



Apontamentos Práticos

Encoding

Preparando a consola para aceitar caracteres wide:

Processos

Estrutura responsável de processos

Registry

DLLs

Ligação explícita

Ligação implícita

Para criar uma DLL

Threads

CreateThread

ThreadProc → protótipo das funções que irão ser lançadas como threads

WaitForSingleObject → função utilizada para esperar que uma thread termine

WaitForSingleObject vs WaitForMultipleObjects

Sincronização

Mutexes

Critical Sections

Eventos

Waitable Timers

Memória partilhada

Exercicio 2 alinea A da Ficha 6

Exercicio 2 b da ficha 6

Exercicio 2 d da ficha 6

Produtor e Consumidor

Código da main do Consumidor

Código da main do Produtor

Lógica final

Named Pipes

Escritor

Leitor

Interface Grafica Win32 GUI

Refrescamento de janelas

Bitmap

Exercicio 6 A

Exercicio 6 B → Deslocar o bitmap

Aula P 28/05/2021 → Sobre resources e afins

Encoding

No Visual Studio:

- Project → Properties → Advanced → Character Set → Se tivermo isto como `Use Unicode Character Set` estamos a dizer que este código roda em sistemas windows recentes, se usarmos `Use Multi-Byte Character Set` definimos que vamos trabalhar sempre em `ASCII`, ou seja, podemos trabalhar com sistemas operativos windows mais antigos.
- Alterando esta propriedade no Visual Studio não é suficiente para podermos trabalhar com caracteres especiais

Nota



Nos projetos temos de usar `Use Unicode Character Set` nos projetos e usar o `_setmode` para a consola perceber caracteres especiais

`char` → 1 byte → caracteres que estão na tabela ASCII

`wchar` → 2 bytes → armazena caracteres UNICODE

Solução (Em manipulação de Strings ou Caracteres):

- `TCHAR` → `TCHAR` é traduzido para `char` ou para `wchar` dependendo da propriedade do projeto
- `_tmain(int argc, TCHAR *argv[])` → assim ao usar o `_tmain` usa `char` ou `wchar` dependendo do que usamos
- `TEXT` → é uma macro do `TCHAR` que no momento da compilação vai ser traduzido para `" "` ou `L" "`

Nota



Esta solução só é feita para manipulação de Strings ou Caracteres, tudo o resto fica igual

Preparando a consola para aceitar caracteres wide:

- `_setmode(_fileno(stdin), _O_WTEXT);`
- `_setmode(_fileno(stdout), _O_WTEXT);`

Processos

- Os processos em Windows servem essencialmente para correr um programa que já é executável, ou seja, é diferente do Linux
- Em Windows não há nenhuma relação entre processo Pai e processo Filho
- Se quiser correr uma função crio uma `Thread`, se quiser correr um `Processo` crio um executavel para esse processo executar esse executavel
- Uma vantagem dos processos é eu poder por exemplo, executar o `mspaint` e conseguir interagir com o mesmo sem ter de criar um programa paint em si

`GetModuleFileName` → é uma maneira de sabermos o nome do próprio programa, podíamos usar `argv[0]` caso fosse um programa com parâmetros

Estrutura responsável de processos

- `STARTUPINFO si` → `si` representará como é que o novo processo vai ser inicializado → tem informações de redirecionamento, posições de consolas, etc etc
- `PROCESS_INFORMATION pi` → `pi` é o resultado da função → tem o pid do processo, tid da thread principal, handle (ponteiro para estrutura) do processo, handle para a thread, etc etc

Registry

O `registry` é um sistema centralizado de armazenamento do Windows acerca de todos os seus aspetos, por exemplo, funcionamento de componentes centrais, controlo de utilizador e politicas, configurações de aplicações, etc etc.

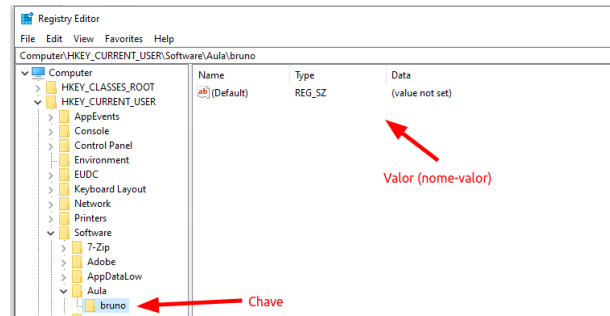
Os elementos principais constituintes do registry são as `chaves` e os `valores`

As chaves:

- Podem conter outras chaves
- Podem conter valores

Os valores:

- São um par `nome-valor`
- Podem assumir vários tipos de dados fazendo parte da especificação do `par-valor` na sua criação



- `HKEY_LOCAL_MACHINE` → Guarda informação física relativamente à máquina, assim como softwares instalados na mesma
- `HKEY_USERS` → Guarda informação das configurações do utilizador
- `HKEY_CURRENT_CONFIG` → Guarda informação corrente, como por exemplo a resolução do monitor e as fontes usadas
- `HKEY_CLASSES_ROOT` → Guarda informação relativamente às extensões dos ficheiros e com que aplicação se deve de abrir um ficheiro com uma extensão conhecida
- `HKEY_CURRENT_USER` → Guarda informação das variáveis de ambiente, preferencias de aplicações referentes ao utilizador atual da máquina

`HKEY` funciona como uma especie de ficheiro que se abre e se fecha

Handle para a chave depois de aberta/criada → `HKEY <chave>`

Fechar handle da chave no fim do programa → `RegCloseKey(<chave>)`

Função que cria uma chave → `RegCreateKeyEx`



Sempre que se usa um caracter especial, por exemplo `TCHAR chave[256] = TEXT("SOFTWARE\\S02\\chaveP5")` tenho de colocar `\\` para dar um `escape`

A API segue uma lógica semelhante ao restante API do Windows:

- **Abre-se** ou **Cria-se** uma chave obtendo-se um **handle**
- **Manipula-se** a chave usando o **handle** obtido
- **Fecha-se** o handle
- **Aplica-se** a esta API as mesmas considerações acerca de **chars, wide chars e TCHAR**

DLLs

Bibliotecas Dinâmicas → O código da biblioteca está isolado num ficheiro à parte, ficheiro .DLL

Bibliotecas Estáticas → Está diretamente agregada à aplicação, todo o código da biblioteca está embutido no executável

Ligação Explícita → Somos nós enquanto programadores que somos responsáveis por carregar a DLL para memória, importar as funções necessárias e depois limpar da memória da DLL → precisamos só do ficheiro .DLL mais nada

Ligação Implícita → O Sistema Operativo é que faz o trabalho explícito todo → O ficheiro .lib é uma pequena biblioteca estática que faz a ponte entre a nossa aplicação e a DLL → precisamos do .DLL, do .lib e do .h

Ligação explícita

Na ligação explícita o ficheiro DLL pode estar onde quisermos, no entanto convém colocar ao pé do executável para depois usar um caminho simples para a poder carregar

Como é que carregamos a biblioteca para memória?

- Com a função **LoadLibrary** → recebe o caminho para a DLL

Como é que acede aos recursos da DLL depois de a ter carregado para memória?

- Temos de trabalhar com ponteiros para variáveis e ponteiros para funções
- **GetProcAddress** → função que permite obter endereço de memória para função ou variável que uma DLL expoe
 - Recebe um **HMODULE** para a DLL que carregamos
 - Recebe o nome da função ou variável que queremos carregar → **Aqui nunca passamos UNICODE**

```
1 #include <Windows.h>
2 #include <Tchar.h>
3 #include <Tchar.h>
4 #include <Tchar.h>
5 #include <Tchar.h>
6 #include <Tchar.h>
7 #include <Tchar.h>
8 #include <Tchar.h>
9 #include <Tchar.h>
10 #include <Tchar.h>
11 #include <Tchar.h>
12 #include <Tchar.h>
13 #include <Tchar.h>
14 #include <Tchar.h>
15 #include <Tchar.h>
16 #include <Tchar.h>
17 #include <Tchar.h>
18 #include <Tchar.h>
19 #include <Tchar.h>
20 #include <Tchar.h>
21 #include <Tchar.h>
22 #include <Tchar.h>
23 #include <Tchar.h>
24 #include <Tchar.h>
25 #include <Tchar.h>
26 #include <Tchar.h>
27 #include <Tchar.h>
28 #include <Tchar.h>
29 #include <Tchar.h>
30 #include <Tchar.h>
31 #include <Tchar.h>
32 #include <Tchar.h>
33 #include <Tchar.h>
34 #include <Tchar.h>
35 #include <Tchar.h>
36 #include <Tchar.h>
37 #include <Tchar.h>
38 #include <Tchar.h>
39 #include <Tchar.h>
40 #include <Tchar.h>
41 #include <Tchar.h>
42 #include <Tchar.h>
43 #include <Tchar.h>
44 #include <Tchar.h>
45 #include <Tchar.h>
46 #include <Tchar.h>
47 #include <Tchar.h>
48 #include <Tchar.h>
49 #include <Tchar.h>
50 #include <Tchar.h>
51 #include <Tchar.h>
52 #include <Tchar.h>
53 #include <Tchar.h>
54 #include <Tchar.h>
55 #include <Tchar.h>
56 #include <Tchar.h>
57 #include <Tchar.h>
58 #include <Tchar.h>
59 #include <Tchar.h>
60 #include <Tchar.h>
61 #include <Tchar.h>
62 #include <Tchar.h>
63 #include <Tchar.h>
64 #include <Tchar.h>
65 #include <Tchar.h>
66 #include <Tchar.h>
67 #include <Tchar.h>
68 #include <Tchar.h>
69 #include <Tchar.h>
70 #include <Tchar.h>
71 #include <Tchar.h>
72 #include <Tchar.h>
73 #include <Tchar.h>
74 #include <Tchar.h>
75 #include <Tchar.h>
76 #include <Tchar.h>
77 #include <Tchar.h>
78 #include <Tchar.h>
79 #include <Tchar.h>
80 #include <Tchar.h>
81 #include <Tchar.h>
82 #include <Tchar.h>
83 #include <Tchar.h>
84 #include <Tchar.h>
85 #include <Tchar.h>
86 #include <Tchar.h>
87 #include <Tchar.h>
88 #include <Tchar.h>
89 #include <Tchar.h>
90 #include <Tchar.h>
91 #include <Tchar.h>
92 #include <Tchar.h>
93 #include <Tchar.h>
94 #include <Tchar.h>
95 #include <Tchar.h>
96 #include <Tchar.h>
97 #include <Tchar.h>
98 #include <Tchar.h>
99 #include <Tchar.h>
100 #include <Tchar.h>
```

Processo de carregamento de variável da DLL de forma explícita

```
1 //Definição e protótipo de função
2 typedef double (*appliedfactor)(double);
3
4 #include <Windows.h>
5 #include <Tchar.h>
6 #include <Tchar.h>
7 #include <Tchar.h>
8 #include <Tchar.h>
9 #include <Tchar.h>
10 #include <Tchar.h>
11 #include <Tchar.h>
12 #include <Tchar.h>
13 #include <Tchar.h>
14 #include <Tchar.h>
15 #include <Tchar.h>
16 #include <Tchar.h>
17 #include <Tchar.h>
18 #include <Tchar.h>
19 #include <Tchar.h>
20 #include <Tchar.h>
21 #include <Tchar.h>
22 #include <Tchar.h>
23 #include <Tchar.h>
24 #include <Tchar.h>
25 #include <Tchar.h>
26 #include <Tchar.h>
27 #include <Tchar.h>
28 #include <Tchar.h>
29 #include <Tchar.h>
30 #include <Tchar.h>
31 #include <Tchar.h>
32 #include <Tchar.h>
33 #include <Tchar.h>
34 #include <Tchar.h>
35 #include <Tchar.h>
36 #include <Tchar.h>
37 #include <Tchar.h>
38 #include <Tchar.h>
39 #include <Tchar.h>
40 #include <Tchar.h>
41 #include <Tchar.h>
42 #include <Tchar.h>
43 #include <Tchar.h>
44 #include <Tchar.h>
45 #include <Tchar.h>
46 #include <Tchar.h>
47 #include <Tchar.h>
48 #include <Tchar.h>
49 #include <Tchar.h>
50 #include <Tchar.h>
51 #include <Tchar.h>
52 #include <Tchar.h>
53 #include <Tchar.h>
54 #include <Tchar.h>
55 #include <Tchar.h>
56 #include <Tchar.h>
57 #include <Tchar.h>
58 #include <Tchar.h>
59 #include <Tchar.h>
60 #include <Tchar.h>
61 #include <Tchar.h>
62 #include <Tchar.h>
63 #include <Tchar.h>
64 #include <Tchar.h>
65 #include <Tchar.h>
66 #include <Tchar.h>
67 #include <Tchar.h>
68 #include <Tchar.h>
69 #include <Tchar.h>
70 #include <Tchar.h>
71 #include <Tchar.h>
72 #include <Tchar.h>
73 #include <Tchar.h>
74 #include <Tchar.h>
75 #include <Tchar.h>
76 #include <Tchar.h>
77 #include <Tchar.h>
78 #include <Tchar.h>
79 #include <Tchar.h>
80 #include <Tchar.h>
81 #include <Tchar.h>
82 #include <Tchar.h>
83 #include <Tchar.h>
84 #include <Tchar.h>
85 #include <Tchar.h>
86 #include <Tchar.h>
87 #include <Tchar.h>
88 #include <Tchar.h>
89 #include <Tchar.h>
90 #include <Tchar.h>
91 #include <Tchar.h>
92 #include <Tchar.h>
93 #include <Tchar.h>
94 #include <Tchar.h>
95 #include <Tchar.h>
96 #include <Tchar.h>
97 #include <Tchar.h>
98 #include <Tchar.h>
99 #include <Tchar.h>
100 #include <Tchar.h>
```

Processo de carregamento de função da DLL de forma explícita

Nunca esquecer de usar o FreeLibrary() para deslinkar a DLL



Basta carregar uma vez a DLL para memória e basta carregar uma vez os recursos dessa DLL e no fim do programa fazer o `FreeLibrary`

Caso dois processo usem a mesma DLL, esta é usada de forma independente, ou seja, não ha partilha de informação uma vez que cada processo vai carregar a DLL para o seu espaço de memória

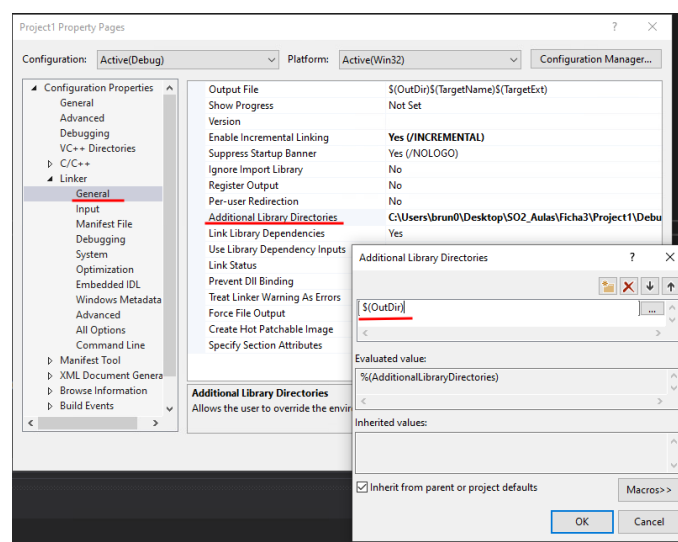
Ligação implícita

```
4 #include <fcntl.h>
5 #include <io.h>
6 #include <windows.h>
7 #include "SO2_F3_DLL.h"
8
9
10 int _tmain(int argc, TCHAR* argv[])
11 {
12     double a, b;
13
14     #ifdef UNICODE
15     _setmode(_fileno(stdin), _O_WTEXT);
16     _setmode(_fileno(stdout), _O_WTEXT);
17     #endif
18
19     do {
20         _tprintf(TEXT("Uma > "));
21         _tscanf(TEXT("%lf"), &a);
22         factor = a;
23         _tprintf(TEXT("Varável da DLL: %lf\n"), factor);
24         _tprintf(TEXT("Uma > "));
25         _tscanf(TEXT("%lf"), &b);
26         _tprintf(TEXT("Função da DLL: %lf\n"), applyFactor(b));
27     } while (factor != -1);
28
29     return 0;
30 }
```

Acrescentado o .h necessário à ligação implícita

Agora aqui já podemos usar a variável e a função da DLL diretamente, depois de fazer a ligação do ficheiro .lib

Uso da ligação implícita



No TP caso usemos a ligação implícita, para partilharmos código, podemos não fazer sempre as configurações do .lib usando isto

Para criar uma DLL

- Convém usar um empty project
- Criar uma macro para o `ifdef` e ir a `Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions` e criar a macro
 - Por norma a macro é sempre `NomeDLL_EXPORTS`

```
#pragma once
#include <windows.h>

#ifdef S02DLL_EXPORTS
#define DLL_IMP_API __declspec(dllexport)
#else
#define DLL_IMP_API __declspec(dllimport)
#endif

// declaracao de variaveis e funcoes , por exemplo
DLL_IMP_API double factor; // variavel
DLL_IMP_API double applyFactor(double v); // funcao
```

```
#include "S02_F3_DLL.h"

double factor = 1;

double applyFactor(double v) {
    return factor * v;
}
```

Ficheiro .c da DLL

```
#pragma once

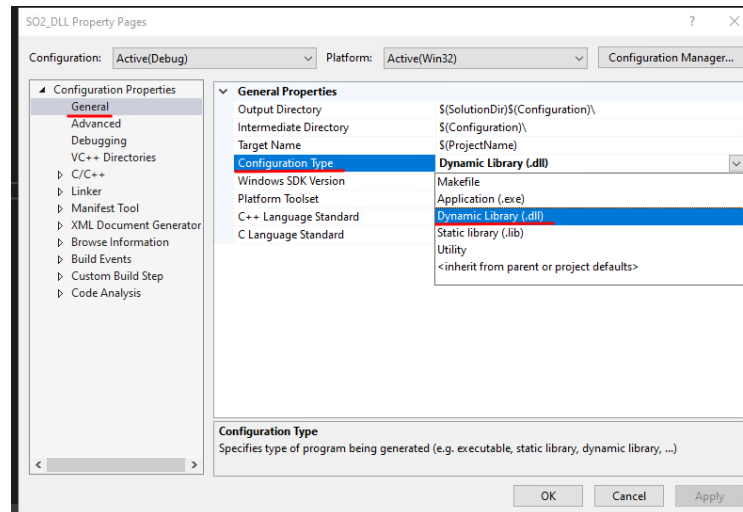
#include <windows.h>

#ifdef S02F3DLL_EXPORTS
#define DLL_IMP_API __declspec(dllexport)
#else
#define DLL_IMP_API __declspec(dllimport)
#endif

DLL_IMP_API double factor;

DLL_IMP_API double applyFactor(double v);
```

Ficheiro .h da DLL



Mudar o output do projeto para DLL

Threads

Quando temos um processo em execução, esse processo pode ter mais do que uma linha de execução.

Quando fazemos um programa básico com uma main so temos 1 linha de execução, caso queiramos fazer mais linhas de execução, temos de usar o conceito de `thread`

Regra geral as `thread` serão uma função com a característica que irão ser executadas em paralelo com a função `main`

As threads não são a unica forma de fazer paralelização de tarefas. Podemos fazer paralelização de tarefas com múltiplos processos

A grande vantagem de usarmos threads é que, como estamos no contexto do mesmo processo, todo o espaço de endereçamento de memória desse processo é partilhado pelas threads, ou seja a comunicação entre as threads é feita de forma muito simples

CreateThread

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress, //ponto de partida da nova thread -> funcao da thread
    __drv_aliasesMem LPVOID lpParameter, //ponteiro para void para podermos passar um parâmetro qualquer para dentro da funcao thread
    DWORD dwCreationFlags, // aqui podemos definir que a thread nao começa logo com o CREATE_SUSPENDED
    LPDWORD lpThreadId
);
// se correr bem retorna um HANDLE para a nova thread
```

ThreadProc → protótipo das funções que irão ser lançadas como threads

```
DWORD WINAPI ThreadProc(
    _In_ LPVOID pParameter
);
```

WaitForSingleObject → função utilizada para esperar que uma thread termine

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds  
);
```

WaitForSingleObject vs WaitForMultipleObjects

WaitForSingleObject esperamos por um único elemento, o **WaitForMultipleObjects** esperamos por varios elementos

```
DWORD WaitForMultipleObjects(  
    DWORD nCount, //numero de elementos que vai ter de esperar  
    const HANDLE *lpHandles, //ponteiro para HANDLES, ou seja, array de HANDLES  
    BOOL bWaitAll, //indifica se queremos esperar por 1 ou por todos  
    DWORD dwMilliseconds //quanto tempo espera  
);
```

NOTA



Ao utilizar o WaitForMultipleObjects tenho de garantir que o HANDLE pelo qual vou esperar está no índice 0. Se o índice que for desbloqueado for o 1 não há problema nenhum, no entanto se o índice que for desbloqueado for o 0 (na 1ª iteração) tenho um problema porque na segunda iteração não terei nenhum HANDLE no índice 0, logo a thread que está no índice 1 nunca vai ser desbloqueada. Resolvo isto fazendo um shift para a esquerda

HOW TO

```
if(indice == 0)  
    hThreads[0] = hThreads[1];
```

Criando uma thread com o `CreateThread` e usando a flag `CREATE_SUSPENDED` elas não começam logo a funcionar, para isso temos de usar o `ResumeThread`

Caso usemos `0` no `CreateThread` o mesmo cria a thread e lança-a logo

O `ResumeThread` diz ao sistema operativo para ir criando as thread todas ao mesmo tempo

Sincronização

Mutexes

- Permitem a duas ou mais threads (ou processos) aguardarem por acesso a uma zona de código (secção crítica) que manipula um recurso partilhado e cujo acesso simultâneo poria em causa a coerência desse recurso

- Os `mutexes` podem ser criados com um nome, ficando acessíveis a mais do que um processo. Se forem criados sem um nome, ficam restringidos a threads do mesmo processo.

No `CreateMutexW` o parâmetro `lpName` é uma string que basicamente representa o nome do Mutex para haver partilha de mutexes entre `Processos`

No `CreateMutexW` o parâmetro `bInitialOwner` diz se queremos começar logo com o lock do mutex para nós ou não. Caso não queiramos, usamos `FALSE`

Para ficarmos com o lock do Mutex para nós temos de usar o `WaitForSingleObject`

Para libertarmos o Mutex fazemos `ReleaseMutex`



Ao fazer exclusão mútua, ou seja, usar um mutex por exemplo, há um impacto enorme na performance do nosso código, uma vez que o código que temos deixou de fazer paralelismo ou seja, cada thread executa de uma vez só, daí ser muito lento

Critical Sections

- São uma forma otimizada de mutexes que `apenas servem para threads dentro do mesmo processo`. Permitem a mesma funcionalidade dos mutexes com menos custo de performance, apesar dessa diferença de performance depender de diversos fatores e não ser linear
- Permitem a especificação de um valor de `spin count`. `Spin count` é o número de vezes que a thread tenta repetidamente, e em espera ativa, obter acesso, antes de desistir e ficar bloqueada caso a `critical section` esteja ocupada.
- `Espera ativa` é em teoria indesejável, mas para valores de spin count baixos, tem-se a possibilidade real de um ganho de performance pois é menos custoso tentar entrar algumas vezes repetidamente e conseguir entrar do que a thread ficar logo bloqueada.
- Isto faz sentido quando se sabe à partida que o recurso que se pretende obter fica bloqueado por períodos muito curtos

Com o `Critical Section` existe um grande acréscimo de performance, usar isto no TP quando trabalhamos com threads do mesmo processo

Para inicializarmos a critical section na main, usamos a função `InitializeCriticalSectionAndSpinCount` que recebe um ponteiro para a critical section criada na main e o spin count que normalmente é 400 ou 500

Depois também temos de passar o ponteiro, criado na função aqui de cima, para as structs que trabalham com as threads

Para fazermos lock da critical section usamos a função `EnterCriticalSection` que recebe um ponteiro para a critical section

Para fazermos unlock da critical section usamos a função `LeaveCriticalSection` que recebe um ponteiro para a critical section

Eventos

- Servem para uma thread `indicar que algo aconteceu a uma ou mais threads que aguardavam por esse algo`
- Por exemplo, indicar a várias threads que `podem começar` a fazer algo
- Pode ser utilizado entre processos, no entanto temos de dar um nome ao evento para ser partilhado entre varios processos
- Usa-se a função `CreateEvent` para criar um evento, no segundo parâmetro escolhemos se ele irá começar com `reset automatico` ou `reset manual`

Quando nós assinalamos um evento com a função `SetEvent`, podemos querer que a notificação assim que seja lida, por uma thread ou um processo, automaticamente faça com que o evento volte a estar num estado `não assinalado` ou podemos querer que um evento depois de ser assinalado continue assinalado e que esse processo tenha de ser feito manualmente por nós com a função `ResetEvent`

- Se o evento for `automatico`, quando ele é assinalado, assim que à uma thread ou processo o use, ele passa para o estado `não assinalado`
- Se o evento for de `reset manual` quando ele é assinalado, ele fica sempre `assinalado` até manualmente mudarmos esse estado
- O `WaitForSingleObject` vai mais uma vez bloquear até esperarmos por um evento

Exemplo da ficha 5, exercício 5

```
/*
 * EVENTO ...
 * Neste caso vamos inicializar as threads todas de uma vez , retirando o CREATE_SUSPENDED do CreateThread
 * logo elas vao começar logo a executar mas vamos garantir que elas começam todas ao mesmo tempo
 * através de um Evento.
 *
 * As threads vao ficar todas bloqueadas num evento, a main ,depois de criar as threads todas, vai assinalar o evento(vai notificar as threads),
 * e elas vão todas desbloquear ao mesmo tempo
 */
```

```
//criamos o evento, atencao que esta a ser criado com reset automatico, logo vai passar a ter
// o estado nao assinalado assim que uma thread o desbloqueie
//logo é preciso assinalar novamente, para isso temos forçosamente de ter um
//evento de reset manual
//hEventGlobal = CreateEvent(NULL, FALSE, FALSE, NULL);
hEventGlobal = CreateEvent(NULL, TRUE, FALSE, NULL);
if (hEventGlobal == NULL) {
    _tprintf(TEXT("Erro a criar o evento\n"));
    return -1;
}
```

Reset automatico

Reset manual

Waitable Timers

- Trata-se de um mecanismo `semelhante aos eventos mas que permitem definir um período de tempo para desbloquear`
- Permitem implementar mecanismos de temporização de forma bastante simples

`Mutex` → só pode aceder 1 de cada vez, daí ser considerado por exclusão mutua

Semaforo → pode acceder N de cada vez

Memória partilhada

O mapeamento de ficheiros em memória permite trabalhar um ficheiro como se de um *array* se tratasse. A forma geral é a seguinte:

- Abre-se um ficheiro **CreateFile**
- Cria-se um objeto **FileMapping** associado a um ficheiro (**CreateFileMapping**).
- De seguida mapeia-se uma vista (uma parte ou o todo) desse objeto numa zona de memória (**MapViewOfFile**). O sistema indica o endereço onde ficou mapeada essa vista. Este endereço varia de caso para caso e não devem ser feitos pressupostos prévios.
- Manipula-se essa zona de memória através do ponteiro obtido no mapeamento da vista.
- Se for necessário por alguma razão, o programador pode forçar a atualização das alterações para o ficheiro num dado momento com **FlushViewOfFile**.
- Quando já não interessar trabalhar o ficheiro remove-se a vista do objeto (**UnmapViewOfFile**).
- Por fim remove-se o objeto **FileMapping** fechando o *handle* obtido com **CreateFileMapping**. O ficheiro deve ser também fechado (**CloseHandle**).

A partilha de memória entre processos recorre ao mesmo objeto `FileMapping`, sendo que neste caso o uso de um ficheiro é opcional. A forma geral é a seguinte

- Um dos processos envolvidos cria um objeto `FileMapping` (`CreateFileMapping`). O ficheiro é acessório. Os restantes processos abrem o objeto `FileMapping` (`OpenFileMapping`).
- Cada um dos processos mapeia uma vista do objeto `FileMapping` para memória (`MapViewOfFile`). Em cada processo, o endereço onde essa vista é colocada é diferente. Não deve ser assumido endereço nenhum exceto aquele que o sistema indica quando mapeia a vista.
- Cada processo usa a zona de memória correspondente à vista como entender (via ponteiro obtido).
- A sincronização é feita inteiramente a cargo do programador.
- No final dos processos desmapeiam a vista (`UnmapViewOfFile`). Quando o último processo fecha o último handle obtido com `CreateFileMapping` ou `OpenFileMapping` (`MapViewOfFile`), o objeto `FileMapping` é removido.

-> Podem-se mapear várias vistas do mesmo objeto *FileMapping*.

-> A vista pode abarcar apenas parte da zona total do objeto *FileMapping*. O início (offset) da vista deve ser múltiplo da unidade mínima de alocação. Essa informação pode ser obtida com **GetSystemInfo** e a estrutura **SYSTEM_INFO**.

-> Normalmente é necessário copiar conteúdo de/para a zona de memória partilhada. A função **CopyMemory** é útil neste sentido.

Exercicio 2 alinea A da Ficha 6

```

#define NITEMS 10
#define FILEMAP_SIZE 1024

#ifdef UNICODE
    _setmode(_fileno(stdin), _O_WTEXT);
    _setmode(_fileno(stdout), _O_WTEXT);
#else
    _setvbuf(stdin, NULL, _IONBF, 0);
    _setvbuf(stdout, NULL, _IONBF, 0);
#endif

// descomenta o teste e crie o arquivo no sistema operativo usando INVALID_HANDLE_VALUE, no 1º argumento
// do CreateFile, o filemapping vai ser criado mas vai ter criado somente os filemaps do sistema operativo
// de tamanho de tamanho de quantidade de bytes que queremos que o filemapping tenha > NUM_PAGES * sizeof(TCHAR)
// (temos que ter um nome também para ser partilhado entre processos)
// hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, NUM_PAGES * sizeof(TCHAR), TEXT("SOO MEMORIA PARTILHADA"));

if (!hFileMap == NULL) {
    _printf(TEXT("Erro no CreateFileMapping(N)"));
    return -1;
}

// descomenta o teste depois de abrir o filemapping para teste de memória
fileWinMap = (TCHAR*)MapViewOfFile(hFileMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

if (!fileWinMap == NULL) {
    _printf(TEXT("Erro no MapViewOfFile(N)"));
    return -1;
}

FlushViewOfFile(fileWinMap, 0);
// descomenta o teste depois de fechar o filemapping
UnmapViewOfFile(fileWinMap);
}

```

Aqui estamos a criar e a mapear , para o nosso espaço de endereçamento, um bloco de **memória partilhada**

```
// Tenho de partilhar o handle para o mutex e para o evento entre threads logo
// uso uma struct para todos os processos partilharem isso
typedef struct {
    TCHAR* fileViewMap; //ponteiro para a memória partilhada
    HANDLE hEvento; //handle para o evento
    HANDLE hMutex; //handle para o mutex
    int terminar; //flag que controla quando as threads terminam
}ThreadDados;
```

Estrutura essencial para a partilha de **Eventos** **Mutexes** e ponteiro para a memória partilhada

```

// Criação do evento reset manual e estado inicial é nao assinalado e tem de ter
// nome uma vez que vai ser partilhado entre processos
dados.hEvento = CreateEvent(NULL, TRUE, FALSE, TEXT("SO2_EVENTO"));

if (dados.hEvento == NULL) {
    _tprintf(TEXT("Erro no CreateEvent\n"));
    UnmapViewOfFile(dados.fileViewMap);
    return -1;
}

// criação do mutex
dados.hMutex = CreateMutex(NULL, FALSE, TEXT("SO2_MUTEX"));
if (dados.hMutex == NULL) {
    _tprintf(TEXT("Erro no CreateMutex\n"));
    UnmapViewOfFile(dados.fileViewMap);
    return -1;
}

```

```

// Thread que vai ler da memoria partilhada
DWORD WINAPI ThreadLer(LPVOID param) {
    ThreadDados* dados = (ThreadDados*)param;

    while (1) {
        // bloqueia à espera que o evento fique assinalado
        WaitForSingleObject(dados->hEvento, INFINITE);

        //se a flag terminar estiver a 1, termina
        if (dados->terminar == 1) {
            break;
        }

        // bloqueia à espera do unlock do mutex
        WaitForSingleObject(dados->hMutex, INFINITE);

        //aqui o evento desbloqueou, posso mostrar a mensagem
        _tprintf(TEXT("Mensagem recebida : %s\n"), dados->fileViewMap);

        // faço release do mutex para mostrar que ja acabei de ler
        ReleaseMutex(dados->hMutex);

        // faço sleep e é importante isto nao estar dentro da zona critica
        Sleep(1000);
    }

    return 0;
}

```

Thread que vai ler da memória partilhada

```

// Thread que vai escrever na memoria partilhada
DWORD WINAPI ThreadEscrever(LPVOID param) {
    ThreadDados* dados = (ThreadDados*)param;
    TCHAR msg[NUM_CHARS]; // array temporario para ler da consola

    while (!(dados->terminar)) {
        // lemos do stdin e colocamos no array temporario msg
        _fgetts(msg, NUM_CHARS, stdin);
        msg[_tcslen(msg)-1] = '\0';

        if (_tcsncmp(msg, TEXT("fim")) == 0)
            dados->terminar = 1;

        //esperamos que o mutex esteja unlock
        WaitForSingleObject(dados->hMutex, INFINITE);

        //convem limpamos a memoria partilhada antes de copiarmos
        ZeroMemory(dados->fileViewMap, NUM_CHARS);

        //vamos copiar o conteudo que esta em msg para a memoria partilhada
        CopyMemory(dados->fileViewMap, msg, _tcslen(msg) * sizeof(TCHAR));

        //libertamos o mutex
        ReleaseMutex(dados->hMutex);

        //assinalamos o evento para que a outra thread ja consiga ler
        SetEvent(dados->hEvento);

        Sleep(500);
        //reset event para ele ficar nao assinalado novamente
        ResetEvent(dados->hEvento);
    }

    return 0;
}

```

Thread que vai escrever na memória partilhada

```

dados.terminar = 0;
HANDLE hThreads[2]; //cada processo cria a thread e leitura e de escrita
hThreads[0] = CreateThread(NULL, 0, &ThreadLer, &dados, 0, NULL);
hThreads[1] = CreateThread(NULL, 0, &ThreadEscrever, &dados, 0, NULL);

if (hThreads[0] != NULL && hThreads[1] != NULL) {
    //se correu tudo bem vamos esperar que ambas as threads terminem
    WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);
}

```

Inicializa as **Threads**

Exercicio 2 b da ficha 6

Só acrescentamos o **Semaforo** para limitarmos o nr de pessoas no chat

```

// antes de criar a memória partilhada vamos criar o semaforo
// o semaforo tem dois slots e os dois vão começar disponíveis
// precisa de um nome para ser partilhado entre processos
hSemaforo = CreateSemaphore(NULL, 2, 2, TEXT("S02_SEMAFORO"));
if (hSemaforo == NULL) {
    _tprintf(TEXT("Erro no CreateSemaphore\n"));
    return -1;
}

_tprintf(TEXT("A aguardar por um slot ... \n"));

// bloqueamos à espera de um slot para nós
WaitForSingleObject(hSemaforo, INFINITE);

// quando ja tivermos espaço , iniciamos
_tprintf(TEXT("Chat a iniciar ... \n"));

```

```

// quando formos terminar este processo, libertamos o Semaforo para indicar que ficou
// um slot disponível
ReleaseSemaphore(hSemaforo, 1, NULL);

```

Exercicio 2 d da ficha 6

Só usamos **WaitableTimers** em vez do Sleep

```

//criacao do waitabel timer
dados.hTimer = CreateWaitableTimer(NULL, TRUE, TEXT("S02_TIMER"));
if (dados.hTimer == NULL) {
    _tprintf(TEXT("Erro no CreateWaitableTimer\n"));
    UnmapViewOfFile(dados.fileViewMap);
    return -1;
}

```

Criação do **WaitableTimer** com **Reset Automático**

```

// Thread que vai ler da memoria partilhada
DWORD WINAPI ThreadLer(LPVOID param) {
    ThreadDados* dados = (ThreadDados*)param;

    LARGE_INTEGER tempo;
    tempo.QuadPart = 1000;

    while (1) {
        // bloqueia à espera que o evento fique assinalado
        WaitForSingleObject(dados->hEvento, INFINITE);

        //se a flag terminar estiver a 1, termina
        if (dados->terminar == 1) {
            break;
        }

        // bloqueia à espera do unlock do mutex
        WaitForSingleObject(dados->hMutex, INFINITE);

        //aqui o evento desbloqueou, posso mostrar a mensagem
        _tprintf(TEXT("Mensagem recebida : %s\n"), dados->fileViewMap);

        // faço release do mutex para mostrar que ja acabei de ler
        ReleaseMutex(dados->hMutex);

        // faço set do waitable timer para 10 segundos
        SetWaitableTimer(dados->hTimer, &tempo, 0, NULL, NULL, 0);
        //espero que o waitable timer chegue ao fim
        WaitForSingleObject(dados->hTimer, INFINITE);
    }

    return 0;
}

```

```

// Thread que vai escrever na memoria partilhada
DWORD WINAPI ThreadEscrever(LPVOID param) {
    ThreadDados* dados = (ThreadDados*)param;
    TCHAR msg[NUM_CHARS]; // array temporario para ler da consola

    LARGE_INTEGER tempo;
    tempo.QuadPart = 500;

    while (!dados->terminar) {

        // lemos do stdin e colocamos no array temporario msg
        _fgetts(msg, NUM_CHARS, stdin);
        msg[_tcslen(msg) - 1] = '\0';

        if (_tcscmp(msg, TEXT("fim")) == 0)
            dados->terminar = 1;

        //esperamos que o mutex esteja unlock
        WaitForSingleObject(dados->hMutex, INFINITE);

        //conven limparmos a memoria partilhada antes de copiarmos
        ZeroMemory(dados->fileMapView, NUM_CHARS);

        //vamos copiar o conteudo que esta em msg para a memoria partilhada
        CopyMemory(dados->fileMapView, msg, _tcslen(msg) * sizeof(TCHAR));

        //libertamos o mutex
        ReleaseMutex(dados->hMutex);

        //assinalamos o evento para que a outra thread ja consiga ler
        SetEvent(dados->hEvento);

        SetWaitableTimer(dados->hTimer, &tempo, 0, NULL, NULL, 0);
        WaitForSingleObject(dados->hTimer, INFINITE);
        //reset event para ele ficar nao assinalado novamente
        ResetEvent(dados->hEvento);
    }

    return 0;
}

```

Em ambas as Threads foram feitos o `SetWaitableTimer` e o `WaitForSingleObject` para nao usar o `Sleep` uma vez que é má prática

Produtor e Consumidor



Este exemplo foi usado `N produtores` e `M consumidores` dai ser preciso um mutex interno para ambos, caso fosse `1 consumidor` e `N produtores` só o produtor é que precisaria de um mutex interno

```

//estrutura para o buffer circular
typedef struct {
    int id;
    int val;
} CelulaBuffer;

//representa a nossa memoria partilhada
typedef struct {
    int nProdutores;
    int nConsumidores;
    int posE; //proxima posicao de escrita
    int posL; //proxima posicao de leitura
    CelulaBuffer buffer[TAM_BUFFER]; //buffer circular em si (array de estruturas)
} BufferCircular;

//estrutura de apoio
typedef struct {
    BufferCircular* memPar; //ponteiro para a memoria partilhada
    HANDLE hSemEscrita; //handle para o semaforo que controla as escritas (controla quantas posicoes estao vazias)
    HANDLE hSemLeitura; //handle para o semaforo que controla as leituras (controla quantas posicoes estao preenchidas)
    HANDLE hMutex;
    int terminar; // 1 para sair, 0 em caso contrario
    int id;
} DadosThreads;

```

Estruturas usadas para o Consumidor e para o Produtor

```

ERROR WINAPI ThreadConsumer(LPVOID param) {
    DadosThread dados = (DadosThread*)param;
    CircularBuffer cel;
    int contador = 0;
    int soma = 0;

    while ((dados->terminar) == 0) {
        //aquí entramos na lógica de nossa teoria

        //esperamos por uma posição para lermos
        WaitForSingleObject(dados->hSemLeitura, INFINITE); // Esperamos que o semáforo de leitura esteja livre

        //esperamos que o mutex esteja livre
        WaitForSingleObject(dados->hMutex, INFINITE); // Esperamos que o mutex esteja livre

        //fazemos o copy de dados do buffer de leitura para o nosso variável cel
        CopyMemory(&cel, &dados->memPar->buffer[dados->memPar->pos], sizeof(CircularBuffer));
        dados->memPar->pos++; //incrementamos a posição de leitura para o próximo consumidor ler na posição seguinte

        //se após o incremento a posição de leitura chegar ao fim, temos de voltar ao início
        if (dados->memPar->pos == TAM_BUFFER)
            dados->memPar->pos = 0; // Lógica necessária para o buffer circular

        //libertamos o mutex
        ReleaseMutex(dados->hMutex); // Libertamos o mutex

        //libertamos o semáforo, temos de libertar uma posição de escrita
        ReleaseSemaphore(dados->hSemEscrita, 1, NULL); // Libertamos o semáforo de escrita para o produtor

        contador++;
        soma += cel.val;
        _printf(TEXT("Cód consumidor %d\n", dados->id, cel.val));
        _printf(TEXT("Cód consumidor %d items\n", dados->id, soma));
    }
    return 0;
}

```

Thread do consumidor

```

DWORD WINAPI ThreadProducer(LPVOID param) {
    DadosThread dados = (DadosThread*)param;
    CircularBuffer cel;
    int contador = 0;

    while ((dados->terminar) == 0) {
        cel.id = dados->id;
        cel.val = rand_alotriado(10, 99);

        //aquí entramos na lógica de nossa teoria

        //esperamos por uma posição para escrevermos
        WaitForSingleObject(dados->hSemEscrita, INFINITE); // Esperamos que o semáforo de escrita esteja livre

        //esperamos que o mutex esteja livre
        WaitForSingleObject(dados->hMutex, INFINITE); // Esperamos que o mutex esteja livre

        //fazemos o copy de dados para o nosso variável cel (para a posição de escrita)
        CopyMemory(&cel, &dados->memPar->buffer[dados->memPar->pos], sizeof(CircularBuffer));
        dados->memPar->pos++; //incrementamos a posição de escrita para o próximo produtor escrever na posição seguinte

        //se após o incremento a posição de escrita chegar ao fim, temos de voltar ao início
        if (dados->memPar->pos == TAM_BUFFER)
            dados->memPar->pos = 0; // Lógica para o buffer ser um buffer circular

        //libertamos o mutex
        ReleaseMutex(dados->hMutex); // Libertamos o mutex

        //libertamos o semáforo, temos de libertar uma posição de leitura
        ReleaseSemaphore(dados->hSemLeitura, 1, NULL); // Liberto o semáforo de leitura para o consumidor conseguir ler

        contador++;
        _printf(TEXT("Prod produtor %d\n", dados->id, cel.val));
        Sleep(rand_alotriado(1, 4) * 1000);
    }
    _printf(TEXT("Prod produtor %d items\n", dados->id, contador));
    return 0;
}

```

Thread do produtor

Código da main do Consumidor

```

int _tmain(int argc, TCHAR* argv[])
{
    HANDLE hFileMap; //handle para o file map
    HANDLE hThread;
    DadosThread dados;
    BOOL primeiroProcesso = FALSE;
    TCHAR comando[100];

#ifdef UNICODE
    _setmode(_fileno(stdin), _O_WTEXT);
    _setmode(_fileno(stdout), _O_WTEXT);
#endif

    //criar semáforo que conta as escritas
    dados.hSemEscrita = CreateSemaphore(NULL, TAM_BUFFER, TAM_BUFFER, TEXT("SO2_SEMAFORO_ESCRITA"));

    //criar semáforo que conta as leituras
    //8 porque não há nada para ser lido e depois podemos ir até um máximo de 10 posições para serem lidas
    dados.hSemLeitura = CreateSemaphore(NULL, 0, TAM_BUFFER, TEXT("SO2_SEMAFORO_LEITURA"));

    //criar mutex para os produtores
    dados.hMutex = CreateMutex(NULL, FALSE, TEXT("SO2_MUTEX_CONSUMIDOR"));

    if (dados.hSemEscrita == NULL || dados.hSemLeitura == NULL || dados.hMutex == NULL) {
        _tprintf(TEXT("Erro no CreateSemaphore ou no CreateMutex\n"));
        return -1;
    }
}

```

Verifica se os semáforos e mutex tiveram algum erro ao serem criados

```

//o openfilemapping vai abrir um filemapping com o nome que passamos no lpname
//se devolver um NULL, se existe e não fazemos a inicialização
//se devolver NULL não existe e vamos fazer a inicialização

hFileMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, TEXT("SO2_MEM_PARTILHADA"));

if (hFileMap == NULL) {
    primeiroProcesso = TRUE;
    //criamos o bloco de memória partilhada
    hFileMap = CreateFileMapping(
        INVALID_HANDLE_VALUE,
        NULL,
        PAGE_READWRITE,
        0,
        sizeof(CircularBuffer), //tamanho de memória partilhada
        TEXT("SO2_MEM_PARTILHADA")); //nome do filemapping, nome que vai ser usado para partilha entre processos

    if (hFileMap == NULL) {
        _tprintf(TEXT("Erro no CreateFileMapping\n"));
        return -1;
    }

    //passamos o bloco de memória para o espaço de endereçamento do nosso processo
    dados->memPar = (CircularBuffer*)MapViewOfFile(hFileMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    if (dados->memPar == NULL) {
        _tprintf(TEXT("Erro no MapViewOfFile\n"));
        return -1;
    }

    if (primeiroProcesso == TRUE) {
        dados->memPar->consumidores = 0;
        dados->memPar->produtores = 0;
        dados->memPar->pos = 0;
        dados->memPar->pos = 0;

        //caso seja o 1º processo, então vamos inicializar a estrutura, caso contrário
        //essa estrutura já foi inicializada
    }
}

```

```

//fazemos o copy de dados para os consumidores para termos os dados corretos
WaitForSingleObject(dados->hMutex, INFINITE);
dados->memPar->consumidores++;
dados->id = dados->memPar->consumidores;
ReleaseMutex(dados->hMutex);

//criamos o thread
hThread = CreateThread(NULL, 0, ThreadConsumer, &dados, 0, NULL);
if (hThread != NULL) {
    _tprintf(TEXT("Lancamos qualquer coisa para sair ...n\n"));
    _getch(&comando[100]);
    dados->terminar = 1;
    //criamos o thread de leitura
    WaitForSingleObject(hThread, INFINITE);
}

UnmapViewOfFile(dados->memPar);
//desalocamos ... mas o fazemos automaticamente quando o processo termina
return 0;

```

Código da main do Produtor

```
int main(int argc, char* argv[])
{
    HANDLE hFileMap; //handle para o file map
    HANDLE hThread;
    dados_t dados;
    BOOL primeiroProcesso = FALSE;
    TCHAR comando[100];

    if (argc < 2)
    {
        _wprintf(L"Uso: %s <comando>\n", argv[0]);
        return -1;
    }

    //cria semaforo que conta as escritas
    dados.hSemEscr = CreateSemaphore(NULL, 100, 100, TEXT("SOZ_SEMAFORO_ESCRITA"));

    //cria semaforo que conta as leituras
    //o porque nao ha nada para ser lido e depois podemos ler até um maximo de 10 posicoes para serem lidas
    dados.hSemLeitura = CreateSemaphore(NULL, 0, 10, TEXT("SOZ_SEMAFORO_LEITURA"));

    //cria mutex para os produtores
    dados.hMutex = CreateMutex(NULL, FALSE, TEXT("SOZ_MUTEX_PRODUTOR"));

    if (dados.hSemEscr == NULL || dados.hSemLeitura == NULL || dados.hMutex == NULL) {
        _wprintf(TEXT("Erro na CreateSemaphore ou na CreateMutex\n"));
        return -1;
    }
}
```

Isto é exatamente igual
como era no consumidor

Ou seja, criação de semaforos
de leitura e escrita e criação
do mutex

```
//se o filemapping vai abrir um filemapping com o nome que passamos no Open
//se devolver um HANDLE ja existe e nao fazemos a inicializacao
//se devolver NULL nao existe e vamos fazer a inicializacao
hFileMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, TEXT("SOZ_MEM_PARTILHADA"));

if (hFileMap == NULL) {
    primeiroProcesso = TRUE;
    //criamos o bloco de memoria partilhada
    hFileMap = CreateFileMapping(
        INVALID_HANDLE_VALUE,
        NULL,
        PAGE_READWRITE,
        0,
        sizeof(BufferCircular), //tamanho da memoria partilhada
        TEXT("SOZ_MEM_PARTILHADA")); //nome do filemapping, nome que vai ser usado para partilha entre processos

    if (hFileMap == NULL) {
        _wprintf(TEXT("Erro na CreateFileMapping\n"));
        return -1;
    }

    //reservamos o bloco de memoria para o acesso de endereçamento do nosso processo
    dados.memPar = (BufferCircular*)MapViewOfFile(hFileMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    if (dados.memPar == NULL) {
        _wprintf(TEXT("Erro na MapViewOfFile\n"));
        return -1;
    }

    if (primeiroProcesso == TRUE) {
        dados.memPar->nConsumidores = 0;
        dados.memPar->nProdutores = 0;
        dados.memPar->spot = 0;
        dados.memPar->spot = 0;
    }
}
```

Este processo (verificar se o filemapping ja esta ou nao
criado, se nao temos de o criar, caso contrario abrimos,
é exatamente igual ao código do consumidor

Caso seja o 1º processo, tambem inicializa
a estrutura, assim como faziamos no consumidor

```
dados.terminar = 0;

//temos de usar o mutex para aumentar o nProdutores para termos os ids corretos
WaitForSingleObject(dados.hMutex, INFINITE);
dados.memPar->nProdutores++;
dados.id = dados.memPar->nProdutores;
ReleaseMutex(dados.hMutex);

//lançamos a thread
hThread = CreateThread(NULL, 0, ThreadProdutor, &dados, 0, NULL);
if (hThread != NULL) {
    _tprintf(TEXT("Escreva qualquer coisa para sair ...\n"));
    _getts_s(comando, 100);
    dados.terminar = 1;

    //esperar que a thread termine
    WaitForSingleObject(hThread, INFINITE);
}

UnmapViewOfFile(dados.memPar);
//CloseHandles ... mas é feito automaticamente quando o processo termina

return 0;
}
```

Continuação da inicialização da estrutura

Lançamos a thread
do produtor e esperamos
que o mesmo escreva
qualquer coisa para tudo
terminar

No fim, fazemos unmap da memória partilhada

Lógica final

Tendo N produtores e M consumidores

• Produtor

- O produtor começa por esperar por uma posição no **semaforo de escrita**, ele **espera usando o WaitForSingleObject**
- Depois **espera pelo mutex** tambem com o **WaitForSingleObject**
- Quando o **mutex estiver livre, ele produz**
- Quando acaba de produzir **liberta o mutex**
- Depois faz **ReleaseSemaphore** do **semaforo de leitura** a dizer que o consumidor já pode consumir

• Consumidor

- Começa por esperar por uma posição no `semaforo de leitura`, ele `espera usando o WaitForSingleObject`
- Depois `espera pelo mutex` também com o `WaitForSingleObject`
- Quando o `mutex estiver livre, ele consome`
- Quando acaba de consumir `liberta o mutex`
- Depois faz `ReleaseSemaphore` do `semaforo de escrita` a dizer que o produtor já pode produzir

Named Pipes

- Os programas escritor e leitor permitem a troca de mensagens entre dois processos através de Named Pipes.
- A troca de mensagens com um dado leitor termina quando o utilizador introduz a palavra "fim" que é lido pelo programa escritor.

Escritor

```

7
8 #define PIPE_NAME TEXT("\\\\.\\pipe\\teste")
9 #define N 10
10
11 // estrutura do named pipe
12 typedef struct {
13     HANDLE hPipe; // handle do pipe
14     OVERLAPPED overlap;
15     BOOL activo; //representa se a instancia do named pipe esta ou nao ativa, se ja tem um cliente ou nao
16 } PipeDados;
17
18
19 typedef struct {
20     PipeDados hPipes[N];
21     HANDLE hEvents[N];
22     HANDLE hMutex;
23     int terminar;
24 } ThreadDados;
25

```

Nome do pipe, tem de ter \\

Estrutura overlapped para uso asincrono

Structs e pipe name

```

//envia mensagem para todos os leitores que estão disponíveis
DWORD WINAPI ThreadMensagens(LPVOID param) {
    ThreadDados* dados = (ThreadDados*)param;
    TCHAR buf[256];
    DWORD n;
    int i;

    do {
        _tprintf(TEXT("[ESCRITOR] frase: "));
        _fgetts(buf, 256, stdin);
        buf[_tcslen(buf) - 1] = '\0';

        //escreve no named pipe
        for (i = 0; i < N; i++) {
            //bloqueamos aqui porque é uma região critica
            WaitForSingleObject(dados->hMutex, INFINITE);

            // este named pipe está ativo? se sim vou escrever nele
            if (dados->hPipes[i].activo) {
                if (!WriteFile(dados->hPipes[i].hPipe, buf, _tcslen(buf) * sizeof(TCHAR), &n, NULL))
                    _tprintf(TEXT("[ERRO] escrever no pipe! (writefile)\n"));
                else
                    _tprintf(TEXT("[ESCRITOR] Enviai %d bytes ao leitor [%d]... (writefile)\n"), n, i);
            }

            //libertamos o mutex
            ReleaseMutex(dados->hMutex);
        }

        while (_tcsncmp(buf, TEXT("fim")))
            dados->terminar = 1;

        //vou assinalar todos os eventos para que nao fique bloqueado na main
        for (i = 0; i < N; i++)
            SetEvent(dados->hEvents[i]);

        return 0;
    }
}

```

Aqui só escrevo no named pipe, verifico se ele está ativo e se posso escrever usando o Mutex

←

Thread para escrever mensagens

```

int _tmain(int argc, LPTSTR argv[]) {

    HANDLE hPipe, hThread, hEventTemp;
    ThreadDados dados;
    int i, numClientes = 0;
    DWORD offset, nBytes;

#ifdef UNICODE
    _setmode(_fileno(stdin), _O_WTEXT);
    _setmode(_fileno(stdout), _O_WTEXT);
#endif

    dados.terminar = 0;
    dados.hMutex = CreateMutex(NULL, FALSE, NULL); //Criação do mutex

    if (dados.hMutex == NULL) {
        _tprintf(TEXT("[Erro] ao criar mutex!\n"));
        return -1;
    }

    for (i = 0; i < N; i++) {

        // aqui passamos a constante FILE_FLAG_OVERLAPPED para o named pipe aceitar comunicações assíncronas
        hPipe = CreateNamedPipe(PIPE_NAME, PIPE_ACCESS_OUTBOUND | FILE_FLAG_OVERLAPPED,
            PIPE_WAIT | PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE,
            N,
            256 * sizeof(TCHAR),
            256 * sizeof(TCHAR),
            1000,
            NULL);

        if (hPipe == INVALID_HANDLE_VALUE) {
            _tprintf(TEXT("[ERRO] Criar Named Pipe! (CreateNamedPipe)"));
            exit(-1);
        }
    }
}

```

main 1

```

// criar evento que vai ser associado à estrutura overlaped
// os eventos aqui tem de ter sempre reset manual e não automático porque temos de delegar essas responsabilidades ao sistema operativo
hEventTemp = CreateEvent(NULL, TRUE, FALSE, NULL);

if (hEventTemp == NULL) {
    _tprintf(TEXT("[ERRO] ao criar evento\n"));
    return -1;
}

dados.hPipes[i].hPipe = hPipe;
dados.hPipes[i].ativo = FALSE;
//temos de garantir que a estrutura overlap está limpa
ZeroMemory(&dados.hPipes[i].overlap, sizeof(dados.hPipes[i].overlap));
//preenchemos agora o evento
dados.hPipes[i].overlap.hEvent = hEventTemp;
dados.hEvents[i] = hEventTemp;

_tprintf(TEXT("[ESCRITOR] Esperar ligação de um leitor... (ConnectNamedPipe)\n"));

// aqui passamos um ponteiro para a estrutura overlap
ConnectNamedPipe(hPipe, &dados.hPipes[i].overlap);

//criacao da thread
hThread = CreateThread(NULL, 0, ThreadMensagens, &dados, 0, NULL);

if (hThread == NULL) {
    _tprintf(TEXT("[Erro] ao criar thread!\n"));
    return -1;
}

```

main 2

```

while (!dados.terminar && numClientes < N) {
    //permite estar bloqueado , à espera que 1 evento do array de eventos seja assinalado
    offset = WaitForMultipleObjects(N, dados.hEvents, FALSE, INFINITE);
    i = offset - WAIT_OBJECT_0; // devolve o índice da instancia do named pipe que está ativa, aqui sabemos em que índice o cliente se ligou

    // se é um índice válido ...
    if (i >= 0 && i < N) {
        _tprintf(TEXT("[ESCRITOR] Leitor %d chegou\n"), i);
        if (GetOverlappedResult(dados.hPipes[i].hPipe, &dados.hPipes[i].overlap, &nBytes, FALSE)) {
            // Se entrarmos aqui significa que a funcao correu tudo bem
            // fazemos reset do evento porque queremos que o WaitForMultipleObject desbloqueio com base noutro evento e nao neste
            ResetEvent(dados.hEvents[i]);

            //vamos esperar que o mutex esteja livre
            WaitForSingleObject(dados.hMutex, INFINITE);
            dados.hPipes[i].ativo = TRUE; // dizemos que esta instancia do named pipe está ativa
            ReleaseMutex(dados.hMutex);

            numClientes++;
        }
    }

    //esperar que a thread termine
    WaitForSingleObject(hThread, INFINITE);

    //desligamos todos os clientes que se ligaram
    for (int i = 0; i < N; i++) {
        _tprintf(TEXT("[ESCRITOR] Desligar o pipe (DisconnectNamedPipe)\n"));

        //desliga todas as instancias de named pipes
        if (!DisconnectNamedPipe(dados.hPipes[i].hPipe)) {
            _tprintf(TEXT("[ERRO] Desligar o pipe! (DisconnectNamedPipe)"));
            exit(-1);
        }
    }
}

exit(0);

```

main 3

Leitor

```

#define PIPE_NAME TEXT("\\\\.\\pipe\\teste")

int _tmain(int argc, LPCTSTR argv[]) {
    TCHAR buf[256];
    HANDLE hPipe;
    int i = 0;
    BOOL ret;
    DWORD n;

    if (argc > 1)
        _tprintf(TEXT("[LEITOR] Esperar pelo pipe '%s' (waitNamedPipe)\n"), PIPE_NAME);

    //espera que exista um named pipe para ler do mesmo
    //Alimenta mutex
    if (!WaitNamedPipe(PIPE_NAME, NPMWAIT_FOREVER)) {
        _tprintf(TEXT("[ERRO] Ligar ao pipe '%s'! (waitNamedPipe)\n"), PIPE_NAME);
        exit(-1);
    }

    _tprintf(TEXT("[LEITOR] Ligação ao pipe do escritor... (CreateFile)\n"));

    //ligamos ao named pipe que se existe nesta altura
    //é nome do named pipe, permissões (tem de ser iguais ao CreateNamedPipe do servidor), 3º parâmetro é aqui,
    //as security attributes, flags de criação OPEN_EXISTING, se o default é FILE_ATTRIBUTE_NORMAL e o 7º é o template é NULL
    hPipe = CreateFile(PIPE_NAME, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hPipe == INVALID_HANDLE_VALUE) {
        _tprintf(TEXT("[ERRO] Ligar ao pipe '%s'! (CreateFile)\n"), PIPE_NAME);
        exit(-1);
    }
}

```

main 1

```

_tprintf(TEXT("[LEITOR] Liguei-me...\n"));

while (1) {
    //le as mensagens enviadas pelo servidor
    ret = ReadFile(hPipe, buf, sizeof(buf), &n, NULL);

    //termina corretamente a string
    buf[n / sizeof(TCHAR)] = '\0';

    if (!ret || !n) {
        _tprintf(TEXT("[LEITOR] %d bytes... (ReadFile)\n"), ret, n);
        break;
    }

    _tprintf(TEXT("[LEITOR] Recebi %d bytes: '%s'... (ReadFile)\n"), n, buf);

    CloseHandle(hPipe);
    Sleep(200);
    return 0;
}

```

main 2

Interface Grafica Win32 GUI

- Na interface gráfica já usamos `WINAPI` em vez de `_tmain`
- Para ter o projeto compatível com `WINAPI` fazemos → `Linker` > `System` > `SUBSYSTEM\WINDOWS`
- `CreateSolidBrush(RGB(255,0,0))` → Estamos a criar uma nova brush, ou seja, podemos usar isto para pintar o fundo da janela
- **Precisamos de incluir o <windowsx.h> por causa das coordenadas da janela**
- O default no switch case da `função Callback` é bastante importante e tem de existir sempre porque diz que não foi efetuado nenhum processamento, apenas se segue o `default` do Windows
- **A escrita da janela deve estar sempre associada a eventos de refresco da janela**

- Se o meu código não está associado a eventos de refrescamento, se eu minimizar a janela e voltar abrir, todo o conteúdo da janela vai desaparecer

Neste exercício o objetivo era carregar N vezes no botão esquerdo do rato, pegar nesse N e mostrar na janela quando carregamos com o botão direito. Caso usemos a tecla do espaço o contador volta a 0 → **Atenção que isto é só exemplo, na prática ha aqui muitas coisas mal, por exemplo o facto de estarmos a pintar na tela num evento que nao é o mais correto pois devia de ser no evento WM_PAINT**

```

LRESULT CALLBACK TrataEventos(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    //handle para o device context
    HDC hdc;
    RECT rect; // representa um retangulo
    static TCHAR c = '0';
    int xPos, yPos;
    static int times = 0; // meti a var a static

    switch (msg)
    {
        // detetar o clique esquerdo do rato
        case WM_LBUTTONDOWN:
            times++;
            c = times + '0';

            break;

        case WM_RBUTTONDOWN:
            xPos = GET_X_LPARAM(lParam); // guarda a posição do rato
            yPos = GET_Y_LPARAM(lParam);

            // temos de ir buscar o handle para o device context
            hdc = GetDC(hwnd);

            // operações de configuração do retangulo
            GetClientRect(hwnd, &rect);
            rect.left = xPos; // indicamos um desvio da esquerda para a direita no x
            rect.top = yPos; // indicamos um desvio do topo da janela para o fundo da janela no y
            SetTextColor(hdc, RGB(255, 255, 255)); // cor do texto
            SetBkMode(hdc, TRANSPARENT); // quero que o fundo por trás do texto seja transparente. podemos usar o SetBkColor e temos uma cor concreta
            // querades de escrita no entanto de forma incorreta porque isto devia de estar em eventos de refrescamento
            DrawText(hdc, &c, 1, &rect, DT_NOCLIP | DT_SINGLELINE);

            // temos de fazer release do device context
            ReleaseDC(hwnd, hdc);
            break;
    }
}

```

← Função Callback

← Static porque não queremos que quando a função Callback volte a ser chamada esta variavel dê reset ao seu valor inicial. É uma maneira de guardarmos o estado da mesma

← Esta a contar quantas vezes carregamos no left mouse button e a passar esse numero para carater

← Aqui vamos desenhar na janela o tal numero que será um carater. Isto é má pratica porque nao devemos usar funções de pintar num case destes, devemos usar no evento WM_PAINT, no entanto isto é só para demonstração inicial

```

case WM_KEYDOWN:
    if (wParam == VK_SPACE) {
        times = 0;
        c = times + '0';
    }

    break;

// evento que o windows envia para a funcao de callback quando alguem carregou no botao de fecho da janela
case WM_CLOSE:
    // handle , texto da janela, titulo da janela, configurações da MessageBox(botoes e icons)
    if (MessageBox(hwnd, TEXT("Tem a certeza que quer sair?"), TEXT("Confirmação"), MB_YESNO | MB_ICONQUESTION) == IDYES) {
        // o utilizador disse que queria sair da aplicação
        DestroyWindow(hwnd);
    }
    break;

case WM_DESTROY: // Destruir a janela e terminar o programa
    PostQuitMessage(0);
    break;

default:
    // Neste exemplo, para qualquer outra mensagem (p.e. "minimizar", "maximizar", "restaurar")
    // não é efectuado nenhum processamento, apenas se segue o "default" do Windows
    return DefWindowProc(hwnd, msg, wParam, lParam);
    break; // break tecnicamente desnecessário por causa do return
}
return(0);
}

```

← Se carregarmos no espaço, aquela conta que tínhamos feito leva reset

← Isto é o default

Refrescamento de janelas

- As janelas em windows podem ser refrescadas a qualquer momento a pedido do sistema operativo. Ou seja, quando eu minimizo uma janela o sistema operativo tem de gerir as aplicações gráficas que estão visíveis, por isso ele vai limpar a janela toda, logo temos de garantir que pintamos a janela várias vezes para este problema não acontecer
- Não podemos pintar ou escrever coisas na janela na função de tratamento de eventos ou o problema não é resolvido**
- Sempre que há um pedido de refrescar a janela, vindo do SO, há um evento que vem para a nossa aplicação, evento este que se chama de `WM_PAINT` → **Toda a operação que escreva na janela tem de estar neste WM_PAINT para resolver o problema de refrescamento**

- O `BeginPaint` vai substituir o `GetDC` e o `EndPaint` vai substituir o `ReleaseDC`, estas funções (`BeginPaint` e `EndPaint`) são as que vão ser chamadas no `WM_PAINT`
- Uma forma indireta de forçar o refrescamento da janela é dizer ao sistema operativo que a janela está inválida e assim ele vai enviar o evento `WM_PAINT` → Para isto podemos usar o `InvalidateRect` no `switch case` que vê quando há uma alteração da posição, no caso da aula, no exercício 5 foi no `case WM_LBUTTONDOWN`
- **Toda a informação que precisamos para pintar a janela com o estado atual dela, tem de ser armazenada de alguma forma, por exemplo numa struct, e temos também de garantir que ela existe sempre que o `WM_PAINT` é chamado**

Bitmap

```
HBITMAP hBmp;
HDC bmpDC;
BITMAP bmp;
int xBitmap;
int yBitmap;
```

- **Estas variáveis vão representar o bitmap em si e têm de ser criadas de maneira global para conseguirem ser acedidas por todos os cases**
- Uma vez que elas vão ter de ser acedidas de maneira `assíncrona` pela main e pela função de tratamento de eventos, não há uma maneira de fugir ao uso de variáveis globais, o máximo que podemos fazer é criar uma struct, meter essas variáveis lá para dentro e passarmos a ter só 1 variável global

Se a informação não tiver de ser acedida por várias funções, existe a alternativa do "static". Mas, se a informação tiver de ser `accedida pela função de tratamento de eventos e por outras funções (por exemplo, pela main)`, aí terá mesmo de recorrer a variáveis globais (na quantidade mínima possível). A função de tratamento de eventos é invocada pelo sistema e não conseguimos passar dados nossos nos argumentos. Por esse motivo, para termos acesso à informação em eventos diferentes (tal como refere), temos de recorrer a estas soluções.

Convém sempre começarmos a programar o bitmap a seguir ao `CreateWindow` e antes do `ShowWindow`

Receita do Bitmap

- Carregamos o bitmap com a função `LoadImage` → devolve um handle genérico e depois temos de fazer cast para o handle do bitmap
- Vamos buscar a informação sobre o handle do bitmap usando a função `GetObject`
- Criamos uma cópia do device context e colocamos em memória usando o `CreateCompatibleDC`
- Aplicamos o bitmap ao device context usando o `SelectObject`
- Definimos as posições iniciais da imagem
- Por fim usamos a função `BitBlt` para fazer uma operação de bitwise e escrever a imagem → isto faz-se dentro do `WM_PAINT` que está dentro da função de tratamento de eventos

Exercicio 6 A

```
//Bitmap
// uma vez que temos de usar estas vars tanto na main como na funcao de tratamento de eventos
// tem de ser uma variavel de tipo de vars globais, dai estarem aqui
HBITMAP hBmp; // handle para o bitmap
HDC bmpDC; // hdc do bitmap
BITMAP bmp; // informacao sobre o bitmap
int xBitmap; // posicao onde o bitmap vai ser desenhado
int yBitmap;
```


Double Buffering

- Em vez de apagarmos a janela e voltarmos a pintar a janela, fazemos este processo numa cópia que está em memória, quando a cópia estiver pronta, fazemos um `BitBlt` e substituímos a janela que está visível pela cópia que está em memória
- Depois na parte do redimensionamento da janela, `já usando o double buffering`, temos de garantir que cópia em memória está atualizada porque a janela original pode ser aumentada ou diminuída → Para isso recriamos a cópia basicamente, ou seja, no `WM_SIZE` dizemos que o `memDC = NULL` para no `WM_PAINT` entrar sempre no `if` e voltar a mandar a janela para memória, neste caso já atualizada

```
//BITMAP
// uma vez que temos de usar estas vars tanto na main como na funcao de tratamento de eventos
// nao ha uma maneira de fugir ao uso de vars globais, dai estarem aqui
HBITMAP hBmp; // handle para o bitmap
HDC bmpDC; // hdc do bitmap
BITMAP bmp; // informacao sobre o bitmap
int xBitmap; // posicao onde o bitmap vai ser desenhado
int yBitmap;

int limDir; // limite direito
HWND hwndGlobal; // handle para a janela
HANDLE hMutex;

HDC memDC = NULL; // copia do device context que esta em memoria, tem de ser inicializado a null
HBITMAP hBitmapDB; // copia as caracteristicas da janela original para a janela que vai estar em memoria
```

Variáveis novas para a memória

Criação de vars globais novas para o bitmap local e da memória

```
// como a posição e a largura da janela e que a imagem se vá movendo
// como movo? Movimentacao (Presso Mouse)
int dir = 1; // 1 para a direita, 0 para a esquerda
int salto = 10; // quantidade de pixels que a imagem salta de cada vez

while (1) {
    // aguarda que o mutex esteja livre
    WaitForSingleObject(hMutex, INFINITE);

    // movimentacao
    xBitmap = xBitmap + (dir * salto);

    // pintura a imagem
    if (xBitmap >= 0) {
        xBitmap = 0;
        dir = 1;
    }
    // limite direito
    else if (xBitmap >= limDir) {
        xBitmap = limDir;
        dir = -1;
    }
    // liberta mutex
    ReleaseMutex(hMutex);

    // desenha na janela que a imagem mudou e fazemos entao de fazer o refresh da janela
    InvalidateRect(hwndGlobal, NULL, FALSE);
    Sleep(1);
}
return 0;
```

Aguarda que o mutex esteja livre

Movimenta no eixo do X

Forçamos o refresh da janela dizendo isso ao SO

Thread da movimentação do bitmap

```
// Limite direito e a largura da janela - largura da imagem
limDir = rect.right - bmp.bmWidth;
hwndGlobal = hwnd;

// cria mutex
hMutex = CreateMutex(NULL, FALSE, NULL);

// cria a thread de movimentacao
CreateThread(NULL, 0, MovimentaImagem, NULL, 0, NULL);
```

Obtenção do limite direito da janela
Criação do mutex global
Criação da thread da movimentação

Isto é feito a seguir ao processo de definição das posições iniciais do bitmap no main

```

LRESULT CALLBACK TrataEventos(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    //handle para o device context
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;

    switch (msg)
    {
        // evento que é disparado sempre que o sistema pede um refreshamento da janela
        case WM_PAINT:
            // Inicio da pintura da janela, que substitui o GetDC
            hdc = BeginPaint(hwnd, &ps);
            GetClientRect(hwnd, &rect);
            // se a copia estiver a NULL, significa que é a 1ª vez que estamos a passar no WM_PAINT e estamos a trabalhar com a copia em memoria
            if (memDC == NULL) {
                // cria copia em memoria
                memDC = CreateCompatibleDC(hdc);
                hBitmap00 = CreateCompatibleBitmap(hdc, rect.right, rect.bottom);
                // aplicamos na copia em memoria as configs que obtemos com o CreateCompatibleBitmap
                SelectObject(memDC, hBitmap00);
                DeleteObject(hBitmap00);

                // operações feitas na copia que é o memDC
                FillRect(memDC, &rect, CreateSolidBrush(RGB(125, 125, 125)));

                WaitForSingleObject(hMutex, INFINITE);
                // operacoes de escrita da imagem - BitBlt
                BitBlt(memDC, xBitmap, yBitmap, bmp.bmwidth, bmp.bmheight, bmpDC, 0, 0, SRCCOPY);
                ReleaseMutex(hMutex);

                // bitBlt da copia que esta em memoria para a janela principal - é a unica operação feita na janela principal
                BitBlt(hdc, 0, 0, rect.right, rect.bottom, memDC, 0, 0, SRCCOPY);

                // Encerra a pintura, que substitui o ReleaseDC
                EndPaint(hwnd, &ps);
                break;
            }
    }
}

```

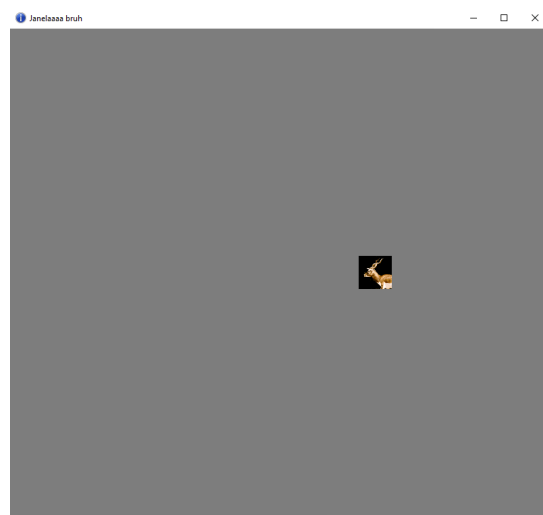
Lógica da WM_PAINT para trabalhar com uma janela em memória e depois substituir a original pela cópia

```

// redimensiona e calcula novamente o centro
case WM_SIZE:
    WaitForSingleObject(hMutex, INFINITE);
    xBitmap = (LOWORD(lParam) / 2) - (bmp.bmwidth / 2);
    yBitmap = (HIWORD(lParam) / 2) - (bmp.bmheight / 2);
    limDir = LOWORD(lParam) - bmp.bmwidth;
    memDC = NULL; // metemos novamente a NULL para que caso haja um resize na janela no WM_PAINT a janela em memoria é sempre atualizada com o tamanho novo
    ReleaseMutex(hMutex);
    break;
}

```

Lógica da WM_SIZE para que se a janela for resized, ao ir para o WM_PAINT o memDC vai a NULL e volta a entrar no primeiro IF



Resultado final

Aula P 28/05/2021 → Sobre resources e afins

- O resource.rc é o sitio onde os recursos ficam criados
- O ficheiro de resource.h faz a ponte entre o recurso e o nosso programa
- Cada elemento que criarmos dentro do recurso vai ter um id e esse ID vai corresponder a um nome que vamos definir no recurso
- Para darmos comportamentos a esses recursos vamos ter de utilizar o nome, ou seja, temos de distinguir os recursos
- Sempre que quisermos alterar o comportamento do recurso vamos incluir o `resource.h` e desta forma temos acesso aos IDs

- **Um acelerador permite definir um conjunto de atalhos de teclado**
- A tabela de strings é importante por exemplo para cada texto (printfs de erro e afins) esteja associada a uma tabela de strings com um ID → Isto é importante na internacionalização do programa, não é obrigatório no TP
- **Dialogbox modal significa que a partir do momento que a abro não consigo fazer nada na janela que está atrás**

- **Podemos resolver o problema da última aula em que só podíamos usar variáveis globais para serem partilhadas entre funções de tratamento de eventos**

- Guardamos tudo numa struct
- Criamos na main a struct
- Na config da janela, temos uma variável chamada `cbClsExtra` em que fazemos `sizeof`(da estrutura)
- A partir daqui tenho duas funções a `SetWindowLongPtr` e a `GetWindowLongPtr`
- `SetWindowLongPtr`, passo o handle da janela, um offset (`GWLP_USERDATA`) e um ponteiro para a estrutura que quero partilhar

```
dadosPartilhados.numOperacoes = 5; // Apenas para testar...
LONG_PTR x = SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR)&dadosPartilhados);
```

Exemplo de `SetWindowLongPtr`

- Depois na função de tratamento de eventos que quero usar aquela struct, basta chamar o `GetWindowLongPtr` e já tenho acesso ao ponteiro para aquela struct

```
dados* dadosPartilhados;

dadosPartilhados = (dados*)GetWindowLongPtr(hWnd, GWLP_USERDATA);
```

Exemplo do `GetWindowLongPtr`