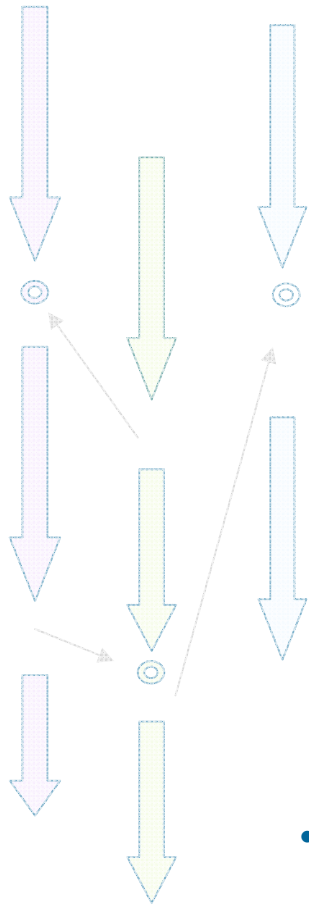


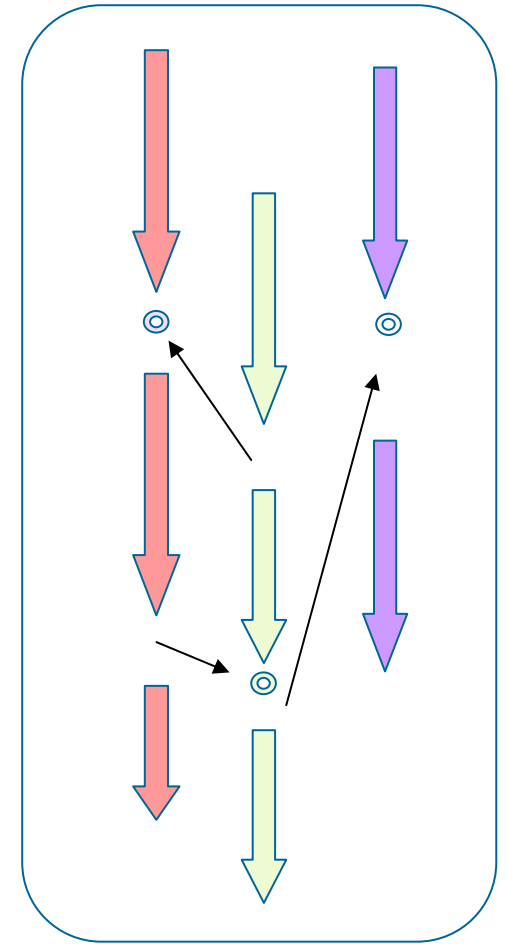
# Mecanismos de Sincronismo da Win32



- Jeffrey Richter, Christophe Nasarre, *Windows via C/C++*, Fifth Edition, Microsoft Press, 2008 [cap. 8 e 9]
- Microsoft, **Microsoft Developer's Network** (MSDN)

# Mecanismos de Sincronismo da Win32

- Critical Sections
- Mutexes
- Semaphores
- Events
- Timers



# Critical Sections

Decl. de uma secção crítica :

```
CRITICAL_SECTION csvar;
```

Funções de manipulação :

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCS );
```

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCS );
```

```
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCS );
```

```
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCS );
```

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

Características :

- *Scope local* ao processo
- *Owner*
- *Ownership count*

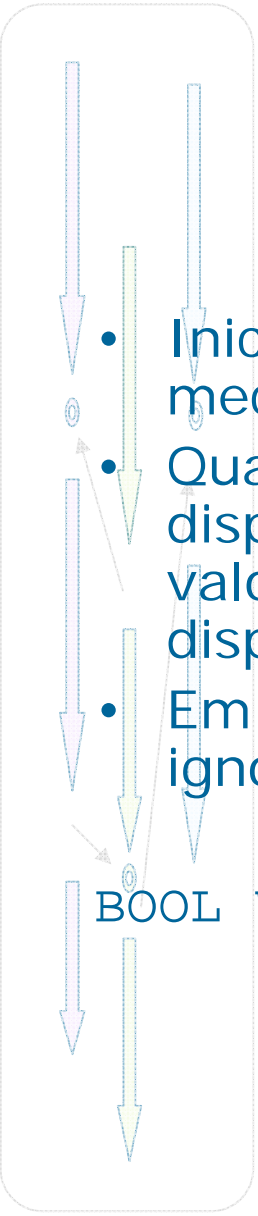


# As várias noções

- **Scope local e global (ao processo)**
  - Entende-se por *scope* local quando o mecanismo só pode ser utilizado dentro de um processo. E de *scope* global quando o mecanismo pode ser utilizado globalmente no sistema, ou seja pode ser utilizado por vários processos
- **Owner (noção de dono - posse da CS)**
  - O mecanismo regista qual a *thread* que tem a posse da CS, só deixando que seja ela a fazer "*Leave...*"
- **Ownership count**
  - O mecanismo permite que a *thread* que tem a posse da CS, possa adquiri-la mais vezes. Como regista o n°. de vezes que a CS foi adquirida ("*Enter...*"), a *thread* terá que fazer igual n°. de "*Leave...*" para que a CS fique livre,



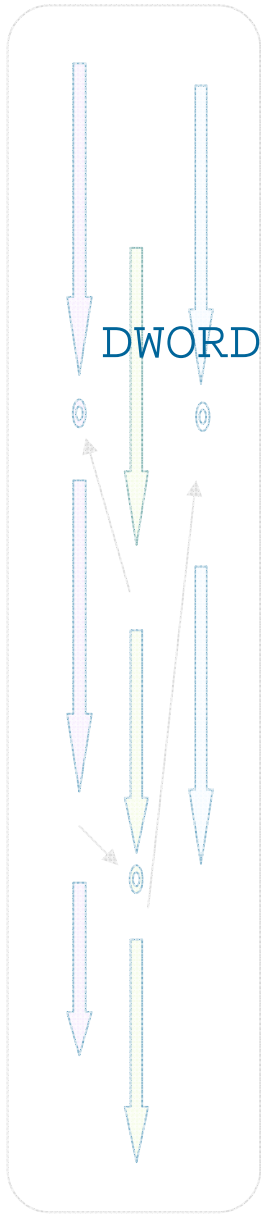
# Critical Sections

- 
- Inicia a critical section e o valor do spin count associado com o mecanismo
  - Quando uma thread tenta adquirir uma critical section não disponível a thread inicia um ciclo de espera activa (durante o valor spin count). Se entretanto a critical section não ficar disponível a thread bloqueia-se num evento do kernel
  - Em arquitecturas mono-processador o valor do spin count é ignorado

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



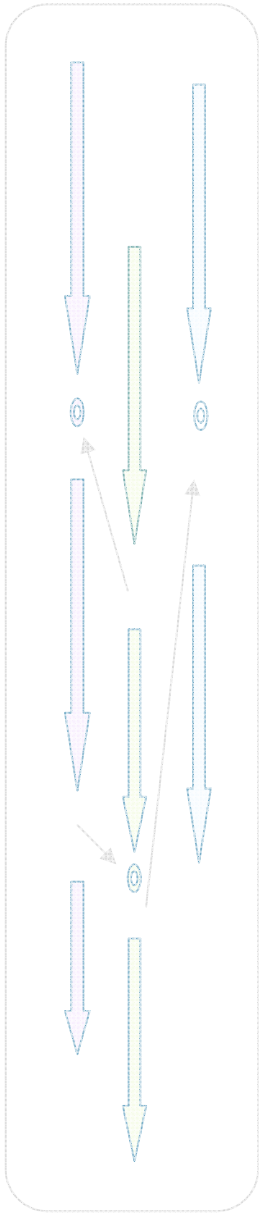
# Critical Sections



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



# Exemplo de Utilização de Critical Sections



```
class CsImpressora : public Impressora {  
    CRITICAL_SECTION csImp;  
public:  
    CsImpressora(bool InitOwner = false) {  
        InitializeCriticalSection(&csImp);  
        if (initOwner) adquirir(); }  
  
    void adquirir() { EnterCriticalSection(&csImp); };  
  
    void libertar() { LeaveCriticalSection(&csImp); };  
  
    ~CsImpressora() { DeleteCriticalSection(&csImp); };  
};
```



# Exemplo com garantia de exclusão mútua utilizando *Critical Sections*

```
DWORD WINAPI IncFunc(LPVOID args)
```

```
{  
    int tmp;  
    for(int i=0; i<50; i++){
```

```
        EnterCriticalSection(&CriticalSection);
```

```
        /* Inicio da região critica */  
        tmp = x;  
        tmp++;  
        x = tmp;  
        /* Fim da região critica */
```

```
        LeaveCriticalSection(&CriticalSection);
```

```
    return 0;  
}
```

C:\WINDOWS\system32\cmd.exe

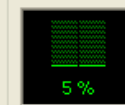
Terminei com o valor de x = 100  
Press any key to continue . . .

Windows Task Manager

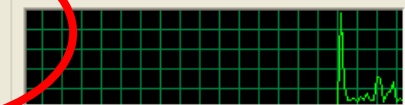
File Options View Help

Applications Processes Performance Networking

CPU Usage



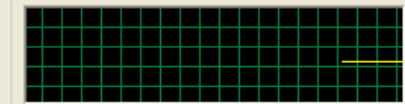
CPU Usage History



PF Usage



Page File Usage History



Totals

Handles	12906
Threads	519
Processes	46

Physical Memory (K)

Total	490992
Available	51692
System Cache	118848

Commit Charge (K)

Total	520588
Limit	1148916
Peak	535880

Kernel Memory (K)

Total	46460
Paged	36756
Nonpaged	9704

Processes: 46

CPU Usage: 5%

Commit Charge: 508M / 1121M





# Exemplo com garantia de exclusão mútua utilizando *Critical Sections*

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

```
int x;
CRITICAL_SECTION cs;
```

```
DWORD WINAPI IncFunc(LPVOID args)
```

```
{
    int tmp;
    for(int i=0; i<50; i++){
        EnterCriticalSection(&cs);
        /* Inicio da região critica */
        tmp = x;
        tmp++;
        x = tmp;
        /* Fim da região critica */
        LeaveCriticalSection(&cs);
    }
    return 0;
}
```

Declarar e iniciar a *Critical Section*

Reservar e libertar a exclusão mútua

```
int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE ht, ht2;
    DWORD threadId, threadId2;
    x = 0;
```

```
InitializeCriticalSection(&cs);
```

```
// Criar as duas tarefas
```

```
ht = CreateThread ( NULL, 0, IncFunc
                    , NULL, NULL, &threadId);
ht2 = CreateThread ( NULL, 0, IncFunc,
                    NULL, NULL, &threadId2);
```

```
//Esperar a terminação das tarefas
```

```
WaitForSingleObject(ht, INFINITE);
WaitForSingleObject(ht2, INFINITE);
_tprintf(TEXT("Terminei com o valor de x
= %d\n"), x);
```

```
DeleteCriticalSection(&cs);
```

```
return 0;
```

```
}
```



# Sincronismo em Objectos do *Kernel* (revisão)

Object states : **signaled, non-signaled**

Wait on object : wait until object becomes signaled

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

```
DWORD WaitForMultipleObjects( DWORD nCount, CONST HANDLE *lpHandles,  
                                BOOL bWaitAll, DWORD dwMilliseconds );
```

Valores de retorno :

For Single W , For Multiple Wait

**WAIT\_OBJECT\_0** , **WAIT\_OBJECT\_0** to (WAIT\_OBJECT\_0 + nCount - 1)

WAIT\_ABANDONED , WAIT\_ABANDONED\_0 to (WAIT\_ABANDONED\_0 + nCount - 1)

WAIT\_TIMEOUT , WAIT\_TIMEOUT

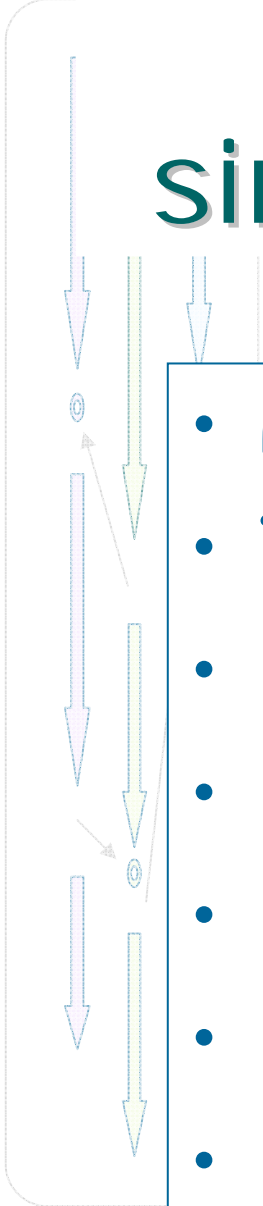
WAIT\_FAILED

dwMilliseconds : 0, Milliseconds or INFINITE

Número máximo de handles : MAXIMUM\_WAIT\_OBJECTS



# Objectos *Kernel* para sincronismo e seu significado

- 
- **Process** ( signaled when last thread terminated )
  - **Thread** ( sig. when terminated )
  - **Mutex** ( sig. on Mutex available )
  - **Semaphore** ( sig when count > 0 )
  - **Event** ( sig. when event is set )
  - **File** (sig. when I/O operation completes )
  - **Timer** (sig. when times expires or set time arrives)



# Mutex

## Funções de manipulação :

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES lpMutexAttributes,  
                    BOOL bInitialOwner, LPCTSTR lpName );
```

```
HANDLE OpenMutex( DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName );
```

```
Adquirir Mutex : Wait...(. . . );
```

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

```
BOOL CloseHandle(HANDLE hMutex );
```

## Características :

- *Scope* global
- *Owner*
- *Ownership count*

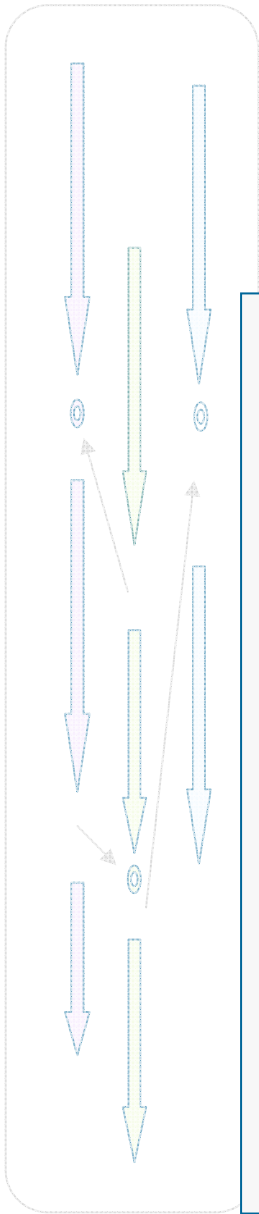
TRUE: Thread became the owner, non-signaled  
FALSE: signaled

Return :  
Handle(**GetLastError**()=ERROR\_ALREADY\_EXISTS)  
or NULL

dwDesiredAccess = MUTEX\_ALL\_ACCESS



# Utilização de *Mutex*



```
class MutImpressora : public Impressora {  
    HANDLE handle;  
public:  
    MutImpressora(bool InitialOwner = false) {  
        handle = CreateMutex( NULL, InitialOwner, NULL );    };  
  
    void adquirir() { WaitForSingleObject(handle, INFINITE);    };  
  
    void libertar() { ReleaseMutex(handle);    };  
  
    ~MutImpressora() { CloseHandle(handle);    };  
};
```



# Exemplo com garantia de exclusão mútua utilizando *MUTEX*

```
DWORD WINAPI IncFunc(LPVOID args)
```

```
{  
    int tmp;  
    for(int i=0; i<50; i++){
```

```
        WaitForSingleObject(hMutex, INFINITE);
```

```
        /* Inicio da região critica */  
        tmp = x;  
        tmp++;  
        x = tmp;  
        /* Fim da região critica */
```

```
        ReleaseMutex(hMutex);
```

```
    return 0;  
}
```

C:\WINDOWS\system32\cmd.exe

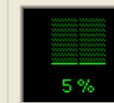
Terminei com o valor de x = 100  
Press any key to continue . . .

Windows Task Manager

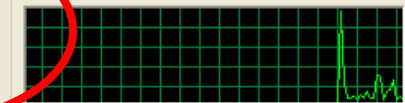
File Options View Help

Applications Processes Performance Networking

CPU Usage



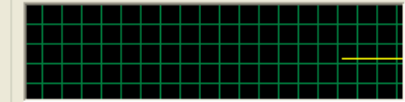
CPU Usage History



PF Usage



Page File Usage History



Totals

Handles	12906
Threads	519
Processes	46

Physical Memory (K)

Total	490992
Available	51692
System Cache	118848

Commit Charge (K)

Total	520588
Limit	1148916
Peak	535880

Kernel Memory (K)

Total	46460
Paged	36756
Nonpaged	9704

Processes: 46

CPU Usage: 5%

Commit Charge: 508M / 1121M



# Exemplo com garantia de exclusão mútua utilizando *MUTEX*

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

```
int x;
HANDLE hMutex;
```

```
DWORD WINAPI IncFunc(LPVOID args)
{
    int tmp;
    for(int i=0; i<50; i++)
    {
        WaitForSingleObject(hMutex, INFINITE);
        /* Inicio da região critica */
        tmp = x;
        tmp++;
        x = tmp;
        /* Fim da região critica */
        ReleaseMutex(hMutex);
    }
    return 0;
}
```

Declarar e iniciar o *Mutex*

Reservar e libertar o *Mutex*

```
int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE ht, ht2;
    DWORD threadId, threadId2;
    x = 0;

    hMutex = CreateMutex(NULL, FALSE, NULL);
    if(hMutex == NULL){
        _tprintf(TEXT("Erro na criação do mutex\n"));
        ExitProcess(0);
    }
    // Criar as duas tarefas
    ht = CreateThread ( NULL, 0, IncFunc
                        , NULL, NULL, &threadId);
    ht2 = CreateThread ( NULL, 0, IncFunc
                        , NULL, NULL, &threadId2);

    //Esperar a terminação das tarefas
    WaitForSingleObject(ht, INFINITE);
    WaitForSingleObject(ht2, INFINITE);
    _tprintf(TEXT("Terminei com o valor de x = %d\n"), x);
    CloseHandle(hMutex);
    return 0;
}
```



# Semaphores

Count > 0, signaled  
Count = 0, non-signal

Características :

- *Scope* global
- Contador
- Máximo

Funções de manipulação :

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
                        LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName );
```

```
HANDLE OpenSemaphore(DWORD dwDesiredAccess,  
                     BOOL bInheritHandle, LPCTSTR lpName );
```

Adquirir Semaphore : **Wait...**(. . . );

dwDesiredAccess = SEMAPHORE\_ALL\_ACCESS

```
BOOL ReleaseSemaphore( HANDLE hSemaphore,  
                      LONG lReleaseCount, LPLONG lpPreviousCount );
```

```
BOOL CloseHandle(HANDLE hSemaphore );
```

```
if (count+ReleaseCount > max) return FALSE;
```





# Utilização de Semáforos

```
class SemImpressora : public Impressora {
    HANDLE handle;
public:
    SemImpressora(bool initialOwner = false) {
        int initValue = initialOwner ? 0 : 1;
        handle = CreateSemaphore( NULL, initValue,1, NULL ); };

    void adquirir() { WaitForSingleObject(handle, INFINITE); };

    void libertar() { ReleaseSemaphore(handle, 1, NULL); };

    ~SemImpressora() { CloseHandle(handle); };
};
```



# A Classe Semáforo – Utilizada nos exercícios anteriores

```
class Semaforo {
private: HANDLE hSemaforo;
public:
    Semaforo(int numUnidades=1, int maxUnidades=MAXLONG, char * nomeSem=NULL) {
        if ((hSemaforo = CreateSemaphore(NULL,numUnidades,maxUnidades,nomeSem))==NULL )
            FatalErrorSystem("Erro ao criar o semaforo(1)");
    }
    Semaforo(char * nomeSem) {
        if ( (hSemaforo = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, nomeSem)) == NULL )
            FatalErrorSystem("Erro ao criar o semaforo(3)");
    }
    ~Semaforo() {
        if (CloseHandle(hSemaforo)==0) FatalErrorSystem("Erro ao fechar o semaforo");
    }
    void P() {
        if ( WaitForSingleObject(hSemaforo, INFINITE) == WAIT_FAILED )
            FatalErrorSystem("Erro na operação de Wait do semáforo");
    }
    void V() {
        if ( ReleaseSemaphore(hSemaforo, 1, NULL) == 0 )
            FatalErrorSystem("Erro na operação de Signal do semáforo");
    }
    void Wait() { P(); }    void Signal() { V(); }
};
```



# Exemplo com garantia de exclusão mútua utilizando *Semaphore*

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

```
int x;
```

```
HANDLE hSem;
```

```
DWORD WINAPI IncFunc(LPVOID args)
```

```
{
```

```
    int tmp;
```

```
    for(int i=0; i<50; i++)
```

```
    {
```

```
        WaitForSingleObject(hSem, INFINITE);
```

```
        /* Inicio da região critica */
```

```
        tmp = x;
```

```
        tmp++;
```

```
        x = tmp;
```

```
        /* Fim da região critica */
```

```
        ReleaseSemaphore(hSem, 1, NULL);
```

```
    }
```

```
    return 0;
```

```
}
```

Declarar e iniciar o *Semaphore*

Reservar e libertar o *Semaphore*

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

```
    HANDLE ht, ht2;
```

```
    DWORD threadId, threadId2;
```

```
    x = 0;
```

```
    hSem = CreateSemaphore(NULL, 1, 1, NULL);
```

```
    if(hSem == NULL){
```

```
        _tprintf(TEXT("Erro na criação do Semaphore\n"));
```

```
        ExitProcess(0);
```

```
    }
```

```
    // Criar as duas tarefas
```

```
    ht = CreateThread ( NULL, 0, IncFunc
                        , NULL, NULL, &threadId);
```

```
    ht2 = CreateThread ( NULL, 0, IncFunc
                        , NULL, NULL, &threadId2);
```

```
    //Esperar a terminação das tarefas
```

```
    WaitForSingleObject(ht, INFINITE);
```

```
    WaitForSingleObject(ht2, INFINITE);
```

```
    _tprintf(TEXT("Terminei com o valor de x = %d\n"), x);
```

```
    CloseHandle(&hSem);
```

```
    return 0;
```

```
}
```



# Events

## Funções de manipulação :

```
HANDLE CreateEvent( LPSECURITY_ATTRIBUTES lpEventAttributes,  
                    BOOL bManualReset, BOOL bInitialState, LPCTSTR lpName );
```

```
HANDLE OpenEvent( DWORD dwAccess, BOOL bInheritHandle, LPCTSTR lpName );
```

```
Esperar até sinalização : Wait...(. . . );
```

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

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

```
BOOL PulseEvent( HANDLE hEvent ); //SR
```

```
BOOL CloseHandle(HANDLE hEvent );
```

## Características :

- *Scope* global
- *Reset*: Manual / Auto

```
dwDesiredAccess = EVENT_ALL_ACCESS
```

<b>Value</b>	: <b>TRUE</b>	, <b>FALSE</b>
<b>bManualReset</b>	: Manual Reset,	Auto-reset
<b>bInitialState</b>	: Signaled	, non-sig

**Manual Reset** for sinc **multiple** threads  
**Auto Reset** for **one** thread



# Auto e manual *reset*

Auto e manual *reset* – define como o *reset* ao evento é realizado

- **Auto-reset**

- Quando o evento fica sinalizado, assim que a primeira *thread* passar pelo evento (Wait...), ocorre um *reset* automático

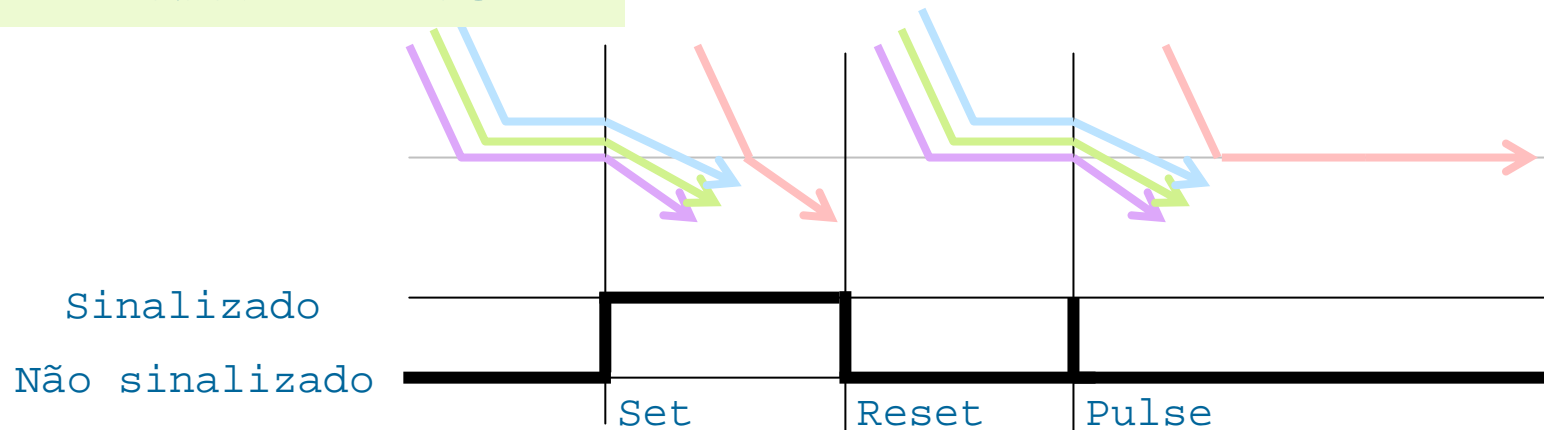
- **Manual-reset**

- O evento não altera o seu estado. Uma vez posto em sinalizado, toda a *thread* já em espera ou que venha a fazer um “Wait...” seguirá sem qualquer bloqueio e sem alterar o estado do evento.



# Comportamento dos *Events*

## Evento de *Reset* MANUAL



## Evento de *Reset* AUTOMÁTICO



# Utilização de Eventos

```
class EveImpressora : public Impressora {  
    HANDLE handle;  
  
public:  
    EveImpressora(bool initialOwner = false) {  
        // Evento automático  
        handle=CreateEvent(NULL,FALSE,initialOwner?FALSE:TRUE,NULL);};  
  
    void adquirir() { WaitForSingleObject(handle, INFINITE); };  
  
    void libertar() { SetEvent(handle); };  
  
    ~EveImpressora() { CloseHandle(handle); };  
};
```

Dependendo da configuração pode ser utilizado com uma semântica semelhante à do *mutex*



# Exemplo de Utilização de Eventos

```
#include <stdio.h>
#include <tchar.h>
#include <windows.h>
int x;
CRITICAL_SECTION cs;
HANDLE hEventoDeArranque;

DWORD WINAPI IncFunc(LPVOID args)
{
    int tmp;

    // esperar pela ordem de arranque
    _tprintf(TEXT("Esperando ordem de
    arranque.\n"));
    WaitForSingleObject(hEventoDeArranque,
        INFINITE);
    _tprintf(TEXT("Vou começar...\n"));

    for(int i=0; i<50; i++)
    {
        EnterCriticalSection(&cs);
        /* Inicio da região critica */
        tmp = x;
        tmp++;
        x = tmp;
        /* Fim da região critica */
        LeaveCriticalSection(&cs);
    }
    return 0;
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    HANDLE ht, ht2;
    DWORD threadId, threadId2;
    x = 0;
    InitializeCriticalSection(&cs);
    hEventoDeArranque = CreateEvent(NULL, TRUE, FALSE, NULL);
    if(hEventoDeArranque == NULL){
        _tprintf(TEXT("Erro na criação do Evento.\n"));
        ExitProcess(0);
    }

    // Criar as duas tarefas
    ht = CreateThread ( NULL, 0, IncFunc, NULL, NULL,
        &threadId);
    ht2 = CreateThread ( NULL, 0, IncFunc, NULL, NULL,
        &threadId2);

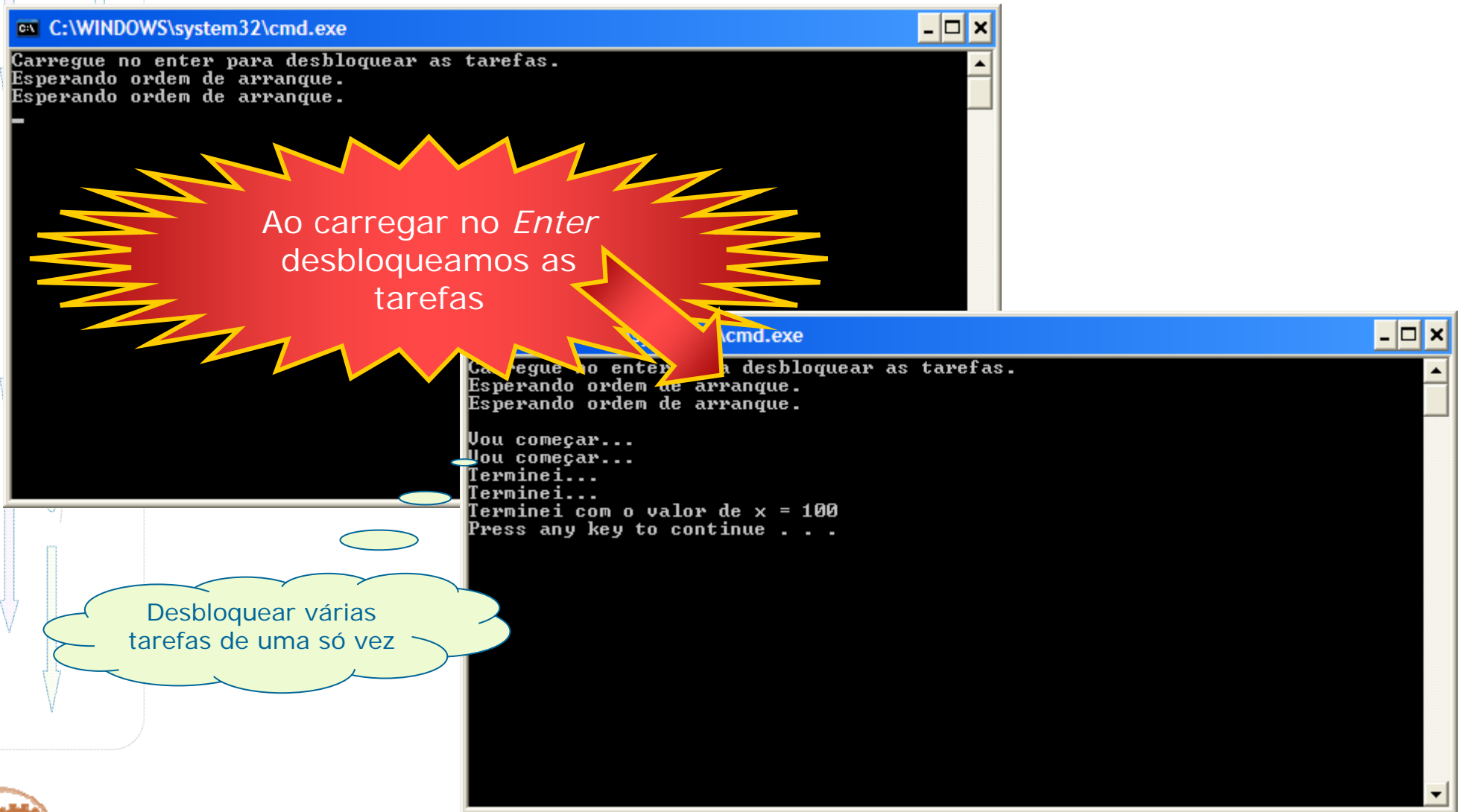
    _tprintf(TEXT("Carregue no enter para desbloquear as
    tarefas.\n"));
    _tscanf(TEXT("%c"), &c);
    SetEvent(hEventoDeArranque);

    //Esperar a terminação das tarefas
    WaitForSingleObject(ht, INFINITE);
    WaitForSingleObject(ht2, INFINITE);
    _tprintf(TEXT("Terminei com o valor de x = %d\n"), x);
    DeleteCriticalSection(&cs);
    CloseHandle(hEventoDeArranque);
    return 0;
}
```





# Exemplo de Utilização de Eventos



# Waitable Timers

## Características :

- *Scope global*
- *Manual / Auto - reset*
- *Periodic / one tick - timer*

## Funções de manipulação :

```
HANDLE CreateWaitableTimer( LPSECURITY_ATTRIBUTES lpTimerAttributes ,  
                             BOOL bManualReset, LPCTSTR lpName );
```

```
HANDLE OpenWaitableTimer(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);
```

```
BOOL SetWaitableTimer(  
    HANDLE htimer,                // handle to a timer object  
    const LARGE_INTEGER *pduetime, // when timer will become signaled  
    LONG lperiod,                 // periodic timer interval  
    PTIMERAPCROUTINE pfnCompletionRoutine, // completion routine  
    LPVOID lpArgToCompletionRoutine, // data for completion routine  
    BOOL fResume                  // flag for resume state  
);
```

Faz reset  
ao timer

```
BOOL CancelWaitableTimer( HANDLE htimer);
```

Não altera o estado  
de sincronismo

< 0, relative time  
> 0, absolute time

```
Wait for timer : Wait...(. . . );
```

```
BOOL CloseHandle(HANDLE hMutex );
```

Windows 98, Windows NT 4.0 and later (MSDN)



# Waitable Timer

- São objectos de sincronismo que se auto-sinalizam:
  - Uma única vez (***one-tick timer***) [period = 0]
  - Em intervalos de tempo regulares (***periodic timer***) [period >0]
- Quanto ao *reset*, podem ser do tipo:
  - ***manual reset***: em que o *reset* tem de ser explícito
  - ***Auto-reset***: assim que uma *thread* concluir o seu "*Wait...*" o *timer* ficará automaticamente não sinalizado
- Os *waitable timer* são sempre criados no estado não sinalizado e inactivos.
  - Para activar o *timer* existe a função: **SetWaitableTimer**
  - Para desactivar existe a função: **CancelWaitableTimer**



# Programação de um *Waitable Timer* com tempo relativo

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <Mmsystem.h>
const int nTimerUnitsPerSecond=10000000; // 10.000.000 unidades de 100 nanosegundos

int APIENTRY _tWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine, int nCmdShow ) {

    HANDLE hTimer;  LARGE_INTEGER li;

    hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
    li.QuadPart = -(10 * nTimerUnitsPerSecond); //(-)tempo relativo ao SetWaitableTimer
    SetWaitableTimer(hTimer, &li, 10*1000, NULL, NULL, FALSE);
    for (int i=0; i < 3; i++) {
        WaitForSingleObject(hTimer, INFINITE);
        PlaySound(TEXT("FSO.WAV"), NULL, SND_FILENAME);
    }
    CancelWaitableTimer(hTimer);
    CloseHandle(hTimer);
    return 0;
} // end WinMain
```

Para despertar em 10 seg. e com período de 10 seg.



# Large integers

**LARGE\_INTEGER** – é uma union

```
typedef union union {  
    struct {  
        DWORD LowPart;  
        LONG HighPart;  
    };  
    LONGLONG QuadPart;  
} LARGE_INTEGER, *PLARGE_INTEGER;
```

Várias utilizações

```
LARGE_INTEGER li, lgStartTime; FILETIME ftUTC;
```

```
-----  
li.LowPart  = ftUTC.dwLowDateTime;  
li.HighPart = ftUTC.dwHighDateTime;  
-----
```

```
li.QuadPart = -(10 * nTimerUnitsPerSecond);  
-----
```

```
li.QuadPart = -1*10000*(__int64)startTime_ms;
```



# Prog. de um W. Timer com uma Data/Hora absoluta

```
HANDLE hTimer; SYSTEMTIME st;
FILETIME ftLocal, ftUTC;  LARGE_INTEGER liUTC;

hTimer = CreateWaitableTimer(NULL, FALSE, NULL); // Cria um auto-reset timer

// A primeira sinalização ocorre em 1 Janeiro 2002 à 1:00 P.M. (local time).
st.wYear      = 2002; // Ano
st.wMonth     = 1;    // Janeiro
st.wDayOfWeek = 0;    // ignorado
st.wDay       = 1;    // dia 1
st.wHour      = 13;   // 13 horas
st.wMinute    = 0;    // 0 minutos
st.wSecond    = 0;    // 0 segundos
st.wMilliseconds = 0; // 0 milissegundos

SystemTimeToFileTime(&st, &ftLocal); // converte system time em tipo FILETIME
LocalFileTimeToFileTime(&ftLocal, &ftUTC); //converte hora local time para UTC time
// Convert FILETIME to LARGE_INTEGER devido a diferentes alinhamentos de memória
liUTC.LowPart  = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;
SetWaitableTimer(hTimer, &liUTC, 24*60*60*1000, NULL,NULL,FALSE); // Activa o timer
```



# Classe "Timer"

Uma interface mais amigável para uso de Timers

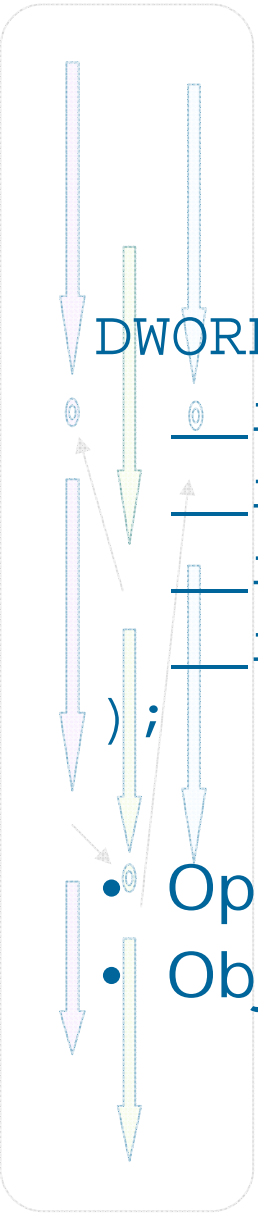
```
class Timer {  
    HANDLE    handle;    // Handler do timer, usado no wait  
public:  
    Timer(bool manual=false) {  
        handle = CreateWaitableTimer(NULL, manual, NULL); }  
  
    void SetTimer(LONG startTime_ms, LONG period_ms) {  
        LARGE_INTEGER lgStartTime; // 100 nanosecs resolution  
        lgStartTime.QuadPart = -1*10000*(__int64)startTime_ms;  
        BOOL res = SetWaitableTimer( handle, &lgStartTime, period_ms,  
                                     NULL, NULL, FALSE );    };  
  
    void CancelTimer() { CancelWaitableTimer(handle); };  
    void WaitUntilTimer() { WaitForSingleObject(handle, INFINITE); };  
    HANDLE GetHandle () { return handle; }; // permite fazer "Wait..."  
    ~Timer() { CloseHandle(handle); };  
};
```

Colocar nas *properties* do projecto:

C/C++→Preprocessor→Preprocessor Definitions→WIN32\_WINNT=0x0400



# SignalObjectAndWait



```
DWORD WINAPI SignalObjectAndWait(  
    __in HANDLE hObjectToSignal,  
    __in HANDLE hObjectToWaitOn,  
    __in DWORD dwMilliseconds,  
    __in BOOL bAlertable  
);
```

- Operação de signal e wait atómica
- Objectos a sinalizar: semaphore, mutex, ou event





# Funções *Interlock*

- A família de funções *interlocked* proporciona um mecanismo simples para a sincronização no acesso a uma variável partilhada por múltiplas tarefas
- Tarefas de processos diferentes podem utilizar este mecanismo se a variável residir numa zona de memória partilhada

```
LONG InterlockedIncrement( LONG volatile * Addend );
```

```
LONG InterlockedDecrement( LONG volatile * Addend );
```

```
LONG InterlockedExchange( LONG volatile * Target, LONG Value );
```

```
LONGLONG InterlockedIncrement64( LONGLONG volatile *Addend );
```

```
LONGLONG InterlockedDecrement64( LONGLONG volatile * Addend );
```

```
LONGLONG InterlockedExchange64( LONGLONG volatile* Target, LONGLONG Value );
```


...

*Ver MSDN para outras funções*



# volatile

- O qualificador de tipo **volatile** declara que o valor da variável pode ser alterado num contexto diferente daquele onde aparece, i.e., alterado no contexto de múltiplas tarefas




```
volatile LONG g_fResourceInUse = FALSE;

DWORD WINAPI ThFunc (LPVOID args)
{
    ...
    while (InterlockedExchange(&g_fResourceInUse, TRUE) == TRUE)
        ;
    ++x;
    InterlockedExchange(&g_fResourceInUse, FALSE);
    ...
}
```



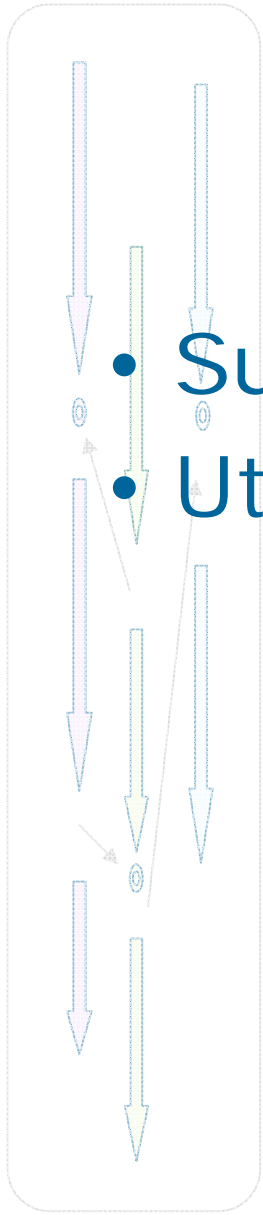
# Condition Variables

- 
- As variáveis de condição podem ser utilizadas para bloquear, atomicamente, uma tarefa até uma determinada condição ser verdadeira
  - Estão sempre associadas a um mecanismo de exclusão
  - A expressão da condição é sempre avaliada em exclusão através do mecanismo de exclusão
    - Se for falsa a tarefa bloqueia na variável de condição e automaticamente liberta a exclusão ficando à espera que a expressão seja verdadeira
    - Quando outra tarefa altera o estado da condição sinaliza a/as tarefas bloqueadas na variável de condição, desbloqueando-as sendo a exclusão adquirida automaticamente para essas tarefas
  - Mecanismo suportado na API de PThreads (norma POSIX para threads) e na WIN32 API (para versões do SO superiores ao Windows vista)



# NT6\* *Condition Variables*

- Suportado em modo utilizador
- Utiliza critical sections ou SRW locks



# NT6 *Condition Variables*



```
CONDITION_VARIABLE conditionVariable;
```

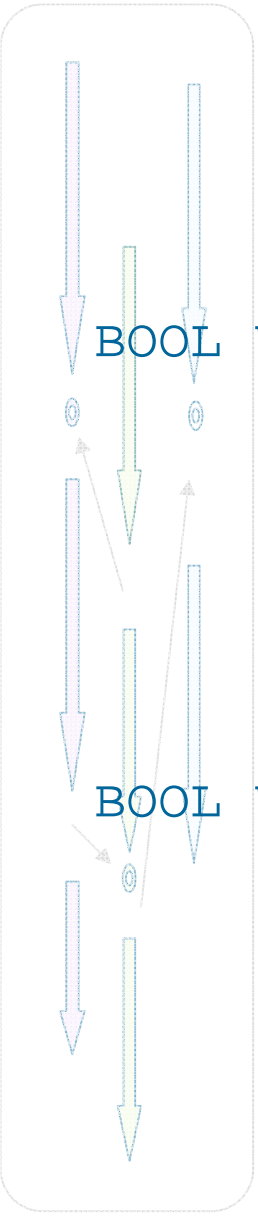
```
VOID WINAPI InitializeConditionVariable (  
    PCONDITION_VARIABLE ConditionVariable  
);
```

```
VOID WINAPI WakeConditionVariable (  
    PCONDITION_VARIABLE ConditionVariable  
);
```

```
VOID WINAPI WakeAllConditionVariable (  
    PCONDITION_VARIABLE ConditionVariable  
);
```



# NT6 *Condition Variables*

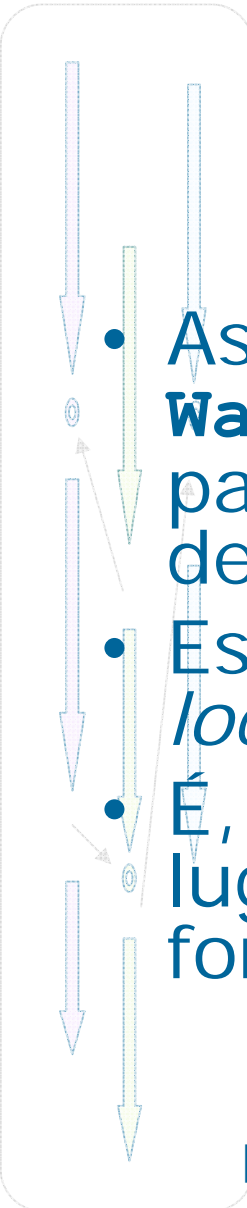


```
BOOL WINAPI SleepConditionVariableCS (  
    PCONDITION_VARIABLE ConditionVariable,  
    PCRITICAL_SECTION CriticalSection,  
    DWORD dwMilliseconds  
);
```

```
BOOL WINAPI SleepConditionVariableSRW (  
    PCONDITION_VARIABLE ConditionVariable,  
    PSRWLOCK SRWLock,  
    DWORD dwMilliseconds,  
    ULONG Flags );
```



# NT6 *Condition Variables*

- 
- As funções **WakeConditionVariable** e **WakeAllConditionVariable** são utilizadas para acordar threads bloqueadas na variável de condição
  - Estas podem ser utilizadas dentro ou fora do *lock* associado à variável de condição
  - É, geralmente, melhor libertar em primeiro lugar o *lock* antes de acordar as threads de forma a reduzir o número de *context switches*

[Ver msdn: [http://msdn.microsoft.com/en-us/library/ms682052\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682052(VS.85).aspx)]



# NT6 CV – Ex Leitores/Escritores prioridade aos escritores

```
class ReadersWritersPWrNT6Conditions:
    public ReadersWritersAccess {
private:
    int nRdInReading;

    int nWrInWriting;
    int nWrInWaiting;

    CRITICAL_SECTION csLock;

    CONDITION_VARIABLE cWaitToWrite;

    CONDITION_VARIABLE cWaitToRead;

public:
    ReadersWritersPWrNT6Conditions() {
        nRdInReading = 0;

        nWrInWriting = 0;
        nWrInWaiting = 0;

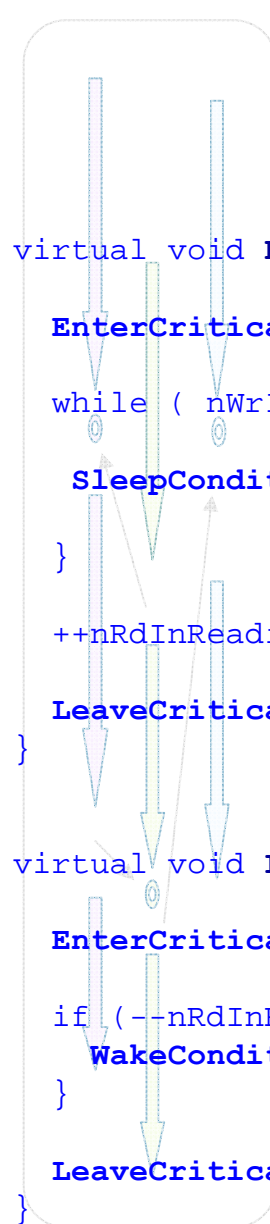
        InitializeCriticalSectionAndSpinCount(&csLock, 4000);
        InitializeConditionVariable(&cWaitToWrite);
        InitializeConditionVariable(&cWaitToRead);
    }
}
```

```
class ReadersWritersAccess
{
public:
    virtual void EnterReader() = 0;
    virtual void LeaveReader() = 0;
    virtual void EnterWriter() = 0;
    virtual void LeaveWriter() = 0;
};
```





# NT6 CV – Ex Leitores/Escritores prioridade aos escritores

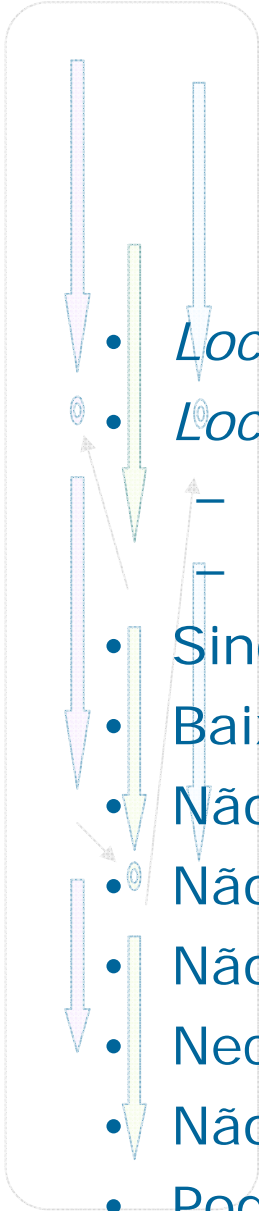


```
virtual void EnterReader() {  
    EnterCriticalSection(&csLock);  
    while ( nWrInWaiting > 0 || nWrInWriting > 0 ) {  
        SleepConditionVariableCS(&cWaitToRead,  
                                &csLock, INFINITE);  
    }  
    ++nRdInReading;  
    LeaveCriticalSection(&csLock);  
}  
  
virtual void LeaveReader() {  
    EnterCriticalSection(&csLock);  
    if (--nRdInReading == 0) {  
        WakeConditionVariable(&cWaitToWrite);  
    }  
    LeaveCriticalSection(&csLock);  
}
```

```
virtual void EnterWriter() {  
    EnterCriticalSection(&csLock);  
  
    ++nWrInWaiting;  
    while ( nRdInReading > 0 || nWrInWriting > 0 ) {  
        SleepConditionVariableCS(&cWaitToWrite,  
                                &csLock, INFINITE);  
    }  
    --nWrInWaiting;  
    ++nWrInWriting;  
    LeaveCriticalSection(&csLock);  
}  
  
virtual void LeaveWriter() {  
    EnterCriticalSection(&csLock);  
  
    --nWrInWriting;  
    if (nWrInWaiting > 0) {  
        WakeConditionVariable(&cWaitToWrite);  
    } else {  
        WakeAllConditionVariable(&cWaitToRead);  
    }  
  
    LeaveCriticalSection(&csLock);  
}
```



# NT6\* Slim Reader/Writer Locks - SRW

- 
- *Lock* to tipo leitores/escritor
  - *Lock* adquirido em:
    - *Shared mode* – “read-only mode”
    - *exclusive mode* – “read/write mode” ou
  - Sincronismo de threads do mesmo processo
  - Baixo recursos de memória (*size of a pointer*)
  - Não têm nenhum objecto do *kernel* associado
  - Não suportam pedidos recursivos
  - Não possuem *spin count* associado
  - Necessitam de iniciação mas não possuem operação remoção
  - Não está definido a ordem que as threads obtém o *lock*
  - Podem ser utilizados com as *condition variables* do Windows



# NT6\* Slim Reader/Writer Locks - SRW



**SRWLOCK** lock;

VOID WINAPI **InitializeSRWLock** (PSRWLOCK SRWLock );

VOID WINAPI **AcquireSRWLockShared** (PSRWLOCK SRWLock);

VOID WINAPI **AcquireSRWLockExclusive** (PSRWLOCK SRWLock);

VOID WINAPI **ReleaseSRWLockShared** (PSRWLOCK SRWLock);

VOID WINAPI **ReleaseSRWLockExclusive** (PSRWLOCK SRWLock);



# NT6 SRW – Ex Leitores/Escritores

```
class ReadersWritersNT6SRW: public ReadersWritersAcess
{
private:
    SRWLOCK srwLock;

public:
    ReadersWritersNT6SRW() {
        InitializeSRWLock(&srwLock);
    }

    void EnterReader() {
        AcquireSRWLockShared(&srwLock);
    }

    void LeaveReader() {
        ReleaseSRWLockShared(&srwLock);
    }

    void EnterWriter() {
        AcquireSRWLockExclusive(&srwLock);
    }

    void LeaveWriter() {
        ReleaseSRWLockExclusive(&srwLock);
    }
};
```

```
class ReadersWritersAcess
{
public:
    virtual void EnterReader() = 0;
    virtual void LeaveReader() = 0;
    virtual void EnterWriter() = 0;
    virtual void LeaveWriter() = 0;
};
```



# Outros mecanismos sincronismo

- **One-Time Initialization**

- WinVista or higher

- [http://msdn.microsoft.com/en-us/library/aa363808\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363808(VS.85).aspx)

- **Interlocked Singly Linked Lists**

- winXP or higher

- [http://msdn.microsoft.com/en-us/library/ms684121\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684121(VS.85).aspx)

- **Timer Queues**

- win2000 or higher

- [http://msdn.microsoft.com/en-us/library/ms686796\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686796(VS.85).aspx)

