Miguel Pimenta Monteiro

O Sistema Operativo Unix

Alguns aspectos da sua API

(*Application Programming Interface*)

1. Programas em C no Unix

1.1 Início e terminação de programas

1.1.1 Início

Quando se solicita ao S.O. a execução de um novo programa (serviço `exec()` no UNIX) este começa por executar uma rotina (no caso de programas em C) designada por *C startup*. Esta rotina é a responsável por chamar a função `main()` do programa, passando-lhe alguns parâmetros, se for caso disso, e por abrir e disponibilizar três "ficheiros" ao programa: os chamados *standard input*, *standard output* e *standard error*. O *standard input* fica normalmente associado ao teclado (excepto no caso de redireccionamento), enquanto que o *standard output* e o *standard error* ficam normalmente associados ao écran (também podem ser redireccionados).

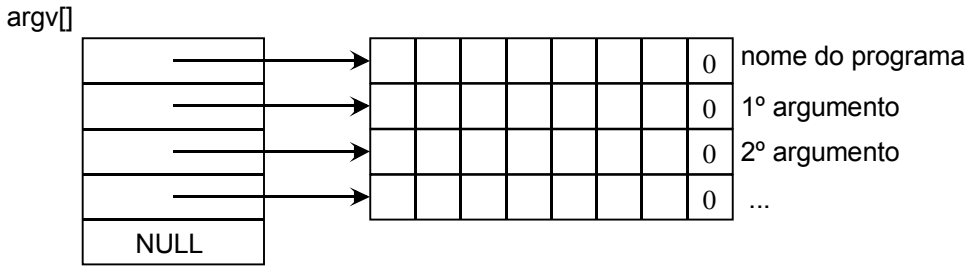
A função `main()` pode ser definida num programa em C de muitas formas:

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

Assim pode ser definida como procedimento (`void`) ou como função retornando um inteiro (`int`). Neste último caso o inteiro retornado é passado a quem chamou a função, ou seja à rotina de *C startup*, que por sua vez o passa ao S.O.

Quando se invoca um programa é possível passar-lhe parâmetros, que são um conjunto de 0 ou mais strings, separadas por espaço(s). Esses parâmetros podem depois ser acedidos em `main()` através dos argumentos `argc` e `argv`.

`argc` - número de argumentos passados, incluindo o próprio nome do programa.
`argv` - array de apontadores para string, apontando para os parâmetros passados ao programa. O array contém um número de elementos igual a `argc`+1. O primeiro elemento de `argv[]` aponta sempre para o nome do programa (geralmente incluindo todo o *path*). O último elemento de `argv[]` contém sempre o apontador nulo (valor `NULL`).



`envp` - array de apontadores para string, apontando para as variáveis de ambiente do programa. Todos os sistemas permitem a definição de variáveis de ambiente da forma `NOME=`*`string`*`;` Cada um dos elementos de `envp[]` aponta para uma string daquela forma (incluindo o `NOME=`). O array contém um número de elementos igual ao número de variáveis de ambiente + 1. O último elemento de `envp[]` contém sempre o apontador nulo (valor `NULL`). A estrutura de `envp[]` é semelhante à de `argv[]`.

Em rigor o parâmetro **envp** da função `main()` não está padronizado (não pertence à definição do ANSI C), mas é implementado em numerosos S.O.s incluindo o UNIX.

1.1.2 Terminação

Um programa em C termina quando a função `main()` retorna (usando `return` *expressão*, no caso de ter sido definida como `int`, e usando simplesmente `return` ou deixando chegar ao fim das instruções, no caso de ter sido definida como `void`). Outra possibilidade é chamar directamente funções terminadoras do programa que são:

```
#include <stdlib.h>

void exit(int status);
```

Termina imediatamente o programa, retornando para o sistema operativo o código de terminação **status**. Além disso executa uma libertação de todos os recursos alocados ao programa, fechando todos os ficheiros e guardando dados que ainda não tivessem sido transferidos para o disco.

```
#include <unistd.h>

void _exit(int status);
```

Termina imediatamente o programa, retornando para o sistema operativo o código de terminação **status**. Além disso executa uma libertação de todos os recursos alocados ao programa de forma rápida, podendo perder dados que ainda não tivessem sido transferidos para o disco.

Quando o programa termina pelo retorno da função `main()` o controlo passa para a rotina de *C startup* (que chamou `main()`); esta por sua vez acaba por chamar `exit()` e esta chama no seu final `_exit()`.

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* de terminação é feito por:

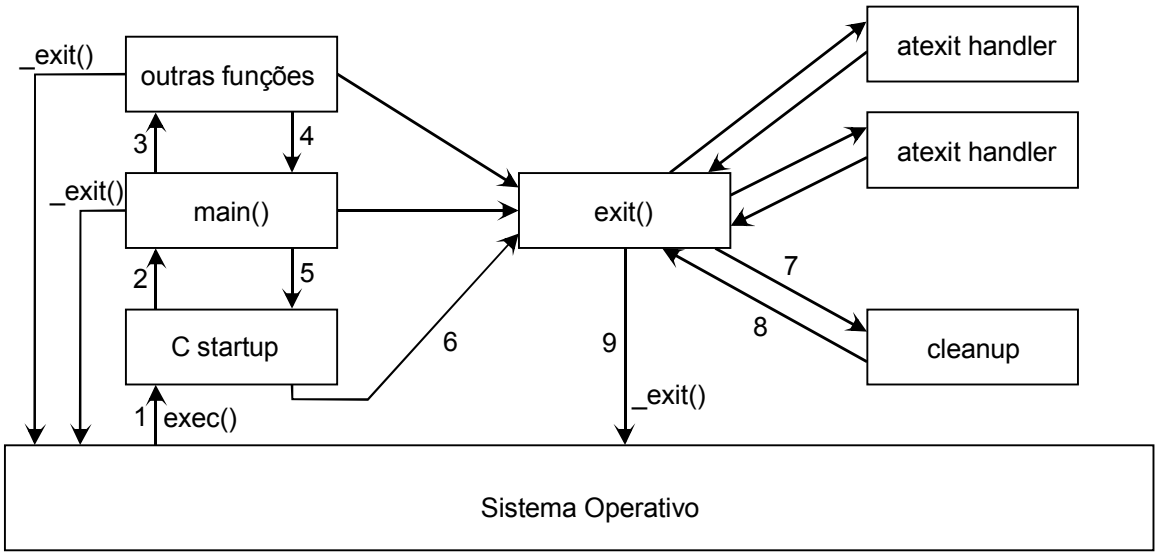
```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Regista o *handler* de terminação **func** (função `void` sem parâmetros). Retorna 0 em caso de sucesso e um valor diferente de 0 em caso de erro.

Na figura seguinte pode ver-se um esquema dos processos de início e terminação de um programa em C.

As funções `exit()` e `_exit()` podem ser vistas como serviços do sistema operativo. De facto correspondem de perto a chamadas directas ao Sistema Operativo (p. ex. `ExitProcess()` e `TerminateProcess()` no caso do Windows NT).



1 .. 9 - Percurso mais frequente

1.2 Processamento dos erros

Grande parte dos serviços dos Sistemas Operativos retornam informação acerca do seu sucesso ou da ocorrência de algum erro que o impediu de executar o que lhe era pedido. No entanto, e em geral, essa informação apenas diz se ocorreu ou não um erro, sem especificar o tipo de erro ou a sua causa.

Para se extrair mais informação relativa ao último erro ocorrido é necessário utilizar outros mecanismos próprios de cada sistema operativo.

A maior parte dos serviços do Unix apenas indica se ocorreu ou não um erro, não especificando o tipo de erro. Esse tipo de erro é colocado, através de um código, numa variável global chamada **`errno`** e que é de tipo inteiro (`int`). No entanto essa variável só contém o código válido imediatamente após o retorno do serviço que causou o erro. Qualquer chamada a outro serviço ou função da biblioteca standard do C pode alterar o valor de **`errno`**. Associadas aos códigos possíveis colocados em **`errno`** existem também constantes simbólicas definidas no ficheiro de inclusão `errno.h`. Alguns exemplos dessas constantes são:

```
ENOMEM
EINVAL
ENOENT
```

No Unix é possível obter e imprimir uma descrição mais detalhada correspondente a cada código de erro. Para imprimir directamente essa descrição pode usar-se o serviço `perror()`:

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

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

Imprime na consola (em *`standard error`*) a string passada no argumento **`string`**, seguida do sinal de dois pontos (:), seguida da descrição correspondente ao código que nesse momento se encontra em `errno`. A linha é terminada com `'\n'`.

Para obter uma descrição colocada numa string deverá usar-se o serviço `strerror()`:

```
#include <string.h>

char *strerror(int errnum);
```

Retorna um apontador para string contendo uma descrição do erro cujo código foi passado no argumento **errnum**. A string retornada é alocada internamente, mas não precisa ser libertada explicitamente. Novas chamadas a `strerror()` reutilizam esse espaço.

Pode ver-se de seguida um exemplo da utilização de `perror()`:

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

void main(void)
{
  char *mycwd;

  if ((mycwd = (char *) malloc(129)) == NULL) { /* alocação de 129 bytes */
    perror("malloc:");
    exit(1);
  }
  if (getcwd(mycwd, 128) == NULL) {   /* obtenção do directório corrente */
    perror("getcwd:");
    exit(1);
  }
  printf("Current working directory: %s\n", mycwd);
  free(mycwd);
}
```

1.3 Medida de tempos de execução

Um programa pode medir o próprio tempo de execução e de utilização do processador através de um serviço do S.O. especialmente vocacionado para isso. Trata-se do serviço `times()`.

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

clock_t times(struct tms *buf);
```

Preenche a estrutura cujo endereço se fornece em **buf** com informação acerca dos tempos de execução do processo.
Retorna o tempo actual do sistema (relógio), medido a partir do arranque.

O serviço anterior preenche uma estrutura que contém 4 campos e é definida em `<sys/times.h>` da seguinte forma:

```
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 (código do sistema) */
}
```

Todos os tempos são medidos em *clock ticks*. Cada sistema define o número de *ticks* por segundo, que pode ser consultado através do serviço `sysconf()`. Este último serviço (ver `man`) fornece o valor de inúmeras constantes dependentes da implementação. A que aqui nos interessa é designada por `_SC_CLK_TCK`.

O tempo retornado por `times()` é relativo a um instante arbitrário anterior (o tempo de arranque do sistema) sendo por isso necessário fazer uma chamada no início do programa e outra perto do final. O tempo total decorrido será a diferença entre essas duas medições.

Apresenta-se de seguida um exemplo do método de medida de tempos.

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

void main(void)
{
  clock_t start, end;
  struct tms t;
  long ticks;
  int k;

  start = times(&t);                          /* início da medição de tempo */
  ticks = sysconf(_SC_CLK_TCK);

  for (k=0; k<100000; k++)
    printf("Hello world!\n");

  printf("\nClock ticks per second: %ld\n", ticks);

  end = times(&t);                              /* fim da medição de tempo */

  printf("Clock:                %4.2f s\n", (double)(end-start)/ticks);
  printf("User time:            %4.2f s\n", (double)t.tms_utime/ticks);
  printf("System time:          %4.2f s\n", (double)t.tms_stime/ticks);
  printf("Children user time:   %4.2f s\n", (double)t.tms_cutime/ticks);
  printf("Children system time: %4.2f s\n", (double)t.tms_cstime/ticks);
}
```