Using the Thread Pool Functions

This example creates a custom thread pool, creates a work item and a thread pool timer, and associates them with a cleanup group. The pool consists of one persistent thread. It demonstrates the use of the following thread pool functions:

- CloseThreadpool
- CloseThreadpoolCleanupGroup
- CloseThreadpoolCleanupGroupMembers
- CloseThreadpoolWait
- CreateThreadpool
- CreateThreadpoolCleanupGroup
- CreateThreadpoolTimer
- CreateThreadpoolWait
- CreateThreadpoolWork
- InitializeThreadpoolEnvironment
- SetThreadpoolCallbackCleanupGroup
- SetThreadpoolCallbackPool
- SetThreadpoolThreadMaximum
- SetThreadpoolThreadMinimum
- SetThreadpoolTimer
- SetThreadpoolWait
- SubmitThreadpoolWork
- WaitForThreadpoolWaitCallbacks

C++

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

//
// Thread pool wait callback function template
//
VOID
CALLBACK
MyWaitCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_WAIT              Wait,
    TP_WAIT_RESULT        WaitResult
    )
{
    // Instance, Parameter, Wait, and WaitResult not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Wait);
    UNREFERENCED_PARAMETER(WaitResult);

    //
    // Do something when the wait is over.
    //
    _tprintf(_T("MyWaitCallback: wait is over.\n"));
}
```

```
//
// Thread pool timer callback function template
//
VOID
CALLBACK
MyTimerCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_TIMER             Timer
    )
{
    // Instance, Parameter, and Timer not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Timer);

    //
    // Do something when the timer fires.
    //
    _tprintf(_T("MyTimerCallback: timer has fired.\n"));

}


//
// This is the thread pool work callback function.
//
VOID
CALLBACK
MyWorkCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_WORK              Work
    )
{
    // Instance, Parameter, and Work not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Work);

    BOOL bRet = FALSE;

    //
    // Do something when the work callback is invoked.
    //
     {
         _tprintf(_T("MyWorkCallback: Task performed.\n"));
     }

    return;
}

VOID
DemoCleanupPersistentWorkTimer()
{
    BOOL bRet = FALSE;
    PTP_WORK work = NULL;
    PTP_TIMER timer = NULL;
    PTP_POOL pool = NULL;
    PTP_WORK_CALLBACK workcallback = MyWorkCallback;
    PTP_TIMER_CALLBACK timercallback = MyTimerCallback;
    TP_CALLBACK_ENVIRON CallBackEnviron;
    PTP_CLEANUP_GROUP cleanupgroup = NULL;
    FILETIME FileDueTime;
    ULARGE_INTEGER ulDueTime;
```

```
    UINT rollback = 0;

    InitializeThreadpoolEnvironment(&CallBackEnviron);

    //
    // Create a custom, dedicated thread pool.
    //
    pool = CreateThreadpool(NULL);

    if (NULL == pool) {
        _tprintf(_T("CreateThreadpool failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 1; // pool creation succeeded

    //
    // The thread pool is made persistent simply by setting
    // both the minimum and maximum threads to 1.
    //
    SetThreadpoolThreadMaximum(pool, 1);

    bRet = SetThreadpoolThreadMinimum(pool, 1);

    if (FALSE == bRet) {
        _tprintf(_T("SetThreadpoolThreadMinimum failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    //
    // Create a cleanup group for this thread pool.
    //
    cleanupgroup = CreateThreadpoolCleanupGroup();

    if (NULL == cleanupgroup) {
        _tprintf(_T("CreateThreadpoolCleanupGroup failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 2;  // Cleanup group creation succeeded

    //
    // Associate the callback environment with our thread pool.
    //
    SetThreadpoolCallbackPool(&CallBackEnviron, pool);

    //
    // Associate the cleanup group with our thread pool.
    // Objects created with the same callback environment
    // as the cleanup group become members of the cleanup group.
    //
    SetThreadpoolCallbackCleanupGroup(&CallBackEnviron,
                                      cleanupgroup,
                                      NULL);

    //
    // Create work with the callback environment.
    //
    work = CreateThreadpoolWork(workcallback,
                                NULL,
                                &CallBackEnviron);

    if (NULL == work) {
```

```
        _tprintf(_T("CreateThreadpoolWork failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 3;  // Creation of work succeeded

    //
    // Submit the work to the pool. Because this was a pre-allocated
    // work item (using CreateThreadpoolWork), it is guaranteed to execute.
    //
    SubmitThreadpoolWork(work);


    //
    // Create a timer with the same callback environment.
    //
    timer = CreateThreadpoolTimer(timercallback,
                                  NULL,
                                  &CallBackEnviron);


    if (NULL == timer) {
        _tprintf(_T("CreateThreadpoolTimer failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 4;  // Timer creation succeeded

    //
    // Set the timer to fire in one second.
    //
    ulDueTime.QuadPart = (ULONGLONG) -(1 * 10 * 1000 * 1000);
    FileDueTime.dwHighDateTime = ulDueTime.HighPart;
    FileDueTime.dwLowDateTime  = ulDueTime.LowPart;

    SetThreadpoolTimer(timer,
                       &FileDueTime,
                       0,
                       0);

    //
    // Delay for the timer to be fired
    //
    Sleep(1500);

    //
    // Wait for all callbacks to finish.
    // CloseThreadpoolCleanupGroupMembers also releases objects
    // that are members of the cleanup group, so it is not necessary
    // to call close functions on individual objects
    // after calling CloseThreadpoolCleanupGroupMembers.
    //
    CloseThreadpoolCleanupGroupMembers(cleanupgroup,
                                       FALSE,
                                       NULL);

    //
    // Already cleaned up the work item with the
    // CloseThreadpoolCleanupGroupMembers, so set rollback to 2.
    //
    rollback = 2;
    goto main_cleanup;

main_cleanup:
    //
```

```
    //
    // Clean up any individual pieces manually
    // Notice the fall-through structure of the switch.
    // Clean up in reverse order.
    //

    switch (rollback) {
        case 4:
        case 3:
            // Clean up the cleanup group members.
            CloseThreadpoolCleanupGroupMembers(cleanupgroup,
                FALSE, NULL);
        case 2:
            // Clean up the cleanup group.
            CloseThreadpoolCleanupGroup(cleanupgroup);

        case 1:
            // Clean up the pool.
            CloseThreadpool(pool);

        default:
            break;
    }

    return;
}

VOID
DemoNewRegisterWait()
{
    PTP_WAIT Wait = NULL;
    PTP_WAIT_CALLBACK waitcallback = MyWaitCallback;
    HANDLE hEvent = NULL;
    UINT i = 0;
    UINT rollback = 0;

    //
    // Create an auto-reset event.
    //
    hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    if (NULL == hEvent) {
        // Error Handling
        return;
    }

    rollback = 1; // CreateEvent succeeded

    Wait = CreateThreadpoolWait(waitcallback,
                                NULL,
                                NULL);

    if(NULL == Wait) {
        _tprintf(_T("CreateThreadpoolWait failed. LastError: %u\n"),
                    GetLastError());
        goto new_wait_cleanup;
    }

    rollback = 2; // CreateThreadpoolWait succeeded

    //
    // Need to re-register the event with the wait object
    // each time before signaling the event to trigger the wait callback.
    //
    for (i = 0; i < 5; i ++) {
        SetThreadpoolWait(Wait,
```

```
                            hEvent,
                            NULL);

        SetEvent(hEvent);

        //
        // Delay for the waiter thread to act if necessary.
        //
        Sleep(500);

        //
        // Block here until the callback function is done executing.
        //

        WaitForThreadpoolWaitCallbacks(Wait, FALSE);
    }

new_wait_cleanup:
    switch (rollback) {
        case 2:
            // Unregister the wait by setting the event to NULL.
            SetThreadpoolWait(Wait, NULL, NULL);

            // Close the wait.
            CloseThreadpoolWait(Wait);

        case 1:
            // Close the event.
            CloseHandle(hEvent);

        default:
            break;
    }
    return;
}

int main( void)
{
    DemoNewRegisterWait();
    DemoCleanupPersistentWorkTimer();
    return 0;
}
```

Related topics

Thread Pools

Community Additions ADD

Was this page helpful?

Your feedback about this content is important.
Let us know what you think.

Yes No

Find us on Facebook

Follow us on Twitter

Read the blog