Threads in MFC

Livro Texto: Multithreading Applications in Win 32 – Jim Beveridge, Robert Wiener

Em MFC temos um novo paradigma de programação em ambiente Windows. Usando a API do Win32 tínhamos que tratar todos os eventos possíveis do ambiente dentro do loop de mensagens após emitir o comando *GetMessage()*; As Microsoft Foundation Classes simplificam muito deste overhead, encapsulando em classes as principais funções da interface.

Conceitualmente é importante distinguir entre dois tipos de threads:

- As **threads de trabalho** (*working threads*) são aquelas que não possuem janela associada. Estas threads realizam funções que não têm nada a ver com interface homem máquina. No exemplo da animação do sistema de tanques, as threads de trabalho poderiam executar as funções de modelamento do tanque, válvulas e outros objetos, do controlador PID, do gerente de alarmes, do anunciador de mensagens (*voice maker*), do gerente de base de dados.

- As *GUI threads* ou **threads de interface** (*user interface threads*) são as threads que possuem fila de mensagem associada. A fila de mensagem é criada quando a thread chama a função *GetMessage()*, *CreateWindow()*, ou similar.

Para criar threads usando apenas as funções da API Win 32 as funções utilizadas eram:
- *CreateThread()*
- *EndThread()*

Para usar a biblioteca de funções run-time do C tínhamos que empregar uma forma alternativa implementada como um novo encapsulamento destas funções:
- *_beginthreadex()*
- *_endthreadex()*

Para criar threads que irão fazer uso das funções em MFC, temos que utilizar as funções:
AfxBeginThread() ou *CWinThread::CreateThread().*

Funções Afx
As funções Afx são funções ofecidas pela biblioteca MFC e que não são membros de classes. Estão funções não necessitam de ser chamadas através de um objeto: podem ser acessadas de qualquer lugar.

Para a função AfxBeginThread faz diferença se a thread a ser iniciada é uma thread de trabalho ou uma thread de interface.

Apenas os dois primeiros parâmetros de AfxBeginThread são obrigatórios. Os demais são default.

AfxBeginThread()

```
CwinThread* AfxBeginThread(
```

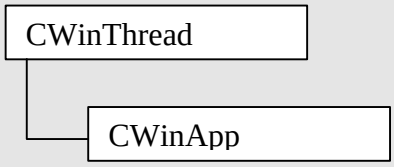
AFX_THREADPROC pfnThreadProc	//	Função representando a nova thread. A função deve ser do tipo: UINT MyControlFunction(LPVOID pParam);
LPVOID pParam	//	Parâmetro de 4 bytes a ser passado à nova thread
int nPriority = THREAD_PRIORITY_NORMAL	//	Prioridade da nova thread (nível). 0-> a thread terá a mesma prioridade da thread pai.
UINT nStackSize = 0	//	Tamanho do stack da nova thread em bytes. 0-> usa valor default.
DWORD dwCreateFlags = 0	//	0 ou omitido: thread começa imediatamente. CREATE_SUSPENDED: thread é criada no estado suspenso.
LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL	//	Atributos de segurança como em *CreateThreads().*
);		

Retorno da função:

Status	Interpretação
NULL	Falha
<> NULL	Ponteiro para objeto da classe CWinThread.

Exemplo 1: Thread de trabalho em MFC

```
/*
 * Numbers.cpp
 * Sample code for "Multithreading Applications in Win32"
 * Demonstrate basic thread startup in MFC
 * using AfxBeginThread.
*/

#include <afxwin.h>

// Classe base de onde você deriva um objeto de aplicação Windows.
// Só um, objeto pode ser definido por aplicação.
// Deve ser global

 CWinApp TheApp;
```




```
UINT ThreadFunc(LPVOID);

int main()
{
   for (int i=0; i<5; i++)  {
      if (AfxBeginThread( ThreadFunc, (LPVOID)i ))
         printf("Thread launched %d\n", i);
   }

   // Wait for the threads to complete.
   Sleep(2000);
   return 0;
}


UINT ThreadFunc(LPVOID n)
```

```
{
   for (int i=0;i<10;i++)
            printf("%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n);
   return 0;
}
```

Este programa é muito mais simples que um programa feito apenas com o uso da API do Win32.

- Nós não utilizamos handles e não tivemos que apagá-los depois de usá-los.
- Nós não usamos outras variáveis para guardar resultados intermediários como a ThreadID
- A ThreadFunc usa um protótipo muito mais simples.

Isto é bom ou ruim ?

- O que aconteceu com a espera do término da thread ?
  O objeto CWinThread é apagado automaticamente quando a thread termina.
  Mas, como eu faço se quiser as funções Wait… e esperar ?

- Quem fecha o handle para a thread ?
  A mesma rotina que termina o objeto.

- Onde ficam armazenados os valores do handle para a thread e a ThreadId ?
  São armazenados em variáveis membro do objeto CWinThread: m_nThreadID e m_hThread.

- MFC verifica o retorno das funções ou precisamos checar nós mesmos estes valores ?
  MFC confere todos os valores de parâmetros passados e retornos das funções e emite mensagens apropriadas em caso de erro.

No exemplo dado o autor não usou a função *WaitForMultipleObjects()* para esperar pelo retorno das threads. É que existe uma situação de corrida. Se a thread for pequena ela pode já ter morrido, quando *AfxBeginThread* retornar. A aplicação não conseguiria, neste caso, usar o ponteiro para a thread para obter o valor do handle. A saída para este problema é definir a variável membro m_bAutoDelete da classe CWinThread. A Thread deve ser criada no estado SUSPENSO.

Exemplo 2: Esperando pelo fim da thread em MFC

```
/* Numbers2.cpp
 * Sample code for "Multithreading Applications in Win32"
 * This is from Chapter 10, Listing 10-2
 * Demonstrate thread startup in MFC using AfxBeginThread, but
 * prevent CWinThread from auto-deletion so that we can wait on
 * the thread.
 */

#include <afxwin.h>

CWinApp TheApp;

UINT ThreadFunc(LPVOID);

int main()
{
   CWinThread* pThreads[5];

   for (int i=0; i<5; i++)
   {
      pThreads[i] = AfxBeginThread(
                    ThreadFunc,
                    (LPVOID)i,
                    THREAD_PRIORITY_NORMAL,
                    0,
                    CREATE_SUSPENDED
                  );
      ASSERT(pThreads[i]);
      pThreads[i]->m_bAutoDelete = FALSE;
      // Chama ResumeThread versão MFC
      pThreads[i]->ResumeThread(); // reativa thread criada
      printf("Thread launched %d\n", i);
   }

   for (i=0; i<5; i++)
   {
      WaitForSingleObject(pThreads[i]->m_hThread, INFINITE);
      delete pThreads[i];  // Agora é você que apaga o objeto
       // Não é necessário chamar CloseHandle() !!!
       // CWinThread irá realizar esta função
   }
   return 0;
}
```

Se 0 imprime mensagem de erro e aborta programa

```
UINT ThreadFunc(LPVOID n)
{
   for (int i=0;i<10;i++)
            printf("%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n);
   return 0;
}
```

Criando threads em MFC sem o uso de *AfxBeginThread()*

AfxBeginThread() encapsula uma série de operações da classe CwinThread. Isto pode ser muito bom para um programador iniciante, mas às vezes é inconveniente para um programador experiente.

O que faz *AfxBeginThread()* ?

1. Aloca um novo objeto do tipo CWinThread no heap.
2. Chama CWinThread::CreateThread e parte a thread no estado SUSPENSO.
3. Define a prioridade da thread.
4. Chama CWinThread::ResumeThread()

Quais as vantagens de se usar sua própria rotina para criar threads em MFC ?

- Pode-se armazenar os dados de inicialização da thread dentro da estrutura de CWinThread.
- Pode-se armazenar informações para comunicação entre threads na estrutura de CWinThread.
- Realizar uma série de ações default como por exemplo desabilitar a auto terminação da thread.

Para fazer isto, vamos criar nossa classe como derivada de CWinThread e vamos adicionar os nossos membros de dados particulares.

Nós iremos definir a função de criação da thread como um membro estático da nossa classe, para evitar os problemas analisados no capítulo anterior. A grande vantagem é poder ter várias funções com o mesmo nome da nossa ThreadFunc. A nossa função deve ser acessada pelo seu nome completo: CUserThread::ThreadFunc().

```
/*
 * NumClass.cpp
```

```
 *
 * Sample code for "Multithreading Applications in Win32"
 * This is from Chapter 10, Listing 10-3
 * Demonstrate worker thread startup in MFC without AfxBeginThread.
*/

#include <afxwin.h>

CWinApp TheApp;

class CUserThread : public CWinThread
{
public:  // Member functions
   // contrutor: passa função como parâmetro
   CUserThread(AFX_THREADPROC pfnThreadProc);

   static UINT ThreadFunc(LPVOID param);

public:  // Member data
   int m_nStartCounter;      // irá indicar a ordem de criação da thread

private: // The "real" startup function
   virtual void Go();
};


// Função construtor
CUserThread::CUserThread(AFX_THREADPROC pfnThreadProc)
 : CWinThread(pfnThreadProc, NULL)
{
   m_bAutoDelete = FALSE;
   // Set the pointer to the class to be the startup value.
   // m_pThreadParams is undocumented, but there is no workaround.
   m_pThreadParams = this;
}

int main()
{
   CUserThread* pThreads[5];

   for (int i=0; i<5; i++)
   {
      // Cria objeto do tipo CUSerThread e passa como parâmetro a
      // função membro estática
      pThreads[i] = new CUserThread( CUserThread::ThreadFunc );
```

Função estática não espera parâmetro oculto **this**

Chama construtor não documentado

Passa **this** como parâmetro

Gera mensagem de diagnóstico se 0.

```
        // Set the appropriate member variable
        pThreads[i]->m_nStartCounter = i;

        // Inicia thread: CreateThread pertence à classe CWinThread
        VERIFY( pThreads[i]->CreateThread() );
        printf("Thread launched %d\n", i);
    }

    for (i=0; i<5; i++)   {
        WaitForSingleObject(pThreads[i]->m_hThread, INFINITE);
        delete pThreads[i];
    }
    return 0;
}

// static
UINT CUserThread::ThreadFunc(LPVOID n) {
    CUserThread* pThread = (CUserThread*)n; // recebe parâmetro this
pThread->Go();       // passa a bola para a função thread
    return 0;
}

void CUserThread::Go()
{
    int n = m_nStartCounter;
    for (int i=0;i<10;i++)
        printf("%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n);
}
```

Membros de dados privados, protegidos e públicos:

- Os membros de dados ***private*** só podem ser acessados por funções membros da própria classe ou membros de classes *friends*.
- Os membros de dados ***public*** podem ser acessados por qualquer função e formam a interface da classe.
- Membros de dados ***protected*** são como membros de dados privados exceto que eles podem ser acessados pelas classes derivadas.

Se uma classe é declarada como classe básica para outra classe, com uso do especificador de acesso *public*, os membros *public* da classe básica serão membros *public* da classe derivada e os membros *protected* da classe básica serão *protected* na classe derivada. Se o especificador de acesso for *private*, os membros *public* e *private* se

tornam *private* na classe derivada e os membros *private* não podem ser acessados.

Resumo:

Especificador de Acesso	Classe base	Classe derivada
Public	*Public* *Protected* *Private*	*Public* *Private* Inacessível
Private	*Public* *Protected* *Private*	*Private* *Private* Inacessível

Iniciando Threads de Interface em MFC

A criação de threads de interface em MFC segue um cardápio muito diferente da criação de threads de trabalho.

AfxBeginThread()

```
CwinThread* AfxBeginThread(
```

CRuntimeClass *pThreadClass,	// Ponteiro classe derivada de *CwinThread* criada pelo usuário.
int nPriority = THREAD_PRIORITY_NORMAL	// Prioridade da nova thread (nível). 0-> a thread terá a mesma prioridade da thread pai.
UINT nStackSize = 0	// Tamanho do stack da nova thread em bytes. 0-> usa valor default.
DWORD dwCreateFlags = 0	// 0 ou omitido: thread começa imediatamente. CREATE_SUSPENDED: thread é criada no estado suspenso.
LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL	// Atributos de segurança como em *CreateThreads()*.
);	

Retorno da função:

Status	Interpretação
NULL	Falha
<> NULL	Ponteiro para objeto da classe CWinThread.

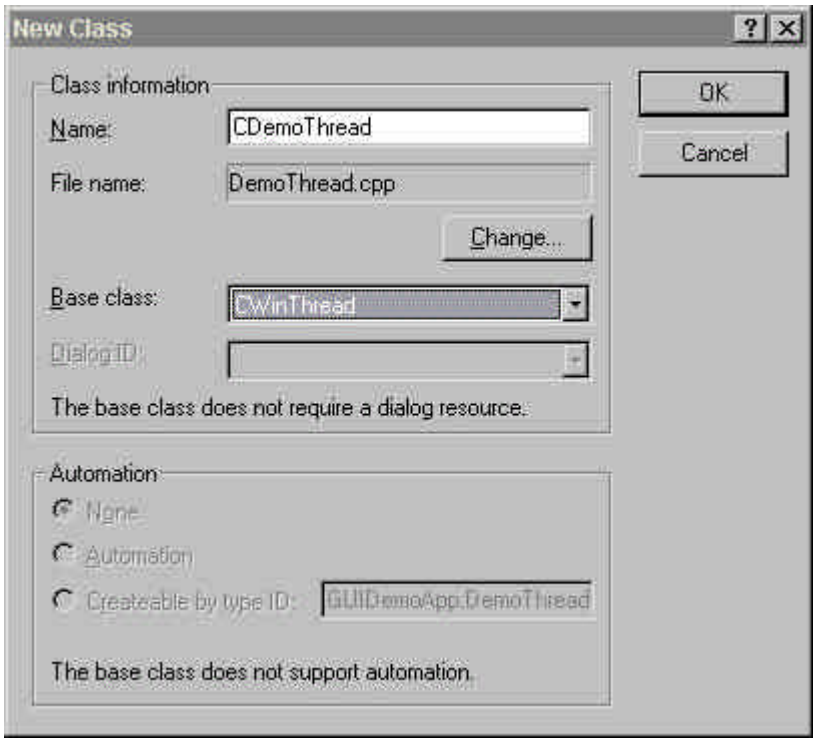
Esta função parece semelhante à anteriormente estudada, exceto pelo primeiro parâmetro e pelo segundo que foi eliminado.
Entretanto, como gerar o objeto derivado da classe CwinThread esperado ?

As funções virtuais de CwinThead poderiam ser sobrepostas por funções próprias que poderíamos construir.

As funções virtuais de CwinThread são:
- InitInstance()
- ExitInstance()
- OnIdle()
- PreTranslateMessage()
- IsIdleMessage()
- ProcessWndProcException()
- ProcessMessageFilter()
- Run()

A maneira mais fácil de gerar este objeto é utilizando o Class Wizard do Visual C++ versão 5:

1. Defina o seu projeto como uma aplicação MFC. Abra o o workspace do projeto.
2. Selecione *ClassWizard* do menu principal.
3. Clique o botão *AddClass.*
4. Escolha *New…* no menu
5. Você verá a janela de diálogo *Create New Class* mostrada abaixo.
6. Entre um novo nome para a classe como CdemoThread.
7. Selecione CwinThread na caixa *Base Class*.
8. Clique no botão *Create*.



Janela de Diálogo do Class Wizard (Visual C++ versão 5.0)

Serão gerados dois arquivos no diretório da sua aplicação: DemoThread.h e DemoThread.cpp.

A classe gerada tem o seguinte aspecto:

Fundamental para que o objeto derivado da classe CObject possa ser criado dinamicamente.

```
class CDemoThread : public CWinThread
{
        DECLARE_DYNCREATE(CDemoThread)
protected:
        CDemoThread();    // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CDemoThread)
        public:
        virtual BOOL InitInstance();
        virtual int ExitInstance();
        //}}AFX_VIRTUAL
```

Trabalha em conjunção com a macro IMPLEMENT_DYNCREATE incorporada ao programa DemoThread. cpp.

```
// Implementation
protected:
        virtual ~CDemoThread();

// Generated message map functions
//{{AFX_MSG(CDemoThread)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG

        DECLARE_MESSAGE_MAP()
};
```

Agora estamos prontos para criar a nossa thread usando AfxBeginThread.

CdemoThread* pThread = (CdemoThread*) AfxBeginThread(RUNTIME_CLASS (CDemoThread));

- MFC irá criar uma instância da classe CdemoThread em runtime.
- Uma vez criada a thread, MFC irá chamar a função InitInstance() que nós iremos prover e entrar no loop de mensagens.

Trabalhando com Objetos MFC

O mapeamento entre objetos MFC e handles Win32 é mantido em armazenamento local.

Limitações:

- Não se pode passar um objeto MFC de uma thread para outra.
- Não se pode passar um ponteiro de um objeto MFC para outra thread (CWnd, CDC, Cpen, Cbrush, Cfont, Cbitmap, Cpalette, etc.)

Motivo: seriam necessários objetos de sincronização para todos estes objetos → MFC ficaria mais lento.

Esta restrição implica que não podemos, por exemplo, passar um ponteiro para CWnd em uma estrutura que é usada por uma thread de trabalho.

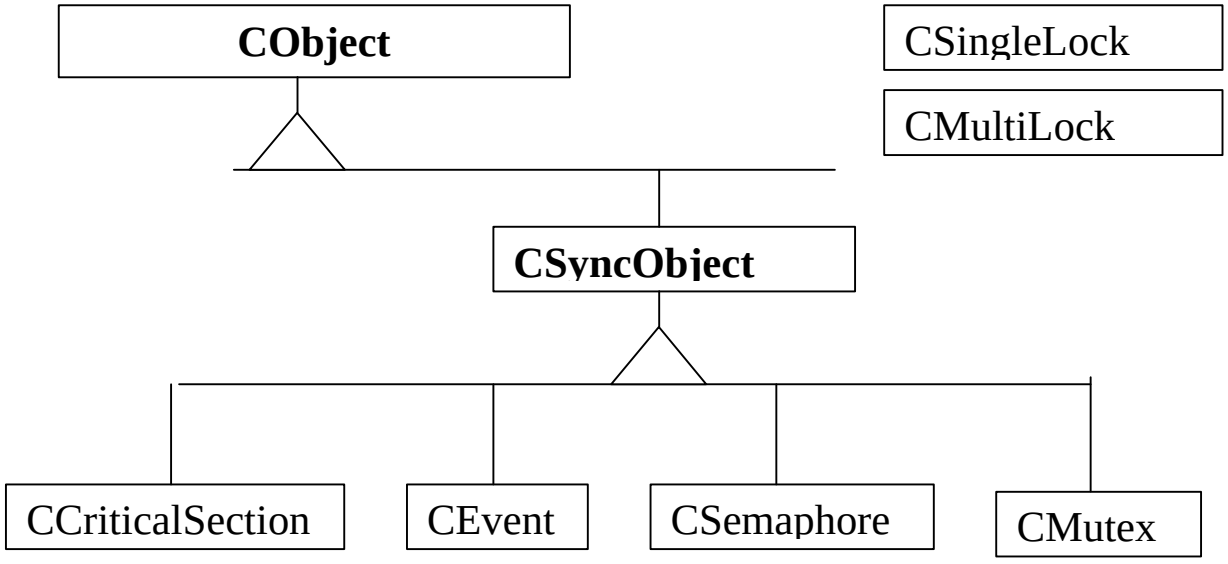
MFC irá verificar através de rotinas de defesa (Asserts) que estas restrições sejam respeitadas. A idéia de se passar o handle de um

objeto para outra thread ao invés do próprio objeto, funciona em alguns casos, mas falha em outros.

Sincronizando em MFC:

MFC possui classes que gerenciam as operações de *lock* e *unlock* dos objetos de sincronização.

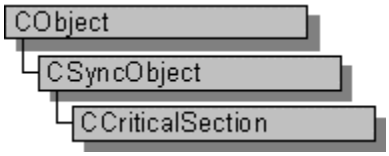
Nome do Objeto	**Nome da classe MFC**
Critical Section	CCriticalSection
Event	CEvent
Semaphore	CSemaphore
Mutex	CMutex



MFC: Diagrama de hierárquico de classes: Objetos de sincronização

Para usar os objetos de sincronização usar a diretiva: #include <afxmt.h>.
Todos os objetos possuem as funções *Lock()* e *Unlock()*.

Classe CCriticalSection



Construtor:

```
CCriticalSection();
```

Métodos:

Lock	Ganha acesso à seção crítica
Unlock	Libera acesso à seção crítica.

Lock()

```
BOOL Lock();
```

Retorno da função:

!=0	Sucesso
0	Falhou

Unlock()

```
BOOL Unlock();
```

Retorno da função:

!=0	Sucesso
0	Falhou

Exemplo de uso: Escrevendo a classe string em MFC

```
class StringV3
{
 public:
        StringV3();
```

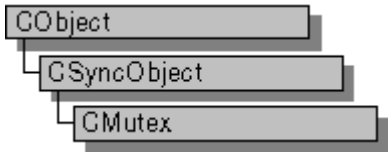
```
        virtual ~StringV3();
        virtual void Set(char* str);
        int GetLenght();
private:
        CCriticalSection m_Sync;
        char * m_pData;
};

StringV3::StringV3()
{
       // The constructor for m_Sync will have
       // already been called automatically because
       // it is a member variable.
       m_pData = NULL;
}

StringV3::~StringV3()
{
     // Use the "array delete" operator.
     // Note: "delete" checks for NULL automatically.
     m_Sync.Lock();
     delete [] m_pData;
     m_Sync.Unlock();
     // The destructor for m_Sync will be
     // called automatically.
}

void StringV3::Set(char *str)
{
     m_Sync.Lock();
     delete [] m_pData;
     m_pData = new char [::strlen(str) + 1];
     ::strcpy(m_pData, str);
     m_Sync.Unlock();
}

int StringV3::GetLenght()
{
     if (m_pData == NULL)
         return 0;
     m_Sync.Lock();
     int len = ::strlen(m_pData);
     m_Sync.Unlock();
     return len;
}
```

Problemas:

- Devemos sempre chamar *Unlock()* antes de retornar.
- Locks não são limpos quando ocorre uma exception.

As classes **CSingleLock** e **CMultiLock** resolvem o problema:

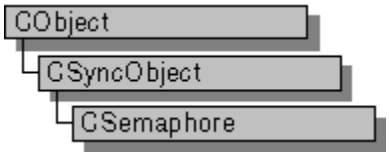
Classe CMutex



Construtor:

```
CMutex(
    BOOL bInitiallyOwn,                 //  = FALSE
                                            TRUE: Acesso ao recurso habilitado.
    LPCTSTR lpszName,                   // = NULL
                                            Nome do objeto. Necessário para usar
                                            Evento entre processos.
    LPSECURITY_ATTRIBUTES               // = NULL
    lpsaAttribute
                                            Atributo de segurança do objeto
    );
```

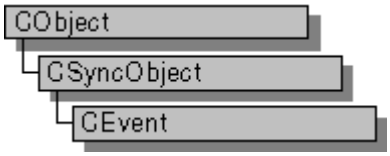
Classe CSemaphore



Construtor:

```
CSemaphore(
    LONG lInitialCount,                 //  = 1

    LONG lMaxCount,                     // = 1
```

```
                                    Valor máximo a ser atingido pelo Semáforo.
LPCTSTR lpszName,           // = NULL
                                    Nome do objeto. Necessário para usar
                                    Semáforo entre processos.
LPSECURITY_ATTRIBUTES       // = NULL
lpsaAttribute
                                    Atributo de segurança do objeto
);
```



Classe CEvent

Construtor:

```
CEvent(
```

```
BOOL bInitiallyOwn,         //  = FALSE
                                    TRUE: Acesso ao recurso habilitado.
BOOL bManualReset,          // = FALSE
                                    TRUE: Evento com reset manual.
LPCTSTR lpszName,           // = NULL
                                    Nome do objeto. Necessário para usar
                                    Evento entre processos.
LPSECURITY_ATTRIBUTES       // = NULL
lpsaAttribute
                                    Atributo de segurança do objeto
);
```

Métodos:

SetEvent	Seta o evento
ResetEvent	Reseta o evento
PulseEvent	Pulsa o evento
Unlock	Permite ao objeto de sincronismo sair de escopo.

SetEvent()

```
BOOL SetEvent();
```

Retorno da função:

!=0	Sucesso
0	Falhou

ResetEvent()

```
BOOL ResetEvent();
```

Retorno da função:

!=0	Sucesso
0	Falhou

PulseEvent()

```
BOOL PulseEvent();
```

Retorno da função:

!=0	Sucesso
0	Falhou

UnLock()

```
virtual BOOL UnLock();
```

Retorno da função:

!=0	A thread possuía o objeto do tipo evento e o evento é do tipo automático.
0	Falha

Classe CSingleLock

Construtor:

```
DWORD CSingleLock(
```

CsyncObject *pObject,	//	Ponteiro para objetos de sincronização.
BOOL bInitialLock	//	= FALSE. Especifica se deve tentar acessar inicialmente o objeto.
);		

Métodos:

IsLocked	Determina se um objeto está trancado
Lock	Espera num objeto de sincronismo
Unlock	Libera um objeto de sincronismo

IsLocked()

```
BOOL IsLocked();
```

Retorno da função:

!=0	Objeto está trancado
0	Não está trancado

Lock()

```
BOOL Lock(
```

DWORD dwTimeOut	//	= INFINITE Tempo de espera máximo para sinalização do objeto
);		

Retorno da função:

!=0	Sucesso

0	Falha

UnLock()

```
BOOL UnLock(
    LONG lCount,            // Valor máximo de acessos liberados. Se o valor
                               fizer com que o número de acessos exceda o
                               máximo permitido para o objeto, o valor não é
                               alterado e retorna FALSE.
    LPLONG lPrevCount       // = NULL
                               Endereço da variável para receber o valor
                               anterior do objeto de sincronismo.
);
```

Retorno da função:

!=0	Sucesso
0	Falha

Classe CMultiLock

Construtor:

```
DWORD CMultiLock(
    CsyncObject *ppObjects[],  // Vetor de ponteiros para  objetos de
                                  sincronização.
    DWORD dwCount,             // Número de objetos em ppObjects
    BOOL bInitialLock          // = FALSE.
                                  Especifica se deve tentar acessar inicialmente
                                  qualquer um dos objetos do vetor.
);
```

Métodos:

IsLocked	Determina se um objeto está trancado
Lock	Espera num objeto de sincronismo
Unlock	Libera um objeto de sincronismo

IsLocked()

```
BOOL IsLocked();
```

Retorno da função:

!=0	Objeto está trancado
0	Não está trancado

Lock()

```
DWORD Lock(
    DWORD dwTimeOut,        // = INFINITE
                            Tempo de espera máximo para sinalização do objeto
    BOOL WaitForAl,         // l = TRUE
                            FALSE: espera por qualquer um.
                            TRUE:  espera por todos
    DWORD dwWakeMask        // = 0
                            Máscara com outras condições que podem abortar o Wait. As condições são as mesmas da função MsgWaitForMultipleObjects
                            // Tipos de entradas do usuário a serem esperadas:
                              QS_ALLINPUT
                              QS_HOTKEY
                              QS_INPUT
                              QS_KEY
                              QS_MOUSE
                              QS_MOUSEBUTTON
                              QS_MOUSEMOVE
                              QS_PAINT
                              QS_POSTMESSGAE
                              QS_SENDMESSAGE
                            QS_TIMER
);
```

Retorno da função:

!=0	Sucesso: WAIT_OBJECT_0.. WAIT_OBJECT_0 + N –1 Índice do objeto sinalizado WAIT_OBJECT_0 + N Um evento está disponível na fila de entrada da thread WAIT_ABNDONED_0..WAIT_ABANDONED_0+N-1 índice do Mutex que abandonou a seção crítica WAIT_TIMEOUT O tempo de espera estipulado expirou
-1	Falha

U n L o c k ( )

```
BOOL UnLock(
    LONG lCount,          // Valor máximo de acessos liberados. Se o valor
                          fizer com o número de acessos exceda o máximo
                          permitido para op objeto, o valor não é alterado
                          e retorna FALSE.
    LPLONG lPrevCount     // = NULL
                          Endereço da variável para receber o valor
                          anterior do objeto de sincronismo.
);
```

Retorno da função:

!=0	Sucesso
0	Falha

Exemplo: Escrevendo a classe string com CSingleLock

```
class StringV4
{
 public:
        StringV4();
        virtual ~StringV4();
        virtual void Set(char* str);
        int GetLenght();
private:
        CSyncObject* m_pLokable;
        char * m_pData;
};

StringV4::StringV4()
{
       m_pData = NULL;
       m_pLockable = new Mutex;
}

StringV4::~StringV4()
{
     delete [] m_pData;
     delete m_pLockable;
}

void StringV4::Set(char *str)
{
     CSingleLock  localLock(m_pLockable, TRUE); // entra na S.Crítica
     delete [] m_pData;
     m_pData = NULL;       // In case new throws an exception
     m_pData = new char[::strlen(str) +1];
     ::strcpy(m_pData, str);
}

int StringV4::GetLenght()
{
     CSingleLock localLock(m_pLockable, TRUE);
     if (m_pData == NULL)
        return 0;
        return::strlen(m_pData);
}
```

Limitações de sincronização com MFC

- Não existe nenhuma versão de função alertável de *WaitForSingleObject()* e *WaitForMultipleObjects()* e assim não é possível utilizar operações de I/O assíncrono usando APCs.
- MFC não suporta a função *MsgWaitForMultipleObjects()* o que torna difícil esperar por mensagens e eventos de sincronização simultaneamente.

MsgWaitForMultipleObjects()

```
DWORD MsgWaitForMultipleObjects(
  DWORD nCount,        // número de handles a esperar, limitado por MAXIMUM_WAIT_OBJECTS.
  LPHANDLE pHandles,   // Array de handles par objetos do kernel
  BOOL fWaitAll,       // TRUE: retorna se todos os handles forem sinalizados.
                       // FALSE: retorna se qualquer handle for sinalizado
  DWORD dwMiliseconds, // Tempo máximo que desejamos esperar
  DWORD dwWakeMask     // Tipos de entradas do usuário a serem esperadas:
                       //   QS_ALLINPUT
                       //   QS_HOTKEY
                       //   QS_INPUT
                       //   QS_KEY
                       //   QS_MOUSE
                       //   QS_MOUSEBUTTON
                       //   QS_MOUSEMOVE
                       //   QS_PAINT
                       //   QS_POSTMESSGAE
                       //   QS_SENDMESSAGE
                       //   QS_TIMER
);
```

Retorno da função:

bWaitAll	Status	Interpretação
TRUE FALSE	WAIT_OBJECT_0 Valor	Todas threads retornaram Valor-WAIT_OBJECT_0= índice da thread que retornou
	WAIT_ABANDONED_0 + x	Uma thread proprietária de um Mutex o abandona sem liberá-lo x é o índice da thread.
	WAIT_OBJECT_0 + nCount	Mensagem foi inserida na fila
	WAIT_TIMEOUT	Ocorreu *timeout*
	WAIT_FAILED	Função falhou

Suportando MsgWaitForMultipleObjects() em MFC

É possível fazer uma aplicação ficar à espera de mensagens ou da sinalização de objetos do kernel. A função responsável por isto é *MsgWaitFor MultipleObjects()*. A dificuldade reside no fato do MFC executar o loop de espera por mensagens.

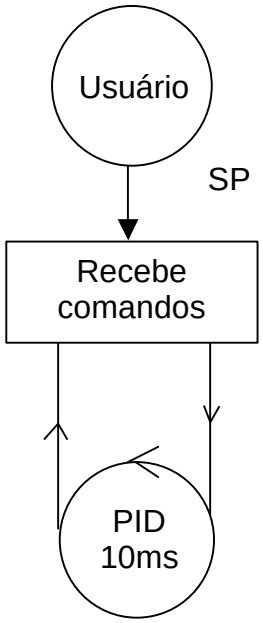
- Uma solução é disparar um thread que ficará a espera do evento de sinalização do objeto e depois enviará uma mensagem à thread que espera pela mensagem. O inconveniente é o acréscimo de overhead.

- A segunda solução seria rescrever o loop de espera por mensagens do MFC. O loop de espera por mensagens reside nas funções virtuais *CwinThread::Run()* e *CwinThread::PumpMessage()*, cujos fontes estão em THRDCORE.CPP em MFC\SRC. Isto é inconveniente por uma série de razões: dificuldade de manter o código quando Microsoft modifica versões, parte do código é não documentado pela Microsoft, dependência de versões, etc.

Exercícios:

1. Um processo executa um loop de controle com as funções de controlador PID. O algoritmo de controle deve ser executado uma vez a cada 10 ms. Este mesmo processo deve aceitar entradas de dados dos usuário para modificação do valor do *set-point* de forma assíncrona.

   Desenvolva este programa em ambiente Windows NT, utilizando duas threads e MFC. O *set-point* será definido pelo usuário através de uma janela de controle. A GUI thread deve esperar mensagens do usuário e eventos do timer, sem poder usar a função *MsgWaitForMultipleObjects().*



2. Modifique o exemplo para demonstrar a comunicação entre threads, apresentado na página 13 do arquivo DLL.doc, para que a thread secundária de serviço receba todas as ordens da thread primária, ou um evento do timer através da função *MsgWaitForMultipleObjects().*