Windows* Threading APIs Cheat Sheets

Visit MSDN* at http://msdn.microsoft.com/ for complete details on API. This summary is for your convenience only.

```
HANDLE hThread =
    CreateThread( LPSECURITY_ATTRIBUTES   lpThreadAttributes,
                  DWORD                   dwStackSize,
                  LPTHREAD_START_ROUTINE  lpStartAddress,
                  LPVOID                  lpParameter,
                  DWORD                   dwCreationFlags,
                  LPDWORD                 lpThreadId );
```

Notes

`lpThreadAttribute`

This is optional security for child processes. It can be `NULL`.

`dwStackSize`

This is stack size in bytes. It can be `0`, which means use default (usually 1 megabyte).

`lpStartAddress`

This is a globally visible function declared `DWORD WINAPI`. This is the function for the thread to begin execution.

`lpParameter`

This is a pointer to the one parameter for "`lpStartAddress`" function. Use a pointer to a structure to pass multiple parameters.

`dwCreationFlags`

This creates a thread and starts or suspends it. Use 0 to start; otherwise use CREATE_SUSPENDED.

`lpThreadId`

This is an output parameter and returns a unique (across the system) integer for the thread. It can be `NULL`.

Returns

`HANDLE` to a thread, or `NULL` if function fails. The `HANDLE` is for a kernel object that is a thread.

Also see:

```
VOID ExitThread( DWORD dwExitCode ); // exit (return from) calling thread
BOOL GetExitCodeThread( HANDLE hThread,
                        LPDWORD lpExitCode ); // call to get code
```

Example

```
#include <windows.h>    // required include file

DWORD WINAPI MyThreadStart(LPVOID p)
{   // Do some work in parallel here.
    // Signal thread exit:
    return(0);  // same as ExitThread(0);
}
main()
{   DWORD dwThreadRet;
    HANDLE hThread =
        CreateThread( NULL, 0,              // security, stackSize
                      MyThreadStart, NULL,  // threadFunc, threadParams
                      0, NULL );            // runFlag, threadIdOut

    // main continues after CreateThread() on its own thread.
}
```

This page intentionally left blank.

Visit MSDN* at http://msdn.microsoft.com/ for complete details on API. This summary is for your convenience only.

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

Notes

nCount

This is the number of handles in the `lpHandles` array.

lpHandles

This is a pointer to an array of handles.

bWaitAll

If this is `TRUE`, waits for all objects in `lpHandles` array to be *signaled*. If `FALSE`, waits for any one handle from the array to be signaled and the return value is the array index.

dwMilliseconds

This is the time-out interval in milliseconds. It can be `INFINITE` for no time-out.

Returns:

`WAIT_FAILED` if the function failed. See MSDN* for more details.

Also see:

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

Example

```
#include <windows.h>    // required include file

main()
{
    HANDLE hThreads[2] ;
    for (int i=0; i<2; i++)
    {
        hThread[i] = CreateThread(NULL,0, MyThreadStart,NULL, 0,NULL);
    }
    // Wait 1000 milliseconds (1 second) maximum for both threads
    // to complete or signal their exit:
    dwRet = WaitForMultipleObjects(2, hThreads, TRUE, 1000);


    HANDLE hMoreThreads[4] ;
    for (int j=0; j<4; j++)
    {
        hMoreThreads[j] = CreateThread(NULL,0, MyThreadStart,NULL, 0,NULL);
    }
    // Wait forever for all 4 threads to signal their exit:
    dwRet = WaitForMultipleObjects(4, hMoreThreads, TRUE, INFINITE);
}
```

This page intentionally left blank.

Visit MSDN* at http://msdn.microsoft.com/ for complete details on API. This summary is for your convenience only.

```
CRITICAL_SECTION csLock;
VOID InitializeCriticalSection ( LPCRITICAL_SECTION csLock );
VOID DeleteCriticalSection     ( LPCRITICAL_SECTION csLock );
VOID EnterCriticalSection      ( LPCRITICAL_SECTION csLock );
VOID LeaveCriticalSection      ( LPCRITICAL_SECTION csLock );
```

Notes

csLock

This is a lightweight, user-space variable to be used like a *mutex* (MUTual EXclusion) or lock.

InitializeCriticalSection()

This function initializes the CRITICAL_SECTION variable. This function must be called before the CRITICAL_SECTION variable can be used.

DeleteCriticalSection()

This function destroys all resources used by the CRITICAL_SECTION variable. This function is called when the CRITICAL_SECTION variable is no longer needed.

EnterCriticalSection()

This function attempts to acquire the CRITICAL_SECTION variable. If another thread has already acquired the lock, this function will block; once the CRITICAL_SECTION variable has been acquired, the function returns.

LeaveCriticalSection()

This function releases the lock, and returns immediately. The thread that releases the lock must be the same thread that obtained (acquired) the lock.

Example

```
#include <windows.h>

int MyShared = 0;          // global shared by all threads
CRITICAL_SECTION MyLock; // shared lock for exclusive access to shared data

DWORD WINAPI MyThreadStart(LPVOID p)
{
    int MyPrivate = DoBigComputation();  // local to each thread

    EnterCriticalSection(&MyLock);
        // The shared global variable (MyShared) is updated one thread at a
        // time from each thread’s own local, private variable (MyPrivate).
        MyShared += MyPrivate;
    LeaveCriticalSection(&MyLock);

    return(0);
}
int main()
{   InitializeCriticalSection(&MyLock);

    // Create N threads here all mapped to MyThreadStart() function.
    // Wait for all threads to signal completion . . .

    DeleteCriticalSection(&MyLock);
    return MyShared;
}
```

This page intentionally left blank.

Visit MSDN* at http://msdn.microsoft.com/ for complete details on API. This summary is for your convenience only.

```
HANDLE hSemaphore =
    CreateSemaphore( LPSECURITY_ATTRIBUTES lpsa,
                     LONG                  lSemInitial,
                     LONG                  lSemMax,
                     LPCSTR                lpSemName);

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

Notes

hSemaphore

This is a handle for the semaphore object.

CreateSemaphore()

This function initializes the semaphore object. This function must be called before the semaphore can be used.

lpsa

This is optional security for the semaphore. It can be `NULL.`

lSemInitial

This is the initial value of the semaphore upon creation. This value must be greater than or equal to zero and less than or equal to `lSemMax`.

lSemMax

This is the maximum value the semaphore. This value must be a positive integer.

lpSemName

This is a pointer to a `NULL` terminated string that specifies the name of the semaphore. Use `NULL` is no name is required. Named semaphores can be accessed by threads in other processes.

ReleaseSemaphore()

This function increments the semaphore object by `cReleaseCount` and returns the previous semaphore count prior to increment.

cReleaseCount

This is the amount to increment the semaphore upon release. This value must be greater than zero.

lpPreviousCount

This returns the value of the semaphore prior to increment. If the value is not needed, `NULL` can be used.

Example

```
#include <windows.h>
#define SLOTS_IN_LIST 10
long numListElements = 0;

HANDLE MySem; // shared semaphore for counting open list slots

DWORD WINAPI MyThreadStart(LPVOID p)
{
    int MyPrivate;
    while (!bDone) {
      MyPrivate = DoSomeComputation();  // local to each thread

      WaitForSingleObject(&MySem, INFINITE);  // space on list?

      // Add MyPrivate to list
      InterlockedIncrement(&numListElements);  // one more item on list

      if (numListElements == SLOTS_IN_LIST) {
          // Empty the list
          numListElements = 0;
          ReleaseSemaphore(MySem, 10, NULL);  // all list slots available
      }
    }
    return(0);
}

int main()
{   mySem = CreateSempahore(NULL, 0, SLOTS_IN_LIST, NULL);

    // Create list structure with SLOTS_IN_LIST elements available

    // Create N threads here all mapped to MyThreadStart() function.
    // Wait for all threads to signal completion . . .

}
```

Visit MSDN* at http://msdn.microsoft.com/ for complete details on API. This summary is for your convenience only.

```
HANDLE hEvent =
    CreateEvent(    LPSECURITY_ATTRIBUTES lpea,
                    BOOL                  bManualReset,
                    BOOL                  bInitialState,
                    LPCSTR                lpSemName);

BOOL SetEvent   ( HANDLE hEvent );

BOOL ResetEvent( HANDLE hEvent );

BOOL PulseEvent( HANDLE hEvent );
```

Notes

`hEvent`

This is a handle for the event object.

`CreateEvent()`

This function initializes the event object. This function must be called before the event can be used.

`lpea`

This is optional security for the event. It can be `NULL.`

`bManualReset`

This boolean sets the type of the event. The created event is a Manual Reset event if **bManualReset** is TRUE; it is an Auto-Reset event if the parameter is FALSE.

`bInitialState`

This Boolean determines that initial signal state of the event. The event is created in a Signaled state if **bInitialState** is TRUE; it is created in a nonsignaled state if the parameter is FALSE.

`SetEvent()`

For a Manual Reset event, this function will signal the event and the event will remain in the signaled state until ResetEvent is called. For an Auto-Reset event, this function will signal the event and leave the event signaled until one thread has waited and released on the event; the event will be reset.

`ResetEvent()`

For a Manual Reset event, this function will reset the event to the nonsignaled state. For an Auto-Reset event, this function will reset the event to the nonsignaled state (this is not typically done since the event will be reset automatically).

`PulseEvent()`

For a Manual Reset event, this function will signal the event and release all threads waiting on the event; the event will be reset to the nonsignaled state. For an Auto-Reset event, this function will signal the event and release only one thread, if such a thread is waiting; the event will be reset. If no threads are waiting on the event, the signal is “lost.”

Example

```
#include <windows.h>

HANDLE hEvents[2]; // 0 is found, 1 is not found

DWORD WINAPI threadFunc(LPVOID arg) {
      BOOL bFound = bigFind() ;

   if (bFound)
   {
      SetEvent(hEvent[0]); // signal data was found
      bigFound() ;
   }
   else
      SetEvent(hEvent[1]); // signal data was not found

   moreBigStuff() ;
   return 0;
}

int main()
{   . . .
   hEvent[0] = CreateEvent(NULL, FALSE, FALSE, NULL); // manual reset
   hEvent[1] = CreateEvent(NULL, FALSE, FALSE, NULL); // manual reset

/* Create thread and do some other work while thread executes search */

   DWORD waitRet = WaitForMultipleObjects(2, hEvent, FALSE, INFINITE);

   switch(waitRet) {
     case WAIT_OBJECT_0:      // found event signaled
           printf("found it!\n");
           ResetEvent(hEvent[0]);  // prepare for next search
           break;
     case WAIT_OBJECT_0+1:    // not found event signaled
           printf("not found\n");
           ResetEvent(hEvent[1]);  // prepare for next search
           break ;
     default:
           printf("wait error: ret %u\n", waitRet);
           break ;
   }
. . .
 }
```