Indicações**

Mazioli, Gleydson. **Guia Foca Linux**. ([www.guiafoca.org](http://www.guiafoca.org)). 2007.

Neves, Julio Cesar. **Programação Shell Linux**. 6.ed. Brasport, 2006.

GNU Bash Reference Manual (<http://www.network-theory.co.uk/docs/bashref/>)

Dicas, notícias e tutoriais sobre Linux (<http://br-linux.org/>)

Comunidade Linux no Brasil (<http://www.vivaolinux.com.br/>)

Apostilas gratuitas e documentação sobre o Linux (<http://www.dicas-l.com.br/>)



# COMANDOS BÁSICOS DO GNU/LINUX

*Prezado aluno,*

*Após ter aprendido um pouco sobre a estrutura básica de diretórios no GNU/Linux descoberto que o seu “diretório home” (onde seus arquivos são armazenados) ficou dentro de “/home/<usuario>”, vamos aprender alguns comandos de manipulação de arquivos e diretórios e com eles criar, remover, copiar, mover arquivos e diretórios dentro do seu próprio “diretório home”.*

*Bom estudo!*

*Nigue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.*

**Fala Professor**

## 3.1. Introdução

Para usar a linha de comando, abra um terminal, conforme visto anteriormente. Um prompt de comando aparecerá aguardando sua digitação, como é apresentado na Figura 3-1. Ao digitar um comando e pressionar <enter>, o interpretador de comandos, denominado Shell, será responsável por interpretar o comando digitado e executá-lo. O Shell é uma camada entre o usuário e o kernel. Existem vários tipos de Shell (interpretadores de comandos), e os mais comuns são: Bourne Shell (**sh**), Bourne-Again Shell (**bash**), Korn Shell (**ksh**) e o C Shell (**csh**).



Figura 3 9: Terminal – interface linha de comando

Por padrão, o **bash** é o shell geralmente utilizado em ambientes GNU/Linux, mas o usuário pode configurar qualquer outro de acordo com sua preferência. No bash, o prompt de comando normalmente é exibido no seguinte formato: o nome do usuário, “@” (arroba), o nome da máquina, “:” dois pontos, o diretório corrente e por fim o “\$” ou “#”, que

serve para identificar se você está conectado como usuário comum ou como usuário administrador do sistema, respectivamente. No caso do exemplo da Figura 3-1, o usuário de login “**sergio**” está conectado na máquina “**ifes**” como usuário comum (não-administrador) devido ao “\$” exibido pelo prompt.

### 3.2. Formato Geral dos Comandos

Na linha de comando do GNU/Linux, os comandos são uma forma de dar ordens ao kernel para que ele execute alguma ação. Geralmente os comandos apresentam o seguinte formato:

**comando [opções] <parametro1> <parametro2> ...**

Os colchetes representam que as opções podem estar presentes ou não na digitação de um comando. Os sinais “<” e “>” indicam que os parâmetros devem ser digitados na linha da comando, e normalmente são obrigatórios, ou seja, devem ser digitados juntamente com o comando.

#### Fala Professor



*Existem dois tipos de comandos: os comandos internos e os externos. Os **comandos internos** são aqueles que se localizam dentro do interpretador de comandos (ou Shell, no nosso caso, o Bash) e não no disco. Eles são carregados na memória RAM do computador junto com o interpretador de comandos.*

Quando você executa um comando, o interpretador de comandos verifica primeiro se ele é um comando interno e, caso não seja, é verificado se é um comando externo. Exemplos de comandos internos são: “**cd, exit, echo, bg, fg, source, help**”. Já os **comandos externos** são aqueles que se localizam no disco. Esses comandos são procurados no disco usando a variável de ambiente PATH e executados assim que encontrados.

### 3.3. Comandos de Manipulação de Arquivos e Diretórios

Nesta seção veremos alguns comandos básicos de manipulação de arquivos e diretórios.

*Em geral, o nome do comando é baseado em um acrônimo das palavras em inglês que eles referenciam: ls – list, cp – copy, mv – move, rm – remove, cd – change dir, etc.*

ngue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

Vejamos alguns exemplos:

### 3.3.1.Comando ls – (list)

Recorde que você já utilizou o comando “ls” como atividade no capítulo anterior para listar arquivos e diretórios. Vamos ver este comando em mais detalhes. A sintaxe deste comando é:

**ls [opções] [arquivo]**

As opções mais comuns para o comando ls são:

Opção	Descrição
-l	Lista os arquivos e diretórios no formato longo, exibindo suas permissões, o dono e o grupo do arquivo/diretório, seu tamanho, data de alteração, etc.
-h	Combinada com a opção -l lista os tamanhos dos arquivos em KBytes, Mbytes ou GBytes.
-a	Exibe todos os arquivos, inclusive os ocultos. Obs: No GNU/Linux os arquivos ocultos começam com “.” (ponto) e não são listados, a menos que esta opção seja utilizada.
-d	Lista os nomes dos diretórios, em vez de seu conteúdo.

A Figura 3-10 mostra um exemplo da execução deste comando:

```
Arquivo Editar Ver Terminal Ajuda
cedad@cead-desktop:~$ ls -l livro.txt
-rw-r--r-- 1 cead cead 27 2009-06-04 18:43 livro.txt
cead@cead-desktop:~$
```

Figura 3 10: Terminal – interface linha de comando

São exibidas várias informações, já que a opção de listagem em formato longo foi utilizada. O primeiro campo representa as permissões do arquivo:

**-rw-r--r--**

O primeiro caracter identifica o tipo de arquivo, se for um “-” é um arquivo regular, se for um “d” será um diretório, mas podem existir outras variações (por exemplo, “l” representa um link, “b” um dispositivo de blocos, “c” um dispositivo orientado a caracter). Os outros nove caracteres na verdade são três triplas, e cada tripla é composta pelas letras “rwx” ou por “-” o que representa as permissões de leitura (read), escrita (write) e execução (execute). O “-” (traço) representa a falta da respectiva permissão. Por exemplo, a sequencia “rwx” representa as permissões de leitura, escrita e execução ao passo que a sequencia “rw-” representa apenas as permissões de leitura e escrita.

A primeira tripla é a permissão do dono do arquivo (u – user). A segunda tripla é a permissão do grupo (g – group), ou seja, dos usuários que pertencem ao mesmo grupo do arquivo. Por fim, a terceira tripla representa as permissões dos demais usuários (o – other).

Os demais campos são o usuário, grupo tamanho, data/hora de modificação e nome do arquivo.

### 3.3.2.Comando pwd – (*print work directory*)

Para saber em que diretório você se encontra, utilize o comando “pwd”, pois ele exibe o diretório corrente de trabalho. Observe o exemplo abaixo.

```
sergio@ifes:~/teste$ pwd  
/home/sergio/teste
```

Nesse caso o diretório corrente de trabalho é “/home/sergio/teste”. E se desejar entrar em outro diretório em vez desse?

### 3.3.3.Comando cd – (*change directory*)

O comando “cd” serve para mudar de diretório. Para utilizá-lo, simplesmente digite “cd” seguido do diretório que você deseja, conforme a sintaxe abaixo:

```
cd [dir]
```

Nesse caso, “dir” representa o diretório ao qual você deseja ir. Por exemplo, supondo que você deseja mudar para o diretório /tmp, basta digitar “cd /tmp”. Observe a sequência de comandos abaixo:

```
sergio@ifes:~/teste$ pwd  
/home/sergio/teste  
sergio@ifes:~/teste$ cd /tmp  
sergio@ifes:/tmp$ pwd  
/tmp  
sergio@ifes:/tmp$
```

Os comandos “pwd” utilizados nesse exemplo servem apenas para provar que a mudança de diretórios foi realizada, na prática, você não precisa digitá-los. Mas e se você quisesse voltar ao diretório anterior “/home/sergio/teste”? Você deve estar respondendo: é simples, basta digitar “cd /home/sergio/teste”. Observe o exemplo abaixo (lembre-se que digitei “pwd” apenas para provar que o diretório foi realmente alterado):

```
sergio@ifes:/tmp$ pwd
/tmp
sergio@ifes:/tmp$ cd /home/sergio/teste/
sergio@ifes:~/teste$ pwd
/home/sergio/teste
```

Vemos que sua resposta está correta. Mas será que existem outras maneiras de fazer isso? A resposta é sim. Observe:

```
sergio@ifes:/tmp$ pwd
/tmp
sergio@ifes:/tmp$ cd ~/teste/
sergio@ifes:~/teste$ pwd
/home/sergio/teste
```

Neste exemplo, no lugar de digitar o caminho completo do meu diretório “home” (**/home/sergio**) eu digitei apenas “~” (til). Em sistemas Unix-like o “~” (til) representa o seu diretório home. Assim, todas as vezes que desejar referenciá-lo basta digitar um “~”. Por último, vejamos um exemplo ainda mais simples de como voltar ao diretório anterior:

```
sergio@ifes:/tmp$ pwd
/tmp
sergio@ifes:/tmp$ cd -
/home/sergio/teste
sergio@ifes:~/teste$ pwd
/home/sergio/teste
```

Ou seja, se você digitar “**cd -**” você volta ao diretório anterior de forma bastante rápida. Outra curiosidade é que, se você digitar apenas “**cd**”, você volta ao seu diretório home. Outro diretório interessante é o diretório representado por “..” (também chamado de upper directory ou diretório “pai” do diretório corrente), através dele você sobe um nível na árvore de diretórios independente de onde esteja. Em outras palavras, se você digitar, por exemplo, “**cd ..**” você irá para o diretório pai do diretório atual. Observe:

```
sergio@ifes:~/teste$ pwd
/home/sergio/teste
sergio@ifes:~/teste$ cd ..
sergio@ifes:~$ pwd
/home/sergio
```

Ao executar o comando “`cd ..`” saí do diretório “`/home/sergio/teste`” e fui para o diretório “pai” desse “`/home/sergio`”. É possível usar formas combinadas, por exemplo “`cd ../../`”, para subir mais de um nível na árvore de diretórios. Além destes diretórios, existe o diretório “`”` (ponto), que representa o diretório corrente, e que é bastante utilizado para simplificar a digitação em comandos de cópia ou execução de programas.

Em resumo, temos os seguintes diretórios especiais:

Diretório	Descrição
.	Representa o diretório corrente (atual).
..	Representa o diretório pai do corrente (upper directory).
~	Representa o diretório home do usuário (ex: <code>/home/sergio</code> )

Agora que você já aprendeu um bocado sobre os diretórios especiais e sobre como passear pelos diretórios e examinar seus conteúdos, surge a pergunta: como criar e remover diretórios? Para responder a essa pergunta, vejamos os dois próximos comandos.

### 3.3.4. Comando `mkdir` – (*make directory*)

O comando `mkdir` serve para criar um novo diretório. Sua sintaxe é:

`mkdir dir...`

em que `directory` representa o nome do diretório a ser criado. Você pode criar um ou vários diretórios em um único comando. Observe a sequência de comandos abaixo

```
sergio@ifes:~$ mkdir so2
sergio@ifes:~$ cd so2/
sergio@ifes:~/so2$ mkdir trab1 trab2
sergio@ifes:~/so2$ pwd
/home/sergio/so2
sergio@ifes:~/so2$ ls -l
total 0
drwxr-xr-x 2 sergio sergio 48 2009-03-05 00:30 trab1
drwxr-xr-x 2 sergio sergio 48 2009-03-05 00:30 trab2
```

Nessa sequência, o primeiro `mkdir` cria um diretório chamado “`so2`”. Em seguida, entramos nesse diretório com o comando “`cd so2/`” e através do comando “`mkdir trab1 trab2`” criamos dois outros diretórios: “`trab1`” e “`trab2`”.

### 3.3.5.Comando rmdir – (*remove directory*)

O comando `rmdir` serve para remover um diretório. Para ser removido, é necessário que o diretório esteja vazio e que você possua permissão de gravação. A sintaxe desse comando é:

```
rmdir dir...
```

em que “dir” representa o nome do diretório a ser removido. Você pode remover um ou vários diretórios em um único comando. Exemplo:

```
sergio@ifes:~/aula_so2$ ls -l
total 0
drwxr-xr-x 2 sergio sergio 48 2009-03-05 00:30 trab1
drwxr-xr-x 2 sergio sergio 48 2009-03-05 00:30 trab2
drwxr-xr-x 2 sergio sergio 48 2009-03-05 02:20 trab3
sergio@ifes:~/aula_so2$ rmdir trab1
sergio@ifes:~/aula_so2$ ls -l
total 0
drwxr-xr-x 2 sergio sergio 48 2009-03-05 00:30 trab2
drwxr-xr-x 2 sergio sergio 48 2009-03-05 02:20 trab3
sergio@ifes:~/aula_so2$ rmdir trab2 trab3
sergio@ifes:~/aula_so2$ ls -l
total 0
```

Nesse exemplo, o primeiro `rmdir` remove um único diretório vazio (`trab1`) e o segundo remove dois ao mesmo tempo (`trab2` e `trab3`).

Até o momento vimos comandos para manipular diretórios, agora veremos alguns comandos para manipular arquivos.

### 3.3.6.Comando cat – (concatenate)

O comando “cat” é utilizado para exibir o conteúdo de um arquivo texto ou binário. A sintaxe desse comando é:

```
cat [opções] [arquivo1]...
```

Observe a utilização do comando “cat”:

```
sergio@ifes:~/teste$ cat livro1.txt
Título: Arquitetura de Sistemas Operacionais
Autor: Francis Berenger Machado & Luiz Paulo Maia
Edição: 4ª
- Ano Publicação: 2007
- Editora: LTC
```

O conteúdo do arquivo é apresentado na tela através do comando “cat”. Você pode exibir o conteúdo de vários arquivos sequencialmente através de um único comando cat. Por exemplo, o comando abaixo:

```
sergio@ifes:~/teste$ cat livro2.txt livro3.txt
Título: Fundamentos de Sistemas Operacionais
Autor: Abraham Silberschatz & Peter Baer Galvin & Greg
Gagne
Edição: 6ª
- Ano Publicação: 2004
- Editora: LTC

Título: Sistemas Operacionais Modernos
Autor: Andrew S. Tanenbaum
Edição: 2ª
- Ano Publicação: 2007
- Editora: Pearson Brasil
```

exibe o conteúdo do arquivo “livro2.txt” seguido do conteúdo do arquivo “livro3.txt”.

### 3.3.7.Comando cp – (*copy*)

O comando cp serve para copiar arquivos e/ou diretórios. Sua sintaxe é:

```
cp [opções] origem destino
```

em que “origem” deve ser substituído pelo nome do arquivo de origem a ser copiado e “destino” pelo nome do arquivo de destino. Se for utilizada a opção “-R” é possível copiar diretórios recursivamente. Além disso, é possível utilizar curingas para especificar vários arquivos de origem a serem copiados.

### 3.3.8.Comando mv – (*move*)

O comando mv serve para mover diretórios ou arquivos de um diretório de origem a um diretório de destino. Também serve para renomear arquivos/diretórios, caso os diretórios de origem e destino sejam os mesmos. Sua sintaxe é semelhante à do comando cp:

```
mv [opções] origem destino
```

Nesse caso o arquivo com o nome “origem” deixa de se chamar assim e passa a se chamar “destino”. Em outras palavras, o arquivo de origem deixa de existir após a execução do comando mv. Outra coisa, para mover diretórios, não é necessária a opção “-R”. Você também pode utilizar curingas para mover vários arquivos ao mesmo tempo.

Seguem alguns exemplos: mudar o nome do arquivo “**arq.txt**” para “**arq1.txt**”

```
mv arq.txt arq1.txt
```

Mover o arquivo “**abc.doc**” do diretório “**/tmp**” para o seu “diretório home”.

```
mv /tmp/abc.txt ~/
```

Como vimos, o comando **mv** serve para mover ou renomear arquivos ou diretórios. Agora vejamos um comando para apagar arquivos ou diretórios.

### 3.3.9.Comando **rm** – (*remove*)

Este comando serve para apagar arquivos; mas, se for usado com a opção “**-r**”, também pode apagar diretórios e subdiretórios. Sua sintaxe é:

```
rm [opções] arq1...
```

em que “**arq1**” é o nome do arquivo a ser apagado. Se quiser apagar mais de um arquivo ao mesmo tempo, basta ir digitando sequencialmente os nomes dos demais arquivos a serem apagados. Além disso, também é possível utilizar curingas para referenciar vários arquivos ao mesmo tempo.

As opções mais comuns para o comando **rm** são:

Opção	Descrição
<b>-i</b>	Solicita confirmação antes de remover o arquivo.
<b>-r</b>	Remove diretórios e seus conteúdos recursivamente.
<b>-f</b>	Ignora arquivos não existentes e nunca pede confirmação

Tome muito cuidado antes de apagar um arquivo para evitar apagá-lo acidentalmente, pois este comando não envia o arquivo para uma lixeira, em vez disso ele simplesmente o apaga.

Como vimos anteriormente, a opção **-r** serve para apagar diretórios, e ela apaga mesmo que os diretórios não estejam vazios. Lembre-se de que o comando “**rmdir**” visto anteriormente apagava apenas diretórios vazios. Assim, uma forma de apagar um diretório (vazio ou não) chamado **dir1** seria:

```
rm -r dir1
```

No entanto, em alguns sistemas a opção de confirmação “-i” é habilitada por padrão – sem necessidade de você digitá-la. Caso tenha certeza de que está apagando o diretório correto, você pode utilizar o seguinte comando para apagar um diretório sem pedido de confirmação:

```
rm -rf dir1
```

esse comando apagará o diretório “dir1” e todos os subdiretórios e arquivos dentro dele.

### Fala Professor



**Assim, tome muito cuidado com esse comando e só o execute caso tenha certeza de que o diretório que está apagando é o diretório correto.**

### 3.3.10. Comando touch

Este comando é usado para criar rapidamente arquivos vazios. Também pode ser utilizado para alterar a data/hora de arquivos pré-existentes. Sua sintaxe é:

```
touch arq1...
```

em que “arq1” é o nome do arquivo vazio a ser criado. Você pode criar vários arquivos ao mesmo tempo. Para isso, basta digitar seus nomes em sequência. Por exemplo, o comando:

```
touch abc def 123
```

criará três arquivos: “abc”, “def” “123”. Você deve estar se perguntando: é possível ter arquivos cujo nome começa por número??? A resposta é sim. Caso um arquivo já exista e seja executado o comando touch nele, o arquivo terá a sua data/hora de modificação alterada, mas seu conteúdo permanecerá intacto.

### Fala Professor



*Você pode utilizar esse comando para criar vários arquivos vazios e testar os comandos de manipulação de arquivos neles. Experimente criar um diretório e populá-lo com o comando touch para, em seguida, utilizar os comandos de copiar, mover e remover arquivos e diretórios.*

Nigra risus at  
e velit at tellus.  
Massa porttitor  
assectetur magna.

## Fala Professor

*Outra observação importante: em ambientes UNIX, para um arquivo ser oculto deve começar com “.” (ponto). Experimente criar alguns arquivos ocultos (por exemplo: “.abc”, “.arq1” e “oculto”) e tente lista-los com o comando “ls -l” e com o comando “ls -la” e observe a diferença.*

### 3.3.11. Referenciando vários arquivos através de “curingas”

Os curingas (também chamados de referência global) servem para especificar um ou mais arquivos ou diretórios do sistema de uma só vez. Esse é um recurso que permite a filtragem do que será listado, copiado, apagado, etc. Existem quatro tipos de curingas no GNU/Linux, mas a tabela abaixo apresenta apenas os dois curingas utilizados mais frequentemente:

Curinga	Descrição
*	Representa qualquer caracter em qualquer quantidade de vezes (zero ou mais vezes).
?	Representa qualquer caracter exatamente uma única vez.

Assim, se você quiser copiar todos os arquivos do diretório “teste1” para o diretório “teste2” deve usar o comando abaixo:

```
cp teste1/* teste2/
```

Vejamos outros exemplos. Copiar os arquivos que começam com as letras “so” e terminem com “.txt” do diretório /tmp para o diretório **corrente** (.):

```
cp /tmp/so*.txt .
```

Copiar os arquivos cujo nome tenha 4 caracteres quaisquer do diretório “dir2” para o “pai” do diretório corrente:

```
cp /dir2/???? .. /
```

Copiar os arquivos cujo nome tenha 4 caracteres e comece com “A” do diretório corrente para o diretório **“home”**:

```
cp A??? ~/
```

Neste caso usamos apenas três interrogações, pois o “A” já conta como um caracter.

Vejamos mais um exemplo: copiar, do diretório **pai** do corrente para o diretório **corrente**, os arquivos que comecem com a letra “a”, terminem com “z” e tenham **dois ou mais** caracteres entre elas.

```
cp ../a??*z .
```

No exemplo acima, a expressão “??\*” representa dois ou mais caracteres, pois o “\*” pode ser substituído por nenhum caracter ou por mais caracteres. As duas interrogações (“??”) garantem que haverá no mínimo dois caracteres.

### 3.3.12. Comando grep (*Get Regular Expression*)

O comando grep pode ser usado para buscar uma sequência de caracteres dentro de um arquivo ou da entrada padrão. Sua sintaxe é:

```
grep expressao arquivo [opcoes]
```

em que “expressao” é a sequência de caracteres – ou a expressão regular – procurada, “arquivo” é o arquivo – ou o conjunto de arquivos definidos por curingas – em que a expressão será procurada.

Por exemplo, suponha que você tem dez arquivos-texto dentro de um diretório e que gostaria de saber em quais deles aparece a palavra “IFES”. Com o comando grep você pode descobrir em quais arquivos estão essa palavra e, de quebra, em qual ou quais linhas dos arquivos encontrados ela aparece. Por exemplo:

```
grep IFES *.txt
```

realiza uma busca pela palavra IFES em todos os arquivos “txt” do diretório corrente. Caso queira que o comando informe também em que linha está a ocorrência, digite a opção “-n” ao final da linha. Se a expressão possui várias palavras com espaço entre elas, você deverá digitar a expressão inteira entre aspas, para evitar que somente a primeira palavra seja entendida como expressão e as demais como nomes de arquivos.

```
grep "Instituto Federal do Espírito Santo" *.txt -n
```

Outro exemplo: suponha que você queira saber quais as linhas de um conjunto de programas em C que possuem a palavra “scanf” e deseja também que seja exibido o número das linhas e que a palavra procurada apareça em cores:

```
grep scanf *.c -n --color
```

O grep também aceita expressões regulares, que permitem referenciar textos de formas mais sofisticadas. No entanto, ensinar expressões regulares foge do escopo deste material.

### 3.3.13. Comando sort

Outro comando interessante é o comando sort, que permite ordenar as linhas de um determinado arquivo texto ou da entrada padrão. Sua sintaxe é:

```
sort [opcoes] [arquivo]
```

em que “opcoes” são as opções de ordenação e “arquivo” é o arquivo a ser ordenado. As opções mais comuns são:

Opção	Descrição
-r	Inverte o resultado da ordenação (decrescente).
-n	Ordena os números de forma numérica e não alfabética que é o padrão.
-c	Verifica se o arquivo está ordenado.

### 3.3.14. Comando head

O comando head exibe o início – as primeiras linhas – de um arquivo ou da entrada padrão. Também é possível especificar qual a quantidade de linhas a ser exibida. Sua sintaxe é:

```
head [opcoes] [arquivo]
```

Geralmente, a opção utilizada é a “-n” que indica quantas linhas do arquivo deverão ser exibidas. Exemplo:

```
head -n 20 abc
```

Nesse exemplo serão exibidas apenas as 20 primeiras linhas do arquivo abc. Caso a opção “-n” não seja especificada, serão exibidas 10 linhas, que é o número padrão.

### 3.3.15. Comando tail

Análogo ao comando head, porém exibe o final – as últimas linhas – de um arquivo ou da entrada padrão. É possível especificar qual a quantidade de linhas a ser exibida. Sua sintaxe é:

```
tail [opcoes] [arquivo]
```

Geralmente a opção utilizada é a “-n”, que indica quantas linhas do final do arquivo deverão ser exibidas. Exemplo:

```
tail -n 30 logs.txt
```

Nesse exemplo serão exibidas apenas as 30 últimas linhas do arquivo logs.txt. Caso a opção “-n” não seja especificada, serão exibidas 10 linhas, que é o número padrão. É possível também usar o comando tail para saltar (não exibir) as n-1 primeiras linhas, basta preceder o número com um sinal de “+” (mais). Por exemplo, o comando

```
tail -n +20 arq.txt
```

informa ao comando para exibir o arquivo a partir da linha 20, saltando as 19 primeiras.

### 3.3.16. Comandos more e less

Algumas vezes a saída de um comando ou programa é muito longa e rápida, e só é possível lê-la se conseguirmos fazer pausas na saída exibindo uma página por vez. Isso é chamado de paginação. Os comandos utilizados para fazerem paginação são os comandos more e less. Por exemplo, suponha que você quisesse ver o conteúdo do comando a seguir:

```
ls -l /etc
```

Provavelmente você não conseguiria ler devido ao grande volume de informação exibido na tela a uma velocidade bastante alta. Para paginar a saída do comando anterior, você deve redirecionar essa saída para a entrada do comando more, da seguinte forma:

```
ls -l /etc | more
```

Digite os dois e compare os resultados. Neste último comando, a “|” (barra vertical) serve para fazer um pipes entre programas, ou seja, redirecionar a saída de um programa para a entrada de outro. Esse programa pausa a saída do ls até que você pressione a barra de espaço (que salta uma página) ou enter (que salta uma linha). O único limitador desse comando é que ele não permite voltar a tela. Mas para isso existe um outro comando: less. O less é análogo ao more, só que, além de exibir de forma paginada, permite também voltar as páginas através da tecla <page-up>. Experimente digitar:

```
ls -l /etc | less
```

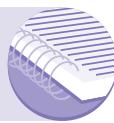
E compare a diferença ao avançar e retroceder nas páginas. Para sair a qualquer momento do comando less, digite “q”.

## Atividades



## Atividades

1. Consulte no Guia Foca Iniciante+Intermediário ([guiafoca.org](http://guiafoca.org)) o que é cada campo exibido através do comando “ls -l”.
2. Compare a saída da execução de dois comandos: “ls -l /bin/cp” e “ls -lh /bin/cp” e informe qual a diferença observada por você.
3. Abra um terminal e com ele recém-carregado execute os comandos para os itens abaixo:
  - a. descubra o diretório atual;
  - b. crie um diretório chamado “aula3”;
  - c. entre neste diretório recém-criado;
  - d. confirme que está nesse diretório, usando o comando que descobre o diretório atual para cada nome/sobrenomes seu, e crie um diretório diferente com um único comando. No meu caso, meu nome é “Sérgio Nery Simões” e criarei três diretórios: “sergio”, “nery” e “simoes”, tudo minúsculo e sem acentos; faça o mesmo com o seu nome;
  - e. apague os diretórios equivalentes ao seu nome do meio para que fiquem apenas dois diretórios; em meu caso apagarei o diretório “nery”;
  - f. agora entre no diretório correspondente ao seu primeiro nome;
  - g. confirme se está no diretório esperado;
  - h. dentro desse diretório crie onze arquivos: “a, b, c, ab, ac, bc, abc, acb, bac, cab, abcd”;
  - i. com os arquivos criados, liste apenas os arquivos com um único caracter no nome;
  - j. liste somente os arquivos cujos nomes possuem 3 caracteres;
  - k. liste apenas os arquivos cujos nomes começam com “a”;
  - l. liste somente os arquivos cujos nomes começam com “a” e terminam com “c”;
  - m. usando o item anterior, liste apenas os arquivos cujo nome possui 3 ou mais caracteres e que comecem com “a”;

**Atividades**

- n. agora copie todos os arquivos para o diretório equivalente ao seu sobrenome;
- o. sem sair do diretório atual, liste o diretório equivalente ao seu sobrenome e confirme que os arquivos estão lá;
- p. agora crie dois arquivos ocultos: “.oculto1” e “.arq2”
- q. liste o conteúdo do diretório normalmente e veja se os arquivos apareceram ou não;
- r. descubra qual a opção necessária para lista arquivos ocultos e liste novamente o diretório;
- s. agora, suba um nível na árvore de diretórios;
- t. exiba o diretório atual;
- u. remova o diretório equivalente ao seu sobrenome;
- v. Liste o diretório atual.

**Indicações**

Mazioli, Gleydson. **Guia Foca Linux.** ([www.guiafoca.org](http://www.guiafoca.org)). 2007.

Neves, Julio Cesar. **Programação Shell Linux.** 6.ed. Brasport, 2006.

GNU Bash Reference Manual (<http://www.network-theory.co.uk/docs/bashref/>)

Dicas, notícias e tutoriais sobre Linux (<http://br-linux.org/>)

Comunidade Linux no Brasil (<http://www.vivaolinux.com.br/>)

Apostilas gratuitas e documentação sobre o Linux (<http://www.dicas-l.com.br/>)

# EDIÇÃO E COMPILAÇÃO DE PROGRAMAS NO GNU/LINUX

Prezado aluno,

Neste capítulo você aprenderá como editar e compilar programas em C no GNU/Linux. Não se trata de um capítulo que ensina programação básica ou lógica de programação, e sim, de como utilizar as ferramentas fornecidas pelo sistema operacional GNU/Linux para editar e compilar seus programas em C.

Além disso, aprenderá comandos de manipulação de processos e envio de sinais. Por último, aprenderá a criar programas para configurar rotinas de tratamento de sinais na linguagem C.

Assumo que você já possui os conhecimentos necessários de programação da linguagem C – como suas estruturas básicas, comandos, etc – obtidos nas disciplinas de programação anteriores. Nos capítulos à frente, aprenderemos como fazer programas que manipulam processos e threads utilizando o GNU/Linux. Este capítulo servirá de base para isto.

Bom estudo!

angue risus at  
e velit at tellus.  
massa porttitor  
nsectetur magna.

Fala Professor

## 4.1. Introdução

Antes de começarmos a editar, compilar e executar programas em C, verifique se o seu sistema GNU/Linux possui os seguintes pacotes instalados: “gcc” (compilador C), “libc” (biblioteca C) e “gedit” (editor de programas) – dica: utilize o synaptic! Programadores mais experientes costumam preferir editores como o “vim” ou “emacs”, além do programa “make”, para controlar a compilação automática de programas. Se esse for o seu caso, sinta-se à vontade, mas este material será baseado no editor “gedit”, por ser de utilização mais simples. Após ter verificado se pelo menos os três primeiros pacotes estão instalados em seu sistema, vamos começar a edição de um programa teste, bem como a configuração do editor de programas.

## 4.2. Edição de Programas

Conforme comentando anteriormente, utilizaremos o programa gedit para editar os programas em C. Se desejar, você pode editar programas em diversas linguagens, graças ao recurso “modo de destaque” (visua-

lização dos comandos em cores), e também ao espaçamento automático, verificação de erros, etc. Para se ter uma pequena idéia das linguagens que podem ser editadas: C, C++, Java, Pascal, Fortran, Perl, PHP, Python, dentre muitas outras. Por último, o gedit também pode ser utilizado simplesmente para editar texto.

Para abrir o gedit, clique em “**Aplicativos=>Acessórios=>Editor de Textos**”. Uma janela semelhante à Figura 4-1 será aberta.

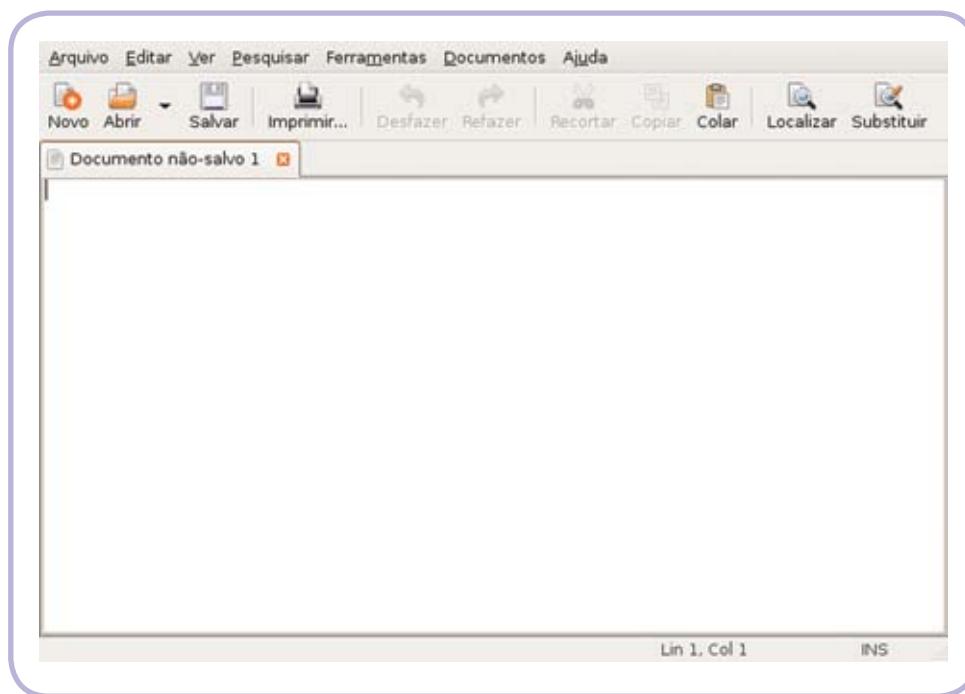


Figura 4-1: Tela inicial do editor Gedit

Agora vamos configurar o Gedit para tornar a programação mais confortável. Clique no menu Editar=>Preferências. Uma janela semelhante à Figura 4-2 é aberta. Essa janela possui várias abas. Na aba “Visão”, além das caixas que já estão habilitadas, habilite também a caixa “exibir número de linhas” e “destacar linha atual”, conforme a Figura 4-2.

Agora vamos configurar como o editor fará endentação do programa. Lembre-se de que usar tabulações para endentar o código é fortemente aconselhável, pois possibilita a configuração do espaçamento desejado do programa.

### Fala Professor



*Para quem não lembra o que é endentação, saiba que é um termo aplicado ao código fonte de um programa para indicar que os elementos hierarquicamente dispostos têm o mesmo avanço relativamente à posição inicial, ou seja, têm o mesmo espaçamento. Na linguagem C, a endentação tem um papel meramente estético, tornando a leitura do código fonte muito mais fácil.*



Figura 4-2: Configurando as preferências do gedit – aba visão

Para configurar como o gedit controla a endentação, clique na aba “Editor” e configure a “largura das tabulações” para 4 (quatro) e marque a caixa “habilitar recuo automático”, conforme a Figura 4-3. Se preferir, pode configurar a largura das tabulações para 3 (três).

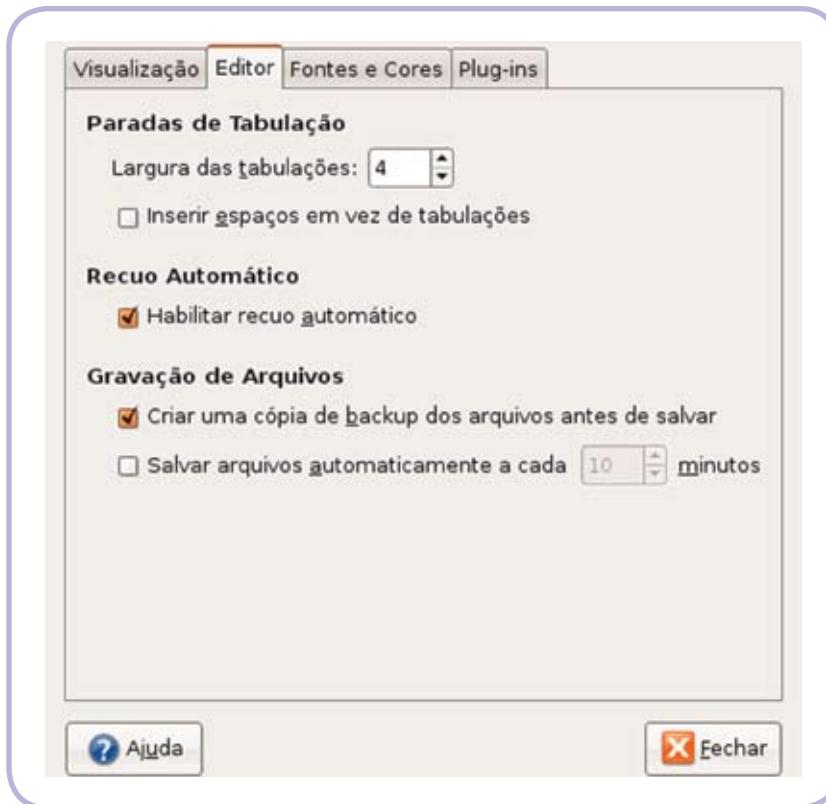


Figura 4-3: Configurando as preferências do gedit – indentação

Lembre-se de usar <tab> para indentar seu programa, ou seja, não utilize espaços pois, se utilizar, essa configuração não vai adiantar. Basicamente, a indentação consiste na adição de tabulações no início de cada linha na quantidade equivalente ao número de blocos em que cada linha está contida. Se quiser tornar a configuração do gedit ainda mais confortável, pode aumentar o tamanho da fonte para 12 ou 14, por exemplo, de acordo com sua preferência.

### Fala Professor



*Obs: algumas pessoas têm dor de cabeça quando estão programando, geralmente devido ao esforço pela utilização de fontes pequenas. Se esse for seu caso, recomendo que aumente o tamanho da fonte antes de começar a programar.*

Mas agora, chega de conversa e vamos editar um programa. Você deve estar pensando: “ufa, finalmente!”. Começaremos com um programa simples. Edite o programa a seguir:

```
1 #include<stdio.h>
2
3 int main ()
4 {
5     printf("Olá\n");
6     return 0;
7 }
```

Não se esqueça de endentar os programas com tabulações em vez de espaços. Após terminar a edição, salve o programa com o nome “prog1.c”. Em seguida, abra um terminal, crie um diretório chamado “SO2”, entre nele e compile este programa com o seguinte comando:

```
gcc prog1.c -o prog1
```

Veremos mais detalhes sobre a compilação na próxima seção. Agora vamos executar o programa. Para isso, digite no terminal de comandos (o cifrão representa meu *prompt* de comando, não digite-o):

```
$ ./prog1
Olá
```

Pronto, o programa funcionou!

Caso não tenha funcionado, verifique se você possui a biblioteca **libc** instalada. Lembre-se de que no GNU/Linux os executáveis não precisam ter extensões (.exe, .com, etc), eles são reconhecidos como executáveis devido a sua permissão de execução (aquele x que você vê quando executa o comando “ls -l”).

Angue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

## Fala Professor

Além disso, a menos que o programa esteja em um diretório registrado na variável de ambiente PATH, será necessário digitar o caminho completo do programa local para que ele possa ser executado – devido a questões de segurança. Eu poderia ter digitado “/home/sergio/SO2/prog1” para executar o programa anterior, mas como já vimos o “.” (ponto) representa o diretório corrente economizando minha digitação.

Agora vejamos em mais detalhes como compilar e executar um programa no GNU/Linux.

### 4.3. Compilação e Execução de Programas no GNU/Linux

Conforme mencionado anteriormente, o compilador que utilizaremos neste material será o GCC (GNU C Compiler). A partir de um ou mais códigos fontes de entrada, o GCC pode traduzi-los para um arquivo de saída. Um arquivo de saída pode ser um arquivo executável, arquivo objeto, arquivo de montagem ou um código C pré-processado.

É importante lembrar que para gerar um arquivo executável é necessário que, além da compilação (tradução do arquivo fonte em módulo-objeto), o GCC realize as ligações entre os módulos-objetos e as bibliotecas, gerando um arquivo executável. Em outras palavras, o GCC pode realizar a compilação e a ligação, esta última é uma etapa necessária para se ter um arquivo executável.

Angue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

## Fala Professor

O compilador GCC pode ser invocado a partir do comando gcc. Sua sintaxe simplificada é:

```
gcc [opcoes] fonte.c -o saida
```

em que “fonte.c” é um programa escrito em C e “saída” é o programa executável. Além disso, o GCC também possui diversas opções, possibilitando uma variedade de formas de compilações diferentes. Dentre as opções mais comuns estão:

Opção	Descrição
-o file	Coloca a saída no arquivo “file”. Aplica-se independentemente do tipo de saída que está sendo produzida, que pode ser um arquivo executável, um arquivo objeto, um arquivo de montagem ou um código C pré-processado.
-c	Compila ou monta os arquivos fontes, mas não realiza a sua ligação. A saída é na forma de um arquivo objeto para cada arquivo fonte. Por padrão o nome do arquivo objeto gerado é o nome da fonte substituindo sua extensão “.c”, “.i”, “.s” por “.o”.
-Wall	Habilita todas as Warnings (avisos) sobre construções consideradas questionáveis, e que são fáceis de prevenir.

Além dessas opções há inúmeras, algumas de otimização, outras referentes a compilação para outras arquiteturas, etc. Recomendo fortemente que sempre utilize a opção “-Wall”.

#### 4.3.1. Exibindo Warnings com a opção “-Wall”

Conforme visto anteriormente, a opção “-Wall” permite a visualização de todos os Warnings (avisos sobre possíveis erros no programa). Para entendermos melhor, vejamos um exemplo de programa com erro e a importância desta opção. Digite o programa a seguir

```

1 #include<stdio.h>
2
3 int main ()
4 {
5     int a, b;
6     a = 4;
7     b = 7;
8
9     printf("valor das variaveis antes do if: a=%d b=%d\n", a, b);
10    if (a = b)
11        printf("as variaveis são iguais:      a=%d b=%d\n", a, b);
12    else
13        printf("as variaveis são diferentes: a=%d b=%d\n", a, b);
14    return 0;
15 }

```

e salve-o com o nome “prog2\_if.c”. Há um erro de lógica nesse programa, mas a maioria das pessoas nem percebe. Agora compile-o com o comando:

```
$ gcc prog2_if.c -o prog2_if
```

O programa vai compilar normalmente. O que você acha que deveria ser impresso na tela? Vamos ver se você acertou. Execute o programa com o comando abaixo:

```
$ ./prog_if
valor das variaveis antes do if: a=4      b=7
as variaveis são iguais:      a=7      b=7
```

O certo seria o programa imprimir que as variáveis `a` e `b` são diferentes, no entanto o programa mostra que são iguais. Por quê? Ora, porque na linha 9 (no `if`), o programador provavelmente achou que estava **comparando** as variáveis “`a`” e “`b`” quando na verdade ele **atribuiu** o conteúdo de “`b`” para “`a`” e em seguida testou se “`a`” era diferente de zero. Assim, as variáveis se tornaram iguais. No entanto, isso não é um erro sintático, pois a linguagem C permite essa construção, atribuir um valor e em seguida testá-lo. Mas para acertar o programa, a expressão “`(a = b)`” deveria ser escrita com dois sinais de igual “`(a == b)`”, que é a forma correta para comparar valores.

*Agora imagine esse tipo de erro – provavelmente de digitação – em um programa com centenas de linhas de código. Seria difícil de encontrar, não é mesmo?! Felizmente o GCC possui uma forma de avisar possíveis erros como esse através da utilização da opção “-Wall”.*

*Angue risus at  
e velit at tellus.  
massa porttitor  
sectetur magna.*

### Fala Professor

Vamos recompilar o programa com essa opção e observar o que acontece:

```
$ gcc prog2_if.c -o prog2_if -Wall
prog_if.c: In function 'main':
prog_if.c:9: warning: suggest parentheses around assignment
used as truth value
```

Com esta opção, o programa emite um aviso referente à construção da linha 9 “`if (a = b)`”. Repare que mesmo assim o programa será compilado até o final, pois não há erro sintático. É como se o compilador lhe dissesse: “aquela construção está meio estranha, tem certeza de que é aquilo mesmo?”. E em grande maioria das vezes tais avisos são erros.

Assim, devido a este e outros tipos de erros semelhantes, recomendando fortemente que se utilize a opção “-Wall” ao compilar programas utilizando o `gcc`.



### Atenção

### Atividades

1. Como exercício, conserte o programa reescrevendo a linha 9 com dois sinais de igual “`(a == b)`”, execute-o novamente com a opção “-Wall” e verifique se é exibido mais algum aviso.



### Atividades

### 4.3.2. Exceções

Sabemos que podem ocorrer exceções durante a execução de alguns programas. Exceções são erros durante a execução de alguma instrução do programa e, quando ocorrem, o sistema operacional é chamado para tratá-los. Se a exceção pode afetar a segurança do sistema, o sistema operacional abortará a execução do programa e por isso são denominadas exceções fatais.

Atenção



Os tipos mais comuns de exceções de segurança são: (i) “tentativa de acesso ilegal à memória” e (ii) “tentativa de execução de uma instrução privilegiada no modo usuário”. No entanto, existem exceções como falta de página (*page fault*), que não impedem o fluxo normal de execução do programa. Também existem as exceções aritméticas (divisão por zero, *overflow*, *underflow*, etc), cujo tratamento pode ser configurado para abortar ou não o programa.

Veremos agora um exemplo de exceção de segurança: “tentativa de acesso ilegal à memória”. Digite o programa a seguir e salve-o com o nome “prog3\_acessoilegal.c”.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main (int argc, char **argv)
5 {
6     int i;
7     float vetor[10];
8     for(i=0; i<1000; i++)
9     {
10         vetor[i] = (float) i;
11         printf ("vetor[%d] = %.f\n", i, vetor[i]);
12     }
13     return 0;
14 }
```

Este programa declara um vetor de 10 posições e tenta acessar 1000 posições. Agora compile o programa com o seguinte comando:

```
$ gcc prog3_acessoilegal.c -o prog3_acessoilegal -Wall
```

e em seguida execute-o e observe sua saída com o comando a seguir – em negrito:

```
$ ./prog3_acessoilegal
vetor[0] = 0
vetor[1] = 1
vetor[2] = 2
vetor[3] = 3
vetor[4] = 4
vetor[5] = 5
vetor[6] = 6
vetor[7] = 7
vetor[8] = 8
vetor[9] = 9
Falha de segmentação
```

Esse erro é chamado de falha de segmentação (*segmentation fault*), pois você está tentando acessar uma posição da memória fora de seu segmento. Como a variável vetor foi declarada com 10 posições (de 0 a 9), ao se tentar acessar a décima primeira posição (posição 10, pois o vetor começa do zero) esse acesso estará fora do seu segmento e por isso será gerada a exceção.

*No entanto, lembre-se de que o sistema verifica apenas se você não está tentando acessar uma área de memória fora de seu segmento. Dentro de seus segmentos, não é realizada nenhuma verificação.*

*Naque risus at  
e velit at tellus.  
massa porttitor  
sectetur magna.*

Fala Professor

*Em outras palavras, se você tivesse outras variáveis declaradas após esta, é provável que o programa sobrescreveria essas variáveis antes de ser gerada uma exceção. Ou seja, você é responsável por acessar as variáveis dentro de seu próprio segmento e, se estiver cometendo erros durante o acesso, o sistema não irá informá-lo.*

Agora vejamos um outro exemplo de exceção. Digite o programa abaixo e salve-o com o nome “prog4\_estouropilha.c”.

```
1#include<stdio.h>
2#include<stdlib.h>
3
4void f()
5{
6    f();
7}
8
9int main ()
10{
11    f();
12    return 0; //esta linha não sera executada;
13}
```

Em seguida, compile e execute o programa conforme os comandos abaixo – em negrito – e observe o que acontece:

```
$ gcc prog4_estouro.c -o prog4_estouro -Wall
$ ./prog4_estouro
Falha de segmentação
```

Esse programa chama a função recursiva f() que não possui base da recursão, ou seja, sem um critério de parada definido. Isso faz com que a função seja chamada inúmeras vezes até que ocorra um estouro de pilha – a pilha não tem mais espaço para armazenar os endereços de retorno.

### Fala Professor



*Lembre-se de que, a cada vez que uma função é chamada, os endereços de retorno são salvos na pilha.*

Assim, o programa tentará salvar os endereços fora do segmento de pilha que pertence a ele, gerando uma falha de segmentação.

## 4.4.Pipes e Redirecionamentos

Nesta seção veremos como fazer, no GNU/Linux, os pipes (redirecionar a saída de um programa como entrada para outro) e o redirecionamento de entrada padrão a partir de um arquivo ou da saída padrão para um arquivo.

### 4.4.1.Pipes

Conforme visto na disciplina de Sistemas Operacionais I, pipes são um tipo de redirecionamento no qual a saída de um programa é redirecionada como entrada para outro. No GNU/Linux os pipes são representados por uma “|” (barra vertical). Por exemplo, a linha de comando:

```
./prog1 | ./prog2 | ./prog3 | ... | ./progN
```

representa que a saída do programa prog1 está redirecionada como entrada ao programa prog2. Este último, por sua vez, tem sua saída redirecionada ao programa prog3 e assim sucessivamente até chegar no programa progN. Isso indica que é possível realizar vários pipes em uma única linha de comando.

Além dos programas, os pipes também podem ser utilizados entre comandos (afinal, os comandos também são programas, certo?!). A partir de agora, usaremos os conceitos programas e comandos como sinônimos.

### Fala Professor

Nigue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Para exemplificar, execute a linha de comando abaixo:

```
ls /etc/ | grep conf | more
```

Nesse exemplo, utilizei 3 comandos: ls, grep e more. O primeiro lista o conteúdo do diretório “/etc”. Esta saída é repassada ao segundo comando, que filtra apenas as linhas que possuírem o termo “conf”. A saída desse segundo comando é repassada ao terceiro comando que exibe seu conteúdo de forma paginada. Tudo isto é possível através de **pipes**.

#### 4.4.2. Redirecionamento de Entrada

Vimos anteriormente que o pipe serve para fazer redirecionamento entre programas. Mas e se quiséssemos redirecionar a entrada padrão para ser lida a partir de um arquivo ou da saída padrão para ser escrita em um arquivo? Nesse caso, não usaríamos o pipe e sim o redirecionamento de entrada ou saída, como vamos ver agora.

### Fala Professor

Nigue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Redirecionar a entrada padrão significa que você, em vez de ficar digitando informações a partir do teclado, fará o programa ler a partir de um arquivo. Isso pode ser usado, por exemplo, para fazer testes em programas, mas serve pra muitas outras coisas também.

Suponha que você está desenvolvendo um trabalho de programação e toda vez que vai testá-lo, o programa pergunta informações sobre “nome, telefone, endereço” e é necessário fornecê-las manualmente (se você já fez algum tipo de trabalho parecido sabe como é chato ficar digitando informações para teste).

Assim, você pode deixar essas informações salvas num arquivo e, quando for executar o programa, basta redirecionar a entrada padrão a partir desse arquivo. O programa, em vez de esperar você digitar as informações a partir do teclado, vai lendo-as a partir do arquivo.

No GNU/Linux, para redirecionar a entrada padrão de um programa para ser lida a partir de um arquivo, basta digitar de forma semelhante ao comando abaixo:

```
./programa < arquivo_entrada
```

em que “programa” representa o programa cuja entrada será redirecionada e “arquivo\_entrada”, o arquivo de entrada que fornecerá as informações ao programa. O sinal de “<” (menor) faz com que o arquivo seja redirecionado como entrada padrão ao programa. Dessa forma, você pode realizar dezenas de testes em seus programas sem a necessidade de ficar digitando as informações manualmente. Agora vamos ver como redirecionar a saída.

#### 4.4.3. Redirecionamento de Saída

Analogamente ao redirecionamento de entrada, o redirecionamento de saída padrão serve para capturar o que é impresso na tela por um comando ou programa e enviar para um arquivo. Em vez de impressas na tela, as informações são escritas em um arquivo. No GNU/Linux, para redirecionar a saída padrão de um programa para ser escrita em um arquivo, basta digitar de forma semelhante ao comando abaixo:

```
./programa > arquivo_saida
```

em que “programa” representa o programa cuja saída será redirecionada e “arquivo\_saida” o arquivo de saída para o qual as informações serão direcionadas. O sinal de “>” (maior) faz com que a saída padrão do programa seja redirecionada para o arquivo. Se o arquivo não existe, será criado; mas, se já existir um arquivo, seu conteúdo será primeiramente apagado para em seguida começar a receber os dados de saída.

Também é possível redirecionar a saída para um arquivo pré-existente sem que este tenha seu conteúdo apagado inicialmente. Para isto, use o seguinte comando:

```
./programa >> arquivo_saida2
```

Ao invés de um sinal de “>” agora temos dois (“>>”). Com isso, o conteúdo prévio do arquivo não será apagado e as informações redirecionadas serão concatenadas ao final do arquivo pré-existente. Caso o arquivo não exista, será criado.

Por fim, é possível combinar os redirecionamentos de entrada, saída e pipes. Observe os próximos exemplos:

```
./programa < arquivo_entrada > arquivo_saida
```

Neste caso, a entrada do programa é proveniente do arquivo “arquivo\_entrada” e sua saída é salva no arquivo “arquivo\_saida”. Como último exemplo, vamos usar tudo ao mesmo tempo: redirecionamento de entrada e saída e pipes. Observe a construção a seguir:

```
./prog1 < entrada | prog2 | prog3 > saida
```

Nesse exemplo, o programa prog1 tem sua entrada redirecionada a partir do arquivo “entrada” e sua saída é redirecionada a outro programa (prog2) através de um pipe. Após isso o programa prog2 tem sua saída redirecionada ao programa prog3. Este último, por sua vez, redireciona sua saída para o arquivo “saída”.

## 4.5. Gerenciamento de Processos no GNU/Linux

O GNU/Linux possui alguns comandos específicos para gerenciar processos. Através desses comandos é possível verificar os processos ativos, descobrindo informações a respeito deles (como PID, utilização do processador, memória, arquivos, swap, etc.). Também possui comandos para enviar sinais aos processos a fim de interrompê-los ou até mesmo abortar sua execução – caso tenha permissão pra isso.

### 4.5.1. Comando ps

O comando ps exibe informações sobre o estado corrente dos processos. Ele pode ser utilizado para descobrir o PID (Process ID) de um processo, a taxa de utilização da CPU naquele instante pelo processo, a quantidade de memória utilizada, a qual usuário o processo está associado, a hora em que o processo foi iniciado e várias outras informações. Sua sintaxe é:

```
ps [opções]
```

Lembre-se de que o comando ps exibe as informações sobre o estado dos processos no momento em que é executado. Em outras palavras, trata-se de uma informação sobre um instante de tempo, como se fosse uma “foto” do estado dos processos naquele instante. Após isso, o estado dos processos continuará mudando. As opções mais comuns do comando ps são:

Opção	Descrição
a	Exibe os processos criados por usuários do sistema.
x	Mostra processos que não são controlados pelo terminal.
u	Exibe o nome de usuário que iniciou o processo e hora em que o processo foi iniciado.
m	Mostra a memória ocupada por cada processo em execução.

Para obter mais informações sobre o comando ps, digite “**man ps**”.

**Atividades****Atividades**

2. Digite o comando `ps` em suas formas variadas – conforme os exemplos a seguir – e compare as saídas apresentadas (Lembre-se de que para sair do comando `less` basta digitar `q`. No último exemplo, não esqueça de trocar o nome “`sergio`” pelo seu login.) Exemplos:

- a. `ps`
- b. `ps a`
- c. `ps x | less`
- d. `ps ax | less`
- e. `ps aux | less`
- f. `ps aux | grep sergio`

3. Por último, você pode também visualizar a árvore de processos e suas interdependências através do comando “`pstree`”. Execute esse comando e comente sua saída (para mais informações sobre esse comando, digite “`man pstree`”).

Agora vamos dar uma olhada no comando `top`, que exibe os processos com maior taxa de utilização da CPU e da memória.

#### 4.5.2.Comando top

Este comando mostra os processos ativos ou parados, taxa de utilização da CPU, quantidade de memória, tempo de CPU, e detalhes sobre o uso da memória RAM, Swap, disponibilidade para execução de programas no sistema, etc.

**Fala Professor**

*Os processos são ordenados pelo comando `top` dos mais pesados para os mais leves, ou seja, quanto maior a utilização de CPU e de memória, maior a chance do processo ser exibido entre os primeiros. Para executá-lo, basta digitar “`top`”.*

Uma vez chamado, o programa top fica continuamente em execução, exibindo os processos que estão sendo executados em seu computador e os recursos utilizados por eles. Se quiser atualizar imediatamente a tela, pressione a barra de espaço. Para obter ajuda, pressione “h”. Para sair do top, pressione a tecla “q”.

A Figura 4-4 apresenta a saída do comando top.

```

Arquivo Editar Ver Terminal Abas Ajuda
top - 15:42:52 up 53 min, 2 users, load average: 0.22, 0.14, 0.17
Tasks: 115 total, 1 running, 114 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.5%us, 0.3%sy, 0.0%ni, 99.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 2066180k total, 1142480k used, 923700k free, 94724k buffers
Swap: 1052248k total, 0k used, 1052248k free, 572152k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
5880 cead 20 0 28676 5832 3384 S 1 0.3 0:02.47 pulseaudio
5973 cead 20 0 24528 17m 6884 S 1 0.8 0:19.62 compiz.real
5489 root 20 0 424m 77m 8248 S 0 3.9 2:01.88 Xorg
  1 root 20 0 2844 1692 544 S 0 0.1 0:01.40 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 0:00.04 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
  7 root 15 -5 0 0 0 S 0 0.0 0:00.02 ksoftirqd/1
  8 root RT -5 0 0 0 S 0 0.0 0:00.00 watchdog/1
  9 root 15 -5 0 0 0 S 0 0.0 0:00.02 events/0
 10 root 15 -5 0 0 0 S 0 0.0 0:00.00 events/1
 11 root 15 -5 0 0 0 S 0 0.0 0:00.00 khelper
 47 root 15 -5 0 0 0 S 0 0.0 0:00.04 kblockd/0
 48 root 15 -5 0 0 0 S 0 0.0 0:00.00 kblockd/1
 51 root 15 -5 0 0 0 S 0 0.0 0:00.00 kacpid

```

Figura 4-4: Utilizando o comando “top” para visualizar os Processos

Como é possível observar na Figura 4-4, há um total de 116 processos no sistema, e desses, 2 estão em execução, 114 dormindo, nenhum parado e nenhum processo no estado zumbi. Observa-se também que 46,2% da CPU está ocupada por processos de usuário (us – user) e 8,6% ocupada pelo sistema (sy – system). O top também apresenta a quantidade de memória total (514332KB), usada (468296KB) e livre (46036KB), bem como a quantidade de memória utilizada para buffers e swap. O processo com maior ocupação é o firefox, com 29.3% de utilização da CPU e 21.3% de ocupação da memória, seu PID é 6315, sua prioridade é 20 e o login do usuário dono do processo é “sergio”. Existem outras informações que não destaquei aqui, mas as mais importantes são essas.

### Atividades

4. Forneça as informações sobre os processos “Xorg” e “gnome-panel” apresentados na Figura 4-4.
  
5. Execute o comando top em sua máquina e dê informações sobre o processo mais pesado de seu sistema.



### Atividades

## Atividades



6. Com o comando **top** em execução, pressione a tecla **h** (help) e descubra qual tecla apertar para:
- alterar o intervalo de atualização
  - matar um processo (kill)
  - exibir apenas processos de um usuário específico

#### 4.5.3. Interrompendo a execução de um processo

Para interromper (finalizar) a execução de um processo que está rodando em primeiro plano (*foreground*) basta pressionar as teclas <CTRL>+<c>. O processo será finalizado e será exibido o prompt de comando. Você também pode finalizar um processo através do comando **kill**.

#### 4.5.4. Parando momentaneamente a execução de um processo

Para parar momentaneamente a execução de um processo rodando em primeiro plano, basta pressionar as teclas <CTRL>+<z>. O processo será pausado e será mostrado o número de seu job e o prompt de comando. Caso deseje retornar à execução de um processo pausado, use o comando **fg** (*foreground*) ou o **bg** (*background*).

## Fala Professor



*Lembre-se de que o programa permanece na memória no ponto em que parou quando ele foi interrompido. Você pode usar outros comandos ou rodar outros programas enquanto o programa atual está interrompido.*

#### 4.5.5. Enviando sinais com o comando kill

O comando **kill**, apesar do nome, serve para **enviar sinais para processos**. E, dentre esses sinais, existem alguns para finalizar a execução de processos. Sua sintaxe é:

```
kill [opcoes] [sinal] [pid]
```

em que “pid” é o número de identificação do processo obtido a partir dos comandos ps ou top, “sinal” é o sinal a ser enviado ao processo e “opções” são as opções desejadas. Para saber os sinais que você pode enviar a um processo, digite o comando kill seguido da opção “-l”. Você obterá uma saída semelhante à do exemplo que segue:

```
$ kill -l
 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) SIGNALRM   15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU    23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM  27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

O sinal 2 (SIGINT), por exemplo, é o sinal enviado pelo sistema operacional após você pressionar <CTRL>+<c> para abortar o processo, lembra?! Já o sinal 11 (SIGSEGV) o sistema operacional envia quando ocorre uma falha de segmentação. Outro exemplo de envio de sinal ocorre após pressionar <CTRL>+<z>, pois o sistema enviará o sinal 19 (SIGSTOP) ao processo, que fará com que ele pare momentaneamente, podendo voltar à execução através dos comandos fg ou bg.

Mas você também pode enviar um sinal manualmente, para isso, deve colocar um “-” (traço) antes do número do sinal (exemplo “-9”, “-15”, etc). Na maioria das vezes, você necessitará utilizar sinais para finalizar um processo, mas pode enviar qualquer um exibido pelo comando “kill -l”.

*Os sinais mais comuns que podem ser enviados a um processo para finalizá-lo são os sinais “-15” (SIGTERM) e o “-9” (SIGKILL). O primeiro (sinal “-15” ou SIGTERM), permite que o processo encerre seus arquivos antes de ser finalizado. Seria uma forma “educada” de encerrar (ou matar) um processo. O outro sinal (-9 ou SIGKILL) já não é tão “educado” assim, pois ele finaliza um processo imediatamente, sem dar chances a ele de salvar seus dados. Normalmente este sinal é usado apenas para matar processos que estão travados.*

*ague risus at  
e velit at tellus.  
massa porttitor  
sectetur magna.*

Fala Professor

Na prática, você pode programar a rotina de tratamento de todos os sinais, exceto o sinal “-9” ou SIGKILL, que força o encerramento do processo. Procure tomar cuidado antes de finalizar processos para evitar finalizar algum processo seu e perder dados importantes. Caso isto aconteça, talvez seja necessário reiniciar a máquina.

Agora vamos fazer alguns testes com o comando kill para matar processos. Para fazer este teste vamos abrir dois terminais. Após abrir um terminal normalmente, abra outro pressionando <CTRL>+<SHIFT>+<n> ou de outra forma que preferir. Agora que você tem dois terminais abertos, vá ao primeiro terminal e digite o comando:

```
cat
```

Digite somente isso, e o comando ficará aguardando a entrada de dados do teclado. Agora vá ao segundo terminal e digite apenas o comando “ps u” que está em negrito e você terá uma saída semelhante à que segue:

```
$ ps u
USER     PID      %CPU      %MEM     VSZ      RSS      TTY      STAT      START      TIME      COMMAND
sergio   16785    0.0      0.3      4704    1890      pts/1      Ss      23:21      0:00      bash
sergio   16808    0.0      0.3      4716    1996      pts/0      Ss      23:22      0:00      bash
sergio   16899    0.0      0.1      3000     596      pts/1      S+      23:29      0:00      cat
sergio   16903    0.0      0.1      2744    1008      pts/0      R+      23:29      0:00      ps u
```

Este comando informa que tenho dois terminais “bash” abertos (em **pts/0** e **pts/1**, que são códigos que identificam a que terminal o processo pertence) e um processo “cat” cujo PID é de 16899, no meu caso, mas que, em seu computador, provavelmente terá códigos diferentes. Vamos matar este processo com o código “-15” ou SIGTERM. Ainda no segundo terminal, digite o comando a seguir e observe (obviamente, troque o PID 16899 pelo PID do processo “cat” apresentado em **seu computador**):

```
$ kill -15 16899
```

Nada parece acontecer, pois no segundo terminal não aparece nenhuma mensagem. No entanto, observe agora o que aconteceu com o primeiro terminal e verá algo parecido com o que segue:

```
$ cat
Finalizado
```

O processo originado pelo comando cat no primeiro terminal foi finalizado a partir do segundo através do envio de um sinal (“-15” ou SIGTERM) pelo kill. Experimente agora finalizar um dos programas bash – algum terminal aberto – de seu computador. Qual a linha de comando que você digitou?

*Curiosidade: além do comando kill há um outro comando chamado killall, que serve para finalizar processos pelo nome em vez de usar o PID.*

Fala Professor

Angue risus ac  
ne velit at tellus.  
massa porttitor  
sectetur magna.

Assim, se você tiver vários processos que foram iniciados pelo comando “programa1”, por exemplo, bastaria digitar o comando

**killall programa1**

para matar todos os processos com o nome “programa1”.

#### 4.5.6. Programando a rotina de tratamento de sinais

Agora vamos ver um exemplo de como capturar os sinais para tratá-los. Digite o programa abaixo e salve-o com o nome “prog5\_tratasinais.c”.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <unistd.h>
5
6 void tratasinal(int sinal)
7 {
8     printf("Sinal recebido: %d.\n", sinal);
9 }
10
11 int main()
12 {
13     signal(1,tratasinal);
14     signal(2,tratasinal);
15     signal(3,tratasinal);
16     signal(4,tratasinal);
17     signal(5,tratasinal);
18     signal(6,tratasinal);
19     signal(7,tratasinal);
20     signal(8,tratasinal);
21     signal(9,tratasinal);
22     signal(10,tratasinal);
23     signal(11,tratasinal);
24     signal(12,tratasinal);
25     signal(13,tratasinal);
26     signal(14,tratasinal);
27     signal(15,tratasinal);
28     printf("Vou entrar em loop infinito.\n");
29     for(;;)
30         sleep(1);
31     return 0; /* < - - esta linha nao sera executada */
32 }
```

Não esqueça dos “*includes*” das bibliotecas `<signal.h>` e `<unistd.h>` (linhas 3 e 4). A função “`signal`” (linhas 13 a 27) recebe como primeiro argumento o número do sinal a ser configurado e como segundo, o nome da rotina de tratamento do respectivo sinal. Dessa forma, o programa configura o recebimento de todos os sinais de 1 a 15 para serem tratados pelo procedimento “`tratasinal`” (que será a nossa rotina de tratamento e neste exemplo fará apenas uma impressão na tela). Mas cada sinal poderia ser configurado para ser tratado por uma rotina diferente. Neste exemplo, a rotina de tratamento apenas imprime na tela o número do sinal recebido. Após digitado e salvo, compile-o com o comando:

```
$ gcc prog5_tratasinais.c -o prog5_tratasinais -Wall
```

Para testá-lo, execute o programa com o comando abaixo:

```
$ ./prog5_tratasinais
```

E observe que o programa não será mais encerrado quando você pressionar <CTRL>+<c>. Em vez disso, o programa exibirá que recebeu o sinal 2. Agora vamos enviar sinais para esse programa através do comando kill.

## Atividades



### Atividades

7. Abra um terminal e execute o programa “**prog5\_tratasinais**”, visto anteriormente. Com esse programa funcionando, abra um **novo terminal** e:
  - a. descubra o PID do programa “**prog5\_tratasinais**”;
  - b. em seguida, envie sinais de 1 a 15 para esse programa e observe a saída;
  - c. houve algum sinal que não foi capturado pelo programa??? Se sim, qual e por quê?
  - d. Altere esse programa para que ele exiba também o **nome** de cada sinal recebido. Por exemplo, caso receba o sinal 2 deverá imprimir: “Sinal recebido: 2 (SIGINT)”. Dica: para facilitar, utilize a construção “switch” da linguagem C.

Para obter mais informações, consulte:

## Indicações



Ribeiro, Uirá. **Sistemas Distribuídos – Desenvolvendo Aplicações de Alta Performance no Linux**. 1.ed. Axcel, 2005.

Mazioli, Gleydson. **Guia Foca Linux**. ([www.guiafoca.org](http://www.guiafoca.org)). 2007.

Neves, Julio Cesar. **Programação Shell Linux**. 6.ed. Brasport, 2006.

The CTDP Linux Programmer’s (<http://www.comptechdoc.org/os/linux/programming/>)

Goldt, Sven. **The Linux Programmer’s Guide**. Version 0.4, 1995



# MANIPULAÇÃO DE PROCESSOS NO GNU/LINUX

Prezado aluno,

Neste capítulo você aprenderá como fazer programas que manipulam processos no GNU/Linux. Começaremos por algumas revisões sobre o conceito de processos, veremos como um processo pode descobrir seus próprios PID e GID, aprenderemos como criar um novo processo utilizando a primitiva fork e finalmente teceremos algumas considerações relacionadas a esses conceitos. Procure fazer todos os programas e testá-los mais de uma vez e, se possível, fazer algumas alterações e ver como eles se comportam.

Bom estudo!

Angue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

## 5.1. Introdução

A grande vantagem dos sistemas multiprogramáveis é que neles os processos são executados concorrentemente, compartilhando, entre outros recursos, a utilização do processador, da memória principal e dos dispositivos de E/S. Além disso, em sistemas com múltiplos processadores, não só existe a concorrência de processos pelo uso do processador, como também a execução simultânea de processos nos diferentes processadores.

Recorde que processo é um programa em execução juntamente com seu contexto e suas áreas de dados, código e pilha.

Angue risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

O contexto de um processo pode ser dividido em duas partes: o contexto de hardware e o contexto de software. O contexto de hardware armazena o conteúdo de todos os registradores. Já o contexto de software armazena informações sobre a identificação, as quotas e os privilégios do processo. Dentre as informações de identificação do processo, destacam-se: PID, nome do processo, nome do usuário, grupo do processo. Conforme vimos no capítulo anterior, no GNU/Linux tais informações podem ser obtidas através dos comandos top e ps.

## Fala Professor

Angus virus ac  
ne velit at tellus.  
massa portitor  
insectetur magna.

Geralmente, cada processo possui suas próprias áreas de dados, código e pilha, que são armazenadas em segmentos de memória. Dessa forma, normalmente teremos os segmentos de dados, código e pilha.

Como já vimos, se um processo tentar acessar uma posição de memória fora de seus segmentos, será gerada uma exceção de segurança, o sistema operacional será chamado e enviará um sinal ao processo para que este seja finalizado.

No GNU/Linux, o primeiro processo a iniciar é o *init* e ele possui PID igual a 1. Com exceção deste processo, todos os processos no GNU/Linux devem estar associados a um processo “pai”. Caso o processo pai termine antes do processo filho, o filho finalizará também ou poderá se tornar filho do processo *init* e continuar executando – depende da configuração do sistema.

### 5.2. Identificação de um processo

Como vimos, todo processo é identificado por um número exclusivo (PID) fornecido pelo sistema operacional. É como se fosse a carteira de identidade do processo, ou seja, não existem dois processos diferentes com o mesmo PID no mesmo instante. Pode até ser que, após a finalização de um processo, outro venha ter o mesmo PID, mas no mesmo instante não pode haver dois processos com o mesmo PID.

Agora vejamos algumas primitivas na linguagem C que permitem descobrir o PID do processo, o PID do pai de um processo (PPID) e o número que identifica o grupo do processo (GID). As primitivas que possuem essa informação encontram-se na biblioteca <unistd.h> e estão descritas a seguir.

```
pid_t getpid()
```

Retorna o número PID do processo em execução.

Agora, observe a próxima primitiva:

```
pid_t getppid()
```

Esta função retorna o número PID do processo pai (*parent pid*), ou seja, do processo que criou o processo em execução através de outras primitivas, como, por exemplo, *fork()*, *system()* ou *exec()*.

E por fim, a última primitiva antes de nosso primeiro programa:

```
pid_t getpgrp()
```

Esta função retorna o ID do grupo do processo. Os grupos de processos são usados para distribuição de sinais entre os processos relacionados. Observe agora um programa exemplo com essas primitivas. Digite o programa a seguir e salve-o com o nome “identificacao.c”.

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     printf("Eu sou o processo de PID %d\n", getpid());
7     printf("Meu pai tem PID %d\n", getpid());
8     printf("Pertenço ao grupo de ID %d\n", getpgrp());
9     return 0;
10 }
```

Agora compile o programa com o comando:

```
$ gcc identificacao.c -o identificacao -Wall
```

Em seguida, execute-o com o comando:

```
$ ./identificacao
Eu sou o processo de PID 20259
Meu pai tem PID 19907
Pertenço ao grupo de ID 20259
```

O processo exibiu seu próprio PID, o PID do seu processo pai e o ID de seu grupo. Agora execute novamente o programa e observe o resultado:

```
$ ./identificacao
Eu sou o processo de PID 20304
Meu pai tem PID 19907
Pertenço ao grupo de ID 20304
```

Note que dessa vez o PID e o ID do processo foram diferentes e provavelmente serão diferentes mais uma vez se o programa for executado novamente, pois o sistema operacional fornece um número de identificação a cada execução e não é possível prever qual o número PID que o processo receberá a cada execução. No entanto, o PID do pai do processo foi o mesmo, porque neste caso o pai do processo principal é o terminal (bash) e, como o programa foi executado duas vezes no mesmo terminal, cada execução ficou associada ao mesmo processo pai (neste

caso o bash com PID 19907, mas no seu computador, provavelmente terá outro PID). Para se certificar disso, basta digitar um ps e observar o número PID do terminal (bash).

<b>\$ ps</b>			
PID	TTY	TIME	CMD
19907	pts/0	00:00:00	bash
20347	pts/0	00:00:00	ps

### 5.3.Criação de processos

Para criar processos em GNU/Linux é necessário utilizar a primitiva *fork()* que, juntamente com primitiva *wait()*, foi a primeira notação de linguagem para especificar concorrência. Grosseiramente falando, a primitiva *fork()* cria um novo processo, ao passo que a *wait()* aguarda a finalização do processo criado para desalocar seus recursos. Em geral, para cada *fork* no programa é necessário um *wait*. A sinopse da primitiva *fork* é apresentada abaixo. Essa primitiva pertence à biblioteca <unistd.h> mas é necessário incluir também a biblioteca <sys/types.h>.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

O processo criado (filho) é uma cópia exata do processo pai, excetuando-se informações de identificação como PID e PPID (parent PID). Além disso, o valor retornado pela função *fork()* é 0 (zero) para o processo filho e é o PID do filho para o processo pai. Através desse valor de retorno, é possível diferenciar se o processo é o pai ou o filho e selecionar códigos diferentes para eles executarem.

#### Fala Professor



*Cada processo tem o seu próprio espaço de endereçamento, com cópias de todas as variáveis, que são independentes em relação às variáveis de outro processo. Após a criação, ambos os processos (pai e filho) executam a instrução seguinte à primitiva fork.*

A sincronização entre os processos pai e filho é feita com a primitiva *wait()*, que bloqueia o processo pai, que a executa, até que um processo filho termine. Em seguida, a primitiva *wait* é responsável por desalocar os recursos alocados para o filho, como área de dados, código, pilha e PCB. A sinopse da primitiva *wait* é dada a seguir:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Agora vamos fazer um programa exemplo, que cria um processo filho através da primitiva fork. Digite o programa a seguir e salve-o com o nome “fork\_simple.c”.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <wait.h>
4
5 int main()
6 {
7     int id;
8     id = fork();
9     if (id == 0) /* processo filho */
10    {
11         printf("\tFilho: (pid=%d) e espero 10 segundos.\n", getpid());
12         sleep(10);
13         printf("\tFilho: Ja esperei e vou embora...\n");
14     }
15     else
16    {
17         printf("Pai: (pid=%d) e espero pelo meu filho=%d\n", getpid(), id());
18         wait(NULL);
19         printf("Pai: Meu filho terminou. Vou terminar tambem!\n");
20     }
21     return 0;
22};
```

Para compilá-lo, utilize o comando:

```
$ gcc fork_simple.c -o fork_simple -Wall
```

Em seguida, execute-o e observe o resultado:

```
$ ./fork_simple
Filho: (pid=8977) e espero 10 segundos.
Pai: (pid=8976) e espero pelo meu filho=8977
Filho: Ja esperei e vou embora...
Pai: Meu filho terminou. Vou terminar tambem!
```

Na linha 8 desse programa, o processo pai (PID=8976) cria um processo filho (PID=8977). A partir da linha 9, a execução passa a ser feita por ambos os processos. No entanto, o processo filho terá a variável “id” igual a zero – que é o retorno padrão da função fork para processos filhos. Já o pai terá a variável “id” com o valor diferente de zero e valerá o PID do processo filho. Assim, na linha 9, o processo filho entra no bloco de código do “if” (linhas 10 a 14) e o pai entra no bloco do else (linhas 16 a 20).

*A primitiva sleep(n), ao ser chamada, faz o processo aguardar n segundos.*

ague risus ac  
e velit at tellus.  
massa portitor  
asectetur magna.

Fala Professor

O filho aguarda 10 segundos (sleep na linha 12) e sai, enquanto o pai fica aguardando pelo filho (devido à função wait na linha 18) para poder sair também. Após o filho terminar, um sinal de aviso é enviado ao pai que o permite sair da função wait e terminar também.

### Fala Professor

Angue risus at  
ne velit at tellus  
massa porttitor  
nsectetur magna.

*Como vimos nesse exemplo, após o processo pai criar o processo filho, ambos os processos utilizam o retorno da função fork – neste caso armazenado na variável id – para selecionar códigos diferentes para execução.*

Agora vamos fazer um programa no qual o processo filho termina, mas o processo pai não executa wait, deixando seus recursos alocados. Nesse caso, chamamos o processo filho de processo zumbi, pois ele já encerrou, mas continua com recursos alocados do sistema. Digite o programa abaixo e salve-o com o nome “zumbi.c”.

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4
5 int main()
6{
7    int id;
8    printf("PAI: (pid=%d) vou criar um processo filho...\n", getpid());
9    id = fork();
10   if (id == 0) /* filho */
11   {
12       printf("\tFilho: (pid=%d) vou aguardar 5 segundos.\n", getpid());
13       sleep(5);
14       printf("\tFilho: vou terminar...\n");
15   }
16   else
17   {
18       printf("PAI: não vou executar wait....\n");
19       for(;;) /* loop infinito */
20       {
21       }
22       wait(0); /* esta linha não será executada */
23   }
24   return 0;
25 }
```

Agora compile-o com o comando:

```
$ gcc zumbi.c -o zumbi -Wall
```

Em seguida, execute-o e observe o resultado.

```

$ ./zumbi
PAI: (pid=9240) vou criar um processo filho...
      FILHO: (pid=9241) vou aguardar 5 segundos...
PAI: não vou executar wait...
      FILHO: vou terminar...
```

## Manipulação de Processos no GNU/Linux

O filho termina, mas o pai não. Devido ao loop infinito nas linhas 19, o pai nunca executará o wait da linha 22, que liberaria os recursos alocados pelo filho. Assim, o processo filho é chamado de zumbi, pois já terminou, mas continua ocupando recursos do sistema. Para confirmar isso, abra um novo terminal e digite o comando em negrito (ps u) a seguir:

```
$ ps u
USER     PID   %CPU   %MEM   VSZ    RSS   TTY   STAT   START   TIME   COMMAND
sergio  7937   0.0    0.3   4724   2048  pts/0  Ss+   03:48  0:00  bash
sergio  9640   66.2   0.0   1628   376   pts/0  R      05:18  0:15  ./zumbi
sergio  9641   0.0    0.0     0     0   pts/0  Z      05:18  0:00  [zum]
                                         <defunct>
sergio  9647   0.0    0.3   4716   1996  pts/1  Rs     05:18  0:00  bash
sergio  9650   0.0    0.1   2744   1004  pts/1  R+    05:18  0:00  ps u
```

Observe o programa com PID 9641: ao lado do nome dele há palavra “**defunct**”, indicando que ele não está mais sendo utilizado. Outra forma de ver isso é através do comando top, digite top nesse segundo terminal e observe algo semelhante à Figura 5-1.

Figura 5-1: Um processo zumbi

Observe no final da segunda linha que o comando top informa que há um processo “zombie” (zumbi em inglês). Agora envie um sinal “-15” (SIGTERM) para terminar esse processo. Por exemplo, no caso da Figura 5-1, bastaria digitar:

```
$ kill -15 9640
```

Em seguida verifique novamente com o comando top se o processo foi encerrado.

Agora vejamos outro exemplo, no qual o processo pai é finalizado, mas o filho continua sendo executado. Digite o programa que segue e salve-o com o nome “orfao.c”.

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5{
6    int id;
7    printf("PAI: meu pid=%d. Vou criar um processo filho.\n", getpid());
8    id = fork();
9    if (id == 0) /* processo filho */
10    {
11        printf("\tFILHO: criado com pid=%d.\n", getpid());
12        printf("\tFILHO: Vou ficar em loop infinito...\n");
13        for(;;);
14    }
15    else /* processo pai */
16    {
17        sleep(5);
18        printf("PAI: (pid=%d) vou deixar o meu filho orfao.\n");
19    }
20    return 0;
21};

```

Agora compile-o com o comando:

```
$ gcc orfao.c -o orfao -Wall
```

Em seguida, execute-o conforme abaixo e observe o resultado.

```
$ ./orfao
PAI: meu pid=10092. Vou criar um processo filho.
      FILHO: criado com pid=10093.
      FILHO: Vou ficar em loop infinito...
PAI: (pid=10092) vou deixar meu filho orfao.
```

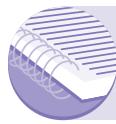
Neste exemplo, o processo filho entra em loop infinito (linha 12) e por isso não termina. O processo pai aguarda 5 segundos (linha 16) e em seguida sai terminando sua execução e deixando o processo filho órfão. Após isso, você pode verificar com o comando ps o processo filho (PID=10093) sendo executado:

\$ ps			
PID	TTY	TIME	CMD
7937	pts/0	00:00:00	bash
10093	pts/0	00:02:52	orfao
10154	pts/0	00:00:00	ps

Se preferir, você pode encerrar o programa orfao com killall.

```
$ killall orfao
```

## Atividades



## Atividades

8. Observe o programa a seguir e responda as perguntas.

```

1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     int i;
7     for(i=0; i<3; i++)
8     {
9         fork();
10    }
11    printf("\tMeu pid=%d.\n", getpid());
12    return 0;
13 };

```

a. Esse programa possui um único procedimento “printf” (linha 10). Quantas linhas serão impressas na tela?

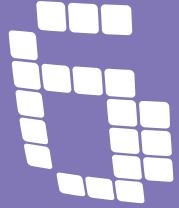
b. E se o número 3 (linha 6) for alterado para 4, quantas linhas serão impressas? Por quê?

9. Descubra, através de manuais na internet, para que serve a primitiva *waitpid()* e como utilizá-la.

10. Faça um programa em que o processo principal crie três filhos. Cada filho deve imprimir seu PID e em seguida aguardar 5 segundos antes de finalizar. O processo principal deve finalizar somente após todos os filhos finalizarem.

*11. Faça um programa em que o processo principal crie um filho, esse filho crie outro filho – neste caso neto do processo principal. O processo neto deve imprimir seu PID e aguardar 5 segundos antes de finalizar. O processo filho também deve imprimir seu PID aguardar o processo neto para finalizar. Por sua vez, o processo pai deve imprimir seu PID e aguardar o processo filho para finalizar.*





# MANIPULAÇÃO DE THREADS NO GNU/LINUX

Prezado aluno,

Neste capítulo você aprenderá como fazer programas que manipulam threads no GNU/Linux. Começaremos por algumas revisões sobre o conceito de threads nas quais veremos que a principal característica das threads é que elas compartilham a área de dados e código. Em seguida veremos como criar threads na linguagem C usando o padrão POSIX e veremos alguns exemplos de programação de threads. Ao final do capítulo, veremos os problemas que podem ser causados quando compartilhamos recursos. Aconselho que você faça todos os programas deste capítulo, para obter um melhor entendimento das informações aqui apresentadas.

Bom estudo!

Ague risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

## 6.1. Introdução

Vimos na disciplina Sistemas Operacionais I que, a partir do conceito de múltiplas threads (*multithread*), foi possível projetar e implementar aplicações concorrentes de forma eficiente, pois um processo pôde ter partes diferentes do seu código sendo executadas em paralelo com um *overhead* (sobrecarga) menor que ao utilizar múltiplos processos.

Como as threads de um mesmo processo compartilham o mesmo espaço de endereçamento, a comunicação entre threads não envolve mecanismos lentos de intercomunicação entre processos, aumentando, consequentemente, o desempenho da aplicação.

Mas lembre-se de que o desenvolvimento de programas que exploram os benefícios da programação multithread não é simples. A presença do paralelismo introduz um novo conjunto de problemas, como a comunicação e sincronização de threads. Existem diferentes modelos para a implementação de threads em um sistema operacional, em que desempenho, flexibilidade e custo devem ser avaliados.

Ague risus ac  
e velit at tellus.  
massa porttitor  
sectetur magna.

Fala Professor

Vimos também que um programa é uma sequência de instruções, composta por desvios, repetições e chamadas a procedimentos e funções. Quando um programa está sendo executado, há um registrador – chamado PC (*Program Counter*) ou IP (*Instruction Pointer*) – que aponta para o endereço da próxima instrução a ser executada. Ao longo da execução de um programa, os endereços de instruções apontados por esse registrador formam um caminho denominado “**fluxo de execução**” ou “**fluxo de controle**” (alguns também chamam de “**linha de execução**”).

## Conceitos



**Thread** pode ser definida como “**linha de execução**” ou “**fluxo de execução**” (“fluxo de controle”) dentro de um programa.

Obs: É importante lembrar que todo processo possui, no mínimo, uma thread.

Em um ambiente monothread, um processo suporta apenas uma única thread. Nesse ambiente, cada processo possui seu próprio contexto de hardware, contexto de software e espaço de endereçamento. Além disso, aplicações concorrentes são implementadas apenas com o uso de múltiplos processos independentes ou subprocessos. A utilização de processos independentes e de subprocessos permite dividir uma aplicação em partes que podem trabalhar de forma concorrente.

Em um ambiente multithread, cada processo pode possuir vários fluxos de execução – ou threads. Essas threads compartilham o mesmo espaço de endereçamento, o mesmo contexto de software e possuem seus próprios contextos de hardware e pilha.

Para que as threads sejam criadas, é necessário que sejam definidas no programa principal. De forma simplificada, uma thread pode ser definida como uma sub-rotina de um programa que pode ser executada de forma assíncrona, ou seja, executada concurrentemente ao programa chamar-dor. O programador deve especificar as threads, associando-as às sub-rotinas assíncronas. Dessa forma, um ambiente multithread possibilita a execução concorrente de sub-rotinas dentro de um mesmo processo.

## Fala Professor



*Por exemplo, suponha que exista um programa principal que realize a chamada de duas sub-rotinas assíncronas – através de threads. Inicialmente, o processo é criado apenas com uma thread, para a execução do programa principal. Quando o programa principal chamar as sub-rotinas, serão criadas as duas threads e estas threads serão executadas independentes da thread do programa principal. Neste processo, as três threads serão executadas concorrentemente.*

Threads compartilham o processador da mesma maneira que processos e passam pelas mesmas mudanças de estado (execução, espera e pronto). Por exemplo, enquanto uma thread espera por uma operação de E/S, outra thread pode ser executada. Para permitir a troca de contexto entre diversas threads, cada thread possui seu próprio contexto de hardware, com o conteúdo dos registradores gerais e específicos. Quando uma thread está sendo executada, seu contexto de hardware está armazenado nos registradores do processador. No momento em que a thread perde a utilização da UCP, as informações são atualizadas no seu contexto de hardware.

Dentro de um mesmo processo, threads compartilham o mesmo contexto de software e espaço de endereçamento com as demais threads; porém, cada thread possui seu contexto de hardware individual e sua pilha individual. Threads são implementadas internamente por meio de uma estrutura de dados denominada **bloco de controle da thread** (*Thread Control Block – TCB*). O TCB armazena, além do contexto de hardware, mais algumas informações relacionadas exclusivamente à thread, como prioridade, estado de execução e bits de estado.

A grande diferença entre aplicações monothread e multithread está no uso do espaço de endereçamento. Processos independentes e subprocessos possuem espaços de endereçamento individuais e protegidos, enquanto threads compartilham o espaço dentro de um mesmo processo. Essa característica permite que o compartilhamento de dados entre threads de um mesmo processo seja mais simples e rápido, se comparado a ambientes monothread.

*Como threads de um mesmo processo compartilham o mesmo espaço de endereçamento, não existe qualquer proteção no acesso à memória, o que permite que uma thread possa alterar facilmente dados de outros. Para que threads trabalhem de forma cooperativa, é fundamental que a aplicação implemente mecanismos de comunicação e de sincronização entre threads, a fim de garantir o acesso seguro aos dados compartilhados na memória.*

Angus risus at  
et velit at tellus.  
massa porttitor  
issectetur magna.

Fala Professor

O uso de multithreads proporciona uma série de benefícios. Programas concorrentes com múltiplas threads são mais rápidos do que programas concorrentes implementados com múltiplos processos, pois operações de criação, chaveamento de contexto e eliminação das threads geram menor *overhead* (sobrecarga). Como as threads dentro de um processo dividem o mesmo espaço de endereçamento, a comunicação

entre elas pode ser realizada de forma rápida e eficiente. Além disso, threads em um mesmo processo podem compartilhar facilmente outros recursos, como descritores de arquivos, temporizadores, sinais, atributos de segurança, etc.

A utilização do processador, dos discos e de outros periféricos pode ser feita de forma concorrente pelas diversas threads, significando melhor utilização dos recursos computacionais disponíveis. Em algumas aplicações, a utilização de threads pode melhorar o desempenho da aplicação apenas executando tarefas em background enquanto operações E/S estão sendo processadas. Aplicações como editores de texto, planilhas, aplicativos gráficos e processadores de imagens são especialmente beneficiadas quando desenvolvidas com base em threads.

Basicamente, existem dois tipos de Threads: as threads em modo usuário e as threads em modo kernel. A principal diferença entre esses dois modelos é o modo como as threads são vistas pelo sistema operacional.

- **Threads em modo usuário (TMU)** são implementadas pela aplicação e não pelo sistema operacional. Para isso, deve existir uma biblioteca de rotinas que possibilite à aplicação realizar tarefas como criação/eliminação de threads, troca de mensagens entre threads e uma política de escalonamento. Nesse modo, o sistema operacional não sabe da existência de múltiplas threads, sendo responsabilidade exclusiva da aplicação gerenciar e sincronizar as diversas threads existentes.
- **Threads em modo kernel (TMK)** são implementadas diretamente pelo núcleo do sistema operacional, por meio de chamadas a rotinas do sistema que oferecem todas as funções de gerenciamento e de sincronização. O sistema operacional sabe da existência de cada thread e pode escaloná-las individualmente. No caso de múltiplos processadores, as threads de um mesmo processo podem ser executadas simultaneamente, aumentando com isso o paralelismo da aplicação.

## 6.2. Implementação de Threads

Agora que já revisamos os conceitos mais importantes sobre threads, vamos estudar como elas podem ser programadas na linguagem C, utilizando o padrão POSIX. O padrão POSIX (*Portable Operating System Interface*) foi definido pelo IEEE (*Institute of Electrical and Electronics Engineers*). Esse padrão define o conjunto de funções que as bibliotecas devem oferecer, permitindo que as aplicações possam ser executadas em qualquer sistema operacional que siga as recomendações do padrão.

Esse padrão POSIX define um conjunto de funções que manipulam threads que a biblioteca “**pthreads**” (POSIX Threads) deve oferecer. Nessa biblioteca (cujo cabeçalho é denominado “*pthread.h*”), a criação de uma thread é realizada através da primitiva “***pthread\_create()***”. Essa primitiva é usada em conjunto com outra, a “***pthread\_join()***”, que serve para aguardar o término de uma thread e desalocar seus recursos.

### 6.2.1.A primitiva ***pthread\_create()***

A primitiva “***pthread\_create()***” cria uma nova thread, concorrente ao processo principal, que executará uma função do programa. Note que isso difere da criação de processos pois, após um “*fork*” ser realizado, o novo processo inicia sua execução pela instrução imediatamente abaixo do “*fork*”, ao passo que, após um “***pthread\_create()***”, a nova thread inicia sua execução a partir de alguma função escolhida do programa principal. A sinopse da primitiva *pthread\_create()* é fornecida a seguir.

```
#include <pthread.h>
int pthread_create(*identificador, *atributo, funcao,
argumentos);
```

em que:

<i>identificador</i>	é um ponteiro para uma variável do tipo <i>pthread_t</i> responsável pela identificação da thread recém-criada.
<i>atributo</i>	é um ponteiro para atributos que são armazenados em uma variável do tipo <i>pthread_attr_t</i> . Permite que o programador defina alguns atributos especiais como política de escalonamento, escopo de execução, dentre outros. Para usar o padrão do sistema, deve ser passado NULL.
<i>funcao</i>	é um ponteiro para a função que será executada pela thread.
<i>argumentos</i>	é um ponteiro do tipo <i>void*</i> que é passado como argumento para a rotina. Contém o endereço de memória dos dados para a thread criada.

O conjunto de instruções a ser executado pela thread é definido no corpo da função. A thread será disparada imediatamente após a primitiva “***pthread\_create()***” e terminará somente ao fim da função ou pela chamada “***pthread\_exit()***” que é análoga à função “*exit()*”.

### 6.2.2.A primitiva ***pthread\_join()***

A primitiva *pthread\_join()* bloqueia o processamento da thread que a chamou até que a thread identificada na função termine normalmente ou seja cancelada. Em seguida, desaloca os recursos da respectiva thread. Geralmente é chamada pelo processo principal para que este aguar-

de o término da thread criada, após o que a thread é desalocada. Normalmente, para cada `pthread_create()` é necessário um `pthread_join()`.

### Fala Professor

A primitiva `pthread_join()` é semelhante a primitiva `waitpid()`.

A sinopse da primitiva `pthread_join` é fornecida a seguir:

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

em que:

<i>thread</i>	Identificação da thread que cujo término vai ser aguardado.
<i>valor</i>	valor retornado pela thread após terminada.

Se o parâmetro *valor* for NULL, o valor de retorno da thread será descartado. Caso contrário, o valor de retorno será um dos argumentos da função `pthread_exit()`.

Quando uma thread comum termina, os recursos de memória utilizados – descritores e pilha – não são liberados até que todas as threads em execução do processo sofram a junção feita pela `pthread_join()`. Dessa forma, esta primitiva precisa ser executada para cada thread criada para evitar desperdício de memória.

### 6.2.3. Implementação de threads em C

As duas primitivas que vimos anteriormente são suficientes para implementarmos threads em C. Agora vamos criar um programa para testar seu funcionamento. Digite o programa a seguir e salve com o nome “`thread1.c`”.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 void* funcao_thread(void* arg)
6 {
7     printf("\tFILHA: sou a nova thread e aguardarei 10s.\n");
8     sleep(10);
9     printf("\tFILHA: terminarei agora");
10    return NULL;
11 }
12 int main()
13 {
14     pthread_t thread;
15     printf("PAI: criando uma nova thread...\n");
16     pthread_create(&thread, NULL, &funcao_thread, NULL);
17
18     printf("PAI: aguardando a conclusão da thread...\n");
19     pthread_join(thread, NULL);
20     printf("PAI: a thread terminou. Vou terminar também.\n");
21     return 0;
22 }
```

Para compilar esse programa é necessário a opção “-l pthread” para que o compilador encontre a biblioteca pthread e faça a ligação. Assim, compile esse programa com comando:

```
$ gcc thread1.c -o thread1 -Wall -l pthread
```

Em seguida, execute-o e observe o resultado:

```
$ ./thread1
PAI: criando uma nova thread...
FILHA: sou a nova thread e aguardarei 10s.
PAI: aguardando a conclusao da thread...
FILHA: terminarei agora.
PAI: a thread terminou. Vou terminar tambem.
```

Esse programa cria uma nova thread através da função “pthread\_create()” (linha 16), que executará a função “funcao\_thread()” (linha 5). Após a criação da nova thread, haverá dois fluxos de execução: o processo principal e a nova thread. Chamarei o processo principal de pai e a nova thread criada de filha. Quando o processo principal (pai) executar a função “pthread\_join()” (linha 19), ele ficará bloqueado aguardando até que a thread filha termine sua execução.

Por sua vez a thread filha aguardará 10 segundos (linha 8: a função *sleep* serve para aguardar um determinado número de segundos) e em seguida terminará. Ao terminar, a thread filha enviará automaticamente um sinal para o processo pai e este sairá do estado suspenso ou bloqueado em que se encontra (linha 19), desalocando os recursos da thread filha. Em seguida, o processo pai terminará.

Agora vejamos um exemplo de como criar duas threads. Digite o programa a seguir e salve-o com o nome “thread2.c”.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void funcao1(char* str)
7 {
8     printf("\tFilhal: vou aguardar 5 segundos...\n");
9     sleep(5);
10    printf("\tFilhal: já aguardei agora vou finalizar...\n");
11 }
12 void funcao2(char* str)
13 {
14     printf("\tFilha2: vou aguardar 10 segundos...\n");
15     sleep(10);
16     printf("\tFilha2: já aguardei agora vou finalizar...\n");
17 }
18 int main ()
19 {
20     void* retval;
21     pthread_t thread_a, thread_b;
22     printf("Pai: Criando duas threads...\n");
23     pthread_create(&thread_a, NULL, (void*)funcao1, NULL);
24     pthread_create(&thread_b, NULL, (void*)funcao2, NULL);
25     printf("Pai: Aguardando as duas threads...\n");
26     pthread_join(thread_a, &retval);
27     pthread_join(thread_b, &retval);
28     printf("Pai: Ambas as threads terminaram...\n");
29     return 0;
30 }
```

Compile-o com o comando:

```
$ gcc thread2.c -o thread2 -Wall -l pthread
```

Em seguida, execute-o e observe o resultado:

```
$ ./thread2
Pai: Criando duas threads...
    Filha1: vou aguardar 5 segundos...
    Filha2: vou aguardar 10 segundos...
Pai: Aguardando as duas threads...
    Filha1: já aguardei agora vou finalizar...
    Filha2: já aguardei agora vou finalizar...
Pai: Ambas as threads terminaram...
```

Nesse programa, o processo principal (pai) cria duas threads (linhas 23 e 24) e em seguida fica bloqueado aguardando o término delas (linhas 26 e 27). Quando ambas terminam, o processo pai termina também. Observe que cada thread, ao ser criada (linhas 23 e 24), é configurada para executar funções diferentes, a primeira thread executa a função “funcao1()” e a segunda a função “funcao2()”, que neste exemplo diferenciam-se apenas no tempo de “sleep” (linhas 9 e 15) – a primeira aguarda 5 segundos enquanto a segunda aguarda 10 –, mas poderiam ser funções totalmente diferentes.

Agora vejamos um exemplo de como passar argumentos para as threads. Digite o programa a seguir e salve-o com o nome “thread3.c”.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void funcao_thread(char* str)
7 {
8     printf("\tRecebi o argumento (%s) do processo pesado\n", str);
9     sleep(2);
10    printf("\tTerminando a thread que recebeu o argumento (%s).\n", str);
11 }
12 int main()
13 {
14     void* retval;
15     pthread_t thread_a, thread_b;
16     printf("Pai: Criando duas threads...\n");
17     pthread_create(&thread_a, NULL, (void*)funcao_thread, "AAA");
18     pthread_create(&thread_b, NULL, (void*)funcao_thread, "BBB");
19     printf("Pai: Aguardando as duas threads...\n");
20     pthread_join(thread_a, &retval);
21     pthread_join(thread_b, &retval); |
22     printf("Pai: Aguardando as duas threads...\n");
23     return 0;
```

Compile o programa com o comando a seguir:

```
$ gcc thread3.c -o thread3 -Wall -l pthread
```

Agora, observe o resultado de sua execução:

```
$ ./thread3
Pai: Criando duas threads...
    Recebi o argumento (AAA) do processo pesado
    Recebi o argumento (BBB) do processo pesado
Pai: Aguardando as duas threads...
    Terminando a thread que recebeu o argumento (AAA) .
    Terminando a thread que recebeu o argumento (BBB) .
Pai: Aguardando as duas threads...
```

Nesse exemplo, as threads são criadas nas linhas 17 e 18 e configuradas para executarem a mesma função “funcao\_thread()”. Nessas linhas também são passados os argumentos “AAA” e “BBB” para a primeira e segunda threads, respectivamente. As threads iniciam sua execução na função “funcao\_thread()” e após receberem seus respectivos argumentos, esperam 2 segundos e terminam. Feito isso, o processo principal que as estava aguardando termina também.

Agora vamos fazer um programa para tentar observar o escalonamento das threads e entender um pouco mais sobre sua concorrência. O programa a seguir cria duas threads e fica exibindo sequências diferentes. O próprio escalonamento das threads vai fazer com que seja exibida de forma alternada a saída da primeira thread com a saída da segunda. Digite o programa que segue e salve-o com o nome “thread4.c”.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 void funcao1(char* str)
6 {
7     int i;
8     for (i=0; i<100000; i++)
9         printf("\t11111111\n");
10    printf("\t\tTerminando a thread 1.\n");
11}
12 void funcao2(char* str)
13{
14    int i;
15    for (i=0; i<100000; i++)
16        printf("\t\t22222222\n");
17    printf("\t\tTerminando a thread 2.\n");
18}
19
20 int main()
21{
22    pthread_t thread_a, thread_b;
23    printf("Pai: Criando duas threads...\n");
24    pthread_create(&thread_a, NULL, (void*)funcao1, NULL);
25    pthread_create(&thread_b, NULL, (void*)funcao2, NULL);
26
27    printf("Pai: Aguardando as duas threads...\n");
28    pthread_join(thread_a, NULL);
29    pthread_join(thread_b, NULL);
30    printf("Pai: Ambas as thread terminaram...\n");
31    return 0;
32}
```

Para compilá-lo, utilize o comando:

```
$ gcc thread4.c -o thread4 -Wall -l pthread
```

Em seguida, execute-o e observe o resultado:

```
$ ./thread4
...
    11111111
    11111111
    11111111
Terminando a thread 1.

...
    22222222
    22222222
    22222222
Terminando a thread 2.
Ambas as thread terminaram...
```

Por razões óbvias, não coloquei aqui as mais de 200 mil linhas de saída, apenas um pequeno trecho para você ter uma idéia. Esse programa cria duas threads que ficam imprimindo seqüências diferentes na tela. Observe que a saída da função (funcao1) usada pela primeira thread (linha 9) exibe uma seqüencia de 1's precedido de uma tabulação. Já a saída da função usada pela segunda thread (linha 16) imprime um seqüencia de 2's precedido de duas tabulações – para facilitar a visualização. É possível observar o escalonamento das saídas alternadamente.

#### 6.2.4.Utilização de threads para aumentar o desempenho

As threads também podem ser utilizadas para aumentar o desempenho de programas. Por exemplo, suponha que você tenha um computador com dois cores (núcleos) de processamento e queira usá-lo para calcular uma integral. (Calma, não precisa ficar com receio, não vou ensinar cálculo aqui, Ok?! Trata-se apenas de mostrar como as threads podem ser usadas para aumentar o desempenho!). Na prática, para o computador, uma integral se resume apenas a uma sequência de somas. Assim, se você fizer metade destas somas em um núcleo e a outra metade em outro provavelmente o cálculo será feito na metade do tempo aproximadamente.

Como exemplo, vou apresentar um programa que calcula o valor de PI através de um integral com uma única thread. Em seguida, apresentarei o mesmo programa, porém utilizando duas threads. Espero que, quando executá-lo no meu computador, que é core duo, o tempo seja menor. Vamos então ao programa, sabendo que o valor de PI pode ser calculado através de uma integral definida de **0** a **1** da função  $f(x)=4/(1+x^2)$ . O programa abaixo calcula essa integral por aproximação de trapézios. Digite-o e salve-o com o nome de “integral.c”.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 double AreaTrapezio (double dx, double h1, double h2) {
7     double area;
8     area = dx * (h1+h2)/2;
9     return area;
10}
11 double f (double x){
12     return (4/(1+x*x));
13}
14 double CalculaArea(double a, double b, int N){
15     int i;
16     double area=0.0, dx, x1, x2, f1, f2;
17     dx = (b-a)/N;
18     for (i=0; i<N; i++){
19         x1 = a + dx * i;
20         x2 = a + dx * (i+1);
21         f1 = f(x1);
22         f2 = f(x2);
23         area += AreaTrapezio (dx, f1, f2);
24     }
25     return area;
26}
27 int main (int argc, char **argv) {
28     double a=0.0, b=1.0, area;
29     int n=50000000;
30     area = CalculaArea (a, b, n);
31     printf ("A área da curva eh:\n%.12lf\n", area);
32     return 0;
33}

```

Compile o programa com o comando:

```
$ gcc integral.c -o integral -Wall
```

Em seguida, use o comando “time” para medir o tempo de execução do programa (basta digitar time no início da linha):

```
$ time ./integral
A área da curva eh:
3.141592653590

real 0m5.808s
user 0m5.808s
sys 0m0.004s
```

O programa gastou 5.808s (real) para calcular a integral numérica de 0 a 1, utilizando 50 milhões de trapézios. Agora vamos fazer esse mesmo cálculo através de duas threads, uma calculará de 0.0 até 0.5 e a outra de 0.5 a 1.0. Como as threads serão calculadas em um computador com dois núcleos (estou executando em um Core Duo) espera-se que o tempo seja menor.

Agora vamos criar o programa que utiliza as duas threads para realizar o cálculo. Para isso, as duas threads criadas devem executar a rotina de cálculo da área em duas partes diferentes e, ao final, devemos juntar os resultados de cada thread. Digite o programa que segue e em seguida salve-o com o nome “integral\_thread.c”.

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

double integral = 0.0;

double AreaTrapezio(double dx, double h1, double h2)
{
    double area;
    area = dx * (h1+h2)/2;
    return area;
}

double f(double x)
{
    return (4/(1+x*x));
}

double CalculaArea(double a, double b, int N)
{
    int i;
    double area=0.0, dx, x1, x2, f1, f2;
    dx = (b-a)/N;
    for (i=0; i<N; i++)
    {
        x1 = a + dx * i;
        x2 = a + dx * (i+1);
        f1 = f(x1);
        f2 = f(x2);
        area += AreaTrapezio (dx, f1, f2);
    }
    return area;
}

void *funcao1()
{
    double area1 = CalculaArea (0.0, 0.5, 25000000);
    integral += area1; /* pode haver condicao de corrida aqui */
    return NULL;
}

void *funcao2()
{
    double area2 = CalculaArea (0.5, 1.0, 25000000);
    integral += area2; /* pode haver condicao de corrida aqui */
    return NULL;
}

int main (int argc, char **argv)
{
```

## Manipulação de Threads no GNU/Linux

```

pthread_t thread1, thread2;
pthread_create(&thread1, NULL, funcao1, NULL);
pthread_create(&thread2, NULL, funcao2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
printf ("A área da curva eh:\n%.12lf\n", integral);
return 0;
}

```

Compile o programa:

```
$ gcc integral_thread.c -o integral_thread -Wall -lp-thread
```

Em seguida, execute-o e observe o resultado:

```
$ time ./integral_thread
```

A área da curva eh:

3.141592653590

```

real 0m2.940s
user 0m5.844s
sys 0m0.000s

```

Como vemos, o programa que usou threads em paralelo gastou apenas 2.940s (real) contra os 5.808s do programa serial (monthread). O aumento no desempenho ocorre porque as threads rodam simultaneamente em processadores diferentes, ou seja, em paralelo. Com isso, observamos que o tempo praticamente caiu pela metade ao executarmos em dois cores simultaneamente, provando assim que as threads podem ser usadas para aumentar o desempenho. Agora vamos ver alguns problemas que podem acontecer quando utilizamos recursos compartilhados entre threads.

### 6.2.5.O problema do compartilhamento de recursos

Até o momento vimos como implementar threads criando aplicações concorrentes. Mas, como sabemos, as aplicações concorrentes muitas vezes necessitam que os processos comuniquem-se entre si. Essa comunicação pode ser implementada por meio de diversos mecanismos, como variáveis compartilhadas na memória principal ou trocas de mensagens. Nessa situação, é necessário que os processos concorrentes tenham sua execução sincronizada através de mecanismos do sistema operacional.

Os mecanismos que garantem a comunicação entre os processos concorrentes e o acesso a recursos compartilhados são chamados **mecanismos de sincronização**. No projeto de sistemas operacionais multiprogramáveis, é fundamental a implementação desses mecanismos, para se garantir a integridade e a confiabilidade na execução de aplicações concorrentes.

## Conceitos



**Condição de Corrida (Race Condition)** é a situação na qual alguns processos acessam simultaneamente uma região crítica, tornando o resultado inconsistente e dependente da ordem em que os acessos são realizados.

Um dos problemas mais comuns quando temos recursos compartilhados é chamado de *race condition*. Esse problema ocorre quando mais de um processo acessa os recursos compartilhados simultaneamente. Para evitar esse problema, apenas um processo por vez poderia acessar o recurso compartilhado enquanto os demais processos deveriam aguardá-lo até a liberação do recurso.

Para vermos como ocorre esse problema, digite o programa abaixo e salve-o com o nome de “race\_condition.c”.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 /* variável global - compartilhada entre as threads*/
7 int conta = 0;
8
9 void *funcao1()
10{
11    int i;
12    for(i=0; i<100000000; i++)
13        conta = conta + 3;
14    printf("Encerrei a thread 1\n");
15    return 0;
16}
17 void *funcao2()
18{
19    int i;
20    for(i=0; i<100000000; i++)
21        conta = conta + 7;
22    printf("Encerrei a thread 2\n");
23    return 0;
24}
25 int main()
26{
27    pthread_t thread1, thread2;
28    pthread_create(&thread1, NULL, funcao1, NULL);
29    pthread_create(&thread2, NULL, funcao2, NULL);
30    pthread_join(thread1, NULL);
31    pthread_join(thread2, NULL);
32    printf("O valor da conta eh: %d\n", conta);
33    return 0;
34}
```

Agora compile-o com o comando:

```
$ gcc race_condition.c -o race_condition -Wall -l pthread
```

Em seguida, execute várias vezes e observe o resultado:

```
$ ./race_condition
Encerrei a thread 1
Encerrei a thread 2
O valor da conta eh: 925443770
$ ./race_condition
Encerrei a thread 1
Encerrei a thread 2
O valor da conta eh: 801939410
$ ./race_condition
Encerrei a thread 1
Encerrei a thread 2
O valor da conta eh: 836119155
$ ./race_condition
Encerrei a thread 1
Encerrei a thread 2
O valor da conta eh: 949457405
$ ./race_condition
Encerrei a thread 1
Encerrei a thread 2
O valor da conta eh: 823170301
```

Esse programa cria duas threads que escrevem simultaneamente em uma variável global compartilhada “conta” (linha 7). A primeira thread soma 100 milhões de vezes o valor 3 na variável conta. A segunda thread soma 100 milhões de vezes o valor 7. Portanto, ao final de ambas as threads, o valor esperado na variável conta deveria ser 1 bilhão (ou seja, 300 milhões mais 700 milhões). No entanto, observamos que o programa foi executado várias vezes e em todas resultou em valores incorretos devido aos acessos simultâneos na variável conta. Como vimos anteriormente – e agora estamos vendo na prática – esse problema é denominado *race condition*.

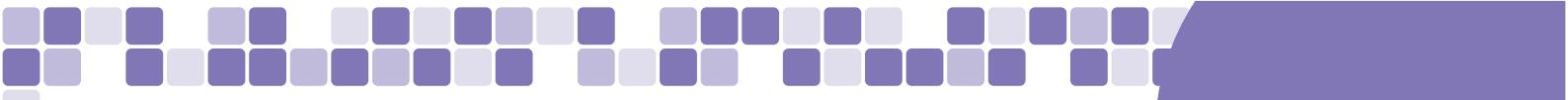
### Atividades

1. Faça todos os programas deste capítulo, execute-os e observe suas saídas.
2. Em seguida, explique o funcionamento de cada programa com as suas palavras.
3. Proponha uma solução teórica – conforme vimos na disciplina de Sistemas Operacionais 1 – para o problema de *race condition* encontrado no último programa.
4. Descubra se há alguma biblioteca no Linux que possibilite implementar de forma prática em C uma solução para o problema de *race condition* encontrado no último programa.



### Atividades





## Referências Bibliográficas

Mazioli, Gleydson. **Guia Foca Linux.** ([www.guiafoca.org](http://www.guiafoca.org)). 2007.

Ribeiro, Uirá. **Sistemas Distribuídos – Desenvolvendo Aplicações de Alta Performance no Linux.** 1.ed. Aexcel, 2005

MACHADO, F.B. e MAIA, L.P. **Arquitetura de Sistemas Operacionais.** 4.ed. LTC, 2007.

SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. **Fundamentos de Sistemas Operacionais.** 6.ed. LTC, 2004.

TANENBAUM, A.S. **Sistemas Operacionais Modernos.** 2.ed. Pearson Brasil, 2007.

OLIVEIRA, R.S., CARISSIMI, A.S., TOSCANI, S.S. **Sistemas Operacionais.** 3.ed. Sagra-Luzzato. 2004.

Site da distribuição Ubuntu no Brasil (<http://www.ubuntu-br.org/>)

Suporte do Ubuntu no Brasil (<http://www.ubuntu-br.org/suporte>)

Site do projeto GNU (<http://www.gnu.org/>)

Informações sobre sistemas Linux (<http://pt.wikipedia.org/wiki/Linux>)

Dicas, notícias e tutoriais sobre Linux (<http://br-linux.org/>)

Comunidade Linux no Brasil (<http://www.vivaolinux.com.br/>)

Apostilas gratuitas e documentação sobre o Linux (<http://www.dicas-l.com.br/>)

POSIX Thread Programming (<https://computing.llnl.gov/tutorials/pthreads/>)

GNU Bash Reference Manual (<http://www.network-theory.co.uk/docs/bashref/>)

The CTDP Linux Programmer's (<http://www.comptechdoc.org/os/linux/programming/>)

## REFERÊNCIAS BIBLIOGRÁFICAS

Goldt, Sven. **The Linux Programmer's Guide**. Version 0.4, 1995

Free Linux Programming Books (<http://www.techbooksforfree.com/linux.shtml>)