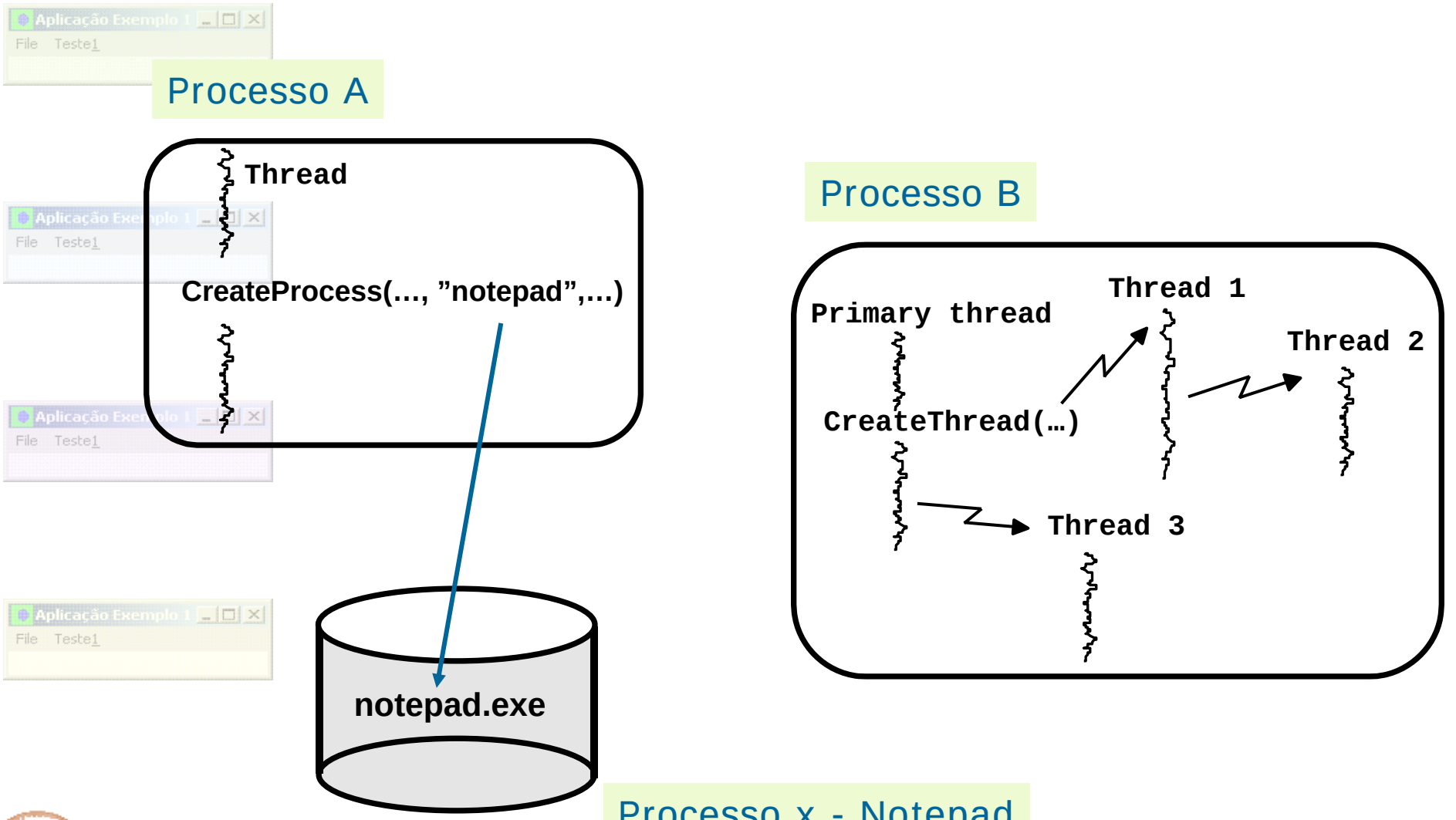
# Processos e Tarefas na Win32

# Processos e Tarefas na Win32

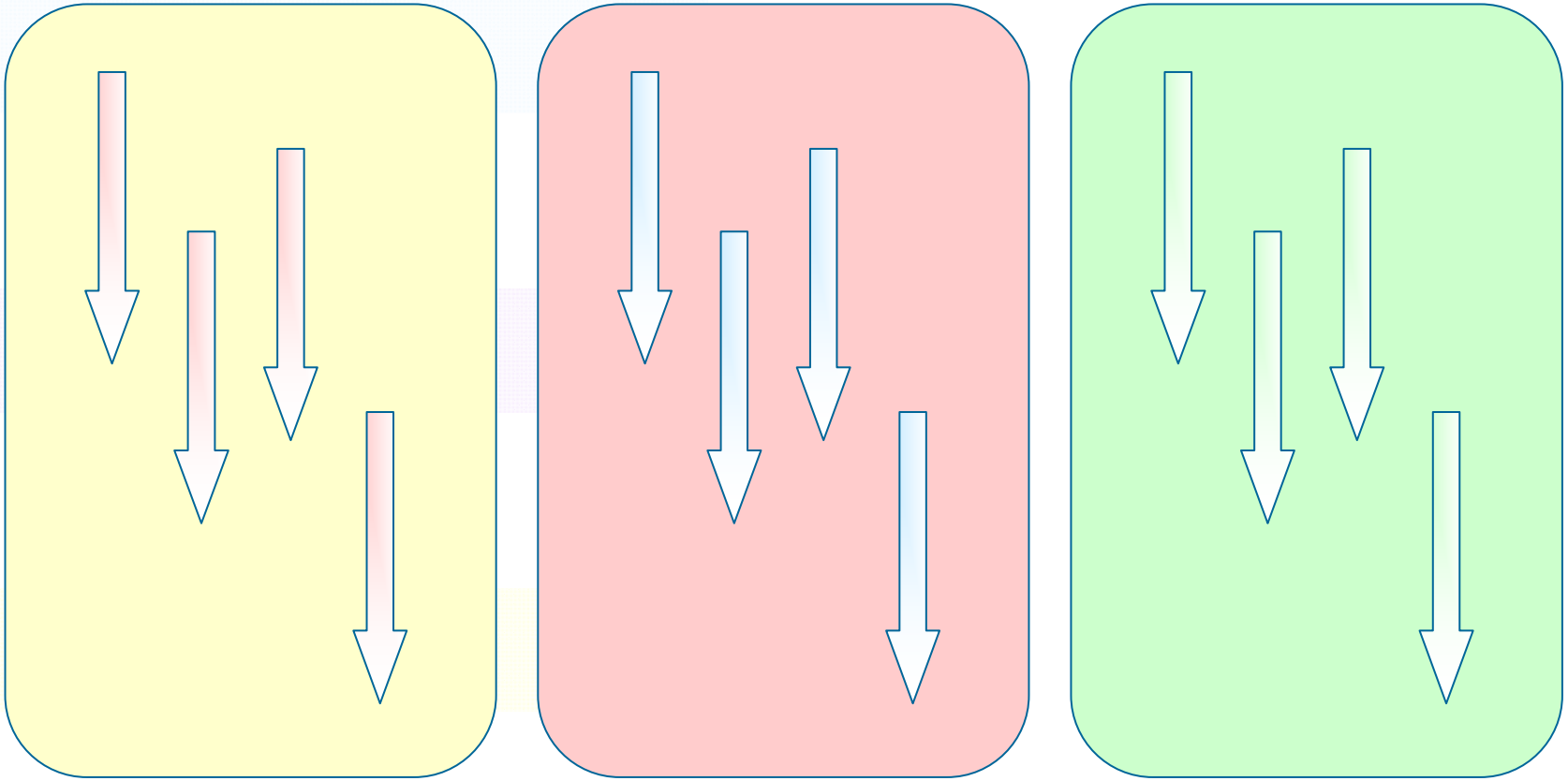

- Um processo é uma instância de uma aplicação em execução e é composto por duas partes:
  - Um objecto do *kernel* que o sistema operativo usa para gerir o processo e onde é guardada informação estatística acerca do processo ao longo do seu ciclo de vida;
  - Um espaço de endereçamento que contém o código executável, os dados, *stack* e espaço para alocação dinâmica de memória.


- Uma *thread* é um fio de execução dentro de um processo

# Processos e Tarefas



# Win32 - Processos



# Criar Processo

```
BOOL CreateProcess(
    LPCTSTR                 lpApplicationName,    // nome
    LPTSTR                  lpCommandLine,        // command line
    LPSECURITY_ATTRIBUTES   lpProcessAttributes,  // at. Segurança prcss
    LPSECURITY_ATTRIBUTES   lpThreadAttributes,   // at. Segurança thrd
    BOOL                    bInheritHandles,      // flag herança handles
    DWORD                   dwCreationFlags,      //
    LPVOID                  lpEnvironment,        // variáveis de ambiente
    LPCTSTR                 lpCurrentDirectory    // directoria corrente
    LPSTARTUPINFO           lpStartupInfo,        // informações de startup
    LPPROCESS_INFORMATION   lpProcessInformation // inf. devolvida
);
```

Sumário :

CreateProcess(..., “fich.exe 100”, ...)

executável argumento(s)

Segue-se a descrição dos principais campos

# Application name & Command line

***Application name*** ***(App)*****:** *String* com o nome [e localização] do ficheiro (a aplicação)

***Command line*** ***(Com)*****:** *String* com a linha de comandos que será passada ao executável.

| *lpApplicationName* | *pCommandLine* | Main(argc, argv) |
|---|---|---|
| App.exe* | "argA argB" | → "argA"<br>→ "argB" |
| NULL | "App.exe argA argB" ** | → "App.exe"<br>→ "argA"<br>→ "argB" |
| App.exe* | "App.exe argA argB" | → "App.exe"<br>→ "argA"<br>→ "argB" |

```
* Exact location or current defaults :
     .exe
     current dir
```

```
Dir. onde o executável "parent" foi lido
Dir. corrente do "parent"
Dir. De system do Windows
Dir. do Windows                          **
Dirs. da Path
```

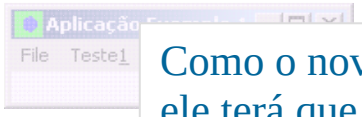
# Herança

## lpProcessAttributes & lpThreadAttributes

Descritores de segurança e atributos de herança para os dois novos `handlers`

## bInheritHandles

*Flag* que condiciona a possibilidade de herança, se for `true` os objectos herdáveis são herdados pelo processo filho




Como o novo processo não contém informação acerca do estado do processo pai, ele terá que receber informação sobre quais os `handles` que herdou através da linha de comando, ou do bloco das variáveis de ambiente ou de uma outra qualquer forma de IPC (InterProcess Communication - comunicação entre processo)


```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD  nLength;
    LPVOID lpSecurityDescriptor;
    BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

# Campo *Creation Flags*

| | |
|---|---|
| **CREATE_NEW_CONSOLE** | O processo executa-se numa nova consola |
| **DETACHED_PROCESS** | O processo é criado sem consola. O processo pode usar a função *AllocConsole* para criar uma nova consola. |
| **CREATE_SUSPENDED** | A *Thread* primária do novo processo inicia-se no estado Suspenso |
| **PRIORITY FLAG** | Prioridade do processo ( ver página seguinte ), *nada* para receber a prioridade do processo criador |

| | |
|---|---|
| **CREATE_DEFAULT_ERROR_MODE** | O processo não herda o modo de controlo de erros do processo pai (função SetErrorMode) |
| **CREATE_NEW_PROCESS_GROUP** | O novo processo passa a ser a raiz de um novo grupo |

| | |
|---|---|
| **CREATE_SEPARATE_WOW_VDM** | Processo a 16 bits que se executa numa *Virtual Dos Machine* (VDM) própria |
| **CREATE_SHARED_WOW_VDM** | (WINNT) a aplicação executa-se numa VDM partilhada |
| **CREATE_UNICODE_ENVIRONMENT** | Caracteres Unicode nas variáveis de ambiente |

Também se pode especificar a prioridade do processo, ver acetato seguinte

# lpStartupInfo & lpProcessInformation

## lpStartupInfo

Especifica dados (ex. a aparência da janela) para o novo processo

```
typedef struct _STARTUPINFO {
  DWORD   cb; // size da estrutura
  LPTSTR  lpReserved;
  LPTSTR  lpDesktop;
  LPTSTR  lpTitle; // window title¹
  DWORD   dwX;     // Window pos
  DWORD   dwY;     // ...
  DWORD   dwXSize; // window size
  DWORD   dwYSize; // ...
  DWORD   dwXCountChars; // ncol¹
  DWORD   dwYCountChars; // nlinhas¹
  DWORD   dwFillAttribute;
  DWORD   dwFlags;      // ...
  WORD    wShowWindow;  // ...
  WORD    cbReserved2;
  LPBYTE  lpReserved2;
  HANDLE  hStdInput;  // redirects
  HANDLE  hStdOutput; // ...
  HANDLE  hStdError;  // ...
} STARTUPINFO, *LPSTARTUPINFO;
```

IN

CreateProcess

OUT

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

* Unique system ID
¹ processos de consola

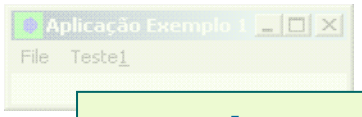
Os handles devem ser fechados quando não forem necessários

## lpProcessInformation

Devolve informação acerca do novo processo e thread

Nota: Os campos não utilizados devem ir a zero (ZeroMemory)

# Terminar processo



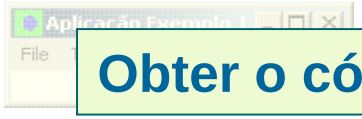
## Terminar o próprio processo

```
VOID ExitProcess( UINT uExitCode );          // qualquer thread
```

Se deixarmos a thread primária retornar do Main, ela cumpre o EXIT que se encontra no código da rotina que faz a chamada ao Main, e termina o processo.

## Terminar um outro processo

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



## Obter o código de finalização do processo

```
BOOL GetExitCodeProcess( HANDLE hProcess, LPDWORD lpExitCode );
```

Devolve STILL_ACTIVE se o processo ainda estiver activo

# Esperar pelo término de um processo

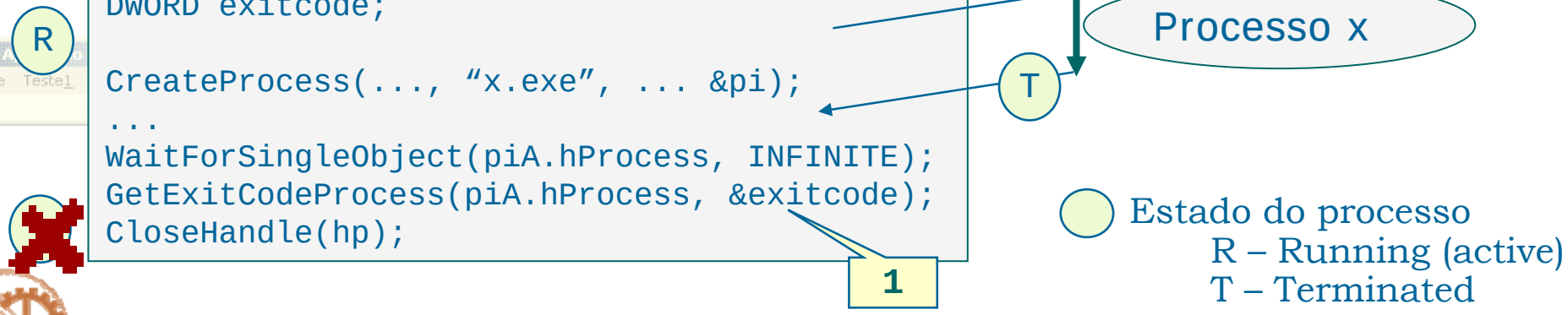
## Esperar o término de um processo

```
CreateProcess(NULL, nomeExe, NULL, NULL, FALSE, 0, NULL, NULL, &si, &piA) )
. . .
WaitForSingleObject(piA.hProcess, INFINITE); // esperar que o processo termine
```

A função **WaitForSingleObject** fica bloqueada até que o *handle* do processo fique sinalizado, o que **ocorrerá quando o processo terminar**.
Devolve WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_FAILED

## Criar um processo e sincronizar com a sua finalização

```
PROCESS_INFORMATION pi;
DWORD exitcode;

CreateProcess(..., "x.exe", ... &pi);
...
WaitForSingleObject(piA.hProcess, INFINITE);
GetExitCodeProcess(piA.hProcess, &exitcode);
CloseHandle(hp);
```



# Consultar os tempos de execução do Processo


**Consultar os tempos associados ao processo**

```
BOOL GetProcessTimes(
  HANDLE hProcess,              // handle to process
  LPFILETIME lpCreationTime, // process creation time, UTC time
  LPFILETIME lpExitTime,      // process exit time,     UTC time
  LPFILETIME lpKernelTime,   // process kernel-mode time
  LPFILETIME lpUserTime );   // process user-mode time
```

```
CreationTime & ExitTime  data em tempo UTC
KernelTime & UserTime    quantidade de tempo
```


# File Times

```
typedef struct _FILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME, *PFILETIME;
```

N * 100 ηSecs UTC ref

```
BOOL FileTimeToLocalFileTime(
  CONST FILETIME *lpFileTime,
  LPFILETIME lpLocalFileTime );
```

N * 100 ηSecs Local ref

```
BOOL FileTimeToSystemTime (
  CONST FILETIME *lpFileTime,
  LPSYSTEMTIME lpSystemTime );
```

Dia, mês, ano,
horas, min, sec, milisec

A estrutura **FILETIME** é um valor a 64 bits (32+32), que contém o número de periodos de 100 nano segundos. Para datas, este nº é ref. a 1 de Janeiro de 1601 até ao tempo corrente UTC.

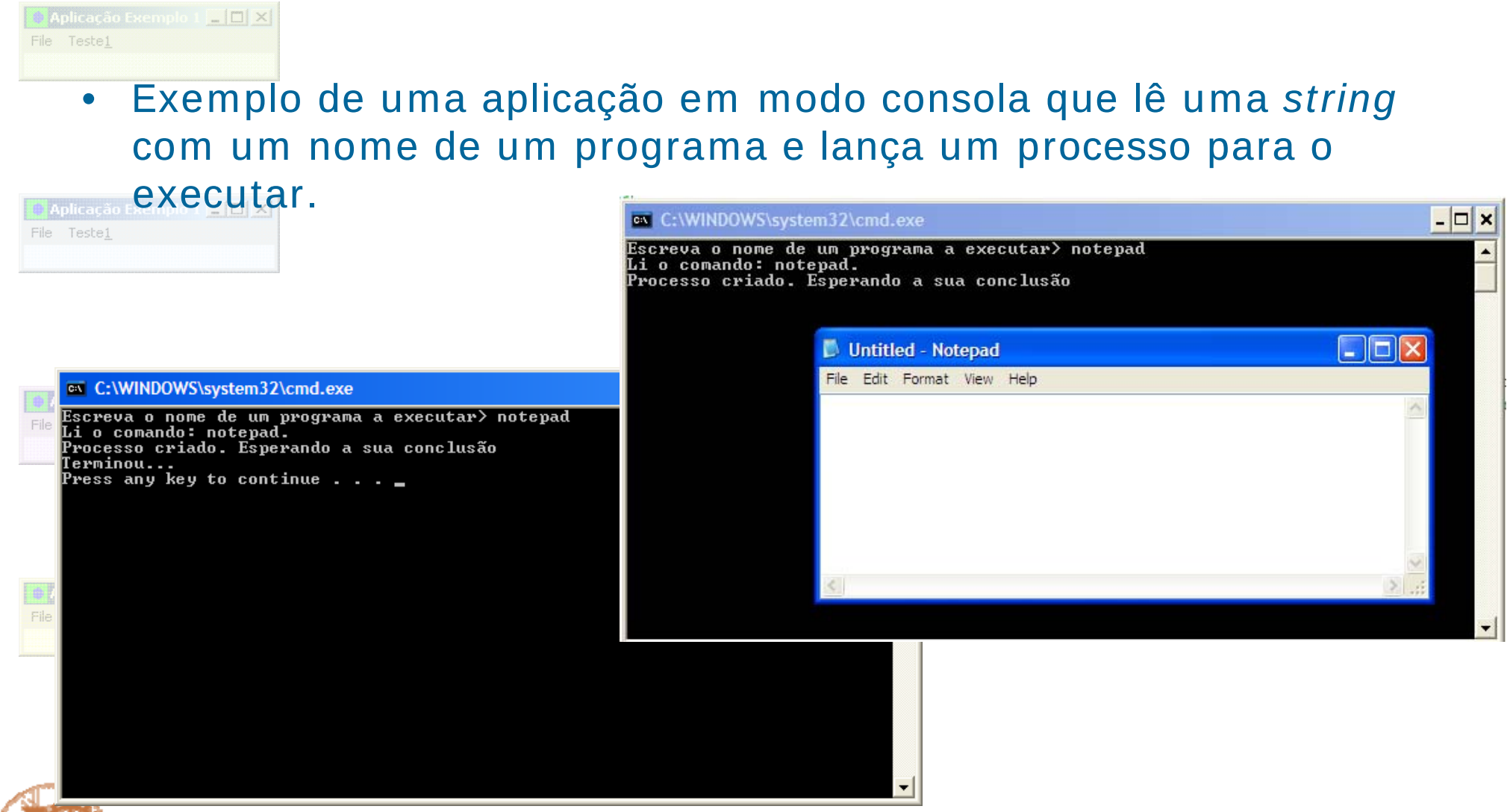
Esta função converte UTC FILETIME num tempo FILETIME mas com ajuste para a hora local

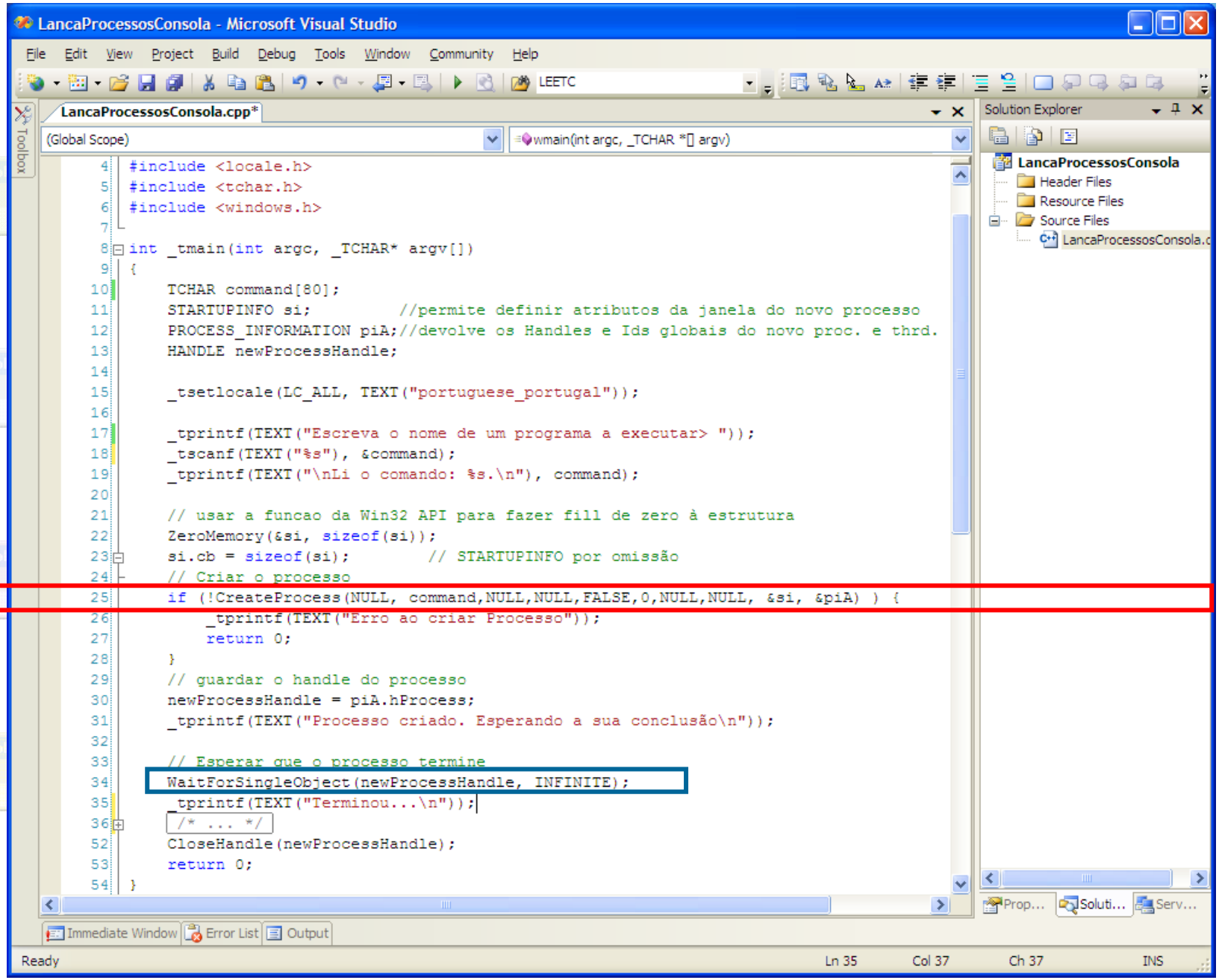
```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Nós, Portugueses, estamos síncronos com UTC pois estamos no fuso GMT + 0

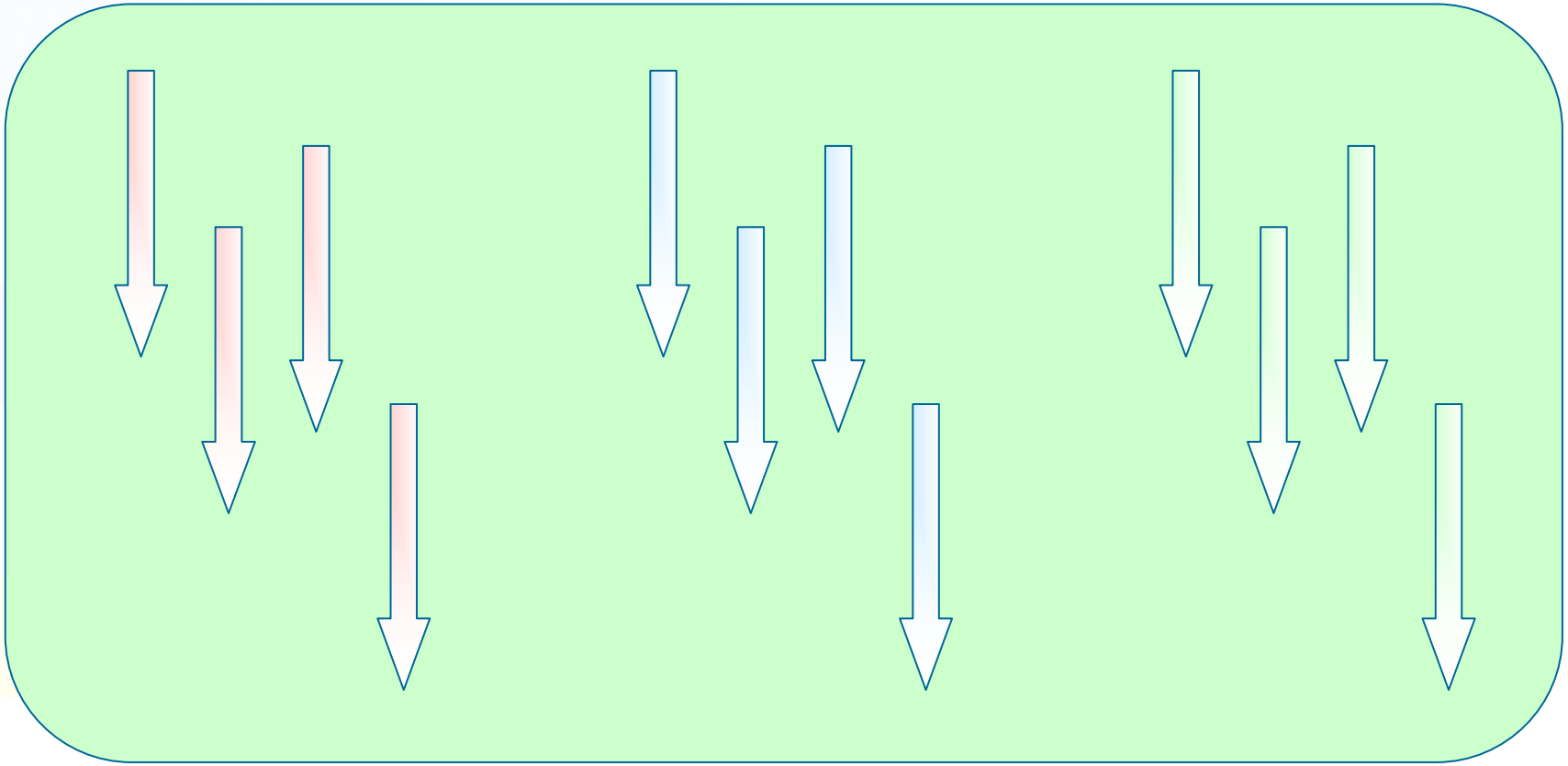
# Exemplo de criação de processos

- Exemplo de uma aplicação em modo consola que lê uma *string* com um nome de um programa e lança um processo para o executar.

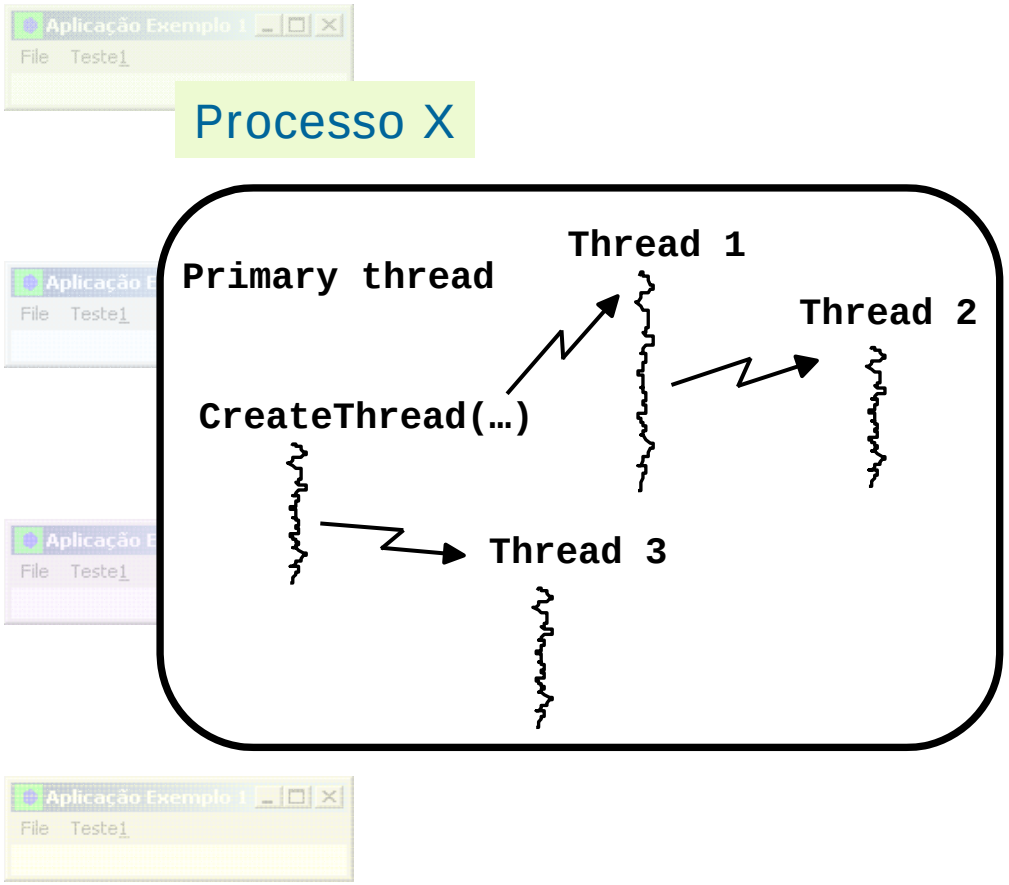



LancaProcessosConsola - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
LancaProcessosConsola.cpp*
(Global Scope)
wmain(int argc, _TCHAR *[] argv)
Solution Explorer
LancaProcessosConsola
Header Files
Resource Files
Source Files
#include <locale.h>
#include <tchar.h>
#include <windows.h>
int _tmain(int argc, _TCHAR* argv[])
{
TCHAR command[80];
STARTUPINFO si; //permite definir atributos da janela do novo processo
PROCESS_INFORMATION piA;//devolve os Handles e Ids globais do novo proc. e thrd.
HANDLE newProcessHandle;
_tsetlocale(LC_ALL, TEXT("portuguese_portugal"));
_tprintf(TEXT("Escreva o nome de um programa a executar> "));
_tscanf(TEXT("%s"), &command);
_tprintf(TEXT("\nLi o comando: %s.\n"), command);
// usar a funcao da Win32 API para fazer fill de zero à estrutura
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si); // STARTUPINFO por omissão
// Criar o processo
if (!CreateProcess(NULL, command,NULL,NULL,FALSE,0,NULL,NULL, &si, &piA) ) {
_tprintf(TEXT("Erro ao criar Processo"));
return 0;
}
// guardar o handle do processo
newProcessHandle = piA.hProcess;
_tprintf(TEXT("Processo criado. Esperando a sua conclusão\n"));
// Esperar que o processo termine
WaitForSingleObject(newProcessHandle, INFINITE);
_tprintf(TEXT("Terminou...\n"));
/* ... */
CloseHandle(newProcessHandle);
return 0;
}
Immediate Window
Error List
Output
Ready
Ln 35
Col 37
Ch 37
INS

# Win32 - Tarefas



# Processo e Tarefas




O fio de execução que inicia o main é designado de thread primária ou principal

# Exemplos de uso de várias tarefas num programa


- ***Foreground and background***
  Uma *thread* para *user input*, e outra para processamento
- ***Asynchronous processing***
  Por exemplo, uma *thread* para fazer *Backup* periódico
- ***Speed execution***
  Execução e I/O de dados em paralelo
- ***Modular program structure***
  Facilidade de desenho de aplicações que tenham que lidar com múltiplos canais de *input* e *output*

# Exemplos de Aplicações *multithread*


**Aplicações e suas *Threads* :**

- **Word** (*user input, repagination, print*)
- **Excel** (*user input, calculation, print*)
- **FileCopy** (*Data copy, cancel*)
- **Base de dados** (um pedido ⇔ uma thread)

Podemos observar o nº corrente de *threads* das várias aplicações no *task Manager*, se adicionarmos a coluna de "*Threads*"

# Criar tarefas


```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES   lpThreadAttributes, // Herança e seg.
  DWORD                   dwStackSize,        // 0 valor por omissão
  LPTHREAD_START_ROUTINE  lpStartAddress,     // routina da thread
  LPVOID                  lpParameter,        // parâmetro
  DWORD                   dwCreationFlags,    // como CREATE_SUSPENDED
  LPDWORD                 lpThreadId          // devolve thread ID
);
```


# Terminar tarefas


## Terminar uma Thread na Win32

```
VOID ExitThread( UINT  ExitCode )
```

Também é chamada implicitamente, quando se termina a função de arranque

## Terminar uma outra tarefa

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode );
```

## Obter o Exit Code de uma Thread que terminou

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpdwExitCode);
```

Devolve : STILL_ACTIVE se a Thread ainda não terminou

# Esperar pelo término de uma tarefa

## Esperar o término de uma thread

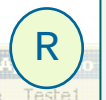
```
HANDLE ht = Createthread( tf, 0);
. . .
WaitForSingleObject(ht, INFINITE); // esperar que a thread termine
```

A função **WaitForSingleObject** fica bloqueada até que o *handle* da thread fique sinalizado, o que **ocorrerá quando a thread terminar**.
Devolve WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_FAILED

## Criar uma thread e sincronizar com a sua finalização

```
HANDLE ht; DWORD exitcode;

ht = CreateThread(..., func, ...);
...
WaitForSingleObject(ht, INFINITE);
GetExitCodeThread(ht, &exitcode);
CloseHandle(ht);
```

```
void func(){
  ...
  return 1;
}
```





1

Estado da thread
R – Running (active)
T – Terminated

# Suspend / Resume / Sleep

**Função para retirar uma *thread* de execução**

```
DWORD SuspendThread( HANDLE  hThread ); // Devolve o nº anterior do SuspCount
```

**Função para colocar uma *thread* em execução**

```
DWORD ResumeThread(HANDLE  hThread); // Devolve o nº anterior de SuspCount
```

Cada thread tem um **Suspend Count** (0, 1, ...)

**Função para colocar a *thread* corrente a dormir**

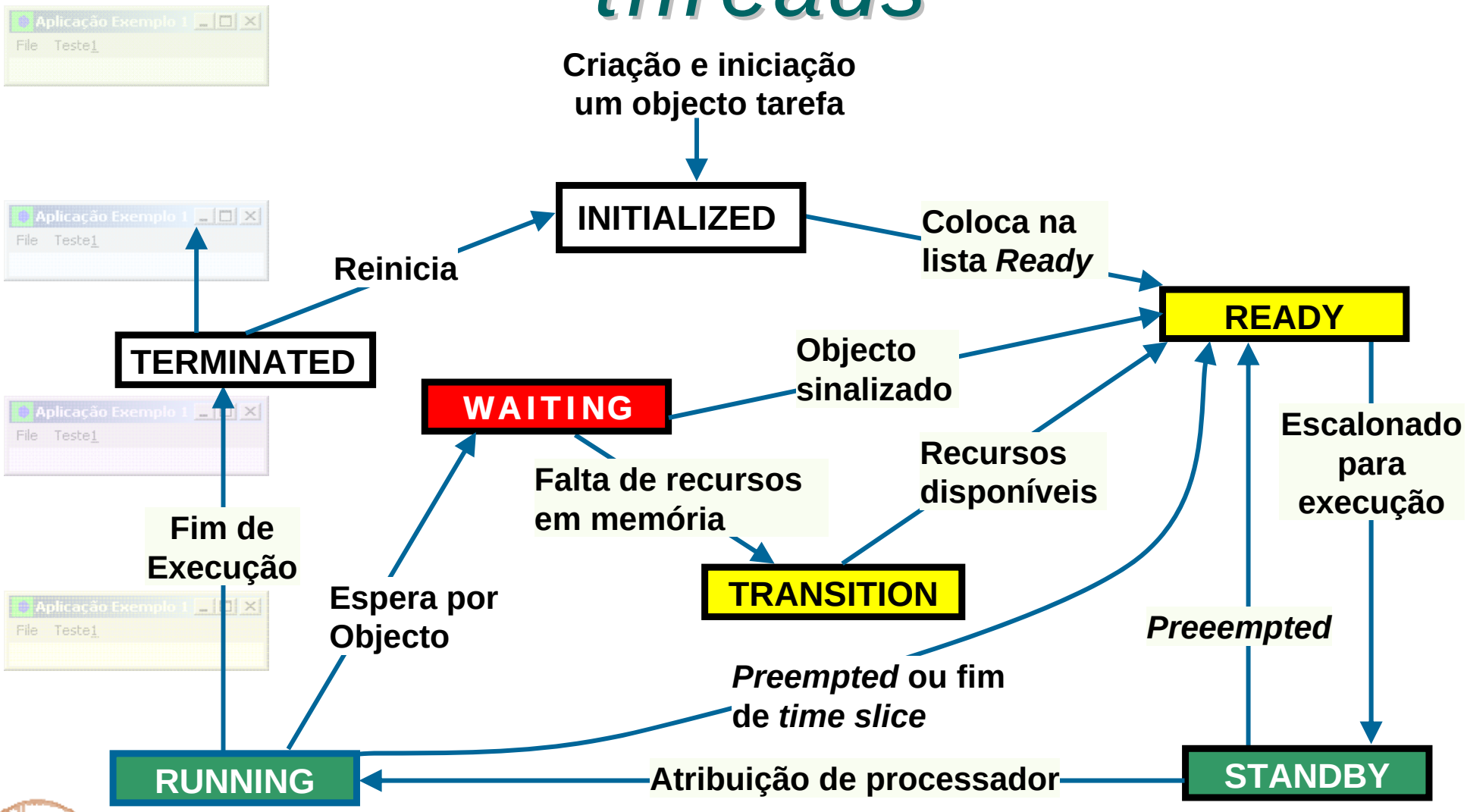
```
VOID Sleep( DWORD dwMilliseconds);
```

**Função para a *thread* desistir do seu *time slice*** (só para >=NT4)

```
BOOl SwitchToThread (VOID);
```

A diferença para Sleep(0) é que o SwichToThread permite a execução de tarefas de menor prioridade

# Diagrama de estados das *threads*

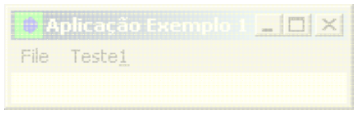


# Consultar os tempos de execução de uma tarefa


**Consultar os tempos associados à Thread**

```
BOOL GetThreadTimes(
  HANDLE hThread,                // handle to Thread
  LPFILETIME lpCreationTime, // thread creation time (UTC time)
  LPFILETIME lpExitTime,     // thread exit time (UTC time)
  LPFILETIME lpKernelTime,   // thread kernel-mode time
  LPFILETIME lpUserTime );   // thread user-mode time
```

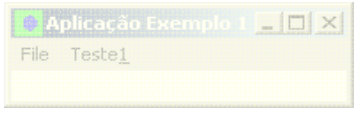
```
CreationTime & ExitTime  data em tempo UTC
KernelTime & UserTime    quantidade de tempo
```

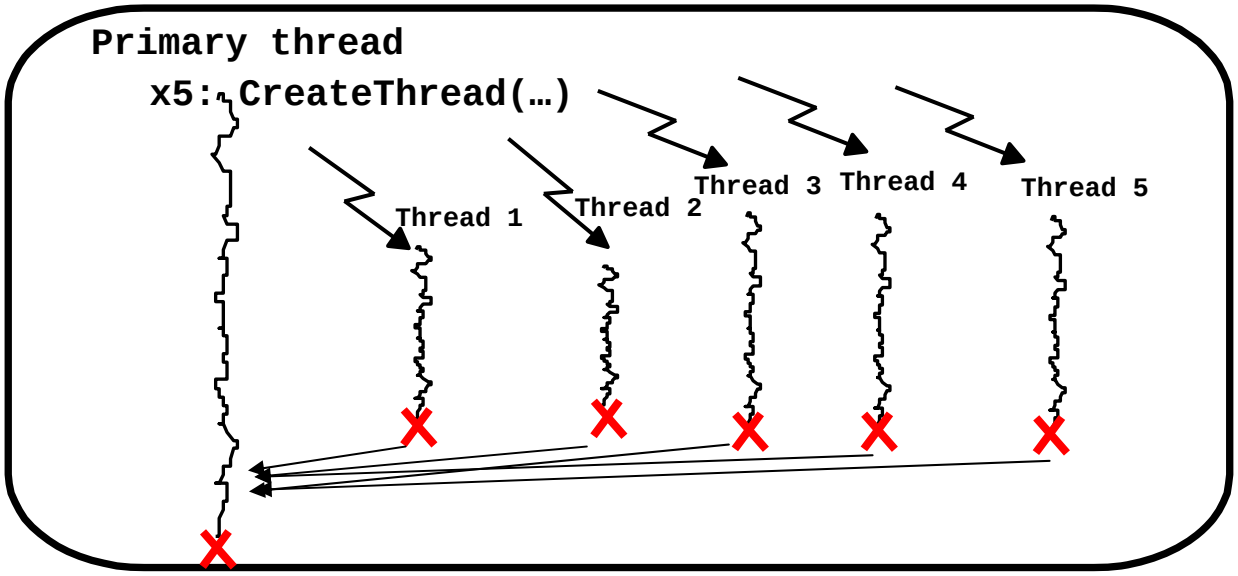



# Exemplo

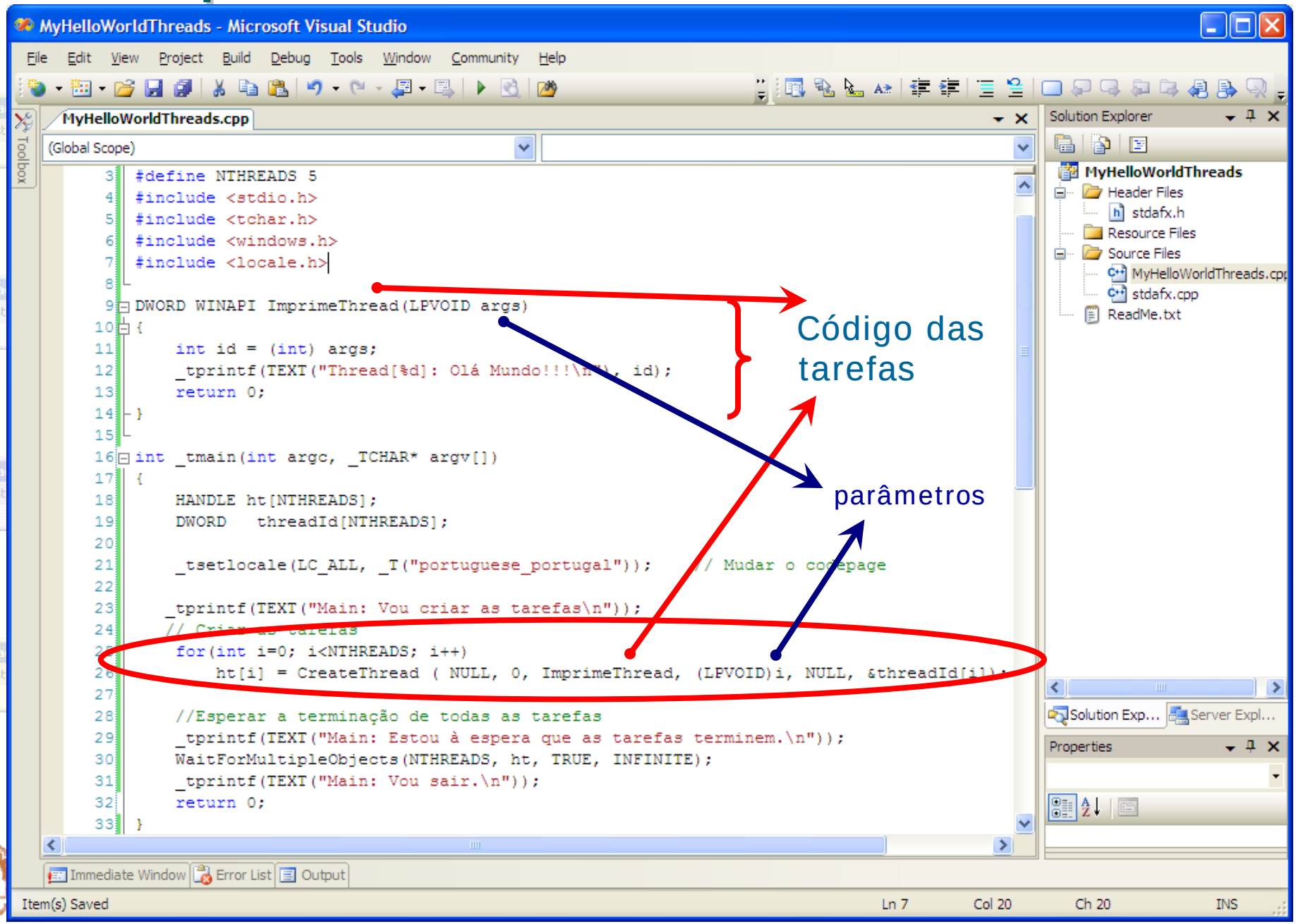

- Criar cinco tarefas para cada uma imprimir:
  - a sua identificação;
  - e a *string* “Olá Mundo”.



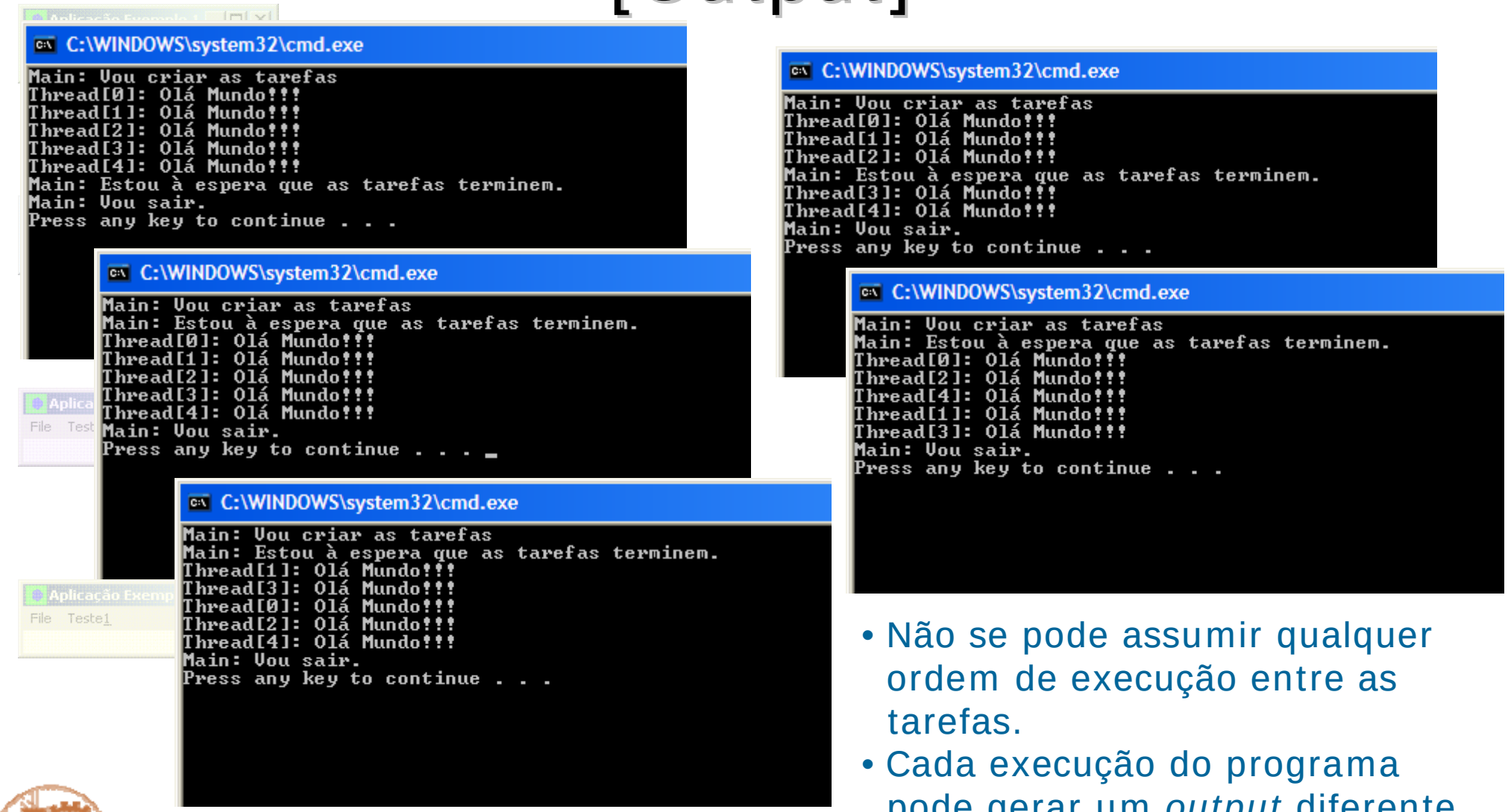



# Exemplo: Olá Mundo com 5 tarefas



# Exemplo: Olá Mundo com 5 tarefas [Output]




- Não se pode assumir qualquer ordem de execução entre as tarefas.
- Cada execução do programa pode gerar um *output* diferente.

# *Multithreading* e a biblioteca C/C++ Run-Time

# *Uso de tarefas e funções da biblioteca do standard C*

- A biblioteca standard C foi desenvolvida por volta de 1970 antes da existência de *Threads* nos Sistemas Operativos;
- A biblioteca não foi desenvolvida tendo em consideração a sua utilização em aplicações *multi-thread* (exº: utiliza variáveis globais como *errno*)
- Em ambiente *multi-thread*, cada *thread* terá de possuir as suas próprias variáveis globais da biblioteca C (uma variável *errno* por cada *thread*).

Biblioteca para aplicações *single thread*: **LIBC.LIB**
Biblioteca para aplicações *multi-thread*: **LIBCMT.LIB**

# Ambiente *multithread*

- Para as aplicações C/C++ se executarem correctamente, deve ser criada uma estrutura de dados, que é associada a cada *thread* e utilizada pelas funções da biblioteca C/C++.
- Quando é utilizada uma função da biblioteca C/C++ essas funções devem procurar, nessa estrutura de dados, as variáveis que partilham entre as várias chamadas das funções da biblioteca

**Como é que o sistema determina que é necessário criar essa nova estrutura de dados quando é criada uma *thread*?**

- O sistema operativo não advinha que as aplicações estão escritas em C/C++ e que estão a utilizar funções que não foram desenvolvidas para ambiente *multi-thread*.
- É da responsabilidade do programador, para criar uma nova *thread*, chamar a função, da biblioteca C/C++, **_beginthreadex()** em vez da chamada de sistema CreateThread().

# Função _beginthreadex

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

- A função _beginthreadex() tem os mesmos parâmetros da função CreateThread(), no entanto os tipos não são exactamente os mesmos (não tem de existir dependência entre os tipos das funções de biblioteca C/C++ com os tipos do Windows);
- Se utilizar funções da biblioteca C/C++ basta substituir a chamada CreateThread() por _beginthreadex();
- Os tipos não são exactamente iguais existindo necessidade de realizar coerções de tipo (cast);
- Para simplificar a utilização da função _beginthreadex() definiu-se a macro **chBEGINTHREADEX** que recebe exactamente os mesmos parâmetros que CreateThread

# Macro para criação de tarefas chBEGINTHREADEX

**Macro para criar uma Thread (para uso das funções do standard C)**

```
HANDLE chBEGINTHREADEX (
   LPSECURITY_ATTRIBUTES    lpThreadAttributes,
   DWORD                    dwStackSize,
   LPTHREAD_START_ROUTINE   lpStartAddress,
   LPVOID                   lpParameter,
   DWORD                    dwCreationFlags,
   LPDWORD                  lpThreadId
);
```

**Para o terminar das Threads devemos manter a coerência:**

```
VOID ExitThread( UINT  ExitCode )          // Termina a thread
void _endthreadex( unsigned ExitCode )     // Lib do C
VOID chENDTHREADEX( UINT ExitCode )  // macro
```

Macros definidas no ficheiro *BeginThreadex.h*
Fazer include de "BeginThreadex.h"

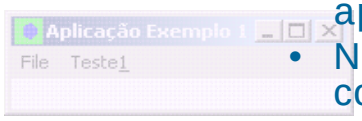
# Macro chBEGINTHREADEX

```
#ifndef _BEGINTHREADEX_H
#define _BEGINTHREADEX_H
#include <process.h>

// Macro chBEGINTHREADEX que chama a função da biblioteca do C _beginthreadex().
// No entanto esta não mantém a compatibilidade com os tipos de dados do WIN32,
por exemplo
// o tipo devolvido, obrigando utilizador a fazer uma coerção para HANDLE.
// A macro tem como objectivo facilitar a utilização da função _beginthreadex()

typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX( lpThreadAttributes, dwStackSize, lpStartAddress, \
                         lpParameter, dwCreationFlags, lpThreadId)        \
            ((HANDLE)_beginthreadex( (void *) (lpThreadAttributes),       \
                                     ( unsigned) (dwStackSize),           \
                                     (PTHREAD_START) (lpStartAddress),    \
                                     (void *) (lpParameter),              \
                                     (unsigned) (dwCreationFlags),        \
                                     (unsigned *) (lpThreadId)))

#endif
```

# Convenção de chamada a funções


- Na definição de funções C/C++ pode-se indicar a convenção de chamada à função que indica a responsabilidade de limpar o stack em relação aos argumentos da função:
  - <*return-type*> **__stdcall** *function-name*[(*argument-list*)]
    - O **código da função chamada** é responsável por eliminar os argumentos do stack (ajuste do stack)


  - <*return-type*> **__cdecl** *function-name*[(*argument-list*)]
    - O **código que chama a função** é responsável por eliminar os argumentos do stack após o retorno da função.
    - No C/C++ podem existir funções com argumentos variáveis implicando esta convenção.



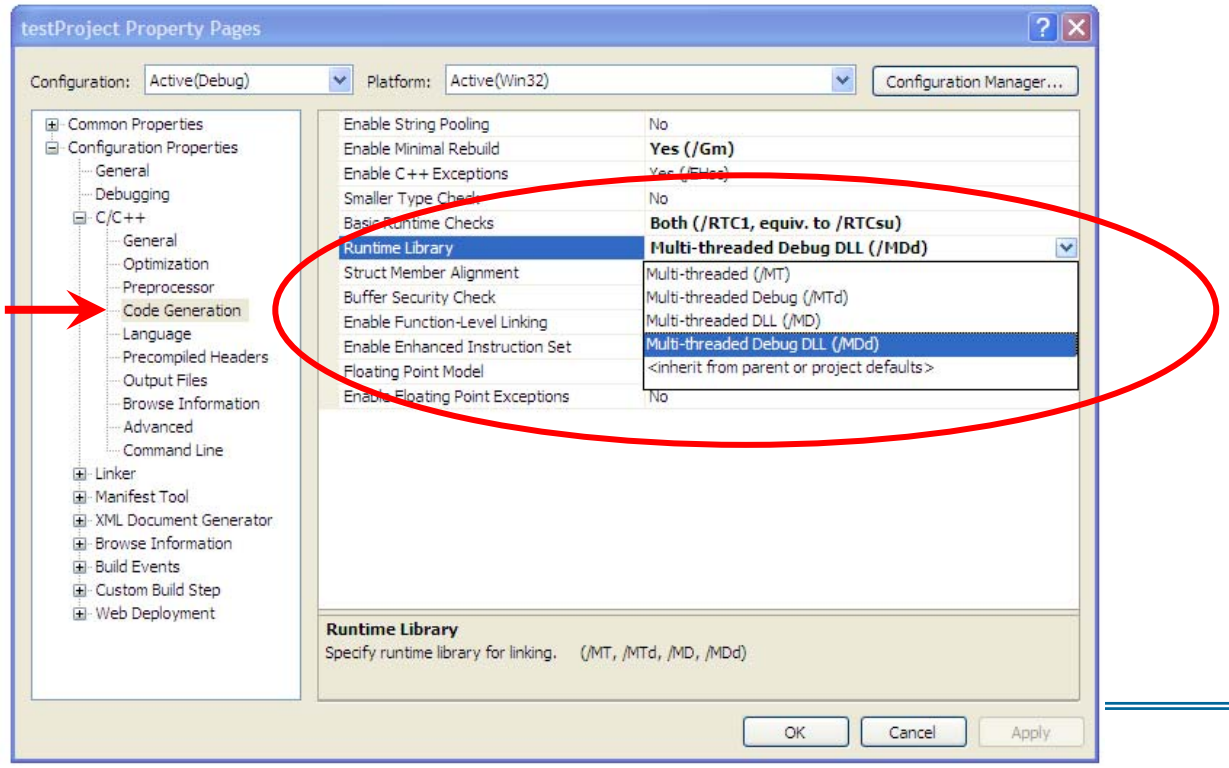

- No ficheiro Windows.h estão definidas as seguintes macros:

```
#define CALLBACK   __stdcall
#define WINAPI     __stdcall
#define WINAPIV    __cdecl
#define APIENTRY   WINAPI
#define PASCAL     __stdcall
```


- Toda a API WIN32 está definida com a convenção WINAPI

# Activar a biblioteca de *MultiThread*

- A função _beginthreadex só existe na versão *multithread* da biblioteca C/C++
- Se estivermos a utilizar a biblioteca *single thread* o *linker* apresenta o erro: *unresolved external symbol*
- Por omissão, quando se cria um novo projecto, o **Visual Studio 2003** utiliza a biblioteca C/C++ para *single thread*. Para o desenvolvimento de aplicações *multithread* temos de alterar a biblioteca C/C++ a utilizar (Project->Settings)
- No **Visual Studio 2005** já só existem bibliotecas *run time* C/C++ *multithread*



# Prioridades e scheduling

# Prioridade do processo

## Classes de prioridade

| | |
|---|---|
| **IDLE_PRIORITY_CLASS** | **Screen-savers, system monitors** |
| **BELOW_NORMAL_CLASS** | * |
| **NORMAL_PRIORITY_CLASS** | **Default** |
| **ABOVE_NORMAL_CLASS** | * |
| **HIGH_PRIORITY_CLASS** | **For time-critical response** |
| **REALTIME_PRIORITY_CLASS** | **For applications that talk directly to hardware** |

* Só a partir do windows 2000

## Funções de manipulação da prioridade do processo

```
DWORD GetPriorityClass(HANDLE hproc);
```

```
BOOL SetPriorityClass(HANDLE hproc, DWORD dwPriorityClass);
```

# Prioridade de uma Thread

```
BOOL SetThreadPriority(HANDLE hThread, int nPriority);
```

```
int GetThreadPriority(HANDLE hThread);
```

**nPriority :**

| | |
|---|---|
| **THREAD_PRIORITY_IDLE** | **1 (RT 16)** |
| **THREAD_PRIORITY_LOWEST** | **processP -2** |
| **THREAD_PRIORITY_BELOW_NORMAL** | **processP -1** |
| **THREAD_PRIORITY_NORMAL** | **processP** |
| **THREAD_PRIORITY_ABOVE_NORMAL** | **processP +1** |
| **THREAD_PRIORITY_HIGHEST** | **processP +2** |
| **THREAD_PRIORITY_TIME_CRITICAL** | **15 (RT 31)** |

# Tabela de Prioridades

Prioridade associada ao processo




Prioridade da thread


| | IDLE | BELOW NORMAL* | NORMAL | ABOVE NORMAL* | HIGH | REAL TIME |
|---|---|---|---|---|---|---|
| **IDLE** | 1 | 1 | 1 | 1 | 1 | 16 |
| **LOWEST** | 2 | 4 | 6 | 8 | 11 | 22 |
| **BELOW NORMAL** | 3 | 5 | 7 | 9 | 12 | 23 |
| **NORMAL** | 4 | 6 | 8 | 10 | 13 | 24 |
| **ABOVE NORMAL** | 5 | 7 | 9 | 11 | 14 | 25 |
| **HIGHEST** | 6 | 8 | 10 | 12 | 15 | 26 |
| **TIME CRITICAL** | 15 | 15 | 15 | 15 | 15 | 31 |

* Windows 2000 e seguintes

# Scheduling

**Queues/Prioridades das Threads : 0* (min) ... 31(max)**

16 – 31 : 16 real time levels
1 – 15 : 15 variable levels (dynamic range)
0 : 1 system level (* Kernel Thread ZeroPage)

Scheduling entre queues/prioridades diferentes :
Priority based, corre a thread da queue mais prioritária

Scheduling na mesma queue/prioridade :
Round robin with Time slice

Quantum
Typical Pentium uniprocessor systems
Clock tick: 15ms
Default Quantum: 30 ms NT4WS, 180 ms NT4SVR

# Scheduling adjustments


**Somente para as *threads* na gama dinâmica de prioridades: 1-15**


**Increase quantum for threads in the Foreground process**

Windows 2000 e seguintes

**Priority Boost Upon wait completion**

Boosts de 1(event, semaphore) a 8 (sound), por cada quantum a prioridade decresce de 1 unidade até ao valor normal

**Priority Boost for Gui threads entering wait state**

Boost de 8, com retorno imediato, mas com quantum dobrado

**Priority Boost For threads not running for a long time**

De segundo a segundo, todas as threads bloqueadas à 300 clock ticks, ficam com prioridade de 15 (com retorno imediato) e quantum dobrado

# Mais algumas funções da WIN32 API...

## relacionadas com processos e tarefas

# Pseudo Handles

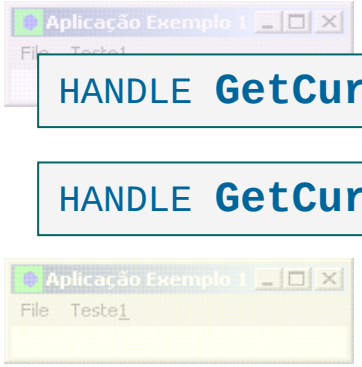
**Pseudo *handles***

São referências que são interpretadas como handles,
como não são verdadeiros handles:

- não incrementam o “handle count”,
- não são herdáveis,
- nem necessitam de ser fechados (CloseHandle)

```
HANDLE GetCurrentThread(void);   // obtém pseudo-handle da thread corrente
```

```
HANDLE GetCurrentProcess(void);  // obtém pseudo-handle do processo corrente
```



# Função *Duplicate handle*

**DuplicateHandle**

Cria um novo *handle* real, quer a "*source*" seja:

um *handle* real, ou

um *pseudo-handle*

```
BOOL DuplicateHandle(
    HANDLE hSourceProcessHandle, // handle to process with handle to duplicate
    HANDLE hSourceHandle,        // handle to duplicate
    HANDLE hTargetProcessHandle, // handle to process to duplicate to
    LPHANDLE lpTargetHandle,     // pointer to duplicate handle
    DWORD dwDesiredAccess,       // access for duplicate handle
    BOOL bInheritHandle,         // handle inheritance flag
    DWORD dwOptions );           // optional actions
```

**Exemplo: Criar um *handle* real para a *thread* corrente**

```
HANDLE hp = GetCurrentProcess(), ht = GetCurrentthread(), hnew;
DuplicateHandle( hp, ht, hp, &hnem, NULL, FALSE, DUPLICATE_SAME_ACCESS);
```

# Threads & Process IDs

Estes IDs, são identificadores únicos no sistema, que identificam univocamente o objecto no sistema, e que são válidos enquanto os referidos objectos (*thread*/processo) existirem.

**Obter o ID de objecto *thread* ou processo**

```
DWORD GetCurrentThreadId(void);
```

```
DWORD GetCurrentProcessId(void);
```

**Abrir um objecto *thread* ou processo através do seu ID**

```
HANDLE OpenThread(
        DWORD dwDesiredAccess, // access right
        BOOL bInheritHandle,   // handle inheritance option
        DWORD dwThreadId  );   // thread identifier
```

```
HANDLE OpenProcess(
        DWORD dwDesiredAccess, // access flag
        BOOL bInheritHandle,   // handle inheritance option
        DWORD dwProcessId );   // process identifier
```

Criam um novo handle para o objecto