Revisão de Ambiente Linux

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Introdução

- A maioria dos sistemas operacionais suportam ambientes de programação com paralelismo;
- entretanto, ambientes UNIX são os mais utilizadas em computação de alto desempenho;
- a grande maioria dos sistemas listados no `top500.org` usam Linux ou variantes;
- sistemas embarcados também ou usam Linux ou sistemas com forte influência UNIX.

Na nossa disciplina, vamos usar Linux. Cabe uma breve revisão para nivelamento.

Introdução

- A maioria das linguagens suporta paralelismo, seja de forma explícita ou implicíta;
- linguagens clássicas ainda são importantes: Fortran, C/C++, etc;
- mas Java, Python, Go, etc também fazem uso de paralelismo.

Na nossa disciplina, irei mostrar exemplos em C. Mas os trabalhos podem ser em outras linguagens.

Linux

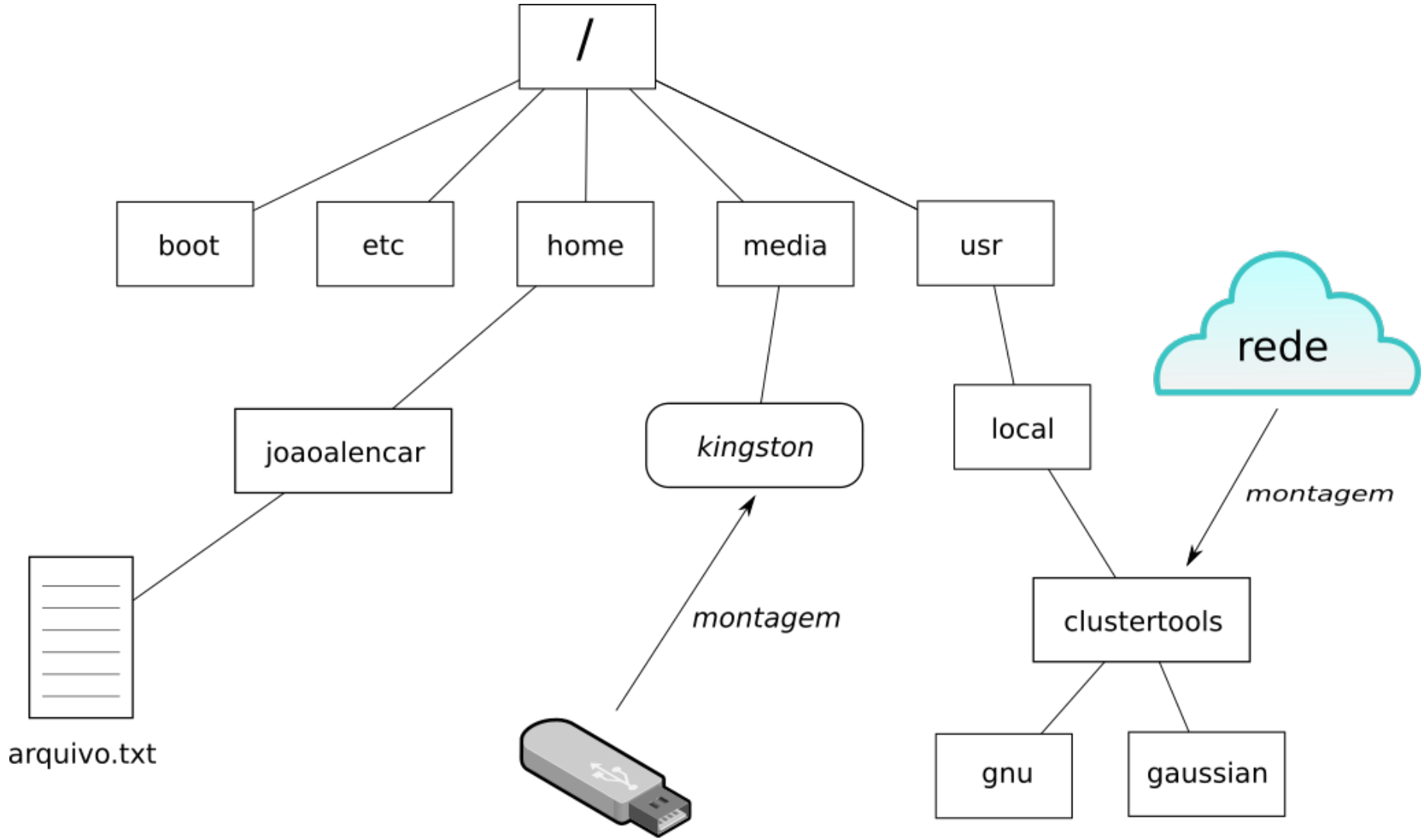
Acesso por linha de comando:

- Uma vez feito o *login* em um sistema, você tem acesso a um terminal;
- nas distribuições *desktop*, o terminal é substituído por um gerenciador de janelas;
- no acesso via terminal, o usuário pode executar comandos e navegar no sistema de arquivos;
- Como é organizado o sistema de arquivos?

Sistema de Arquivos e Diretórios

- No Linux, uma partição é escolhida para conter a base do sistema;
- essa partição é chamada de raiz (/);
- a partir da raiz, diretórios e arquivos são criados;
- caso seja necessário, outra partição pode ser *montada* no caminho de um diretório existente;
  - *pendrives*, discos externos ou diretórios de rede são montados;
  - cabe ao administrador decidir o caminho a partir da raiz.
- assim como no Windows, alguns diretórios são padronizados;
- /home/joaoalencar/arquivo.txt .

Sistema de Arquivos e Diretórios



Permissões no Linux

- Cada diretório ou arquivo pertence a um usuário e a um grupo.
- as permissões no Linux são agrupadas em três categorias: **usuário, grupo e outros**.
- cada categoria tem três tipos de permissão: **leitura, escrita e execução**.
- cada tipo de permissão pode estar ativada ou não.
- para usuários de desenvolvimento em C, a permissão mais importante é a execução. Veremos adiante como configuração.

O Prompt de Comandos BASH

- O *Bash* é o ambiente de linha de comando do Linux;
- o *prompt* terminado com o caractere $ indica que é um usuário comum;
- o *Bash* já possui alguns comandos embutidos e permite executar programas que estão no sistema de arquivos do Linux;
- o *Bash* interpreta a primeira palavra fornecida como um comando. Esse comando pode ser embutido ou está em algum diretório do $PATH (explicaremos adiante);
- o usuário também pode fornecer o caminho completo.

Comandos Básicos

- Vamos revisar alguns comandos básicos;
- manipulação de arquivos e diretórios;
- são importantes para quando acessamos um servidor remoto.

ls - Lista arquivos

- ls [diretório]
  - -l = Lista em formato detalhado.
  - -a = Todos os arquivos, ocultos inclusive.
  - -r = Ordem alfabética reversa.
  - -h = Formato humano.
  - -R = Incluir subdiretórios.
- *

cd - Muda o diretório atual

- cd [diretório]
  - . = Diretório atual.
  - .. = Diretório acima.
  - ~ = Diretório do usuário.
  - / = Diretório raiz.
  - - = Último diretório.

cp - Copia Arquivos e Diretórios

- cp [opções] arq-origem arq-destino
  - -i = Modo interativo.
  - -v = Mostra o que está sendo copiado.
  - -r = Copia recursivamente (diretórios e subdiretórios).

mv - Move arquivos e Diretórios

- mv [opções] arq-origem arq-destino
  - Opções semelhantes ao cp.

rmdir - Remove um diretório

- rmdir nome-do-diretório
  - -v = Descreve o que está fazendo.
  - -p = Remove todos os diretórios no caminho.

rm - Deleta arquivos e diretórios.

- rm [opções] arquivo
  - -f = Ignora arquivos inexistentes e não pede nenhuma confirmação.
  - -I = Sempre pede confirmação.
  - -r = Remove diretórios.

tar - Agrupar Vários Arquivos em Um

- tar [opções] arquivo_tar arquivo
  - -c = criar um novo arquivo .tar (não pode ser usada com -x).
  - -f = indica que o destino é um arquivo.
  - -p = preserva as permissões.
  - -r = adiciona arquivos a um .tar já existente.
  - -t = lista o conteúdo de um .tar.
  - -u = adiciona arquivos ao .tar somente se eles ainda não estão lá.
  - -v = mostra o nome de cada arquivo.
  - -x = retira os arquivos de um .tar (não pode ser usado com -c).
- tar -cvf download.tar Downloads/
- tar -xvf download.tar
- Não é o mesmo que compactar!!!

gzip - Compactar Arquivos

- gzip [opções] arquivo
  - -c = escreve na saída padrão, ignorando o arquivo.
  - -f = força a compactação.
  - -r = compacta recursivamente.
  - -v = mostra o nome de cada arquivo.
  - -d = descompacta.
- gzip download.tar
- gzip -d download.tar.gx
- tar -xzvf download.tar.gz

Editor de Texto

- Existem várias opções para editores de texto no Linux: vim, emacs, nano, etc;
- o **nano** é o mais simples no terminal;
- você pode começar logo que o programa inicia, o comando CTRL-O permite salvar o arquivo;
- o comando CTRL-W permite fazer buscas no texto;
- para recortar e colar:
  - Inicie o local da seleção com CTRL-^;
  - mova o cursor até a posição que deseja recortar;
  - recorte com CTRL-K;
  - mova para a posição de destino e cole com CTRL-U;
- CTRL-X sai do editor.

Variáveis de Ambiente

- Algumas variáveis já são definidas pelo sistema, elas permitem configurar o ambiente;
- A variável PATH contém diretórios com programas executáveis:
  - Para executar um programa no terminal sem ter que digitar o caminho completo, devemos adicionar o diretório que contém o executável na variável PATH;
  - Os diretórios são listados em ordem de busca, separados por : ;
  - O arquivo do programa precisa estar com a permissão de execução.
- O usuário não precisa configurar essa variável toda vez que for usar o sistema;
- O arquivo **.bashrc**, presente no diretório do usuário, permite configurar variáveis;
- É um arquivo oculto que é carregado **toda vez** que o usuário faz *login*.

Exemplo de **.bashrc**

```
# Adicionar os executáveis da pasta apps ao £PATH
export PATH=/home/usuario01/apps:$PATH

# Adicionar JAVA ao PATH
export JAVA_HOME=/opt/jdk
export PATH=$JAVA_HOME/bin:$PATH

# Executáveis do CUDA
export CUDA_HOME=/opt/cuda/9.0/
export PATH=$CUDA_HOME/bin:$PATH
```

SSH

```
jmhal@saturn:~$ ssh servidor.quixada.ufc.br
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-138 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

32 packages can be updated.
30 updates are security updates.


*** System restart required ***
No mail.
Last login: Fri Jan 25 09:47:40 2019 from 191.190.87.200
jmhal@servidor:~$
```

SSH - Formato geral

```
$ ssh -p 2200 usarioremoto@servidor.quixada.ufc.br
```

- *usarioremoto* é o usuário da máquina remota que você tem acesso;
- se for o mesmo nome de usuário local, pode ser omitido;
- o número 2200 é a porta TCP na qual o servidor SSH está escutando;
- se a porta utilizada for a 22, essa opção é desnecessária.

O normal é que após executar o comando, seja requisitada a senha. No lugar da senha, você pode fornecer o caminho de um arquivo *chave*:

```
$ ssh -i chave.pem usarioremoto@servidor.quixada.ufc.br
```

o *login* pode prosseguir sem senha.

SCP - Copiar Arquivos Através do SSH

```
# Copiar um arquivo do servidor01 para o servidor02
$ scp usuario01@servidor01:/home/usuario01/teste.txt \
usuario02@servidor02:/home/usuario02/teste.txt

# Copiar um arquivo do servidor01 para a máquina local
$ scp usuario01@servidor01:/home/usuario01/teste.txt \
 teste.txt

# Copiar um arquivo local para o servidor
$ scp teste.txt usuario01@servidor01:/home/usuario01/
```

Caso a porta não seja a 22, podemos utilizar a opção $-P$ para informar a porta. Para copiar diretórios, usamos a opção $-r$.

Compilação e Instalação de Programas C/C++

- Até o momento:
  - Sabemos manipular arquivos e diretórios;
  - Somos capazes de transferir arquivos remotos, compactados ou não;
  - Podemos editar arquivos de texto.
- Vamos partir agora para a instalação de programas.
- Existem duas principais opções para instalar programas no Linux:
  - Pacotes da Distribuição;
  - Compilação de Código Fonte.

Pacotes da Distribuição

- Os programas são compilados pelos mantenedores da distribuição;
- O usuário baixa um arquivo compactado (pacote) com o programa;
- Dentro do pacote, além do programa, existe um *script* que copia os arquivos binários e configura as variáveis de ambiente;
- Em geral, os programas e bibliotecas são copiados para diretórios na pasta /usr e os arquivos de configuração para o /etc;
- Programas ficam acessíveis para todos os usuários;
- Exige acesso administrativo (usuário *root*).

Pacotes da Distribuição

- Vantagens:
  - Existem ferramentas que facilitam a instalação (apt-get, yum, dnf, etc);
  - Os mantenedores das distribuições tem experiência na compilação, gerando executáveis estáveis e livres de *bugs*;
  - Por ser o método mais popular, é adotado pela maioria dos tutoriais.
- Desvantagens:
  - Dependem da distribuição. Muitas vezes, um mesmo programa tem pacotes com nomes diferentes no Red Hat e no Ubuntu;
  - Os executáveis não são otimizados para o desempenho e sim para serem compatíveis com o maior número de máquinas;
  - A maioria dos programas científicos não estão nos repositórios das principais distribuições;
  - Instalações antigas possuem versões defasadas dos pacotes;
  - Ao instalar em um *cluster*, o pacote precisa ser instalado em todas as máquinas.

Compilação de Código Fonte

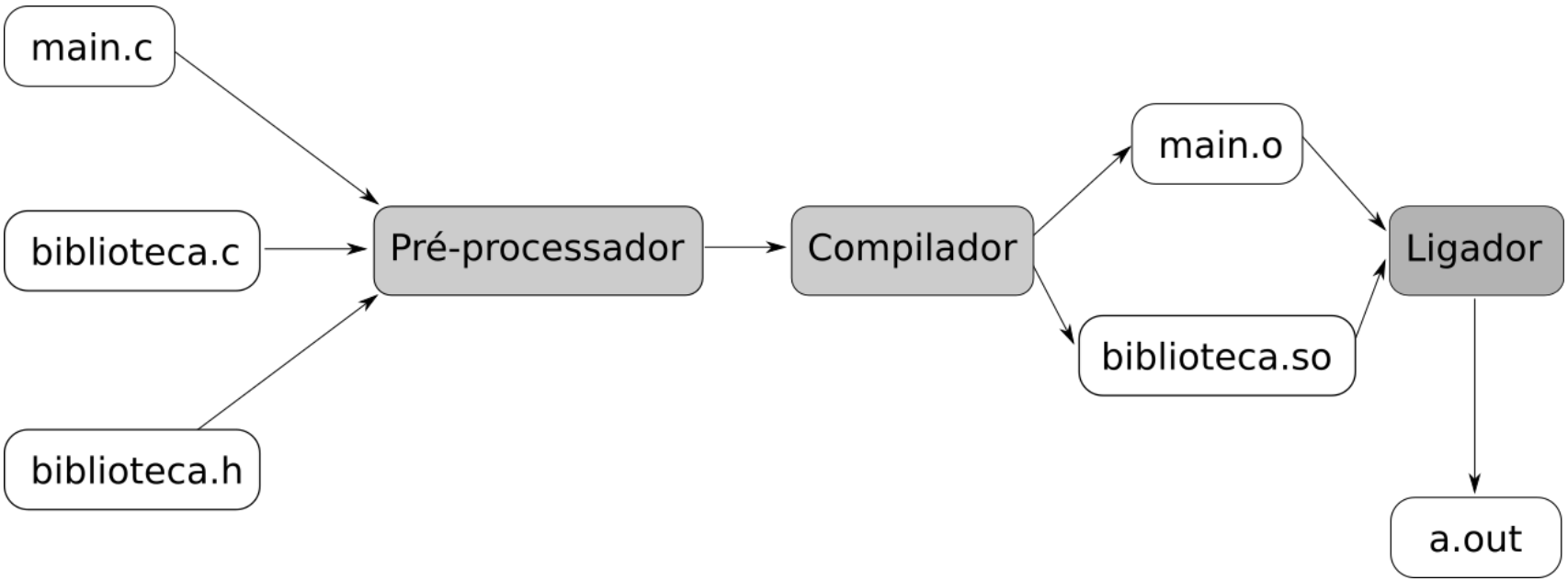
- O usuário baixa o código do desenvolvedor do programa e faz a compilação;
- É preciso ter um compilador instalado, mas não é preciso entender a linguagem de programação do programa;
- A maioria dos desenvolvedores usam ferramentas que permitem automatizar boa parte do processo de compilação (make, cmake, ant, etc);
- Ao contrário do pacote, o usuário tem que definir onde serão instalados os executáveis;
- As dependências precisam ser instaladas antes da compilação do programa;
- Não exige acesso administrativo (usuário *root*).

Como vamos desenvolver código paralelo durante a disciplina, é importante saber compilar programas no UNIX.

Compilação de Código Fonte

- Vantagens:
  - Funciona para qualquer distribuição, muitas vezes até em outros sistemas operacionais (BSD, MacOS, etc);
  - O usuário pode customizar a compilação para tirar o máximo proveito da máquina alvo;
  - Como o local de instalação pode ser personalizado, permite instalar em uma pasta compartilhada em todo o *cluster*.
- Desvantagens:
  - Exige mais conhecimento por parte do usuário;
  - A resolução de dependências pode exigir várias compilações;
  - Necessita de configuração manual das variáveis de ambiente;
  - O processo de compilação pode demorar vários minutos.

Visão Geral do Processo de Compilação



Exemplo de Compilação - Programa em C

- Vamos compilar um simples programa em C para entender as etapas;
- Considere uma pasta *projeto* com o seguinte conteúdo:
  - Subdiretório *include* com o arquivo *biblioteca.h*;
  - Subdiretórios *lib* e *bin* vazios;
  - Subdiretório *src* com os arquivos *main.c* e *biblioteca.c*.
- Nós vamos criar uma biblioteca e usá-la em um programa principal;
- A divisão do programa em vários arquivos é regra nos programas científicos.

Exemplo de Compilação - Programa em C

```c
/*** Conteúdo de biblioteca.h ***/
void funcao_da_biblioteca();

/*** Conteúdo de biblioteca.c ***/
#include <stdio.h>
void funcao_da_biblioteca() {
   printf("Olá Mundo da Biblioteca.\n");
   return;
}

/*** Conteúdo de main.c ***/
#include <stdio.h>
#include "biblioteca.h"
int main(int argc, char *argv[]) {
   funcao_da_biblioteca();
   return 0;
}
```

Exemplo de Compilação - Programa em C

- O exemplo é trivial, mas já envolve muitos passos;
- Um programa científico, com centenas de arquivos fonte, compilado dessa forma seria impraticável;
- A maioria utiliza ferramentas (*autoconf*) para automatizar a compilação
  1. Informações sobre o ambiente são inseridas em um arquivo de configuração;
  2. Um *script* chamado *configure* realiza testes e coleta mais informações sobre o sistema;
  3. O comando *make* invoca o compilador várias vezes para construir o projeto.

Exemplo de Compilação - Programa em C

```
$ cd projeto
$ gcc -Iinclude/ -c src/main.c -o bin/main.o
$ gcc -Iinclude/ -c src/biblioteca.c -fPIC\\
-o bin/biblioteca.o
$ gcc -shared bin/biblioteca.o -o lib/libbiblioteca.so
$ gcc bin/main.o -Llib/ -lbiblioteca -o bin/a.out
$ bin/a.out
bin/a.out: error while loading shared libraries
$ LD_LIBRARY_PATH=lib/ bin/a.out
Olá Mundo da Biblioteca.
```

Podemos atualizar o *.bashrc* para configura a variável *LD_LIBRARY_PATH*.

Exemplo de Compilação - OpenMPI

```
$ mkdir sources
$ cd sources
$ wget \\
https://www.open-mpi.org//software/ompi/v3.0/downloads/openmpi-3.0.0.tar.bz2
$ tar -xjvf openmpi-3.0.0.tar.bz2
$ mkdir ~/install/openmpi
$ cd openmpi-3.0.0
$ ./configure --prefix=/home/usuario01/install/openmpi
$ make
$ make install
```

Exemplo de Compilação - OpenMPI

```
# OpenMPI configuration for .bashrc
export OPENMPI_DIR=/home/usuario01/install/openmpi
export PATH=$OPENMPI_DIR/bin:$PATH
export LD_LIBRARY_PATH=$OPENMPI_DIR/lib:$LD_LIBRARY_PATH
export MANPATH=$OPENMPI_DIR/share/man/:$MANPATH

export OMPI_MCA_btl_vader_single_copy_mechanism=none
export OMPI_MCA_btl_sm_use_knem=0
```

Conclusão

- Utilizar a linha de comando em ambientes Linux é importante quando acessamos um servidor remoto;
- a maioria dos supercomputadores ou *clusters* de alto desempenho são acessados via SSH/SCP;
- é importante saber como compilar programas, ferramentas e bibliotecas a partir do código fonte.