OO – Engenharia Eletrônica

-

Programação em C/C++

Slides 19 A: Introdução à *Multithreading.*

Introdução à *Multithreading*: execução concorrente de tarefas.

Exemplos usando a API do Windows.

Prof. Jean Marcelo SIMÃO, DAELN/UTFPR
Aluno monitor: Vagner Vengue.

Threads (API do Windows)

Obs.: Material inicial elaborado por Eduardo Cromack Lippmann no 2o Semestre de 2007. O Lippmann era então aluno da disciplina em questão e estagiário em projeto do Professor da disciplina.

Threads - Win32 API

- O Win32 API (também conhecido como Windows API) é um arquétipo (*framework*) baseado em C para a criação de aplicações para Windows e está disponível desde as primeiras versões deste sistema operacional.

Funções básicas para *threads*

Threads - Win32 API

CreateThread

```
HANDLE WINAPI CreateThread(
                __in        LPSECURITY_ATTRIBUTES lpThreadAttributes,
                __in        SIZE_T dwStackSize,
                __in        LPTHREAD_START_ROUTINE lpStartAddress,
                __in        LPVOID lpParameter,
                __in        DWORD dwCreationFlags,
                __out       LPDWORD lpThreadId
);
```

- Cria e executa uma *thread*.
- Parâmetros:
  - *lpThreadAttributes*: atributos de segurança. Pode ser passado NULL para usar os valores padrões.
  - *dwStackSize*: quantidade inicial de memória que a *thread* precisará. Pode ser passado 0(zero) para usar o valor padrão.
  - *lpStartAddress*: função que a *thread* deve executar.
  - *lpParameter*: parâmetro passado para a função.
  - *dwCreationFlags*: flags que indicam o modo de criação da *thread*. Pode ser passado 0(zero) para usar os valores padrões.
  - *lpThreadId*: identificador da *thread*.

Threads - Win32 API

ExitThread

```
VOID WINAPI ExitThread(
        __in        DWORD dwExitCode
);
```

- Encerra a *thread* que a executa.
- Parâmetro:
  - *dwExitCode*: zero para indicar o término normal.

Threads - Win32 API

TerminateThread

```
BOOL WINAPI TerminateThread(
                __in_out     HANDLE hThread,
                __in         DWORD dwExitCode
);
```

- Força o término de uma *thread*.
- Não é a maneira mais indicada de se terminar uma *thread*, pois ela pode deixar tarefas importantes sem acabar. Na maioria da vezes, o mais indicado é deixar que a *thread* encerre sozinha, de acordo com a estrutura lógica do programa.

Threads - Win32 API

GetCurrentThreadId e GetThreadId

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

- Retorna o valor do identificador da *thread* que a executa.

```
DWORD WINAPI GetThreadId(
        __in        HANDLE Thread
);
```

- Retorna o valor do identificador de uma *thread* específica.

Threads - Win32 API

SetThreadPriority

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

- Modifica a prioridade de uma *thread*.
- A prioridade de uma *thread* no Windows é a combinação da prioridade do processo com a prioridade da *thread*. Esta prioridade pode ser representada por uma da seguintes constantes:

  THREAD_PRIORITY_ABOVE_NORMAL,
  THREAD_PRIORITY_BELOW_NORMAL,
  THREAD_PRIORITY_HIGHEST,
  THREAD_PRIORITY_IDLE,
  THREAD_PRIORITY_LOWEST,
  THREAD_PRIORITY_NORMAL,
  THREAD_PRIORITY_TIME_CRITICAL.

- Parâmetros:
  - *hThread:* variável de referência para a *thread.*
  - *nPriority:* prioridade da *thread.*

Threads - Win32 API

GetThreadPriority

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

- Retorna a prioridade de uma *thread.*

Threads - Win32 API

Sleep e SwitchToThread

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

- Suspende a execução da *thread* por uma quantidade mínima de milissegundos.

```
BOOL WINAPI SwitchToThread( void );
```

- Informa ao sistema operacional que ele pode executar outra *thread*.
- A execução, ou não, de outra *thread*, depende da forma como o sistema operacional escalona (organiza a execução) as *threads*.

Funções para sincronização (Mutex)

Threads - Win32 API

Mutexes

Revisando:

- São utilizados para **exclusão mutua**, ou seja, permitem que apenas uma *thread* por vez acesse um recurso.

- Se uma *thread* tenta acessar um recurso que outra *thread* está bloqueando, é impedida e libera o processador para que outras *threads* executem. Isso garante que uma *thread* não desperdice processamento porque está aguardando por um recurso bloqueado por outra *thread*.

Threads - Win32 API

CreateMutex

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

- Cria um objeto de mutex.
- Parâmetros:
  - *lpMutexAttributes*: atributos de segurança. Pode ser passado NULL para usar os valores padrões.
  - *bInitialOwner*: indica se a *thread* que cria o mutex deve obter o acesso inicial a ele.
  - *lpName*: nome do mutex, caso seja ele usado em processos diferentes.

Threads - Win32 API

ReleaseMutex

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

- Libera a “posse” do objeto de mutex.
- Deve ser chamada pela *thread* que está mantendo o acesso ao mutex.

Funções para sincronização (CRITICAL_SECTION)

Threads - Win32 API

InitializeCriticalSection e DeleteCriticalSection

- CRITICAL_SECTION é mais uma opção que o Windows disponibiliza para sincronização de *threads*. Sua funcionalidade é a mesma do mutex, porém, apresenta métodos um pouco mais simples de se trabalhar. Diferentemente do mutex, só pode ser usado dentro do mesmo processo.

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

- Inicializa um objeto de CRITICAL_SECTION com os valores padrões.

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

- Libera os recursos utilizados por objetos de CRITICAL_SECTION.

Threads - Win32 API

EnterCriticalSection e LeaveCriticalSection

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

- Aguarda pela “posse” de um objeto de CRITICAL_SECTION.

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

- Libera a “posse” de um objeto de CRITICAL_SECTION.

Funções para sincronização (Semáforos)

Threads - Win32 API

Semáforos

Revisando:

- São mecanismos que **permitem que um determinado número de *threads* tenham acesso a um recurso**. Agindo como um contador que não deixa ultrapassar um limite.

- No momento em que um objeto de semáforo é criado, é especificada a quantidade máxima de *threads* que ele deve permitir. Então cada *thread* que queira acessar o recurso, deve chamar uma função que decrementa em 1 o semáforo (*down*) e, após utilizar o recurso, chamar uma função que incremente em 1 o semáforo (*up*). Quando o contador do semáforo chega a zero, significa que o número de *threads* chegou ao limite e o recurso ficará bloqueado para as *threads* que chegarem depois, até que pelo menos uma das *threads* que estão utilizando o recurso o libere, incrementado o contador do semáforo.

Threads - Win32 API

CreateSemaphore

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

- Cria um objeto de semáforo.
- Parâmetros:
  - *lpSemaphoreAttributes*: atributos de segurança. Pode ser NULL, para usar os valores padrões.
  - *lInitialCount*: valor inicial do contador do semáforo. Deve ser maior ou igual a zero e menor ou igual ao valor máximo do semáforo.
  - *lMaximumCount*: valor máximo do contador so semáforo, ou seja, o númerio máximo de *threads* que ele deve permitir simultaneamente.
  - *lpName*: nome do semáforo, caso ele seja usado em processos diferentes.

Threads - Win32 API

ReleaseSemaphore

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

- Aumenta o contador do semáforo especificado em um determinado valor. Deve ser usada para indicar que uma *thread* esta liberando a "posse" do objeto de semáforo.
- Parâmetros:
  - *hSemaphore*: variável de referência para o semáforo.
  - *lReleaseCount*: valor a ser aumentado no contador do semáforo.
  - *lpPreviousCount*: ponteiro para um inteiro, onde é colocado o valor anterior do contador do semáforo. Pode ser NULL, se o valor antigo do semáforo não for necessário.

Funções para sincronização (Funções de “espera” e CloseHandle)

Threads - Win32 API

WaitForSingleObject

- Funções de espera permitem que uma *thread* bloqueie a sua própria execução, até que uma condição seja alcançada, como o término de execução de outra *thread* ou o acesso a um determinado recurso.
- Podem ser usadas em conjunto com: mutexes, semáforos e *threads*.

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

- Aguarda por um objeto (um mutex ou o término de execução de uma *thread*, por exemplo).
- Parâmetros:
  - *hHandle*: variável de referência para obejto que deve ser esperado.
  - *dwMilliseconds*: a quantidade máxima de milissegundos que a função deve esperar. Pode ser a constante INFINITE, para que o tempo não expire.

Threads - Win32 API

WaitForMultipleObjects

```
DWORD WINAPI WaitForMultipleObjects(
                __in        DWORD nCount,
                __in        const HANDLE* lpHandles,
                __in        BOOL bWaitAll,
                __in        DWORD dwMilliseconds
);
```

- Aguarda por um *array* de objetos (mutexes, ou o término de execução de uma *threads*, por exemplo).
- Parâmetros:
  - *nCount*: tamanho do *array* de objetos.
  - *lpHandles*: *array* de objetos.
  - *bWaitAll*: TRUE se a função deve esperar por todos os objetos do *array*, ou FALSE, caso ela precise apenas esperar por pelo menos um objeto.
  - *dwMilliseconds:*: a quantidade máxima de milissegundos que a função deve esperar. Pode ser a constante INFINITE, para que o tempo não expire.

Threads - Win32 API

CloseHandle

```
BOOL WINAPI CloseHandle(
        __in        HANDLE hObject
);
```

- Libera os recursos reservados para um objeto do tipo HANDLE (na linguagem C: typedef void* HANDLE).
- Deve ser usada sempre que for utilizado um objeto do tipo HANDLE, como, por exemplo:
  - Mutex,
  - Semaphore,
  - *Thread*.

Funções para sincronização (Mensagens)

Threads - Win32 API

Mensagens

Revisando:

- Diferentemente de mutex e semáforos, as mensagens não são específicas para o controle de acesso a um recurso. Elas são usadas como uma **forma de comunicação entre *threads***.

- Basta definir um conjunto de mensagens e trocá-las entre as *threads*.

- Pode ser útil quando se tem *threads* executando de forma continua, pois cada uma delas precisa verificar a sua fila de mensagens de tempos em tempos (em laço).

Threads - Win32 API

PostThreadMessage

```
BOOL PostThreadMessage(
  DWORD idThread,
  UINT Msg,
  WPARAM wParam,
  LPARAM lParam
);
```

- Envia uma mensagem para a fila de mensagens de uma *thread*.
- Parâmetros:
  - *idThread*: identificador da *thread.*
  - *Msg*: mensagem.
  - *wParam*: informação adicional e opcional, que pode ser enviada juntamente com a mensagem.
  - *lParam*: informação adicional e opcional, que pode ser enviada juntamente com a mensagem.

Threads - Win32 API

GetMessage

```
BOOL GetMessage(
  LPMSG lpMsg,
  HWND hWnd,
  UINT wMsgFilterMin,
  UINT wMsgFilterMax
);
```

- Verifica a fila de mensagens de uma *thread* e, caso não tenha mensagens, permanece aguardando até que tenha.
- Parâmetros:
  - *lpMsg*: um ponteiro para um objeto de MSG, o qual receberá a mensagem.
  - *hWnd*: variável de referência para uma janela pertencente a *thread*. Pode ser passado NULL, para que a *thread* receba todas as mensagens enviadas a ela.
  - *wMsgFilterMin e wMsgFilterMax*: indicam o intervalo de mensagens (constantes inteiras) que a *thread* deve receber. Pode ser zero, para que a *thread* receba todas as mensagens.

Exemplo 01

Exemplo 01

```
//  DWORD, LPVOID, HANDLE ...  são Windows Data Types

//  DWORD - 32-bit unsigned integer. The range is 0 through 4294967295 decimal.
//  This type is declared in WinDef.h as follows:
//  typedef unsigned long DWORD;

//  LPVOID - Pointer to any type.
//  This type is declared in WinDef.h as follows:
//  typedef void* LPVOID;

//  HANDLE - Handle to an object.
//  This type is declared in WinNT.h as follows:
//  typedef PVOID HANDLE;

//  LPVOID - Pointer to any type.
//  This type is declared in WinDef.h as follows:
//  typedef void *LPVOID;


#include <iostream>

using namespace std;

#include <windows.h>


bool thread_ligada;

DWORD thread_id;


DWORD WINAPI escreveAlgo( LPVOID lpParam )

{

    while ( thread_ligada )

    {

        cout << "Executando thread..." << endl;

    }

}
```

Exemplo 01

```
int main()
{
    cout << "Digite ENTER para iniciar e parar a thread..." << endl;
    cin.get();              // aguarda um ENTER do usuário;
    thread_ligada = true;

    // inicia uma nova thread, passando como parâmetro
    // o "id" da thread e a função que ela deve executar;
    HANDLE hndThread;
    hndThread = CreateThread(NULL, 0, escreveAlgo, NULL, 0, &thread_id);

    cin.get();              // aguarda um ENTER do usuário;
    thread_ligada = false;  // informa a thread que ela deve parar;

    if ( NULL != hndThread )
        WaitForSingleObject(hndThread, INFINITE);

    CloseHandle( hndThread );
    return 0;
}
```

Exemplo 01

Resultado do programa Exemplo 01.

Exemplo 02

(Exemplo de uso de mutex)

Exemplo 02 (mutex)

```
#include <iostream>
#include <windows.h>
using namespace std;

DWORD thread_1_id;
DWORD thread_2_id;

HANDLE meu_mutex;

// função passada para a thread 1;
DWORD WINAPI tarefa_1( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    WaitForSingleObject(meu_mutex, INFINITE);
    for (char* s = "123456"; *s != '\0'; s++)
    {
        cout << *s;
    }
    ReleaseMutex( meu_mutex );
    //---------------------------------------
}
```

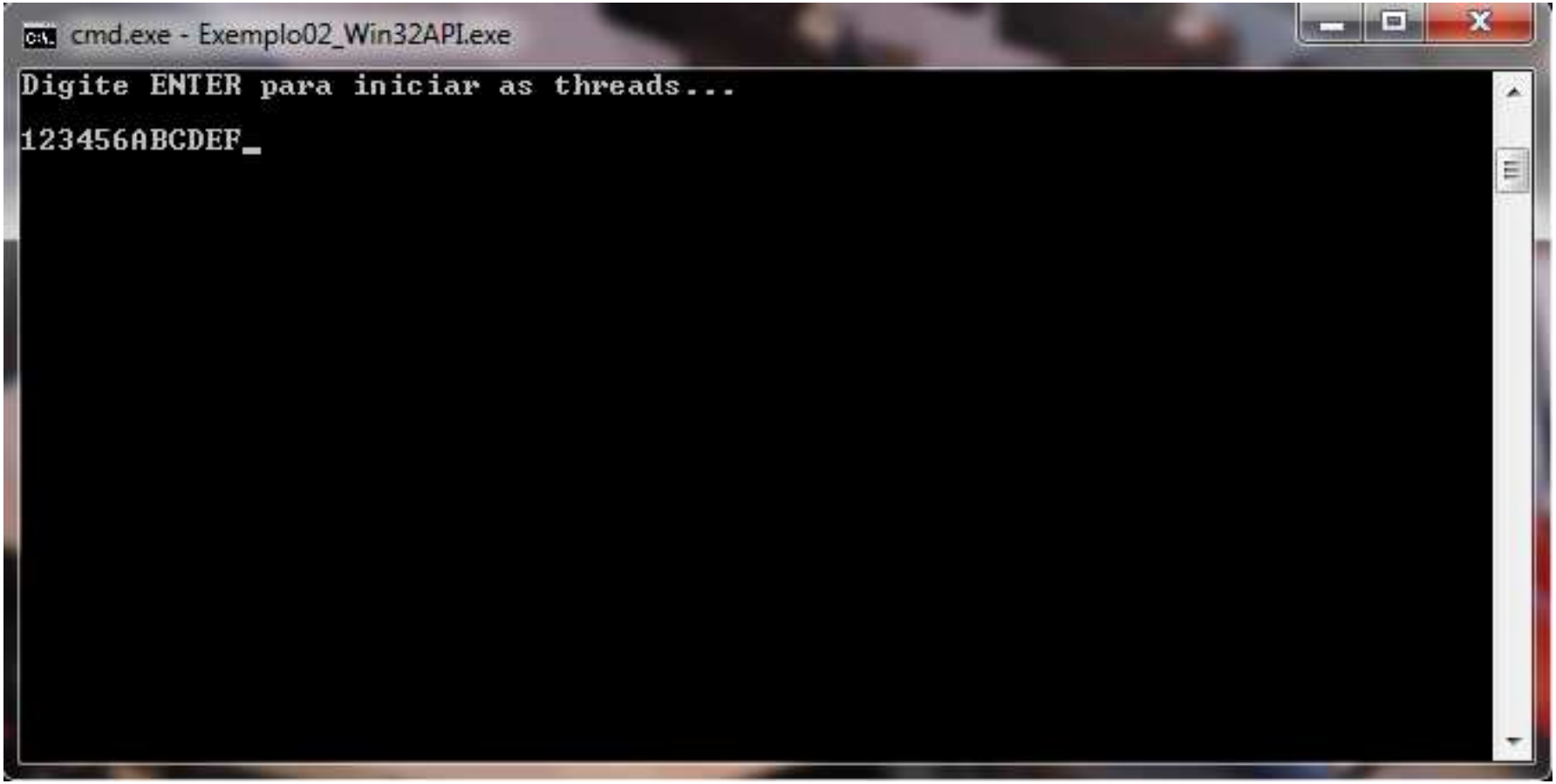
```
// função passada para a thread 2;
DWORD WINAPI tarefa_2( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    WaitForSingleObject(meu_mutex, INFINITE);
    for (char* s = "ABCDEF"; *s != '\0'; s++)
    {
        cout << *s;
    }
    ReleaseMutex( meu_mutex );
    //---------------------------------------
}
```

Exemplo 02 (mutex)

```
int main()
{
    HANDLE hndThread_1, hndThread_2;

    cout << "Digite ENTER para iniciar as threads..." << endl;
    cin.get();              // aguarda um ENTER do usuário;

    meu_mutex = CreateMutex(NULL, FALSE, NULL);
    if ( NULL == meu_mutex ) {
        cout << "Erro ao criar o mutex." << endl;
        return 0;
    }

    // inicia as threads, passando como parâmetro
    // o "id" da thread e a função que cada uma deve executar;
    hndThread_1 = CreateThread(NULL, 0, tarefa_1, NULL, 0, &thread_1_id);
    hndThread_2 = CreateThread(NULL, 0, tarefa_2, NULL, 0, &thread_2_id);
    // faz com que a thread principal espere as threads acabarem;
    if ( NULL != hndThread_1 ) {   WaitForSingleObject(hndThread_1, INFINITE);        }
    if ( NULL != hndThread_2 ) {   WaitForSingleObject(hndThread_2, INFINITE);        }

    CloseHandle( hndThread_1 );
    CloseHandle( hndThread_2 );
    CloseHandle( meu_mutex );
    cin.get();              // aguarda um ENTER do usuário;
    return 0;
}
```

Exemplo 02 (mutex)

Resultado do programa Exemplo 02.



Exemplo 03

(Exemplo de uso de CRITICAL_SECTION)

Exemplo 03 (CRITICAL SECTION)

```
#include <iostream>
#include <windows.h>
using namespace std;

DWORD thread_1_id;
DWORD thread_2_id;

CRITICAL_SECTION critical;

// função passada para a thread 1;
DWORD WINAPI tarefa_1( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    EnterCriticalSection( &critical );
    for (char* s = "123456"; *s != '\0'; s++)
    {
        cout << *s;
    }
    LeaveCriticalSection( &critical );
    //---------------------------------------
}
```

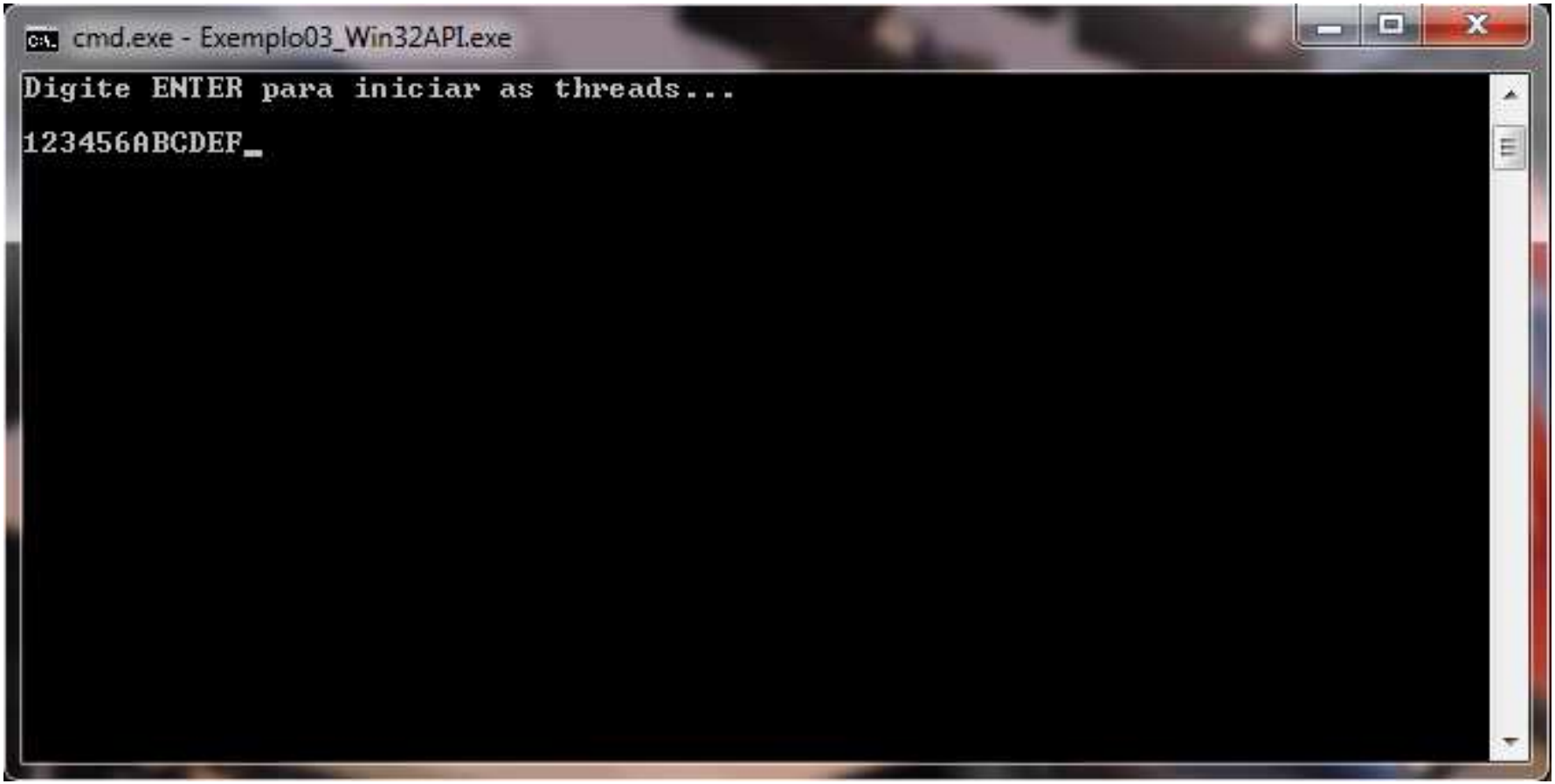
```
// função passada para a thread 2;
DWORD WINAPI tarefa_2( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    EnterCriticalSection( &critical );
    for (char* s = "ABCDEF"; *s != '\0'; s++)
    {
        cout << *s;
    }
    LeaveCriticalSection( &critical );
    //---------------------------------------
}
```

Exemplo 03 (CRITICAL SECTION)

```
int main()
{
    HANDLE hndThread_1, hndThread_2;

    cout << "Digite ENTER para iniciar as threads..." << endl;
    cin.get();              // aguarda um ENTER do usuário;

    InitializeCriticalSection( &critical );

    // inicia as threads, passando como parâmetro
    // o "id" da thread e a função que cada uma deve executar;
    hndThread_1 = CreateThread(NULL, 0, tarefa_1, NULL, 0, &thread_1_id);
    hndThread_2 = CreateThread(NULL, 0, tarefa_2, NULL, 0, &thread_2_id);
    // faz com que a thread principal espere a 'thread_1' e a 'thread_2' acabarem;
    if ( NULL != hndThread_1 ) {    WaitForSingleObject(hndThread_1, INFINITE);        }
    if ( NULL != hndThread_2 ) {    WaitForSingleObject(hndThread_2, INFINITE);        }

    CloseHandle( hndThread_1 );
    CloseHandle( hndThread_2 );
    DeleteCriticalSection( &critical );

    cin.get();              // aguarda um ENTER do usuário;
    return 0;
}
```

Exemplo 03 (CRITICAL SECTION)

Resultado do programa Exemplo 03.



Exemplo 04

(Exemplo de uso de semáforo)

Exemplo 04 (semáforo)

```
#include <iostream>
#include <windows.h>
using namespace std;

DWORD thread_1_id;
DWORD thread_2_id;

HANDLE meu_semaforo;

// função passada para a thread 1;
DWORD WINAPI tarefa_1( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    WaitForSingleObject(meu_semaforo,
        INFINITE);
    for (char* s = "123456"; *s != '\0'; s++)
    {
        cout << *s;
    }
    ReleaseSemaphore( meu_semaforo, 1, NULL );
    //---------------------------------------
}
```

```
// função passada para a thread 2;
DWORD WINAPI tarefa_2( LPVOID lpParam )
{
    //------------------------ REGIÃO CRÍTICA
    WaitForSingleObject(meu_semaforo,
        INFINITE);
    for (char* s = "ABCDEF"; *s != '\0'; s++)
    {
        cout << *s;
    }
    ReleaseSemaphore( meu_semaforo, 1, NULL );
    //---------------------------------------
}
```

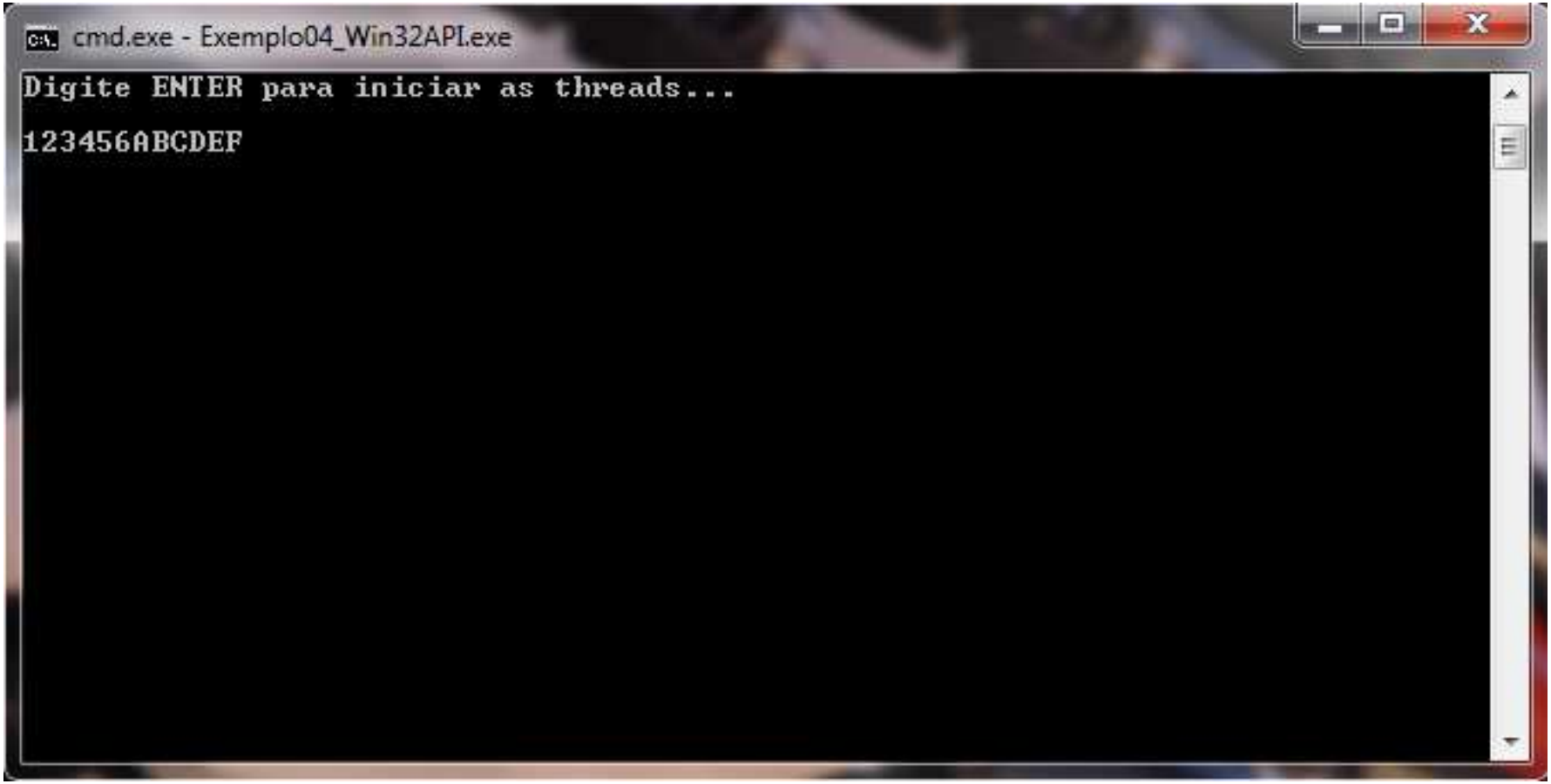
Exemplo 04 (semáforo)

```
int main() {
    HANDLE hndThread_1, hndThread_2;

    cout << "Digite ENTER para iniciar as threads..." << endl;
    cin.get();                  // aguarda um ENTER do usuário;

    meu_semaforo = CreateSemaphore(NULL, 1, 1, NULL);
    if ( NULL == meu_semaforo ) {
         cout << "Erro ao criar o semaforo. " << GetLastError() << endl;
         cin.get();
         return 0;
    }
    // inicia as threads, passando como parâmetro
    // o "id" da thread e a função que cada uma deve executar;
    hndThread_1 = CreateThread(NULL, 0, tarefa_1, NULL, 0, &thread_1_id);
    hndThread_2 = CreateThread(NULL, 0, tarefa_2, NULL, 0, &thread_2_id);
    // faz com que a thread principal espere as outras threads acabarem;
    if ( NULL != hndThread_1 ) {    WaitForSingleObject(hndThread_1, INFINITE);        }
    if ( NULL != hndThread_2 ) {    WaitForSingleObject(hndThread_2, INFINITE);        }

    CloseHandle( hndThread_1 );
    CloseHandle( hndThread_2 );
    CloseHandle( meu_semaforo );

    cin.get();                  // aguarda um ENTER do usuário;
    return 0;
}
```

Semáforo binário (realiza exclusão mutua).

Exemplo 04 (semáforo)

Resultado do programa Exemplo 04.



Exemplo 05

(Exemplo de uso de semáforo)

Exemplo 05 (semáforo)

```
#include <iostream>
#include <windows.h>
using namespace std;

// número de threads;
#define NUM_THREADS     5
// número de threads permitidas pelo
    semáforo;
#define NUM_SEMAFORO    3

DWORD vtThread_id[NUM_THREADS];
int thread_ativa[NUM_THREADS];

// referência para o objeto de semáforo;
HANDLE meu_semaforo;
```

```
// função passada para as threads;
DWORD WINAPI tarefa( LPVOID lpParam )
{
    int* valor = static_cast<int*>( lpParam ); //
    recupera o id da thread;
    int id = *valor;
    //------------------------ REGIÃO CRÍTICA
    WaitForSingleObject(meu_semaforo,INFINITE);
    for (int i = 0; i < 5; i++)
    {
        thread_ativa[ id ] = 1;
        // marca a thread como ativa e aguarda até
    ser desmarcada pela thread principal;
        while ( 1 == thread_ativa[ id ] )
        {
            Sleep(100);
        }
    }
    thread_ativa[ id ] = -1;// informa a thread
    principal que já acabou o seu trabalho;
    ReleaseSemaphore( meu_semaforo, 1, NULL );
    //-------------------------------------
}
```

Exemplo 05 (semáforo)

```
void monitorar()
{
    for (int i = 0; i < NUM_THREADS; i++)
    {
        cout << "Thread " << i << "\t";
    }
    cout << endl;

    bool threads_executando;
    do{
        // verifica se há pelo menos 1 thread
        // executando;
        threads_executando = false;
        for (int i = 0; i < NUM_THREADS; i++)
        {
            if ( -1 != thread_ativa[ i ] )
            {
                threads_executando = true;
            }
        }
    if ( threads_executando )
    {
        // imprime no Console as threads que
    estão executando e que estão ativas, ou
    seja, estão com acesso ao semáforo;
        for (int i = 0; i < NUM_THREADS; i++)
        {
            if ( 1 == thread_ativa[i] )
            {
                cout << "|";
                thread_ativa[ i ] = 0;
            }
            cout << "\t\t";
        }
        cout << endl;
        Sleep(500);
     }
   }while ( threads_executando );
}
```

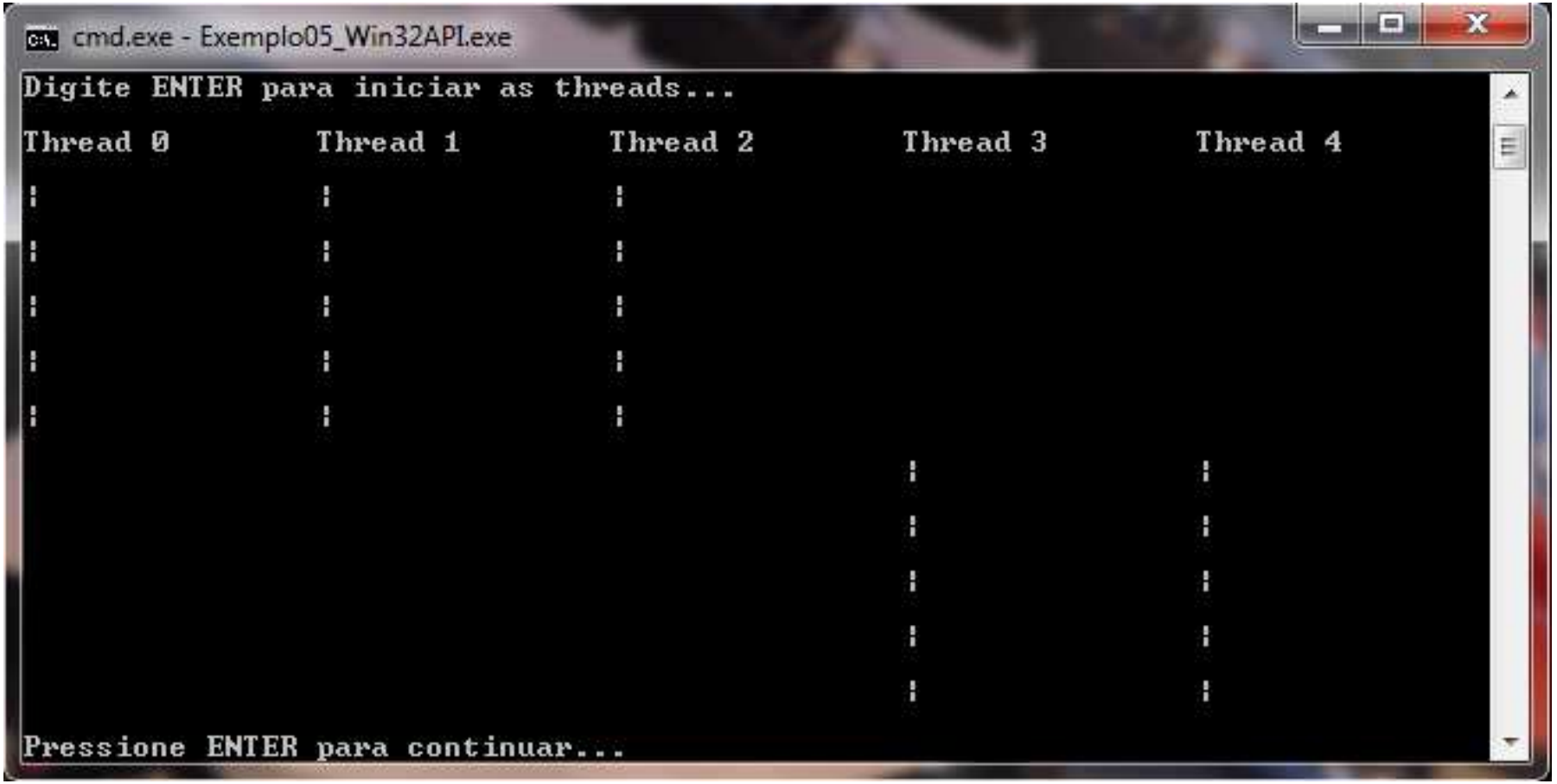
Exemplo 05 (semáforo)

```
int main() {
    HANDLE vtHndThreads[NUM_THREADS];

    cout << "Digite ENTER para iniciar "
        << "as threads..." << endl;
    cin.get();

    for (int i = 0; i < NUM_THREADS; i++)
    {
        thread_ativa[ i ] = 0;
    }

    // cria o semáforo;
    meu_semaforo = CreateSemaphore(NULL,
        NUM_SEMAFORO, NUM_SEMAFORO, NULL);
    if ( NULL == meu_semaforo )
    {
        cout << "Erro ao criar o semaforo. "
            << GetLastError() << endl;
        cin.get();
        return 0;
    }
```

```
    // inicia as threads, passando o "id" da
    thread e a função que cada uma deve
    executar;
    for (int i = 0; i < NUM_THREADS; i++)
    {
        vtHndThreads[i] = CreateThread(NULL,
            0, tarefa, (LPVOID) &i, 0,
            &vtThread_id[i]);

        Sleep(100);
    }

    monitorar();    // função que mostra as
    threads que estão ativas;

    for (int i = 0; i < NUM_THREADS; i++)
    {
        CloseHandle( vtHndThreads[i] );
    }
    CloseHandle( meu_semaforo );

    cout << "Pressione ENTER para "
        << "continuar..." << endl;
    cin.get();
    return 0;
}
```

Exemplo 05 (semáforo)

Resultado do programa Exemplo 05.



Exemplo 06

(Exemplo de uso de mutex)

Exemplo 06 (mutex)

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

// Constante para o Número de Threads
#define NumeroDeThreads   4
// Uma variável Handle é um "ponteiro
    especial"

HANDLE MeuMutex;
void AtivarThread();

int main()
{
  HANDLE Vetor_De_Threads[NumeroDeThreads];
  DWORD  Identificador_Da_Thread;
  int i;

  // Cria um mutex (sem dono)
  MeuMutex = CreateMutex(NULL, FALSE, NULL);
  if ( MeuMutex == NULL )
  {
    printf("Erro na funcao CreateMutex: %d\n",
      GetLastError() );
    return 0;
  }
```

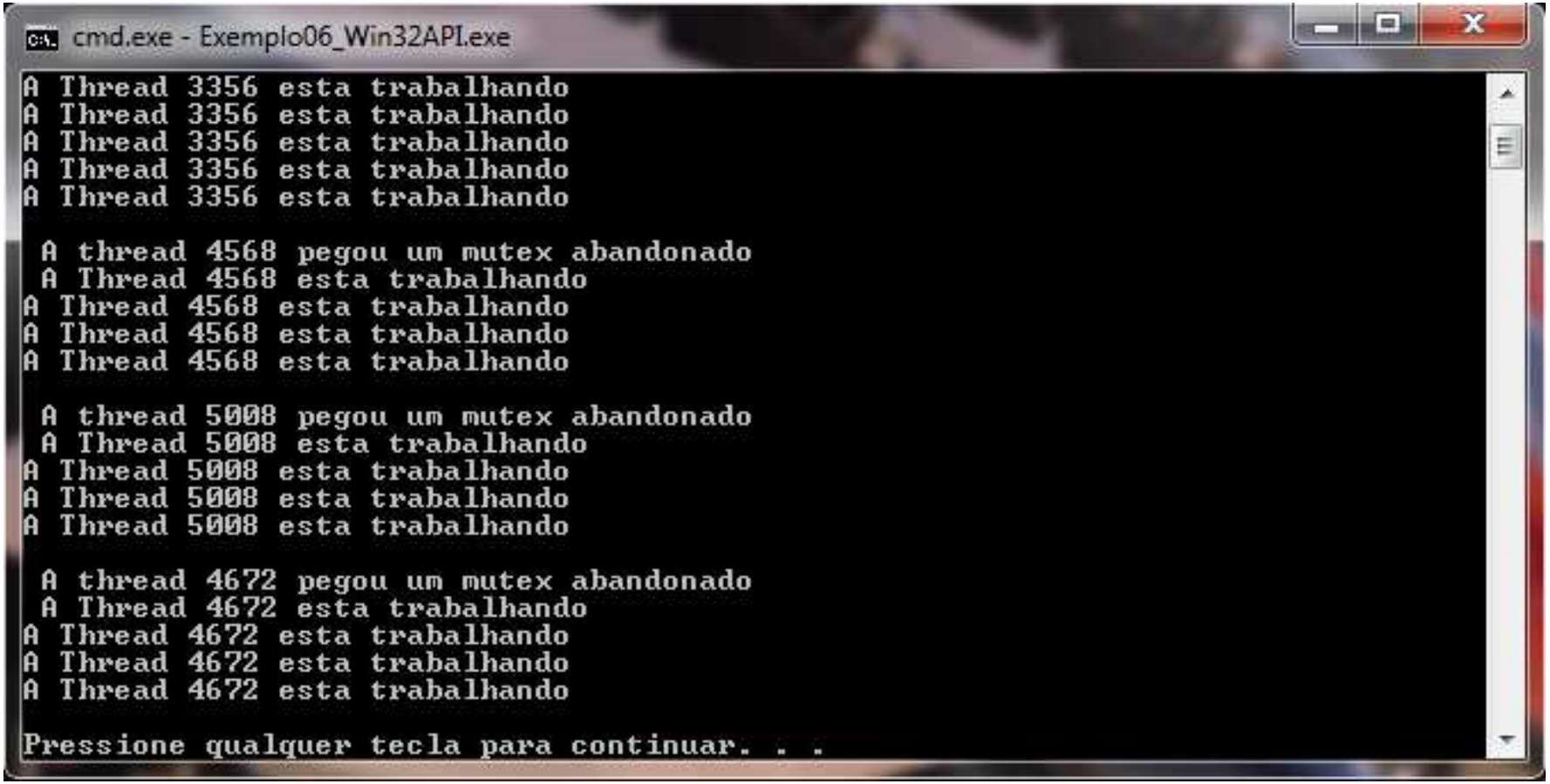
```
  // Criação das Threads
  for (i = 0; i < NumeroDeThreads; i++) {

    Vetor_De_Threads[i]=CreateThread(NULL, 0,
      (LPTHREAD_START_ROUTINE)AtivarThread,
      NULL, 0, &Identificador_Da_Thread);

    if ( Vetor_De_Threads[i] == NULL ) {
      printf("Erro em CreateThread: %d\n",
          GetLastError() );
      return 0;
    }
  }

  // espera os threads
  WaitForMultipleObjects( NumeroDeThreads,
      Vetor_De_Threads, TRUE, INFINITE);

  // Destroi as variáveis
  for ( i = 0; i < NumeroDeThreads; i++ ) {
    CloseHandle( Vetor_De_Threads[ i ] );
  }
  CloseHandle( MeuMutex ) ;

  system ( "PAUSE" );
  return 0;
}
```

Exemplo 06 (mutex)

```
void AtivarThread()
{
    int Ciclos = 0;
    int EstadoAtual;
    while ( Ciclos < 5 )
    {
        // Requisita a posse do mutex.
        EstadoAtual = WaitForSingleObject( MeuMutex, INFINITE );

        switch ( EstadoAtual )
        {
            case WAIT_OBJECT_0:     // caso a thread se aposse do mutex
            {
                DWORD identificador = 0;
                // Retrieves the thread identifier of the calling thread.
                identificador = GetCurrentThreadId();
                printf ( "A Thread %d esta trabalhando \n", identificador );
                Ciclos++;
            } break;
            case WAIT_ABANDONED:    // caso o thread se aposse de um mutex abandonado
            {
                printf(" \n A thread %d pegou um mutex abandonado \n", GetCurrentThreadId());
                Ciclos++;
            }
        }
    }
}
```

Exemplo 06 (mutex)

Resultado do programa Exemplo 06.



Exemplo 07

(Exemplo de prioridade de *threads*)

Exemplo 07 (prioridades)

```
#include <iostream>
#include <windows.h>
using namespace std;

DWORD thread_1_id;
DWORD thread_2_id;
CRITICAL_SECTION critical;
bool threads_ativas;

// função passada para a thread 1;
DWORD WINAPI tarefa_1( LPVOID lpParam )
{
   int cont = 0;
   while ( threads_ativas ) {
      //--------------------- REGIÃO CRÍTICA
      EnterCriticalSection( &critical );
      cout << "Tarefa 1 executando..."
         << endl;
      LeaveCriticalSection( &critical );
      //-----------------------------------
      cont ++;
      if ( 50 == cont ) {
         cont = 0;
         SwitchToThread();
      }
   }
}
```

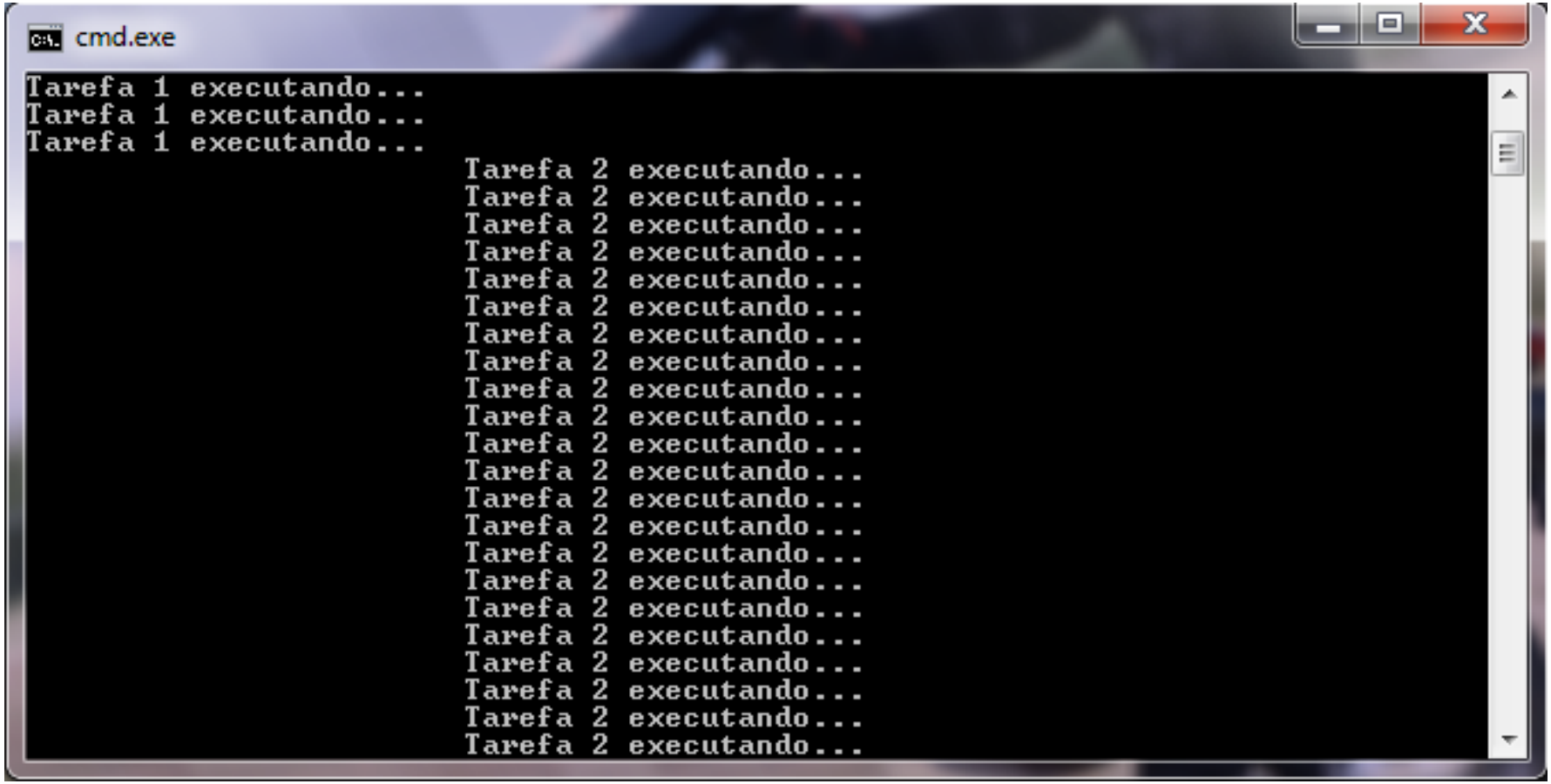
```
// função passada para a thread 2;
DWORD WINAPI tarefa_2( LPVOID lpParam )
{
   int cont = 0;
   while ( threads_ativas )
   {
      //--------------------- REGIÃO CRÍTICA
      EnterCriticalSection( &critical );
      cout << "\t\t\tTarefa 2 executando..."
         << endl;
      LeaveCriticalSection( &critical );
      //-----------------------------------


      cont ++;
      if ( 50 == cont ) {
         cont = 0;
         SwitchToThread();
      }
   }
}
```

Exemplo 07 (prioridades)

```
int main()
{
    HANDLE hndThread_1, hndThread_2;

    cout << "Digite ENTER para iniciar "
        << "e parar as threads..." << endl;
    cin.get(); // aguarda um ENTER do usuário;

    InitializeCriticalSection( &critical );
    threads_ativas = true;

    // inicia as threads, passando como
    parâmetro o "id" da thread e a função que
    cada uma deve executar;
    hndThread_1 = CreateThread(NULL, 0,
        tarefa_1, NULL, 0, &thread_1_id);
    hndThread_2 = CreateThread(NULL, 0,
        tarefa_2, NULL, 0, &thread_2_id);

    SetThreadPriority( hndThread_1,
        THREAD_PRIORITY_LOWEST);
    SetThreadPriority( hndThread_2,
        THREAD_PRIORITY_HIGHEST);

    cin.get(); // aguarda um ENTER do usuário;

    threads_ativas = false;
    // faz com que a thread principal espere
    as demais threads acabarem;
    if ( NULL != hndThread_1 ){
        WaitForSingleObject( hndThread_1,
            INFINITE);
    }

    if ( NULL != hndThread_2 ){
        WaitForSingleObject( hndThread_2,
            INFINITE);
    }

    CloseHandle( hndThread_1 );
    CloseHandle( hndThread_2 );

    DeleteCriticalSection( &critical );
    return 0;
}
```

Exemplo 07 (prioridades)

Resultado do programa Exemplo 07.



Exemplo 08

(Exemplo de mensagens entre *threads*)

Exemplo 08 (mensagem)

```
#include <iostream>
#include <windows.h>
using namespace std;

#define MSG_COMPLETE     WM_USER + 0
#define MSG_TAREFA_1     WM_USER + 1
#define MSG_TAREFA_2     WM_USER + 2

DWORD thread_id;

DWORD WINAPI executaTarefa( LPVOID lpParam )
{
   MSG msg;
   BOOL bRet;
   do{
      // aguarda uma mensagem;
      bRet = GetMessage( &msg, NULL, 0, 0);

      if ( ! bRet )
      {
         cout << "Ocorreu um erro no "
            << "recebimento da msg." << endl;
         cin.get();
      }
      else
      {
         // verifica a mensagem;
         switch ( msg.message )
         {
            case MSG_TAREFA_1:
               cout << "Executando tarefa 1..."
                  << endl;
               break;

            case MSG_TAREFA_2:
               cout << "Executando tarefa 2..."
                  << endl;
               break;

            case MSG_COMPLETE:
               cout << "Tarefas completas!"
                  << endl;
               return 0;
         }
      }
   } while( bRet );
   return 0;
}
```

Conforme - http://msdn.microsoft.com/en-us/library/ms644958(v=vs.85).aspx

MSG Structure - Contains message information from a thread's message queue. **Syntax**

```
typedef struct tagMSG  {
HWND   hwnd;
UINT   message;
WPARAM wParam;
LPARAM lParam;
DWORD  time;
POINT  pt;
} MSG, *PMSG, *LPMSG;
```

Members

hwnd - Type: HWND
A handle to the window whose window procedure receives the message. This member is NULL when the message is a thread message.

Message- Type: UINT
The message identifier. Applications can only use the low word; the high word is reserved by the system.

wParam - Type: WPARAM
Additional information about the message. The exact meaning depends on the value of the **message** member.

lParam - Type: LPARAM
Additional information about the message. The exact meaning depends on the value of the **message** member.

time - Type: DWORD
The time at which the message was posted.

pt - Type: POINT
The cursor position, in screen coordinates, when the message was posted.

Requirements

Minimum supported client Windows 2000 Professional**Minimum supported server**Windows 2000 Server**Header**
Winuser.h (include Windows.h)

See Also

Reference GetMessage PeekMessage PostThreadMessage
Conceptual Messages and Message Queues

Send comments about this topic to Microsoft
Build date: 5/3/2011

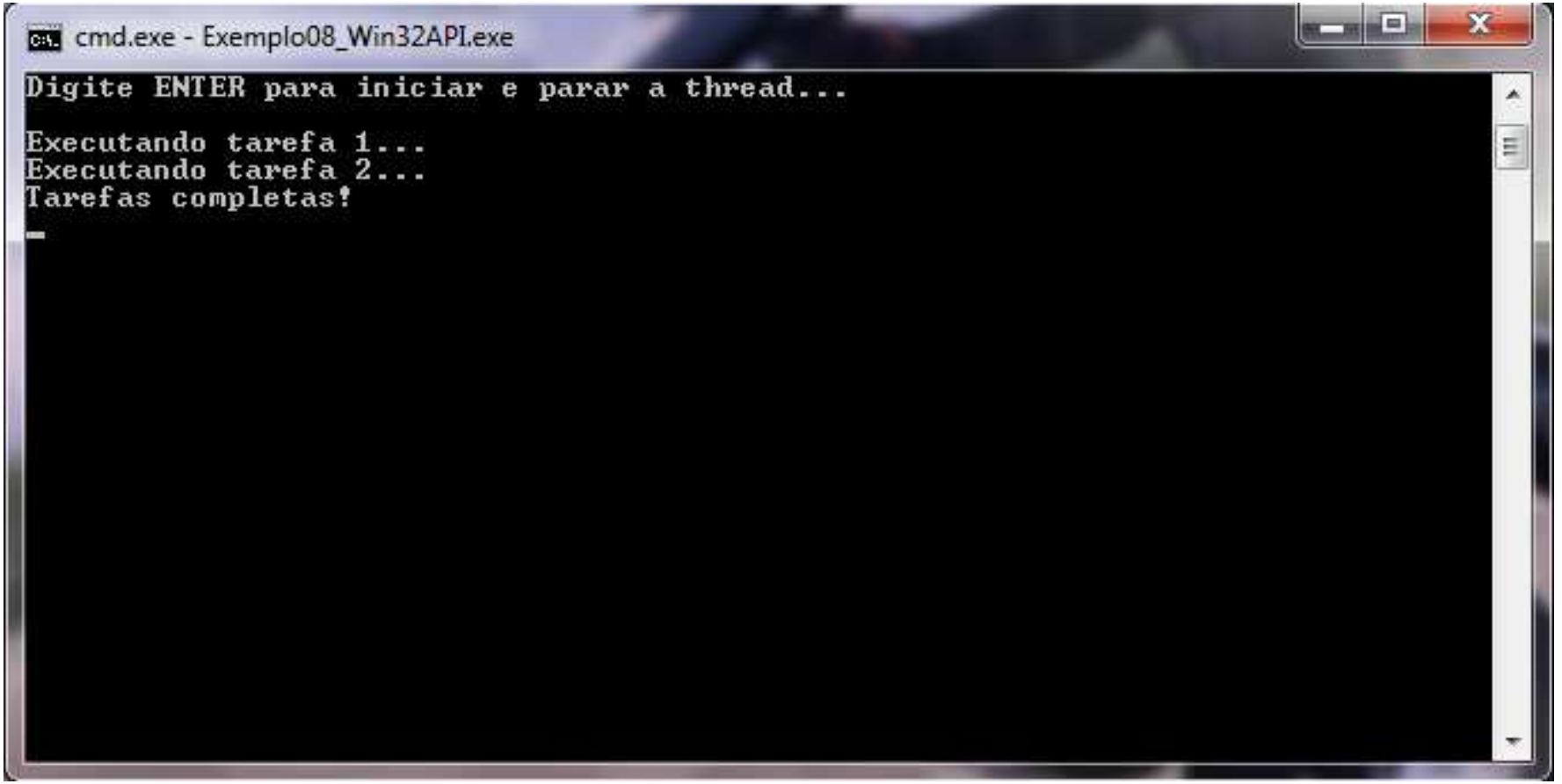
Exemplo 08 (mensagem)

```
int main()
{
    cout << "Digite ENTER para iniciar e parar a thread..." << endl;
    cin.get();              // aguarda um ENTER do usuário;
    // inicia uma nova thread, passando como parâmetro
    // o "id" da thread e a função que ela deve executar;
    HANDLE hndThread = CreateThread(NULL, 0, executaTarefa, NULL, 0, &thread_id);

    // envia uma mensagem, verificando o envio correto;
    while ( PostThreadMessage( thread_id, MSG_TAREFA_1, 0, 0 ) == 0 )
        Sleep(200);
    while ( PostThreadMessage( thread_id, MSG_TAREFA_2, 0, 0 ) == 0 )
        Sleep(200);
    while ( PostThreadMessage( thread_id, MSG_COMPLETE, 0, 0 ) == 0 )
        Sleep(200);

    cin.get();
    // aguarda a thread acabar;
    if ( NULL != hndThread )
        WaitForSingleObject(hndThread, INFINITE);

    CloseHandle( hndThread );
    return 0;
}
```

A fila de mensagens demora um tempo para ser criada, portanto, as primeiras mensagens envidas podem falhar. Para amenizar esse problema, pode-se utilizar um laço, que só pára quando a mensagem for enviada.

Exemplo 08 (mensagem)

Resultado do programa Exemplo 08.



Bibliografias relativas a *Threads*.

- TANENBAUM, Andrew Stuart: **Sistemas Operacionais Modernos.** Prentice Hall. 3ª Edição 2010.

- Microsoft Developer Network – Processes and Threads. Disponível em: <http://msdn.microsoft.com/en-us/library/ms684841(v=VS.85).aspx>. Acesso em: 05 nov 2010.

- Microsoft Developer Network - Synchronization. Disponível em <http://msdn.microsoft.com/en-us/library/ms686353(v=VS.85).aspx>. Acesso em: 05 nov 2010.