Multithreading with C and Win32

Home | Overview | How Do I | Sample

Microsoft Visual C++ provides support for creating multithread applications with 32-bit versions of Microsoft Windows: Windows NT and Windows 95. You should consider using more than one thread if your application needs to manage multiple activities, such as simultaneous keyboard and mouse input. One thread can process keyboard input while a second thread filters mouse activities. A third thread can update the display screen based on data from the mouse and keyboard threads. At the same time, other threads can access disk files or get data from a communications port.

With Visual C++, there are two ways to program with multiple threads: use the Microsoft Foundation Class library (MFC) or the C run-time library and the Win32 API. For information on creating multithread applications with MFC, read the Multithreading with C++ and MFC articles after reading these articles about multithreading in C.

This article family explains the features in Visual C++ that support the creation of multithread programs.

> **Note** Win32s does not support multithreading. Calls to the Win32 APIs and C run-time library functions mentioned in this article family will return an error.

What do you want to know more about?

Multithread Programs

Home | Overview | How Do I | Sample

A thread is basically a path of execution through a program. It is also the smallest unit of execution that Win32 schedules. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all of the process's resources.

A process consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority may have to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to "balance" the CPU load.

Each thread in a process operates independently. Unless you make them visible to each other, the threads execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, must coordinate their work by using semaphores or another method of interprocess communication. See Writing a Multithreaded Win32 Program for more information about synchronizing threads.

Library Support for Multithreading

Home | Overview | How Do I | Sample

If one thread is suspended by the Win32 scheduler while executing the **printf** function, one of the program's other threads might start executing. If the second thread also calls **printf**, data might be corrupted. To avoid this situation, access to static data used by the function must be restricted to one thread at a time.

You do not need to serialize access to stack-based (automatic) variables because each thread has a different stack. Therefore, a function that uses only automatic (stack) variables is reentrant. The standard C run-time libraries, such as LIBC, have a limited number of reentrant functions. A multithread program needing to use C run-time library functions that are normally not reentrant should be built with the multithread library LIBCMT.LIB.

The Multithread C Libraries: LIBCMT.LIB and MSVCRT.LIB

The support library LIBCMT.LIB is a reentrant library for creating multithread programs. The MSVCRT.LIB library, which calls code in the shared MSVCRT40.DLL, is also reentrant. When your application calls functions in these libraries, the following rules may apply:

- All library calls must use the C (**__cdecl**) calling convention; programs compiled using other calling conventions (such as **__fastcall** or **__stdcall**) must use the standard include files for the run-time library functions they call.
- Variables passed to library functions must be passed by value or cast to a pointer.

Programs built with LIBCMT.LIB do not share C run-time library code or data with any dynamic-link libraries they call.

Alternatives to LIBCMT.LIB and MSVCRT.LIB

If you build a multithread program without using LIBCMT.LIB, you must do the following:

- Use the standard C libraries and limit library calls to the set of reentrant functions.
- Use the Win32 API thread management functions, such as CreateThread.
- Provide your own synchronization for functions that are not reentrant by using Win32 services such as semaphores and the EnterCriticalSection and LeaveCriticalSection functions.

Warning The multithread library LIBCMT.LIB includes the **_beginthread** and **_endthread** functions. The **_beginthread** function performs initialization without which many C run-time functions will fail. You must use **_beginthread** instead of **CreateThread** in C programs built with LIBCMT.LIB if you intend to call C run-time functions.

The Multithread Libraries Compile Option

To build a multithread application that uses the C run-time libraries, you must tell the compiler to use a special version of the libraries (LIBCMT.LIB). To select these libraries, first open the Project Settings dialog box (Build menu) and click the C/C++ tab. Select Code Generation from the

Category drop-down list box. From the Use Run-Time Library drop-down box, select Multithreaded. Click OK to return to editing.

From the command line, the Multithread Library compiler option (/MT) is the best way to build a multithread program with LIBCMT.LIB. This option, which is automatically set when you specify a multithreaded application when creating a new project, embeds the LIBCMT library name in the object file.

Include Files for Multithreading

Home | Overview | How Do I | Sample

The Microsoft Visual C++ include files contain conditional sections for multithread applications using LIBCMT.LIB. To compile your application with the appropriate definitions, you can:

- Compile with the Multithread Library compiler option described in the previous section.
- Define the symbolic constant **_MT** in your source file or on the command line with the /D compiler option.

Standard include files declare C run-time library functions as they are implemented in the libraries. If you use the Full Optimization (/Ox) or fastcall Calling Convention (/Gr) option, the compiler assumes that all functions should be called using the register calling convention. The run-time library functions were compiled using the C calling convention, and the declarations in the standard include files tell the compiler to generate correct external references to these functions.

See Compiling and Linking Multithread Programs for examples of how to use the **_MT** constant.

C Run-Time Library Functions for Thread Control

Home | Overview | How Do I | Sample

All Win32 programs have at least one thread. Any thread can create additional threads. A thread can complete its work quickly and then terminate, or it can stay active for the life of the program.

The LIBCMT and MSVCRT C run-time libraries provide two functions for thread creation and termination: **_beginthread** and **_endthread**.

The **_beginthread** function creates a new thread and returns a thread identifier if the operation is successful. The thread terminates automatically if it completes execution, or it can terminate itself with a call to **_endthread**.

> **! Warning** If you are going to call C run-time routines from a program built with LIBCMT.LIB, you must start your threads with the **_beginthread** function. Do not use the Win32 functions **ExitThread** and **CreateThread**. Using **SuspendThread** can lead to a deadlock when more than one thread is blocked waiting for the suspended thread to complete its access to a C run-time data structure.

The _beginthread Function

The _beginthread function creates a new thread. A thread shares the code and data segments of a process with other threads in the process, but has its own unique register values, stack space, and current instruction address. The system gives CPU time to each thread, so that all threads in a process can execute concurrently.

The **_beginthread** function is similar to the CreateThread function in the Win32 API but has these differences:

- The **_beginthread** function lets you pass multiple arguments to the thread.
- The **_beginthread** function initializes certain C run-time library variables. This is important only if you use the C run-time library in your threads.
- **CreateThread** provides control over security attributes. You can use this function to start a thread in a suspended state.

The **_beginthread** function returns a handle to the new thread if successful or –1 if there was an error.

The _endthread Function

The _endthread function terminates a thread created by **_beginthread**. Threads terminate automatically when they finish. The **_endthread** function is useful for conditional termination from within a thread. A thread dedicated to communications processing, for example, can quit if it is unable to get control of the communications port.

Sample Multithread C Program

Home | Overview | How Do I | Sample

BOUNCE.C is a sample multithread program that creates a new thread each time the letter **a** or **A** is typed. Each thread bounces a “happy face” of a different color around the screen. Up to 32 threads can be created. The program’s normal termination occurs when **q** or **Q** is typed. See Compiling and Linking Multithread Programs for details on compiling and linking BOUNCE.C.

```
/*  Bounce - Creates a new thread each time the letter 'a' is typed.
 *  Each thread bounces a happy face of a different color around the screen.
 *  All threads are terminated when the letter 'Q' is entered.
 *
 *  This program requires the multithread library. For example, compile
 *  with the following command line:
 *      CL /MT BOUNCE.C
 */

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

#define MAX_THREADS  32

/* getrandom returns a random number between min and max, which must be in
 * integer range.
 */
#define getrandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))

void main( void );                      /* Thread 1: main */
void KbdFunc( void  );                  /* Keyboard input, thread dispatch */
void BounceProc( char * MyID );         /* Threads 2 to n: display */
void ClearScreen( void );               /* Screen clear */
void ShutDown( void );                  /* Program shutdown */
void WriteTitle( int ThreadNum );       /* Display title bar information */

HANDLE  hConsoleOut;                    /* Handle to the console */
HANDLE  hRunMutex;                      /* "Keep Running" mutex */
HANDLE  hScreenMutex;                   /* "Screen update" mutex  */
int     ThreadNr;                       /* Number of threads started */
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;    /* Console information */


void main()                             /* Thread One */
{
    /* Get display screen information & clear the screen.*/
    hConsoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
    GetConsoleScreenBufferInfo( hConsoleOut, &csbiInfo );
    ClearScreen();
    WriteTitle( 0 );
    /* Create the mutexes and reset thread count. */
    hScreenMutex = CreateMutex( NULL, FALSE, NULL );   /* Cleared */
    hRunMutex = CreateMutex( NULL, TRUE, NULL );       /* Set */
    ThreadNr = 0;

    /* Start waiting for keyboard input to dispatch threads or exit. */
    KbdFunc();

    /* All threads done. Clean up handles. */
    CloseHandle( hScreenMutex );
    CloseHandle( hRunMutex );
    CloseHandle( hConsoleOut );
}

void ShutDown( void )                   /* Shut down threads */
{
    while ( ThreadNr > 0 )
    {
            /* Tell thread to die and record its death. */
            ReleaseMutex( hRunMutex );
            ThreadNr--;
```

```
    }
    /* Clean up display when done */
    WaitForSingleObject( hScreenMutex, INFINITE );
    ClearScreen();
}

void KbdFunc( void )                    /* Dispatch and count threads. */
{
    int         KeyInfo;

    do
    {
        KeyInfo = _getch();
        if( tolower( KeyInfo ) == 'a' && ThreadNr < MAX_THREADS )
        {
            ThreadNr++;
            _beginthread( BounceProc, 0, &ThreadNr );
            WriteTitle( ThreadNr );
        }
    } while( tolower( KeyInfo ) != 'q' );

    ShutDown();
}

void BounceProc( char *MyID )
{
    char      MyCell, OldCell;
    WORD      MyAttrib, OldAttrib;
    char      BlankCell = 0x20;
    COORD     Coords, Delta;
    COORD     Old = {0,0};
    DWORD     Dummy;

/* Generate update increments and initial display coordinates. */
    srand( (unsigned) *MyID * 3 );
    Coords.X = getrandom( 0, csbiInfo.dwSize.X - 1 );
    Coords.Y = getrandom( 0, csbiInfo.dwSize.Y - 1 );
    Delta.X = getrandom( -3, 3 );
    Delta.Y = getrandom( -3, 3 );

    /* Set up "happy face" & generate color attribute from thread number.*/
    if( *MyID > 16)
        MyCell = 0x01;                  /* outline face */
    else
        MyCell = 0x02;                  /* solid face */
    MyAttrib =  *MyID & 0x0F;           /* force black background */

    do
    {
        /* Wait for display to be available, then lock it. */
        WaitForSingleObject( hScreenMutex, INFINITE );

        /* If we still occupy the old screen position, blank it out. */
        ReadConsoleOutputCharacter( hConsoleOut, &OldCell, 1, Old, &Dummy );
        ReadConsoleOutputAttribute( hConsoleOut, &OldAttrib, 1, Old, &Dummy );
        if (( OldCell == MyCell ) && (OldAttrib == MyAttrib))
            WriteConsoleOutputCharacter( hConsoleOut, &BlankCell, 1, Old, &Dummy );

        /* Draw new face, then clear screen lock */
        WriteConsoleOutputCharacter( hConsoleOut, &MyCell, 1, Coords, &Dummy );
        WriteConsoleOutputAttribute( hConsoleOut, &MyAttrib, 1, Coords, &Dummy );
        ReleaseMutex( hScreenMutex );

        /* Increment the coordinates for next placement of the block. */
        Old.X = Coords.X;
        Old.Y = Coords.Y;
        Coords.X += Delta.X;
        Coords.Y += Delta.Y;

        /* If we are about to go off the screen, reverse direction */
        if( Coords.X < 0 || Coords.X >= csbiInfo.dwSize.X )
        {
            Delta.X = -Delta.X;
            Beep( 400, 50 );
        }
        if( Coords.Y < 0 || Coords.Y > csbiInfo.dwSize.Y )
        {
            Delta.Y = -Delta.Y;
            Beep( 600, 50 );
```

```
        }
    }
    /* Repeat while RunMutex is still taken. */
    while ( WaitForSingleObject( hRunMutex, 75L ) == WAIT_TIMEOUT );

}

void WriteTitle( int ThreadNum )
{
    char    NThreadMsg[80];

    sprintf( NThreadMsg, "Threads running: %02d.  Press 'A' to start a thread,'Q' to quit.", ThreadNum );
    SetConsoleTitle( NThreadMsg );
}

void ClearScreen( void )
{
    DWORD    dummy;
    COORD    Home = { 0, 0 };
    FillConsoleOutputCharacter( hConsoleOut, ' ', csbiInfo.dwSize.X * csbiInfo.dwSize.Y, Home, &dummy );
}
```

Writing a Multithreaded Win32 Program

Home | Overview | How Do I | Sample

When you write a program with multiple threads, you must coordinate their behavior and use of the program's resources. You must also make sure that each thread receives its own stack.

Sharing Common Resources Between Threads

> **Note** For a similar discussion from the MFC point of view, see Multithreading: Programming Tips and Multithreading: When to Use the Synchronization Classes.

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, static data, and heap memory, are shared by all threads in the process. Threads using these common resources must be synchronized. Win32 provides several ways to synchronize resources, including semaphores, critical sections, events, and mutexes.

When multiple threads are accessing static data, your program must provide for possible resource conflicts. Consider a program where one thread updates a static data structure containing *x*,*y* coordinates for items to be displayed by another thread. If the update thread alters the *x* coordinate and is preempted before it can change the *y* coordinate, the display thread may be scheduled before the *y* coordinate is updated. The item would be displayed at the wrong location. You can avoid this problem by using semaphores to control access to the structure.

A mutex (short for *mut*ual *ex*clusion) is a way of communicating among threads or processes that are executing asynchronously of one another. This communication is usually used to coordinate the activities of multiple threads or processes, typically by controlling access to a shared resource by "locking" and "unlocking" the resource. To solve this *x*,*y* coordinate update problem, the update thread would set a mutex indicating that the data structure is in use before performing the update. It would clear the mutex after both coordinates had been processed. The display thread must wait for the mutex to be clear before updating the display. This process of waiting for a mutex is often called "blocking" on a mutex because the process is blocked and cannot continue until the mutex clears.

The BOUNCE.C program shown in the previous section uses a mutex named `ScreenMutex` to coordinate screen updates. Each time one of the display threads is ready to write to the screen, it calls **WaitForSingleObject** with the handle to `ScreenMutex` and constant **INFINITE** to indicate that the **WaitForSingleObject** call should block on the mutex and not time out. If `ScreenMutex` is clear, the wait function sets the mutex so other threads cannot interfere with the display and continues executing the thread. Otherwise, the thread blocks until the mutex clears. When the thread completes the display update, it releases the mutex by calling **ReleaseMutex**.

Screen displays and static data are only two of the resources requiring careful management. For example, your program may have multiple threads accessing the same file. Because another thread may have moved the file pointer, each thread must reset the file pointer before reading or writing. In addition, each thread must make sure that it is not preempted between the time it positions the pointer and the time it accesses the file. These threads should use a semaphore to coordinate access to the file by bracketing each file access with **WaitForSingleObject** and **ReleaseMutex** calls. The following code fragment illustrates this technique:

```
HANDLE    hIOMutex= CreateMutex (NULL, FALSE, NULL);

WaitForSingleObject( hIOMutex, INFINITE );
fseek( fp, desired_position, 0L );
```

```
fwrite( data, sizeof( data ), 1, fp );
ReleaseMutex( hIOMutex);
```

Thread Stacks

All of an application's default stack space is allocated to the first thread of execution, which is known as thread 1. As a result, you must specify how much memory to allocate for a separate stack for each additional thread your program needs. The operating system will allocate additional stack space for the thread, if necessary, but you must specify a default value.

The first argument in the **_beginthread** call is a pointer to the **BounceProc** function, which will execute the threads. The second argument specifies the default stack size for the thread. The last argument is an ID number that is passed to **BounceProc**. **BounceProc** uses the ID number to seed the random number generator and to select the thread's color attribute and display character.

Threads that make calls to the C run-time library or to the Win32 API must allow sufficient stack space for the library and API functions they call. The C **printf** function requires more than 500 bytes of stack space, and you should have 2K of stack space available when calling Win32 API routines.

Because each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. The only global variables in the BOUNCE.C program are either mutexes or variables that never change after they are initialized.

Win32 also provides Thread-Local Storage (TLS) to store per-thread data. See Thread Local Storage for more information.

Compiling and Linking Multithread Programs

Home | Overview | How Do I | Sample

To compile and link the multithread program BOUNCE.C from within the development environment

1. Click **New** on the **File** menu, then click the **Projects** tab.
2. On the **Projects** tab, click **Console Application**, and name the project.
3. Add the file containing the C source code to the project.
4. On the **Project** menu, click **Settings**. In the **Project Settings** dialog box, click the **C/C++** tab. Select **Code Generation** from the **Category** drop-down list box. From the **Use Run-Time Library** drop-down box, select **Multithreaded**. Click OK.
5. Build the project by clicking the **Build** command on the **Build** menu.

To compile and link the multithread program BOUNCE.C from the command line

1. Ensure that the Win32 library files and LIBCMT.LIB are in the directory specified in your LIB environment variable.
2. Compile and link the program with the CL command-line option /MT:

   ```
   CL /MT BOUNCE.C
   ```

3. If you choose not to use the /MT option, you must take these steps:

   - Define the **_MT** symbol before including header files. You can do this by specifying `/D _MT` on the command line.
   - Specify the multithread library and suppress default library selection.

   The multithread include files are used when you define the symbolic constant **_MT**. You can do this with the CL command line option `/D _MT` or within the C source file before any include statements, as follows:

   ```
   #define _MT
   #include <stdlib.h>
   ```

Avoiding Problem Areas with Multithread Programs

Home | Overview | How Do I | Sample

There are several problems you might encounter in creating, linking, or executing a multithread C program. Some of the more common ones are described here. (For a similar discussion from the MFC point of view, see Multithreading: Programming Tips.)

Problem	Probable cause
You get a message box showing that your program caused a protection violation.	Many Win32 programming errors cause protection violations. A common cause of protection violations is the indirect assignment of data to null pointers. This results in your program trying to access memory that does not “belong” to it, so a protection violation is issued. An easy way to detect the cause of a protection violation is to compile your program with debugging information, and then run it through the debugger in the Visual C++ environment. When the protection fault occurs, Windows transfers control to the debugger, and the cursor is positioned on the line that caused the problem.
Your program generates numerous compile and link errors.	If you attempt to compile and link a multithread program without defining the symbolic constant **_MT**, many of the definitions required for the multithread library will be missing. If you are using the Visual C++ development environment, make sure that the Project Settings dialog box specifies multithread libraries. From the command line, define **_MT** to CL with /MT or /D _MT, or use `#define _MT` in your program. You can eliminate many potential problems by setting the compiler’s warning level to one of its highest values and heeding the warning messages. By using the level 3 or level 4 warning level options, you can detect unintentional data conversions, missing function prototypes, and use of non-ANSI features.

Thread Local Storage (TLS)

Home | Overview | How Do I | Sample

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process may allocate locations in which to store thread-specific data. Dynamically bound (run-time) thread-specific data is supported by way of the TLS API (TlsAlloc, TlsGetValue, TlsSetValue, TlsFree). Win32 and the Visual C++ compiler now support statically bound (load-time) per-thread data in addition to the existing API implementation.

API Implementation for TLS

Thread Local Storage is implemented through the Win32 API layer as well as the compiler. For details, see the Win32 API documentation for TlsAlloc, TlsGetValue, TlsSetValue, and TlsFree.

The Visual C++ compiler includes a keyword to make thread local storage operations more automatic, rather than through the API layer. This syntax is described in the next section, Compiler Implementation for TLS.

Compiler Implementation for TLS

To support TLS, a new attribute, **thread**, has been added to the C and C++ languages and is supported by the Visual C++ compiler. This attribute is an extended storage class modifier, as described in the previous section. Use the **__declspec** keyword to declare a **thread** variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

See Also Rules and Limitations for TLS