Programação em UNIX
Introdução

Jorge Silva
MIEIC / FEUP

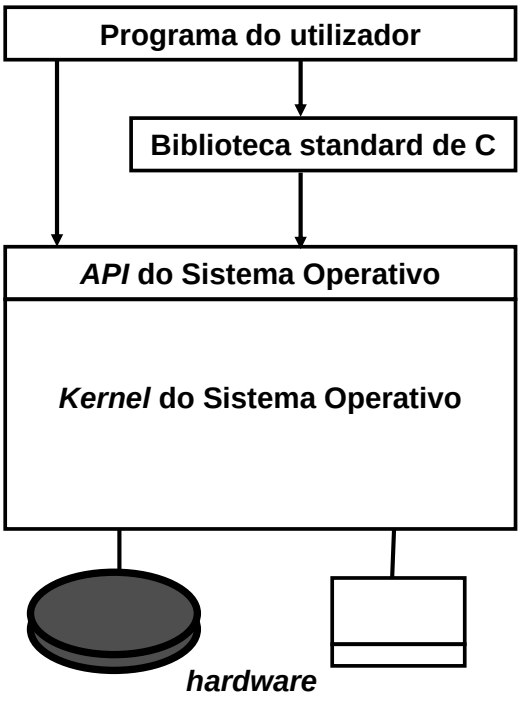
Objectivos

No final desta aula, os estudantes devem ser capazes de:

- **Explicar a diferença entre chamadas a funções da *API* e da Biblioteca standard de C**
- **Compilar um programa em C e usar o manual *online***
- **Aceder aos argumentos da linha de comandos e às variáveis de ambiente**
- **Tratar situações de erro de execução de um programa**
- **Medir tempos de execução de um programa**
- **Descrever o percurso "normal" de execução de um programa em C**

INTRODUÇÃO

- **Os programas pedem serviços ao Sistema Operativo através de chamadas ao sistema.**
- **Uma chamada ao sistema é um ponto de entrada directa no *kernel*.**
- **O *kernel* é um conjunto de módulos de *software* que executam em *modo privilegiado*, significando que têm controlo total sobre os recursos do sistema.**
- **As funções da biblioteca de C podem ou não invocar chamadas ao sistema.**



Manual do UNIX

A maior parte dos sistemas UNIX têm documentação *online*, as designadas *man pages* (páginas do manual).

Tradicionalmente, estas páginas estão divididas em secções:

- **1- *user commands***
- **2- *system calls***
- **3- *C library functions***
- **…**
- **8- *system maintenance***
- ***…***

As páginas do manual referem-se aos itens colocando o número da secção entre parêntesis.

- **Exemplo: *open (2)***

Cada página também está organizada em secções:

- **HEADER: o título da página em questão**
- **NAME: um sumário**
- **SYNOPSIS: descreve o uso**
- **AVAILABILITY: indica se está disponível no sistema**
- **DESCRIPTION: descreve o que faz o comando ou a função**
- **RETURN VALUES: indica os valores retornados (se aplicável)**
- **ERRORS: sumariza os valores de *errno* e as condições de erro**
- **FILES: lista os ficheiros que o comando ou a função usa**
- **SEE ALSO: lista comandos/funções relacionadas ou outras secções**
- **ENVIRONMENT: lista variáveis de ambiente relevantes**
- **NOTES: informação acerca de utilizações pouco usuais ou características de implementação**
- **BUGS: lista problemas conhecidos**

As páginas do manual podem ser consultadas com o utilitário *man*.

- ***man [section] word***
- ***man -k keyword***

Exemplos: executar os seguintes comandos e interpretar o resultado

- ***man ls***
- ***man man***
- ***man intro***
- ***man write***
- ***man 1 write***
- ***man 2 write***
- ***man -k mode***

A página de `write(1)` contém informação sobre um comando.
A página de `write(2)` descreve uma chamada ao sistema.

As páginas do manual também podem ser consultadas no *help browser* do Linux ou na *web*:

- ***linux.die.net***
- ***manpages.ubuntu.com***
- ***...***

Chamadas ao sistema e Funções da biblioteca de C

Quando se usam chamadas ao sistema ou funções da biblioteca de C, convém consultar o manual para saber o protótipo da função, as *header files* necessárias, os parâmetros a incluir na chamada e o tipo de resultado obtido.

Exemplo: página do manual referente à função *write(2)*

- **SYNOPSIS**

  #include <unistd.h>

  ssize_t write(int filedes, const void *buf, size_t nbyte);

  A página indica que esta função escreve *nbyte* bytes de *buf* para um ficheiro especificado por *filedes* e que retorna o nº de bytes efectivamente escritos

Compilação de programas

O compilador de C, *cc* ou *gcc*,
traduz programas fonte em C
em módulos objecto ou em módulos executáveis.

A compilação é feita em várias etapas:

- **o preprocessador expande macros e inclui as *header files***
- **o compilador faz vários passos pelo código traduzindo-o primeiro p/ linguagem *assembly* da máquina alvo e depois para linguagem máquina**
- **o resultado é um módulo objecto constituído por código máquina e tabelas de referências por resolver**
- **este módulo objecto é ligado a outros módulos para formar um executável em que todas as referências estão resolvidas**

Programa em C:

```
/ * Programa hello.c */

#include <stdio.h>

int main(void)
{
 printf("Hello world !\n");
 exit(0);
}
```

Compilar:

```
> cc hello.c
```

- **por omissão, o executável fica no ficheiro `a.out`**

ou

```
> cc -o hello hello.c
```

Executar:

```
> ./hello
Hello world !
>
```

- **se o directório actual não estiver no PATH é preciso acrescentar `./`**
- **`PATH = $PATH:.` (na linha de comandos) acrescenta o directório actual ao PATH**
- **alternativa: editar o ficheiro `.profile`, `.bash_profile`, ou equivalente, e fazer login**

Consultar no manual de `cc` ou `gcc` outras opções do compilador.
Recomenda-se a utilização da opção `-Wall`

Desenvolvimento de programas

Header files

- Para programar em C, precisamos de *header files* que contêm definições de constantes e declarações de chamadas ao sistema ou à biblioteca da linguagem.
- A maior parte destas *header files* estão localizadas em `/usr/include` e seus subdirectórios.
- É possível especificar outros directórios onde devem ser procurados *include files*, para além dos directórios *standard*, usando o *switch* de compilação `-I`

```
> cc -I/usr/myname/include -o prog1 prog1.c
```

- Para procurar *header files* contendo certas definições ou protótipos de funções pode usar-se o comando `grep`

```
> grep _SC_CLK_TCK /usr/include/*.h
```


Library files

- *Library* (biblioteca) - colectânea de funções pré-compiladas que foram escritas de modo a serem reutilizáveis.
- As bibliotecas *standard* estão em `/lib` ou `/usr/lib` ou `usr/local/lib`
- Os nomes das bibliotecas começam sempre por lib.
  O resto do nome indica o tipo de biblioteca
  (ex: `libc`, indica a biblioteca de C, e `libm`, a biblioteca matemática).
  A última parte do nome indica o tipo de biblioteca:
  - » `.a` - biblioteca estática
    - se houver vários programas que usem uma mesma função de uma biblioteca, quando os programas estiverem a correr "simultâneamente" existirão várias cópias da função em memória
  - » `.so` ou `.sa` - biblioteca partilhada
    - o código das funções da biblioteca pode ser partilhado por vários programas
  - » Em Linux, por omissão, são usadas as bibliotecas partilhadas.
    Para forçar a utilização de bibliotecas estáticas
    deve-se incluir a opção `-static` ao invocar o compilador de C.

Início de um programa

Quando se solicita ao S.O. a execução de um novo programa este começa por executar uma rotina, designada C *startup*
(no caso da linguagem C)

Esta rotina

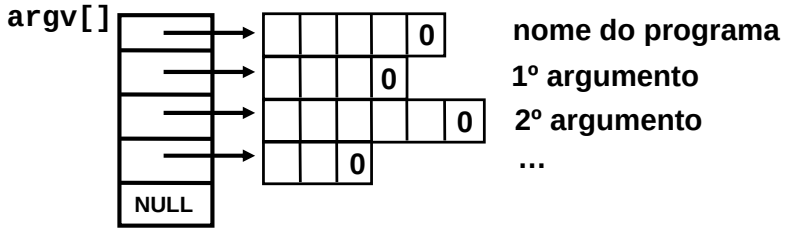
- **vai buscar ao *kernel* os argumentos da linha de comandos e das variáveis de ambiente**
- **abre e disponibiliza 3 "ficheiros" ao programa (*standard input, standard output e standard error*)**
- **invoca a função `main()` do programa**

A função `main()`

Pode ser definida de muitas formas:

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

`argc` - nº de argumentos da linha de comandos, incluindo o nome do programa executado
`argv` - *array* de apontadores p/*strings*, apontando os parâmetros passados ao programa
envp - *array* de apontadores p/*strings*, apontando variáveis de ambiente do programa



Terminação de um processo

Terminação normal:
- executar `return` na função `main()`
- invocar `exit()`
- invocar `_exit()`

Terminação anormal:
- invocar `abort()`
- quando recebe certos sinais (não tratáveis) (v. adiante)

Terminação com `exit()`

```
#include <stdlib.h>
void exit(int status);
```

(ANSI C)

- Termina imediatamente o programa retornando o código de terminação `status` para o S.O.
- Liberta todos os recursos atribuídos ao programa, fecha os ficheiros abertos e transfere dados que ainda não tenham sido guardados p/ o disco

`status`

- a maior parte dos sistemas operativos permite testar o exit status do último processo executado
  - ex: > `echo $?` (na Bourne e na Korn *shell*; `$status` na C *shell*)
- valores habituais
  - 0 (zero) - se não aconteceu erro
  - <> 0 - se aconteceu erro

Terminação com `_exit()`

```
#include <stdlib.h>
void _exit(int status);
```

(POSIX)

- Termina imediatamente o programa retornando o código de terminação `status` para o S.O.
- Liberta todos os recursos atribuídos ao programa, de forma rápida
- Podem ser perdidos dados que ainda não tenham sido guardados

O `status` fica indefinido se

- `exit()` ou `_exit()` forem invocadas sem especificar `status`
- `main()` fizer `return` sem especificar o valor de retorno
- `main()` atingir o fim sem fazer `return`, `exit` ou `_exit`

A função `atexit()`

A função `exit()` pode executar, antes de terminar, uma série de rotinas (*handlers*), que tenham sido previamente registadas para execução no final do programa.

Estas rotinas são executadas por ordem inversa do seu registo.

O registo destes *handlers* é feito através da função `atexit()`

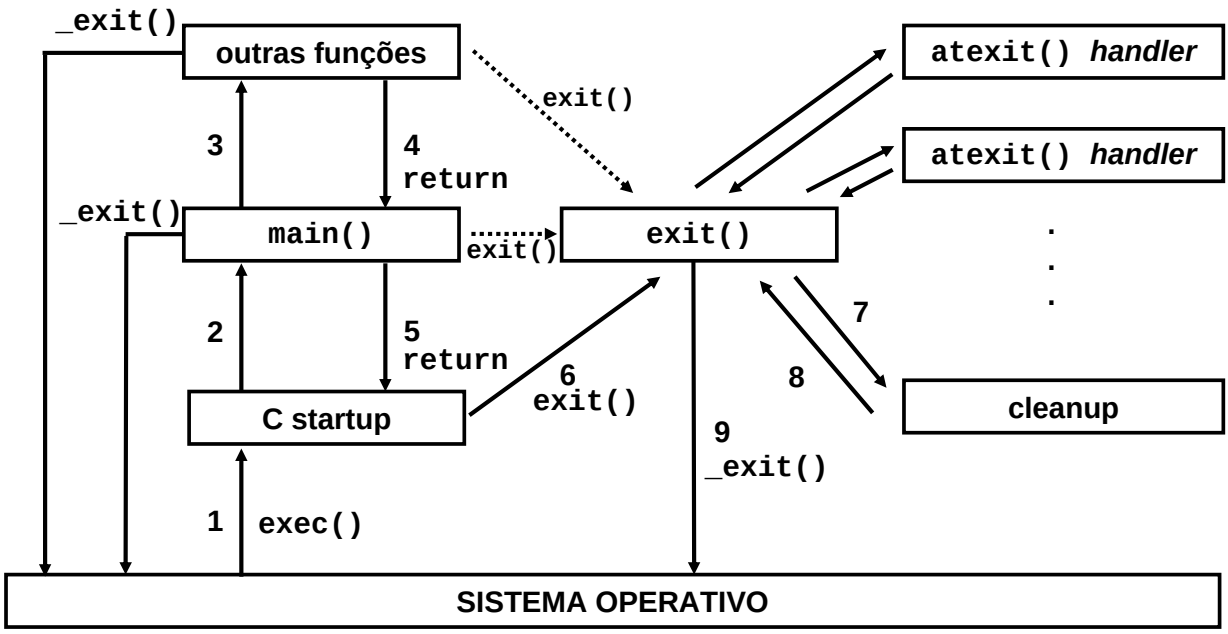
```
#include <stdlib.h>
int atexit(void (*func) (void));

retorno: 0 se OK; <>0 se erro
```

O argumento é o endº de uma função sem argumentos que retorna void

```
exemplo:
...
if (atexit(exithand2)!=0) {
...}
...
```

```
static void exithand2(void) {
...
}
```



1 … 9 - percurso mais frequente

Tratamento de erros

- Em geral, as chamadas ao sistema retornam um valor especial quando acontece um erro, por exemplo:
  - um valor negativo (frequentemente -1)
  - um apontador nulo.
- O tipo de erro que ocorreu é colocado numa variável global `errno`, do tipo `int`.
- O valor desta variável deve ser analisado imediatamente após a chamada que originou o erro.
- O ficheiro `errno.h` define
  - a variável `errno`
  - constantes para cada valor que `errno` pode assumir

ex.:

```
#define EPERM    1 /* Not owner */
#define ENOENT   2 /* No such file or directory */
#define ESRCH    3 /* No such process */
...
```

Tratamento de erros (cont.)

Funções da biblioteca de C, úteis quando ocorrem erros:

```
#include <stdio.h>

void perror (const char *msg);
```

- **Mostra a *string* `msg`, seguida de `": "`, seguida de uma descrição do último erro que ocorreu numa chamada ao sistema.**
- **Se não houver erro a reportar, mostra a *string* `"Error 0"`.**

```
#include <string.h>

char *strerror (int errnum);
```

- **Esta função retorna um apontador para uma *string* que contém uma descrição do erro cujo código foi passado no argumento `errnum` (que é tipicamente o valor de `errno`)**

Medida de tempos de execução

```
#include <sys/times.h>

clock_t times(struct tms *buf);

Preenche a estrutura cujo endereço se fornece em buf
Retorna o tempo actual do sistema, medido a partir de um instante arbitrário,
passado (ex: o arranque do sistema; o instante depende da versão do S.O.)
```

```
struct tms {
clock_t tms_utime;  /* tempo de CPU gasto em código do processo */
clock_t tms_stime;  /* tempo de CPU gasto em código do sistema
                       chamado pelo processo */
clock_t tms_cutime; /* tempo de CPU dos filhos (código próprio) */
clock_t tms_cstime; /* tempo de CPU dos filhos (cód. do sistema) */
```

Todos os tempos são medidos em *clock ticks*.
O número de *ticks* por segundo pode ser determinado usando `sysconf()`:

```
ticks_seg = sysconf(_SC_CLK_TCK);
```

VER EXEMPLO DE UTILIZAÇÃO NO "MATERIAL DE APOIO"