OO – Engenharia Eletrônica

-

Programação em C/C++

Slides 19: *Multithreading.*

Threads: Linhas de processamento.

Exemplos usando a API do Windows.

Prof. Jean Marcelo SIMÃO

Threads

Introdução/Revisão

Threads

Linhas de processamento

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 - O que são?

- Cada algoritmo requer uma seqüência de processos por parte da máquina e cada seqüência requer um espaço de memória.
- Uma seqüência de processos em um local de memória caracterizam uma *Thread*.

Ainda não entendi...

- Quando criamos um programa, desenvolvemos um algoritmo. Quando abrimos esse programa, ele passa a ocupar uma certa memória. Por meio dela que o programa irá executar todas as suas funções uma após a outra.

Então cada programa só tem uma *Thread*?

- NÃO necessariamente!
- Programas podem ser desenvolvidos para executar tarefas concorrentemente, ou seja, ocupar mais de um local de memória e executar mais de uma linha de processamento.

E quando que isto se faz necessário?

- A utilidade mais relacionada às *Threads* é a execução das "*Background Tasks*" (tarefas de segundo plano).
- Criando programas com várias *Threads* obtemos um menor tempo de resposta da máquina em certos contextos, como os de tarefas em segundo plano.
- Funções podem permanecer em *loop* sem atrapalhar a linha de processamento do programa.

Quando não se faz necessário?

- Quando uma tarefa está condicionada à finalização de outra.

Exemplo prático:

Threads

Para
Sistema Operacional
Windows

Threads em Windows

- Além de *pthread*, na programação (C/C++) para S.O. Windows, há outras alternativas.
- Pode-se utilizar as seguintes alternativas:
  - **C Run-time library** / Microsoft Run-time Library.
  - Win 32 API.
  - Namespace System::Threading do C++ .net

C Run-time Library

Microsoft Run-time Library

C Run-time Library

- *C Run-time Library* faz parte da *Microsoft Run-time Library*.

- "A *Microsoft Run-time Library* provê rotinas para programar o sistema operacional *Microsoft Windows*. Estas rotinas automatizam muitas tarefas comuns de programação que não são providas pelas linguagens C e C++" [Microsoft Visual C++ Help].

A Microsoft Run-time Libray faz apelo a Win32 API, simplificando chamadas de funções

Exemplo de Threads

C Run-time Library

Exemplo:

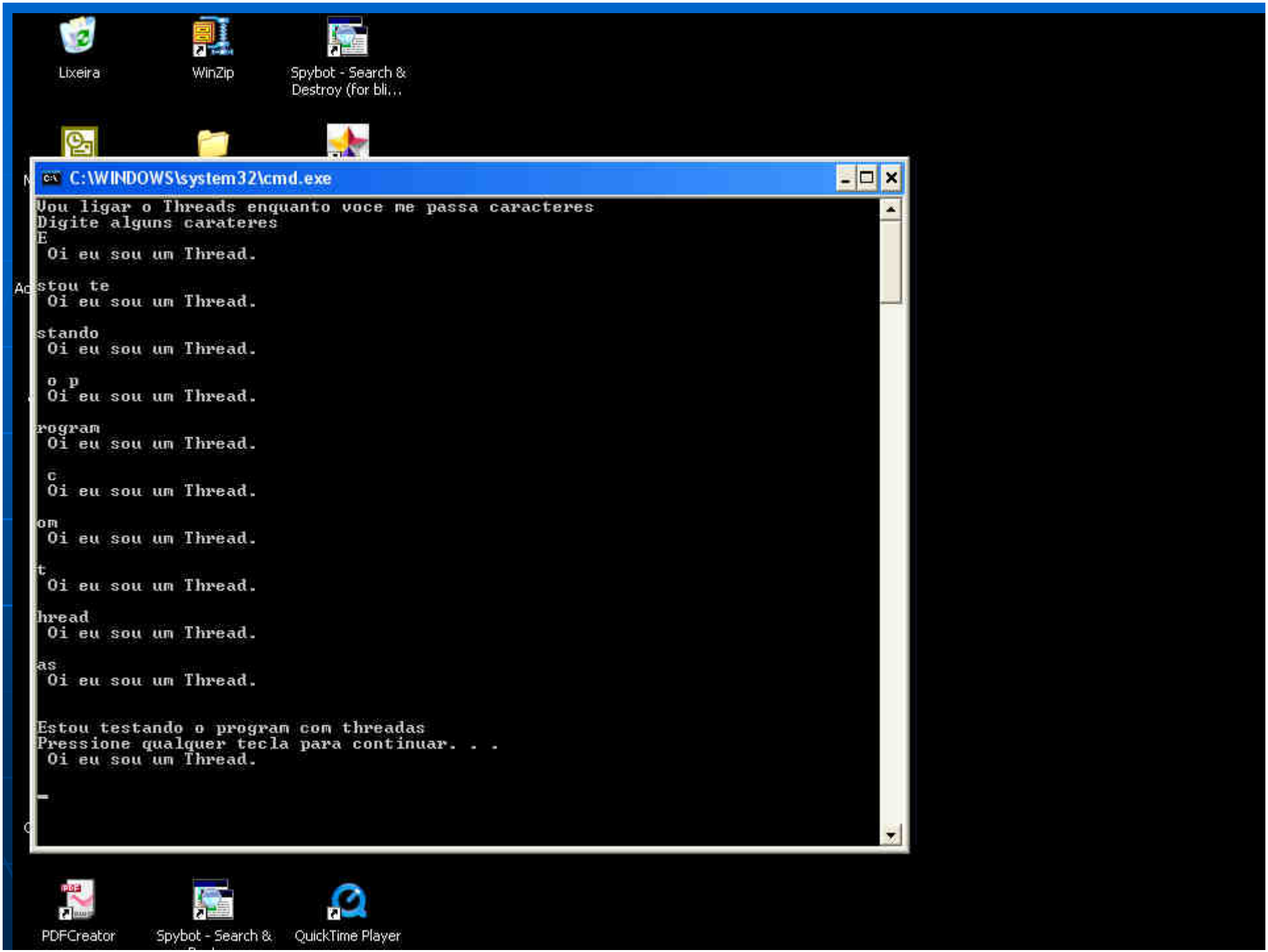
- O programa a seguir vai receber do usuário uma string enquanto uma OUTRA *Thread* imprime mensagens na tela.

- Este exemplo utiliza a função **_beginthread** da C run-time library que faz parte da Microsoft Run-time Library.

The Microsoft run-time library provides routines for programming for the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages.

```c
#include <process.h>            /*_beginthread, _endthread */
#include <stdio.h>
#include <windows.h>

void MeuThread ( void* i );

char frase [100];

int k = 1;

int  main ()
{
  puts ( "Vou ligar o Threads enquanto voce digita caracteres‟ );

 // Esta função abaixo faz parte de run-time library.
  _beginthread ( MeuThread, 0, NULL );

  puts ( "Digite alguns carateres" );
  gets ( frase );

  k = 0;
  puts ( frase );

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

```c
void MeuThread ( void *i )
{
   while ( k )
   {
        Sleep ( 1000 );
        puts ( " " );
        puts ( " Oi eu sou um Thread. " );
        puts ( " " );
    }
}
```

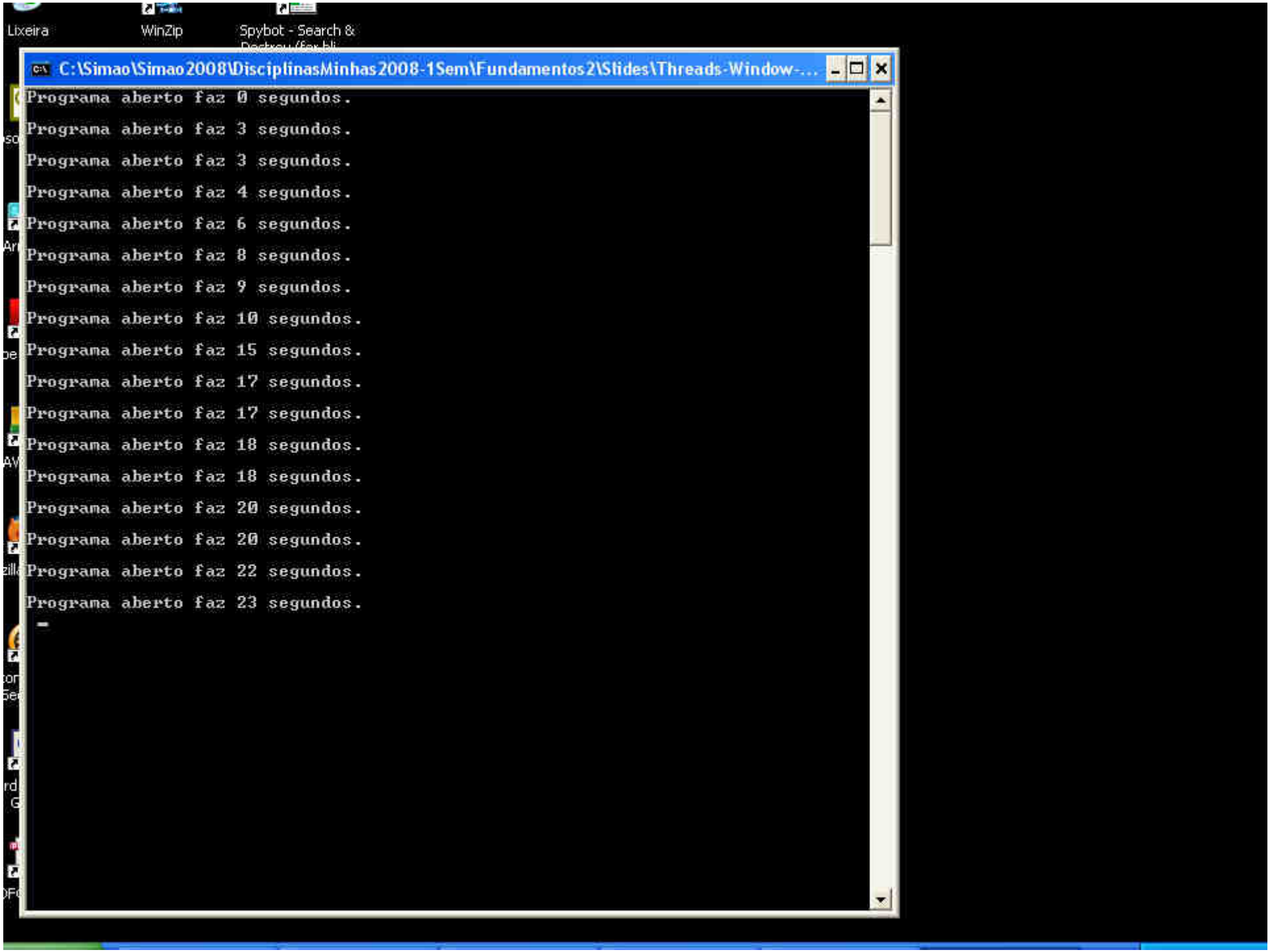

Lixeira
WinZip
Spybot - Search & Destroy (for bli...
C:\WINDOWS\system32\cmd.exe
Vou ligar o Threads enquanto voce me passa caracteres
Digite alguns carateres
E
Oi eu sou um Thread.
stou te
Oi eu sou um Thread.
stando
Oi eu sou um Thread.
o p
Oi eu sou um Thread.
rogram
Oi eu sou um Thread.
c
Oi eu sou um Thread.
om
Oi eu sou um Thread.
t
Oi eu sou um Thread.
hread
Oi eu sou um Thread.
as
Oi eu sou um Thread.
Estou testando o program com threadas
Pressione qualquer tecla para continuar. . .
Oi eu sou um Thread.
PDFCreator
Spybot - Search &
QuickTime Player

- Apesar do *Thread* dificultar a escrita, o programa recebe e armazena corretamente a mensagem, isso prova que são linhas de processamento independentes.

Outro Exemplo - 1

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

int tempo = 0;

void relogio ( void *i );

int main ()
{
   // Esta função abaixo faz parte de run-time library.
   _beginthread ( relogio, NULL , 0 );

   while ( 1 )
   {
      printf ( "Programa aberto faz %i segundos. \n ", tempo);
      getchar();
   }

   return 0;
}
```

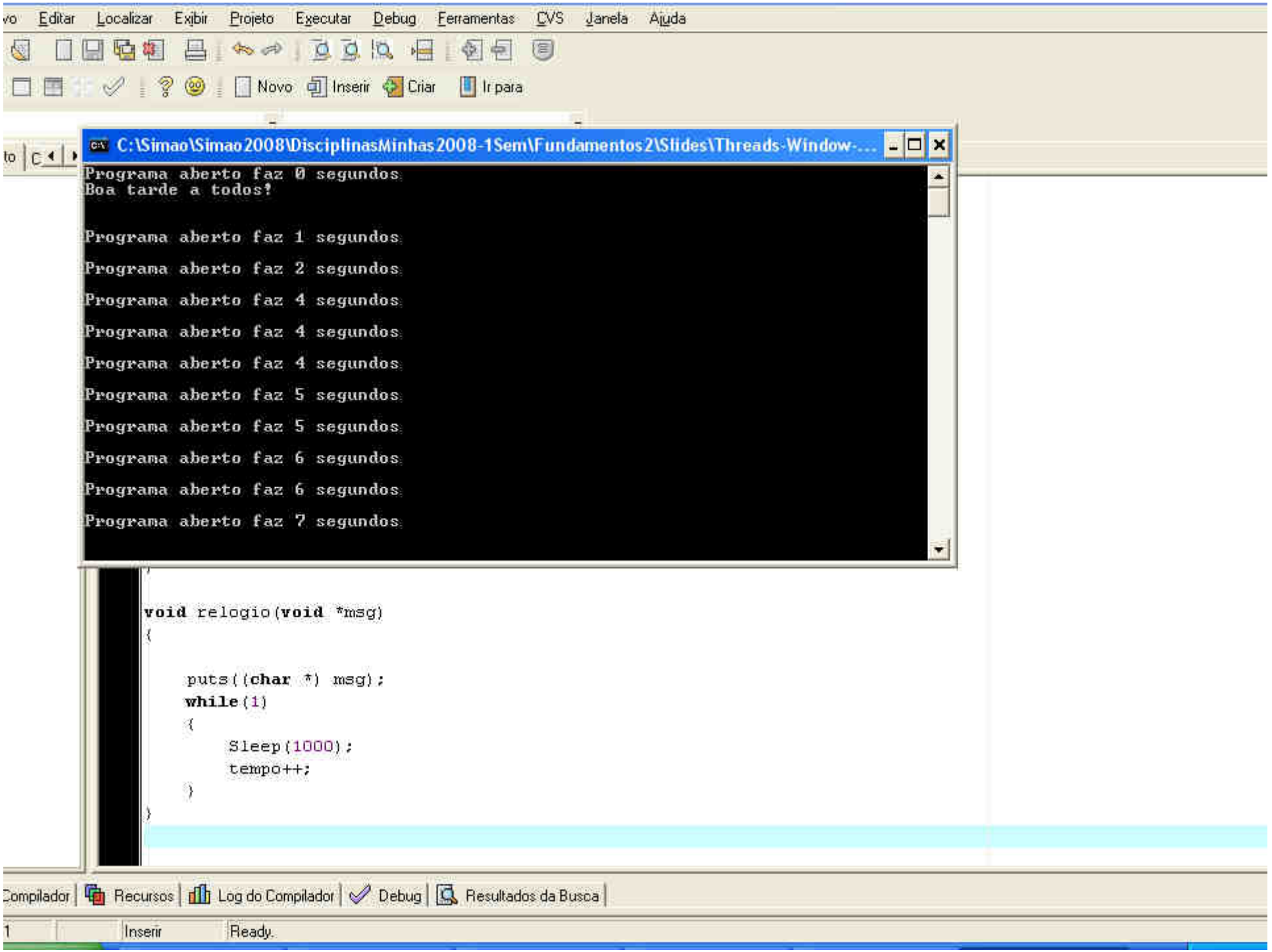
```
void relogio ( void *i )
{
   while ( 1 )
   {
      Sleep (1000);
      tempo++;
   }
}
```


Lixeira
WinZip
Spybot - Search &
C:\Simao\Simao2008\DisciplinasMinhas2008-1Sem\Fundamentos2\Slides\Threads-Window-...
Programa aberto faz 0 segundos.
Programa aberto faz 3 segundos.
Programa aberto faz 3 segundos.
Programa aberto faz 4 segundos.
Programa aberto faz 6 segundos.
Programa aberto faz 8 segundos.
Programa aberto faz 9 segundos.
Programa aberto faz 10 segundos.
Programa aberto faz 15 segundos.
Programa aberto faz 17 segundos.
Programa aberto faz 17 segundos.
Programa aberto faz 18 segundos.
Programa aberto faz 18 segundos.
Programa aberto faz 20 segundos.
Programa aberto faz 20 segundos.
Programa aberto faz 22 segundos.
Programa aberto faz 23 segundos.

Outro Exemplo - 2

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

int tempo = 0;

void relogio ( void *msg );

int main ( int argc, char* argv[] )
{
   char mensagem [ 50 ];
   strcpy ( mensagem, "Boa tarde a todos! \n" );
   _beginthread ( relogio, 0, (void *) mensagem );
   //Sleep(100);
   while ( 1 )
   {
      printf ( "Programa aberto faz %i segundos \n", tempo);
      getchar ();
   }
   return 0;
}
```

```
void relogio ( void *msg )
{

   puts ( (char *) msg );

   while ( 1 )
   {
      Sleep ( 1000 );
      tempo++;
   }

}
```


Editar Localizar Exibir Projeto Executar Debug Ferramentas CVS Janela Ajuda
Novo Inserir Criar Ir para
C:\Simao\Simao2008\DisciplinasMinhas2008-1Sem\Fundamentos2\Slides\Threads-Window-...
Programa aberto faz 0 segundos
Boa tarde a todos!
Programa aberto faz 1 segundos
Programa aberto faz 2 segundos
Programa aberto faz 4 segundos
Programa aberto faz 4 segundos
Programa aberto faz 4 segundos
Programa aberto faz 5 segundos
Programa aberto faz 5 segundos
Programa aberto faz 6 segundos
Programa aberto faz 6 segundos
Programa aberto faz 7 segundos
void relogio(void *msg)
{
puts((char *) msg);
while(1)
{
Sleep(1000);
tempo++;
}
}
Recursos
Log do Compilador
Debug
Resultados da Busca
Inserir
Ready.

Entendendo

_beginthread

Beginthread

From Wikipedia, the free encyclopedia

In the C programming language, the beginthread function is declared in process.h header file. This function creates a new thread of execution within the current process.

Contents

Prototype

```
unsigned long _beginthread(void(* Func)(void*), unsigned Stack_size, void *Arg);
```

Func

Thread execution starts at the beginning of the function func. To terminate the thread correctly, func must call _endthread, freeing memory allocated by the run time library to support the thread.

Stack_size

The operating system allocates a stack for the thread containing the number of bytes specified by stack_size. If the value of stack_size is zero, the operating system creates a stack the same size as that of the main thread.[1]

Arg

The operating system passes Arg to Func when execution begins. Arg can be any 32-bit value cast to void*.

Return value

Returns the operating system handle of the newly created thread. If unsuccessful, the function returns -1 and sets errno.

Compiler switches

To compile a program using multiple threads with the Microsoft C/C++ Compiler, you must specify the /MT switch (or /MTd, for debug programs).

References

^ MSDN _beginthread

_beginthread, _beginthreadex

The _beginthread function creates a thread that begins execution of a routine at start_address. The routine at start_address must use the __cdecl calling convention and should have no return value. When the thread returns from that routine, it is terminated automatically. For more information about threads, see Multithreading.

_beginthreadex resembles the Win32 CreateThread API more closely than _beginthread does. _beginthreadex differs from _beginthread in the following ways:

- _beginthreadex has three additional parameters: initflag, security, and threadaddr. The new thread can be created in a suspended state, with a specified security (Windows NT only), and can be accessed using thrdaddr, which is the thread identifier.
- The routine at start_address passed to _beginthreadex must use the __stdcall calling convention and must return a thread exit code.
- _beginthreadex returns 0 on failure, rather than -1L.
- A thread created with _beginthreadex is terminated by a call to _endthreadex.

The _beginthreadex function gives you more control over how the thread is created than _beginthread does. The _endthreadex function is also more flexible. For example, with _beginthreadex, you can use security information, set the initial state of the thread (running or suspended), and get the thread identifier of the newly created thread. You are also able to use the thread handle returned by _beginthreadex with the synchronization APIs, which you cannot do with _beginthread.

It is safer to use _beginthreadex than _beginthread. If the thread generated by _beginthread exits quickly, the handle returned to the caller of _beginthread might be invalid or, worse, point to another thread. However, the handle returned by _beginthreadex has to be closed by the caller of _beginthreadex, so it is guaranteed to be a valid handle if _beginthreadex did not return an error.

You can call _endthread or _endthreadex explicitly to terminate a thread; however, _endthread or _endthreadex is called automatically when the thread returns from the routine passed as a parameter. Terminating a thread with a call to endthread or _endthreadex helps to ensure proper recovery of resources allocated for the thread.

_endthread automatically closes the thread handle (whereas _endthreadex does not). Therefore, when using _beginthread and _endthread, do not explicitly close the thread handle by calling the Win32 CloseHandle API. This behavior differs from the Win32 ExitThread API.

Note: For an executable file linked with Libcmt.lib, do not call the Win32 ExitThread API; this prevents the run-time system from reclaiming allocated resources. _endthread and _endthreadex reclaim allocated thread resources and then call ExitThread.

The operating system handles the allocation of the stack when either _beginthread or _beginthreadex is called; you do not need to pass the address of the thread stack to either of these functions. In addition, the stack_size argument can be 0, in which case the operating system uses the same value as the stack specified for the main thread.

Arglist is a parameter to be passed to the newly created thread. Typically it is the address of a data item, such as a character string. arglist can be NULL if it is not needed, but _beginthread and _beginthreadex must be provided with some value to pass to the new thread. All threads are terminated if any thread calls abort, exit, _exit, or ExitProcess.

The locale of the new thread is inherited from its parent thread. If per thread locale is enabled by a call to _configthreadlocale (either globally or for new threads only), the thread can change its locale independently from its parent by calling setlocale or _wsetlocale. For more information, see Locale.

For mixed and pure code, _beginthread and _beginthreadex both have two overloads, one taking a native calling-convention function pointer, the other taking a __clrcall function pointer.
-The first overload is not application domain-safe and never will be. If you are writing mixed or pure code you must ensure that the new thread enters the correct application domain before it accesses managed resources. You can do this, for example, by using call_in_appdomain Function.
- The second overload is application domain-safe; the newly created thread will always end up in the application domain of the caller of _beginthread or _beginthreadex.

Requirements for *_beginthreadex* and *_beginthread*: <process.h>

Atividades ou Exercícios

- Estudar outras funções da C Run-time Library.
- Estudar a utilização de *mutex* com C Run-time Library.
- Estudar a utilização de *semáforos* com C Run-time Library.
- Pesquisar/Estudar como programar com com C Run-time Library de maneira orientada a objetos.
  - *www.cpdee.ufmg.br/~seixas/PaginaATR/Download/DownloadFiles/Theadsin**C++**.PDF*

MUTEX

- Tarefas mutuamente exclusivas.
- Servem para bloquear o acesso de uma região, caso já exista um *Thread* ativo nela.
- Quando essa região está desocupada, o outro *Thread* se inicia automaticamente.

Semáforo

- *Threads* são realizáveis a partir do momento em que foram criadas. Portanto todas os *threads* que um programa criar serão executadas concorrentemente à *main()* e às outras *threads*.

- Semáforos são ferramentas que limitam o número de *threads* 'simultâneas'.

Windows 32 API

Windows API

Win 32 API

- O Win32 API (também conhecido como Windows API) é um arquétipo (*framework*) baseado em C para a criação de aplicações Windows, e já estava disponível à partir das primeiras versões do Windows.

Exemplo de Threads

Win32 API

Exemplo de Mutex

Este exemplo utiliza funções da Win 32 API.

Obs.: *The Win32 API (also known as the Windows API) is a C-based framework for creating Windows applications, and has been around since Windows 1.0.*

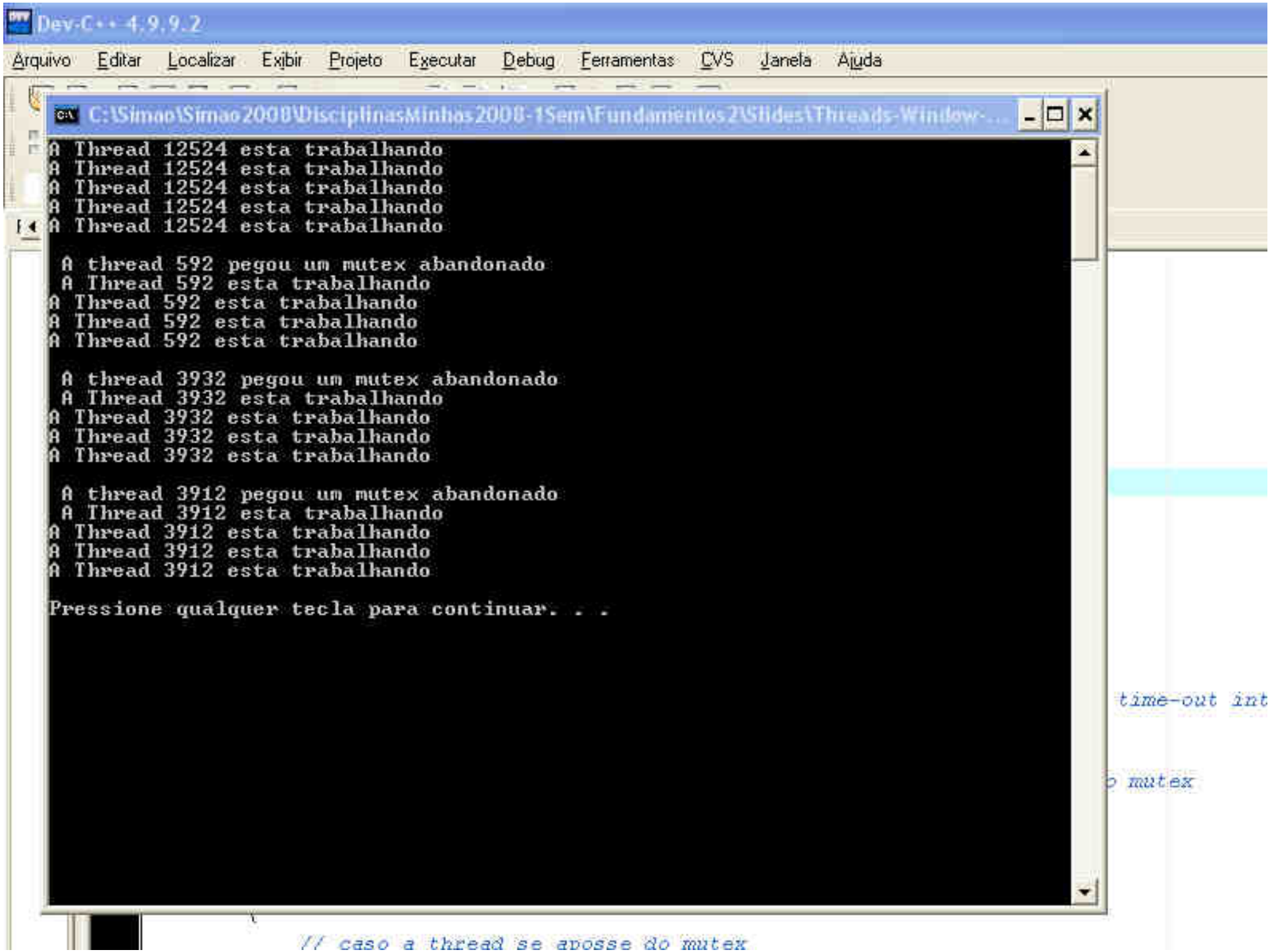
```
//  A quem interessar possa :

//  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;


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

#define NumeroDeThreads 4     // Constante para o Número de Threads

HANDLE MeuMutex;              // Uma variável Handle é um "ponteiro especial" ...

void AtivarThread ( );
```

http://en.wikipedia.org/wiki/Handle_(computing)

http://en.wikibooks.org/wiki/Windows_Programming/Handles_and_Data_Types

```
int main () {
   HANDLE Vetor_De_Threads [ NumeroDeThreads ];  // Vetor que será usado para manipular os Threads criados
   DWORD  Identificador_Da_Thread;               // DWORD : 32-bit unsigned integer. The range is 0 through 4294967295 decimal.
   int i;

   // Cria um mutex (sem dono)
   MeuMutex = CreateMutex  (    NULL,          // parâmetro relacionado a segurança do mutex
                                FALSE,         // parâmetro que indica a ausência de dono
                                NULL           // parâmetro que dá um nome ao mutex          );

   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,                                  // parâmetro relacionado a segurança da thread
                                               0,                                     // stack size padrão
                                               ( LPTHREAD_START_ROUTINE ) AtivarThread,
                                               NULL,                                  // argumentos da função executada
                                               0,                                     // criação de flags
                                               &Identificador_Da_Thread               // ID do thread                    );

      if  ( Vetor_De_Threads [ i ] == NULL )
      {
         printf ( " Erro na funcao CreateThread: %d \n ", GetLastError() );
         return 0;
      }
   } // Threads criadas e rodando

   WaitForMultipleObjects ( NumeroDeThreads , Vetor_De_Threads, TRUE, INFINITE );          // Espera os threads
   for ( i = 0; i < NumeroDeThreads; i++ )  {  CloseHandle ( Vetor_De_Threads [ i ] );  }  // Destroi as variáveis

   CloseHandle ( MeuMutex );   printf ( "\n" );
   system ( "PAUSE" );
   return 0;
}
```

```
void AtivarThread ( )
{
   int Ciclos = 0;
   int EstadoAtual;

   // Requisita a posse do mutex.
   while ( Ciclos < 5 )
   {
      // Waits until the specified object is in the signaled state or the time-out interval elapses.
      EstadoAtual = WaitForSingleObject
                              (
                                  MeuMutex,    // handle para o mutex
                                  INFINITE     // sem time-out
                               );

      switch ( EstadoAtual )
      {
         // caso a thread se aposse do mutex
         case WAIT_OBJECT_0 :
         {
            DWORD identificador = 0;
            identificador = GetCurrentThreadId();    // Retrieves the thread identifier of the calling thread.

            printf ( " A Thread %d esta trabalhando \n ", identificador );
            Ciclos++;
         }
         break;

         // caso o thread se aposse de um mutex abandonado
         case WAIT_ABANDONED :
         {
            printf ( " \n A thread %d pegou um mutex abandonado \n ", GetCurrentThreadId() );
            Ciclos++;
         }
      }
   }
}
```


Dev-C++ 4.9.9.2
Arquivo Editar Localizar Exibir Projeto Executar Debug Ferramentas CVS Janela Ajuda
C:\Simao\Simao2008\DisciplinasMinhas2008-1Sem\Fundamentos2\Slides\Threads-Window-...
A Thread 12524 esta trabalhando
A Thread 12524 esta trabalhando
A Thread 12524 esta trabalhando
A Thread 12524 esta trabalhando
A Thread 12524 esta trabalhando
A thread 592 pegou um mutex abandonado
A Thread 592 esta trabalhando
A Thread 592 esta trabalhando
A Thread 592 esta trabalhando
A Thread 592 esta trabalhando
A thread 3932 pegou um mutex abandonado
A Thread 3932 esta trabalhando
A Thread 3932 esta trabalhando
A Thread 3932 esta trabalhando
A Thread 3932 esta trabalhando
A thread 3912 pegou um mutex abandonado
A Thread 3912 esta trabalhando
A Thread 3912 esta trabalhando
A Thread 3912 esta trabalhando
A Thread 3912 esta trabalhando
Pressione qualquer tecla para continuar. . .
time-out int
o mutex
// caso a thread se aposse do mutex

Atividades ou Exercícios

- Estudar outras funções da Win32 API.
- Estudar a utilização de *semáforos* com WIN32 API.

Semáforo - relembrando

- *Threads* são realizáveis a partir do momento em que foram criadas. Portanto todas os *threads* que um programa criar serão executadas concorrentemente ao *main()* e às outras *threads*.

- Semáforos são ferramentas que limitam o número de *threads* 'simultâneas'.

Ponto *Net*

Namespace System::Threading
do C++ .net

Namespace System::Threading do C++ .net

- Um espaço de nomes do C++ para .net que facilita a manipulação de *threads*.

Exemplo de Threads

Namespace System::Threading
do C++ .net

Exemplo de Semáforo

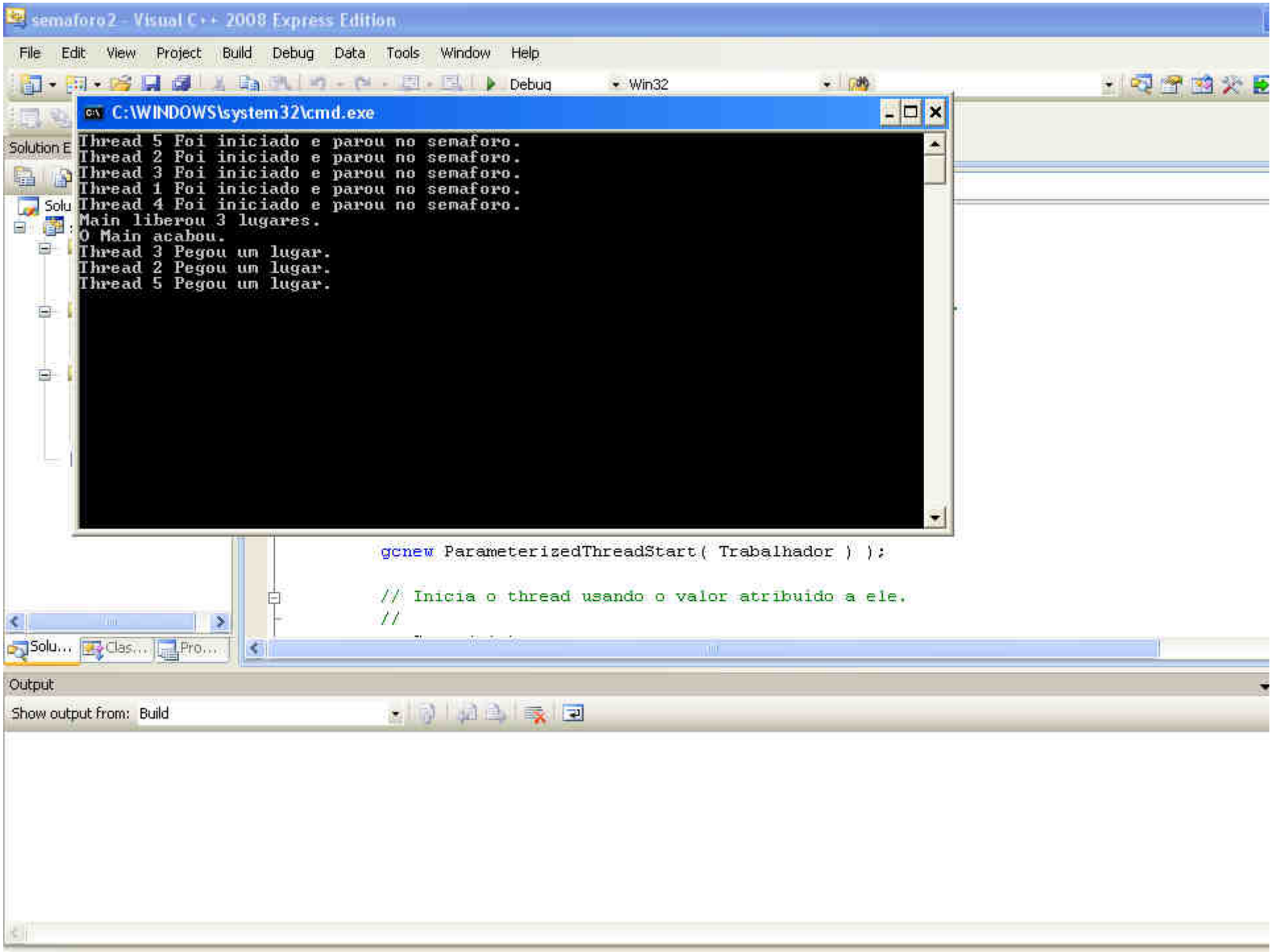
Este exemplo utiliza o *namespace System::Threading* do C++ .net

```
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>

#using <System.dll>

using namespace System;
using namespace System::Threading;

public ref class Exemplo
{

private:
  // Criação do semáforo.
   static Semaphore^ Semaforo;

  // Intervalo de tempo para sincronizar o programa.
  static int intervalo;

public:

      // . . .
```

```
// . . .
static void Main ()
{
    // Cria um semáforo que atende até 3 pedidos concorrentes.  Seu estado inicial é zero, ou seja, está bloqueando
    // qualquer thread que venha a ser executado
    Semaforo = gcnew Semaphore ( 0, 3 );

    // Cria e inicia 5 threads.
    for ( int i = 1; i <= 5;  i++ )
    {
      Thread^ t = gcnew Thread (  gcnew ParameterizedThreadStart ( Trabalhador ) );
      // Inicia o thread usando o valor atribuído a ele.
      t->Start ( i );
    }

    // Espera meio segundo para que de tempo de todos as threads se iniciarem e pararem no semáforo.
    Thread::Sleep ( 500 );

    // Quando o método Release() for chamado, haverá a liberação  de um determinado número de vagas. Isso significa que o semáforo vai
    // trocar o estado inicial 0, pelo estado indicado pelo método release(). No nosso exemplo esse estado pode ir até três.
    Console::WriteLine ( L"Main liberou 3 lugares." );
    Semaforo->Release ( 3 );
    Console::WriteLine ( L"O Main acabou." );
}
// . . .
```

```
// . . .
private:
  static void Trabalhador ( Object^ num )
  {
    // Cada Trabalhador começa encontrando o semáforo.
    Console::WriteLine ( L"Thread {0} Foi iniciado e parou no semaforo.",  num );
    Semaforo->WaitOne();

    // Aqui é adicionado um intervalo de tempo que serve apenas para melhorar
    // a visualização das saídas. Isso é feito de forma que cada thread espere
    // um pouco mais que o anterior.
    int Intervalo = Interlocked::Add( Intervalo, 100 );

    Console::WriteLine ( L"Thread {0} Pegou um lugar.", num );
    // Apesar do nome Trabalhador, a única tarefa atribuída ao thread
    // é dormir  =/ .

    Thread::Sleep( 1000 + Intervalo );

    Console::WriteLine ( L"Thread {0} liberou um lugar.", num );
    Semaforo->Release();
  }
};
// . . .
```

```
// . . .
int main ()
{
        Exemplo ex;

        ex.Main();

        getchar ();

        return ( 0 );
}
```


semaforo2 - Visual C++ 2008 Express Edition
File Edit View Project Build Debug Data Tools Window Help
Debug
Win32
C:\WINDOWS\system32\cmd.exe
Thread 5 Foi iniciado e parou no semaforo.
Thread 2 Foi iniciado e parou no semaforo.
Thread 3 Foi iniciado e parou no semaforo.
Thread 1 Foi iniciado e parou no semaforo.
Thread 4 Foi iniciado e parou no semaforo.
Main liberou 3 lugares.
O Main acabou.
Thread 3 Pegou um lugar.
Thread 2 Pegou um lugar.
Thread 5 Pegou um lugar.
gcnew ParameterizedThreadStart(Trabalhador));
// Inicia o thread usando o valor atribuido a ele.
//
Output
Show output from: Build

Exercício

- Refazer o exemplo de semáforos usando as funções da C Run-time Library...
  - Obs.: Será necessário pesquisar outras funções além das apresentadas.
  - *www.cpdee.ufmg.br/~seixas/PaginaATR/Download/DownloadFiles/TheadsinC++.PDF*
- Refazer o exemplo de semáforos usando as funções da Win 32 API...
  - Obs.: Será necessário pesquisar outras funções além das apresentadas.

Bibliografias relativas a *Threads*.

- SCHILDT, H.: **The Art of C++.** McGraw-Hill Osborne Media. 1ª Edição (Paperback) 2004. ISBN-10: 0072255129
- RICHARD, H. C.; KUO-CHUNG, T.: **Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs.** Wiley-Interscience (Paperback) 2005. ISBN-10: 0471725048
- HUGHES C; HUGHES T.: **Object-Oriented Multithreading Using C++.** Wiley; (Paperback) 1997. ISBN-10: 0471180122.
- HUGHES C; HUGHES T.: **Professional Multicore Programming: Design and Implementation for C++ Developers**. Wrox (Paperback) 2008. ISBN-10: 0470289627