Apontamentos Práticos

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 user 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 caraceteres 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 exexutar 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, podiamos 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.

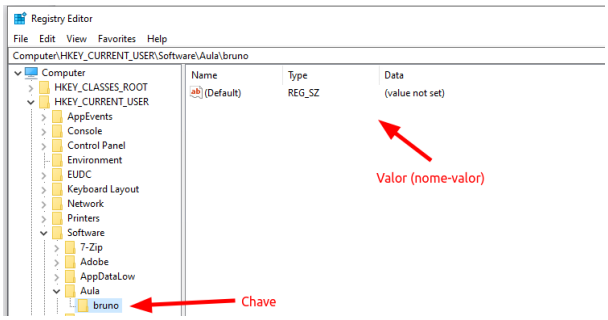
Os elementos principais constituintes do registry são as `chaves` e os `valores`

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 fisica relativamente à maquina, 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 variaveis 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\\SO2\\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 considereções acerca de `chars,wide chars e TCHAR`

DLLs

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

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

Ligação Explicita → `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 Implicita → `O Sistema Operativo é que faz o trabalho explicito 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 explicita

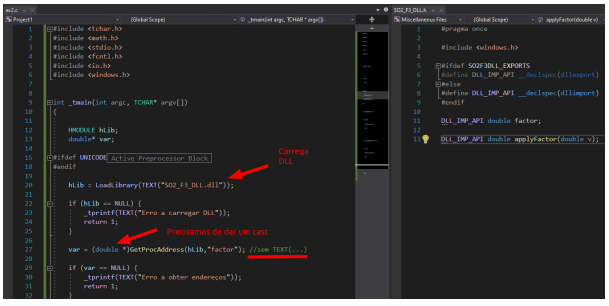
Na ligação explicita o ficheiro DLL pode estar onde quisermos, no entanto convem colocar ao pé do executavel para depois usar um caminho simples para a poder carregar

Como é que carregamos a biblioteca para memória?

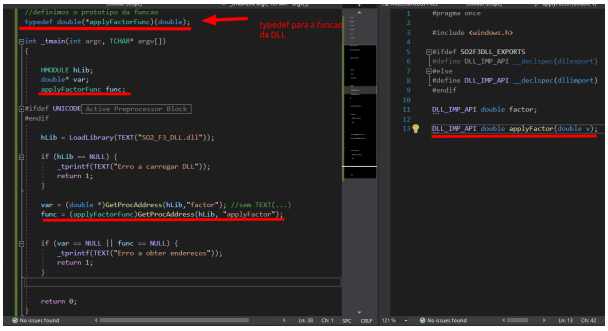
- Com a função `LoadLibrary` → recebe o caminho para a DLL

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

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



`Processo de carregamento de variavel da DLL de forma explicita`



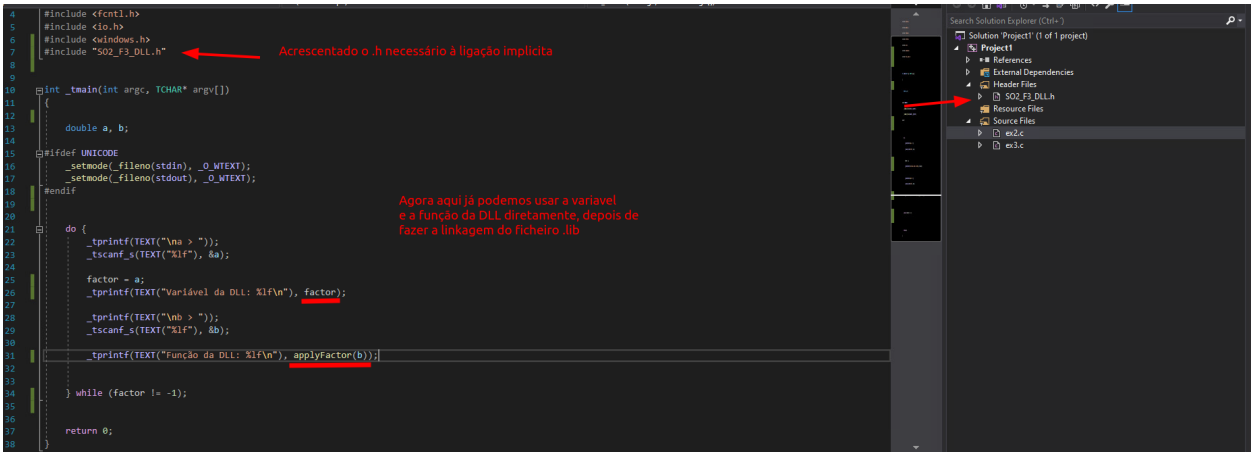
`Processo de carregamento de função da DLL de forma explicita`

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

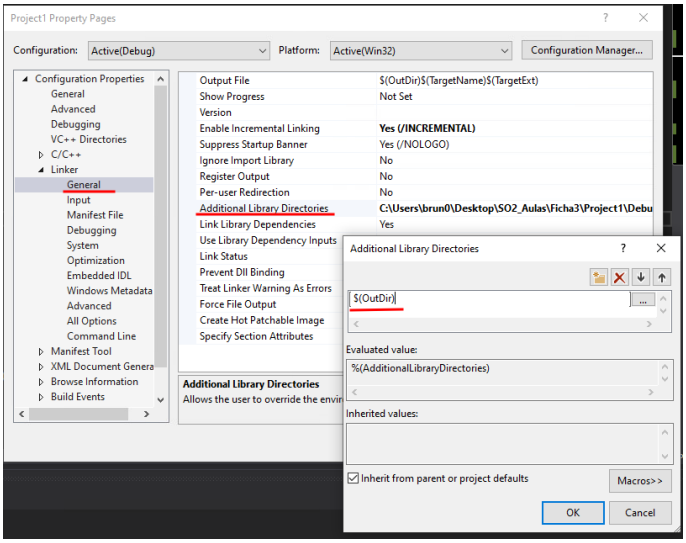
> 💡 Basta carregar uma vez a DLL para memória e basta carregar uma vez os recursos dessa DLL e no fim do programa fazer o `FreeLibrary`

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 implicita



`Uso da ligação implicita`

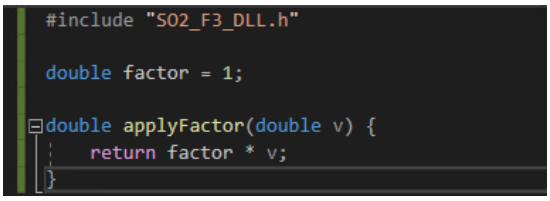


No TP caso usemos a ligação implicita, para partilharmos código, podemos nao fazer sempre as configurações do .lib usando isto

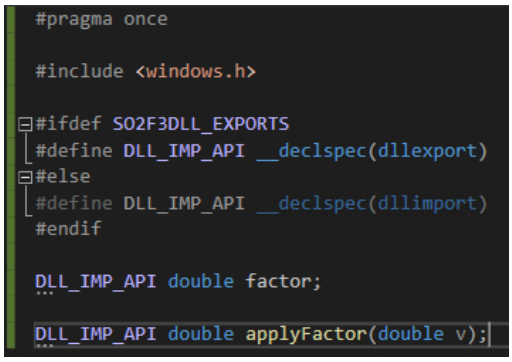
Para criar uma DLL

- Convem 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`

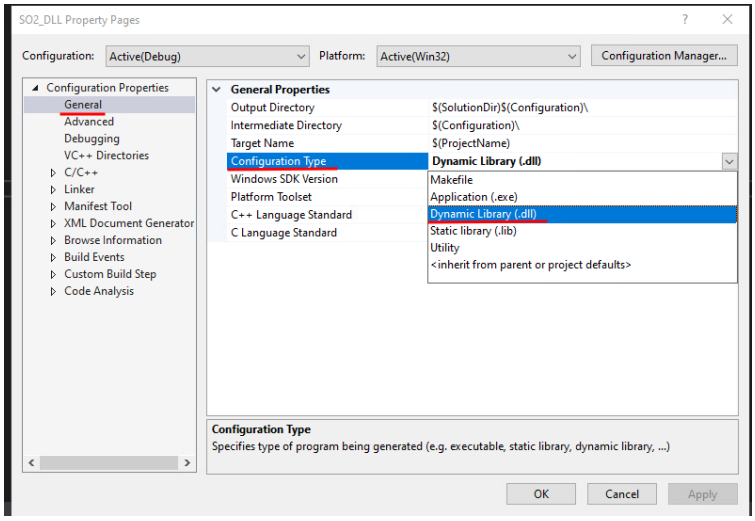
```
#pragma once
#include <windows.h>

#ifdef SO2DLL_EXPORTS
#define DLL_IMP_API __declspec(dllexport)
#else
#define DLL_IMP_API __declspec(dlimport)
#endif

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



Ficheiro .c da DLL



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 caracteristica que irão ser executadas em paralelo com a função `main`

As threads não são a unica forma de fazer parelelização de tarefas. Podemos fazer parelelização 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 → prótotipo das funções que irão ser lançadas como threads

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

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 indice que for desbloqueado for o 1 não há problema nenhum, no entanto se o indice que for desbloqueado for o 0 (na 1ª iteração) tenho um problema porque na segunda iteração não terei nenhum HANDLE no indice 0, logo a thread que está no indice 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 critica)` 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 acessiveis a mais do que um processo. Se forem cirados 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 nao 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 mutua, ou seja, usar um mutex por exemplo, ha um impacto enorme na performance do nosso código, uma vez que o código que temos deixou de fazer pararelismo 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 cout` 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 indesejavel, 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 periodos muito curtos

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

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

Depois tambem temos de passar o ponteiro, criado na funcao 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, exercicio 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 `semalhante 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, dai ser considerado por exclusão mutua

`Semaforo` → pode aceder 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 actualização 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 fica inteiramente a cargo do programador.
- No final os 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

```
HANDLE hFileMap;
TCHAR* fileViewMap;

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

    // deixamos de ter o Createfile e delegamos isso ao sistema operativo usando INVALID_HANDLE_VALUE, no 1º argumento
    // ou seja, o filemapping vai ser criado mas vai ser criado assente nos ficheiros de paginação do sistema operativo
    // temos de colocar a quantidade de bytes que queremos que o filemapping tenha -> NUM_CHARS * sizeof(TCHAR)
    // tem de ter uma nome tambem para ser partilhado entre processos
    hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, NUM_CHARS * sizeof(TCHAR), TEXT("SO2_MEMORIA_PARTILHADA"));
    if (hFileMap == NULL) {
        _tprintf(TEXT("Erro no CreateFileMapping\n"));
        return -1;
    }

    //mapeia-se vista desse objeto(hFileMap) numa zona de memória
    fileViewMap = (TCHAR*)MapViewOfFile(hFileMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

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

    FlushViewOfFile(fileViewMap, 0);
    //obrigatorio fazer unmap da vista
    UnmapViewOfFile(fileViewMap);
```

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

```c
// 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;
}
```

```c
// 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

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

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

        //convem limparmos 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

```c
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

```c
// antes de criar a memória partilhada vamos criar o semaforo
// o semaforo tem dois slots e os dois vão começar disponiveis
// precisa de um nome para ser partilhado entre processos
hSemaforo = CreateSemaphore(NULL, 2, 2, TEXT("SO2_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"));
```

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

Exercicio 2 d da ficha 6

Só usamos `WaitableTimers` em vez do Sleep

```c
//criacao do waitabel timer
dados.hTimer = CreateWaitableTimer(NULL, TRUE, TEXT("SO2_TIMER"));
if (dados.hTimer == NULL) {
    _tprintf(TEXT("Erro no CreateWaitableTimer\n"));
    UnmapViewOfFile(dados.fileViewMap);
    return -1;
}
```

Criação do `WaitableTimer` com `Reset Automático`

```c
// 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;
}
```

```c
// 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);

        //convem limparmos 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);

        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;
}
```

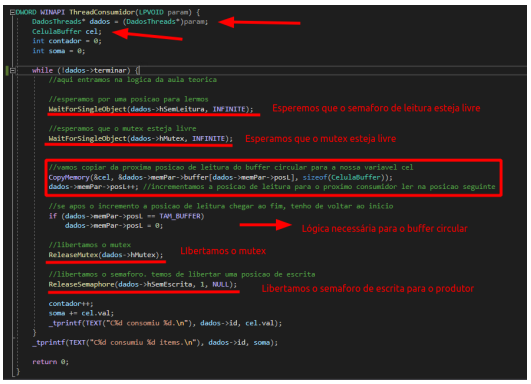
Em ambas as Threads foram feitos o `SetWaitableTimer` e o `WaitForSingleObject` para nao usar o `Sleep` uma vez que é má prática

Produtor e Consumidor

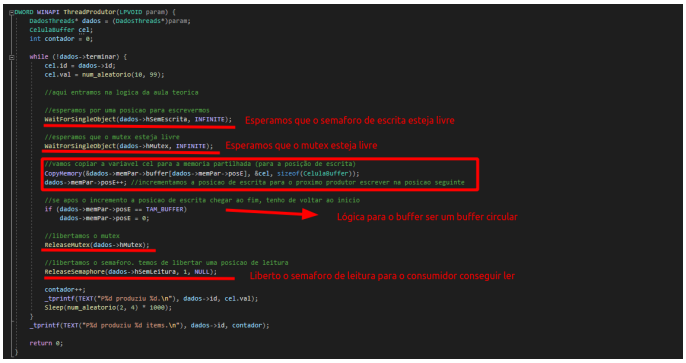
> 💡 Este exemplo foi usado `N produtores` e `M consumidores` dai ser preciso um mutex interno para ambos, caso fosse `1 consumidor` e `N produtores` só o produtor é que precisaria de um mutex interno

```c
//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 contrário
    int id;
}DadosThreads;
```

`Estrutruras usadas para o Consumidor e para o Produtor`

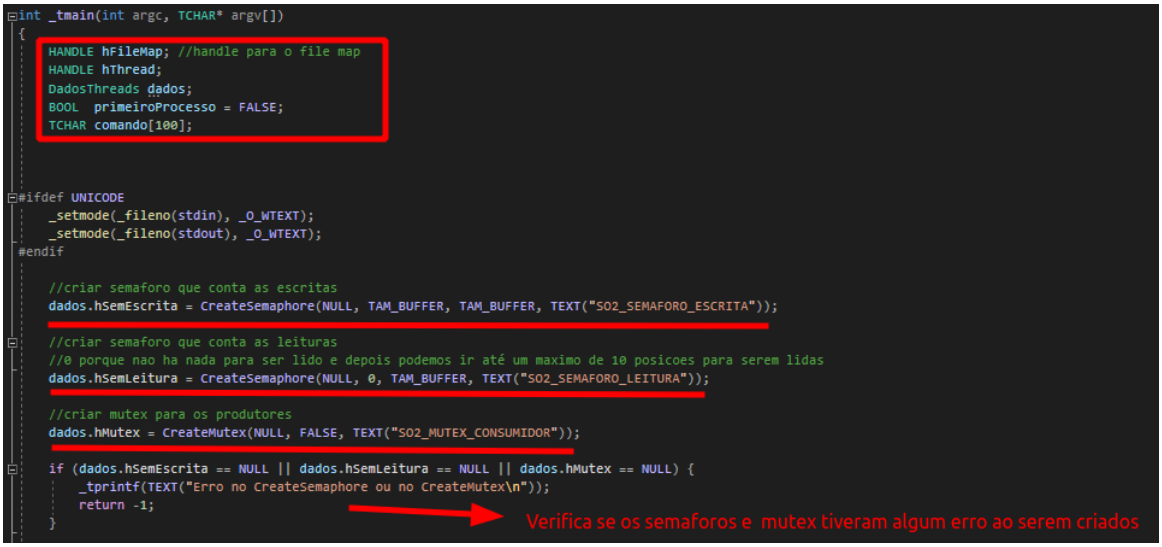


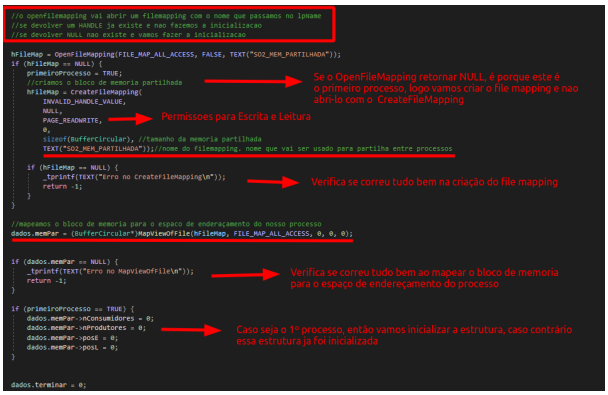
Thread do consumidor

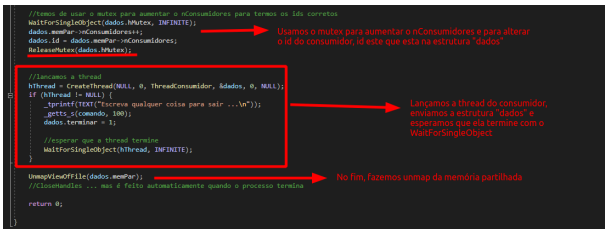


Thread do produtor

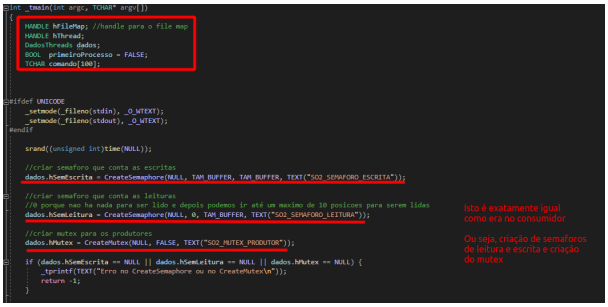
Código da main do Consumidor

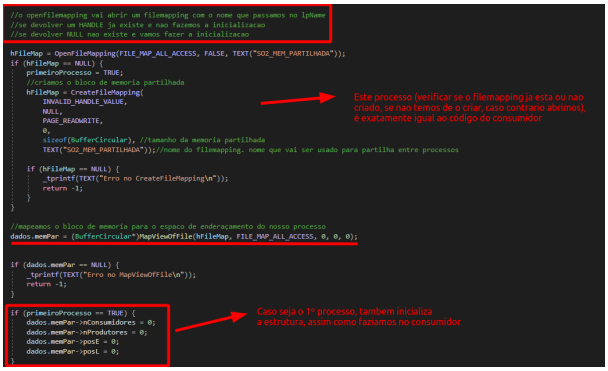






Código da main do Produtor





Lógica final

`Tendo N produtores e M consumidores`

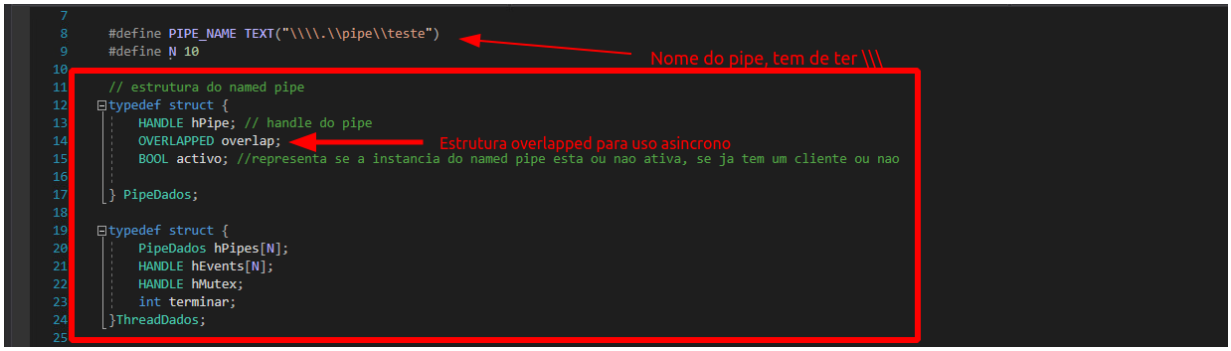
- **Produtor**
  - O produtor começa por esperar por uma posição no `semaforo de escrita` , ele `espera usando o WaitForSingleObject`
  - Depois `espera pelo mutex` tambem com o `WaitForSingleObject`
  - Quando o `mutex estiver livre, ele produz`
  - Quando acaba de produzir `liberta o mutex`
  - Depois 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` tambem 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



`Structs e pipe name`

`Thread para escrever mensagens`

```c
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 assincronas
        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);
        }
```

Crio a struct para a thread

Inicializo os campos da struct

Criação do named pipe

main 1

```c
    // criar evento que vai ser associado à esturtura overlaped
    // os eventos aqui tem de ter sempre reset manual e nao 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].activo = 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;
}
```

Crio o evento que vai ser associado à estrutura overlaped

Preencho o resto da estrutura

Garanto que a estrutura overlap esta limpa e coloco na estrutura da thread

Crio a thread para ler mensagens

main 2

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

    // se é um indice 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].activo = 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

```c
// ...
#define PIPE_NAME TEXT("\\\\.\\pipe\\teste")

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

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

    _tprintf(TEXT("[LEITOR] Esperar pelo pipe '%s' (WaitNamedPipe)\n"),PIPE_NAME);

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

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

    //ligamo-nos ao named pipe que ja existe nesta altura
    //1º nome do named pipe, 2ºpermissoes (têm de ser iguais ao CreateNamedPipe do servidor), 3ºshared mode 0 aqui,
    //4º security atributes, 5ºflags de criação OPEN_EXISTING, 6º 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 == NULL) {
        _tprintf(TEXT("[ERRO] Ligar ao pipe '%s'! (CreateFile)\n"), PIPE_NAME);
        exit(-1);
    }
```

Nome do named pipe

main 1

```c
    _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 %d... (ReadFile)\n"), ret, n);
            break;
        }

        _tprintf(TEXT("[LEITOR] Recebi %d bytes: '%s'... (ReadFile)\n"), n, buf);
    }
    CloseHandle(hPipe);
    Sleep(200);

    return 0;
```

Importante fechar este handle do named pipe

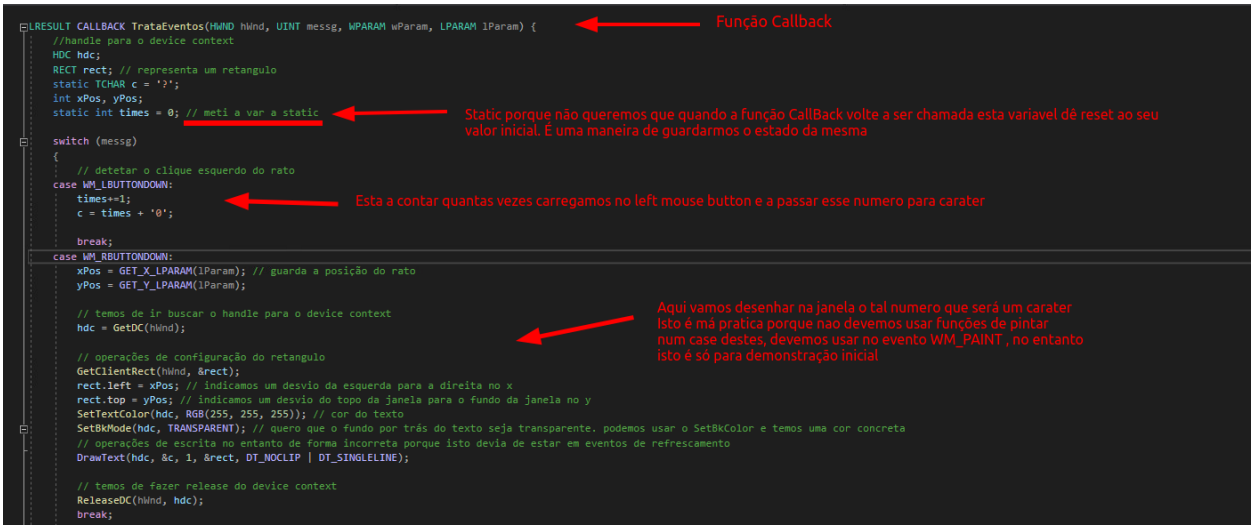
main 2

Interface Grafica Win32 GUI

- Na interface gráfica ja usamos `WINAPI` em vez de `_tmain`
- Para ter o projeto compativel com `WINAPI` fazemos → `Linker` > `System` > `SUBSYSTEM\WINDOWS`
- CreateSolidBrush(RBG(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 refrescamento da janela**

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

Neste exercicio 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



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 visiveis, 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 exercicio 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 tambem de garantir que ela existe sempre que o WM_PAINT é chamado**

Bitmap

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

- **Estas variaveis 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 `assincrona` 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 acedida 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
// 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; // informação sobre o bitmap
int xBitmap; // posicao onde o bitmap vai ser desenhado
int yBitmap;
```

Criação das vars referentes ao bitmap de maneira global

```
hWnd = CreateWindow(
    szProgName,          // Nome da janela (programa) definido acima
    TEXT("Janelaaaa bruh"),// Texto que figura na barra do título
    WS_OVERLAPPEDWINDOW,  // Estilo da janela (WS_OVERLAPPED= normal)
    CW_USEDEFAULT,       // Posição x pixels (default=à direita da última)
    CW_USEDEFAULT,       // Posição y pixels (default=abaixo da última)
    CW_USEDEFAULT,       // Largura da janela (em pixeis)
    CW_USEDEFAULT,       // Altura da janela (em pixels)
    (HWND)HWND_DESKTOP, // handle da janela pai (se se criar uma a partir de
                        // outra) ou HWND_DESKTOP se a janela for a primeira,
                        // criada a partir do "desktop"
    (HMENU)NULL,         // handle do menu da janela (se tiver menu)
    (HINSTANCE)hInst,    // handle da instância do programa actual ("hInst" é
                        // passado num dos parâmetros de WinMain()
    0);                 // Não há parâmetros adicionais para a janela

HDC hdc; // representa a propria janela
RECT rect;

// carregar o bitmap
hbmp = (HBITMAP)LoadImage(NULL, TEXT("ola.bmp"), IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
GetObject(hbmp, sizeof(bmp), &bmp); // vai buscar info sobre o handle do bitmap

hdc = GetDC(hwnd);

// criamos copia do device context e colocar em memoria
bmpDC = CreateCompatibleDC(hdc);

// aplicamos o bitmap ao device context
SelectObject(bmpDC, hbmp);

ReleaseDC(hwnd, hdc);
```

Carregamos o bitmap com a função LoadImage

Info sobre o handle do bitmap usando o GetObject

Criamos uma cópia do device context e colocamos em memoria usando o CreateCompatibleDC

```
// EXEMPLO
// 800 px de largura, imagem 400px de largura
// ponto central da janela 400 px(800/2)
// imagem centrada, começar no 300px e acabar no 420 px
// (800/2) - (40/2) = 400 - 20 = 380px

// definir as posições inicias da imagem
GetClientRect(hWnd, &rect);
xBitmap = (rect.right / 2) - (bmp.bmWidth / 2);
yBitmap = (rect.bottom / 2) - (bmp.bmHeight / 2);
```

Definimos as posições inicias do bitmap

Definimos as posições iniciais do bitmap

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


    switch (mesig)
    {
    // o evento que é disparado sempre que o sistema pede um refrescamento da janela
    case WM_PAINT:
        // Início da pintura da janela, que substitui o GetDC
        hdc = BeginPaint(hwnd, &ps);

        // operações de escrita da imagem - BitBlt
        BitBlt(hdc, xBitmap, yBitmap, bmp.bmWidth, bmp.bmHeight, bmpDC, 0, 0, SRCCOPY);


        // Encerro a pintura, que substitui o ReleaseDC
        EndPaint(hwnd, &ps);
        break;
    // redimensiona e calcula novamente eo centro
    case WM_SIZE:
        xBitmap = (LOWORD(lParam) / 2) - (bmp.bmWidth / 2);
        yBitmap = (HIWORD(lParam) / 2) - (bmp.bmHeight / 2);
        break;

    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(exit Status)"
        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, mesig, wParam, lParam));
    }
```

Agora usamos o BeginPaint para substituir o GetDC

Fazemos a operação de BitBlt para escrever a imagem

Por fim, usamos o EndPaint para substituir o ReleaseDC

Sempre que a janela é redimensionada ,calculamos a posição da imagem

Operação de BitBlt feito na função de tratamento de eventos dentro do WM_PAINT

Resultado final

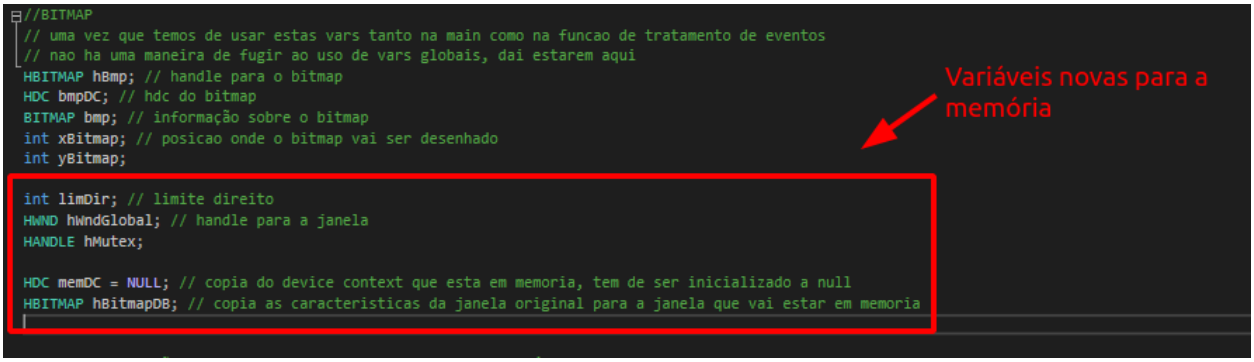
Exercicio 6 B → Deslocar o bitmap

- Neste exercicio o objetivo é movimentar a imagem na horizontal não havendo na mesma o problema da cintilação e ao redimensionar a janela a mesma continua a movimentar-se na horizontal de maneira correta
- **Um dos problemas era o problema da concorrência em que estamos aceder às variáveis** `xBitmap` **e** `yBitmap` → Conseguimos resolver isto com um mutex na thread da movimentação, no `BitBlt` e no `WM_SIZE` uma vez que é aqui que mexemos nas tais variaveis `xBitmap` e `yBitmap`

> 💡 Como ha o problema da cintilação, uma maneira de o resolver e a mais correta é, em vez de estarmos sempre a pintar a janela com valores, copiamos o objeto para memória, manipulamos o mesmo e depois trocamos o objeto que esta ativo pelo objeto alterado que está em memória → `Double Buffering`

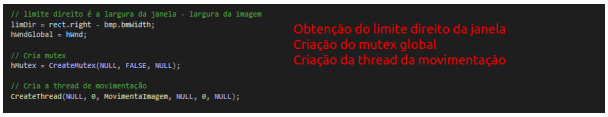
Double Buffering

- 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 substituimos a janela que está visível pela cópia que está em memória
- Depois na parte do redimencionamento da janela, `já usando o double buffering`, temos de garantir que cópia em memória está atualizada porque a janela original pode ser aumentada ou diminuida → 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

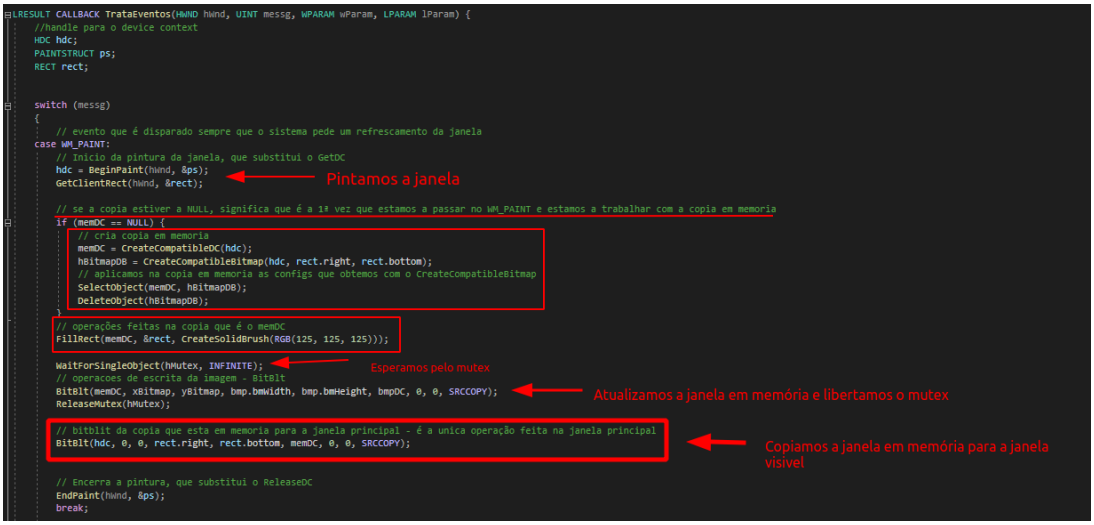


`Criação de vars globais novas para o bitmap local e da memória`

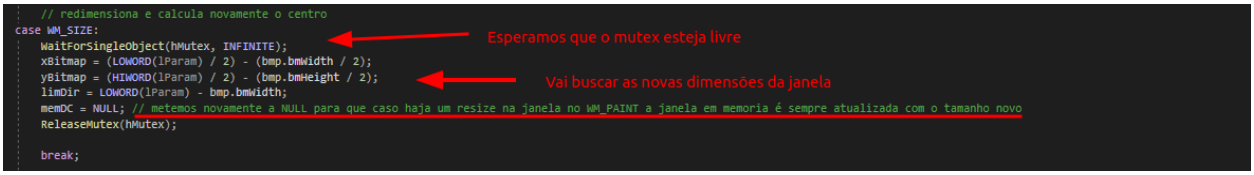
`Thread da movimentação do bitmap`



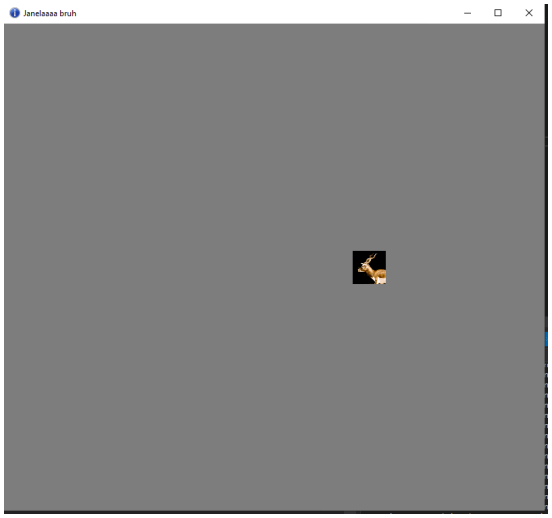
`Isto é feito a seguir ao processo de definição das posições iniciais do bitmap no main`



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



`Lógica do 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  aum 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, nao é obrigatorio no TP
- **Dialogbox modal significa que a partir do momento que a abro nao consigo fazer nada na janela que está atrás**

- **Podemos resolver o problema da ultima aula em que so podiamos usar variaveis globais para serem partilhadas entre funções de tratamento de eventos**
  - `Guardarmos tudo numa struct`
  - `Criamos na main a struct`
  - `Na config da janela, temos uma variavel 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 estruta 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 ja tenho acesso ao ponteiro para aquela struct

```
dados* dadosPartilhados;

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

`Exemplo do GetWindowLongPtr`