Threads on Windows:

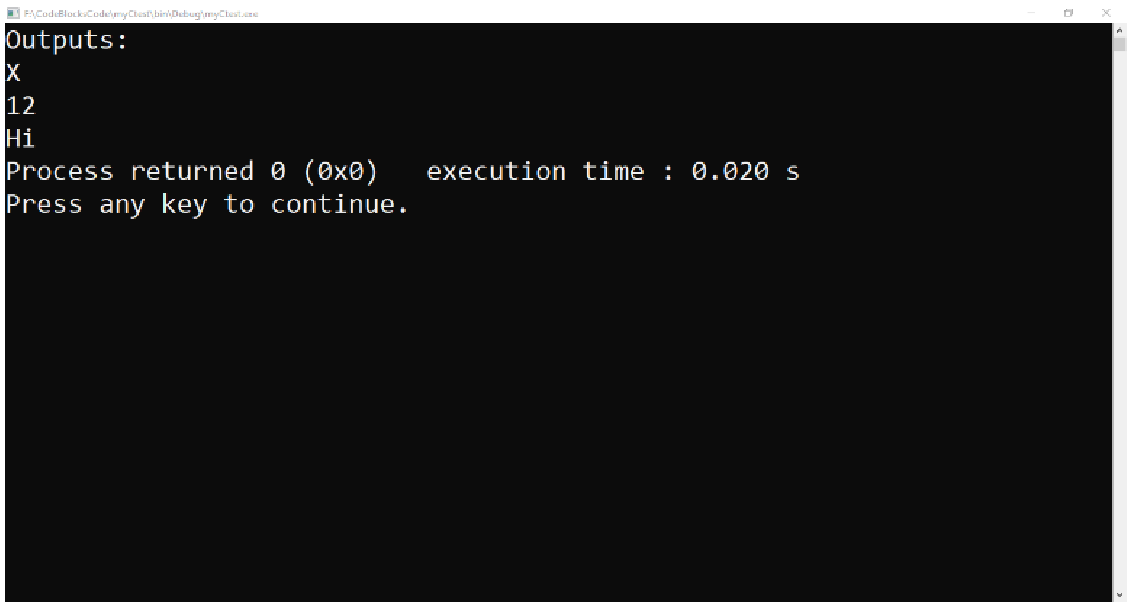
All the code given is part of the `<Windows.h>` library

Typical thread function:

```
//any alteration to the typical declaration for the thread function might change
//the function behavior
DWORD WINAPI ThreadFunction(LPVOID lpParameter)
{
    // The new thread will start here
    return 0;
}
```

A typical thread function in C on windows would take its input arguments in the form of a pointer to void. This pointer to void should be cast into the type of variable sent as the programmer desires.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
int main()
{
    char v1='X';
    int v2 =12;
    char v3[3] = "Hi";
    LPVOID  arg1 = &v1;
    LPVOID  arg2 = &v2;
    LPVOID  arg3 = &v3;
    LPVOID  *ptrs[3]= {arg1,arg2,arg3};
    printf("Outputs:");
    printf("\n%c", *(char*)ptrs[0]);
    printf("\n%d", *(int*)ptrs[1]);
    printf("\n%s", (char**)ptrs[2]);
    return 0;
}
```



```
Outputs:
X
12
Hi
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

Thread creating function:

```
HANDLE WINAPI CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,  // Thread attributes
SIZE_T dwStackSize,                        // Stack size
LPTHREAD_START_ROUTINE lpStartAddress,     // Thread start address
LPVOID lpParameter,                        // Parameter to pass to the thread
DWORD dwCreationFlags,                     // Creation flags
LPDWORD lpThreadId                         // Thread id
);
```

1- **lpThreadAttributes** [in, optional] (default: NULL)
A pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If **lpThreadAttributes** is NULL, the handle cannot be inherited.The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If **lpThreadAttributes** is NULL, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the primary token of the creator.
2- **dwStackSize** [in] (default: 0)
The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable.
3- **lpStartAddress** [in]
A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread.
4- **lpParameter** [in, optional] (default: NULL)
A pointer to a variable to be passed to the thread.
5- **dwCreationFlags** [in] (default: 0)
The flags that control the creation of the thread.
6- **lpThreadId** [out, optional] (default: NULL)
A pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

Wait for Thread function:

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

1- [in] **hHandle**:
   A handle to the object. For a list of the object types whose handles can be specified, see the following Remarks section. If this handle is closed while the wait is still pending, the function's behavior is undefined. The handle must have the SYNCHRONIZE access right. For more information, see Standard Access Rights.
2- [in] **dwMilliseconds**: (default: INFINITE)
   The time-out interval, in milliseconds. If a nonzero value is specified, the function waits until the object is signaled or the interval elapses. If **dwMilliseconds** is zero, the function does not enter a wait state if the object is not signaled; it always returns immediately. If **dwMilliseconds** is **INFINITE**, the function will return only when the object is signaled.

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

1- [in] **nCount:**
   The number of object handles in the array pointed to by **lpHandles**. The maximum number of object handles is **MAXIMUM_WAIT_OBJECTS**. This parameter cannot be zero.
2- [in] **lpHandles**:
   An array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles to objects of different types. It may not contain multiple copies of the same handle. If one of these handles is closed while the wait is still pending, the function's behavior is undefined. The handles must have the SYNCHRONIZE access right. For more information, see Standard Access Rights.
3- [in] **bWaitAll**:
   If this parameter is TRUE, the function returns when the state of all objects in the **lpHandles** array is signaled. If FALSE, the function returns when the state of any one of the objects is set to signaled. In the latter case, the return value indicates the object whose state caused the function to return.
4- [in] **dwMilliseconds**: (default: INFINITE)
   The time-out interval, in milliseconds. If a nonzero value is specified, the function waits until the object is signaled or the interval elapses. If **dwMilliseconds** is zero, the function does not enter a wait state if the object is not signaled; it always returns immediately. If **dwMilliseconds** is **INFINITE**, the function will return only when the object is signaled.

```
MAXIMUM_WAIT_OBJECTS is a defined value on windows = 64
```

Example of a Thread function:

```
#include <Windows.h>
DWORD WINAPI DoStuff(LPVOID lpParameter)
{
    // The new thread will start here
    printf("Thread function executing!\n");
    return 0;
}

int main()
{
    // Create a new thread which will start at the DoStuff function
    HANDLE hThread = CreateThread(
        NULL,    // Thread attributes
        0,       // Stack size (0 = use default)
        DoStuff, // Thread start address
        NULL,    // Parameter to pass to the thread
        0,       // Creation flags
        NULL);   // Thread id
    if (hThread == NULL)
    {
        // Thread creation failed.
        // More details can be retrieved by calling GetLastError()
        return 1;
    }

    // Wait for thread to finish execution
    WaitForSingleObject(hThread, INFINITE);

    // Thread handle must be closed when no longer needed
    CloseHandle(hThread);

    return 0;
}
```

Creating a mutex:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,  // pointer to security attributes
  BOOL bInitialOwner,                       // flag for initial ownership
  LPCTSTR lpName                            // pointer to mutex-object name
);
```

1-**lpMutexAttributes** (default: NULL)

Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If **lpMutexAttributes** is NULL, the handle cannot be inherited.

2-**bInitialOwner** (default: FALSE)

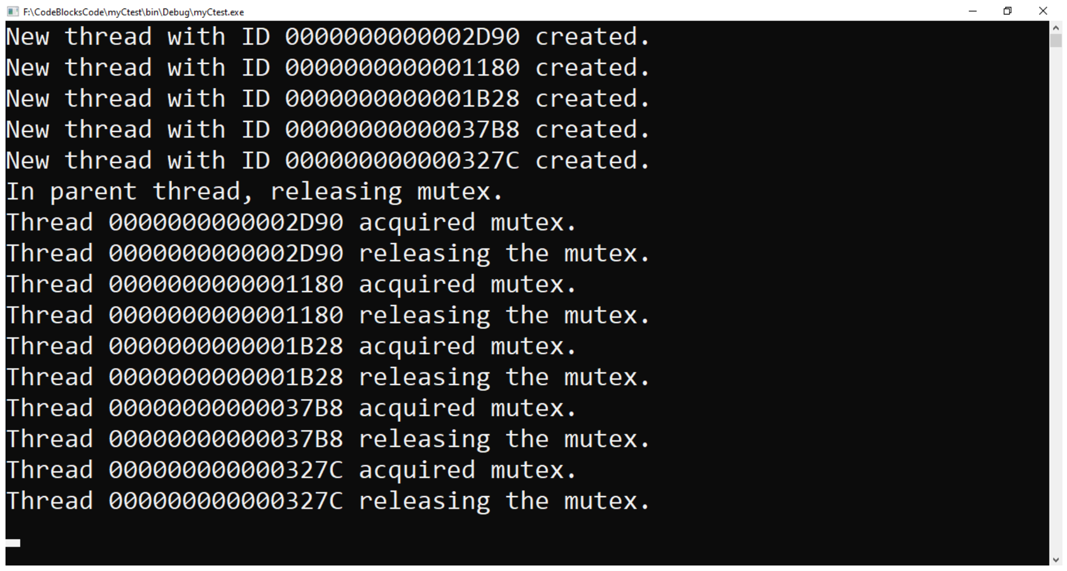
Specifies the initial owner of the mutex object. If this value is TRUE and the caller created the mutex, the calling thread obtains ownership of the mutex object. Otherwise, the calling thread does not obtain ownership of the mutex. To determine if the caller created the mutex, see the Return Values section.

3-**lpName** (default: NULL)

Pointer to a null-terminated string specifying the name of the mutex object. The name is limited to **MAX_PATH** characters and can contain any character except the backslash path-separator character (\). Name comparison is case sensitive. If **lpName** matches the name of an existing named mutex object, this function requests **MUTEX_ALL_ACCESS** access to the existing object. In this case, the **bInitialOwner** parameter is ignored because it has already been set by the creating process. If the **lpMutexAttributes** parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored. If **lpName** is NULL, the mutex object is created without a name. If **lpName** matches the name of an existing event, semaphore, waitable timer, job, or file-mapping object, the function fails and the **GetLastError** function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same name space.

```
#include <Windows.h>
const int Num_Threads = 5;
//create global variable mutex
HANDLE hMutex ;
//thread function
DWORD WINAPI ThreadFunc(LPVOID params)
{
//save the thread ID
    DWORD dwThreadID = GetCurrentThreadId();
    //Wait for Mutex to be released
    DWORD dwResult = WaitForSingleObject(hMutex, INFINITE);
//Thread code is stuck in previous line until the mutex is released
    printf("Thread %p acquired mutex.\n", dwThreadID);
//hold the mutex for 1 second
    Sleep(1000);
    printf("Thread %p releasing the mutex.\n", dwThreadID);
    ReleaseMutex(hMutex);
    return 0;
}

int main()
{
    int i = 0;
    DWORD dwNewThreadID;
    HANDLE hChildThreads[Num_Threads];
    //initialize the mutex
    hMutex = CreateMutex(NULL, TRUE, NULL);
//create a group of threads for testing mutex
    while (i < Num_Threads)
    {
        hChildThreads[i] = CreateThread(NULL, NULL, &ThreadFunc, NULL, 0,
&dwNewThreadID);
        printf("New thread with ID %p created.\n", dwNewThreadID);
        i++;
    }
    printf("In parent thread, releasing mutex.\n");
    ReleaseMutex(hMutex);
    Sleep(6000);
    return 0;
}
```



```
F:\CodeBlocksCode\myCtest\bin\Debug\myCtest.exe
New thread with ID 0000000000002D90 created.
New thread with ID 0000000000001180 created.
New thread with ID 0000000000001B28 created.
New thread with ID 00000000000037B8 created.
New thread with ID 000000000000327C created.
In parent thread, releasing mutex.
Thread 0000000000002D90 acquired mutex.
Thread 0000000000002D90 releasing the mutex.
Thread 0000000000001180 acquired mutex.
Thread 0000000000001180 releasing the mutex.
Thread 0000000000001B28 acquired mutex.
Thread 0000000000001B28 releasing the mutex.
Thread 00000000000037B8 acquired mutex.
Thread 00000000000037B8 releasing the mutex.
Thread 000000000000327C acquired mutex.
Thread 000000000000327C releasing the mutex.
```

Creating a semaphore:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG                  lInitialCount,
  LONG                  lMaximumCount,
  LPCSTR                lpName
);
```

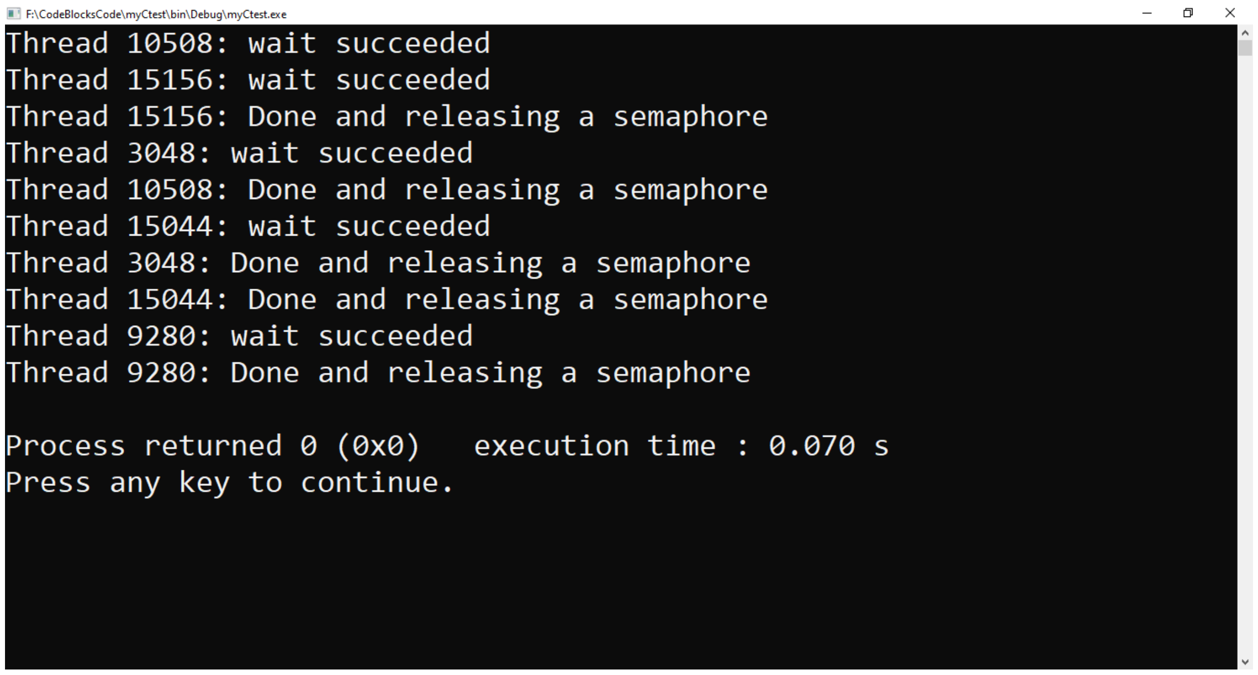
1- [in, optional] **lpSemaphoreAttributes**: (default: NULL)
   A pointer to a SECURITY_ATTRIBUTES structure. If this parameter is NULL, the handle cannot be inherited by child processes. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new semaphore. If this parameter is NULL, the semaphore gets a default security descriptor. The ACLs in the default security descriptor for a semaphore come from the primary or impersonation token of the creator.
2- [in] **lInitialCount**:
   The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to **lMaximumCount**. The state of a semaphore is signaled when its count is greater than zero and non-signaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the **ReleaseSemaphore** function.
3- [in] **lMaximumCount**:
   The maximum count for the semaphore object. This value must be greater than zero.
4- [in, optional] **lpName**: (default: NULL)
   The name of the semaphore object. The name is limited to MAX_PATH characters. Name comparison is case sensitive. If **lpName** matches the name of an existing named semaphore object, this function requests the SEMAPHORE_ALL_ACCESS access right. In this case, the **lInitialCount** and **lMaximumCount** parameters are ignored because they have already been set by the creating process. If the **lpSemaphoreAttributes** parameter is not NULL, it determines whether the handle can be inherited, but its security-descriptor member is ignored. If **lpName** is NULL, the semaphore object is created without a name.

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

1- [in] **hSemaphore**:
   A handle to the semaphore object. The **CreateSemaphore** or **OpenSemaphore** function returns this handle. This handle must have the SEMAPHORE_MODIFY_STATE access right. For more information, see Synchronization Object Security and Access Rights.
2- [in] **lReleaseCount**:
   The amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount would cause the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns FALSE.
3- [out, optional] **lpPreviousCount**:
   A pointer to a variable to receive the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

```
#include <windows.h>
#include <stdio.h>
#define MAX_SEM_COUNT 2
#define THREADCOUNT 5
//create global variable semaphore
HANDLE ghSemaphore;
DWORD WINAPI ThreadProc( LPVOID );
int main( void )
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
    int i;
    // initialize the semaphore with initial and max counts of 2
    //this means that the maximum allowed number of semaphores is 2
    //while the initially available is 2
    //usually initial value and max count are initialized by the same value
    ghSemaphore = CreateSemaphore(
                      NULL,           // default security attributes
                      MAX_SEM_COUNT,  // initial count
                      MAX_SEM_COUNT,  // maximum count
                      NULL);          // unnamed semaphore
    // Create worker threads
    for( i=0; i < THREADCOUNT; i++ )
    {
        aThread[i] = CreateThread(
                         NULL,       // default security attributes
                         0,          // default stack size
                         &ThreadProc,
                         NULL,       // no thread function arguments
                         0,          // default creation flags
                         &ThreadID); // receive thread identifier
    }
    // Wait for all threads to terminate
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
    // Close thread and semaphore handles
    for( i=0; i < THREADCOUNT; i++ )
        CloseHandle(aThread[i]);
    CloseHandle(ghSemaphore);
    return 0;
}
DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    // Try to enter the semaphore gate.
    DWORD dwWaitResult = WaitForSingleObject(
                             ghSemaphore,   // handle to semaphore
                             INFINITE);           // zero-second time-out interval
    printf("Thread %d: wait succeeded\n", GetCurrentThreadId());
    // Simulate thread spending time on task
    Sleep(5);
    // Release the semaphore when task is finished
    printf("Thread %d: Done and releasing a semaphore\n",
GetCurrentThreadId());
    ReleaseSemaphore(
        ghSemaphore,  // handle to semaphore
        1,            // increase count by one
        NULL) ;       // not interested in previous count

    return 0;}
```



```
Thread 10508: wait succeeded
Thread 15156: wait succeeded
Thread 15156: Done and releasing a semaphore
Thread 3048: wait succeeded
Thread 10508: Done and releasing a semaphore
Thread 15044: wait succeeded
Thread 3048: Done and releasing a semaphore
Thread 15044: Done and releasing a semaphore
Thread 9280: wait succeeded
Thread 9280: Done and releasing a semaphore

Process returned 0 (0x0)   execution time : 0.070 s
Press any key to continue.
```