Instituto Superior de Engenharia

Politécnico de Coimbra

Sistemas Operativos

CTeSP Tecnologias e Programação de Sistemas de Informação (Cantanhede)

Professor: João Leal

joao.leal@isec.pt

UNIgreen The green European University

PRR Plano de Recuperação e Resiliência

REPÚBLICA PORTUGUESA

Financiado pela União Europeia NextGenerationEU

Transformar comandos em *script*

Instituto Superior de Engenharia
Politécnico de Coimbra

- Podemos executar comandos a partir do shell, mas também é possível inserir comandos em um arquivo e depois configurá-lo para ser executável.

- Quando executamos o arquivo, os comandos são rodados um após o outro. Estes arquivos executáveis são chamados scripts e representam uma ferramenta absolutamente essencial para qualquer administrador de sistema Linux.

Transformar comandos em *script*

Instituto Superior de Engenharia
Politécnico de Coimbra

- Podemos dizer que o Bash, além de um shell, é uma verdadeira linguagem de programação.

```
$ echo "Hello World!"
Hello World!
```

- Utilizando um redirecionamento de arquivo para enviar este comando para um novo arquivo **`new_script`**.

```
$ echo 'echo "Hello World!"' > new_script
$ cat new_script
echo "Hello World!"
```

Transformar comandos em *script*

Instituto Superior de Engenharia
Politécnico de Coimbra

- Podemos simplesmente digitar o nome do script, assim como qualquer outro comando

```
$ new_script
/bin/bash: new_script: command not found
```

- Sabemos que **new_script** existe na nossa localização atual, mas a mensagem de erro não diz que o arquivo não existe, mas sim que o comando não existe.

Comandos e Path

- Quando digitamos o comando ls no shell, por exemplo, estamos na verdade a executar um arquivo chamado ls que existe no nosso sistema. Para comprovar, use which:

```
$ which ls
/bin/ls
```

Comandos e Path

- O Bash tem uma variável de ambiente que contém todos os diretórios em que ficam os comandos que queremos executar.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games:/usr/local/games:/sn
ap/bin
```

- O shell espera encontrar um comando em cada um desses locais, delimitado por dois pontos (:).

Comandos e Path

- Note-se que /bin está presente, mas é seguro supor que nossa localização atual não está. O Shell procurará por **new_script** em cada um desses diretórios, mas **não o encontrará** e, portanto, **exibirá a mensagem de erro**.

- Existem três soluções para esse problema:
  - podemos mover **new_script** para um dos diretórios de PATH,
  - podemos adicionar o diretório atual a PATH,
  - podemos mudar a maneira de chamar o script.

Comandos e Path

- A última solução (mudar a maneira de chamar o script) é a mais fácil, pois simplesmente requer que especifiquemos o local atual ao chamar o script usando ponto e barra (./).

```
$ ./new_script
/bin/bash: ./new_script: Permission denied
```

- A mensagem de erro já está diferente, o que indica que fizemos alguns progressos.

Permissões de execução

- A primeira investigação que um utilizador deve fazer é usar **ls -l** para examinar o arquivo:

```
$ ls -l new_script
-rw-rw-r-- 1 user user 20 May 30 12:12 new_script
```

Permissões de execução

- Podemos ver que as permissões para este arquivo estão definidas como 664 por padrão. Ainda não configuramos este arquivo para ter permissões de execução.

```
$ chmod +x new_script
$ ls -l new_script
-rwxrwxr-x 1 user user 20 May 30 12:12 new_script
```

Permissões de execução

- Este comando concede permissões de execução a todos os utilizadores.

- Atenção que pode ser um risco de segurança, mas, por enquanto, este é um nível aceitável de permissão.

```
$ ./new_script
Hello World!
```

- Agora já é possível executar nosso script.

Definir o intérprete

- É possível simplesmente inserir texto em um arquivo, configurá-lo como executável e executá-lo.

- O **new_script** ainda é tecnicamente um arquivo de texto normal, mas pode ser interpretado pelo Bash. Mas e se ele estiver escrito em Perl ou Python?

- É extremamente recomendável especificar o tipo de intérprete que queremos usar na primeira linha de um script.

Definir o intérprete

- Essa linha é chamada de **bang line**, ou, **shebang**. Indica ao sistema como queremos que aquele arquivo seja executado.

- Em Bash, usamos o caminho absoluto para o nosso executável do Bash (utilizando ***which***):

```
$ which bash
/bin/bash
```

Definir o intérprete

- O **shebang** começa com um sinal de **hash e um ponto de exclamação**, seguidos pelo caminho absoluto.

- Vamos abrir **new_script** num editor de texto e inserir o **shebang**.

- Aproveitamos também a oportunidade para inserir um comentário em nosso script.

Definir o intérprete

- Os comentários são ignorados pelo intérprete. Eles são escritos para o benefício de outros utilizadores que queiram entender o script.

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
echo "Hello World!"
```

Definir o intérprete

- Podemos fazer uma alteração adicional no nome do arquivo: salvando o arquivo como **new_script.sh**.

- O sufixo do arquivo ".sh" não altera em nada a execução do arquivo. É uma convenção que os scripts do bash sejam rotulados com .sh ou .bash para identificá-los mais facilmente, assim como os scripts em Python são geralmente identificados com o sufixo .py.

Variáveis

- As variáveis são uma parte importante de qualquer linguagem de programação, e para o Bash também.

- Quando iniciamos uma nova sessão no terminal, o shell já define algumas variáveis automaticamente (exemplo: a variável PATH).

- Chamamos a estas variáveis, variáveis de ambiente, porque geralmente definem características do nosso ambiente shell.

Variáveis

- É possível modificar e adicionar variáveis de ambiente, mas por enquanto vamos configurar as variáveis dentro do nosso script. Vamos modificar o script da seguinte forma:

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=Joao
echo "Olá $username!"
```

Variáveis

- Neste caso, criamos uma variável chamada **username** e atribuímos-lhe o valor Joao.

- Note que não há espaços entre o nome da variável, o sinal de igual ou o valor atribuído.

Variáveis

- Na linha seguinte, usamos o comando echo com a variável, mas há um cifrão ($) na frente do nome da variável. Isso é importante, pois indica para o shell que queremos tratar **username** como uma variável, e não apenas como uma palavra normal.

- Ao digitar **$username** no comando, indicamos que queremos executar uma substituição, trocando o nome de uma variável pelo valor atribuído a essa variável.

Variáveis

- Ao executar o novo script, obtemos esta saída:

```
$ ./new_script.sh
Olá Joao!
```

- As variáveis devem conter apenas caracteres alfanuméricos ou sublinhados (underline) e diferenciam maiúsculas de minúsculas.

- A substituição de variáveis também pode ter o formato ${username}, com a adição de { }.

Variáveis

- As variáveis no Bash têm um tipo implícito, sendo consideradas strings.

- Isso significa que executar funções matemáticas no Bash é mais complicado do que seria noutras linguagens de programação, como C/C++.

Variáveis


```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=Joao
x=2
y=4
z=$x+$y
echo “Olá $username!"
echo "$x + $y"
echo "$z"
```

Variáveis


```
$ ./new_script.sh
Olá Joao!
2 + 4
2+4
```

Variáveis

- Vamos fazer a seguinte alteração no valor da nossa variável **username**:

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=Joao Leal
echo “Olá $username!"
```

Variáveis

- A execução desse script resultará num erro:

```
$ ./new_script.sh
./new_script.sh: line 5: Leal: command not found
Hello !
```

Variáveis

- Lembre-se de que o Bash é um intérprete e, como tal, interpreta o script linha por linha.

- Neste caso, ele interpretou corretamente username=Joao como a definição de uma variável username com o valor Joao. Mas, em seguida, ele interpretou que o espaço indicava o final dessa tarefa e que Leal era o nome de um comando.

Variáveis

- Para que o espaço e o nome Leal sejam incluídos como o novo valor da variável, colocamos aspas duplas (") no nome.

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username="Joao Leal"
echo "Olá $username!"

$ ./new_script.sh
Olá Joao Leal!
```

Variáveis

- Uma coisa importante a ser observada no Bash é que aspas duplas e aspas simples (') se comportam de maneira muito diferente.

- As **aspas duplas são consideradas "fracas"** porque permitem que o intérprete efetue a substituição dentro das aspas.

- **Aspas simples são consideradas "fortes"** por impedirem a substituição.

Variáveis

- Considere o seguinte exemplo:

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=“Joao Leal"
echo “Olá $username!"
echo ‘Olá $username!'
$ ./new_script.sh
Olá Joao Leal!
Olá $username!
```

Variáveis

- No segundo comando echo, o intérprete foi impedido de substituir $username por Joao Leal, e assim a saída é interpretada literalmente.

Argumentos

- Os argumentos podem ser passados para o script na execução e modificam o comportamento do script. É fácil implementá-los.

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=$1
echo “Olá $username!"
```

Argumentos

- Em vez de atribuir um valor a username diretamente dentro do script, atribuímos o valor de uma nova variável $1. Ela faz referência ao valor do primeiro argumento.

```
$ ./new_script.sh Joao
Olá Joao!
```

- Os nove primeiros argumentos são tratados desta maneira.

Argumentos

- Eis um exemplo usando apenas dois argumentos:

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é
recomendável documentar todos os scripts.
username1=$1
username2=$2
echo “Olá $username1 e $username2!“

$ ./new_script.sh Joao Joana
Olá Joao e Joana!
```

Argumentos

- Existe uma consideração importante ao usar argumentos: no exemplo, temos dois argumentos, Joao e Joana, atribuídos a $1 e $2, respectivamente. Se o segundo argumento estiver ausente, por exemplo, o shell não emitirá uma mensagem de erro. O valor de $2 será simplesmente nulo, ou nada.

```
$ ./new_script.sh Joao
Olá Joao e !
```

Argumentos

- Seria recomendável introduzir alguma lógica no script para que diferentes condições afetem a saída que desejamos imprimir.

- Começaremos introduzindo outra variável útil e, em seguida, criaremos declarações ***if***.

Retorno de Argumentos

- Enquanto variáveis como $1 e $2 contêm o valor dos argumentos posicionais, a variável $# contém o número de argumentos.

```
#!/bin/bash
# Este é nosso primeiro comentário. Também é recomendável documentar todos os scripts.
username=$1
echo "Olá $username!"
echo "Número de Argumentos: $#."
```

```
$ ./new_script.sh Joao Joana
Olá Joao!
Número de Argumentos: 2.
```

Lógica Condicional

- Iremos apenas abordar a sintaxe dos condicionais no Bash, que é diferente da maioria das outras linguagens de programação.

- Primeiro, vamos rever nosso objetivo. Temos um script simples capaz de imprimir uma saudação para um único utilizador. Se houver algo diferente de um único utilizador, temos de imprimir uma mensagem de erro.

Lógica Condicional

- A condição que estamos a testar é o número de utilizadores, que está contido na variável $#.

- Gostaríamos de saber se o valor de $# é 1.

- Se a **condição for verdadeira**, a ação executada será saudar o utilizador.

- Se a **condição for falsa**, imprimiremos uma mensagem de erro.

Lógica Condicional

- Sintaxe:

```
#!/bin/bash
# Um script simples para saudar um único utilizador.
if [ $# -eq 1 ]
then
     username=$1
      echo "Olá $username!"
else
      echo "Escreva, por favor, um argumento."
fi
echo "Número de Argumentos: $#."
```

Lógica Condicional

- A lógica condicional está contida entre **if** e **fi**. A condição a ser testada fica entre **[ ]** e a ação a tomar caso a condição seja verdadeira é indicada após **then**.

- Observe os espaços entre os **[ ]** e a lógica contida neles. A omissão desses espaços causará erros.

- O script exibirá a saudação ou a mensagem de erro. Mas imprime sempre a linha **Number of arguments**.

Lógica Condicional


```
$ ./new_script.sh
Escreva, por favor, um argumento.
Número de Argumentos: 0.
$ ./new_script.sh Joao
Olá Joao!
Número de Argumentos: 1.
```

- Tome nota da declaração **if**. Usamos **-eq** para fazer uma comparação numérica. Nesse caso, testamos se o valor de $# é igual a um.

Lógica Condicional

- As outras comparações que podemos realizar são:
  - `-ne` Diferente de
  - `-gt` Maior que
  - `-ge` Maior que ou igual a
  - `-lt` Menos que
  - `-le` Menos que ou igual a

Gerir Utilizadores e Permissões

- No Linux, cada pessoa que usa o sistema tem um utilizador. A gestão de utilizadores permite:
  - Criar novos utilizadores;
  - Modificar os seus dados (nome, grupos, permissões);
  - Remover utilizadores que já não precisam de acesso;
  - Atribuir grupos, que organizam os utilizadores com funções semelhantes.

Apoio Tarefa Final

- As permissões controlam quem pode:
  - Ler (r), escrever (w) e executar (x) ficheiros ou programas;
  - A gestão é feita por **`chmod`**, **`chown`** e **`usermod`**.

Apoio Tarefa Final

Exemplos:

Criar utilizador

```
sudo useradd -m joana
sudo passwd joana
```

Modificar utilizador

```
sudo usermod -l novo_nome joana     # Mudar o nome de utilizador
sudo usermod -aG sudo joana         # Adicionar ao grupo sudo
```

Automatizar Tarefas com Bash Script

- O Bash é o terminal do Linux, e permite escrever scripts para:
  - Executar várias tarefas automaticamente, como backups, atualizações ou relatórios;
  - Agendar com cron, para correr a intervalos regulares (diário, semanal, etc.).

Apoio Tarefa Final

- Exemplo de tarefas:
  - Criar cópias de segurança de ficheiros importantes;
  - Verificar espaço em disco;
  - Gerar relatórios de utilizadores ativos.

Apoio Tarefa Final

Exemplos:

Backup diário de uma pasta

```
#!/bin/bash
DATA=$(date +%Y-%m-%d)
ORIGEM="/home/utilizador/documentos"
DESTINO="/home/utilizador/backups/backup_$DATA.tar.gz"
tar -czf $DESTINO $ORIGEM
echo "Backup criado em $DESTINO"
```

Apoio Tarefa Final

Exemplos:

Guardar como backup.sh e tornar executável:

```
chmod +x backup.sh
```

Agendar com cron

```
crontab -e
```

Programas Simples em C

- **Gestão de Processos**
  - Em C, podemos criar processos com `fork()`;
  - Um processo pai pode esperar pelo filho com `wait()` — isto é sincronização;

Exemplo: um programa cria um processo para calcular algo, e espera pelo fim.

Programas Simples em C

- **Manipulação de Ficheiros e Diretórios**
  - C permite abrir, ler, escrever e fechar ficheiros com funções do sistema:
    - `open()`, `read()`, `write()`, `close()`;
  - Também é possível alterar permissões ou criar diretórios.

Programas Simples em C

- **Receção e Tratamento de Sinais**
  - Sinais são mensagens que o sistema envia a processos:
    - Ex: **`SIGINT`** (Ctrl+C), **`SIGKILL`**, **`SIGTERM`**;
  - O programa pode tratar sinais para reagir de forma personalizada *(ex: não encerrar com Ctrl+C)*.

Apoio Tarefa Final

Exemplos: **Gestão de Processos**

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Filho: PID = %d\n", getpid());
    } else {
        wait(NULL); // Sincronização
        printf("Pai: processo filho terminou.\n");
    }
    return 0;
}
```

Apoio Tarefa Final

Exemplos:

Manipulação de Ficheiros

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

int main() {
    int fd = open("exemplo.txt", O_WRONLY | O_CREAT, 0644);
    write(fd, "Olá Linux!\n", 11);
    close(fd);
    return 0;
}
```

Apoio Tarefa Final

Simulador de Escalonamento de Processos

- Os sistemas operativos escalonam processos para decidir qual processo corre a seguir;

- Algoritmos de escalonamento:
  - **FCFS (First Come First Served)** – primeiro a chegar, primeiro a ser executado;
  - **SJF (Shortest Job First)** – o processo mais curto vai primeiro;
  - **Round Robin** – cada processo recebe uma fatia de tempo igual (com quantum);

Apoio Tarefa Final

- Um simulador é um programa que recria este comportamento em forma de exercício prático.

Apoio Tarefa Final

Exemplos:

FCFS, Round Robin, SJF (simplificado)

```
#include <stdio.h>

typedef struct {
    int id, tempo_chegada, tempo_execucao;
} Processo;
(…)
```

Apoio Tarefa Final


```
(…)
void FCFS(Processo p[], int n) {
    int tempo_atual = 0;
    for (int i = 0; i < n; i++) {
        if (tempo_atual < p[i].tempo_chegada)
            tempo_atual = p[i].tempo_chegada;
        printf("Processo %d começa em %d\n", p[i].id,
tempo_atual);
        tempo_atual += p[i].tempo_execucao;
    }
} (…)
```

Apoio Tarefa Final


```
(…)
int main() {
    Processo processos[] = {
        {1, 0, 5},
        {2, 2, 3},
        {3, 4, 1}    };
    int n = sizeof(processos)/sizeof(Processo);
    printf("=== FCFS ===\n");
    FCFS(processos, n);
    return 0;
}
```


Questions?