

# Sistemas Operacionais II

Técnicas Básicas para programação  
em GNU/LINUX



# Interação com o Ambiente de Execução

- Parâmetros de **main**
  - argc
  - argv

# Lista de Argumentos

- Quando um programa é chamado pela shell, a **lista de argumentos** contém a linha toda de comando que foi executada:
  - nome do programa
  - argumentos
- Exemplo:
  - `ls -s /`

# Lista de Argumentos

- A função **main** pode acessar a lista de argumentos através dos parâmetros **argc** e **argv**
  - **argc**: inteiro que contém o número de itens na lista de argumentos
  - **argv**: vetor de ponteiros de caracteres, cujo tamanho é *argc* e seus elementos apontam para os elementos da lista de argumentos (*strings* terminadas em NUL)

# Uso de argc e argv

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

```
int main (int argc, char* argv[])
```

```
{
```

```
    printf ("O nome do programa é '%s'.\n", argv[0]);
```

```
    printf ("Este programa foi chamado com %d argumentos.\n", argc - 1);
```

```
    /* Algum argumento de linha de comando foi especificado? */
```

```
    if (argc > 1) {
```

```
        /* Sim, imprima-os. */
```

```
        int i;
```

```
        printf ("Os argumentos são:\n");
```

```
        for (i = 1; i < argc; ++i)
```

```
            printf (" %s\n", argv[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

*arglist.c*

# Convenções de linha de comando no GNU/Linux

- Quase todos os programas do GNU/Linux seguem algumas convenções nos argumentos da linha de comando
- Argumentos que os programas esperam estão em duas categorias:
  - Opções (ou *flags*)
    - Modificam comportamentos
  - Outros argumentos
    - Oferecem entradas. Por exemplo: nomes de arquivos

# Opções

- Tem duas formas:
  - Opções curtas
    - Um único hífen e um único caractere (letra maiúscula ou minúscula)
    - Mais fáceis de digitar
  - Opções longas
    - Dois hifens, seguido de um nome feito de letras maiúsculas, minúsculas e hifens
    - Mais fáceis lembrar
    - Mais fáceis de ler (em scripts, por exemplo)

# Opções

- Em geral, os programas oferecem as duas formas:
  - Exemplo: -h e --help
- Algumas opções requerem um argumento em seguida
  - Exemplo: --output helloworld
  - ls -s /
  - ls --size /



# Usando *getopt\_long*

- *getopt\_long* faz a análise gramatical (*parse*) dos parâmetros
  - Torna esta tarefa menos tediosa
  - Aceita as formas curtas e longas das opções
  - Para usá-la inclua o arquivo de cabeçalho `<getopt.h>`

# Usando *getopt\_long*

- Para usar *getopt\_long* você deve fornecer duas estruturas
  - String de caracteres onde cada caractere é uma opção válida
    - Uma opção que requer um argumento é seguida de dois pontos (:)
      - Exemplo: *ho:v* indica que as opções válidas são -h, -o, e -v, onde a segunda recebe um argumento
  - Conjunto de elementos *struct options* (próximo slide)

# Usando *getopt\_long*

- Cada elemento é uma opção longa e tem 4 campos:
  - nome da opção longa (string de caracteres sem os dois hifens)
  - 1 caso a opção tenha argumento, ou 0 caso contrário
  - NULL
  - Caractere constante indicando a opção curta sinônimo da opção longa
- O último elemento do conjunto deve ter zero (ou NULL) em todos os campos.

```
const struct option long_options[]  
= {  
    {'help',           0, NULL, 'h'},  
    {'output',         1, NULL, 'o'},  
    {'verbose',        0, NULL, 'v'},  
    {NULL,             0, NULL, 0}  
};
```

# Usando *getopt\_long*

- *getopt\_long* é chamado com os seguintes parâmetros:
  - argc
  - argv
  - string de opções curtas
  - conjunto de estruturas de opções longas

# Usando *getopt\_long*

- Cada vez que você chama *getopt\_long*, ele faz a análise gramatical de uma única opção, retornando a letra da opção curta, ou -1 caso não haja mais opções
- Tipicamente, você deve chamá-lo em um loop, para processar todas as opções que o usuário especificou
- Se encontra uma opção inválida, imprime uma mensagem de erro e retorna ? (ponto de interrogação)
  - A maioria dos programas termina quando isso ocorre, mostrando informações de uso
- Quando termina de analisar todas as opções, a variável global *optind* contém o índice (em *argv*) do primeiro argumento que não é opção

## getopt\_long.c

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

/* O nome deste programa. */
const char* program_name;

/* Imprime as informações de uso do programa para o STREAM (tipicamente
   stdout ou stderr), e sai do programa com o código EXIT_CODE. Não retorna. */

void print_usage (FILE* stream, int exit_code)
{
    fprintf (stream, "Uso: %s opções [ arquivoentrada ... ]\n",
             program_name);
    fprintf (stream,
            "  -h --help           Mostra estas informações de uso.\n"
            "  -o --output filename Escreve saída para um arquivo.\n"
            "  -v --verbose        Imprime mensagens verbosas.\n");
    exit (exit_code);
}

/* Ponto de entrada do programa principal. ARGV contém o números de elementos na
   lista de argumentos; ARGV é um conjunto de ponteiros para eles. */

int main (int argc, char* argv[])
{
    int next_option;

    /* Uma string listando letras de opções curtas válidas. */
    const char* const short_options = "ho:v";
    /* Um conjunto descrevendo opções longas válidas. */
    const struct option long_options[] = {
        { "help", 0, NULL, 'h' },
        { "output", 1, NULL, 'o' },
        { "verbose", 0, NULL, 'v' },
        { NULL, 0, NULL, 0 } /* Requerido no final do conjunto. */
    };
};
```

```

/* O nome do arquivo que receberá a saída do programa , ou NULL
para a saída padrão. */
const char* output_filename = NULL;
/* Mostrar ou não mensagens verbosas. */
int verbose = 0;

/* Lembrar o nome do programa, para incorporá-lo nas mensagens.
O nome está armazenado em argv[0]. */
program_name = argv[0];

do {
    next_option = getopt_long (argc, argv, short_options,
                                long_options, NULL);
    switch (next_option)
    {
        case 'h': /* -h ou --help */
            /* Usuário requisitou informações de uso. Imprima para a saída
            padrão, e então saia com o código de saída zero (terminação normal).
            */
            print_usage (stdout, 0);

        case 'o': /* -o ou --output */
            /* Esta opção tem um argumento, o nome do arquivo. */
            output_filename = optarg;
            break;

        case 'v': /* -v ou --verbose */
            verbose = 1;
            break;
    }
}

```

```

case '?': /* O usuário especificou uma opção inválida. */
    /* Imprima informação de uso do erro padrão, e saia com o código de
    saída um (indicando terminação anormal). */
    print_usage (stderr, 1);

case -1: /* Fim das opções. */
    break;

default: /* Algo mais: inesperado. */
    abort ();
}
while (next_option != -1);

/* Fim das opções. OPTIND aponta para o primeiro argumento que não é
opção. Para efeito de demonstração, imprima-os se a opção verbose foi
especificada. */
if (verbose) {
    int i;
    for (i = optind; i < argc; ++i)
        printf ("Argumentos: %s\n", argv[i]);
}

/* O programa principal vai aqui. */

return 0;
}

```

# Fluxos de Entrada/Saída Padrão

- A biblioteca C padrão fornece fluxos de entrada e saída (*stdin* e *stdout*)
  - Usados por *scanf*, *printf*, etc.
  - Uso das entradas e saída padrão seguem a tradição do UNIX, permitindo o uso de *pipes* e redirecionamento de saída



# Fluxos de Entrada/Saída Padrão

- Fluxo de erro padrão (*stderr*)
  - Permite separar mensagens de aviso e de erro da saída normal do programa
    - Exemplo: você pode redirecionar a saída do programa para um arquivo, mas continuar vendo as mensagens de erro no console
  - Uso:
    - `fprintf(stderr, ("Erro: ..."));`
  - Acessíveis com os comandos de I/O do UNIX (`read`, `write`, etc.) através de descritores de arquivos
    - 0 para *stdin*, 1 para *stdout*, 2 para *stderr*

# Fluxos de Entrada/Saída Padrão

- Redirecionar ambas as saídas (saída padrão e erro padrão) para arquivo ou *pipe* no Bash:
  - `program > output_file.txt 2>&1`
  - `program 2>&1 | filter`
    - Sintaxe `2>&1` indica que o descritor de arquivo 2 (stderr) deve ser fundido no descritor 1 (stdout)
      - Deve proceder um redirecionamento para arquivo
      - Deve preceder um redirecionamento por pipe

# Fluxos de Entrada/Saída Padrão

- **stdout** tem um buffer
  - Saída não vai para o console até que o buffer esteja cheio, **stdout** seja fechado, ou o programa termine normalmente
  - Você pode forçar o esvaziamento do buffer com
    - `fflush(stdout)`
- **stderr** não tem buffer
  - Saída vai diretamente para o console

# Exercício 1

- A. Implemente os seguintes trechos de código em dois programas diferentes e veja qual é a diferença entre as saídas obtidas:

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    while(1)
    {
        printf(".");
        usleep(10000);
    }
}
```

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    while(1)
    {
        fprintf(stderr, ".");
        usleep(10000);
    }
}
```

- B. Como fazer o primeiro código ter o mesmo comportamento do segundo **acrescentando** apenas uma linha de código?

# Código de Saída de Programas

- Quando um programa termina, ele indica seu status com um código de saída
  - 0 indica execução com sucesso
  - Outro número indica que ocorreu um erro
    - Alguns programas utilizam diferentes valores diferentes de zero para indicar qual erro ocorreu
- Na maioria das *shells* obtém-se o código de saída do último programa executado com a variável **\$?**
  - echo \$?

# Código de Saída de Programas

- Exemplo: execute `echo $?` após cada um dos comandos abaixo:
  - `ls /`
  - `ls arquivoquenaoexiste`
- Em C ou C++, o código de saída é o código que é retornado pela função **main**
- Outros métodos de retornar códigos de saída e códigos de saída especiais de programas terminados de forma anormal (por sinais) serão vistos mais tarde no curso

# O Ambiente

- O GNU/Linux oferece a todo programa em execução um conjunto de variáveis de ambiente
  - Pares variável/valor onde ambos são *strings* de caracteres
  - Por convenção, nomes de variáveis de ambiente são escritos com todas as letras maiúsculas

# O Ambiente

- Algumas variáveis de ambiente comuns:
  - **USER** contém seu nome de usuário
  - **HOME** contém o caminho para seu diretório *home*
  - **PATH** contém uma lista de diretórios separados por dois pontos na qual o Linux procura os comandos a serem executados
  - **DISPLAY** contém o nome e número de tela do servidor *X Window* no qual as janelas de programas gráficos aparecerão

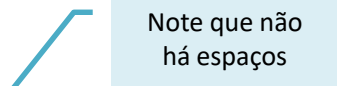


# O Ambiente

- O shell também tem um ambiente (como qualquer outro programa)
- O comando *printenv* mostra as variáveis de ambiente do shell
- Para acessar variáveis de ambiente use `$varname`
  - Exemplos:
    - `echo $USER`
    - `echo $HOME`

# O Ambiente

- Para colocar uma variável do shell no ambiente
  - Exemplo:
    - EDITOR=emacs
    - export EDITOR
  - Ou:
    - export EDITOR=emacs



Note que não  
há espaços

# O Ambiente

- Em programas, utilize a função **getenv** de `<stdlib.h>`, que recebe um nome de variável e retorna uma string com seu valor, ou NULL, se tal variável não estiver definida no ambiente
- Para configurar uma variável de ambiente use **setenv**
- Para limpar uma variável de ambiente use **unsetenv**

# O Ambiente

- Imprimindo o ambiente de execução

```
#include <stdio.h>

/* A variável ENVIRON contém o ambiente. */
extern char** environ;

int main ()
{
    char** var;
    for (var = environ; *var != NULL; ++var)
        printf ("%s\n", *var);
    return 0;
}
```

*print-env.c*

# O Ambiente

- Quando um novo programa é iniciado, ele herda uma cópia do ambiente do programa que o invocou
  - O shell, caso tenha sido invocado iterativamente
  - Variáveis de ambiente são comumente usadas para comunicar informações de configurações aos programas

# O Ambiente

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

*client.c*

```
int main ()
{
    char* server_name = getenv ("SERVER_NAME");
    if (server_name == NULL)
        /* A variável de ambiente SERVER_NAME não foi configurada. Use o
        padrão. */
        server_name = "server.my-company.com";

    printf ("accessando servidor %s\n", server_name);
    /* Acesse o servidor aqui... */

    return 0;
}
```

Parte de um programa cliente de rede

# Usando arquivos temporários

- As vezes um programa precisa de um arquivo temporário para guardar grandes quantidades de dados por um período ou para passá-los a outro programa
- Em sistemas GNU/Linux arquivos temporários ficam no diretório /tmp

# Usando arquivos temporários

- Cuidados ao usar arquivos temporários:
  - Mais de uma instância do programa pode executar ao mesmo tempo (pelo mesmo usuário ou usuários diferentes). Cada instância deve ter diferentes nomes de arquivos temporários para que não haja colisão.
  - As permissões do arquivo temporário devem ser configuradas de forma que usuários não autorizados não possam alterar a execução do programa ao modificar ou substituir arquivos temporários.
  - Nomes de arquivos temporários devem ser gerados de forma que não possam ser preditos externamente; caso contrário, um invasor poderia explorar a latência entre o teste para verificar se um dado nome já está em uso e a abertura de um novo arquivo temporário



# Usando arquivos temporários

- O GNU/Linux oferece as funções, **mkstemp** e **tmpfile**, que cuidam dessas limitações para você
  - mkstemp
    - Cria um nome de arquivo temporário único a partir de um modelo, cria o arquivo com permissões que só o usuário atual possa acessá-lo e abre o arquivo para leitura/escrita.
    - Template: string de caracteres seguido de XXXXXX, onde cada X é substituído por um caractere para que o nome seja único
    - O valor de retorno é um descritor de arquivo, que pode ser usado com as funções de escrita.

# Usando arquivos temporários

- Arquivos criados com `mkstemp` não são excluídos automaticamente, o programador deve fazê-lo, caso contrário o arquivo `/tmp` ficará cheio
  - Se o arquivo não será usado por outros programas, convém chamar *unlink* no arquivo temporário imediatamente.
    - Isto removerá a entrada de diretório, mas o arquivo não será excluído enquanto estiver aberto. Será excluído automaticamente quando for fechado, mesmo que o programa seja terminado de forma anormal.

# Usando arquivos temporários

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

/* Um manipulador para um arquivo temporário criado com
write_temp_file. Nesta implementação, é apenas um descritor de
arquivo. */
typedef int temp_file_handle;

/* Escreve LENGTH bytes do BUFFER em um arquivo temporário. O
arquivo temporário é desvinculado imediatamente. Retorna um
manipulador para o arquivo temporário. */

temp_file_handle write_temp_file (char* buffer, size_t length)
{
    /* Cria nome de arquivo e arquivo. O XXXXXX será substituído por
    caracteres que tornem o nome de arquivo único. */
    char temp_filename[] = "/tmp/temp_file.XXXXXX";
    int fd = mkstemp (temp_filename);
    /* Desvincula o arquivo imediatamente, de forma que ele seja removido
    quando o descritor de arquivo for fechado. */
    unlink (temp_filename);
    /* Escreve o número de bytes do arquivo primeiro. */
    write (fd, &length, sizeof (length));
    /* Agora escreve os dados. */
    write (fd, buffer, length);
    /* Usa o descritor de arquivos como manipulador para o arquivo
    temporário. */
    return fd;
}
```

```
/* Lê o conteúdo de um arquivo temporário TEMP_FILE criado como
write_temp_file. O valor de retorno é um novo buffer alocado daquele
conteúdo, o qual o chamador precisa desalocar com free.
*LENGTH é configurado com o tamanho do conteúdo, em bytes. O
arquivo temporário é removido. */

char* read_temp_file (temp_file_handle temp_file, size_t*
length)
{
    char* buffer;
    /* O manipulador TEMP_FILE é um descritor de arquivo para o arquivo
    temporário. */
    int fd = temp_file;
    /* Volta para o início do arquivo. */
    lseek (fd, 0, SEEK_SET);
    /* Lê o tamanho dos dados no arquivo temporário. */
    read (fd, length, sizeof (*length));
    /* Aloca um buffer e lê os dados. */
    buffer = (char*) malloc (*length);
    read (fd, buffer, *length);
    /* Fecha o descritor de arquivos, o qual irá fazer com que o arquivo
    temporário desapareça. */
    close (fd);
    return buffer;
}
```

*temp\_file.c*

# Uso de arquivos temporários

- **tmpfile**

- Pode ser usada se você estiver usando as bibliotecas de entrada e saída do C, e não precisa passar arquivos temporários para outros programas.
- Cria um arquivo temporário e retorna um ponteiro para ele.
- O arquivo temporário já é desvinculado, como no exemplo anterior, portanto é excluído automaticamente quando o ponteiro de arquivo é fechado (fclose) ou quando o programa termina.

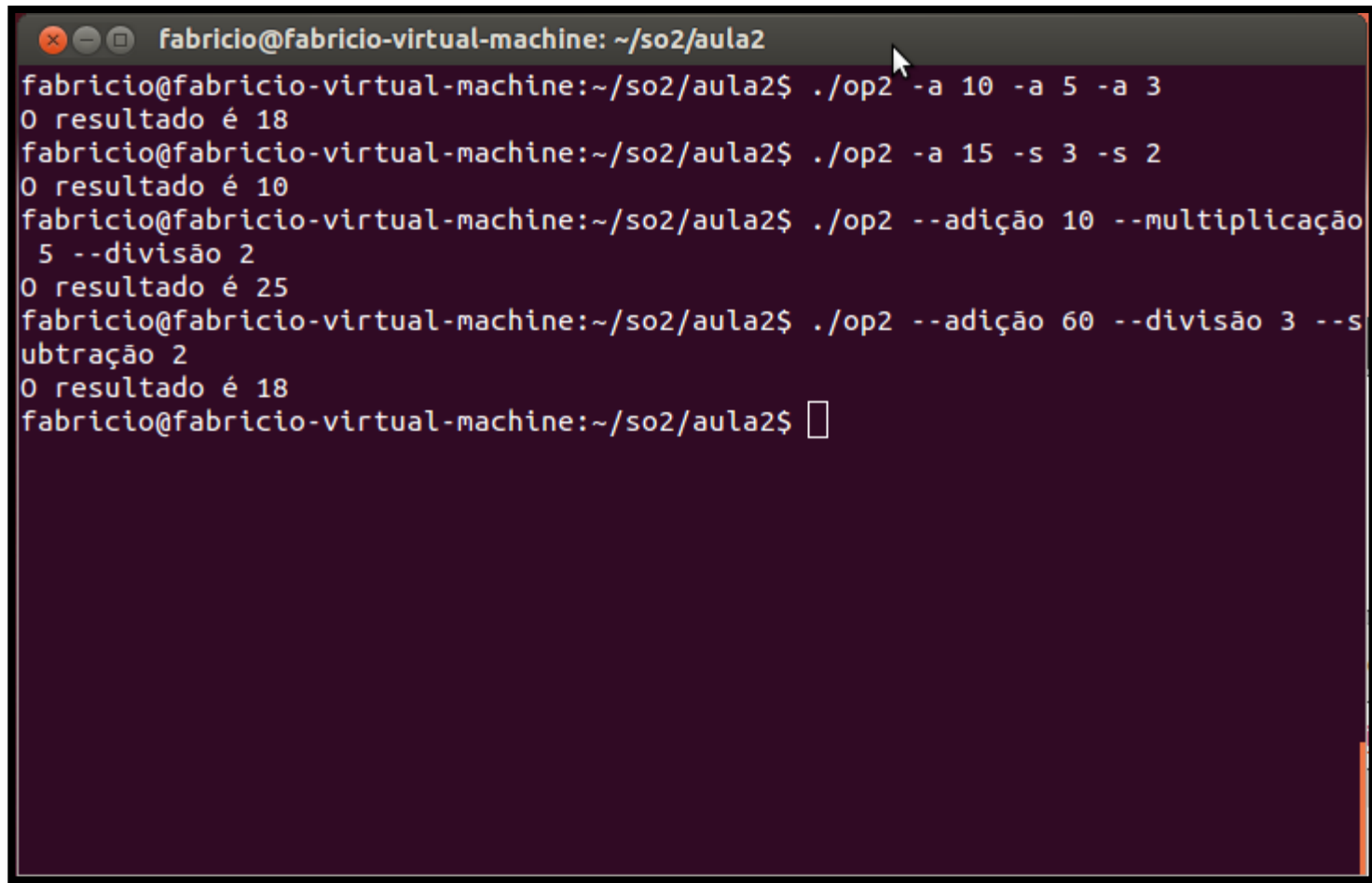
# Exercício 2

- Faça um programa que aceite como parâmetros:
  - -a -s -m -d      --adição --subtração --multiplicação --divisão
- O programa:
  - deve aceitar números naturais como argumentos de cada opção e aplicar a operação escolhida.
  - deve incluir uma opção de ajuda.
  - não deve permitir divisão por zero.
  - deve exibir mensagens de erro no fluxo de erro e resultados no fluxo de saída padrão.
  - deve usar o código de retorno 0 quando a execução for bem-sucedida e valores negativos para erros.

# Exercício 2

- Exemplos de Uso:
  - `./calc -a 10 -a 5 -a 3`  
O resultado é 18.
  - `./calc -a 15 -s 3 -s 2`  
O resultado é 10.
  - `./calc --adição 10 --multiplicação 5 --divisão 2`  
O resultado é 25.
  - `./calc --adição 60 --divisão 3 --subtração 2`  
O resultado é 18.

# Exercício 2



```
fabricio@fabricio-virtual-machine: ~/so2/aula2
fabricio@fabricio-virtual-machine:~/so2/aula2$ ./op2 -a 10 -a 5 -a 3
O resultado é 18
fabricio@fabricio-virtual-machine:~/so2/aula2$ ./op2 -a 15 -s 3 -s 2
O resultado é 10
fabricio@fabricio-virtual-machine:~/so2/aula2$ ./op2 --adição 10 --multiplicação
5 --divisão 2
O resultado é 25
fabricio@fabricio-virtual-machine:~/so2/aula2$ ./op2 --adição 60 --divisão 3 --s
ubtração 2
O resultado é 18
fabricio@fabricio-virtual-machine:~/so2/aula2$
```

# Referências Bibliográficas

1. [MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex; \*Advanced Linux Programming\*. New Riders Publishing: 2001. Cap. 2.1](#)

