Miguel Pimenta Monteiro

A API Win32 dos Sistemas Operativos Windows

Alguns aspectos

1. Programas em C em Win32

1.1 Início e terminação de programas

1.1.1 Início

Os sistemas operativos Windows (Windows 9x/Me/NT/2000) partilham uma API de programação comum, geralmente designada por Win32. Essa API de programação é constituída por um elevado número de serviços ou funções, definidos utilizando a linguagem de programação C. Iremos ver, no que se segue, alguns aspectos dessa API, mas apenas para o chamado modo consola, em que a interface com o utilizador se realiza através de uma janela de texto (consola). O outro modo de funcionamento de programas, usando a API Win32, é o modo GUI (*Graphical User Interface*), que compreende mais algumas centenas de serviços.

Quando se solicita ao S.O. a execução de um novo programa (serviço `CreateProcess()` em Win32), 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/consola (também podem ser redireccionados).

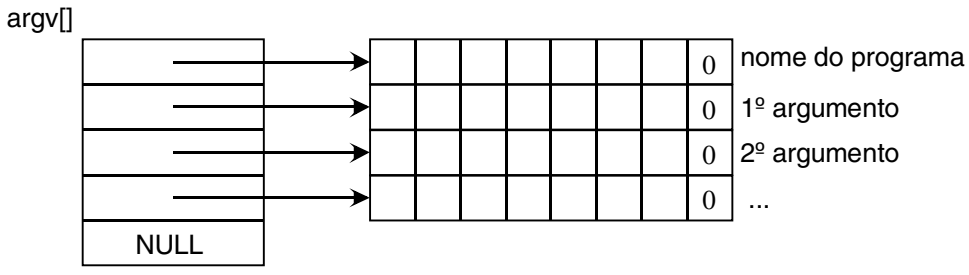
A função `main()` pode ser definida num programa em C de muitas formas:

```
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 transmite ao Sistema Operativo.

Quando se invoca um programa é possível passar-lhe parâmetros, que são um conjunto de 0 ou mais *strings*, separadas por espaços. Esses parâmetros podem depois ser acedidos na função `main()` através dos seus 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 (podendo incluir 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 quer em Win32, quer em 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, definidas na biblioteca standard do C:

```
#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 <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 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 outra função da biblioteca standard da linguagem:

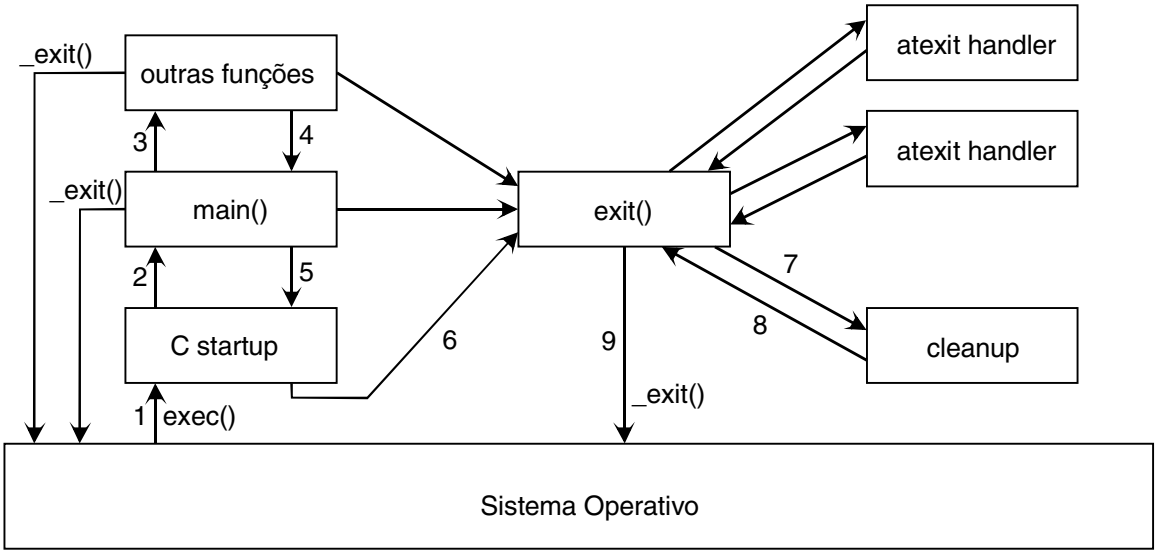
```
#include <stdlib.h>

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 correpondem de perto a chamadas directas Win32 (`ExitProcess()` e `TerminateProcess()`).



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.

No caso da API Win32, sempre que chama um dos seus serviços, o seu valor de retorno indica se ocorreu ou não um erro. Consoante o tipo de serviço, a ocorrência de um erro pode ser indicada através de um valor booleano (*false* indica erro), ou de um valor inteiro ou apontador especial (p. ex. o valor `NULL`). No entanto, o retorno de um serviço em erro pode ter diversas causas que não são discriminadas no valor de retorno. Para se obter a causa específica do erro é necessário invocar uma outra função Win32, que deve ser chamada imediatamente após o retorno em erro de qualquer serviço (terá de qualquer maneira de ser invocada antes da próxima chamada a um outro serviço Win32). Essa função é:

```
#include <windows.h>

DWORD GetLastError(void);
```

Retorna o código de erro da última chamada a um serviço Win32. Se na última chamada não ocorreu nenhum erro o valor retornado é 0. Se ocorreu, é retornado um código que indica especificamente a causa do erro. Estão definidos algumas centenas de códigos de erros.

Cada código de erro retornado por `GetLastError()` está associado a uma constante simbólica (com nome) definida num ficheiro de inclusão inserido em "`windows.h`". Algumas dessas constantes são, por exemplo:

```
ERROR_FILE_NOT_FOUND
ERROR_ACCESS_DENIED                 (falta de permissões de acesso)
```

```
ERROR_NOT_ENOUGH_MEMORY
ERROR_NOT_READY
...
```

Todas as funções e tipos da API Win32 estão declarados em diversos ficheiros de inclusão da linguagem C (com a extensão ".h"). No entanto existe sempre um ficheiro de inclusão "windows.h" que inclui todos os outros. Assim, sempre que se usa qualquer serviço ou função Win32, basta incluir no nosso programa o referido ficheiro "windows.h". A partir deste momento, nas caixas de definição dos serviços Win32, não se indicará mais qual o ficheiro de inclusão que contém a declaração desse serviço (é sempre "windows.h", directa ou indirectamente).

A API Win32 define uma série de tipos (dos parâmetros dos serviços e dos seus valores de retorno) que estão mapeados para os tipos standard da linguagem C nos ficheiros de inclusão, usando a instrução `typedef ...`. Alguns desses tipos são os seguintes:

`WORD`	(inteiro sem sinal de 16 bits - `unsigned short`)
`DWORD`	(inteiro sem sinal de 32 bits - `unsigned long`)
`BOOL`	(representa um valor booleano, associa as constantes `TRUE` e `FALSE`)
`HANDLE`	(representa um objecto do S.O., p. ex. ficheiro, processo, *thread*, ..., através de um valor inteiro único no sistema)

Estes tipos (e outros) são usados com muita frequência na declaração dos serviços Win32 mas, como já se disse, estão todos definidos, directa ou indirectamente em "windows.h".

Geralmente, quando ocorre um erro, queremos imprimir uma mensagem contendo uma descrição da causa do erro. No entanto, devido às possíveis numerosas causas de erros, isso seria bastante complicado se apenas dispuséssemos de um código indicativo dessa causa. Felizmente existe um serviço que, dado o código de erro, nos dá uma string, em linguagem corrente, explicativa da causa do erro. Esse serviço é o seguinte:

```
DWORD FormatMessage(DWORD flags, void *source, DWORD messg_id, DWORD lang_id,
                    char *buffer, DWORD size, va_list *args);
```

Este serviço é bastante genérico e serve para obter e formatar mensagens de diversas fontes, incluindo as mensagens de erro de sistema. Apenas para este fim, interessam-nos os parâmetros **`flags`**, **`messg_id`** e **`buffer`**. O parâmetro **`flags`** define o comportamento da função e quais dos outros parâmetros fazem sentido. Para obter as mensagens de erro de sistema dever-se-á passar em **`flags`** os seguintes valores (constantes simbólicas), ligados com o operador lógico *or* bit a bit (|): `FORMAT_MESSAGE_ALLOCATE_BUFFER` e `FORMAT_MESSAGE_FROM_SYSTEM`. Com estes valores, o parâmetro **`source`** deve ser `NULL`, assim como **`args`**. O parâmetro **`lang_id`** poderá ser 0 (idioma do sistema operativo), e **`size`** também. O serviço preenche o string **`buffer`**, que é alocado dinamicamente pela função (para isso passa-se em buffer o endereço de um apontador - ver exemplo abaixo). No parâmetro **`messg_id`** deverá ser passado o código de erro obtido com `GetLastError()`.

A função retorna o número de caracteres escritos em **`buffer`**, ou 0, se tiver ocorrido um erro.

Quando `FormatMessage()` aloca o buffer que contém a mensagem, este terá que ser explicitamente libertado após o seu uso, o que poderá ser feito com a função da biblioteca standard do C `free()` (ou com o serviço Win32 `LocalFree()`).

Poderemos ver de seguida um exemplo que contém uma função genérica de escrita de mensagens de erro, onde passamos como parâmetros uma string inicial a ser escrita antes da

mensagem de sistema e o código de erro:

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

...
void pr_msg(char *header, DWORD err_code)
{
  char *message;

  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
      NULL, err_code, 0, (char *)&message, 0, NULL);
  printf("%s: %s\n", header, message);
  LocalFree(message);
}
...
hf = CreateFile(...);
if (hf == INVALID_HANDLE_VALUE)
  pr_msg("Error opening file", GetLastError());
...
```

No exemplo vê-se também uma chamada à função que foi definida, `pr_msg()`, quando se detecta um erro na tentativa de abertura de um ficheiro (serviço `CreateFile()`).

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 especialmente vocacionado para isso. Em Win32, quando se invoca um programa (ficheiro executável, com a extensão "`.exe`") este passa a constituir um processo no sistema operativo. Por sua vez um processo é visto como uma colecção de fluxos de execução independentes, designados por *threads*. Um processo contém sempre um *thread*, que se inicia na função `main()` do programa, podendo posteriormente criar outros que serão escalonados de forma independente. Se quisermos medir os tempos de utilização do processador de todos os *threads* de um processo deveremos utilizar o serviço:

```
BOOL GetProcessTimes(HANDLE hp, FILETIME *creat_time, FILETIME *exit_time,
                     FILETIME *kernel_time, FILETIME *user_time);
```

O parâmetro **`hp`** representa o processo (handle) do qual queremos recolher os tempos de execução. Um processo pode obter o seu próprio handle, e quem lançou um novo processo também tem acesso a esse handle. `FILETIME` é uma estrutura com 2 campos de 32 bits cada, representando uma quantidade de 64 bits. Os parâmetros **`creat_time`** e **`exit_time`** são endereços de estruturas `FILETIME` que são preenchidas com os intantes de criação e terminação do processo. Estes instantes são expressos como a quantidade de unidades de 100 ns que decorreram desde 1 de Janeiro de 1601, tempo de Greenwich. Os parâmetros **`kernel_time`** e **`user_time`** são preenchidos com o tempo de CPU gasto pelo processo (todos os *threads*) em modo kernel (a executar chamadas ao sistema) e em modo utilizador (a executar instruções do programa) em unidades de 100 ns.

O serviço retorna 0 (`FALSE`) em caso de erro.

A utilização directa da estrutura `FILETIME` é pouco prática. Quando `FILETIME` representa um instante de tempo (caso dos parâmetros **`creat_time`** e **`exit_time`** do serviço anterior), pode ser convertida para outra estutura mais prática chamada `SYSTEMTIME`, cuja definição é a seguinte:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
```

```
        WORD wMonth;
        WORD wDayOfWeek;
        WORD wDay;
        WORD wHour;
        WORD wMinute;
        WORD wSecond;
        WORD wMilliseconds;
    } SYSTEMTIME;
```

A estrutura `SYSTEMTIME` representa o instante em hora, minuto, segundo e milisegundo do dia, mês e ano do calendário que estiver configurado para o utilizador. Indica também o dia da semana no campo `wDayOfWeek` (0 = Domingo, 1 = segunda-feira, etc).

A conversão de `FILETIME` para `SYSTEMTIME` faz-se com o serviço:

`BOOL FileTimeToSystemTime(const FILETIME *ft, SYSTEMTIME *st);`

Em **`ft`** fornece-se o endereço da estrutura `FILETIME` a converter e em **`st`** o endereço da estrutura `SYSTEMTIME` que irá ser preenchida.

Retorna 0 em caso de erro.

Quando `FILETIME` representa um intervalo de tempo (em unidades de 100 ns) pode ser directamente copiada para um inteiro de 64 bits se o compilador suportar. A API Win32 define tipos inteiros de 64 bits como `LONGLONG` e `ULONGLONG` (com e sem sinal).

Os tipos `LONGLONG` e `ULONGLONG` só são mapeados para tipos intrínsecos da linguagem C se o compilador suportar inteiros de 64 bits. O compilador Visual C++ suporta inteiros de 64 bits como extensão da linguagem C. São eles `__int64` e `unsigned __int64`. Ainda usando o compilador Visual C++, estes inteiros podem ser operados aritmeticamente, e escritos directamente ou convertidos para string com `printf()` e `sprintf()`, usando o especificador "`%I64d`" ou "`%I64u`".

Existe um serviço em tudo idêntico a `GetProcessTimes()` para obter os mesmos tempos relativos a um *thread* específico de um processo. É apenas necessário fornecer o handle do *thread*, em vez do processo. Esse serviço chama-se `GetThreadTimes()`.

É ainda de referir que estes serviços só são suportados pelos sistemas Windows NT/2000.

Apresenta-se de seguida um exemplo do método de medida de tempos, através de um programa (`timep.exe`) que lança como processo um outro programa, espera que termine, e obtém os tempos de execução do processo lançado.

```
/* timep.c */
#include <windows.h>
#include <stdio.h>

char *skip_arg(char *line)
{
  while (*line != '\0' && *line != ' ' && *line != '\t')
    line++;
  while (*line == ' ' || *line == '\t')
    line++;
  return line;
}
```

```
void main(void)
{
  STARTUPINFO stup;
  PROCESS_INFORMATION pinfo;
  LONGLONG ct, et, kt, ut, elapse;
  SYSTEMTIME sct, set;
  char *call;

  call = GetCommandLine();    // string com a linha de comando
  call = skip_arg(call);      // salta o 1.o argumento
  GetStartupInfo(&stup);      // necessário para a criação de um novo processo

  CreateProcess(NULL, call, NULL, NULL, TRUE, NORMAL_PRIORITY_CLASS, NULL, NULL,
                &stup, &pinfo);  // cria novo processo com parâmetros de chamada
  WaitForSingleObject(pinfo.hProcess, INFINITE);  // espera que termine

  GetProcessTimes(pinfo.hProcess, (FILETIME *)&ct, (FILETIME *)&et,
                  (FILETIME *)&kt, (FILETIME *)&ut);

  CloseHandle(pinfo.hProcess);  // fecha handle do processo terminado
  CloseHandle(pinfo.hThread);   // fecha handle do thread terminado

  elapse = et - ct;
  FileTimeToSystemTime((FILETIME *)&ct, &sct);
  FileTimeToSystemTime((FILETIME *)&et, &set);
  printf("Start time:  %02d:%02d:%02d.%03d\n", sct.wHour, sct.wMinute,
                                               sct.wSecond, sct.wMilliseconds);
  printf("End time:    %02d:%02d:%02d.%03d\n", set.wHour, set.wMinute,
                                               set.wSecond, set.wMilliseconds);
  printf("Elapsed:     %.3f ms\n", (double)elapse/10000);
  printf("Kernel:      %.3f ms\n", (double)kt/10000);
  printf("User:        %.3f ms\n", (double)ut/10000);
}
```

Poderíamos usar como programa a cronometrar o seguinte:

```
/* exp.c */
#include <stdio.h>

void main(void)
{
  int k;

  for (k=0; k<10000; k++)
    printf("Hello world!\n");
}
```

A medida de tempos deste programa seria então feita como:

```
D:\progs>timep exp.exe
...
Hello world!
Start time:  13:10:27.624
End time:    13:10:33.713
Elapsed:     6088.755 ms
Kernel:      150.216 ms
User:        60.086 ms
```

Apesar das medidas de tempo efectuadas com `GetProcessTimes()` serem em unidades de 100 ns (1 décimo de µs) a resolução do relógio utilizado é apenas de cerca de 10 ms nas arquitecturas intel. No entanto, o hardware dos PC's contém um contador com uma resolução superior a 1 µs que pode ser acedido através de serviços Win32 próprios. Com este contador é

possível medir intervalos de tempo com essa resolução. O contador começa a funcionar quando se liga o PC.

A frequência do contador (número de *ticks* por segundo) pode ser obtida com o serviço:

BOOL QueryPerformanceFrequency(LARGE_INTEGER *freq);

Preenche a estrutura LARGE_INTEGER (uma união em que um dos componentes é um inteiro de 64 bits), cujo endereço é passado em **freq**, com a frequência do contador de alta resolução, se existir.

Retorna 0 no caso do contador não existir.

O valor do contador pode ser lido em qualquer momento com:

BOOL QueryPerformanceCounter(LARGE_INTEGER *count);

Preenche a posição apontada por **count** com o valor corrente do contador de alta resolução.

Retorna 0 no caso do contador não existir.

Um exemplo da utilização destes serviços para medir intervalos de tempo pode ser:

```
void main(void)
{
  LONGLONG freq, count1, count2;

  QueryPerformanceCounter((LARGE_INTEGER *)&count1);
  QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
  printf("Frequência: %I64d ticks/s\n", freq);
  QueryPerformanceCounter((LARGE_INTEGER *)&count2);
  printf("Contagem: 1: %I64d 2: %I64d dif: %I64d\n", count1, count2,
                                                     count2-count1);
  printf("Tempo: %I64d µs\n", (count2-count1)*1000000/freq);
}
```

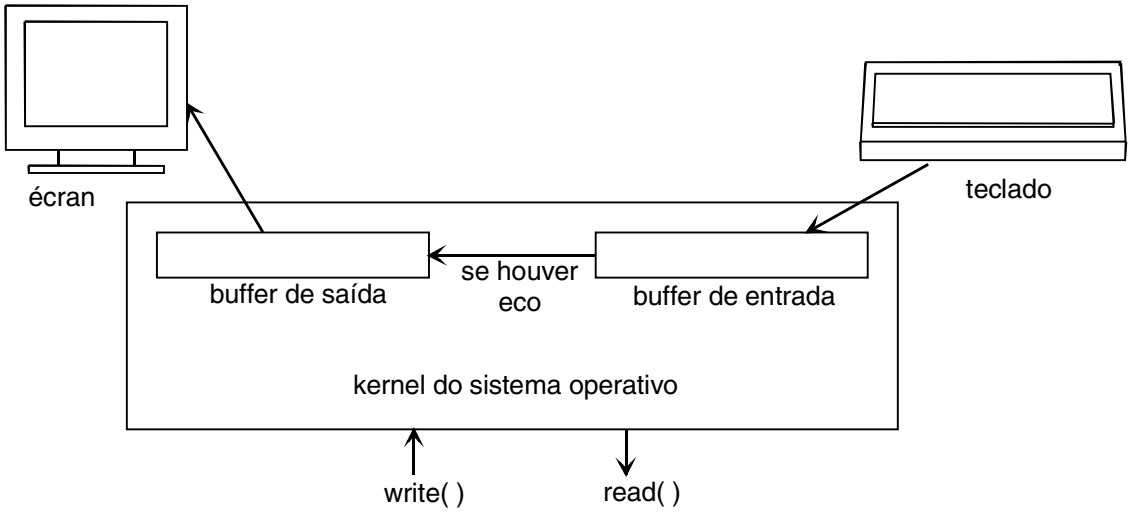
Um possível resultado da execução do programa poderia ser:

```
D:\progs>freq
Frequência: 1193182 ticks/s
Contagem: 1: 19547897062 2: 19547926004 dif: 28942
Tempo: 24256 µs
```

2. Entrada/Saída na consola

2.1 Leitura e escrita na consola

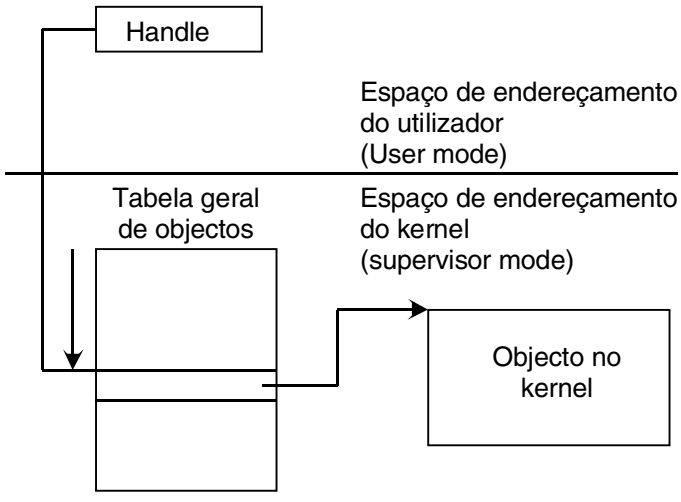
A consola (teclado + écran) é vista geralmente nos S.O.'s como um ou mais ficheiros onde se pode ler ou escrever texto. Esses ficheiros são normalmente abertos pela rotina de C *startup*. A biblioteca standard do C inclui inúmeras funções de leitura e escrita directa nesses ficheiros (`printf()`, `scanf()`, `getchar()`, `putchar()`, `gets()`, `puts()`, ...). No entanto podemos aceder também a esses periféricos através de serviços dos S.O.'s.



Implementação típica da consola num sistema operativo

Em Win32 os ficheiros e outros objectos criados no kernel são acedidos através de um número (handle) que os representa. Através da invocação de vários serviços do S.O. o kernel cria para os seus processos estruturas que representam objectos geridos pelo sistema (ficheiros, processos, *threads*, *pipes*, semáforos, blocos de memória partilhada, etc.). Esses objectos são mantidos pelo kernel no seu próprio espaço de endereçamento (memória exclusiva do kernel) e nunca são acedidos directamente pelos processos que solicitaram a sua criação. Estes apenas obtêm um handle que mais não é do que um índice para a tabela geral de objectos mantida pelo kernel. Quando um processo necessita de manipular um destes objectos terá de o fazer através da invocação de um serviço do S.O., passando-lhe o respectivo handle. Será depois o kernel que efectuará a manipulação pedida.

Na figura seguinte pode ver-se um esquema mostrando a relação entre um handle no espaço de endereçamento de um processo e o objecto a que ele se refere, mantido no kernel.



A consola em Win32 é uma janela de texto para onde são enviadas as mensagens escritas na saída standard e no dispositivo (ficheiro) de erro standard. Se essa janela que representa a consola estiver activa, o teclado fica também associado à entrada standard. Para obter os handles que representam estes 3 componentes da consola devemos usar o serviço:

```
HANDLE GetStdHandle(DWORD std_handle);
```

Obtém o handle de uma das três componentes da consola. O parâmetro **std_handle** pode ser apenas uma das seguintes três constantes simbólicas:
`STD_INPUT_HANDLE` - entrada standard (geralmente teclado);
`STD_OUTPUT_HANDLE` - saída standard (geralmente écran);
`STD_ERROR_HANDLE` - dispositivo onde são escritas mensagens de erro (geralmente écran).

Retorna o handle correspondente a um dos componentes da consola ou o valor `INVALID_HANDLE_VALUE`, no caso de erro.

Os três componentes da consola podem ser redireccionados utilizando os símbolos da shell (`<`, `>`, ou `|`) ou programaticamente utilizando outro serviço:

```
BOOL SetStdHandle(DWORD std_handle, HANDLE hf);
```

Redirecciona um componente da consola (**std_handle**) para um outro ficheiro cujo handle é passado em **hf**. O parâmetro **std_handle** pode tomar um dos 3 valores indicados no serviço anterior.

Retorna 0 (`FALSE`) no caso de erro.

O método normal para redireccionar programaticamente algum dos componentes da E/S standard é chamar `SetStdHandle()`, seguido da chamada de `GetStdHandle()`.

A consola possui uma série de propriedades que podem ser modificadas. Por defeito, a leitura (do teclado) é feita linha a linha (apenas após `CR`) e com eco (no écran).

Para aceder e modificar as propriedades da consola usam-se os serviços:

```
BOOL GetConsoleMode(HANDLE hcons, DWORD *mode);
```

Preenche **mode** com as flags de propriedades activas para o componente da consola indicado pelo handle **hcons**.

Retorna 0 no caso de erro.

```
BOOL SetConsoleMode(HANDLE hcons, DWORD mode);
```

Activa as propriedades indicadas em **mode** para o componente da consola indicado em **hcons**.

Retorna 0 no caso de erro.

As propriedades da consola são indicadas através de constantes simbólicas ligadas entre si com o operador *or* bit a bit (`|`).

Algumas das propriedades são:

ENABLE_LINE_INPUT - Leituras apenas após CR. (activa por defeito);
ENABLE_ECHO_INPUT - Caracteres entrados no teclado ecoados para o écran. (activa por defeito);
ENABLE_PROCESSED_INPUT - Processa alguns caracteres especiais como Ctrl-C, backspace e outros, antes da leitura no teclado. (activa por defeito);
ENABLE_PROCESSED_OUTPUT - Processa certos caracteres especiais antes de estes atingirem a janela da consola. (activa por defeito);
ENABLE_WRAP_AT_EOL_OUTPUT - Muda de linha automaticamente, quando o comprimento desta excede o comprimento do écran (da janela que o representa). (activa por defeito);

Exemplo: Colocar o teclado no modo de leitura imediata sem eco.

```
hConsole  =  GetStdHandle(STD_INPUT_HANDLE);
GetConsoleMode(hConsole, &mode);
SetConsoleMode(hConsole, 0);          /* Desactiva todas as propriedades */
...                                   /* Utiliza o teclado nesse modo */
SetConsole Mode(hConsole, mode);      /* Repõe situação */
```

Para leitura do teclado e escrita no écran, e uma vez obtidos os respectivos handles, podemos usar os serviços normais de leitura/escrita em ficheiros (descritos mais à frente). No entanto a API Win32 possui também serviços específicos para leitura/escrita na consola. Têm a desvantagem de só poderem ser usados se os componentes da consola não tiverem sido redireccionados para outros ficheiros.

```
BOOL ReadConsole(HANDLE hcons_input, void *buffer, DWORD nch_to_read,
                 DWORD *nch_read, void *reserved);
```

Tenta ler do teclado **nch_to_read** caracteres e coloca-os no buffer apontado por **buffer**. O número de caracteres efectivamente lido é indicado através de **nch_read**. O parâmetro **reserved** deve ser NULL. O parâmetro **hcons_input** terá de ser o handle do componente de entrada standard da consola.

Retorna 0 no caso de erro.

Se o teclado não tiver a propriedade ENABLE_LINE_INPUT só faz sentido ler um carácter de cada vez. Povavelmente, mesmo que se tente ler mais do que um carácter, o serviço retornará quando o utilizador teclar o primeiro, trazendo **nch_read** o valor 1.

```
BOOL WriteConsole(Handle hcons_output, void *buffer, DWORD nch_to_write,
                  DWORD *nch_written, void *reserved);
```

Tenta escrever na consola **nch_to_write** caracteres provenientes do buffer apontado por **buffer**, previamente preenchido. O número de caracteres efectivamente escrito é indicado através de **nch_written**. O parâmetro **reserved** deve ser sempre NULL, enquanto que **hcons_output** terá de ser um handle do componente de entrada ou do erro standard.

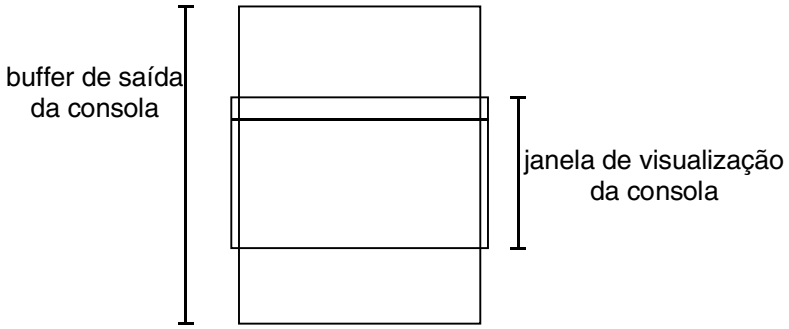
Retorna 0 no caso de erro.

2.2. Outros serviços da consola

Existem muitas outras funções que lidam com a consola do Windows. Alguns dos mais utilizados são descritos a seguir.

A área de escrita da consola é representada por um buffer de memória (buffer de saída) com uma certa capacidade de linhas e colunas de caracteres. Geralmente a janela da consola apenas deixa ver algumas dessas linhas (as últimas escritas), tendo quase sempre menores

dimensões que o buffer de saída. É possível deslizar o conteúdo do buffer de saída na janela usando os controlos de *scroll* desta. Pode observar-se na figura seguinte uma possível relação do posicionamento da área de visualização da janela da consola relativamente ao buffer de saída.



Para conhecer o programaticamente as características do buffer de saída e da janela da consola podemos usar o serviço:

```
BOOL GetConsoleScreenBufferInfo(HANDLE hcons_output,
                                CONSOLE_SCREEN_BUFFER_INFO *cons_info);
```

Dado o handle da saída standard em **hcons_output**, este serviço preenche uma estrutura CONSOLE_SCREEN_BUFFER_INFO apontada por **cons_info** com informação sobre a consola e buffer de saída.

Retorna 0 em caso de erro.

A definição da estrutura CONSOLE_SCREEN_BUFFER_INFO é então:

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
  COORD      dwSize;
  COORD      dwCursorPosition;
  WORD       wAttributes;
  SMALL_RECT srWindow;
  COORD      dwMaximumWindowSize;
} CONSOLE_SCREEN_BUFFER_INFO ;
```

O campo dwSize especifica o tamanho do buffer de saída, enquanto que dwCursorPosition especifica o local onde está o cursor. srWindow especifica o canto superior esquerdo e inferior direito, no buffer de saída, da área visível na janela da consola. wAttributes indica as cores dos caracteres e de fundo da janela da consola, enquanto que dwMaximumWindowSize indica o tamanho máximo que a janela pode ter. As estruturas COORD e SMALL_RECT são definidas como:

```
typedef struct _COORD {
  SHORT X;
  SHORT Y;
} COORD;

typedef struct _SMALL_RECT {
  SHORT Left;
  SHORT Top;
  SHORT Right;
  SHORT Bottom;
} SMALL_RECT;
```

As cores de escrita dos caracteres e cor de fundo da janela da consola são especificadas através

da ligação *or* bit a bit das seguintes constantes:

`FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY.`

As cores de foreground (texto) e background (fundo) podem ser combinadas, dando 8 variações possíveis. Além disso pode juntar-se o atributo `INTENSITY`, resultando em tonalidades mais claras. Assim temos 16 possíveis cores para texto e fundo.

A programação dessas cores pode ser efectuada com:

`BOOL SetConsoleTextAttribute(HANDLE cons_output, WORD attributes);`

Programa os atributos (cor) do texto e fundo através da ligação com *or* bit a bit das constantes mencionadas atrás, passadas no parâmetro **`attributes`**.

O serviço seguinte é muito utilizado para apagar parte ou a totalidade do buffer de saída.

```
BOOL FillConsoleOutputCharacter(HANDLE hcons_output, char ch, DWORD len,
                                COORD coord, DWORD *nch_written);
```

Transfere para a consola o mesmo carácter, indicado em **`ch`**, um número de vezes indicado em **`len`**, a partir da posição indicada em **`coord`**. O número de caracteres efectivamente escrito é indicado através de **`nch_written`**. O canto superior esquerdo tem coordenadas (0, 0).

Retorna 0 no caso de erro.

É possível apagar o conteúdo da janela da consola usando o seguinte pedaço de código:

```
CONSOLE_SCREEN_BUFFER_INFO cons_info;
COORD origin = {0, 0};
DWORD n;

hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
GetConsoleScreenBufferInfo(hConsole, &cons_info);
FillConsoleOutputCharacter(hConsole, ' ', cons_info.dwSize.X*cons_info.dwSize.Y,
                           origin, &n);
```

É também possível colocar o cursor em qualquer posição na janela da consola invocando:

`BOOL SetConsoleCursorPosition(HANDLE hcons_output, COORD cursor_pos);`

Coloca o cursor na posição indicada por **`cursor_pos`**. O parâmetro **`hcons_output`** deverá ser o handle de um componente de escrita na consola.

Retorna 0 em caso de erro.

Nota sobre a escrita de caracteres acentuados na consola:
Em Windows NT/2000 as janelas gráficas e a consola usam códigos de caracteres diferentes (ANSI (ISO8859-1) e DOS Code Pages, respectivamente). As strings que são escritas com um editor gráfico (como é o do Visual C++) ficam codificadas com os caracteres gráficos e não aparecem correctamente na consola se esta utilizar um code page herdado do DOS.

Para evitar o aparecimento de caracteres errados podemos utilizar várias soluções. Uma delas será usar um editor de consola. Outra será usar um editor gráfico e traduzir as strings que contêm caracteres acentuados para a codificação da consola. Para isso existe o serviço:

```
BOOL CharToOem(char *source, char *destination);
```

Traduz a string **source** codificada em ANSI, para a string **destination**, codificada no code page DOS activo. As duas strings têm tamanhos iguais; apenas muda a codificação.

Uma terceira solução será a utilização da codificação ANSI na consola. Isso pode ser feito na própria consola com o comando chcp e especificando o code page 1252 (ANSI), ou programaticamente com o serviço:

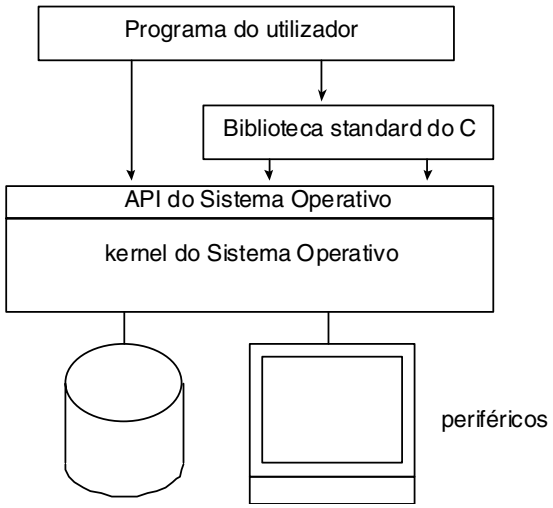
```
BOOL SetConsoleOutputCP(UINT cp_id);
```

Passa a usar na consola a codificação do code page **cp_id**, se for possível.

Para que esta terceira solução funcione é necessário usar para a consola uma fonte *truetype* (p. ex. Lucida Console) e não uma fonte bitmapped (raster), como geralmente é a configuração por defeito.

3. Serviços de acesso a ficheiros

A biblioteca standard da linguagem C (*ANSI C library*) contém um conjunto rico de funções de acesso, criação, manipulação, leitura e escrita de ficheiros em disco e dos 3 componentes da consola. A utilização dessas funções torna os programas independentes do Sistema Operativo, uma vez que a biblioteca standard é a mesma para todos os S.O.'s. No entanto introduz uma camada extra entre o programa e o acesso aos objectos do S.O. (ficheiros, consola, etc) como se ilustra na figura seguinte.



A posição da biblioteca standard do C

Assim, para se conseguir um pouco mais de performance, é possível chamar directamente os serviços dos S.O.'s, pagando-se o preço dos programas assim desenvolvidos não poderem ser compilados, sem alterações, noutros Sistemas Operativos.

Apresentam-se de seguida alguns dos serviços referentes à criação, leitura e escrita, fecho e eliminação de ficheiros nos sistemas operativos Windows, conforme especificados na API Win32.

3.1. Criação, acesso e eliminação de ficheiros

Os ficheiros em Win32, à semelhança dos 3 componentes da consola, são também representados por handles. É este handle que é passado aos diversos serviços para efectuar as acções que pretendemos sobre os ficheiros. Como é bem conhecido, o sistema de ficheiros no sistema operativo Windows é representado por uma hierarquia de directórios em árvore, onde em cada um desses directórios é possível colocar ficheiros. A raiz dessa árvore representa uma partição de um disco, devidamente formatada num dos sistemas reconhecidos, e é designada por uma letra do alfabeto (p. ex. C:/).

Para criar, ou abrir um ficheiro já existente, obtendo o respectivo handle, temos de invocar o serviço `CreateFile()`, que contém 7 parâmetros:

```
HANDLE CreateFile(char *name, DWORD access, DWORD share_mode,
                  SECURITY_ATTRIBUTES *sa,  DWORD create,
                  DWORD attrs_flags, HANDLE htemplate);
```

Abre um ficheiro já existente ou cria um novo ficheiro.

Retorna um `HANDLE` do ficheiro aberto, ou `INVALID_HANDLE_VALUE` no caso de erro.

Parâmetros:
name - nome do ficheiro incluindo o path; esta função pode ser utilizada para abrir outros objectos que não ficheiros (*pipes*, portos série, *mailslots*, …), tendo então convenções próprias.

access - indica o tipo de acesso utilizando as constantes simbólicas GENERIC_READ e GENERIC_WRITE; para abrir um ficheiro simultaneamente para leitura e escrita fazer o *or* bit a bit das duas constante (|).

share_mode - pode usar-se a combinação (*or* bit a bit dos seguintes valores):
- 0 - o ficheiro não pode ser partilhado;
- FILE_SHARE_READ - outros processos podem abrir este mesmo ficheiro para leitura;
- FILE_SHARE_WRITE - outros processos podem abrir este mesmo ficheiro para escrita.

No caso de partilha devem usar-se outros mecanismos para impedir actualizações simultâneas do mesmo local do ficheiro (*locks*).

sa - especifica alguns parâmetros de segurança; usar NULL para a segurança por defeito.

create - especifica a forma de abertura ou criação; pode ser uma combinação dos seguintes valores (|):
- CREATE_NEW - falha a chamada se já existir um ficheiro com o mesmo nome;
- CREATE_ALWAYS - um ficheiro já existente será apagado e criado um novo;
- OPEN_EXISTING - falha a chamada se o ficheiro não existir;
- OPEN_ALWAYS - abre um ficheiro existente, criando-o se não existir
- TRUNCATE_EXISTING - o comprimento do ficheiro passa a 0 se já existir

attrs_flags - mais atributos e flags relativos ao ficheiro a abrir ou criar; pode usar-se uma combinação dos seguintes valores (|), entre outros:
- FILE_ATTRIBUTE_NORMAL - nada de especial;
- FILE_ATTRIBUTE_READONLY - não é possível escrever ou apagar o ficheiro;
- FILE_FLAG_DELETE_ON_CLOSE - o ficheiro é apagado quando o último handle que o refere for fechado;
- FILE_FLAG_OVERLAPPED - usado nos acessos assíncronos ao ficheiro;
- FILE_FLAG_WRITE_THROUGH - não são usadas caches de escrita;
- FILE_FLAG_NO_BUFFERING - sem buffers ou caches; neste caso as leituras ou escritas têm de ser múltiplos do tamanho de um sector no disco;
- FILE_FLAG_RANDOM_ACCESS - vai-se fazer um acesso não sequencial; apenas uma indicação ao S.O. para optimizar a cache;
- FILE_FLAG_SEQUENTIAL_ACCESS - vai-se fazer um acesso sequencial; apenas uma indicação ao S.O. para optimizar a cache.

htemplate - handle de um ficheiro GENERIC_READ que especifica os atributos e flags a aplicar ao ficheiro criado nesta chamada, ignorando o parâmetro anterior; normalmente utiliza-se o valor NULL para este parâmetro.

As chamadas mais frequentes a este serviço são para abrir um ficheiro já existente para leitura, o que pode ser feito simplesmente por:

```
hfile = CreateFile(nome, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
```

ou para criar um novo ficheiro, para escrita:

```
hfile = CreateFile(nome, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL, NULL);
```

Claro que é possível combinar estes modos de abertura.

A leitura e escrita em ficheiros e noutros objectos, como os componentes da consola podem ser efectuadas com os serviços genéricos Win32 de leitura e escrita:

```
BOOL ReadFile(HANDLE hf, void *buffer, DWORD nb_to_read, DWORD * nb_read,
              OVERLAPPED *over);

BOOL WriteFile(HANDLE hf, void *buffer, DWORD nb_to_write, DWORD * nb_written,
               OVERLAPPED *over);
```

Lê ou escreve no ficheiro (ou outro objecto) representado pelo handle **hf** um número de bytes provenientes de **buffer** igual a **nb_to_read** ou **nb_to_write**; o número de bytes efectivamente lido ou escrito é indicado através de **nb_read** ou **nb_written**. Não estando a flag de abertura do ficheiro FILE_FLAG_OVERLAPPED activa, deve **over** ser NULL.

Estes serviços retornam 0 se ocorrer um erro.

Para os sistemas de ficheiros Windows, os ficheiro não têm qualquer estrutura interna sendo sempre vistos como uma pura sequência de bytes. A estrutura de um ficheiro apenas pode ser imposta ou interpretada pelos programas do utilizador, quando da escrita ou leitura dos seus dados. Após uma leitura ou escrita bem sucedida, o apontador do ficheiro (indicador da posição no ficheiro (em bytes) a partir da qual se faz a próxima leitura ou escrita) é incrementado do número de bytes efectivamente lidos ou escritos.

Quando um ficheiro é aberto é posicionado no seu início (posição 0). Pode ser reposicionado programaticamente através do serviço:

```
DWORD SetFilePointer(HANDLE hf, long dist_to_move, long *dist_to_move_high,
                     DWORD move_method);
```

Este serviço suporta distâncias a mover, indicadas em bytes, na posição do apontador do ficheiro de 32 ou 64 bits; se **dist_to_move_high** for NULL, **dist_to_move** representa o número de bytes a mover na posição do apontador de ficheiro; essa distância é tomada com sinal (pode ser negativa, significando então, recuar). No caso de **dist_to_move_high** apontar para um valor, esse valor é tomado como a parte mais significativa de um inteiro de 64 bits (**dist_to_move** contém os 32 bits menos significativos).
O parâmetro **move_method** especifica uma de três origens possíveis para o deslocamento do apontador de ficheiro, através de uma das constantes simbólicas:

- FILE_BEGIN - início do ficheiro
- FILE_CURRENT - posição actual do ficheiro
- FILE_END - fim do ficheiro

A função retorna a posição do ficheiro depois do deslocamento. Se essa posição necessitar mais de 32 bits, a parte mais significativa é colocada no local apontado por **dist_to_move_high**, que não poderá ser NULL. A indicação de erro é dada pelo retorno da constante INVALID_SET_-FILE_POINTER.

Um exemplo da utilização e teste de erro do serviço pode ser:

```
// caso 1: dist_to_move_high = NULL

ptr = SetFilePointer(hFile, lDistance, NULL, FILE_BEGIN) ;

if (ptr == INVALID_SET_FILE_POINTER) {
    error = GetLastError() ;
  // Tratar o erro
  ...
}
```

```
// caso 2: dist_to_move_high != NULL

ptr_low = SetFilePointer (hFile, lDistLow, &lDistHigh, FILE_BEGIN) ;
if (ptr_low == INVALID_SET_FILE_POINTER &&          // INVALID_SET_FILE_POINTER
   (error = GetLastError()) != NO_ERROR ) {         // pode ser um valor válido
  // Tratar o erro
  ...
}
```

Os ficheiros e outros objectos representados por um handle fecham-se chamando o serviço:

`BOOL CloseHandle(HANDLE hobj);`

Fecha o objecto representado por **`hobj`**.

Retorna 0 no caso de erro.

Sempre que qualquer objecto representado por um handle já não for necessário a um programa, o handle deverá ser fechado usando quase sempre `CloseHandle()`. O objecto representado pelo handle e mantido pelo kernel do sistema operativo só será libertado, e a sua memória recuperada, quando todos os handles que o referem forem fechados.

Pode-se apagar fisicamente do disco um ficheiro usando:

`BOOL DeleteFile(char *file_name);`

Apaga do disco o ficheiro designado. O ficheiro deve estar fechado e o utilizador do serviço deve ter permissão para o fazer.

Retorna 0 em caso de erro.

3.2. Utilização de ficheiros temporários

Por vezes é necessário criar ficheiros temporários, existentes apenas durante a execução de um programa.

Os seguintes serviços podem ser úteis para formar os nomes desses ficheiros temporários:

`DWORD GetTempPath(DWORD size, char *buffer);`

Obtém o nome do directório que o sistema usa para criar ficheiros temporários. Em **`buffer`** indica-se o endereço de um buffer que irá ser preenchido com o nome pedido (que é terminado com `\`, p. ex. `c:\temp\`) e em **`size`** o seu tamanho, sem contar com o 0 terminador das strings em C.

Retorna o número de caracteres escritos ou necessários para **`buffer`**. Se o valor retornado for maior do que **`size`**, **`buffer`** não é preenchido. O retorno de 0 indica um erro.

O tamanho máximo de um nome de ficheiro ou directório, incluindo todo o caminho desde a raiz de um sistema de ficheiro nunca pode exceder `MAX_PATH`, que é uma constante definida no sistema operativo.

Sabendo o nome do directório de temporários poder-se-á obter um nome de ficheiro único e ao mesmo tempo criá-lo, sem o abrir e sem obter o seu handle com:

```
UINT GetTempFileName(char *path_name, const char *prefix, UINT unique,
                     char *temp_name);
```

Encontra um nome de ficheiro único e cria-o no directório indicado em **path_name**. O nome do ficheiro criado usa os 3 primeiros caracteres de **prefix** concatenados com um valor hexadecimal que garante a unicidade do nome no directório indicado (mais a extensão `.tmp`). Para este comportamento o parâmetro **unique** tem de ser 0. O nome utilizado é retornado em **temp_name**. Para garantir a suficiência do buffer cujo endereço se passa em **temp_name** este deve ter um tamanho `MAX_PATH+1`. Para criar o ficheiro no directório corrente dever-se-á passar "." em **path_name**.

Retorna o valor numérico que faz parte do nome do ficheiro criado, ou 0 se ocorrer um erro.

Se o serviço anterior suceder, o ficheiro temporário é criado mas é fechado. Para o utilizar é agora necessário abri-lo com `CreateFile()`. Se quisermos apagar o ficheiro quando do seu fecho, é possível especificar no parâmetro **attrs_flags** de `CreateFile()` a flag `FILE_FLAG_DELETE_ON_CLOSE` ou, em alternativa, apagá-lo com `DeleteFile()`, depois de fechado.

4. Ficheiros e directórios

Como é bem conhecido, os directórios contêm ficheiros e possivelmente outros sub-directórios. Cada directório não é mais que um ficheiro especial onde são armazenadas informações relativas a cada ficheiro que contém. Essa informação compreende o nome do ficheiro, tempos de criação e acesso, atributos do ficheiro e o local no disco onde se encontram os dados que o ficheiro contém.

4.1. Informação sobre ficheiros e handles

Grande parte das características de um ficheiro podem ser obtidas através de um serviço que preenche uma estrutura com esses valores. O ficheiro deverá ser previamente aberto e obtido o seu handle:

```
BOOL GetFileInformationByHandle(HANDLE hf, BY_HANDLE_FILE_INFORMATION *info);
```

Preenche a estrutura apontada por info com diversa informação relativa ao ficheiro representado pelo handle hf.

Retorna 0 em caso de erro.

A informação é colocada numa estrutura `BY_HANDLE_FILE_INFORMATION`, cuja definição é a seguinte:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    dwVolumeSerialNumber;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    nNumberOfLinks;
  DWORD    nFileIndexHigh;
  DWORD    nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION;
```

No campo `dwFileAttributes` colocam-se os atributos do ficheiro como a ligação *or* bit a bit de certas constantes. Algumas dessas constantes, cujo significado é evidente, são:

- `FILE_ATTRIBUTE_ARCHIVE`
- `FILE_ATTRIBUTE_DIRECTORY`
- `FILE_ATTRIBUTE_COMPRESSED`
- `FILE_ATTRIBUTE_ENCRYPTED`
- `FILE_ATTRIBUTE_HIDDEN`
- `FILE_ATTRIBUTE_NORMAL`
- `FILE_ATTRIBUTE_READONLY`
- `FILE_ATTRIBUTE_SYSTEM`
- `FILE_ATTRIBUTE_TEMPORARY`

O atributo `FILE_ATTRIBUTE_NORMAL` só está presente quando mais nenhum outro está.

Os campos `ftCreationTime`, `ftLastAccessTime` e `ftLastWriteTime` indicam instantes especificados no formato `FILETIME` (ver *1.3. Medida dos tempos de execução*, atrás). Os campos `nFileSizeHigh` e `nFileSizeLow` formam uma quantidade de 64 bits, indicando o tamanho do ficheiro, em bytes. Os campos `dwVolumeSerialNumber` e `nFileIndexHigh` + `nFileIndexLow` (64

bits) identificam de forma única a partição do disco e o ficheiro, podendo ser usados para comparar se 2 handles se referem, ou não, ao mesmo ficheiro.
A informação obtida em conjunto no serviço anterior pode ser também obtida parcialmente com os serviços:

```
DWORD GetFileAttributes(const char * file_name);
```

Retorna os atributos do ficheiro especificado em **file_name** ou -1 se ocorrer um erro.

```
DWORD GetFileSize(HANDLE hf,  DWORD *file_size_high);
```

Retorna em bytes o tamanho do ficheiro, preenchendo também o parâmetro **file_size_high**. O valor deste parâmetro, junto com o valor de retorno, constitui uma quantidade de 64 bits. Um erro é indicado pelo retorno de -1 (0xFFFFFFFF). No entanto, como este valor também pode ser válido deverá chamar-se sempre GetLastError() para confirmar. Se o tamanho do ficheiro for menor do que 4 Gbytes, **file_size_high** pode ser NULL.

```
BOOL GetFileTime(HANDLE hf,  FILETIME *creation,  FILETIME *last_access,
                 FILETIME *last_write);
```

Obtém os 3 tempos associados a um ficheiro preenchendo os parâmetros **creation**, **last_access** e **last_write**.

Para conhecer se um determinado handle se refere a um ficheiro ou outro objecto podemos ainda usar o serviço:

```
DWORD GetFileType(HANDLE hf);
```

Determina o tipo de um handle. Retorna uma das constantes: FILE_TYPE_UNKNOWN, FILE_TYPE_DISK (ficheiro), FILE_TYPE_CHAR (consola) ou FILE_TYPE_PIPE (pipe anónimo ou com nome).

4.2. Directório corrente

Quando um processo inicia a sua execução tem sempre associado um directório por defeito ou directório corrente. É neste directório onde são abertos e criados os ficheiros relativamente aos quais não se indica nenhum caminho (ou se indica o caminho .\). Inicialmente este directório é, de forma geral, o directório onde está o ficheiro executável do processo.

O directório corrente pode ser conhecido e mudado com os serviços:

```
DWORD GetCurrentDirectory(DWORD buf_length, char *buffer);
```

Preenche **buffer** com o nome do directório corrente. Em **buf_length** indica-se o tamanho de **buffer** que deve ser suficiente para conter o nome do directório corrente.

Retorna o número de caracteres escritos ou necessários para escrever. Se o valor retornado for maior do que **buf_length**, **buffer** não é preenchido. O valor de retorno 0 indica um erro.

```
BOOL SetCurrentDirectory(char *path_name);
```

Muda o directório corrente para path_name.

4.3 Criação e eliminação de directórios

Os directórios não podem ser criados através de `CreateFile()` (mas podem ser abertos), nem apagados com `DeleteFile()`.

Existem 2 serviços exclusivos para a criação e remoção de directórios. São eles:

```
BOOL CreateDirectory(const char *path_name,  SECURITY_ATTRIBUTES *sa);
```

Cria o directório **path_name**, com os atributos de segurança especificados em **sa**. Para usar os atributos de segurança por defeito, sa pode ser `NULL`.

```
BOOL RemoveDirectory(const char *path_name);
```

Remove o directório **path_name**, que deverá estar vazio. A impossibilidade de remoção é indicada com um retorno igual a 0.

4.4. Procura de ficheiros e listagem de directórios

É possível procurar num directório um nome de ficheiro, ou até um padrão, usando os caracteres ditos *wildcards* (? - para representar qualquer carácter, e * - para representar qualquer cadeia de caracteres). Estas pesquisas usam inicialmente um serviço, FindFirstFile(), que faz uma primeira procura e retorna um handle para um objecto do S.O. Usando esse handle é posível encontrar outros ficheiros que satisfaçam o padrão especificado.

Assim para iniciar uma pesquisa usa-se o serviço:

```
HANDLE FindFirstFile(const char *pattern, WIN32_FIND_DATA *file_data);
```

Inicia uma procura de nomes de ficheiros ou sub-directórios indicados em **pattern**. O parâmetro **pattern** pode conter caracteres *wildcard* (? e/ou *) ou ser um nome completamente especificado. Pode ou não conter um caminho completo desde a raiz de um disco. Se for encontrado pelo menos um ficheiro que satisfaça a especificação em **pattern**, é preenchida a estrutura apontada por **file_data** e é retornado um handle para um objecto de pesquisa.

O serviço retorna `INVALID_HANDLE_VALUE` se não for encontrado nenhum ficheiro que satisfaça **pattern**.

Se o parâmetro **pattern** do serviço anterior especificar um padrão com *wildcards* e estivermos interessados em saber se há mais ficheiros ou sub-directórios que o satisfaçam, podemos continuar a pesquisa, usando agora o serviço:

```
BOOL FindNextFile(HANDLE hfind, WIN32_FIND_DATA *find_data);
```

Em **hfind** passa-se um handle para um objecto de pesquisa obtido com `FindFirstFile()`. Se for encontrado mais algum ficheiro que satisfaça o padrão aí especifidado a estrutura apontada por **find_data** é preenchida com os novos dados, e o serviço retorna um valor diferente de 0 (`TRUE`).

Se não se encontrar mais nenhum ficheiro que satisfaça o padrão indicado em `FindFirstFile()` o serviço retorna 0 (`FALSE`).

É possível chamar repetidamente `FindNextFile()` para obter todos os ficheiros ou sub-directó-

rios que satisfaçam o padrão especificado, até o serviço retornar `FALSE`.

A estrutura `WIN32_FIND_DATA`, que é preenchida sempre que é encontrado um ficheiro, tem a seguinte definição:

```
typedef struct _WIN32_FIND_DATA {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[MAX_PATH];
  TCHAR    cAlternateFileName[14];
} WIN32_FIND_DATA;
```

Existem campos que contêm os atributos do ficheiro (ou sub-directório), os tempos relacionados e o tamanho, de forma semelhante à estrutura `BY_HANDLE_FILE_INFORMATION`, descrita atrás. Além disso o campo `cFileName` contém o nome do ficheiro ou directório e o campo `cAlternateFileName` contém um nome alternativo no formato 8.3 (8 caracteres de nome + 3 de extensão (DOS)).

Após uma pesquisa, e sempre que o objecto de pesquisa, cujo handle foi retornado em `FindFirstFile()`, já não for necessário, é indispensável fechá-lo. O fecho deste handle usa um serviço especial diferente de `CloseHandle()`. É uma das raras ocasiões em que `CloseHandle()` não pode ser utilizado. O fecho do handle de pesquisa faz-se então com:

`BOOL FindClose(HANDLE hfind);`

Fecha o handle de pesquisa passado em hfind.

Retorna 0 em caso de erro.

Mostra-se de seguida o código uma pequena rotina que lista o conteúdo de um directório cujo caminho se passa como parâmetro.

```
void list_directory(char *path)
{
  char current[MAX_PATH+1];
  HANDLE hfind;
  WIN32_FIND_DATA info;

  GetCurrentDirectory(MAX_PATH+1, current);
  SetCurrentDirectory(path);
  if ((hfind = FindFirstFile("*", &info)) == INVALID_HANDLE_VALUE) {
    SetCurrentDirectory(current);
    return;
  }
  do {
    printf("%s ", info.cFileName);
    if (info.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
      printf("<DIR>\n");
    else
      printf("\n");
  } while (FindNextFile(hfind, &info));
  FindClose(hfind);
  SetCurrentDirectory(current);
  return;
}
```

5. Criação e terminação de processos

Como se sabe o sistema operativo Windows reconhece e executa directamente *threads*. No entanto os *threads* estão agrupados em processos e o conceito de processo também existe em Win32. O processo é a unidade de alocação de recursos (memória, ficheiros, objectos de sincronização, etc.). O *thread* é apenas a unidade de execução (um fluxo de instruções).
Quando em Win32 se cria um processo, este automaticamente cria também um *thread* (o *thread* principal), que fica pronto a executar (iniciando-se na função `main()` de um programa em C).

5.1. Criação de um processo

Comecemos por ocupar-nos de momento apenas com processos com um único *thread*.

O serviço Win32 que cria novos processos é a função `CreateProcess()` que também cria automaticamente o *thread* principal (a criação de outros *threads* do mesmo processo necessita de outros serviços). Ao contrário do UNIX não há primeiro a duplicação do código do processo pai (com `fork()`) e depois possivelmente a substituição do código duplicado (com `exec()`). O serviço `CreateProcess()` lança sempre um novo processo a partir de código contido num ficheiro executável. A relação pai-filho em Windows não é muito forte; o pai conhece o identificador e handle do filho, mas o contrário é mais difícil.

O serviço `CreateProcess()` contém dez argumentos:

```
#include <windows.h>                /* como sempre */

BOOL CreateProcess(char *ImageName,
                   char *CommandLine,
                   SECURITY_ATTRIBUTES *saProcess,
                   SECURITY_ATTRIBUTES *saThread,
                   BOOL fInheritHandles,
                   DWORD fdwCreate,
                   void *Environment,
                   char *CurDir,
                   STARTUPINFO *StartupInfo,
                   PROCESS_INFORMATION *ProcInfo);
```

Retorna `FALSE` (0) se não for possível criar o novo processo, e um valor diferente de 0 se houve sucesso.

5.1.1. Parâmetros de `CreateProcess()`

Através dos 2 primeiros argumentos especifica-se o nome do ficheiro executável que contém o código do novo processo e seus argumentos.

`ImageName` é uma string que poderá conter o nome do ficheiro executável, incluindo ou não o *path* completo (caso não inclua assume-se o directório corrente). Este parâmetro poderá também ser `NULL`; neste caso o nome do executável obtém-se do parâmetro seguinte.

`CommandLine` é também uma string que especifica a linha de comando do novo processo (ou seja o nome do executável e seus parâmetros). Se o parâmetro anterior for `NULL` é aqui que se vem buscar o nome do executável usado para o novo processo (é o nome que se obtém até ao primeiro espaço). Este argumento poderá ser `NULL` (mas não simultaneamente com o anterior).

`saProcess` e **`saThread`** são apontadores para estruturas `SECURITY_ATTRIBUTES`. Estes

parâmetros servem para especificar as permissões de acesso dos novos processo e *thread*. Quando se passa o valor `NULL` para estes parâmetros são especificadas as permissões por defeito (ou seja *read*, *write*, *execute* e *delete*, para o *owner* do processo que chama este `CreateProcess()`). Além das permissões, estes parâmetros também servem para especificar se os handles destes novos objectos (processo e *thread* principal) são ou não herdáveis. Ver a secção **herança**, mais à frente. A definição da estrutura `SECURITY_ATTRIBUTES` é a seguinte:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;       /* tamanho desta estrutura (em bytes) */
    void *lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Quando se pretende que os handles que vão ser criados sejam herdáveis, é necessário passar um apontador para uma estrutura deste tipo previamente preenchida com o campo `bInheritHandle` com o valor `TRUE`; o campo `lpSecurityDescriptor` pode ser `NULL` para continuar a indicar as permissões por defeito. Por exemplo para declarar e inicializar uma estrutura deste tipo podemos usar:

```
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
```

`fInheritHandles` é o próximo argumento de `CreateProcess()`. Especifica se o novo processo pode ou não herdar handles do pai (para herdá-los efectivamente é ainda necessário que esses handles tenham sido criados com o atributo *herdável* que é especificado no campo `bInheritHandle` da estrutura `SECURITY_ATTRIBUTES` do serviço que cria o handle).

`fdwCreate` permite especificar uma série de flags que controlam algumas propriedades do novo processo, nomeadamente quanto à consola onde vai executar. Algumas dessa flags são:

- `CREATE_NEW_CONSOLE` - o novo processo terá a sua consola própria numa nova janela;
- `DETACHED_PROCESS` - o novo processo inicia-se sem estar ligado inicialmente a qualquer consola (pode sempre criar uma com o serviço `AllocConsole()`);

se nenhuma das flags anterior for especificada o novo processo usará a consola do pai.

- `CREATE_SUSPENDED` - o novo processo ficará com o seu *thread* principal suspenso (sem executar) até que alguém execute o serviço `ResumeThread()`, referindo o *thread* suspenso;
- `CREATE_NEW_PROCESS_GROUP` - o novo processo inicia um novo grupo de processos;
- Também é possível indicar neste parâmetro a prioridade do novo processo usando um dos seguintes valores (ordem decrescente de prioridade): `REALTIME_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS` (valor por defeito) e `IDLE_PRIORITY_CLASS`.

`Environment` e **`Curdir`** devem apontar respectivamente para um bloco de definição de variáveis de ambiente e para o nome do directório que passará a ser o corrente para o novo processo. As variáveis de ambiente são definidas da forma indicada na Capítulo 1. Quando estes parâmetros não forem especificados (ou seja se usarmos o valor `NULL`) esses valores são os mesmos do pai.

`StartupInfo` aponta sempre para uma estrutura do tipo `STARTUPINFO` que especifica o aspecto e outras propriedades da janela e consola do processo. Através deste parâmetro é ainda possível redireccionar os três componentes da consola para outros handles de ficheiro (que têm de ser herdáveis, sendo também `fInheritHandles` de `CreateProcess()` obrigatoriamente `TRUE` para funcionar). Veja-se a secção **StartupInfo** mais à frente.

`ProcInfo` é o último parâmetro e deve ser um apontador para uma estrutura do tipo `PROCESS_INFORMATION`. Este apontador não pode ser `NULL`. Quando o serviço retorna, esta

estrutura vem preenchida com os identificadores e handles do novo processo e *thread*. A definição da estrutura `PROCESS_INFORMATION` é a seguinte:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess, hThread;
    DWORD dwProcessId, dwThreadId;
} PROCESS_INFORMATION;
```

5.1.2. Identificação dos processos

Os processos e *threads* são acedidos Win32 através de um identificador (único no sistema) e através de um handle (nr. que identifica a entrada que desvreve o processo ou outro objecto (p. ex. ficheiro aberto) na tabela de objectos do sistema). Certos serviços requerem um identificador, enquanto que outros requerem um handle. Estes handles podem (e devem) ser fechados (com `CloseHandle()`) no processo pai (quando já não se necessitar deles), sem que isso afecte o processo. A única coisa que é destruída quando se fecha um handle de processo ou *thread* é o acesso do pai ao filho.

Um processo pode obter o seu próprio identificador e handle através dos serviços:

```
DWORD GetCurrentProcessId();

HANDLE GetCurrentProcess();
```

O handle obtido a partir de `GetCurrentProcess()` não é herdável e só pode ser usado no próprio processo. Um handle de processo herdável pode sempre ser criado a partir do seu identificador com:

```
HANDLE OpenProcess(DWORD access, BOOL inherit, DWORD IDProcess);
```

`access` especifica as propriedades pretendidas para o handle; pode ser construído pelo *or* (`|`) entre os seguintes valores (entre outros): `SYNCHRONIZE` - o handle pode ser usado nos serviços `Wait...()`; `PROCESS_TERMINATE` - o handle pode ser usado nos serviços que terminam processos; `PROCESS_QUERY_INFORMATION` - o handle pode ser usado nos serviços que obtêm informação acerca do processo; `PROCESS_ALL_ACCESS` - combina todas as flags - o handle pode ser usado em todos os serviços.
`inherit` especifica se o handle é ou não herdável.
`IDProcess` é o identificador do processo para o qual se pretende obter o handle.

Retorna `NULL` no caso de não ser possível obter o handle.

É de notar que o handle retornado pelo serviço `CreateProcess()` tem a propriedade `PROCESS_ALL_ACCESS`.

5.2. Herança de handles

Todos os objectos do kernel dos sistemas Windows (ficheiros, processos, *threads*, *pipes*, áreas de memória partilhada, semáforos, etc) são representados por handles que de alguma forma referenciam o objecto na tabela geral de objectos mantida pelo kernel. Todos estes objectos são criados por serviços (geralmente designados por `Create...()`) que retornam ou fornecem a quem os chama o respectivo handle.

É geralmente possível especificar para estes objectos uma série de permissões de acesso quando da sua criação (leitura, escrita, etc, e também quem tem essas permissões) e também

se o handle do objecto é ou não herdável.

O valor de um handle herdável pode ser passado a um processo descendente, usando qualquer mecanismo de comunicação, e ser usado por este como se tivesse sido por ele criado. Para que isto seja possível é necessário que:

- o objecto tenha sido criado de modo a fornecer um handle herdável ou se obtenha um duplicado herdável do objecto; e
- o processo ao qual se vai passar o handle possa herdar handles.

A segunda condição satisfaz-se se em `CreateProcess()` se indicar o parâmetro `fInheritHandles` com o valor `TRUE`.

Para satisfazer a primeira condição deverá preencher-se e passar ao serviço `Create...()` uma estrutura `SECURITY_ATTRIBUTES` onde o campo `bInheritHandle` é `TRUE`. Alternativamente, se já dispusermos de um handle não-herdável de um objecto, é possível criar um duplicado desse handle que seja herdável, utilizando o serviço seguinte:

```
BOOL DuplicateHandle (HANDLE hSourceProcess,
                      HANDLE hSource,
                      HANDLE hTargetProcess,
                      HANDLE *hTarget,
                      DWORD fdwAccess,
                      BOOL fInherit,
                      DWORD fdwOptions);
```

É necessário fornecer o handle do processo que pede o serviço (**`hSourceProcess`**), o handle a duplicar (**`hSource`**) e o handle do processo onde se pretende que o duplicado seja válido (**`hTargetProcess`**). Este processo pode ser o mesmo que pede o serviço. O handle duplicado é retornado através do apontador **`hTarget`**. Os três últimos argumentos especificam as propriedades do novo handle.
`fdwAccess` especifica as propriedades de acesso do novo handle; essas propriedades podem ser numerosas e dependem do tipo de handle que se está a duplicar (ver o *help* para este serviço); no caso de processos essas propriedades são as já indicadas para o serviço `OpenProcess()`, descrito acima; este parâmetro é ignorado se **`fdwOptions`** contiver o valor `DUPLICATE_SAME_ACCESS`.
`fInherit` especifica se o novo handle é (`TRUE`) ou não (`FALSE`) herdável por processos criados a partir do indicado em **`hTargetProcess`**.
Finalmente **`fdwOptions`** pode conter os valores `DUPLICATE_CLOSE_SOURCE`, que indica que o handle original é fechado, e `DUPLICATE_SAME_ACCESS`, que indica que o novo handle tem as mesmas propriedades do original.

5.3. StartupInfo

A estrutura `STARTUPINFO` é utilizada no serviço `CreateProcess()` para especificar as propriedades da janela principal do novo processo, quando e se este a criar (um novo processo terá uma nova consola se em `CreateProcess()` se passar a flag `CREATE_NEW_CONSOLE` ao parâmetro `fdwCreate`).

Esta estrutura poderá ser preenchida com os valores relativos ao processo corrente através do serviço:

```
void GetStartupInfo(STARTUPINFO *StartupInfo);
```

`StartupInfo` aponta para uma estrutura `STARTUPINFO` que é preenchida.

A definição da estrutura `STARTUPINFO` é então a seguinte:

```
typedef struct _STARTUPINFO {
    DWORD   cb;
    char *  lpReserved;
    char *  lpDesktop;
    char *  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    BYTE *  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

O significado de alguns dos campos desta estrutura é :

- **`cb`** - tamanho em bytes da estrutura (`sizeof(STARTUPINFO)`);
- **`lpTitle`** - título da janela da consola;
- **`dwX`** e **`dwY`** - posição, em pixels, do canto superior esquerdo da janela;
- **`dwXSize`** e **`dwYSize`** - tamanho, em pixels, da janela (para janelas gráficas);
- **`dwXCountChars`** e **`dwYCountChars`** - tamanho, em caracteres, da janela (consola);
- **`dwFillAttribute`** - cor do texto e fundo: pode ser uma combinação (`|`) das seguintes oito flags - `FOREGROUND_BLUE`, `FOREGROUND_GREEN`, `FOREGROUND_RED`, `FOREGROUND_-INTENSITY`, `BACKGROUND_BLUE`, `BACKGROUND_GREEN`, `BACKGROUND_RED`, `BACKGROUND_INTENSITY`;
- **`hStdInput`**, **`hStdOutput`**, **`hStdError`** - handles de redireccionamento das três componentes da consola;
- **`dwFlags`** - especifica se algum ou alguns dos outros campos são ou não ignorados; esses campos não serão ignorados se este parâmetro contiver os seguintes valores:
  - `STARTF_USEPOSITION`: não ignora `dwX` e `dwY`;
  - `STARTF_USESIZE`: não igora `dwXSize` e `dwYSize`;
  - `STARTF_USECOUNTCHARS`: não ignora `dwXCountChars` e `dwYCountChars`;
  - `STARTF_USEFILLATTRIBUTE`: não ignora `dwFillAttribute`;
  - `STARTF_USESTDHANDLES`: não ignora `hStdInput`, `hStdOutput` e `hStdError`.

Como exemplo apresenta-se o código para criar um novo processo com a entrada e saída standard redireccionadas para ficheiros (com handles `hFile1` e `hFile2`):

```
...
STARTUPINFO si;
...
GetStartupInfo(&si);
si.hStdInput = hFile1;                          // redirecciona entrada
si.hStdOutput = hFile2;                         // redirecciona saída
si.hStdError = GetStdHandle(STD_ERROR_HANDLE);  // mantém erro
si.dwFlags = STARTF_USESTDHANDLES;
...
CreateProcess(…, TRUE, …, &si, …);      // herda handles e STARTUPINFO
...
```

Não é necessário criar uma nova janela de consola para redireccionar os handles das suas

componentes num novo processo.

5.4. Terminação de um processo e código de terminação

Um processo pode terminar-se a si próprio de forma correcta através do serviço:

```
void ExitProcess(UINT ExitCode);

ExitCode - código de terminação do processo.
```

que é semelhante à função da biblioteca standard `exit()` (ver Capítulo 1).
Esta função não retorna. O processo e todos os seus *threads* são forçados a terminar. Na chamada ao serviço é especificado o código de terminação do processo.

O código de terminação pode ser consultado por outro processo invocando:

```
BOOL GetExitCodeProcess(HANDLE hProcess, DWORD *ExitCode);
```

onde **hProcess** é o handle do processo cujo código de terminação se pretende; **ExitCode** deve apontar para uma `DWORD` que irá conter esse código.

Retorna `FALSE` em caso de erro.

O handle **hProcess** necessita da propriedade `PROCESS_QUERY_INFORMATION` para este serviço ser bem sucedido. Um valor possível que o serviço pode colocar em **ExitCode** é `STILL_ACTIVE`, isto se o processo ainda não tiver terminado.

Quando um processo termina é retirado da memória, mas a informação relativa ao processo permanece na tabela de objectos até todos os handles referentes ao processo serem fechados (com `CloseHandle()`).

Um processo pode terminar outro (e todos os seus *threads*) se possuir um handle deste com a propriedade `PROCESS_TERMINATE`. O serviço a usar é então:

```
BOOL TerminateProcess(HANDLE hProcess, UINT ExitCode);
```

hProcess é o handle do processo a terminar, e **ExitCode** é o código de terminação que aquele retornará.

Este serviço deve ser usado apenas como último recurso (o normal é o processo terminar por si próprio). Se o processo assim terminado usar bibliotecas dinâmicas (DLLs) estas não serão notificadas da sua terminação.

5.5. Sincronização simples entre processos

O método mais simples (mas também limitado) de sincronizar dois processos é um esperar que o outro termine. Existem dois serviços em Win32 capazes de efectuar essa espera. É possível esperar por um único processo, ou o primeiro de um conjunto, ou ainda por todos os processos de um conjunto especificado. Estes serviços suportam ainda a indicação de um período de *time-out*. Estes serviços são muito genéricos podendo ser utilizados para efectuar esperas (que bloqueiam os *threads* que os chamam) por muitas situações, como veremos mais à frente.

Os serviços são, então:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeOut);
```

onde **hObject** é o handle do processo (ou outro objecto) a esperar e **dwTimeOut** indica o tempo máximo de espera expresso em ms (para não haver *timeout* indicar o valor INFINITE);
retorna WAIT_OBJECT_0 (normal), quando o processo indicado em **hObject** terminar, ou WAIT_TIMEOUT, quando o período de *timeout* decorrer (se for diferente de INFINITE) sem que o processo termine.

```
DWORD WaitForMultipleObjects(DWORD nObjects, HANDLE Objects[],
                             BOOL fWaitAll, DWORD dwTimeOut);
```

Objects é um array de handles para os processos a esperar e **nObjects** o seu número de elementos; se **fWaitAll** for TRUE espera-se por todos os processos indicados; caso contrário espera-se apenas pelo primeiro que terminar; **dwTimeOut** tem o mesmo significado do serviço anterior.

Retorna WAIT_OBJECT_0 + n, com 0 ≤ n < nObjects, indicando qual o processo que terminou (subtrair WAIT_OBJECT_0), ou simplesmente WAIT_OBJECT_0 se **fWaitAll** tiver sido TRUE; WAIT_TIMEOUT pode também ser um valor de retorno, com o mesmo significado do serviço anterior.

Como se disse, estes dois serviços são serviços genéricos de espera por certos estados dos objectos do kernel (neste caso estamos a usá-los para esperar pelo estado de terminação de processos). Mais à frente iremos ver a sua utilização com outros objectos que não processos.

Quando um processo termina (via exit(), fim da função main() de um programa em C, ExitProcess() ou TerminateProcess()) as seguintes acções são sempre executadas:

- Todos os handles de objectos abertos pelo processo são fechados;
- Todos os *threads* do processo terminam;
- O estado do processo passa a terminado, satisfazendo assim os serviços Wait...() que esperavam no handle deste processo;
- O código de terminação passa do valor STILL_ACTIVE para o valor especificado no método de terminação utilizado.

A terminação do processo não causa a terminação dos seus descendentes, nem remove a informação acerca do processo, mantida pelo kernel, até que todos os handles que o referem tenham sido fechados.

5.6. Exemplo

Pretende-se escrever um programa "grepMP" que procura a ocorrência de uma string *pattern* numa lista de ficheiros de texto. O programa deve ser invocado da forma:

```
grepMP pattern F1 F2 ... Fn
```

onde F1 até Fn são os nomes de N ficheiros de texto onde se deve procurar a string *pattern*.

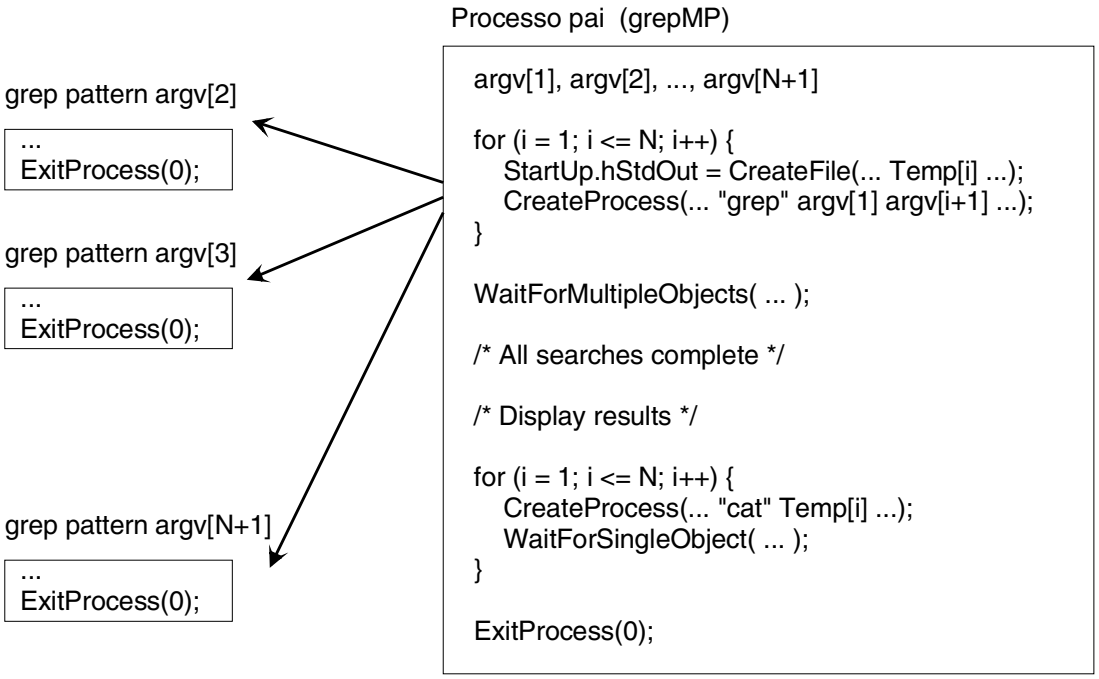
Admite-se a existência de um programa "grep" que procura uma string *pattern* num ficheiro, escrevendo na saída standard (geralmente a consola) todas as linhas do ficheiro que contêm a string *pattern*; a invocação deste programa deve fazer-se da forma:

```
grep pattern file_name
```

O programa "grepMP" deverá criar N processos "grep" a executar "simultaneamente", passar-

-lhes a string `pattern` e o nome de um ficheiro, redireccionar-lhes a saída standard para um ficheiro temporário, e esperar que todos terminem; quando todos terminarem deve escrever, por ordem, os resultados de todos.

Um esquema genérico de "`grepMP`" poderá ser então:



Presume-se ainda a existência do programa "`cat`" que lista na saída standard o conteúdo de um ficheiro de texto cujo nome lhe é passado como parâmetro.

O primeiro ciclo de criação dos N processos "`grep`" será então:

```
#include <windows.h>

typedef char TF[MAX_PATH];

int main(int argc, char *argv[])
{
  int k;
  HANDLE htempfile;
  SECURITY_ATTRIBUTES StdOut = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
  char CommandLine[MAX_PATH + 100];
  STARTUPINFO StartUp;
  PROCESS_INFORMATION ProcInfo;
  HANDLE  *Procs;                              /* para array de handles */
  TF *TempFile;                                  /* para array de nomes */

  TempFile = (TF *) malloc((argc-2)*sizeof(TF));
                                             /*array de nomes temporários*/
  Procs = (HANDLE *) malloc((argc-2)*sizeof(HANDLE));
                                              /*array de handles de procs*/
  GetStartupInfo(&StartUp);    /* preenche StartUp com valores actuais */

  for (k=0; k<argc-2; k++) {
    sprintf(CommandLine, "grep %s %s", argv[1], argv[k+2]);
                                                    /* constrói chamada */
    GetTempFileName(".", "grp", 0, TempFile[k]);
                                        /* Obtém nome temporário único */
```

```
      /* O handle htempfile é herdável - ver StdOut */

      htempfile = CreateFile(TempFile[k], GENERIC_READ | GENERIC_WRITE,
                             FILE_SHARE_READ | FILE_SHARE_WRITE, &StdOut,
                             CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      StartUp.dwFlags = STARTF_USESTDHANDLES;
      StartUp.hStdOutput = htempfile;    /* redirecciona Standard Output */
      StartUp.hStdError = GetStdHandle(STD_ERROR_HANDLE);

               /* Cria o processo para executar "grep pattern ficheiro" com
                             saída standard redireccionada  (em StartUp) */

      CreateProcess(NULL, CommandLine, NULL, NULL, TRUE, 0, NULL, NULL,
                    &StartUp, &ProcInfo);
      CloseHandle(ProcInfo.hThread);
            /* todos os handles devem ser fechados quando não necessários*/
      CloseHandle(htempfile);
      Procs[k] = ProcInfo.hProcess;
                              /* constrói array de handles dos processos */
    }

         /* Espera que todos os processos lançados terminem */

    WaitForMultipleObjects(argc-2, Procs, TRUE, INFINITE);
    for (k=0; k<argc-2; k++)
      CloseHandle(Procs[k]);       /* fecha todos os handles de processo */
    ..........
    free(TempFile);
    free(Procs);
    return 0;
  }
```

Os ficheiros temporários devem ser apagados explicitamente antes de "`grepMP`" terminar (usar `DeleteFile(nome_do_ficheiro)`).

6. Serviços de *threads*

Como se sabe o sistema operativo Windows reconhece e executa directamente *threads*. Quando em Win32 se cria um processo este automaticamente cria também um *thread* (o *thread* principal) que fica pronto a executar (iniciando-se na função `main()` de um programa em C). O serviço `CreateProcess()` retorna através do seu último argumento o *handle* e o identificador do *thread* principal.

6.1. Criação e terminação de *threads*

A criação de novos *threads* dentro de um processo tem de ser feita com a invocação de serviços apropriados do Sistema Operativo.

Quando se pretende criar um programa *multithread* o código destes tem de fazer parte integrante do programa. Cada *thread* inicia-se numa função do programa, cujo endereço é passado ao serviço de criação de um novo *thread*, com um cabeçalho bem definido. A partir daí a sua execução fica independente (ou melhor, concorrente) dos outros *threads* do processo.

Assim para criar um novo *thread* num processo deverá invocar-se o serviço `CreateThread()`:

```
HANDLE CreateThread(SECURITY_ATTRIBUTES *sa,
                    DWORD StackSize,
                    THREAD_START_ROUTINE *StartAddr,
                    void *ThreadPointer,
                    DWORD fdwCreate,
                    DWORD *IDThread);
```

onde **`sa`** é um apontador para uma estrutura de atributos de segurança e herança do novo *thread* (ver capítulo anterior); passando o valor `NULL` são utilizados atributos por defeito;
`StackSize` indica em bytes o tamanho da stack exclusiva do novo *thread*; indicando aqui o valor 0 é criada uma stack por defeito;
em **`StartAddr`** deverá indicar-se apenas o nome (que significa o endereço) da função que é o início do novo *thread*; essa função deverá ter sido declarada com o seguinte cabeçalho:

```
DWORD WINAPI nome_da_função(void *arg)
```

é de notar que esta função recebe como parâmetro um apontador genérico que poderá conter o endereço de quaisquer outros dados;
este argumento da função (`void *arg`) que inicia o *thread* é passado no serviço através do seu 4º argumento ou seja **`ThreadPointer`**;
o argumento **`fdwCreate`** normalmente tem um de dois valores: 0 - o *thread* criado fica pronto a executar, ou `CREATE_SUSPENDED`, em que o *thread* fica num estado de suspenção, necessitando de um outro serviço para ficar pronto a executar;
finalmente **`IDThread`** deverá ser um apontador para uma `DWORD` onde o serviço irá colocar o identificador do *thread* criado.

O serviço retorna o handle do novo *thread* no caso de haver sucesso, ou o valor `INVALID_HANDLE_VALUE` no caso de erro.

O próprio *thread* pode obter o seu identificador e handle, à semelhança do que acontece com os processos, através dos serviços:

```
DWORD GetCurrentThreadId(void);

HANDLE GetCurrentThread(void);
```

O handle retornado por `GetCurrentThread()` é um handle que apenas pode ser usado no

próprio *thread*, não sendo por isso herdável. Para construir um handle herdável deve-se usar o serviço `DuplicateHandle()` (capítulo anterior) ou o seguinte:

```
HANDLE OpenThread(DWORD fDesiredAccess, BOOL fInheritHandle,
                  DWORD ThreadId);
```

onde através de **`fDesiredAccess`** é possível especificar as propriedades do handle retornado usando, entre outros, os valores `SYNCHRONIZE`, `THREAD_QUERY_INFORMATION`, `THREAD_SUSPEND_RESUME`, `THREAD_TERMINATE` e `THREAD_ALL_ACCESS`; este último funciona como o *or* (|) de todos os outros;
`fInheritHandle` indica se o handle retornado é ou não herdável e **`ThreadId`** é o identificador do *thread*.

Em caso de erro o serviço retorna `NULL`.

O handle retornado por `CreateThread()` tem sempre como propriedade `THREAD_ALL_ACCESS`.

Um *thread* termina quando a função indicada em `CreateThread()` retornar. Repare-se que essa função retorna uma `DWORD`; o valor retornado passa a ser o código de terminação do *thread*.

Outra forma de terminar correctamente um *thread* é através da invocação, por parte do próprio, do serviço:

```
void ExitThread(DWORD ExitCode);
```

onde o parâmetro **`ExitCode`** indica o respectivo código de terminação.

Um *thread* pode terminar-se a si ou a outro *thread* através do serviço descrito a seguir. No entanto a execução desse serviço só deve ser usada como último recurso uma vez que causa uma terminação imediata, sem se proceder à libertação da memória da stack própria do *thread*, nem à notificação de terminação de possíveis DLLs usadas pelo *thread*. Normalmente cada *thread* deve terminar por si próprio, quando a função que foi o seu início retorna, ou através do serviço `ExitThread()`.

```
void TerminateThread(HANDLE hThread, DWORD ExitCode);
```

O parâmetro **`hThread`** é o *handle* do *thread* a terminar e **`ExitCode`** o respectivo código de terminação. É necessário que o handle usado em **`hThread`** tenha a propriedade `THREAD_TERMINATE`.

É também possível ir buscar o código de terminação de um *thread* já terminado. O serviço para isso é:

```
BOOL GetExitCodeThread(HANDLE hThread, DWORD *ExitCode);
```

O parâmetro **`hThread`** é o handle do *thread* e **`ExitCode`** é um apontador para um local onde será colocado, pelo serviço, o código com que o *thread* terminou. Se o *thread* interrogado ainda não tiver terminado esta `DWORD` é preenchida com o valor `STILL_ACTIVE`. É ainda necessário que o handle passado tenha a propriedade `THREAD_QUERY_INFORMATION`.

Como se viu atrás, um *thread* pode ser criado no estado de suspenção. Na realidade cada *thread* tem associado um valor chamado *suspend count*. Se este valor for maior ou igual a 1 o

thread estará nesse estado de suspenção, se for igual a 0 não. Quando o *thread* é criado com `CREATE_SUSPENDED` o valor de *suspend count* é 1.

Existem dois serviços que incrementam e decrementam o *suspend count*. São eles respectivamente:

```
DWORD SuspendThread(HANDLE hThread);

DWORD ResumeThread(HANDLE hThread);
```

O parâmetro **`hThread`** terá de possuir a propriedade `THREAD_SUSPEND_RESUME`.

Retornam o valor anterior do *suspend count* (ou `0xFFFFFFFF` no caso de erro).

Os serviços `WaitForSingleObject()` e `WaitForMultipleObjects()` já descritos no capítulo anterior podem também ser utilizados com os *handles* de *threads*, esperando que eles terminem.

Mostra-se de seguida um pequeno exemplo da criação e terminação de um *thread* secundário a partir do principal.

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

DWORD WINAPI thread_func(void *arg);

int main(void)
{
    DWORD code, id;
    HANDLE hThr;

    printf("Thread principal\n");
    hThr = CreateThread(NULL, 0, thread_func, NULL, 0, &id);

    WaitForSingleObject(hThr, INFINITE);
    GetExitThreadCode(hThr, &code);
    CloseHandle(hThr);
    printf("Thread secundário terminou com código %d\n", code);
    return 0;
}

DWORD WINAPI thread_func(void *arg)
{
    printf("Thread secundário\n");
    return 20;
}
```

Todas as variáveis globais do programa ao qual os *threads* pertencem podem naturalmente ser acedidas por estes, sendo por isso comuns a todos os *threads*. No entanto esse acesso terá de ser feito com cuidado (recorde a noção de secções críticas e o uso de semáforos).

6.2. Dados locais a um *thread*

Se um *thread* for constituído por várias funções em C, poderá haver interesse em definir dados acessíveis apenas por essas funções. A API Win32 designa esses dados, comuns apenas às várias funções de um *thread*, por *thread local storage* ou TLS.

O TLS é constituído por uma matriz de apontadores genéricos (`void *`) em que cada *thread*

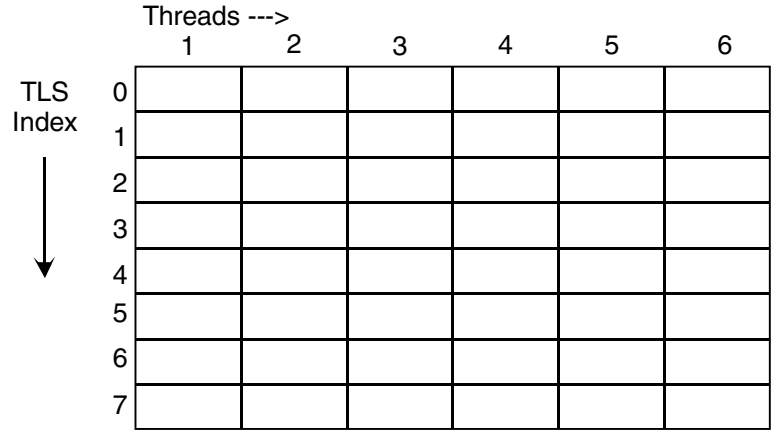
tem acesso apenas a uma coluna. Inicialmente essa matriz não existe (não contém nenhuma linha). No entanto é possível acrescentar (e libertar) novas linhas a essa matriz através dos serviços:

```
DWORD TlsAlloc(void);

BOOL TlsFree(DWORD dwIndex);
```

`TlsAlloc()` retorna um índice da linha criada na matriz TLS (>= 0), ou -1 (`0xFFFFFFFF`) se não for possível criar mais linhas. Qualquer índice retornado por `TlsAlloc()` (uma linha da matriz TLS) pode ser passado a `TlsFree()` para libertar a respectiva linha.

Qualquer *thread* de um processo pode criar novas linhas na matriz TLS, no entanto é de toda a conveniência que seja apenas um *thread* a tratar da gestão da TLS.



Matriz TLS

Qualquer *thread* pode aceder e modificar os valores armazenados na respectiva coluna (que são sempre apontadores genéricos (`void *`)) através de dois serviços, existentes para esse efeito. No entanto, antes de utilizar os serviços, o programador deve assegurar-se que o índice TLS (linha da matriz) é válido, isto é, foi alocado com `TlsAlloc()` e não foi libertado com `TlsFree()`.

Assim, quando um *thread* necessitar de armazenamento exclusivo, a ser usado apenas nas suas funções, deverá criar esse espaço (com `malloc()`) e colocar o respectivo apontador na TLS (com `TlsSetValue()`). Qualquer das funções do *thread* poderá posteriormente aceder a esse apontador através de `TlsGetValue()` e usar a memória para onde aponta.

```
void *TlsGetValue(DWORD Index);

BOOL TlsSetValue(DWORD Index, void *TlsValue);
```

`TlsGetValue()` retorna o valor armazenado na entrada **`Index`** da coluna da matriz TLS pertencente ao *thread* que invoca o serviço, enquanto que `TlsSetValue()` transfere o valor **`TlsValue`** para essa entrada.

`TlsGetValue()` retorna `NULL` no caso de erro, enquanto que `TlsSetValue()` retorna `FALSE`.

Mostra-se de seguida um pequeno exemplo de demonstração da TLS, onde se cria uma linha da matriz TLS e se preenchem e acedem algumas das suas entradas (uma por cada *thread* existente).

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

#define THREADCOUNT 4
DWORD dwTlsIndex;

void CommonFunc(void)
{
   void *lpvData;

   lpvData = TlsGetValue(dwTlsIndex);
   printf("common: thread %u: lpvData=%lx\n", GetCurrentThreadId(),
                                              lpvData);
   Sleep(5000);   /* suspende o thread por 5000 ms */
}

DWORD WINAPI ThreadFunc(void *arg)
{
   void *lpvData;

   lpvData = malloc(256);
   TlsSetValue(dwTlsIndex, lpvData);
   printf("thread %u: lpvData=%lx\n", GetCurrentThreadId(), lpvData);

   CommonFunc();

   lpvData = TlsGetValue(dwTlsIndex);
   free(lpvData);

   return 0;
}

int main(void)
{
   DWORD IDThread;
   HANDLE hThread[THREADCOUNT];
   int i;

   dwTlsIndex = TlsAlloc();
   for (i = 0; i < THREADCOUNT; i++)
      hThread[i] = CreateThread(NULL, 0, ThreadFunc, NULL, 0,
                                &IDThread);
   WaitForMultipleObjects(THREADCOUNT, hThread, TRUE, INFINITE);
   return 0;
}
```

6.3. A biblioteca standard do C e a programação *multithreaded*

A biblioteca standard do C foi especificada e implementada muito antes da programação com vários *threads* se ter começado a generalizar. Algumas das suas funções utilizam variáveis globais (ou estáticas - retendo um valor de uma chamada para outra), o que torna o seu código não reentrante e, por isso, impróprio para ser usado num ambiente *multithreaded*. Diz-se que essas funções não são *thread-safe*. Estariam nesse caso, por exemplo, as funções que modificam a variável global `errno`.

Recentemente os fornecedores de compiladores e de software de sistema re-implementaram a biblioteca standard do C de modo a torná-la *thread-safe*. No entanto, a utilização dessas novas bibliotecas pode ter requisitos especiais.

O sistema de desenvolvimento Visual C++ compreende 3 versões diferentes da biblioteca

standard do C. Duas delas são estáticas (o seu código é fisicamente ligado ao programa executável) e uma é dinâmica (código partilhado por todos os programas que o utilizam).

Uma das implementações estáticas (LIBC.LIB) destina-se apenas a programas com um único *thread*, não sendo por isso *thread-safe*. A outra implementação estática (LIBCMT.LIB) e a implementação dinâmica (MSVCRT.DLL) são *thread-safe*.

Quando se desenvolve um programa *multithreaded*, que utilize chamadas à biblioteca standard, é então necessário garantir que o compilador está configurado para o uso de uma biblioteca *thread-safe*. (configurar o Visual C++ usando *Project --> Settings ... --> C/C++ --> Category: Code Generation --> Use run-time library: Multithreaded ou Multithreaded DLL*). Por outro lado, é necessário inicializar estas bibliotecas (nessa inicialização cria-se uma área de memória para uso exclusivo de cada *thread*). No caso do Visual C++ essa inicialização faz-se chamando a função `_beginthreadex()` em vez de `CreateThread()`. Para libertar de imediato toda a memória afectada a cada *thread* estes devem terminar com uma chamada a `_endthreadex()` (em vez de `ExitThread()`).

```
#include <process.h>

unsigned long _beginthreadex(void *security, unsigned stack_size,
                             unsigned (__stdcall *start_address)(void *),
                             void *arglist, unsigned initflag,
                             unsigned *thrdaddr);
```

Cria um novo *thread*, usando os parâmetros de `CreateThread()` e inicializando a biblioteca standard *thread-safe*.

```
#include <process.h>

void _endthreadex(unsigned retval);
```

Termina um *thread* com código de terminação **`retval`**. Liberta toda a memória previamente alocada pela biblioteca standard.

7. Objectos de sincronização em Win32

São vários os objectos de sincronização suportados directamente pela API Win32, consoante os fins em vista. Entre os que são suportados iremos ver os seguintes: CRITICAL_SECTIONS's, mutexes, semáforos e eventos. Os objectos de sincronização servem, como se sabe, para a protecção do acesso a informação comum, quando temos vários processos ou *threads* a executar em paralelo ou concorrentemente.

7.1. Critical Sections

Os objectos de sincronização mais simples destinam-se à exclusão mútua de secções críticas definidas nos *threads* de um único processo e chamam-se naturalmente CRITICAL_-SECTION's.

Os serviços a usar com estes objectos são apenas quatro e definem-se do seguinte modo:

```
void InitializeCriticalSection(CRITICAL_SECTION *CriticalSection);

void DeleteCriticalSection(CRITICAL_SECTION *CriticalSection);

void EnterCriticalSection(CRITICAL_SECTION *CriticalSection);

void LeaveCriticalSection(CRITICAL_SECTION *CriticalSection);
```

Antes de podermos utilizar um objecto deste tipo, temos necessariamente de o criar com o serviço `InitializeCriticalSection()` passando-lhe um apontador para uma variável do tipo `CRITICAL_SECTION`, que é preenchida pelo sistema.

Depois desta criação/inicialização as secções críticas existentes nos vários *threads* de um processo podem ser protegidas envolvendo-as nas chamadas a `EnterCriticalSection()` e `LeaveCriticalSection()`. Quando um *thread* executa `EnterCriticalSection()` estando um outro *thread* no meio de uma outra secção crítica, protegida pela mesma variável `CRITICAL_SECTION`, bloqueia e fica à espera que o *thread* que está de posse da secção crítica execute o serviço `LeaveCriticalSection()`.

Quando uma variável `CRITICAL_SECTION` inicializada já não for mais necessária, deve ser libertada com uma chamada a `DeleteCriticalSection()`.

Exemplo:

Thread 1:

```
CRITICAL_SECTION cs;     /* global ao processo */
...
InitializeCriticalSection(&cs);
...
EnterCriticalSection(&cs);
...   /* secção crítica */
LeaveCriticalSection(&cs);
...
DeleteCriticalSection(&cs);
...
```

Thread 2:

```
...
EnterCriticalSection(&cs);
...    /* secção crítica */
LeaveCriticalSection(&cs);
...
```

7.2. Mutexes

Os mutexes são objectos que servem também para implementar a exclusão mútua de secções críticas, mas têm mais funcionalidades do que as CRITICAL_SECTION's.

Os mutexes podem ser utilizados em *threads* que pertencem a processos diferentes, possuem handles, e só podem ser pertença de um *thread* de cada vez (ou de nenhum). Os *threads* implementam a exclusão mútua com mutexes usando os serviços `WaitForSingleObject()` ou `WaitForMultipleObjects()`, já descritos. Uma chamada dos serviços anteriores só será bem sucedida se os mutexes especificados não pertencerem a nenhum *thread*. Após um serviço de `WaitFor...()` bem sucedido num ou mais mutexes os mutexes passam a pertencer a este *thread* até serem libertados. Quando um *thread* possui um mutex, outro que execute um dos serviços `WaitFor...()` ficará aí à espera (bloqueado) que o mutex seja libertado.

Os mutexes são criados, abertos por outro processo que não o que o criou, e libertados, através dos serviços:

```
HANDLE CreateMutex(SECURITY_ATTRIBUTES *sa, BOOL InitialOwner,
                   char * MutexName);

HANDLE OpenMutex(DWORD DesiredAccess, BOOL InheritHandle, char * MutexName);

BOOL RealeaseMutex(HANDLE hMutex);
```

O parâmetro **`sa`** especifica os atributos de segurança do mutex (`NULL` para os atributos por defeito); se quizermos que o mutex assim criado seja herdável por um processo filho o campo **`bInheritHandle`** de **`sa`** deverá ser preenchido com o valor `TRUE`. Quando **`InitialOwner`** é `TRUE` o mutex criado fica logo a pertencer ao *thread* que chamou `CreateMutex()`; todos os mutexes são criados com um nome, especificado em **`MutexName`**, para poderem ser utilizados noutros processos.
O parâmetro **`DesiredAccess`** de `OpenMutex()` especifica algumas propriedades do mutex assim aberto; geralmente usa-se o valor `MUTEX_ALL_ACCESS`. **`InheritHandle`** especifica se o handle obtido é ou não herdável e **`MutexName`** o nome de um mutex já existente e criado noutro processo com `CreateMutex()`.
`ReleaseMutex()` liberta o mutex que já seja pertencente ao *thread*.

Outro processo (ou melhor, um *thread* de outro processo) que não aquele que criou o mutex pode abri-lo com `OpenMutex()`, especificando o seu nome em **`MutexName`**.

Um *thread* que possua um mutex (já na sua posse quando foi criado (**`InitialOwner`** igual a `TRUE`), ou obtido com uma chamada a um dos serviços `WaitFor...()`) deverá libertá-lo com `RealeaseMutex()` (apenas o *thread* que possui o mutex é que pode libertá-lo).

Quando um mutex já não for necessário, todos os processos devem fechá-lo com `CloseHandle()`. Quando o último handle que se refere a um dado mutex for fechado o sistema destrói-o.

Exemplo:

Processo 1:

```
HANDLE mutex;

mutex=CreateMutex(NULL, FALSE,
                  "My_mutex");
...
WaitForSingleObject(mutex, INFINITE);
...   /* secção crítica */
RealeaseMutex(mutex);
...
CloseHandle(mutex);
...
```

Processo 2;

```
HANDLE mutex;

mutex=OpenMutex(MUTEX_ALL_ACCESS,
                FALSE, "My_mutex");
...
WaitForSingleObject(mutex, INFINITE);
...     /* secção crítica */
RealeaseMutex(mutex);
...
CloseHandle(mutex);
...
```

7.3. Semáforos genéricos

Esta 3ª categoria de objectos é muito semelhante aos mutexes, mas são capazes de manter um valor maior ou igual a 0. Quando um semáforo é criado o seu valor é inicializado com o valor indicado na chamada ao serviço. Sempre que um *thread* chama `WaitFor...()` com handles de semáforos, o serviço será bem sucedido se o valor do, ou dos semáforos, for positivo. A operação de `WaitFor...()` decrementa esse valor de uma unidade. Se o valor do semáforo for 0 quando o *thread* chama `WaitFor...()`, então este bloqueará até que o valor do semáforo se torne pelo menos 1. Os valores dos semáforos são incrementados com o serviço `RealeaseSemaphore()`.

Os serviços que lidam com semáforos são então:

```
HANDLE CreateSemaphore(SECURITY_ATTRIBUTES *sa, long SemInitial, long SemMax,
                       char * SemaphoreName);

HANDLE OpenSemaphore(DWORD DesiredAccess, BOOL InheritHandle,
                     char * SemaphoreName);

BOOL RealeaseSemaphore(HANDLE hSemaphore, long RealeaseCount,
                       long *PreviousCount);
```

Os semáforos são criados com um nome (**`SemaphoreName`**), um valor inicial (**`SemInitial`**) maior ou igual a zero e menor ou igual a um máximo (**`SemMax`**) que terá de ser maior ou igual a 1.

Outro processo poderá abrir um semáforo através do seu nome com `OpenSemaphore()`; o seu parâmetro **`DesiredAccess`** deve ser `SEMAPHORE_ALL_ACCESS`. Quando da libertação de um semáforo (corresponde à operação de signal) poderá especificar-se um valor (**`RealeaseCount`**) para incrementar o valor do semáforo; o valor que o semáforo tem antes de ser incrementado é devolvido através de um apontador para um longo (**`PreviousCount`**).

Exemplo:

Processo 1:

```
HANDLE sem;
long semval;

sem=CreateSemaphore(NULL, 1, 1,
                    "My_sem");

...
WaitForSingleObject(sem, INFINITE);
...   /* secção crítica */
RealeaseSemaphore(sem, 1, &semval);
...
CloseHandle(sem);
...
```

Processo 2;

```
HANDLE sem;
long semval;

sem=OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                  FALSE, "My_sem");

...
WaitForSingleObject(sem, INFINITE);
... /* secção crítica */
RealeaseSemaphore(sem, 1, &semval);
...
CloseHandle(sem);
...
```

Não existe nenhuma função que decremente o valor de um semáforo mais do que 1. Para isso será necessário chamar um dos serviços WaitFor...() mais do que vez, mas de forma indivisível. Isso pode ser feito como se mostra a seguir:

```
EnterCriticalSection(&cssem);
WaitForSingleObject(hSem, INFINITE);
WaitForSingleObject(hSem, INFINITE);
LeaveCriticalSection(&cssem);
...                          /* secção guardada pelo semáforo */
RealeaseSemaphore(hSem, 2, &previous);
```

7.4. Eventos

Finalmente o último tipo de objectos de sincronização em Win32 que iremos ver são os eventos. Um evento existe num de 2 estados: sinalizado ou não-sinalizado. Um *thread* que execute um dos serviços `WaitFor...()` num ou mais handles de eventos fica à espera que esse evento seja sinalizado (ou avança imediatamente se já estiver sinalizado).

São cinco os serviços relacionados com eventos:

```
HANDLE CreateEvent(SECURITY_ATTRIBUTES *sa, BOOL ManualReset, BOOL InitialState,
                   char *EventName);

HANDLE OpenEvent(DWORD DesiredAccess, BOOL InheritHandle, char * EventName);

BOOL SetEvent(HANDLE hEvent);

BOOL ResetEvent(HANDLE hEvent);

BOOL PulseEvent(HANDLE hEvent);
```

Os eventos são criados com um nome, à semelhança dos objectos anteriores; com um estado inicial (sinalizado se **`InitialState`** for `TRUE` e não-sinalizado no caso contrário) e do tipo manual ou automático consoante o valor `TRUE` ou `FALSE` de **`ManualReset`**. São abertos noutro processo com `OpenEvent()` devendo **`DesiredAccess`** ser igual a `EVENT_ALL_ACCESS`.

Um *thread* sinaliza um evento com `SetEvent()`. Se o evento for automático, um único outro *thread* que esteja à espera deste evento avançará; depois disso o evento passará imediatamente a não-sinalizado (se não houver nenhum *thread* à espera do evento, este permanecerá sinalizado até que um *thread* chame `WaitFor...()` para o evento). Se o evento for manual todos os *threads* que estão à espera do evento serão libertados e poderão avançar, permanecendo o evento sinalizado.

O serviço `ResetEvent()` passa um evento ao estado de não-sinalizado.

O serviço `PulseEvent()` deve ser chamado apenas para eventos manuais e que estejam não--sinalizados. Este serviço passa momentaneamente o evento a sinalizado, permitindo que todos os *threads* que estão à espera desse evento possam prosseguir. O evento passa a não--sinalizado depois disso.

Exemplo:
Sincronização entre um *thread* (`main`) que modifica informação comum e 4 outros que consultam essa informação, usando eventos. Quando o *thread* modificador está a modificar a informação, nenhum dos outros *threads* poderá estar a consultá-la.

```
#define NUMTHREADS 4

HANDLE hGlobalWriteEvent;     // Evento manual e global

void main(void)
{
    HANDLE hReadEvents[NUMTHREADS], hThreads[NUMTHREADS];

    hGlobalWriteEvent = CreateEvent(
        NULL,         // no security attributes
        TRUE,         // manual-reset event
        TRUE,         // initial state is signaled
        "WriteEvent"  // object name
```

```
        );

    // Create multiple threads and an automatic event object
    // for each thread. Each thread sets its event object to
    // signaled when it is not reading from the shared buffer.

    for(i = 1; i <= NUMTHREADS; i++)
    {
        hReadEvents[i] = CreateEvent(            // Create the auto-reset event.
            NULL,     // no security attributes
            FALSE,    // auto-reset event
            TRUE,     // initial state is signaled
            NULL);    // object not named

        hThreads[i] = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE) ThreadFunction,
            &hReadEvents[i],                    // pass event handle
            0, &IDThread);
    }
    WriteToBuffer();

    WaitForMultipleObjects(NUMTHREADS, hThreads, TRUE, INFINITE);
}

VOID WriteToBuffer(VOID)
{
    ResetEvent(hGlobalWriteEvent);              // Reset hGlobalWriteEvent

    WaitForMultipleObjects(                     // Wait for all reading threads.
        NUMTHREADS,   // number of handles in array
        hReadEvents,  // array of read-event handles
        TRUE,         // wait until all are signaled
        INFINITE);    // indefinite wait

    ....   // Write to buffer

    SetEvent(hGlobalWriteEvent);                // Set hGlobalWriteEvent

    for(i = 1; i <= NUMTHREADS; i++)          // Set all read events to signaled
        SetEvent(hReadEvents[i]);
}

void WINAPI ThreadFunction(void *lpParam)
{
    HANDLE hEvents[2];

    hEvents[0] = *(HANDLE*)lpParam;               // thread's read event
    hEvents[1] = hGlobalWriteEvent;

    WaitForMultipleObjects(
        2,            // number of handles in array
        hEvents,      // array of event handles
        TRUE,         // wait till all are signaled
        INFINITE);    // indefinite wait
    // Read event automatically reset

    .... // Read the buffer

    SetEvent(hEvents[0]);                   // Set the read event to signaled
}
```

8. Comunicação entre processos

8.1. Sinais

Os Sistemas Operativos Windows suportam apenas uma implementação muito reduzida dos sinais. Os sinais reconhecidos são apenas SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV e SIGTERM, conforme definidos no ficheiro de inclusão `signal.h`.

As funções relativas a sinais implementadas em Windows são apenas as obrigatórias na biblioteca standard do C, ou seja, `signal()`, `raise()` e `abort()`.

Existe um substituto do serviço `sleep()` do UNIX:

```
#include <windows.h>

void Sleep(DWORD Milliseconds);
```

Bloqueia o *thread* que o executa o número de ms especificado em **`Milliseconds`**.

A chamada `Sleep(0)` faz com que o *thread* que a executa perca de momento o processador (vai para o estado de pronto).

Em substituição dos sinais os sistemas Windows têm outros mecanismos equivalentes, que se referem a seguir:

- o chamado manuseamento estruturado de excepções para tratamento de erros gerados pelo hardware;
- handlers de controlo da consola para envio entre processos de "eventos";
- os serviços `TerminateProcess()` e `TerminateThread()` para substituir o envio do sinal `SIGKILL`.

Os processos com consola podem receber os seguintes eventos, cuja acção por defeito é uma chamada a `ExitProcess()`:

`CTRL_C_EVENT` - quando se tecla Control-C, ou enviado de outro processo;
`CTRL_BREAK_EVENT` - quando se tecla Control-Break, ou enviado de outro processo;
`CTRL_CLOSE_EVENT` - quando a janela da consola é fechada;
`CTRL_LOGOFF_EVENT` - quando o utilizador faz logoff;
`CTRL_SHUTDOWN_EVENT` - quando o sistema está prestes a terminar.

A acção por defeito pode ser alterada instalando um handler que é chamado quando o processo recebe um destes eventos. A instalação do handler faz-se com um serviço semelhante a `signal()`:

```
BOOL SetConsoleCtrlHandler(HANDLER_ROUTINE *Handler, BOOL Add);
```

onde **`Handler`** deve ser o nome da função que se pretende instalar.
O parâmetro **`Add`** indica, quando é `TRUE`, que o novo handler deve ser acrescentado à lista de handlers activos, devendo ser retirado quando é `FALSE`. Se **`Handler`** for `NULL` e **`Add`** `TRUE` isso faz com que o evento `CTRL_C_EVENT` seja sempre ignorado.

Retorna `FALSE` no caso de erro.

A função que irá servir como handler no serviço anterior deverá ser definida e declarada como:

```
BOOL WINAPI HandlerName(DWORD CtrlType);
```

O parâmetro **CtrlType** recebe do sistema uma das 5 constantes identificadoras do evento já descritas atrás. Se o handler tratar o tipo de evento que lhe é indicado deve retornar TRUE. No caso contrário deve retornar FALSE.

Um processo pode enviar a outros, em certas condições, um dos eventos CTRL_C_EVENT ou CTRL_BREAK_EVENT. Deve para isso usar um serviço próprio, que é:

```
BOOL GenerateConsoleCtrlEvent(DWORD Event, DWORD ProcGrpId);
```

Onde **Event** é umas das constantes CTRL_C_EVENT ou CTRL_BREAK_EVENT; o parâmetro **ProcGrpId** representa o identificador de um grupo de processos.

Retorna FALSE se ocorrer um erro.

Para que este serviço tenha sucesso é necessário que tenha sido definido um grupo de processos. Define-se um grupo de processos sempre que se usa a flag CREATE_NEW_PROCESS_GROUP em CreateProcess(). Todos os novos processos criados por este e que partilham a mesma consola estarão no mesmo grupo de processos. O identificador do grupo, necessário em GenerateConsoleCtrlEvent(), é o identificador (ProcessId) do processo pai (aquele que foi criado com CREATE_NEW_PROCESS_GROUP). O serviço envia o evento especificado em **Event** (apenas um dos eventos CTRL_C_EVENT ou CTRL_BREAK_EVENT) a todos os processos pertencentes ao grupo.

No serviço anterior, se o parâmetro **ProcGrpId** for 0, o evento especificado é então enviado a todos os processos que partilham a consola pertencente ao processo que invoca o serviço.

8.2. Pipes anónimos

Os pipes anónimos em Win32 são idênticos aos pipes do UNIX. O serviço de criação do pipe retorna 2 handles: um de leitura de um dos lados do pipe e o outro de escrita no outro lado do pipe. O problema principal reside em passar estes handles para o ou os processos filhos que vão utilizar o pipe. Essa passagem pode fazer-se através de herança ou através do redireccionamento prévio da entrada ou saída standard.

O serviço de criação de um pipe é então:

```
BOOL CreatePipe(HANDLE *hRead, HANDLE *hWrite, SECURITY_ATTRIBUTES *sa,
                DWORD SizePipe);
```

Os parâmetros **hRead** e **hWrite** trazem do serviço os handles dos lados receptor e emissor do pipe, cujas operações de leitura e escrita se fazem com os serviços ReadFile() e WriteFile(). O parâmetro **SizePipe** indica o tamanho do buffer associado ao pipe. O valor 0 para este parâmetro indica um tamanho por defeito. **sa** que poderá indicar os atributos de segurança, indica também se os handles criados do pipe são ou não herdáveis. Normalmente, para serem utilizados em processos descendentes terão que ser herdáveis.

Retorna FALSE no caso de ocorrer algum erro.

Exemplo

Escrita de um programa `pipe` que lança dois filhos, estabelecendo uma ligação por pipe entre a saída standard do primeiro e a entrada standard do segundo. Uma linha de comando para a chamada de `pipe` poderá ser:

```
c:> pipe prog1 args = prog2 args
```

```
#include <windows.h>
#include <string.h>

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

/* Pipe together two programs whose names are on the command line:
            pipe command1 = command2
   where the two commands are arbitrary strings.
   command1 uses standard input, and command2 uses standard output.
   Use = so as not to conflict with the DOS pipe. */
{
      DWORD i;
      HANDLE hReadPipe, hWritePipe;
      char Command1[MAX_PATH];
      SECURITY_ATTRIBUTES PipeSA =       /* We need inheritable handles. */
                                 {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
      PROCESS_INFORMATION ProcInfo1, ProcInfo2;
      STARTUPINFO StartInfoCh1, StartInfoCh2;
      char *CommandLine = GetCommandLine();
                                       /* Get command line as a string */

      /* Startup info for the two child processes. */

      GetStartupInfo (&StartInfoCh1);
      GetStartupInfo (&StartInfoCh2);

      CommandLine = strchr(CommandLine, ' ');
                                          /* Advances until next space */
      while (*CommandLine++ == ' ');               /* Advances spaces */

      i = 0;

      /* Get the two commands. */

      while (*CommandLine != '=' && *CommandLine != '\0') {
            Command1[i] = *CommandLine;
            CommandLine++; i++;
      }
      Command1[i] = '\0';
      CommandLine = strchr(CommandLine, ' ');
      while (*CommandLine++ == ' ');

      /* Create an anonymous pipe with default size.
            The handles are inheritable. */

      CreatePipe(&hReadPipe, &hWritePipe, &PipeSA, 0);

      /* Set the output handle to the inheritable pipe handle,
            and create the first processes. */

      StartInfoCh1.hStdInput  = GetStdHandle(STD_INPUT_HANDLE);
      StartInfoCh1.hStdError  = GetStdHandle(STD_ERROR_HANDLE);
      StartInfoCh1.hStdOutput = hWritePipe;
      StartInfoCh1.dwFlags = STARTF_USESTDHANDLES;
```

```
      CreateProcess (NULL, (LPTSTR)Command1, NULL, NULL,
                  TRUE,             /* Inherit handles. */
                  0, NULL, NULL, &StartInfoCh1, &ProcInfo1);
      CloseHandle (hWritePipe);

      /* Repeat (symmetrically) for the second process. */

      StartInfoCh2.hStdInput  = hReadPipe;
      StartInfoCh2.hStdError  = GetStdHandle (STD_ERROR_HANDLE);
      StartInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
      StartInfoCh2.dwFlags = STARTF_USESTDHANDLES;

      CreateProcess (NULL, (LPTSTR)CommandLine, NULL, NULL,
                  TRUE,             /* Inherit handles. */
                  0, NULL, NULL, &StartInfoCh2, &ProcInfo2);
      CloseHandle (hReadPipe);

      /* Wait for both processes to complete.
            The first one should finish first,
            although it really does not matter. */

      WaitForSingleObject (ProcInfo1.hProcess, INFINITE);
      WaitForSingleObject (ProcInfo2.hProcess, INFINITE);
      CloseHandle (ProcInfo1.hThread); CloseHandle (ProcInfo1.hProcess);
      CloseHandle (ProcInfo2.hThread); CloseHandle (ProcInfo2.hProcess);
      return 0;
}
```

8.3. Pipes com nome

Os pipes com nome em Win32 são sempre bidireccionais, mas de resto são semelhantes aos FIFOs do UNIX.

Têm de ser criados com o serviço `CreateNamedPipe()` que retorna um handle para o processo criador. Este processo não necessita de abrir o pipe e é geralmente o servidor. Os processos que necessitam de comunicar com o servidor têm de abrir o pipe para obter o seu handle com o serviço `CreateFile()`. Leituras e escritas de ambos os lados são efectuadas com `ReadFile()` e `WriteFile()`. Existem, entre outros, dois serviços para sincronizar as comunicações entre clientes e servidor: `WaitNamedPipe()`, que quando chamado por um cliente fica à espera que o servidor esteja livre para receber uma mensagem através do pipe; e `ConnectNamedPipe()`, chamado pelo servidor, indicando que está à espera de uma comunicação por parte de um cliente.

```
HANDLE CreateNamedPipe(char *PipeName, DWORD OpenMode, DWORD PipeMode,
                       DWORD MaxInstances, DWORD SizeOutBuffer,
                       DWORD SizeInBuffer, DWORD TimeOut,
                       SECURITY_ATTRIBUTES *sa);
```

O parâmetro **`PipeName`**, é o nome do pipe e tem de ser da forma `"\\.\pipe\[path]name"`; a abertura de um pipe já criado com `CreateFile()` tem de usar o mesmo nome;
`OpenMode` especifica um dos seguintes valores:

- `PIPE_ACCESS_DUPLEX`, para um pipe bidireccional;
- `PIPE_ACESS_INBOUND`, para um pipe do cliente para o servidor apenas; ou
- `PIPE_ACCESS_OUTBOUND`, para um pipe do servidor para o cliente apenas;

`PipeMode` pode ter três pares de valores mutuamente exclusivos:

- `PIPE_TYPE_BYTE` ou `PIPE_TYPE_MESSAGE`, que indica se o pipe é utilizado para um stream de bytes ou de mensagens (escritas individuais de blocos de bytes);
- `PIPE_READMODE_BYTE` ou `PIPE_READMODE_MESSAGE`, onde o primeiro indica que as leituras

podem ler qualquer nr. de bytes e o segundo, que só pode ser utilizado com PIPE_TYPE_MESSAGE, indica que apenas se pode ler uma mensagem completa;
- PIPE_WAIT ou PIPE_NOWAIT, onde se usa normalmente o primeiro (chamadas bloqueantes);

MaxInstances, é o número de clientes simultâneos, utilizado para construir servidores multi-*-thread*; **SizeOutBuffer** e **SizeInBuffer**, indicam o tamanho dos buffers de entrada e saída; deverá utilizar-se o valor 0 para um tamanho por defeito gerido pelo sistema; **TimeOut**, indica em ms o tempo de espera máximo do serviço WaitNamedPipe() se este usar a flag NMPWAIT_USE_DEFAULT_WAIT; finalmente o parâmetro **sa** indica os atributos de segurança, normalmente NULL, ou especificando que o handle é herdável.

O serviço retorna o handle do pipe, ou o valor INVALID_HANDLE_VALUE no caso de erro.

Normalmente o processo que cria o pipe com nome é considerado o processo servidor que fica à espera que algum outro processo se ligue a ele através do pipe. Para efectuar esse espera, de forma a não ocupar o processador, pode utilizar o serviço seguinte:

```
BOOL ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED Ovlp);
```

Coloca o processo à espera por uma ligação no pipe cujo handle é indicado em **hNamedPipe** por parte de um cliente. O parâmetro **Ovlp** é normalmente NULL.

Retorna quando houver uma ligação ao pipe por parte de um cliente. O retorno do valor FALSE indica um erro.

Quando o serviço anterior retorna (sem erro) existe uma ligação estabelecida através do pipe entre cliente e servidor. Após a troca de informação entre os processos via pipe, e se o servidor quiser esperar por nova ligação no mesmo pipe terá agora de desligar essa ligação antes de invocar novamente o serviço ConnectNamedPipe(). Para isso usa-se o serviço:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Desliga, do cliente, o lado do servidor do pipe indicado por **hNamedPipe**.

Retorna FALSE em caso de erro.

Quando um cliente pretende ligar-se a um pipe criado por um servidor, que fica à espera usando o serviço ConnectNamedPipe(), deverá usar o serviço WaitNamedPipe() para também esperar que o pipe do servidor esteja livre. Quando este retorna, deverá tentar-se a ligação imediatamente, usando o serviço CreateFile() com o nome do pipe. Se este último suceder a ligação está estabelecida, podendo-se efectuar a comunicação com os serviços WriteFile() e ReadFile().

Temos então para WaitNamedPipe():

```
BOOL WaitNamedPipe(char *PipeName, DWORD TimeOut);
```

Espera que o servidor execute o serviço ConnectNamedPipe(). O parâmetro **PipeName** indica o nome do pipe, criado pelo servidor, enquanto que **TimeOut** pode ser uma das constantes NMPWAIT_USE_DEFAULT_WAIT ou NMPWAIT_WAIT_FOREVER; no 1º caso a chamada espera até ao tempo máximo indicado em CreateNamedPipe(), enquanto que no 2º espera sempre até haver um ConnectNamedPipe(), tudo no lado do servidor.

Retorna FALSE no caso de erro.

Exemplo

Neste exemplo existe um servidor que recebe pedidos de execução de comandos por parte dos clientes, enviando depois para estes as possíveis respostas desses comandos. O servidor lê e executa um comando de cada vez através do mesmo pipe.

Cliente:

```
#include <windows.h>
#include <string.h>

#define MAX_RQRS_LEN 0x1000

int main (int argc, char *argv[])
{
      HANDLE hNamedPipe;
      char *pargv = GetCommandLine();
      char ServerPipeName[] = "\\\\.\\pipe\\server";
      char Request[MAX_RQRS_LEN];
      char Response[MAX_RQRS_LEN];
      DWORD nRead, nWrite;

      pargv = strchr(pargv, ' ');
      while (*pargv++ == ' ');

            /* Put the command line in a request to the server. */

      sprintf(Request, "%s", pargv);

            /* Run the command. */

      WaitNamedPipe(ServerPipeName, NMPWAIT_WAIT_FOREVER);
      hNamedPipe = CreateFile (ServerPipeName,
                               GENERIC_READ | GENERIC_WRITE,
                               0, NULL, OPEN_EXISTING,
                               FILE_ATTRIBUTE_NORMAL, NULL);

            /* Write the request. */

      WriteFile(hNamedPipe, Request, MAX_RQRS_LEN, &nWrite, NULL);

            /* Read each response and send it to std out. */

      while(ReadFile(hNamedPipe, Response, MAX_RQRS_LEN, &nRead, NULL))
            printf ("%s", Response);

      CloseHandle(hNamedPipe);
      return 0;
}
```

Servidor:

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

#define MAX_RQRS_LEN 0x1000
#define SERVER_PIPE "\\\\.\\pipe\server"

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

```
{
      BOOL Done = FALSE;
      HANDLE hNamedPipe, hTempFile;
      DWORD nXfer;
      STARTUPINFO StartInfo;
      SECURITY_ATTRIBUTES TempSA = {sizeof(SECURITY_ATTRIBUTES),
                                    NULL, TRUE};
      PROCESS_INFORMATION ProcInfo;
      char TempFile = "Results.tmp";
      FILE *fp;
      char Request[MAX_RQRS_LEN];
      char Response[MAX_RQRS_LEN];

      /* Startup info for the child and parent processes. */

      GetStartupInfo(&StartInfo);

      /* Create a single named pipe instance for clients to communicate
         with. We will use ConnectNamedPipe to wait for clients to connect
         (connecting clients will be refused if a previous request is in
         process). */

      hNamedPipe = CreateNamedPipe(SERVER_PIPE, PIPE_ACCESS_DUPLEX,
                    PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE | PIPE_WAIT,
                    1, 0, 0, CS_TIMEOUT, NULL);

      while (!Done) {                              /* Main Command Loop. */
            printf("Server is awaiting next request.\n");
            ConnectNamedPipe(hNamedPipe, NULL);
            ReadFile(hNamedPipe, Request, MAX_RQRS_LEN, &nXfer, NULL);
            printf("Request is: %s\n", Request);

            Done = strcmp(Request, "$Quit") == 0;

            if (Done) continue;   /* Stop the server on "$Quit" command. */

            /* Create a process to execute the command.
               Its output will go to temporary file. */

            hTempFile = CreateFile(TempFile, GENERIC_READ | GENERIC_WRITE,
                                   FILE_SHARE_READ | FILE_SHARE_WRITE,
                                   &TempSA, CREATE_ALWAYS,
                                   FILE_ATTRIBUTE_NORMAL, NULL);

            StartInfo.hStdOutput = hTempFile;
            StartInfo.hStdError = hTempFile;
            StartInfo.dwFlags = STARTF_USESTDHANDLES;
            if (!CreateProcess(NULL, Request, NULL, NULL,
                               TRUE, 0, NULL, NULL, &StartInfo, &ProcInfo))
                  WriteFile(hTempFile,
                            "Server could not process request.\n",
                            35, &nXfer, NULL);
            else {
                  CloseHandle (ProcInfo.hThread);
                  WaitForSingleObject (ProcInfo.hProcess, INFINITE);
                  CloseHandle (ProcInfo.hProcess);
            }
            CloseHandle (hTempFile);

            /* Send the temp file, one line at a time, to the client. */

            fp = fopen(TempFile, "r");
            while ((fgets(Response, MAX_RQRS_LEN, fp) != NULL))
                  WriteFile(hNamedPipe, Response, strlen(Response) + 1,
```

```
                              &nXfer, NULL);
            fclose (fp);
            DisconnectNamedPipe(hNamedPipe);
            DeleteFile(TempFile);
      }                       /* End of while loop and server operation. */

      DisconnectNamedPipe(hNamedPipe);
      CloseHandle(hNamedPipe);
      ExitProcess(0);
}
```

9. Memória partilhada

A API Win32 suporta a criação de blocos de memória partilhada entre processos através de um mecanismo utilizado também para mapear ficheiros normais (existentes no disco) no espaço de endereçamento de um processo.

Descreveremos de seguida apenas a forma de criar e utilizar os blocos de memória partilhada nos Sistemas Operativos Windows, sem entrarmos nos detalhes do mapeamento de ficheiros em memória.

Estes blocos são criados através de um objecto (represenrtado por um handle, como sempre) que se chama um *FileMapping*. O serviço para criar um *FileMapping* é o seguinte:

```
HANDLE CreateFileMapping(HANDLE hFile, SECURITY_ATTRIBUTES *sa, DWORD Protect,
                         DWORD MaxSizeHigh, DWORD MaxSizeLow, char *MapName);
```

Para a criação de blocos de memória partilhada são necessários alguns valores particulares dos parâmetros de criação do *FileMapping*. Assim, o parâmetro **`hFile`** terá de ter o valor `(HANDLE) 0xFFFFFFFF` (para mapear ficheiros deve ser o handle de um ficheiro aberto); **`sa`** indica os atributos de segurança deste objecto, e como sempre, para os atributos por defeito, deverá ser `NULL`, ou deverá ter o campo **`bInheritHandle`** em `TRUE`, se quisermos que o handle seja herdável; **`Protect`** indica o tipo do bloco de memória que se pretende, e pode tomar um dos valores:

- `PAGE_READONLY` - para blocos de leitura apenas, o que é pouco útil no caso de memória; ou
- `PAGE_READWRITE` - para blocos onde é possível a leitura e a escrita;

o tamanho do bloco indica-se (em bytes) através de um número de 64 bits dividido em 2 parâmetros de 32 bits cada, **`MaxSizeHigh`** e **`MaxSizeLow`** (nas máquinas de 32 bits **`MaxSizeHigh`** terá que ser 0); finalmente **`MapName`** contém uma string com o nome que se vai atribuir ao *FileMapping* criado - este nome serve depois para aceder a este *FileMapping* noutro processo.

O serviço retorna `NULL` no caso de erro.

Uma vez criado o objecto FileMapping este poderá ser acedido noutro processo digerente. Para isso o processo que pretenda aceder a este *FileMapping* (e através dele ao bloco de memória partilhado) deverá abri-lo, utilizando o nome com que foi criado. O serviço para efectuar essa abertura é então:

```
HANDLE OpenFileMapping(DWORD DesiredAccess, BOOL Inherit, char *Name);
```

Em **`DesiredAccess`** especifica-se uma das flags `PAGE_READONLY` ou `PAGE_READWRITE` com significado idêntico ao já indicado no serviço de criação, enquanto que no parâmetro **`Inherit`** se indica se o handle aqui obtido pode ou não ser herdado pelos processos filhos. Finalmente **`Name`** é o nome do *FileMapping* a abrir e que foi especificado quando da sua criação noutro processo.

Retorna `NULL` no caso de erro.

Na posse de um handle de um *FileMapping* (criado ou aberto neste processo ou simplesmente passado por outro mecanismo) é agora possível obter um apontador para uma área do bloco de memória representado pelo *FileMapping*. Este apontador pode permitir o acesso a todo o bloco de memória ou simplesmente (mais raramente) para uma área parcial desse bloco.

O serviço para obter o apontador é então:

```
void *MapViewOfFile(HANDLE hMapObject, DWORD Access, DWORD OffsetHigh,
                    DWORD OffsetLow, DWORD SizeMap);
```

O parâmetro **hMapObject** é o handle do *FileMapping*; **Access** deve ser um dos valores FILE_MAP_READ, FILE_MAP_WRITE ou FILE_MAP_ALL_ACCESS; **OffsetHigh** e OffsetLow, formam um valor de 64 bits indicando um *offset* (em bytes) relativamente ao início do bloco de memória representado pelo *FileMapping*, devendo este valor ser um múltiplo de 64 Kbytes; **SizeMap** indica o tamanho do sub-bloco que se quer aceder através do apontador (genérico) retornado (se **SizeMap** for 0 isso indica todo o bloco).

Retorna NULL no caso de erro.

Através do apontador obtido (que é um apontador genérico - `void *`) pode agora aceder-se às suas posições, fazendo um *cast* para o tipo de dados adequado.

Quando um processo, na posse de um *FileMapping* e de um apontador através dele, já não necessitar desta vista do mapeamento deve libertá-la, antes de proceder ao estabelecimento de outra vista, ou antes de libertar totalmente o *FileMapping*. A libertação de uma vista faz-se com o serviço:

```
BOOL UnmapViewOfFile(void *Address);
```

O parâmetro **Address** é o apontador obtido com o serviço anterior.

Retorna FALSE no caso de erro.

A libertação de *FileMapping*'s faz-se com o serviço `CloseHandle()`. Quando todos os handles referentes a um *FileMapping* forem fechados o bloco de memória partilhada é destruído (libertado) automaticamente pelo sistema operativo.