Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0036 – Sistemas Operacionais

Conceitos Básicos de Linux e Linguagem C

Prof. Gustavo Girão
girao@imd.ufrn.br

Roteiro

- Concietos
- Kernel vs MicroKernel
- Linux
  - Comandos Básicos
  - Manipulação de Processos
  - Fluxos e Arquivos
- Chamadas de Sistema
- Ponteiros em Linguagem C

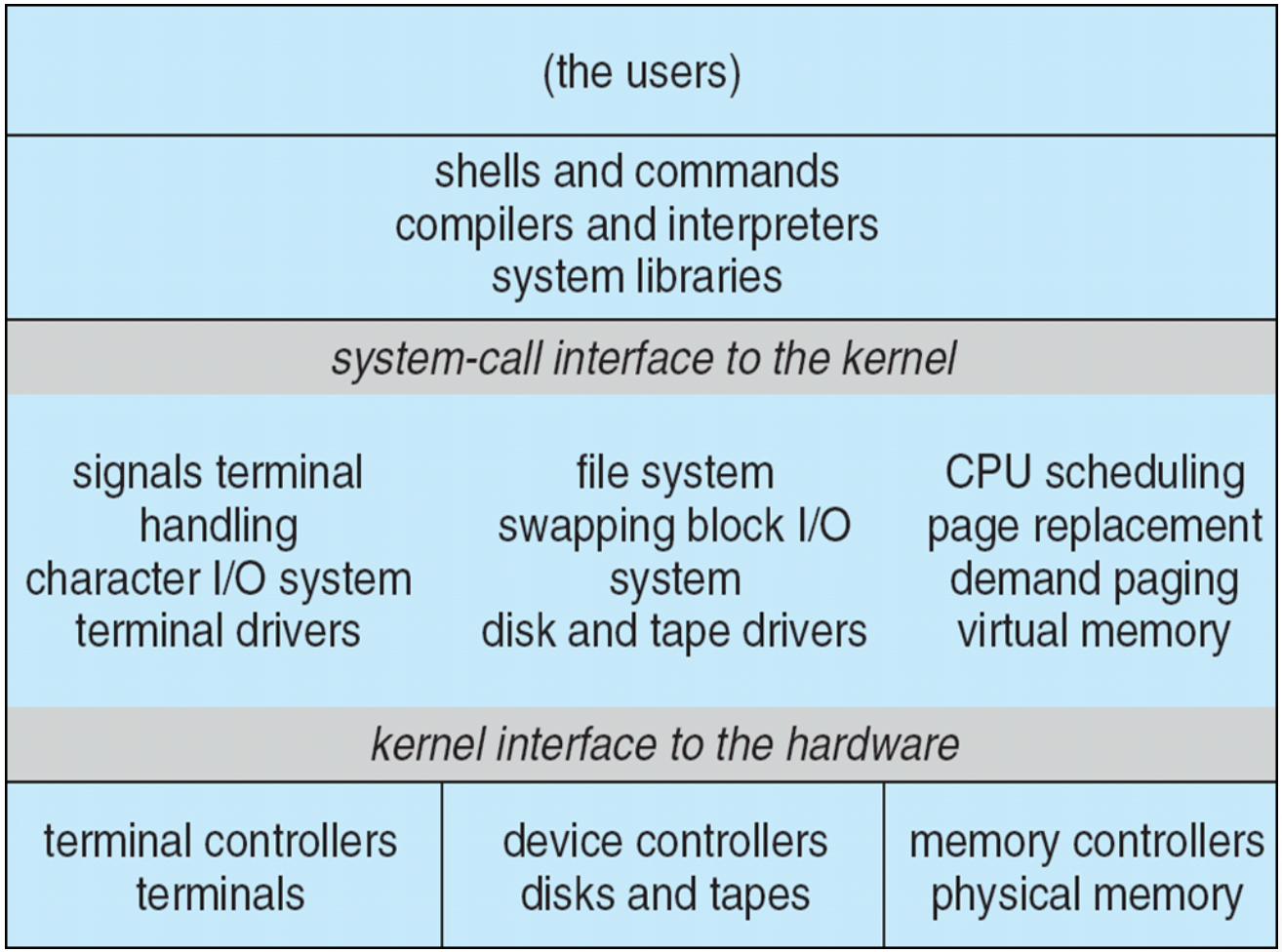
Conceitos

- Criado em 1969 por Ken Thompson e Dennis Ritchie do Bell Labs. Derivado do MULTICS
- Sua terceira versão foi escrita em C: linguagem desenvolvida na própria bell labs para suportar o UNIX
- Contava com vários grupos de desenvolvimento mesmo for a da Bell Labs e At&T (parceira comercial).
  - A mais influente? Universidade de Berkley (Berkeley Software Distributions - **BSD**)
- Vários projetos de padronização procuraram consolidar variantes do UNIX e caracterizar uma interface de programação UNIX
  - IEEE -> POSIX

Conceitos

- Projetado para ser um sistema de time-sharing
- Tem uma interface padrão simples (shell) que pode ser substituída
- Sistema de arquivos com árvore de diretórios multinível
- Suporta multiplos processos. Um processo pode facilmente criar novos processos
- Alta prioridade para ter um sistema interativo provendo facilidades para o programador

Conceitos


(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory

Kernel

- O UNIX consiste de duas partes separáveis:
- Kernel: tudo abaixo da interface de chamada de sistema e acima do hardware
  - Provê sistema de arquivos, escalonamento da CPU, gerência de memória e outras funções do SO
- Utilitários (system programs): usam as chamadas de sistema suportadas pelo kernel para proverem funções úteis como compilação e manipulação de arquivos
  - Um dos mais importantes?
    - O Shell (que **cerca** o kernel)

Linux – árvore de diretórios

Diretório do root

A primeira hierarquia do sistema de arquivos ou somente:

/

Hierarquia primária

/bin/	Binários principais dos usuários
/boot/	Arquivos do sistema de Boot
/dev/	Arquivos de dispositivos
/etc/	Arquivos de configuração do sistema
/home/	Diretório dos usuários comuns do sistema
/lib/	Bibliotecas essenciais do sistema e os módulos do kernel
/media/	Diretório de montagem de dispositivos
/mnt/	Diretório de montagem de dispositivos - *Mesmo que "media"*
/opt/	Instalação de programas não oficiais da distribuição ou por conta do usuário
/sbin/	Armazena arquivos executáveis que representam comandos administrativos. Exemplo: shutdown
/srv/	Diretório para dados de serviços fornecidos pelo sistema
/tmp/	Diretório para arquivos temporários
/usr/	Segunda hierarquia do sistema, onde ficam os usuários comuns do sistema e programas
/var/	Diretório com arquivos variáveis gerados pelos programas do sistema. Exemplo: logs, spool de impressoras, e-mail e cache
/root/	Diretório do usuário root – usuário root tem total poderes sobre o sistema, podendo instalar, desinstalar e configurá-lo.
/proc/	Diretório virtual controlado pelo Kernel com configuração total do sistema.

http://nitro-infomatica.blogspot.com

Linux – Comandos Básicos

- ***history*** – histórico de todos os comandos executados nesta sessão
- ***Setas para cima ou para baixo*** – navega entre os comandos executados
- ***ctrl+r*** - busca por comandos recentemente executados
- ***!****comando* – executa o comando com os ultimos parametros utilizados
- ***man comando*** – abre a pagina do comando no manual

Linux – Comandos Básicos

- ***ls*** – lista arquivos em um diretório
- ***touch*** – cria ou atualiza um arquivo
- ***grep*** – busca e exibe padrões encontrados no arquivo de entrada
  - *-b*: numero de bytes entre um resultado e outro
  - *-e*: busca de multiplos padrões
  - *-R*: busca recursiva
  - Expressões regulares:
    - *** -> tudo**
    - **^A -> começam com a letra A**
    - **A$ -> terminam com a letra A**
- Pipe ("|"): redireciona a saida de um comando para a entrada de outro

Linux – Comandos Básicos

- ***cut*** – corta porções selecionadas de cada linha do arquivo
  - -d"c": especifica um delimitador 'c' ao invés do tab
  - -f#: especifica campos separados pelo delimitador

- Pipe ("|"): redireciona a saida de um comando para a entrada de outro

Exercícios

- Liste as linhas do arquivo **capitais** onde duas vogais quaisquer aparecem juntas.
- No mesmo arquivo, liste os estados ou capitais que começam por vogal.
- Agora liste apenas as linhas onde os estados começam por vogal.

Linux – Fluxos e Arquivos

- ***cp, mv, rm*** – copia, move e remove arquivos
  - *cp -R*: copia recursivamente
  - *cp -p*: preserva os atributos
  - *cp –n*: não sobreescreve arquivos existentes
- ***mkdir*** - cria um diretório
- ***rmdir*** – remove um diretório
- ***find*** – busca um arquivo em um diretório
- Redirecionamentos (“>”, “<“)

Linux – Processos

- ***top*** – exiba e atualize as informações de todos os processos em execução
  - -n #: limita o numero maximo de processos exibidos
  - -o key: ordena de acordo com um parametro
- ***ps*** – exibe o status dos processos ativos
- ***kill*** – encerra a execução de um processo

Chamadas de Sistema

- Chamadas de sistema definem a interface de programação do UNIX
- O conjunto de utilitários comumente disponíveis definem a interface do usuário
- A interface com o programador e com o usuário definem o contexto com o kernel precisa suportar
- As 3 principais categorias de chamadas de sistema no UNIX são:
  - Manipulação de arquivos
  - Controle de processos
  - Manipulação de informação

Chamadas de Sistema

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

LINGUAGEM C

Recebendo argumentos por linha de comando

- Para receber argumentos vindos da execução do programa, utilizamos as estruturas argc e argv previstas no padrão ANSI
- **Argc** é um valor inteiro que representa o número de argumentos utilizados ao executar o programa (incluindo o nome do proprio programa)
- **Argv** é um vetor que armazena cada argumento passado (incluindo o nome do programa)

- A maneira de se utilizar é a seguinte:
  - int main (int argc, char *argv[])

Ponteiros em Linguagem C

- Ponteiros são tipos especiais de variáveis
- Representam um endereço na memória
- São utilizados para
  - Passagem de parametros por referencia
  - Manipulação dinâmica de estruturas de dados
  - Alocação dinâmica de memória
- Em uma declaração de variável, utiliza-se o simbolo * para representar que aquela variável é um ponteiro e "aponta" para um espaço de memória daquele tipo
  - o operador * é um "operador de indireção"
- O operador de endereço & representa o endereço em si

Ponteiro Nulo

- Existe um valor especial para indicar que um ponteiro não aponta para nenhum compartimento válido de memória: é o ponteiro nulo.
- Este valor é denotado:
  - `NULL`
- Temos garantia que se aplicarmos o operador de endereço `&` ao uma variável ou qualquer outra abstração da memória, o resultado nunca será `NULL`.

- Há um erro de execução se aplicarmos o operador de indireção `*` a `NULL`.

Aritmética de ponteiros

- Seja `p` um ponteiro para um valor do tipo `t`.
  - Suponha que o valor de `p` seja o endereço `A`
  - Suponha que o número de bytes para representar um valor do tipo `t` seja `q`: `sizeof(t)` é `q`.
- A expressão `p + 1` é o endereço `A + q`
- A expressão `p + i` é o endereço `A + (i-1)×q`
- A expressão `p - 1` é o endereço `A - q`
- A expressão `p - i` é o endereço `A - (i-1)×q`

Apelidos

- Duas expressões são apelidos se
  - ambas são ponteiros (ou seja seu valor é algum endereço)
  - tem o mesmo valor.

- Logo, se *P1* e *P2* são apelidos, se alteramos o valor no endereço apontado por *P1*, também alteramos o valor no endereço apontado pela *P2*.

Aritmética de ponteiros e arranjos

- Seja `v` um arranjo de valores do tipo `t`.
  - `v` é o endereço do primeiro elemento do arranjo
  - `&v` e `&v[0]` são iguais
  - `*v` e `v[0]` são apelidos
  - `*(v+1)` e `v[1]` são apelidos
  - `*(v+i)` e `v[i]` são apelidos

Exercícios

- Suponha que as seguintes declarações foram realizadas:
  - ```
    int v[] = {5, 15, 34, 54, 14, 2, 52, 72};
    int *p = &v[1];
    int *q = &v[5];
    ```
- Qual o valor de `*(p + 3)`?
- Qual o valor de `*(q - 3)`?
- Qual o valor de `q-p`?
- A expressão `p < q` tem valor verdadeiro ou falso?
- A expressão `*p < *q` tem valor verdadeiro ou falso?
    ```

Alocação Dinâmica de Memória

- A alocação dinâmica permite ao programador criar variáveis em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado.
- Deve ser utilizada quando não se sabe ao certo quanto de memória será necessário para o armazenamento das informações
  - A quantidade pode ser determinadas em tempo de execução conforme a necessidade do programa.
  - Dessa forma evita-se o desperdício de memória ou a falta de memória.

Alocação Dinâmica de Memória

- O padrão C ANSI define algumas funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**. As principais são:
  - malloc
  - calloc
  - free

Malloc

- A função **malloc()** serve para alocar memória e tem o seguinte protótipo:

```
void *malloc (unsigned int num);
```

- A funçao pega o número de bytes que queremos alocar (**num**), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado.

Malloc

- A função retorna um ponteiro genérico (void *) para o ínicio da memória alocada que deverá conter espaço suficiente para armazenar num bytes.
- O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro.
- Se não houver memória suficiente para alocar a memória requisitada a função **malloc()** retorna um ponteiro nulo.

Exemplo com malloc

```
 #include <stdio.h>
#include <stdlib.h>
int main (void) {
     int *p;
     int a;
      ...
     /* Determina o valor de a em algum lugar */
     p=(int *)malloc(a*sizeof(int));
     if (!p) {
                    printf ("** Erro: Memoria
Insuficiente**");
                    exit;
       }
      ...
     return 0;
}
```

Calloc

- A função **calloc()** também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc (unsigned int num, unsigned int size);
```

- A funçao aloca uma quantidade de memória igual a **num * size**, isto é, aloca memória suficiente para uma matriz de **num** objetos de tamanho **size**.
- Retorna um ponteiro **void *** para o primeiro byte alocado.

Calloc

- O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro.
- Se não houver memória suficiente para alocar a memória requisitada a função **calloc()** retorna um ponteiro nulo.
- Em relação a malloc, calloc tem uma diferença (além do fato de ter protótipo diferente): calloc inicializa o espaço alocado com 0.

Exemplo com calloc

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
  int *p;
  int a,i;
  ...
  p=(int *)calloc(a, sizeof(int));
  if (!p) {
       printf ("** Erro: Memoria Insuficiente**");
       exit;
  }
  ...
  for (i=0; i<a ; i++)
       p[i] = i*i;
  ...
  return 0;
}
```

Free

- Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária.
- Para isto existe a função **free()** cujo protótipo é:

```
void free (void *p);
```

Free

- Basta então passar para **free()** o ponteiro que aponta para o início da memória alocada.
- Mas você pode se perguntar: como é que o programa vai saber quantos bytes devem ser liberados?
- Ele sabe pois quando você alocou a memória, ele guardou o número de bytes alocados numa "tabela de alocação" interna.

Exemplo com free

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  int *p;
  int a;
  ...
  p= (int *)malloc(a*sizeof(int));
  if (!p) {
       printf ("** Erro: Memoria Insuficiente **");
       exit;
  }
  ...
  free(p);
  ...
  return 0;
}
```

Exemplo de alocação dinâmica de vetor

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

float *Alocar_vetor_real (int n) {
  float *v;         /* ponteiro para o vetor */
   /* verifica parametros recebidos */
  if (n < 1) {
       printf ("** Erro: Parametro invalido **\n");
        return (NULL);
   }
  /* aloca o vetor */
  v = (float *)calloc (n, sizeof(float));
  if (v == NULL) {
      printf ("** Erro: Memoria Insuficiente **");
      return (NULL);
  }
   /* retorna o ponteiro para o vetor */
   return (v);
}
```

Exemplo de alocação dinâmica de vetor

```
float *Liberar_vetor_real (float *v) {
   if (v == NULL)
        return (NULL);
  /* libera o vetor */
  free(v);
  /* retorna o ponteiro */
  return (NULL);
}
int main () {
  float *p;
  int a;
  ...
  p = Alocar_vetor_real (a);
  ...
  p = Liberar_vetor_real (p);
  return(0);
}
```

Exercício

- Escreva um programa C que recebe como parâmetro 3 valores inteiros: *a*, *b* e *c*.
  - O programa deve alocar dinamicamente dois vetores de tamanho *a*
    - o primeiro deve ter todos os elementos iguais a *b*.
    - o segundo deve ter todos os elementos iguais a *c*.
  - O programa deve imprimir o vetor resultante da multiplicação dos dois vetores

Alocação Dinâmica de Matrizes

- A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, ou seja é um ponteiro para ponteiro, o que é denominado *indireção múltipla*.
- A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro.

Alocação Dinâmica de Matrizes

- Um exemplo de implementação para matriz real bidimensional é fornecido a seguir.
- A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz.
- Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

Exemplo

```
#include <stdio.h>
#include <stdlib.h>
float **Alocar_matriz_real (int m, int n) {
  float **v;  /* ponteiro para a matriz */
  int   i;
  /* verifica parametros recebidos */
  if (m < 1 || n < 1) {
       printf ("** Erro: Parametro invalido **\n");
       return (NULL);
  }
   /* aloca as linhas da matriz */
  v = (float *)calloc (m, sizeof(float *)); // Vetor de m ponteiros para float
  if (v == NULL) {
       printf ("** Erro: Memoria Insuficiente **");
       return (NULL);
  }
  /* aloca as colunas da matriz */
  for ( i = 0; i < m; i++ ) {
       v[i] = (float *)calloc (n, sizeof(float)); // m vetores de n floats
       if (v[i] == NULL) {
              printf ("** Erro: Memoria Insuficiente **");
              return (NULL);
        }
    }
  return (v);
}
```

Exemplo

```
float **Liberar_matriz_real (int m, int n, float **v) {
  int  i;
  if (v == NULL)
       return (NULL);
  /* verifica parametros recebidos */
   if (m < 1 || n < 1) {
      printf ("** Erro: Parametro invalido **\n");      \
      return (v);
  }
  for (i=0; i<m; i++)
       /* libera as linhas da matriz */
       free (v[i]);
  /* libera a matriz (vetor de ponteiros) */
  free (v);
  /* retorna um ponteiro nulo */
  return (NULL);
}
```

Exemplo

```
int main (void) {
  float **mat;  /* matriz a ser alocada */
  int   l, c;   /* numero de linhas e colunas da
  matriz */
  int i, j;
  ...
  mat = Alocar_matriz_real (l, c);
  for (i = 0; i < l; i++)
      for ( j = 0; j < c; j++)
            mat[i][j] = i+j;
  ...
  mat = Liberar_matriz_real (l, c, mat);
  ...
  return(0)
}
```

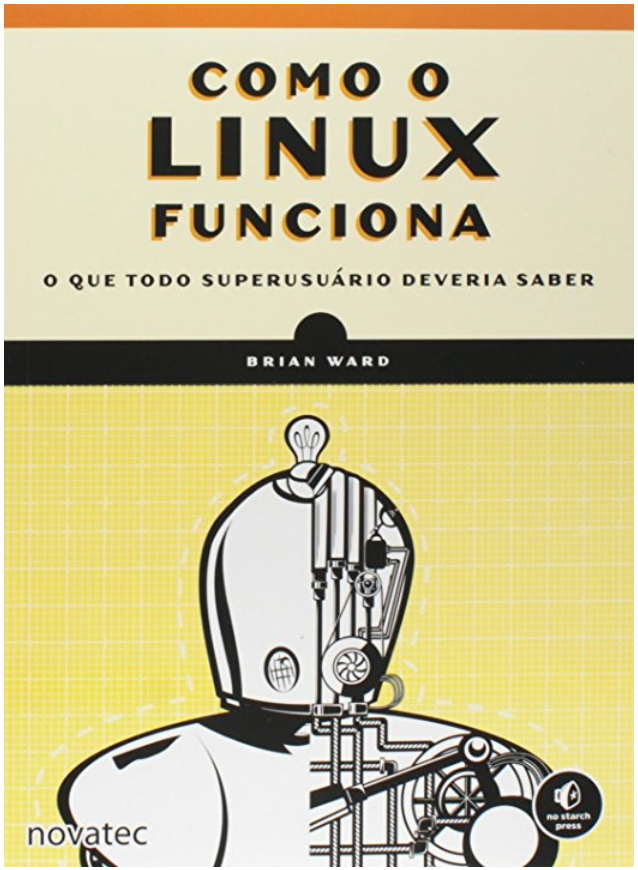
Exercício

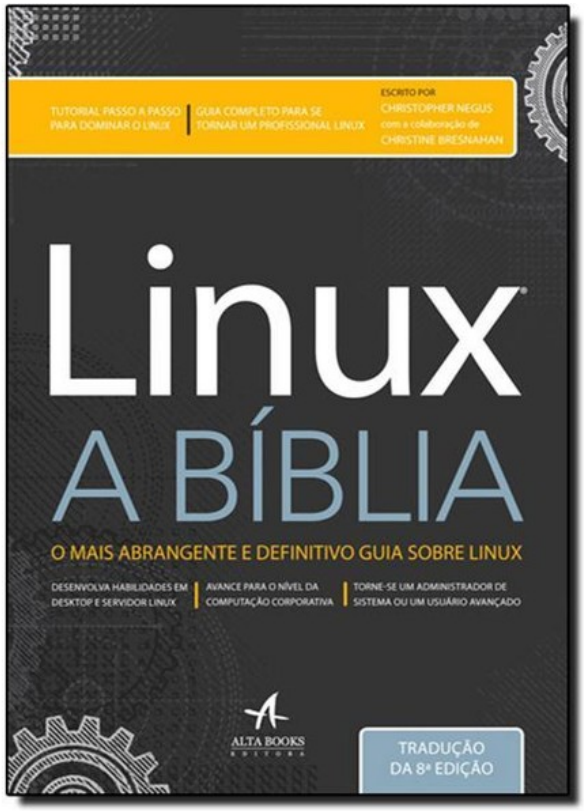
- Escreva um programa C que recebe como parâmetro 3 valores inteiros: *a*, *b* e *c*.
  - O programa deve alocar dinamicamente duas MATRIZES quadradas de tamanho *a*
    - a primeira deve ter todos os elementos iguais a *b*.
    - a segunda deve ter todos os elementos iguais a *c*.
  - O programa deve imprimir a matriz resultante da multiplicação das duas matrizes

Referências

- WARD, Brian; **Como o Linux Funciona:** o que todo superusuário deveria saber. 1. ed. São Paulo: Novatec, 2015.
- NEGUS, Christopher. **Linux a Bíblia**. 1. ed. São Paulo: Alta Books, 2014.

Referências





Próxima Aula

- Gerência de Processos: Conceitos