Ficha nº 5 – Sincronização – I
Mutexes, Critical Sections, Eventos, Waitable Timers

Âmbito da matéria

Os exercícios desta ficha abordam:

- Sincronização de *threads* com *Mutexes* e *Critical Sections*.
- Coordenação de *threads* com Eventos e *Waitable Timers*.
- Obtenção de dados de e para *threads*.

Pressupostos:

- Conhecimento de algoritmia e de programação em C e C++.
- Conhecimentos de conceitos de base em SO (1º semestre).
- Conhecimento da matéria das aulas anteriores e das aulas teóricas, em particular, *threads*.

Referências bibliográficas

- Material das aulas teóricas
- Capítulos 6 e 8 do Livro Windows System Programming (da bibliografia)
- MSDN:

  Synchronization Objects

  https://docs.microsoft.com/pt-pt/windows/win32/sync/synchronization-objects

  Wait Functions

  https://msdn.microsoft.com/en-us/library/windows/desktop/ms687069(v=vs.85).aspx

  Time Functions

  https://docs.microsoft.com/en-us/windows/win32/sysinfo/time-functions

  Estes links apontam para o topo dos tópicos. Os sítios contêm links para os assuntos mais específicos subordinados ao tema.

Introdução e contexto

Esta ficha aborda questões básicas de sincronização, mais concretamente:

- Acesso em exclusão mútua a dados por *threads* em execução concorrente.
- Coordenação de acontecimentos produzidos por diversas *threads*.

Nesta ficha vai organizar o seu programa em diversas *threads*, e esse assunto deverá estar já claro. O foco será a forma como essas *threads* se coordenam no tempo de forma a efetuarem um trabalho global de forma coerente. Vão ser usados os seguintes mecanismos de sincronização:

- **Mutexes**
  - Permitem a duas ou mais *threads* (ou processos) aguardarem por acesso a uma zona de código (”secção crítica”) que manipula um recurso (ex., dados) partilhados 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, o ficam restringidos a *threads* do mesmo processo.
- **Critical Sections**
  - Não confundir com o conceito de zona de código – “secção crítica”.
  - 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*. 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 bloqueado e mais tarde ser reativada. Isto faz sentido quando se sabe à partida que o recurso que se pretende obter fica bloqueado por períodos muito curtos.
- **Eventos**
  - Servem para uma *thread* indicar que “algo” aconteceu a uma ou mais *threads* que aguardavam por esse algo. Exemplo de aplicação: aguardar que uma *thread* conclua uma tarefa qualquer mas continuar em execução, não se podendo assim aguardar que essa *thread* termine. Outro exemplo: indicar a várias *threads* que “podem começar” a fazer algo.
  - Os eventos são bastante versáteis e permitem: deixar passar apenas uma das *threads* que aguardam ou todas. A evitar: confundir o cenário de aplicação com cenários de exclusão mútua, pois são completamente diferentes.
- **Waitable Timers**
  - Trata-se de um mecanismo semelhante aos eventos mas que permite definir um período de tempo. Permite implementar mecanismos de temporização (“despertadores”) de forma bastante simples.

As funções de espera são iguais. As funções de criação e de libertação variam consoante o mecanismo.
Esta introdução não dispensa as aulas teóricas

Exercícios

Estes exercícios estão relacionados entre si. Cada exercício acrescenta ou modifica algo em relação ao anterior. Deve resolver todos os exercícios pela ordem indicada e seguir as indicações dadas pelo professor.

A forma mais eficiente de resolução implica, a cada novo exercício, modificar o código do anterior. De forma a não perder informação, antes de iniciar cada novo exercício, copie o código do anterior para outro local.

Pode utilizar *CreateThread* em vez de *_beginthreadex*, mas evite usar funções de biblioteca C *standard*.

1. Construa um projeto *Console Win32* vazio, sem recurso a qualquer tipo de código inicial gerado pelo IDE. Adicione a esse projeto um ficheiro ficha5.c (recorde que a extensão determina C vs. C++) e coloque nesse ficheiro o código apresentado mais adiante nesta ficha (listagem 1). O código está incompleto e o programa resultante não faz nada, mas deve compilar e produzir um executável. Garanta que o seu projeto está correto e consegue gerar um executável.

2. Usando o projeto do exercício anterior como ponto de partida, escreva um programa que determine a quantidade de número múltiplos de 3 existentes entre 1 e um limite superior indicado pelo utilizador. A sua implementação deve obedecer aos seguintes requisitos:

    - A contagem é feita por uma ou mais *threads*. O espaço de pesquisa é dividido entre essas *threads*. Exemplo: 1 a 6000 com 3 *threads*: a primeira analisa o intervalo entre 1 e 2000, a segunda entre 2001 e 4000, e a terceira entre 4001 e 6000. Estes valores são apenas exemplos.

    - Sempre que é encontrado um número múltiplo de 3, a *thread* que o encontra incrementa imediatamente um contador que é comum a todas as *threads*.

    - As *threads* devem executar em simultâneo uma com as outras. Para garantir que quando vai lançar uma nova *thread* a anterior não *começou-e-já-acabou*, crie todas as *threads* inicialmente suspensas e depois "des-suspenda-as" todas. Não dê intervalos de pesquisa demasiado pequenos, mas também não demasiado grandes para não perder tempo. Se o tempo de execução das *threads* for 3 a 6 segundos, estará bem.

    - Não deve haver variáveis globais. A informação de controlo de cada *thread* inclui, entre outros: qual a gama a pesquisar, a *flag* "deve continuar", e o ponteiro para o contador (o contador é o mesmo para todas as *threads* e, por isso, é indicado indiretamente através de um ponteiro). Defina uma estrutura com estes dados e cada *thread* receberá um ponteiro para uma instância dessa estrutura.

    - Não utilize nenhum mecanismo de sincronização exceto aguardar que as *threads* terminem.

    a) Contrua o programa.

    b) Execute o programa com uma *thread* e um limite superior de forma que o tempo de execução ande na casa dos 10 segundos. Anote: o limite que usou, a quantidade de múltiplos de 3, o tempo de execução.

c) Experimente o programa usando um número crescente de *threads* (2, 3, 4, etc.). Repare no que acontece a:

   i. Tempo total de execução: à medida que vai acrescentando *threads*, o tempo total de execução diminui, o que significa que está a aumentar a performance global do programa. Explique este fenómeno.

   ii. A partir de um certo ponto (número de *threads*), o tempo de execução deixa de diminuir. Eventualmente até pode subir ligeiramente. Explique a razão. Sugestão: execute o *task manager* e verifique quantos núcleos tem no seu processador.

   iii. O número total de múltiplos de 3 deixa de estar correto: é menor do que o esperado, e é mais errado quanto maior o número de *threads*. Explique este fenómeno.

3. Utilize um *mutex* para garantir o acesso em exclusão mútua ao contador global. Deve adicionar o *handle* deste *mutex* aos dados fornecidos às *threads*.

   a) Execute o programa com uma *thread* usando o mesmo limite superior que no exercício anterior. Verifique que o tempo de execução aumentou bastante quando comparado com o mesmo cenário (uma *thread*). Explique a razão.

   b) Execute o programa com duas *threads* e depois três, usando o mesmo limite superior. Verifique que:

      i. A quantidade de múltiplos de 3 está, agora, certa, independentemente do número de *threads* que usa.

      ii. O tempo de execução é maior que no caso em que não usava o *mutex*, e que o aumento pode ser pior quanto mais *threads* tem.

   Se o tempo de execução for muito grande, não experimente com mais *threads*, ou então, diminua o limite superior.

4. Repita o exercício anterior usando ***CriticalSections*** em vez de *Mutexes*.

   a) Verifique se ganha algum tempo de execução quando muda de Mutex para CriticalSection (mantendo tudo o resto igual).

   b) Experimente vários valores para o *Spin Count* e veja o impacto no tempo de execução.

5. Crie as *threads* no estado inicial não-suspenso e utilize um ***Evento*** para conseguir a garantia de que as todas as *threads* começam o seu trabalho ao mesmo tempo. Para tal, a primeira ação que as *threads* fazem é aguardar num evento. Esse evento está inicialmente não-assinalado e é colocado como assinalado pela função *main* quando esta deseja que as *threads* comecem o seu trabalho.
   Este exercício serve apenas para tomar contacto com um evento e não tem a ver com alternativas a *mutexes/critical sections*.

6. Pretende-se que se deixe de especificar um limite superior. Em vez disso, as *threads* ficam a trabalhar até que o utilizador as mande parar. O limite superior deixa de existir e deixa de se poder dividir um intervalo por *n threads*, uma vez que esse intervalo não é conhecido. A estratégia passa a ser a seguinte: cada *thread* passa a trabalhar com blocos de 10000 números. A próxima *thread* a ser criada vai consultar alguns dados de controlo para saber qual o próximo bloco de 10000 números. O mesmo acontece quando uma *thread* já esgotou o seu bloco e vai buscar outro.
   O trabalho das *threads* termina quando o utilizador introduz a palavra "fim". Cada *thread* verifica a condição de paragem apenas no fim do seu bloco atual.
   Deve cumprir o seguinte requisito adicional

   - A cada novo bloco testado por uma *thread* (10000 nessa *thread*), será impresso o caracter "*" no ecrã. Quem faz isso é a função *main*. Este pormenor implica a necessidade de uma *thread* nova e de um evento cuja configuração é diferente da do evento que usou no exercício 5. Verifique que consegue perceber a diferença deste evento para o anterior.

7. Modifique a solução anterior da seguinte forma: o tempo máximo deixa de ser controlado pela introdução da palavra "fim". Em vez disso, é perguntado ao utilizador qual o tempo máximo (aproximadamente) em segundos que deseja esperar. Tudo o resto mantém-se.
   A *thread* nova que foi necessária no exercício 6 deixa de ser necessária. Tente obter uma solução em que não cria essa *thread* adicional.

(Código inicial na próxima página)

Listagem de Programas

Código de partida para os exercícios desta ficha – listagem 1.

```
#include <windows.h>
#include <tchar.h>
#include <math.h>

#include <stdio.h>
#include <fcntl.h>
#include <io.h>


// funcionalidade relacionada com temporização

static double PerfCounterFreq; // n ticks por seg.

void initClock() {
	LARGE_INTEGER aux;
	if (!QueryPerformanceFrequency(&aux))
		_tprintf(TEXT("\nSorry - No can do em QueryPerfFreq\n"));
	PerfCounterFreq = (double)(aux.QuadPart); // / 1000.0;
	_tprintf(TEXT("\nTicks por sec.%f\n"), PerfCounterFreq);
}

__int64 startClock() {
	LARGE_INTEGER aux;
	QueryPerformanceCounter(&aux);
	return aux.QuadPart;
}

double stopClock(__int64 from) {
	LARGE_INTEGER aux;
	QueryPerformanceCounter(&aux);
	return (double)(aux.QuadPart - from) / PerfCounterFreq;
}

// estrutura de dados para controlar as threads

typedef struct {
	// ...
	int x; // remover este inteiro. Está aqui apenas para este código compilar
} TDados;

// função da(s) thread(s)
// ...

// número * máximo * de threads
// podem (e devem) ser menos
#define MAX_THREADS 20
```

```
int _tmain(int argc, TCHAR * argv[]) {

	// matriz de handles das threads
	HANDLE hThreads[MAX_THREADS];

	// Matriz de dados para as threads;
	TDados tdados[MAX_THREADS];

	// número efectivo de threads
	int numthreads;

	// limite superior
	unsigned int limsup;

	// variáveis para cronómetro
	__int64 clockticks;
	double duracao;

	unsigned int range;
	unsigned int inter;

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

	initClock();
	_tprintf(TEXT("\nLimite sup. -> "));
	_tscanf(TEXT("%u"), &limsup);
	_tprintf(TEXT("\nNum threads -> "));
	_tscanf(TEXT("%u"), &numthreads);
	if (numthreads > MAX_THREADS)
		numthreads = MAX_THREADS;

	// FAZER prepara e cria threads
	//	manda as threads começar

	clockticks = startClock();

	// FAZER aguarda / controla as threads
	//	manda as threads parar

	duracao = stopClock(clockticks);
	_tprintf(TEXT("\nSegundos=%f\n"), duracao);

	// FAZER apresenta resultados

	// Cód. ref. para aguardar por uma tecla - caso faça falta
	// _tprintf(TEXT("\nCarregue numa tecla"));
	// _gettch();

	return 0;
}
// Este código é apenas uma ajuda para o exercício. Se quiser, mude-o
```

Resumo das funções API mais centrais a estes exercícios

Notas

- Existem mais funções que estas
- Atenção à questão TCHAR/ …A() / …W()
- As funções *wait* são comuns aos vários mecanismos
- Os *handles* devem ser fechados quando o objeto deixa de ser necessário (*CloseHandle*)
- Cada mecanismo têm os seus próprios detalhes importantes. Exemplo: comportamento do *mutex* (ver alguns detalhes abaixo). Este resumo não contempla esses pormenores.

As páginas seguintes têm excertos extensos de material da Microsoft. Esse conteúdo está, naturalmente, sujeito a propriedade intelectual e pertence à Microsoft. Os excertos são aqui colocados em contexto de divulgação académica e não devem ser transcritos para outros contextos.

- **Funções de espera**

WaitForSingleObjectEx

Waits until the specified object is in the signaled state, an I/O completion routine or asynchronous procedure call (APC) is queued to the thread, or the time-out interval elapses.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobjectex

```
DWORD WaitForSingleObjectEx(
  HANDLE hHandle,
  DWORD  dwMilliseconds,
  BOOL   bAlertable
);
```

WaitForMultipleObjectsEx

Waits until one or all of the specified objects are in the signaled state, an I/O completion routine or asynchronous procedure call (APC) is queued to the thread, or the time-out interval elapses.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitformultipleobjectsex

```
DWORD WaitForMultipleObjectsEx(
  DWORD        nCount,
  const HANDLE *lpHandles,
  BOOL         bWaitAll,
  DWORD        dwMilliseconds,
  BOOL         bAlertable
);
```

- **Mutexes**

CreateMutexExA

Creates or opens a named or unnamed mutex object and returns a handle to the object. If the function succeeds, the return value is a handle to the newly created mutex object.
If the function fails, the return value is NULL.
The thread that owns a mutex can specify the same mutex in repeated wait function calls without blocking its execution. Typically, you would not wait repeatedly for the same mutex, but this mechanism prevents a thread from deadlocking itself while waiting for a mutex that it already owns. However, to release its ownership, the thread must call ReleaseMutex once for each time that the mutex satisfied a wait.

Two or more processes can call CreateMutex to create the same named mutex. The first process actually creates the mutex, and subsequent processes with sufficient access rights simply open a handle to the existing mutex. This enables multiple processes to get handles of the same mutex, while relieving the user of the responsibility of ensuring that the creating process is started first. When using this technique, you should not use the CREATE_MUTEX_INITIAL_OWNER flag; otherwise, it can be difficult to be certain which process has initial ownership.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexexa

```
HANDLE CreateMutexExA(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  LPCSTR                lpName,
  DWORD                 dwFlags,
  DWORD                 dwDesiredAccess
);
```

OpenMutexW

Opens an existing named mutex object.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw

```
HANDLE OpenMutexW(
  DWORD   dwDesiredAccess,
  BOOL    bInheritHandle,
  LPCWSTR lpName
);
```

ReleaseMutex

Releases ownership of the specified mutex object.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-releasemutex

```
BOOL ReleaseMutex(
  HANDLE hMutex
);
```

Padrão de uso

***Thread* 1 / Processo 1:**

```
HANDLE hMutex;

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

***Thread 2* / Processo 2:**

```
HANDLE mutex;

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

- **<u>Critical Sections</u>**

InitializeCriticalSection

Initializes a critical section object.
The threads of a single process can use a critical section object for mutual-exclusion synchronization. There is no guarantee about the order in which threads will obtain ownership of the critical section, however, the system will be fair to all threads.
The process is responsible for allocating the memory used by a critical section object, which it can do by declaring a variable of type CRITICAL_SECTION. Before using a critical section, some thread of the process must initialize the object.
After a critical section object has been initialized, the threads of the process can specify the object in the EnterCriticalSection, TryEnterCriticalSection, or LeaveCriticalSection function to provide mutually exclusive access to a shared resource. For similar synchronization between the threads of different processes, use a mutex object.
A critical section object must be deleted before it can be reinitialized. Initializing a critical section that has already been initialized results in undefined behavior.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-initializecriticalsection

```
void InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

InitializeCriticalSectionAndSpinCount

Initializes a critical section object and sets the spin count for the critical section. When a thread tries to acquire a critical section that is locked, the thread spins: it enters a loop which iterates spin count times, checking to see if the lock is released. If the lock is not released before the loop finishes, the thread goes to sleep to wait for the lock to be released.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-initializecriticalsectionandspincount

```
BOOL InitializeCriticalSectionAndSpinCount(
  LPCRITICAL_SECTION lpCriticalSection,
  DWORD              dwSpinCount
);
```

DeleteCriticalSection

Releases all resources used by an unowned critical section object.
The caller is responsible for ensuring that the critical section object is unowned and the specified CRITICAL_SECTION structure is not being accessed by any critical section functions called by other threads in the process. If a critical section is deleted while it is still owned, the state of the threads waiting for ownership of the deleted critical section is undefined.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-deletecriticalsection

```
void DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

SetCriticalSectionSpinCount

Sets the spin count for the specified critical section. Spinning means that when a thread tries to acquire a critical section that is locked, the thread enters a loop, checks to see if the lock is released, and if the lock is not released, the thread goes to sleep.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-setcriticalsectionspincount

```
DWORD SetCriticalSectionSpinCount(
  LPCRITICAL_SECTION lpCriticalSection,
  DWORD              dwSpinCount
);
```

EnterCriticalSection

Waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership.
After a thread has ownership of a critical section, it can make additional calls to EnterCriticalSection or TryEnterCriticalSection without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. The thread enters the critical section each time EnterCriticalSection and TryEnterCriticalSection succeed. A thread must call LeaveCriticalSection once for each time that it entered the critical section.
If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.
If a critical section is deleted while it is still owned, the state of the threads waiting for ownership of the deleted critical section is undefined.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-entercriticalsection~

```
void EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

TryEnterCriticalSection

Attempts to enter a critical section without blocking. If the call is successful, the calling thread takes ownership of the critical section.
If the critical section is successfully entered or the current thread already owns the critical section, the return value is nonzero.
If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-tryentercriticalsection

```
BOOL TryEnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

LeaveCriticalSection

Releases ownership of the specified critical section object.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-leavecriticalsection

```
void LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection
);
```

Padrão de uso

***Thread* 1 / Processo 1:**

```
// handle acessível às threads envolvidas

CRITICAL_SECTION cs;
...
InitializeCriticalSection(&cs);

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

***Thread 2* / Processo 1:**

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

- **<u>Eventos</u>**

CreateEventExA

Creates or opens a named or unnamed event object and returns a handle to the object.
The dwFlags parameter can be one or more of the following values.

Value	Meaning
CREATE_EVENT_INITIAL_SET 0x00000002	The initial state of the event object is signaled; otherwise, it is nonsignaled.
CREATE_EVENT_MANUAL_RESET 0x00000001	The event must be manually reset using the ResetEvent function. Any number of waiting threads, or threads that subsequently begin wait operations for the specified event object, can be released while the object's state is signaled. If this flag is not specified, the system automatically resets the event after releasing a single waiting thread.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createeventexa

```
HANDLE CreateEventExA(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  LPCSTR                lpName,
  DWORD                 dwFlags,
  DWORD                 dwDesiredAccess
);
```

OpenEventA

Opens an existing named event object.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openeventa

```
HANDLE OpenEventA(
  DWORD  dwDesiredAccess,
  BOOL   bInheritHandle,
  LPCSTR lpName
);
```

SetEvent

Sets the specified event object to the signaled state.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-setevent

```
BOOL SetEvent(
  HANDLE hEvent
);
```

ResetEvent

Sets the specified event object to the nonsignaled state.
The state of an event object remains nonsignaled until it is explicitly set to signaled by the SetEvent or PulseEvent function. This nonsignaled state blocks the execution of any threads that have specified the event object in a call to one of the wait functions.
The ResetEvent function is used primarily for manual-reset event objects, which must be set explicitly to the nonsignaled state. Auto-reset event objects automatically change from signaled to nonsignaled after a single waiting thread is released.
Resetting an event that is already reset has no effect.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-resetevent

```
BOOL ResetEvent(
  HANDLE hEvent
);
```

PulseEvent

Sets the specified event object to the signaled state and then resets it to the nonsignaled state after releasing the appropriate number of waiting threads.
This function is unreliable and should not be used. It exists mainly for backward compatibility.

https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-pulseevent

- **Waitable Timers**

https://docs.microsoft.com/en-us/windows/win32/sync/waitable-timer-objects

A *waitable timer object* is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.

Table 1

Object	Description
manual-reset timer	A timer whose state remains signaled until **SetWaitableTimer** is called to establish a new due time.
synchronization timer	A timer whose state remains signaled until a thread completes a wait operation on the timer object.
periodic timer	A timer that is reactivated each time the specified period expires, until the timer is reset or canceled. A periodic timer is either a periodic manual-reset timer or a periodic synchronization timer.

A thread uses the **CreateWaitableTimer** or **CreateWaitableTimerEx** function to create a timer object. The creating thread specifies whether the timer is a manual-reset timer or a synchronization timer. The creating thread can specify a name for the timer object. Threads in other processes can open a handle to an existing timer by specifying its name in a call to the **OpenWaitableTimer** function. Any thread with a handle to a timer object can use one of the wait functions to wait for the timer state to be set to signaled.

- The thread calls the **SetWaitableTimer** function to activate the timer. Note the use of the following parameters for **SetWaitableTimer**:

- Use the *lpDueTime* parameter to specify the time at which the timer is to be set to the signaled state. When a manual-reset timer is set to the signaled state, it remains in this state until **SetWaitableTimer** establishes a new due time. When a synchronization timer is set to the signaled state, it remains in this state until a thread completes a wait operation on the timer object.

- Use the *lPeriod* parameter of the **SetWaitableTimer** function to specify the timer period. If the period is not zero, the timer is a periodic timer; it is reactivated each time the period expires, until the timer is reset or canceled. If the period is zero, the timer is not a periodic timer; it is signaled once and then deactivated.

A thread can use the **CancelWaitableTimer** function to set the timer to the inactive state. To reset the timer, call **SetWaitableTimer**. When you are finished with the timer object, call **CloseHandle** to close the handle to the timer object.

The behavior of a waitable timer can be summarized as follows:

- When a timer is set, it is canceled if it was already active, the state of the timer is nonsignaled, and the timer is placed in the kernel timer queue.

- When a timer expires, the timer is set to the signaled state. If the timer has a completion routine, it is queued to the thread that set the timer. The completion routine remains in the asynchronous procedure call (APC) queue of the thread until the thread enters an alertable wait state. At that time, the APC is dispatched and the completion routine is called. If the timer is periodic, it is placed back in the kernel timer queue.

- When a timer is canceled, it is removed from the kernel timer queue if it was pending. If the timer had expired and there is still an APC queued to the thread that set the timer, the APC is removed from the thread's APC queue. The signaled state of the timer is not affected.

CreateWaitableTimerExW

Creates or opens a waitable timer object and returns a handle to the object.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createwaitabletimerexw

```
HANDLE CreateWaitableTimerExW(
  LPSECURITY_ATTRIBUTES lpTimerAttributes,
  LPCWSTR               lpTimerName,
  DWORD                 dwFlags,
  DWORD                 dwDesiredAccess
);
```

OpenWaitableTimerW

Opens an existing named waitable timer object.
The OpenWaitableTimer function enables multiple processes to open handles to the same timer object. The function succeeds only if some process has already created the timer using the CreateWaitableTimer

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openwaitabletimerw

```
HANDLE OpenWaitableTimerW(
  DWORD   dwDesiredAccess,
  BOOL    bInheritHandle,
  LPCWSTR lpTimerName
);
```

SetWaitableTimer

Activates the specified waitable timer. When the due time arrives, the timer is signaled and the thread that set the timer calls the optional completion routine.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-setwaitabletimer

```
BOOL SetWaitableTimer(
  HANDLE              hTimer,
  const LARGE_INTEGER *lpDueTime,
  LONG                lPeriod,
  PTIMERAPCROUTINE    pfnCompletionRoutine,
  LPVOID              lpArgToCompletionRoutine,
  BOOL                fResume
);
```

CancelWaitableTimer

Sets the specified waitable timer to the inactive state.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-cancelwaitabletimer

```
BOOL CancelWaitableTimer(
  HANDLE hTimer
);
```

Considerações adicionais

Synchronization Object Security and Access Rights

https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-object-security-and-access-rights