

**Licenciatura em Engenharia Informática, Faculdade de Ciências e Tecnologias**

**Universidade do Algarve**

**Ano letivo 2018/2019**

# **Sistemas Operativos**

## **Relatório Prático Guião 1: “Linguagem Bash”**

Docente:  
**Amine Berquia**

Aluno:  
**Guilherme “Matrix” Dias, nº 61237**

## Índice

Intro.....	3
Sistemas Operativos.....	3
O que são.....	3
Kernel.....	3
Bootloader.....	3
Compilação e portabilidade.....	4
If(!kernel).....	4
Kernel.end();.....	4
Componentes.....	4
Bash.....	6
Why Bash So Good.....	6
Mas o que é Bash mesmo?.....	7
Velocidade.....	7
A Quem Serve Saber Bash?.....	8
Exercícios.....	9
P1.....	10
A.....	10
B.....	11
C.....	12
D.....	13
E.....	15
P2.....	16
P3.....	17
P4.....	18
P5.....	19
P6.....	19
P7.....	20
Notas.....	24
Conclusão.....	25
Referências.....	25
Bibliografia.....	25

## Intro

Com este relatório pretende-se explicar por alto explicar conceitos sobre sistemas operativos, a maneira como funcionam, as suas componentes, como também um breve resumo sobre bash.

Para além disto será também resolvido o Guião 1, sendo explicadas detalhadamente as resoluções dos exercícios.

# Sistemas Operativos

## O que são

Sistemas operativos têm visivelmente vindo a evoluir e mudar drasticamente ao longo do tempo, mas o objetivo destes tem sempre sido exatamente o mesmo, por definição <sup>[1]</sup>:

“system software that manages computer hardware and software resources and provides common services for computer programs.”

## Kernel

O kernel serve, a um nível fundamental, principalmente para a o suporte de tudo o resto que o sistema possa ter necessidade de correr.

Este trata de coisas como alocação de memória, paging, segmentation, lidar com interrupts, etc.

Em todos os sistemas (para uso geral), existe um CPU. Porém nem todos os CPUs são iguais, têm arquiteturas diferentes, e, por vezes o modo de operar em CPUs da mesma arquitetura podem ser diferentes.

## Bootloader

O bootloader é o que, num cenário perfeito, trata das preparações exclusivas da arquitetura na qual o kernel vai correr, antes de este ser lançado. Por exemplo, os processadores x86 mais recentes todos começam por emular modo de 16-bits, para estabelecer compatibilidade <sup>[2]</sup>. O linux por exemplo é um sistema operativo que foi desenvolvido para ser portavel através de todas as arquiteturas desenhadas para servirem um uso generalizado, e por isto, para cada arquitetura (ou subset da mesma), tem que ser feito um bootloader, que depois de correr, irá passar execução ao entry point do kernel<sup>[3]</sup>.

## Compilação e portabilidade

Agora usando mais uma vez como exemplo o kernel do linux, que é programado totalmente em C, podemos concluir que, se houver compiladores para uma arquitetura e alguém disposto a programar um bootloader, o kernel do linux é “facilmente” portátil para qualquer arquitetura.

### If(!kernel)

Se não existisse um kernel, programas e aplicações teriam que se preocupar com alocar a sua própria memória, estabelecer o seu próprio meio de fazer multitasking, e muitas outras coisas difíceis, que tornariam o seu desenvolvimento não só perto de impossível, como também à partida incompatível com qualquer outro(a) que quisesse correr em simultâneo.

### Kernel.end();

A funcionalidade e propósito base do kernel é portanto, não comunicar com o hardware ou estabelecer uma interface com o utilizador, mas sim tirar partido de standards bem estabelecidos na comunidade do hardware (por exemplo paging e protection levels) para que outras aplicações (por exemplo firmware, ou, no outro lado do espectro, web browsers) possam correr sem terem que se preocupar com o seu meio.

## Componentes

Um sistema operativo é constituído sempre obrigatoriamente por um kernel, mas para além deste existem muitas outras componentes que são importantes, e, para uso comum, essenciais ao mesmo.

Visto que um kernel sozinho, por definição “não tem que fazer nada”, é preciso drivers (firmware) para comunicar com periféricos e outro hardware. Este é por isso o exemplo mais básico.

Estas componentes podem ou não ser considerados parte do kernel. O desenvolvedor do kernel e a maneira como é feito ditam se ou não o são.

Sem o firmware que vem agregado, os sistemas operativos não poderiam sequer comunicar com um utilizador de maneira nenhuma. Por isso existem, sem ir em muito detalhe, estes (como principais):

Como (maioritariamente) parte do kernel:

- [Program Execution](#)
- [Interrupts](#)
- [Modes](#)
- [Memory Management](#)
- [Virtual Memory](#)



- [Multitasking](#)
- [File System](#)
- [Device Drivers](#)

Como parte do resto do sistema operativo:

- [Networking](#)
- [UI](#)

Há na wiki também uma secção sobre “[segurança](#)”, mas eu acredito que (num cenário perfeito), o OS não deve por norma estar aberto a ameaças externas (não haver exploits, etc..), e que o utilizador deve ser responsável por não tomar decisões estúpidas (executar programas nos quais não tem confiança por exemplo). Se estes dois fatores fossem verdadeiros, não haveria maneira de um sujeito malicioso ter acesso ao sistema, e com isto, necessidade para “segurança”.

Acredito também (por prática e observação) que se algum dos dois não for verdadeiro, o sistema é extremamente fácil de comprometer, e dificilmente outro tipo de segurança não intrusivo consegue parar o que está a acontecer. E mesmo sendo intrusivo, há muitas maneiras de “hackar” à volta dessas medidas, e por isto tenho defendido sempre que senso comum e fazer os updates, combinados, são melhor que todos os antivirus à face da terra juntos.

## Bash

Bash é na minha opinião uma das ferramentas mais impressionantes criadas até à data.

Não pela sua versatilidade (que é também em si extremamente impressionante), mas pela sua consistência e usabilidade ao longo de vários sistemas operativos.

“Bash is a Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell.” [\[3\]](#)

### Why Bash So Good

Esta secção vai ser única e exclusivamente sobre a minha opinião.

Vamos começar por olhar para as alternativas do bash.

MAC-OS usa bash também por isso aí faz pouca diferença.

Windows usa o seu “Batch” (?) que é uma variante de DOS. Acho que quem já teve experiência com ambos, sabe que batch não se compara de todo a bash (tem muito menos funcionalidade).

Ou seja. Bash não tem muitas alternativas. Mas o seu standard é sempre o mesmo.

A maioria das linguagens que são usadas para criar aplicações (e que foram criadas primariamente para este propósito, ou pelo menos com ele em mente), têm sempre algum tipo de problema com a sua disponibilidade.

C++ tem problemas com portabilidade, em que entre linux, windows, e macos têm que ser usadas funções e bibliotecas diferentes, fazendo com que o mesmo código tenha que ser reescrito para cada um.

Python tem o problema das versões, em que saíram várias ao longo do tempo, e todo o código teve que ser portado para a versão nova, e muitas pessoas ainda consistentemente programam em versões antigas (2.7), visto que infelizmente ainda não foi descontinuado.

Bash porém, funciona em todos os “environments” em que é posto. É singular.

Código de bash que funciona em linux de ARM funcionará em MacOS de x86, e ninguém tem problemas nem desentendimentos.

## Mas o que é Bash mesmo?

Pode ser difícil de compreender o alcance de bash, dada a sua sintaxe e versatilidade.

Em c++, alguém escreve int ou double ou if, e faz sempre a mesma coisa. Faz parte do standard.

Em bash, a grande, grande maioria tudo o que se faz, não faz parte efetivamente da linguagem.

Penso que muita gente pense que (por exemplo) a keyword “ls” faça parte da linguagem.

Não faz.

O que é brilhante em relação a Bash é que o volume de ações que suporta, são scripts e programas.

Basicamente, bash tem as suas keywords e operadores, mas a maioria de operações que suporta, vêm do “\$PATH”.

O que é o \$PATH? É uma “environment variable”, onde estão listados todos os ficheiros que o Bash pode utilizar. Quando um programador faz um “ls”, o bash vai iterar através do PATH, até encontrar algum ficheiro chamado ls\*, e vai tentar executá-lo, passando-lhe os parâmetros.

Esse programa ls vai ter “dois returns”. Um vai ser tudo o que ele tenha passado para o standard output, e outro vai ser o seu exit code.

Podemos ver onde estão estes programas com o “type”.

```
trist(Matrix):~$ type -a ls
ls is aliased to `ls --color=auto'
ls is /bin/ls
```

Bash tem também uns “[builtins](#)”, como o read, type, printf, echo, etc.. <sup>[4]</sup>, que “fazem parte” da linguagem em si. Tecnicamente, os únicos que fazem parte são os especificados no POSIX standard.

```
trist(Matrix):~$ type -a read
read is a shell builtin
```

Por isso, resumidamente, bash é nada mais que:

- Operadores
- Builtins
- Programas ou scripts listados no \$PATH

## Velocidade

Bash não é a linguagem de scripting mais rápida que existe. Nem perto.

Bash é interpreted, o que quer dizer que, ao contrário de programas compilado, he existe já um programa que vai olhar para o código, e corre-lo “manualmente”. Este programa é o famoso /bin/bash. É por isto mais lento <sup>[5]</sup>.

É também lento porque a grande maioria dos comandos (sejam scripts ou programas), têm que ser executados. Ou seja, para programas, têm que ser carregados para memória, corridos, e terminados; isto não é um processo rápido (apesar de ser um processo – pun intended).

Se for um script, tem que (mais uma vez) ser interpreted, mas antes disso tem que ser criado um context para o mesmo. Este último passo não é necessariamente lento, mas é um fator.

## A Quem Serve Saber Bash?

Bash, apesar de extremamente versátil, é, como a grande maioria das linguagens de programação, dispendioso (a nível de tempo) de se aprender.

Alguém cujo tempo seja passado primeiramente em windows, tiraria muito pouco proveito de aprender bash.

Alguém cujo tempo seja passado em linux, mas cujo trabalho envolva um tipo específico de software, também tira pouco proveito de aprender bash.

Eu pessoalmente passo uma quantidade razoável de tempo em linux, a trabalhar em coisas que vão desde desenvolvimento de sistemas operativos até programação em python (com os vários níveis intremédios presentes – C, c++, ocasionalmente java, etc). Vejo também os ocasionais filmes, e no geral diria que faço um bom uso do sistema operativo, mas nunca senti uma necessidade muito grande de mexer com bash.

Quando digo bash, estou a falar de scripting, visto alguém que faça uso pesado da linha de comandos pode continuar a não saber fazer um script em bash.

Diria que bash é primeiramente utilizado por pessoas que têm uma área de trabalho em que certas tarefas podem ser repetitivas.

O maior uso que comecei a dar a bash foi quando comecei a ter que compilar projetos grandes, nos quais algo do género poderia útil:

```
gcc $(ls src/*.c) -o bin/
```

Pouco depois comecei a usar “make”, por isso deixei o bash durante um período de tempo.

Um bom exemplo de alguém que tire grande vantagem de Bash scripting pode ser por exemplo um administrador de um website, que se veja na necessidade de mudar muitos ficheiros de sítio ao mesmo tempo, fazer modificações, encontrar código, etc..

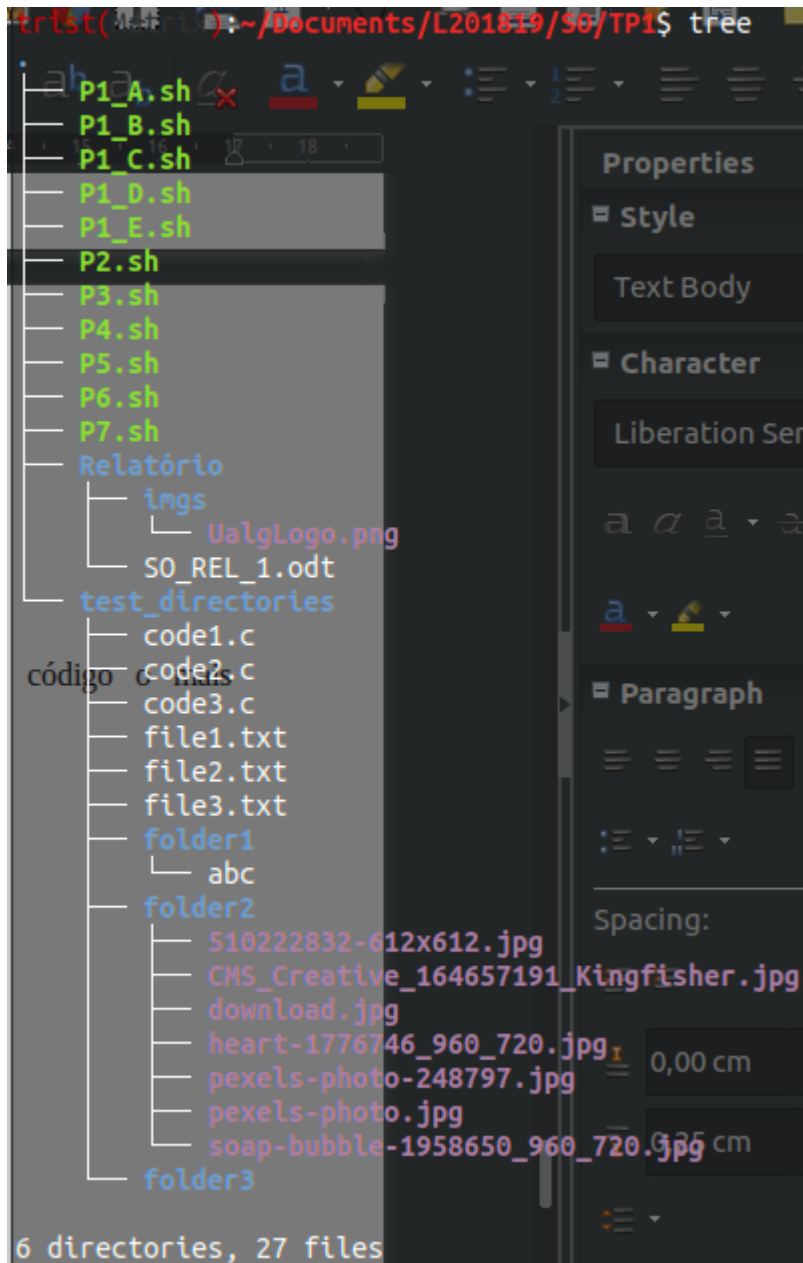


## Exercícios

Resolverei agora os exercícios do guião, tendo como objetivo explicar o código o mais detalhadamente possível, como que a explicar a mim próprio quando não sabia.

Conhecimento ditado em exercícios, não será explicado novamente nos próximos.

O layout da resolução é o seguinte:



```
tr1st(777) (~): ~/Documents/L201819/S0/TP1$ tree
.
├── P1_A.sh
├── P1_B.sh
├── P1_C.sh
├── P1_D.sh
├── P1_E.sh
├── P2.sh
├── P3.sh
├── P4.sh
├── P5.sh
├── P6.sh
├── P7.sh
├── Relatório
│   └── imgs
│       └── UalgLogo.png
├── SO_REL_1.odt
├── test_directories
│   ├── code1.c
│   ├── code2.c
│   ├── code3.c
│   ├── file1.txt
│   ├── file2.txt
│   ├── file3.txt
│   ├── folder1
│   │   └── abc
│   ├── folder2
│   │   ├── 510222832-612x612.jpg
│   │   ├── CMS_Creative_164657191_Kingfisher.jpg
│   │   ├── download.jpg
│   │   ├── heart-1776746_960_720.jpg
│   │   ├── pexels-photo-248797.jpg
│   │   ├── pexels-photo.jpg
│   │   └── soap-bubble-1958650_960_720.jpg
│   └── folder3
└── código e testes
```

6 directories, 27 files

## P1

### A

```
1  #!/bin/bash                                trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P1_A.sh c
2  ls *.c                                     code1.c code2.c code3.c
```

Esquerda → código; Direita → Execução e resultado

Linha 1: “#!/bin/bash”

Esta linha serve para ditar que este script é para ser executado pelo interpreter de bash. [\[6\]](#)

Na maioria dos casos isto é inútil, visto o interpreter default é o de bash, mas num cenário em que este não seja o caso (por exemplo alguém ter mudado por preferência), é útil.

Exemplo:

Estou com ideias de programar a minha própria linguagem de scripting, e caso o faça, posso simplesmente, para executar um script em dita linguagem, fazer “#!/interpreterLocation”.

Se quiser pôr essa linguagem como default, em todos os meus scripts de bash, se não puser binbash, vão tentar executar na minha linguagem e dar erro.

Podemos também ter um interpreter a correr outro:

```
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ /bin/bash
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ exit
exit
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$
```

Linha 2: “ls \*.c”

- ls → este comando serve para listar todos os ficheiros e directorios no directorio em que se está (não recursivamente).
- \* → é um símbolo que representa uma wildcard, ou seja qualquer texto
- . → literalmente só um ponto, não tem significado além disso
- \$ → representa uma variável. O interpreter vai tentar encontrar palavra que se siga, e se houver uma variável com esse nome, o cifrão e a palavra serão trocados pelo valor dessa variável.
- 1 → nome da variável. As variáveis com nomes que são números representam os argumentos passados quando o comando foi chamado. (exemplo: cmd abc bcd cde → \$1 == “abc”, \$2 == “bcd”, \$3 == “cde”).

Ou seja, este comando lista todos os ficheiros cujo nome tenha “[parametro 1]” no fim.

Como passamos “c” ele listou todos os directórios com “.c” no fim do seu nome. (aka extensão)

## B

```
1 #!/bin/bash
2 du -s -B1 */ | awk -v x=$1 '$1 > x { print $2 }'
```

trist(matrix):~/Documents/L201819/S0/TP1/test\_directories\$ ../P1\_B.sh 4096  
folder1/  
folder2/

Comando du: Estimativa de espaço ocupado. ([Man page](#))

Parametro -s: Usado para prevenir recursividade.

Parametro -B: Usado para mostrar o tamanho em Bytes (8 bits)

1: usado como divisor (se fosse 2, uma pasta com 2048 bytes apresentar-se-ia como se com 1024).

Poderia procurar-se por Kb com -B1024, assim uma pasta com 2kb daria o numero 2 em vez de 2048. (Pensei usar isto como parâmetro adicional, mas não queria estar a complicar).

\*/: todos os diretórios são representados por uma barra no fim, por isso \*/ lista apenas diretórios.

| : passar o output do comando anterior pelo comando que se segue

Comando awk: Sublinguagem para processar padrões ([Man page](#))

Visto que o awk só processa uma linha de cada vez, e o du dá os resultados em várias linhas, é uma mistura perfeita.

Parametro -v x=\$1: O awk, subdivide os seu inputs da mesma maneira que o bash o faz. Por espaços (se não houver aspas), e cria o seu próprio contexto. Por esta razão tem um parâmetro (-v), que serve para passar variáveis lá para dentro. “x=\$1” indica que, no contexto do awk, a variável x vai ter o valor que, neste contexto do bash, o \$1 tem.

“\$1 > x { print \$2 }”: Na sintaxe do awk, variaveis passadas por -v, não precisam de ter \$.

Para cada linha do seu input, o awk subdivide-o em parâmetros, o output normal do du é este:

```
90112 folder1/
712704 folder2/
4096 folder3/
```

Por isso, dentro do awk, \$1 vai ser o número de bytes, e \$2 vai ser o nome da pasta.

Assim, comparamos o número de bytes de cada pasta ao input do script, e se for maior, o awk dá de volta o nome da pasta, caso contrário, deita fora.

Temos portanto assim um comando não recursivo que nos dá diretórios com tamanho em bytes maior ao passado no primeiro argumento.

## C

```
1 #!/bin/bash
2 find * -type f -printf "%p %T@\n" | \
3 awk -v from=$(date -d "$1" +%s.%N") -v to=$(date -d "$2" +%s.%N") '$2 >= from && $2 <= to { print $1 }'
```

trist(Matrix):~/Documents/L201819/SO/TP1/test\_directories\$ ./P1\_C.sh "10/12/2018 19:25" "10/12/2018 19:27"

folder2/download.jpg  
folder2/CMS\_Creative\_164657191\_Kingfisher.jpg  
folder2/heart-1776746\_960\_720.jpg  
folder2/510222832-612x612.jpg  
folder2/soap-bubble-1958650\_960\_720.jpg  
folder2/pexels-photo.jpg

Comando find: Procurar por ficheiros numa hierarquia de diretórios ([Man Page](#))

Parametro \*: procurar todos os ficheiros e diretórios no diretório atual (recursivamente).

Parametro “-type f”: Tipo = ficheiro

Parametro “-printf “%p %T@\n””: Apresentar os ficheiros formatados com nome primeiro, e depois a timestamp da data de modificação dos ficheiros (%p representa o nome %T@ representa tempo de modificação → timestamp).

Temos agora algo novo: “\$(expressão)” - isto é substituído pelo resultado de “expressão”.

Neste caso temos “date -d “\$1/2” +%s.%N”.

Date é um comando para lidar com datas e tempos ([Man Page](#)).

O parâmetro -d serve para receber uma data (que neste caso vai ser os inputs), e o parâmetro + é seguido de uma string que representa como expelir os resultados. Neste caso %s é a timestamp (segundos desde 1 de Janeiro de 1970 aka epoch), e %N é os nanosegundos.

```
trist(Matrix):~/Documents/L201819/SO/TP1/test_directories$ find * -type f -printf "%p ||| %t\n"
```

code1.c ||| Fri Oct 12 18:49:48.0792322250 2018  
code2.c ||| Fri Oct 12 18:49:53.2038416540 2018  
code3.c ||| Fri Oct 12 18:54:41.202082028012018  
file1.txt ||| Wed Oct 10 22:50:50.6221662210 2018  
file2.txt ||| Wed Oct 10 22:50:53.0223537270 2018  
file3.txt ||| Wed Oct 10 22:50:55.4185387780 2018  
folder1/abc ||| Thu Oct 11 18:40:16.6761701650 2018  
folder2/download.jpg ||| Fri Oct 12 19:25:19.4155570840 2018  
folder2/CMS\_Creative\_164657191\_Kingfisher.jpg ||| Fri Oct 12 19:25:25.1443916090 2018  
folder2/heart-1776746\_960\_720.jpg ||| Fri Oct 12 19:25:30.8772220020 2018  
folder2/510222832-612x612.jpg ||| Fri Oct 12 19:25:40.2945760110 2018  
folder2/pexels-photo-248797.jpg ||| Fri Oct 12 19:24:58.1364153440 2018  
folder2/soap-bubble-1958650\_960\_720.jpg ||| Fri Oct 12 19:25:54.4925941500 2018  
folder2/pexels-photo.jpg ||| Fri Oct 12 19:25:13.7787313390 2018

Aqui ^ estão impressas as datas de modificação de todos os ficheiros.

```
trist@rix:~/Documents/L201819/SO/TP1/test_directories$ find . -type f -printf "%p ||| %T@\n"
code1.c ||| 1539366588.0792322250
code2.c ||| 1539366593.2038416540
code3.c ||| 1539366881.2020820280
file1.txt ||| 1539208250.6221662210
file2.txt ||| 1539208253.0223537270
file3.txt ||| 1539208255.4185387780
folder1/abc ||| 1539279616.6761701650
folder2/download.jpg ||| 1539368719.4155570840
folder2/CMS_Creative_164657191_Kingfisher.jpg ||| 1539368725.1443916090
folder2/heart-1776746_960_720.jpg ||| 1539368730.8772220020
folder2/510222832-612x612.jpg ||| 1539368740.2945760110
folder2/pexels-photo-248797.jpg ||| 1539368698.1364153440
folder2/soap-bubble-1958650_960_720.jpg ||| 1539368754.4925941500
folder2/pexels-photo.jpg ||| 1539368713.7787313390
```

Aqui ^ as timestamps

```
trist@rix:~/Documents/L201819/SO/TP1/test_directories$ date -d "10/12/2018 19:25" +%s.%N
1539368700.000000000
```

Aqui ^ a timestamp de 10/12/2018 19:25

Depois como já aprendemos no exercício anterior, o awk faz o resto do trabalho.

Ficamos assim com um comando que imprime o nome de todos os ficheiros cuja data está compreendida entre a providenciada no primeiro parâmetro e a providenciada no segundo.

## D

Não consegui compreender o objetivo do exercício D.

“Listar o login (...) dos utilizadores do sistema.”

# login

/ˈlɒɡɪn/

*noun*

an act of logging in to a computer, database, or system.

• a password or code used when logging in.

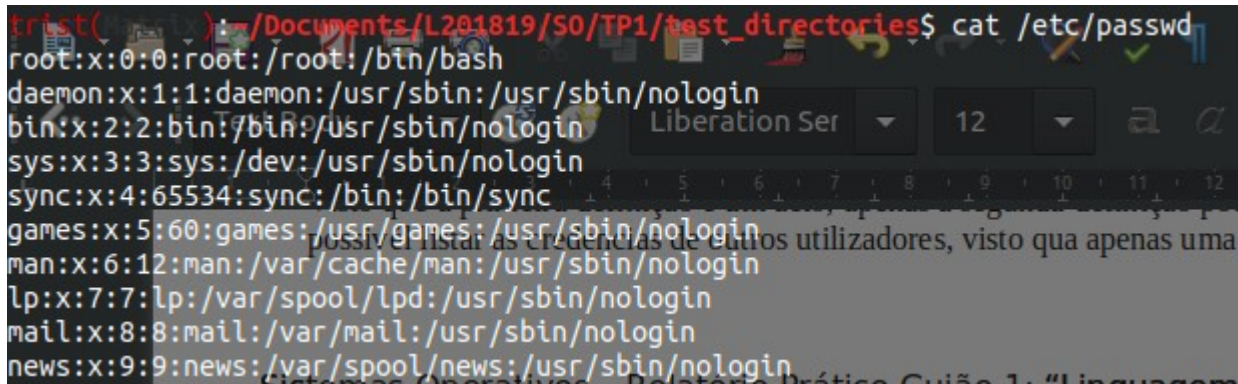
“you need to remember your user login”

Visto que a primeira definição é um acto, apenas a segunda definição pode fazer sentido, e não é possível listar as credencias de outros utilizadores, visto que apenas uma hash é guardada.



Posso porém fazer a outra metade do enunciado (“Listar o (...) nome dos utilizadores do sistema”).

Existe um ficheiro /etc/passwd que contém os nomes dos utilizadores, em conjunto com mais alguma informação:

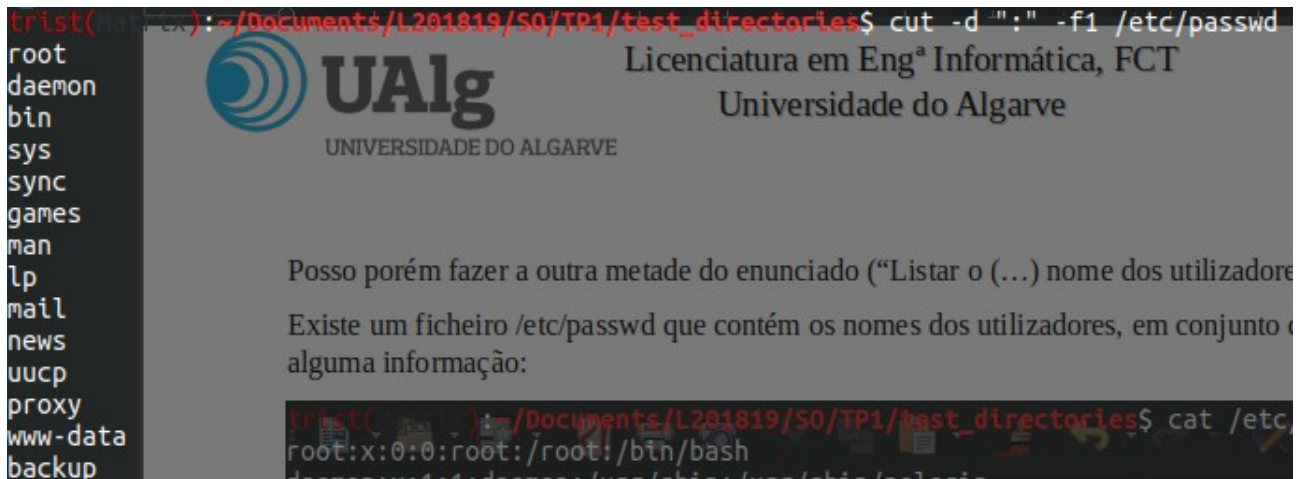


```
trist( ~ ) : ~/Documents/L201819/S0/TP1/test_directories$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

E se fizermos “cut -d “:” -f1 /etc/passwd” podemos listar só os users.

“-d” é data, ou seja “:”

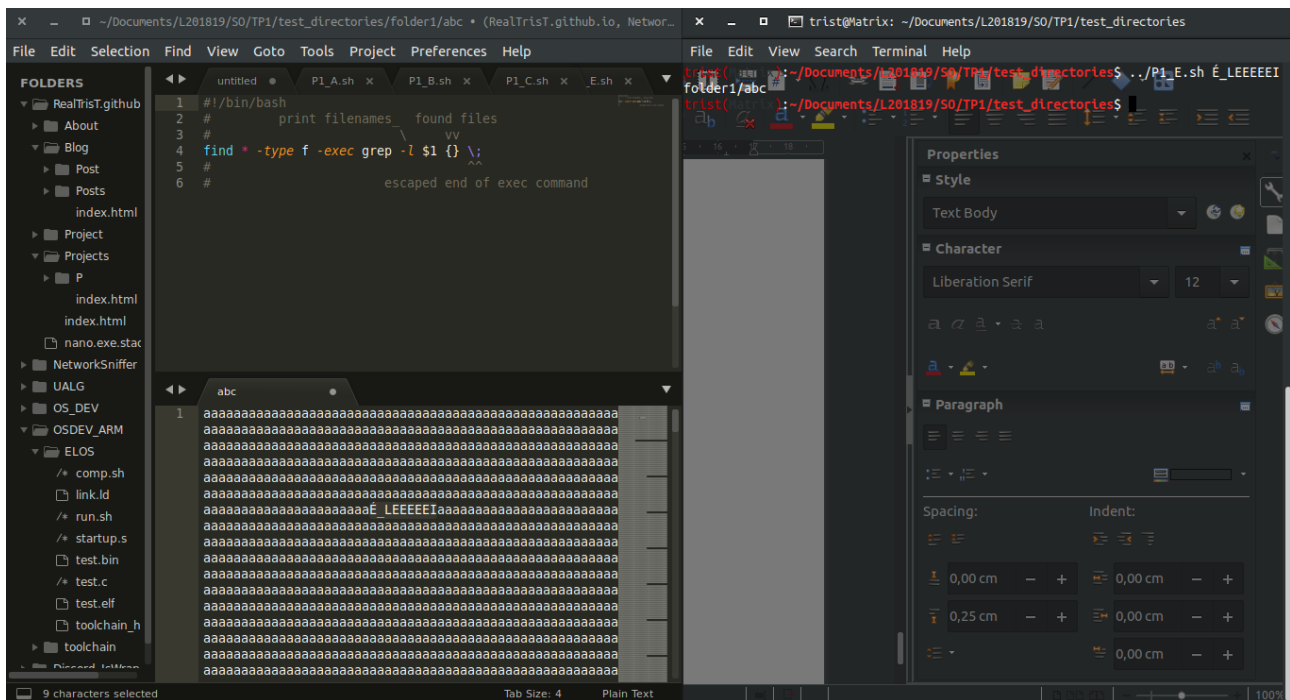
“-f” é “field”, neste caso queremos parar ao primeiro “:”, por isso metemos 1  
depois o nome do ficheiro.



```
trist( ~ ) : ~/Documents/L201819/S0/TP1/test_directories$ cut -d ":" -f1 /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
```

O meu nome de utilizador “trist” está mais embaixo (a lista é extensa).

**E**



Estamos mais uma vez a usar o “find”.

Mas agora temos o parâmetro `-exec`. Este em si é simples. Executa um comando após a execução do `find`.

Grep é o comando que vai tentar encontrar dentro dos ficheiros uma certa string.

Os parâmetros passados para o `grep` são o `-l`, que faz com que o `grep` imprima os nomes dos ficheiros em vez dos caracteres, linhas e ficheiros nos quais encontrou.

Depois a string a encontrar, e depois os ficheiros.

O find automaticamente substitui “{ }” por todos os ficheiros que encontrou, e assim o grep imprime todos os ficheiros nos quais encontrou \$1.

O “\;” serve para indicar que o fim do comando passado ao parâmetro exec é ali.

Poderia fazer-se, para menos confusão “`grep -l $1 $(find * -type f)`”. Talvez mais confusão? Eu diria referência pessoal.

## P2

```
1 result=1
2 for i in $(seq $1);
3 do
4     #Similar to the let command, the (( ... )) construct permits arithmetic expansion and evaluation.
5     #http://tldp.org/LDP/abs/html/dblparens.html
6     result=$((result * $i))
7 done
8 echo $result
```

trist(Matrix):~/Documents/L201819/S0/TP1\$ ./P2.sh 5  
120  
trist(Matrix):~/Documents/L201819/S0/TP1\$

O comando “seq x” gera um output “1 2 3 ... x”.

E o for funciona por “arrays”.

Ou seja, “for i in ‘1 2 3 4 5’”.

O que vai correr o corpo do loop (de “do” até “done”), quantos números houver (neste caso 5), e i vai ser o numero que está a ser “processado”.

A primeira vez que o loop corre, i vai ser 0, segunda 1, etc..

O comentário diz que `$((expressão))` faz com que “expressão” seja avaliada como uma expressão aritmética (uma conta basicamente) [\[7\]](#).

No fim “echo” imprime o resultado. “echo” é usado para imprimir o que vem assegurar até ao fim de linha.

Ou seja temos um script que faz o fatorial.



## P3

```
1 #echo $(ls *$1 | sed -e 's/\.*$//')
2 #regex:
3 # s/ <- substitution indicator
4 # \. <- escaped dot
5 # . <- single instance
6 # * <- any number of the previous (single instance of a character)
7 # $ <- end of the string
8 # // <- first indicates the beginning of the second pattern,
9 #      second indicates the end of the s operator
10 #http://tldp.org/LDP/abs/html/x23170.html
11
12 for fil in $(ls *$1 | sed -e 's/\.*$//');do
13     mv $fil$1 $fil$2
14 done
15
```

```
trist@matrix:~/Documents/L201819/SO/TP1/test_directories$ ls
code1.c code3.c file2.txt folder1 folder3
code2.c file1.txt file3.txt folder2
trist@matrix:~/Documents/L201819/SO/TP1/test_directories$ ../P3.sh .c .yeet
code1.yeet code3.yeet file2.txt folder1 folder3
code2.yeet file1.txt file3.txt folder2
trist@matrix:~/Documents/L201819/SO/TP1/test_directories$
```

Aqui há algo de novo. O sed.

Vai ser passado para o “sed” tudo o que saiu do “ls”, e o sed vai aplicar uma regex a cada um dos nomes dos ficheiros [\[8\]](#), removendo-os. Assim, a cada iteração do for, \$fil vai ter o valor do nome dos ficheiros encontrados, mas sem a extensão.

Visto que ambos os argumentos vão ser extensões, podemos depois fazer um mv \$fil\$1 (nome do ficheiro + extensão inicial) para \$fil\$2 (nome do ficheiro + nova extensão).

Mas porquê um “mv”?! “mv” supostamente MOVE ficheiros (???).

Bem é simples, podemos pensar em mover ficheiros como se fosse copiar para um sítio novo, e apagar o velho.

Assim, estamos a copiar o velho para um com uma extensão nova, e a apagar o velho.

Ficamos assim com um ficheiro igual, mas com um nome diferente. Ou seja, efetivamente renomeamos o ficheiro.



## P4

```
1 cat $1 $2 > $3
2 wc -l < $3

trist(Matix):~/Documents/L201819/S0/TP1/test_directories$ cat code1.yeet
ey u gal bot
trist(Matix):~/Documents/L201819/S0/TP1/test_directories$ cat code2.yeet
aye yeh goi beye
trist(Matix):~/Documents/L201819/S0/TP1/test_directories$ ./P4.sh code1.yeet code2.yeet aye.txt
2
trist(Matix):~/Documents/L201819/S0/TP1/test_directories$ cat aye.txt
ey u gal bot
aye yeh goi beye
trist(Matix):~/Documents/L201819/S0/TP1/test_directories$
```

“cat” serve para concatenar ficheiros, em que os parâmetros são os ficheiros a serem concatenados, procede depois a imprimir conteúdo da concatenação para a consola.

O operador “>” depois de um comando faz com que o seu output seja posto num ficheiro.

Por isso estamos a concatenar os ficheiros providenciados nos 2 primeiros parâmetros para dentro do ficheiro do terceiro.

“wc -l” serve então para listar o número de linhas de um dado input.

```
trist(Matix):~/Documents/L201819/S0/TP1/test_directories$ wc -l
asdasds nos 2 primeiros parâmetros para dentro
ds
d
asde um dado input.
4
```

Como podemos ver, espera um input pela parte do utilizador.

Por isso fazemos o reverso. Utilizamos o operador “<” para providenciar um ficheiro como input.

Simple :) )



## P5

```
1 for i in $(seq 0 5);do
2     echo "$i x $1 = $((i * $1))"
3 done
4
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P5.sh 4 5
0 x 4 = 0
1 x 4 = 4
2 x 4 = 8
3 x 4 = 12
4 x 4 = 16
5 x 4 = 20
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$
```

Este é bastante simples, e já compreendemos tudo até aqui, excepto o “seq” ter 2 parâmetros.

O primeiro parâmetro do seq é onde começa, e o último onde acaba.

## P6

```
1 if [ $((($1 % 4)) -ne 0)];then
2     echo 28
3 elif [ $((($1 % 100)) -ne 0)]; then
4     echo 29
5 elif [ $((($1 % 400)) -ne 0)]; then
6     echo 28
7 else
8     echo 29
9 fi
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P6.sh 2018
28
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P6.sh 2019
28
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P6.sh 2020
29
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$ ../P6.sh 2021
28
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories$
```

Aqui temos algo novo. O “if” e o “%”.

Em bash os ifs não são tão simples como na maior parte das linguagens, mas são ainda assim razoavelmente simples.

O típico “else if” é “elif”, e é determinado o fim do corpo do if por um “fi” (faz lembrar pascal).

Fora isso, em vez dos típicos parênteses que temos na maioria das linguagens, temos parênteses retos, e em vez de operadores como “<”, “>”, “==” ou “!=”, temos “[parametros](#)”.

Neste caso, queremos ver se o ano é bisexto, por isso usamos o devido [algoritmo](#) para o calendário gregoriano. Temos assim um script que, dado o ano, nos dá os dias que fevereiro tem.

## P7

```

1  imgList=""
2  tindex=0
3
4  mkdir -p mins
5  for fil in $(ls *.jpg | sed -e 's/\.*$//');do
6      convert $fil.jpg -resize 600x400^ -gravity Center -extent 600x400 mins/${fil}_min.jpg
7
8      imgList[$tindex]="$fil"
9      tindex=$((tindex+1))
10 done
11
12 tindex=0
13 imageElements=""
14 for fil in ${imgList[@]}; do
15     echo "
16         <!DOCTYPE html>
17         <html>
18             <head>
19                 <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">
20                 <style>
21                     a {text-decoration: none; display: inline-block; padding: 8px 16px;}
22                     a:hover {background-color: #ddd; color: black;}
23                     .previous {background-color: #f1f1f1; color: black;}
24                     .next {background-color: #4CAF50; color: white;}
25                 </style>
26             </head>
27             <body>
28                 <center>
29                     <h2>$1: $fil</h2>
30                     <a href=\"${imgList[$(($tindex-1) % (${#imgList[@}]))]}].html\" class=\"previous\">
31                         &laquo; Previous
32                     </a>
33                     <a href=\"../index.html\" class=\"next\">Back</a>
34                     <a href=\"${imgList[$(($tindex+1) % (${#imgList[@}]))]}].html\" class=\"previous\">
35                         Next &raquo;
36                     </a><br><br><img src=\"../${fil}.jpg\">
37                 </center>
38             </body>
39         </html> " > mins/$fil.html

```

Neste exercício pretende-se criar um conjunto de miniaturas de imagens pre-existent, e um modo de apresentação em HTML.

A apresentação em html vai ter um index.html onde estarão todas as miniaturas (como thumbnails), e terá depois páginas individuais para cada imagem. Não irei explicar o html ou css visto que não era esse o objetivo deste exercício.

Linha 1 & 2: criar variáveis. ImgList será um array com os nomes dos ficheiros de imagens, tindex será uma variável que será utilizada em vários loops através do programa.

Linha 4: criar um directorio para as miniaturas caso não exista, -p para ser silencioso caso já exista.

Linha 5-10: para cada jpg no directorio atual, criar uma miniatura de 600x400 no directorio mins e adicionar o nome para o array.

Linha 13: criar uma variável para ter os elementos de html que vão acabar em index.html.

Linhas 30 & 34: Gerar o url das imagens anteriores e próximas. (terá um botão)



```
40
41     imageElements+="
42     <div class=\"responsive\">
43     <div class=\"gallery\">
44     <a href=\"mins/${fil}.html\">
45     <img src=\"mins/${fil}_min.jpg\" alt=\"Cinque Terre\" width=\"600\" height=\"400\">
46     </a>
47     <div class=\"desc\">
48     ${fil}
49     </div>
50     </div>
51     </div>\"
52
53     tindex=$((tindex+1))
54 done
55
56 echo \"
57 <!DOCTYPE html>
58 <html>
59     <head>
60     <style>
61     div.gallery {border: 1px solid #ccc;}
62     div.gallery:hover {border: 1px solid #777;}
63     div.gallery img {width: 100%; height: auto;}
64     div.desc {padding: 15px; text-align: center;}
65     * {box-sizing: border-box;}
66     .responsive {padding: 0 6px; float: left; width: 24.99999%;}
67     @media only screen and (max-width: 700px) {
68     .responsive {width: 49.99999%; margin: 6px 0;}
69     }
70     @media only screen and (max-width: 500px) {
71     .responsive {width: 100%;}
72     }
73     .clearfix:after {content: \"\"; display: table; clear: both;}
74     </style>
75     </head>
76     <body>
77     <h2>
78     $1
79     </h2>
80
81     $imageElements
82
83     <div class=\"clearfix\">
84     </div>
85     <div style=\"padding:6px;\">
86     </div>
87     </body>
88     </html>\" > index.html
89
90
91
92 read -p "Would you like to open the generated (\"$1\") website? [Y/N] " resp
93 case $resp in
94     "Y")xdg-open index.html ;;
95     "y")xdg-open index.html ;;
96     "N")echo ended ;;
97     "n")echo ended ;;
98 esac
```

Linha 41: adicionar à string um elemento para cada imagem a ser gerada para a página index.html

Linha 56: Gerar a página index.html.

Linhas 92 – 98: ler input do utilizador e determinar se quer abrir a página gerada no seu browser.

```

trist(Matrix):~/Documents/L201819/S0/TP1/test_directories/folder2$ ls
510222832-612x612.jpg heart-1776746_960_720.jpg soap-bubble-1958650_960_720.jpg
CMS_Creative_164657191_Kingfisher.jpg pexels-photo-248797.jpg
download.jpg pexels-photo.jpg
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories/folder2$ ../../P7.sh "É LEEEEEI"
Would you like to open the generated ("É LEEEEEI") website? [Y/N] n
ended
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories/folder2$ ls
510222832-612x612.jpg heart-1776746_960_720.jpg pexels-photo-248797.jpg
CMS_Creative_164657191_Kingfisher.jpg index.html pexels-photo.jpg
download.jpg mins soap-bubble-1958650_960_720.jpg
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories/folder2$ ls mins/
510222832-612x612.html scale=1\> download_min.jpg pexels-photo.html
510222832-612x612_min.jpg heart-1776746_960_720.html pexels-photo_min.jpg
CMS_Creative_164657191_Kingfisher.html heart-1776746_960_720_min.jpg soap-bubble-1958650_960_720.html
CMS_Creative_164657191_Kingfisher_min.jpg pexels-photo-248797.html soap-bubble-1958650_960_720_min.jpg
download.html pexels-photo-248797_min.jpg
trist(Matrix):~/Documents/L201819/S0/TP1/test_directories/folder2$

```

Temos assim o “website” gerado.

É LEEEEEI



510222832-612x612



CMS\_Creative\_164657191\_Kingfisher



download



heart-1776746\_960\_720



pexels-photo-248797



pexels-photo



soap-bubble-1958650\_960\_720

Index.html ^



**É LEEEEEI: 510222832-612x612**

« Previous   **Back**   Next »



Click na primeira imagem ^

**É LEEEEEI: CMS\_Creative\_164657191\_Kingfisher**

« Previous   **Back**   Next »



Click no “Next” ^

## Notas

Foram usados arrays para isto. Arrays são simples:

“variable[index]=value” é como se mete um valor no array (não é precisa inicialização ou alocação)

“\${variable[@]}” representa o array em string (todos os valores seguidos), o que foi útil para o for loop.

“\${#variable[@]}” representa a quantidade de elementos no array

Temos também o read, que é um builtin.

A sintaxe é “read variável”, em que variável se torna o que o user escrever na consola.

-p é uma mensagem que é “impressa” antes de o read esperar pelo input do utilizador.

Case é mais uma estrutura de controlo que se usa assim:

```
case $variavel in
```

```
    “valor possível 1”) código_a_executar_1 ;;
```

```
    “valor possível 2”) código_a_executar_2 ;;
```

```
    “valor possível n”) código_a_executar_n ;;
```

```
esac # <- fim do case.
```

O convert também é simples, o “^” assegurar a 600x400 indica que se deve cortar a imagem até chegar a esse tamanho, -gravity é o que é que se corta (“Center” significa que se deve cortar de fora para o centro, enquanto por exemplo “left” começaria a cortar da direita para a esquerda, e de cima e baixo até a um ponto médio), “-extent” ajuda a especificar como o corte funciona.

“xdg-open” serve para detetar qual o tipo de ficheiro, e abri-lo na aplicação pré-definida pelo utilizador.



## Conclusão

Podemos assim não só ganhar uma apreciação geral sobre como bash funciona, mas também pelas partes constituintes de um sistema operativo.

Enterti-me a fazer os exercícios e espero por uma próxima.

## Referências

- [1]: [Wiki: Operating System](#)
- [2]: [x86 Reference Manual](#)
- [3]: [Wiki: Bash](#)
- [4]: [Bash Builtin Commands](#)
- [5]: [Stackoverflow: Why are interpreted languages slower than compiled ones?](#)
- [6]: [Unix.Stackexchange: What are the contents of /bin/bash, and what would happen if overwritten?](#)
- [7]: [Bash: The double parentheses construct](#)
- [8]: [Sed: Basic Operators](#)

## Bibliografia

Os meus conhecimentos de bash vieram de documentação, tentativa, e stackoverflow.

A grande maioria dos meus conhecimentos sobre sistemas operativos adquiri há uma quantidade considerável de tempo quando fiz um kernel module para linux e um mini-sistema operativo para x86, por isso não irá estar aqui tudo, mas aqui vai a minha tentativa:

- [wiki.osdev.org](http://wiki.osdev.org)
- [stackoverflow.com](http://stackoverflow.com)
- [unix.stackexchange.com](http://unix.stackexchange.com)
- [askubuntu.com](http://askubuntu.com)
- [guidedhacking.com](http://guidedhacking.com)
- [man7.org/linux/man-pages/](http://man7.org/linux/man-pages/)