Critical Section

Windows C++
Synchronization Techniques

Critical Section

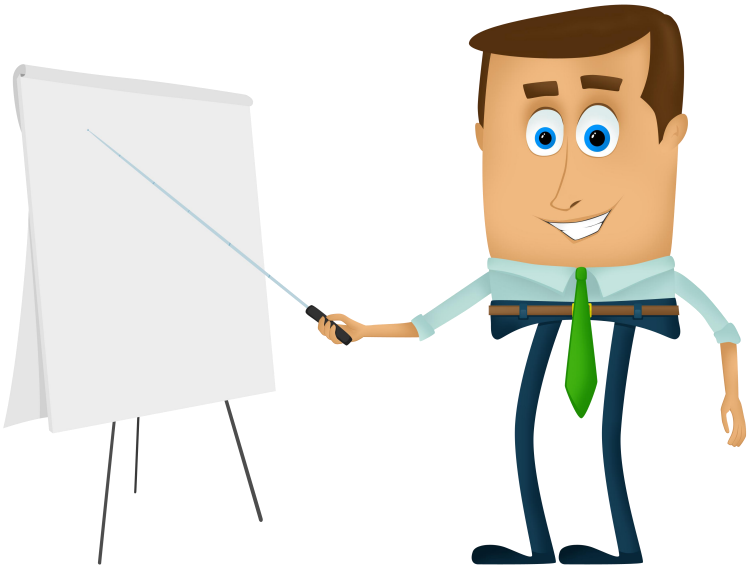
- Light weighted
- Inter-thread synchronization
- Not shared across processes.
- Not Kernel Object

Win32 : CRITICAL_SECTION

MFC : CCriticalSection

Introduction to Windows C++ Synchronization Techniques

Watch Introduction to Windows C++ Synchronization Techniques

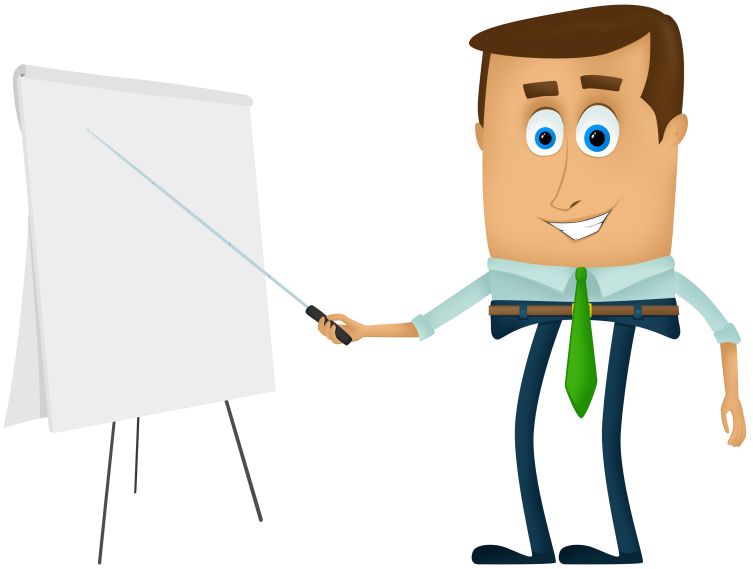


Usage of Win32 CRITICAL_SECTION

Step 1

Include the header file <windows.h>

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

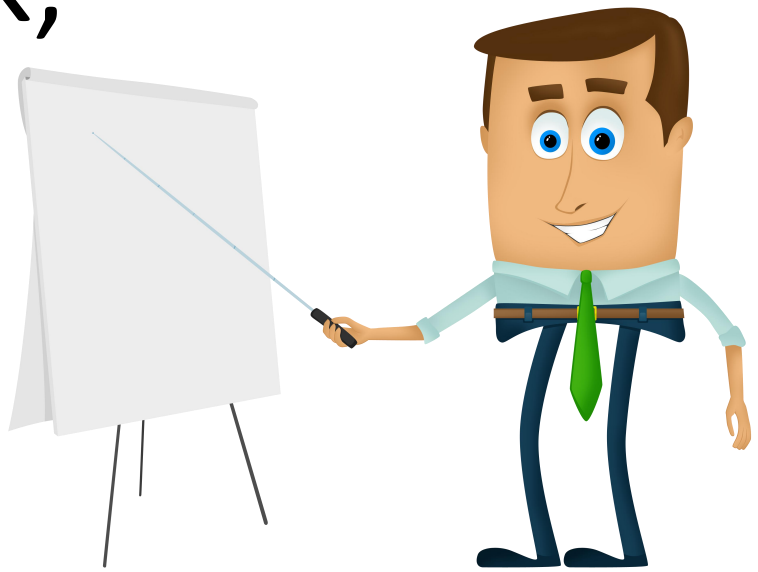


Usage of Win32 CRITICAL_SECTION

Step 2

Declare the CRITICAL_SECTION variable.

```
CRITICAL_SECTION csLock;
```



Usage of Win32 CRITICAL_SECTION

Step 3

Initialize the CRITICAL_SECTION variable.

```
::InitializeCriticalSection( &csLock );
```

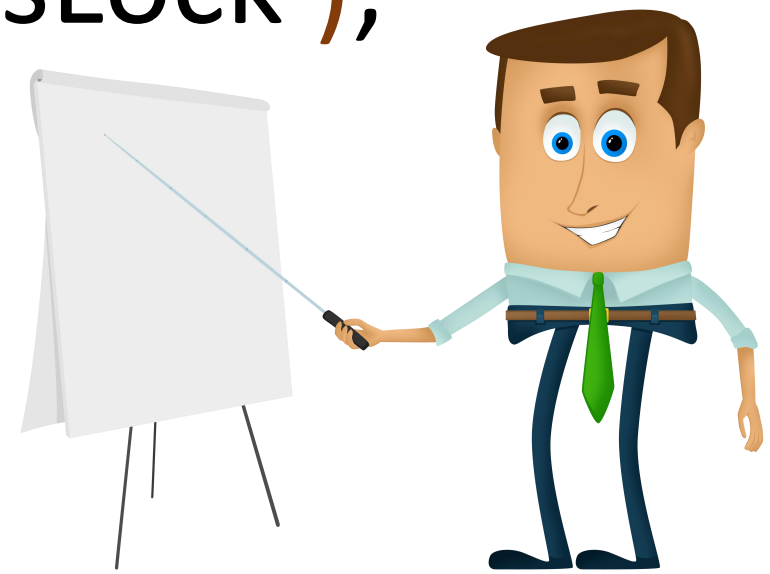
Do NOT re-Initialize an already initialized CRITICAL_SECTION object when it is under use by any thread. Doing so result in Undefined Behaviour

Usage of Win32 CRITICAL_SECTION

Step 4

Accquire the CRITICAL_SECTION lock.

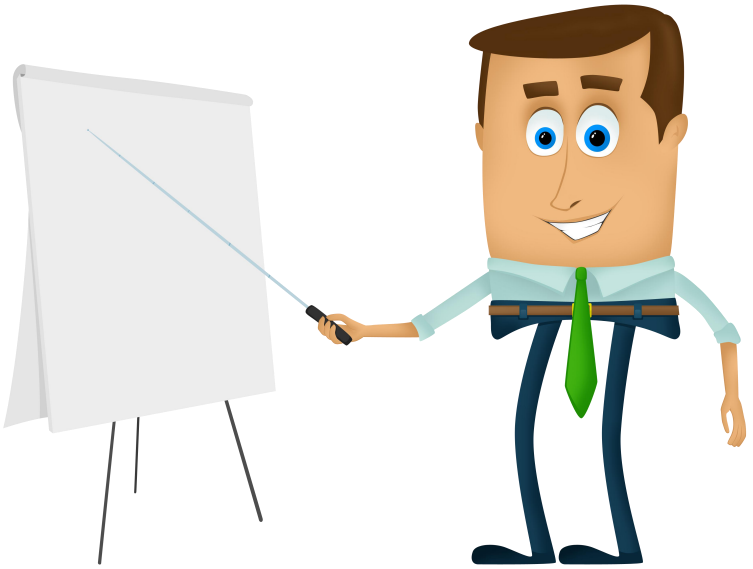
```
::EnterCriticalSection( &csLock );
```



Usage of Win32 CRITICAL_SECTION

Step 5

Execute the synchronized code section.



Usage of Win32 CRITICAL_SECTION

Step 6

Release the CRITICAL_SECTION lock.

```
::LeaveCriticalSection( &csLock );
```

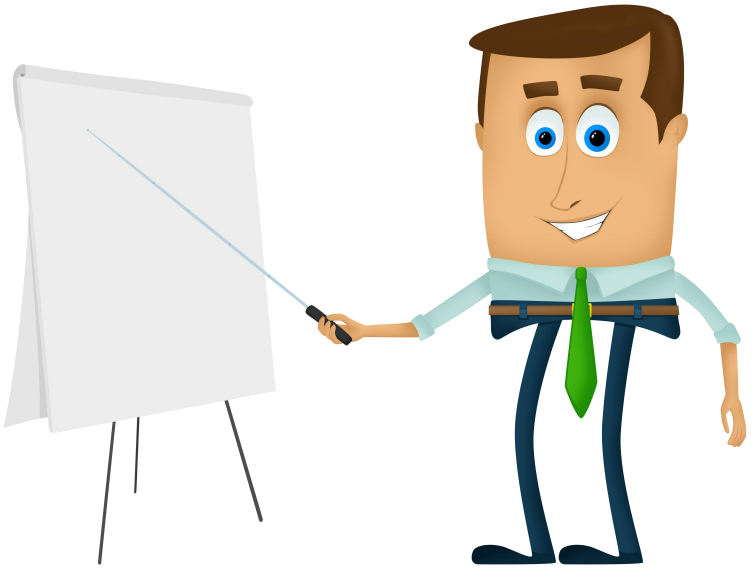
msdn:

If a thread calls ::LeaveCriticalSection when it does not have ownership of the specified critical section object, an error occurs that may cause another thread using EnterCriticalSection to wait indefinitely.

Usage of Win32 CRITICAL_SECTION

Step 7

Execute Step 4 to Step 6 until the synchronized execution is completed by each thread.



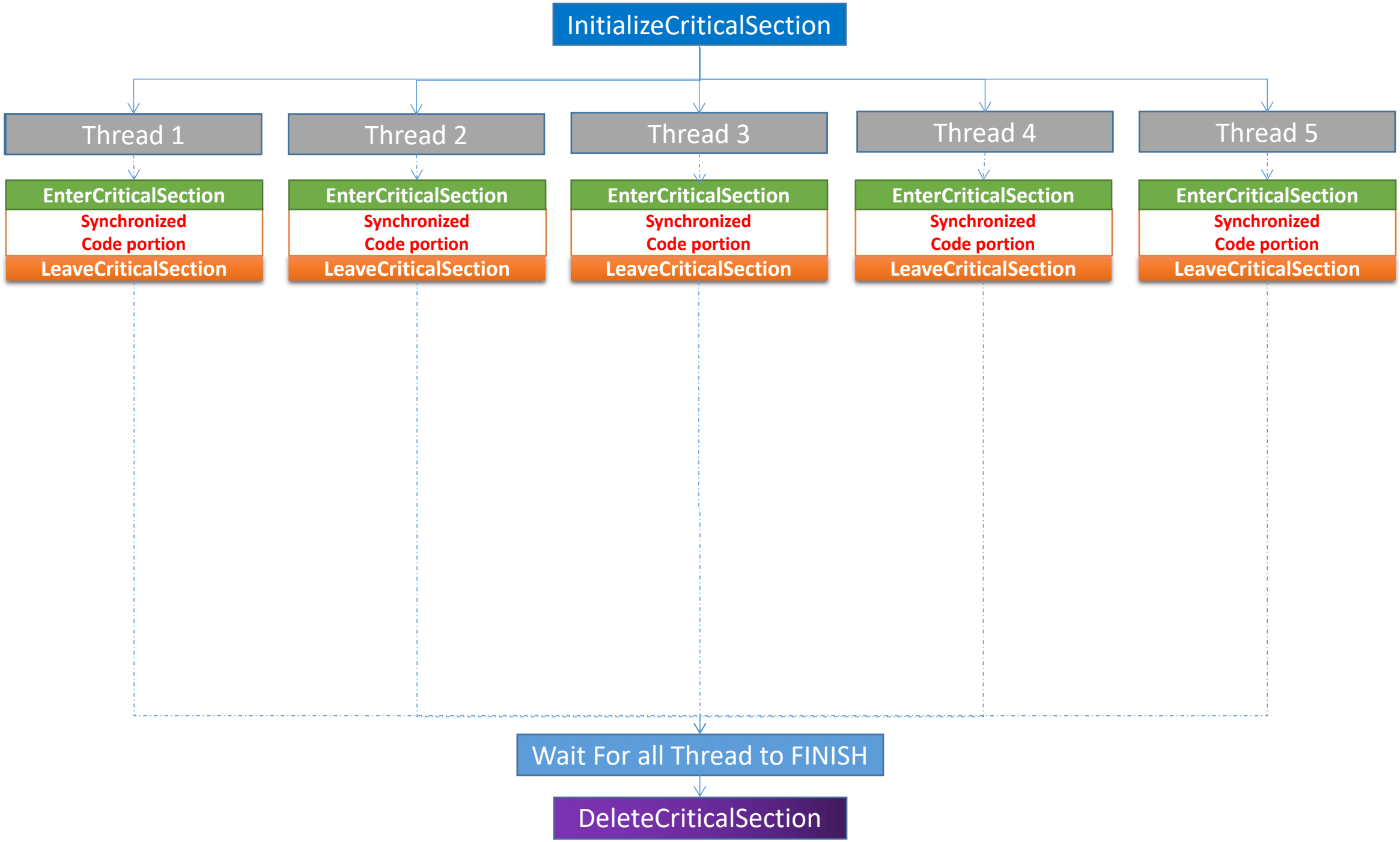
Usage of Win32 CRITICAL_SECTION

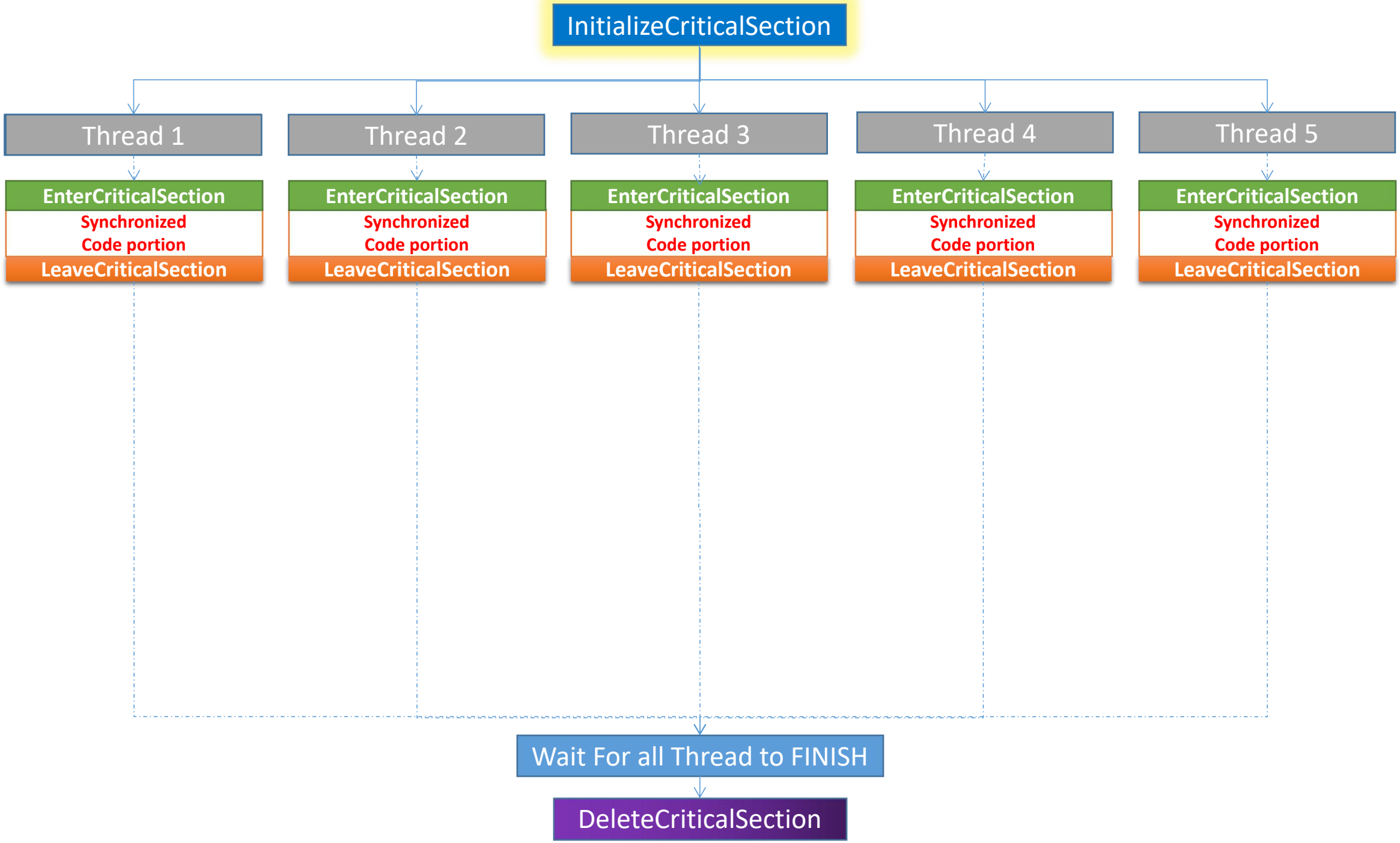
Step 8

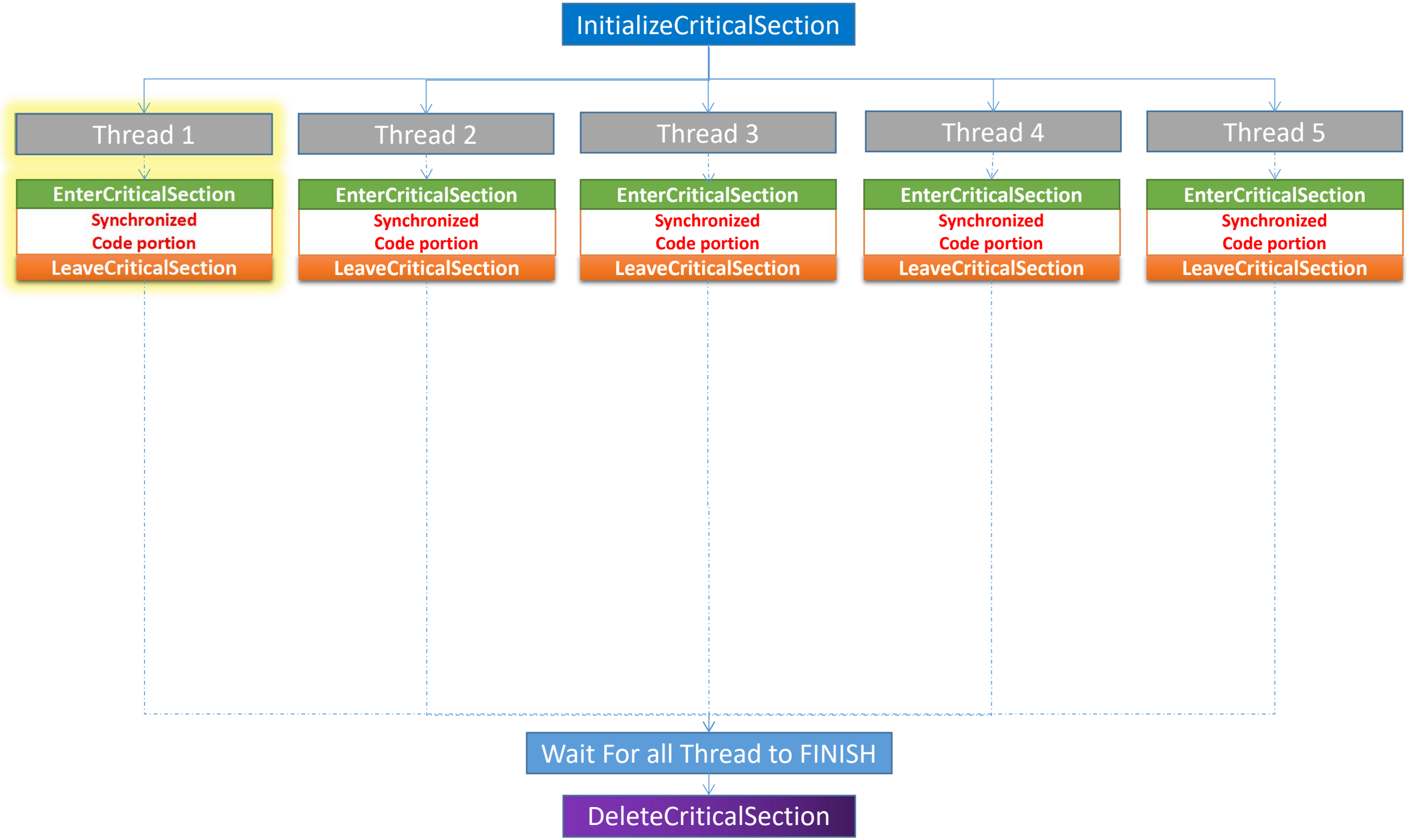
Delete the CRITICAL_SECTION object.

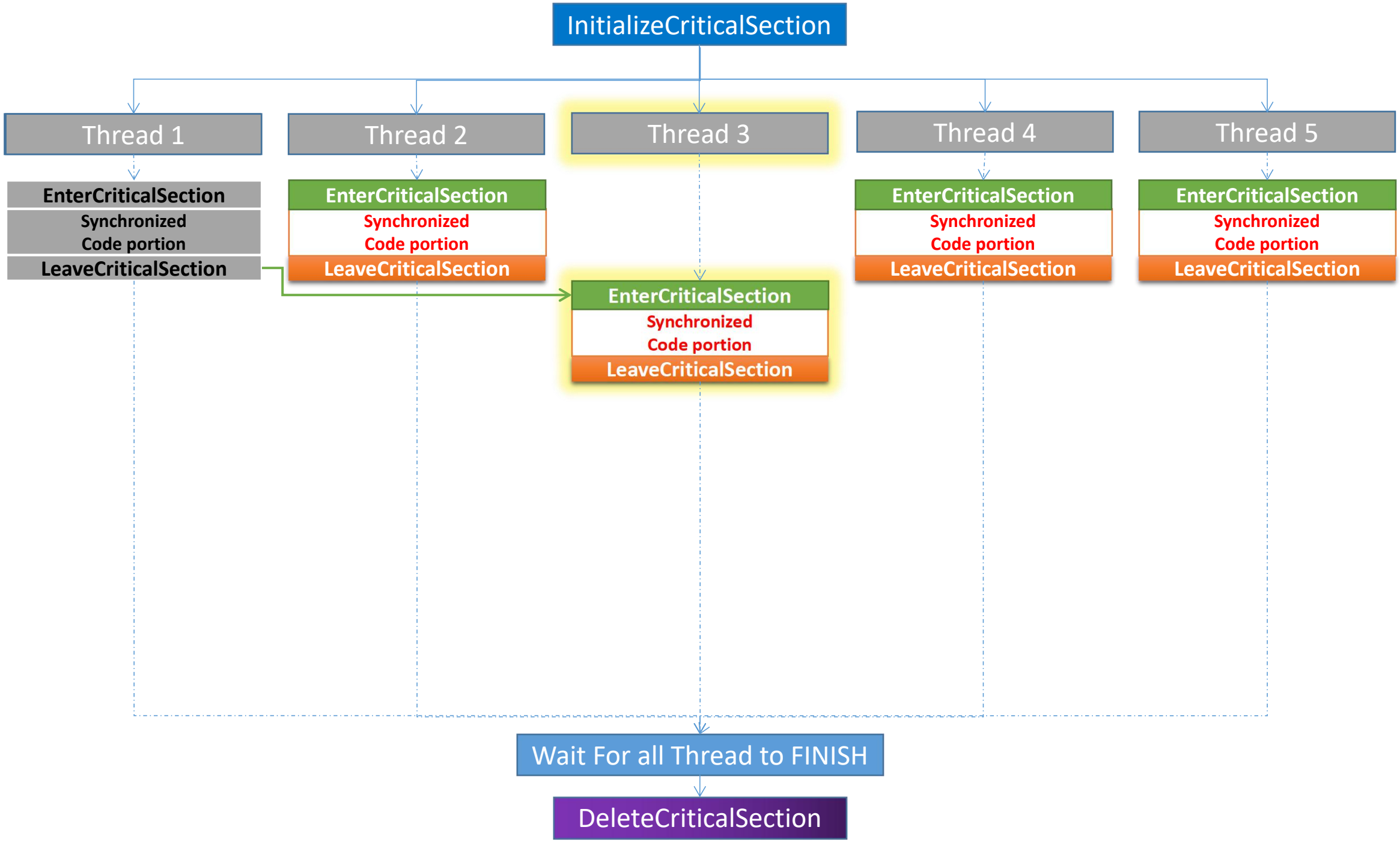
```
::DeleteCriticalSection( &csLock );
```

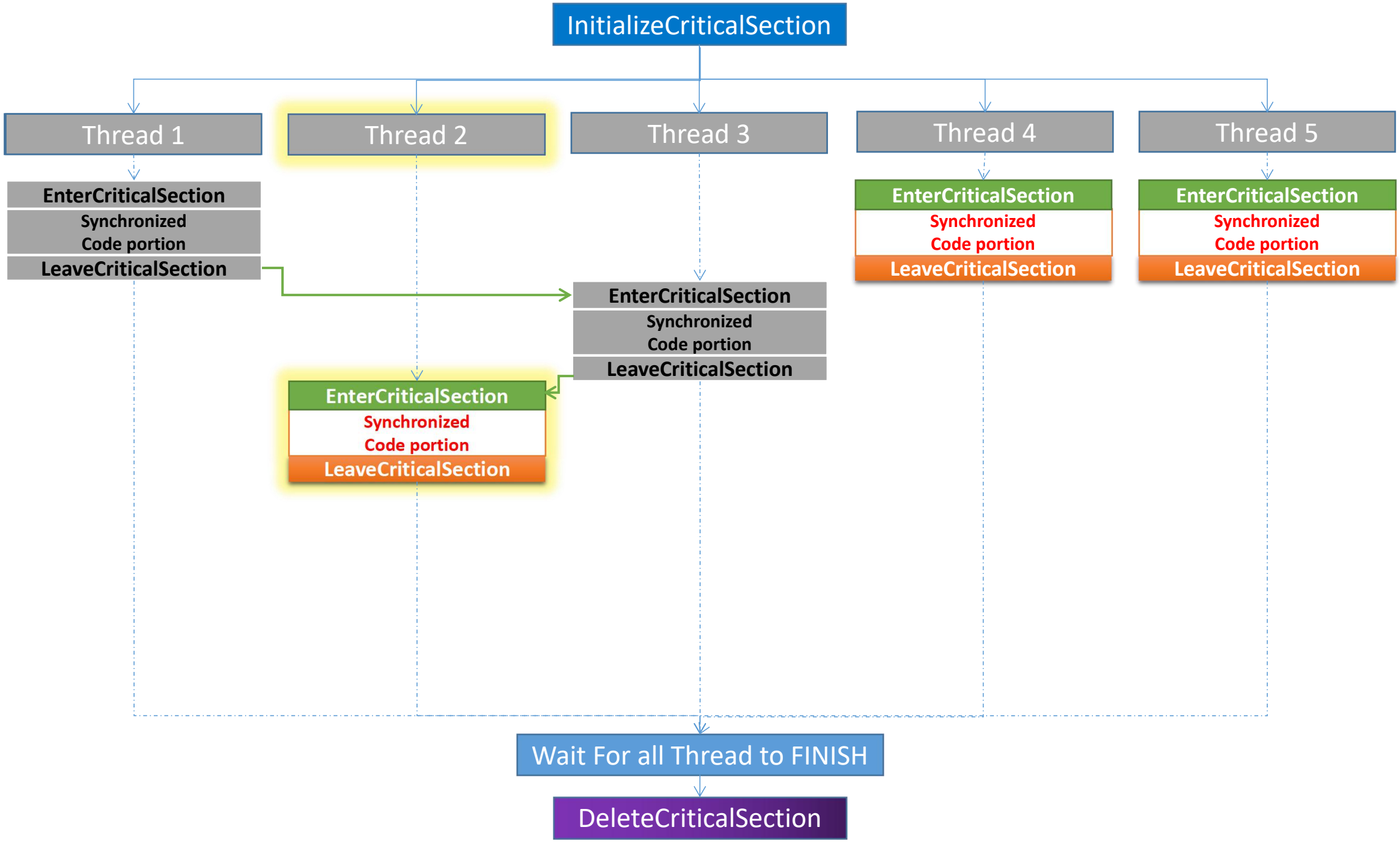
Do NOT delete an already deleted/non initialized CRITICAL_SECTION object .
Doing so result in Undefined Behaviour
After deletion, that critical section object can no longer be used for synchronization.

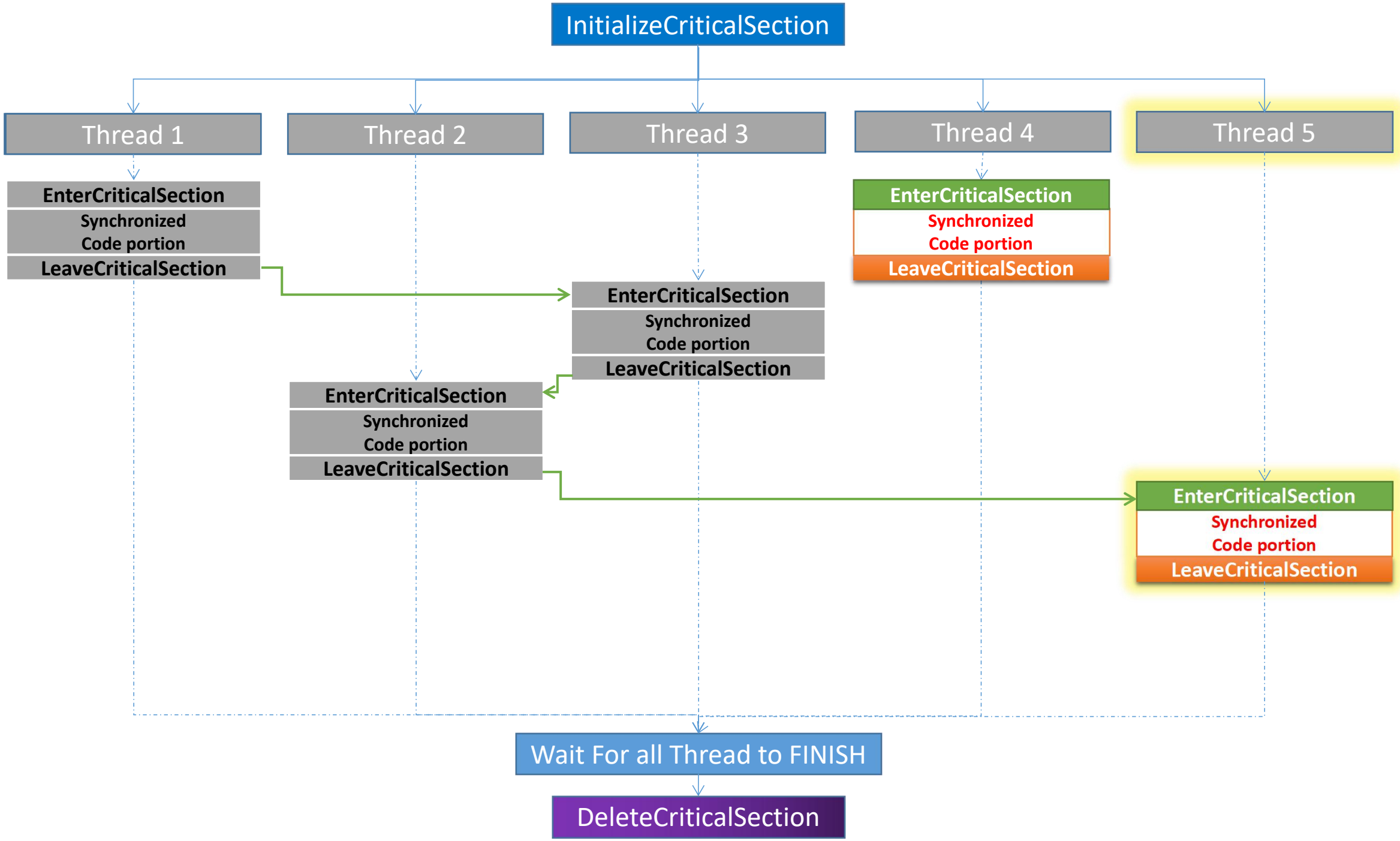

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

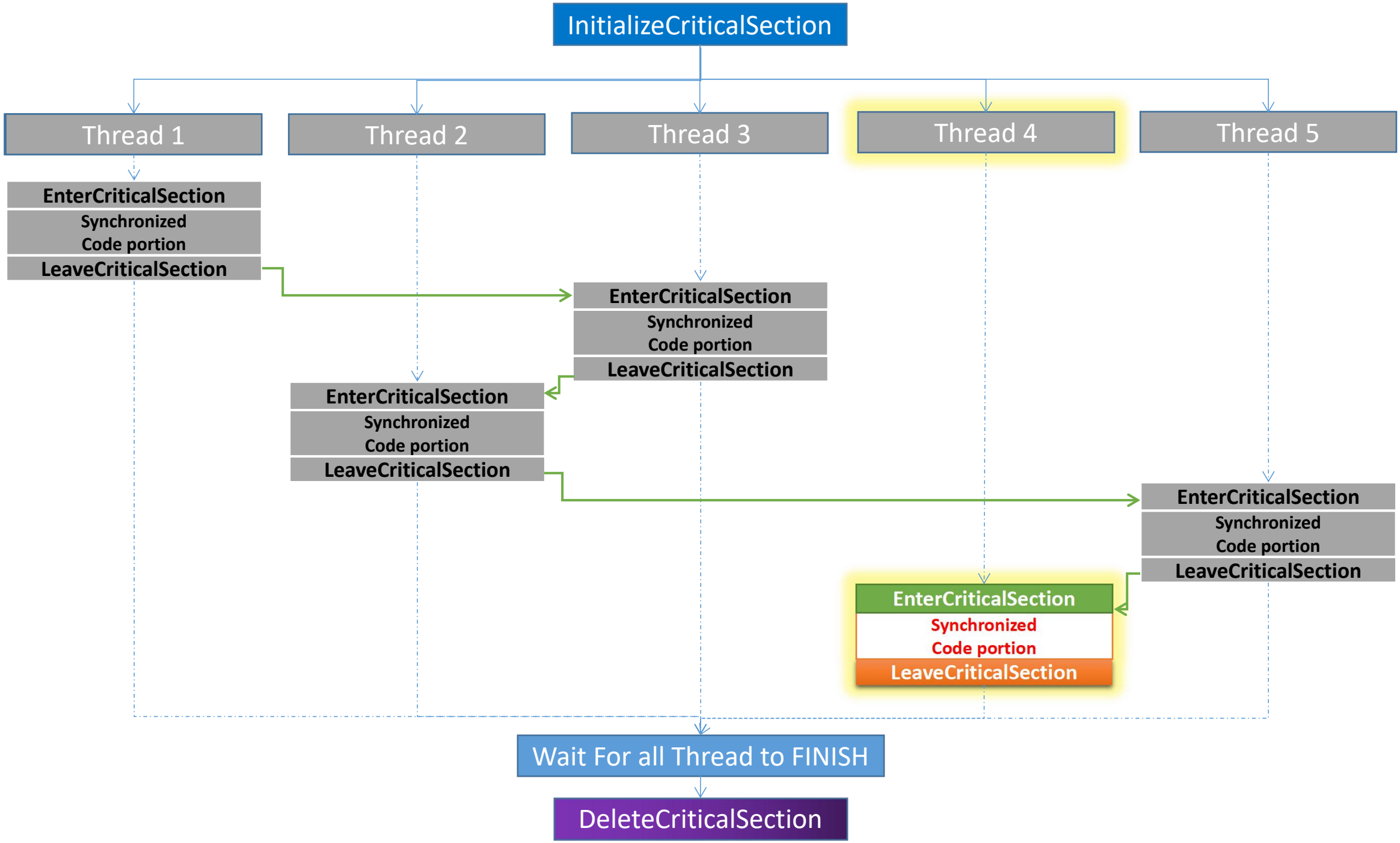

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

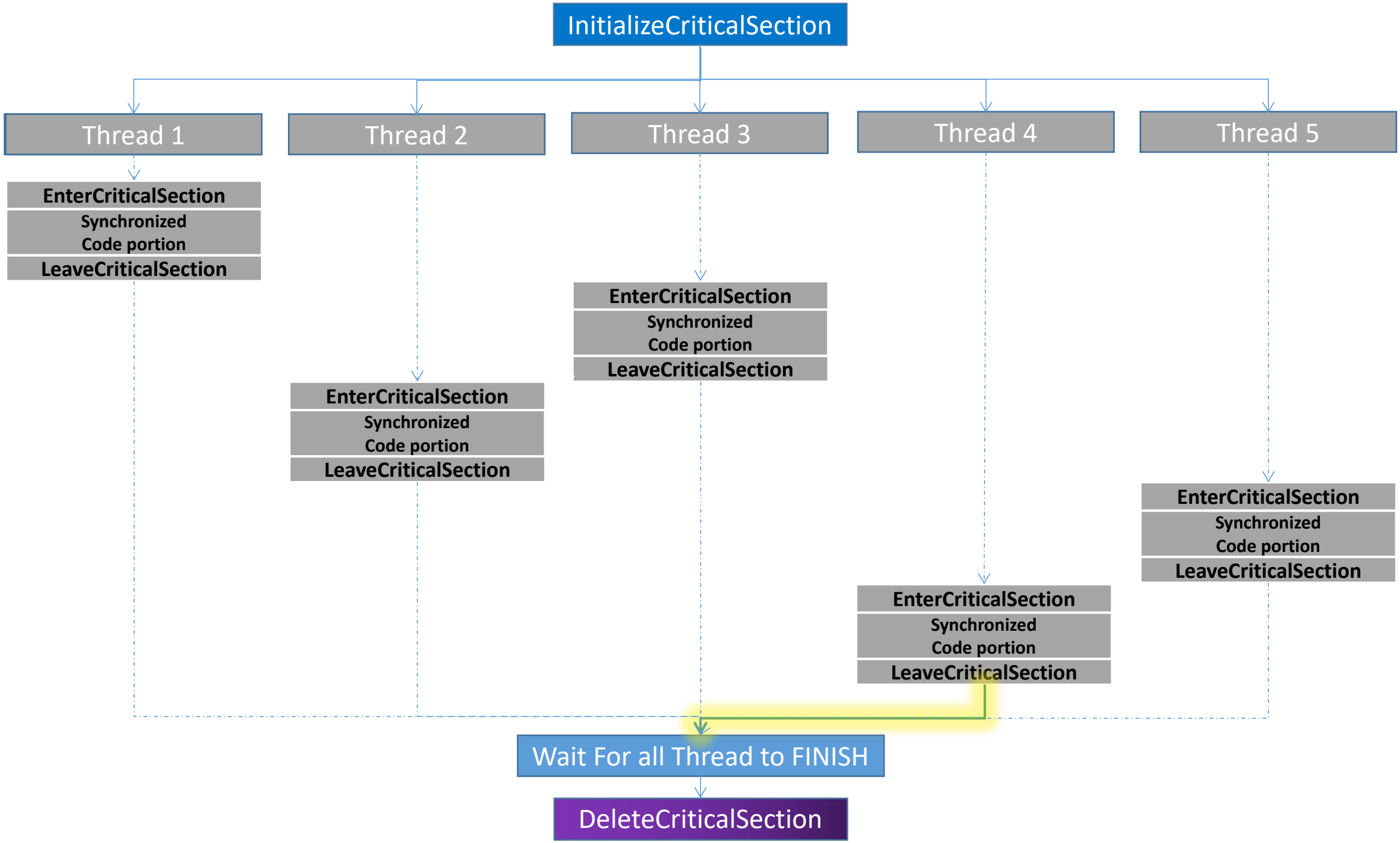

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

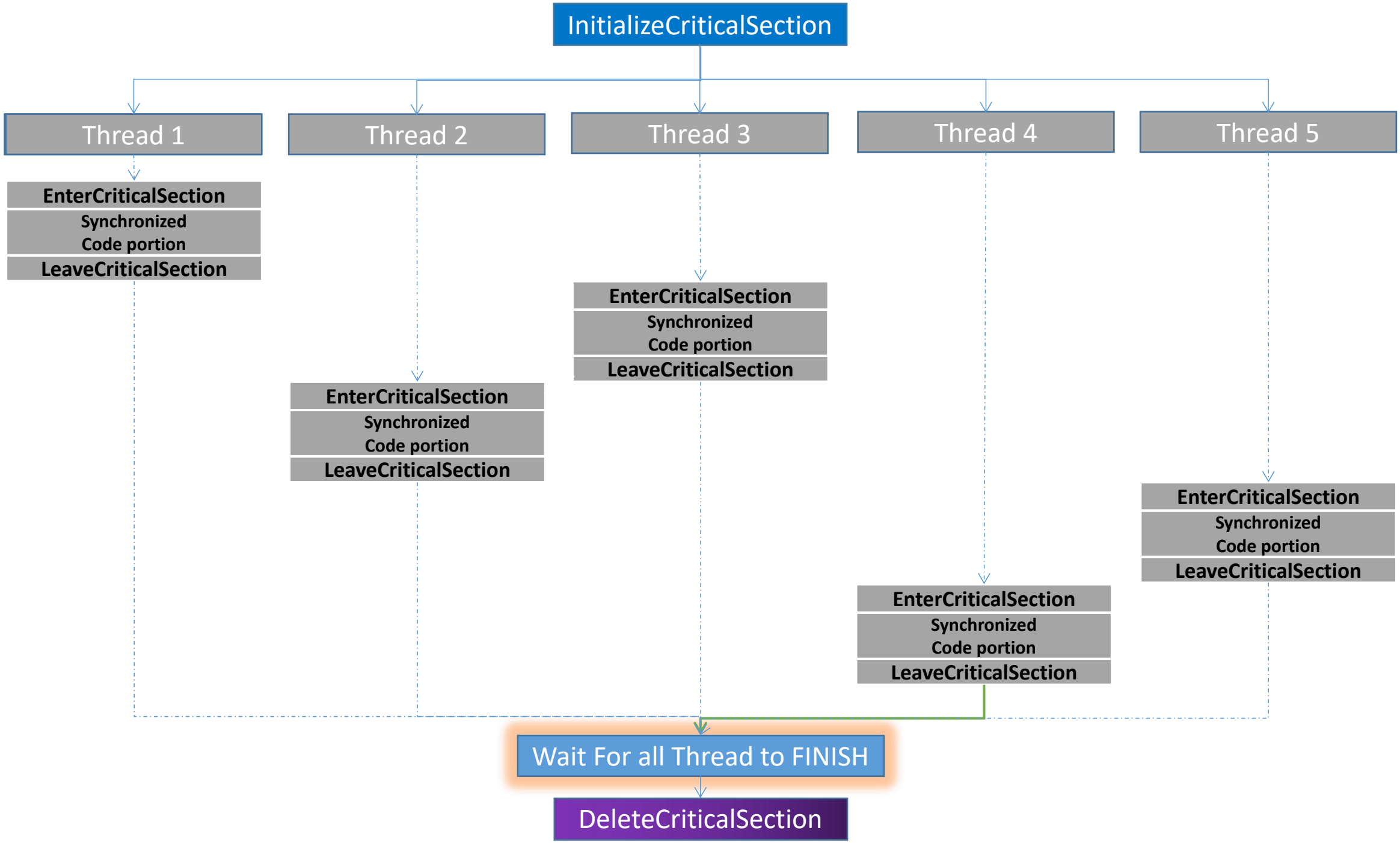

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

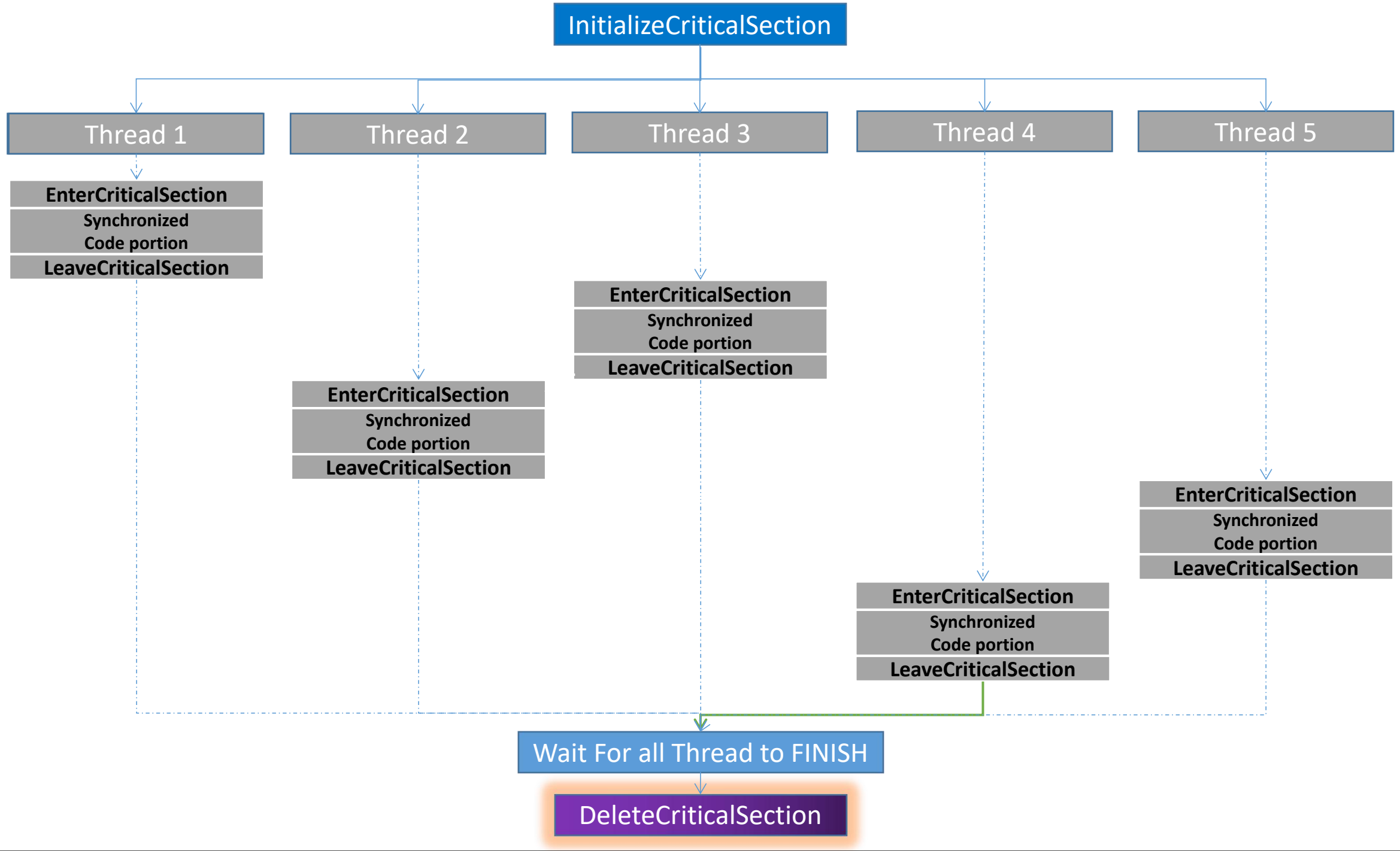

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection


InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection


InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection


InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

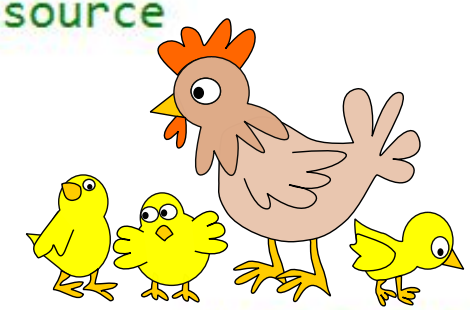

InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection


InitializeCriticalSection
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
EnterCriticalSection
Synchronized
Code portion
LeaveCriticalSection
Wait For all Thread to FINISH
DeleteCriticalSection

```cpp
#include <iostream>
#include <Windows.h>                                    // Step 1 : Include the header file <windows.h>

CRITICAL_SECTION g_csLock;                              // Step 2 : Declare the CRITICAL_SECTION variable

void PerformOperationInThreads( const int nThreadCount_i );
void CriticalFunctionality( const int& nThreadIndex_i );
DWORD WINAPI ThreadFunction( PVOID pVoid );

int main(){
    ::InitializeCriticalSection( &g_csLock );           // Step 3 : Initialize CRITICAL_SECTION object.

    PerformOperationInThreads( 6 );  // Create and start different thread that calls ThreadFunction().

    ::DeleteCriticalSection( &g_csLock );               // Step 8: Delete the CRITICAL_SECTION object.
    return 0;
}

DWORD WINAPI ThreadFunction( PVOID pVoid ){
    ::EnterCriticalSection( &g_csLock );                // step 4 : Accquire the CRITICAL_SECTION lock.

    CriticalFunctionality( *reinterpret_cast<int*>( pVoid )); // step 5 : Execute the synchronized code section

    ::LeaveCriticalSection( &g_csLock );                // step 6 : Release the CRITICAL_SECTION lock.
    return 0;
}                                                       // Step 7: Execute step 4 to 6 for all threads.
```

Win32 CRITICAL_SECTION

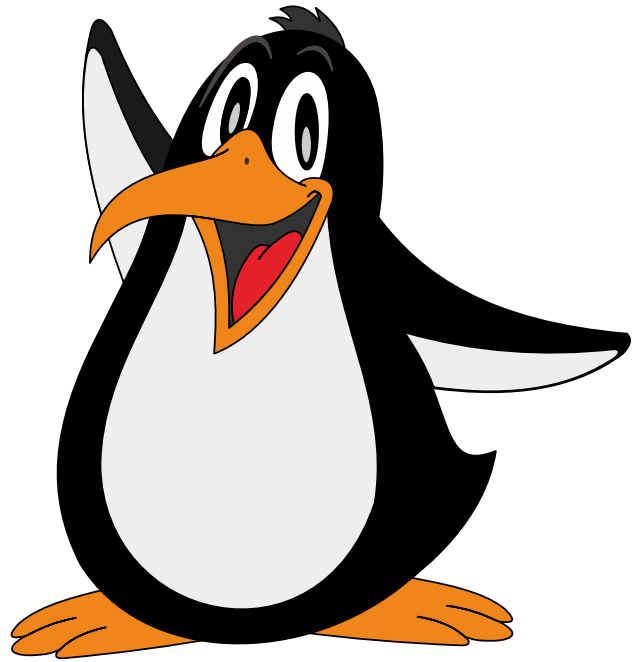
```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    //  The following three fields control entering and exiting the critical
    //  section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```



Displaying a Critical Section details

From Windows Server 2003 SP1 and Later, an initialized Critical Section holds following information

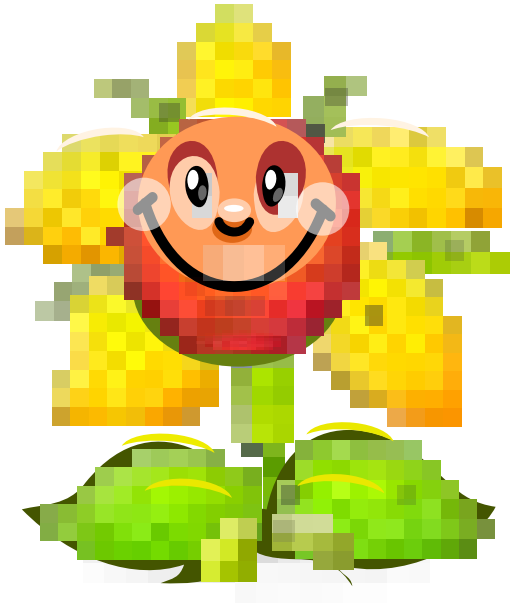
1. Lock Count
2. Recursion count
3. Owning Thread ID



Critical Section details

1. Lock Count

- ✓ Lock Status
- ✓ Whether any thread waiting for lock is woken from sleep or not
- ✓ Remaining number of threads waiting for lock



Critical Section details

1. Lock Count

E.g Lock Count = -22

Binary form : -10110

- **Critical section is locked** (0->Locked & 1-> Not locked)
- 1->None of thread has been woken for this lock
- **5 Threads wating for the lock**

E.g Lock Count = -118

Binary form : -01110110

- Critical section is locked
- No thread has been woken for this lock
- **29 Threads wating for the lock**

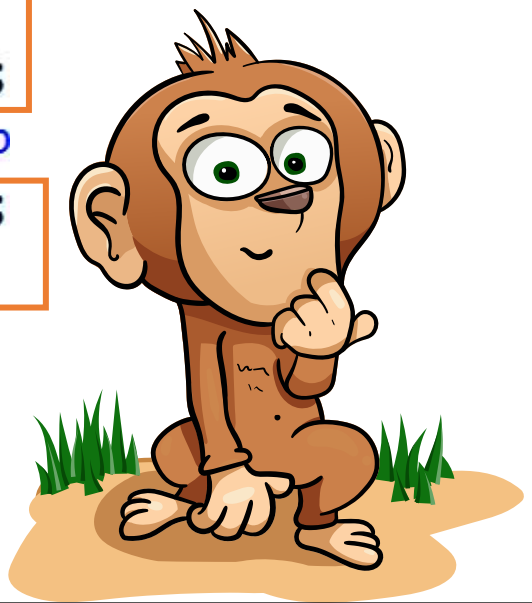
Critical Section details

2. RecursiveCount

E.g RecursiveCount = 1 → EnterCriticalSection is called only once in the current thread
So It should call LeaveCriticalSection 1 time only.

RecursiveCount = 2 → EnterCriticalSection is recursively called TWICE in the current thread.
So It should call LeaveCriticalSection 2 times.

```
DWORD WINAPI ThreadFunction( PVOID pVoid ){
    ::EnterCriticalSection( &g_csLock );
        ::EnterCriticalSection( &g_csLock );
            CriticalFunctionality( *reinterp
        ::LeaveCriticalSection( &g_csLock );
    ::LeaveCriticalSection( &g_csLock );
    return 0;
}
```

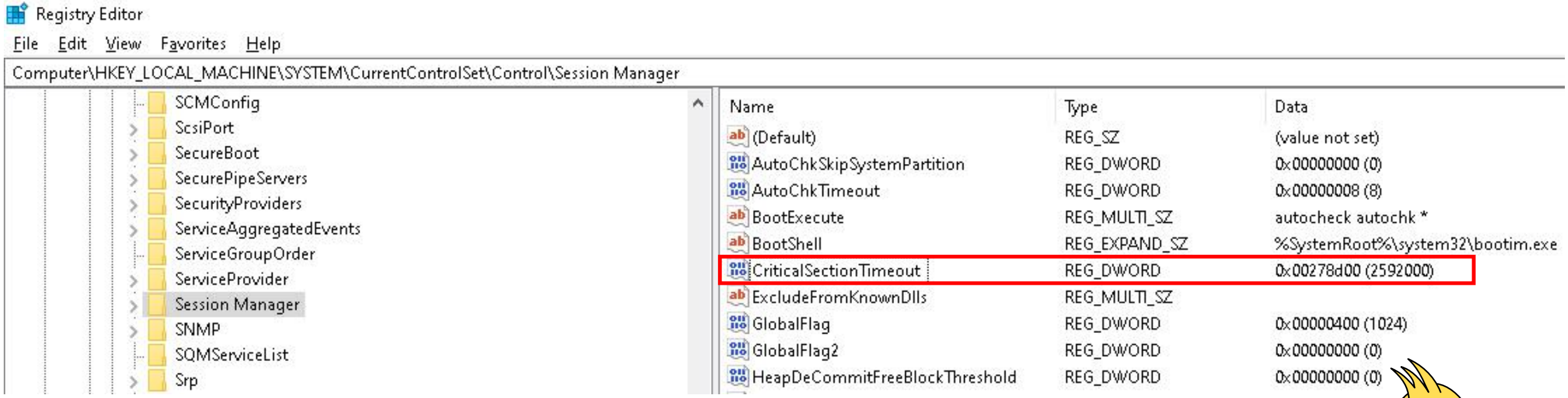


Critical Section details

3. OwningThread

E.g OwningThread= 0x000000000000476c → Thread ID of current Lock Owning thread

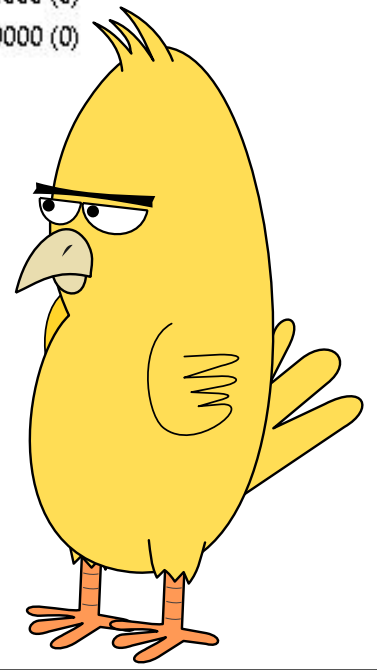
Facts



CriticalSectionTimeout = 2592000 Seconds

= 4 Years

If a dead lock occurs, Just wait for 4 years



Important Points about CRITICAL_SECTION

- Do not call 'InitializeCriticalSection' if the critical section is under use.
- Always delete the Critical Section using 'DeleteCriticalSection'. An Initialized critical section should be deleted only once
- Ensure that for each EnterCriticalSection, there exist a call to LeaveCriticalSection(Including Exception scenarios also).

Using MFC CCriticalSection

- Prefer using CCriticalSection in MFC Application.
- CCriticalSection is a wrapper class around win32 CRITICAL_SECTION
- CCriticalSection automatically handles Initialization & Deletion of critical section.
- use CCriticalSection::Lock API to lock
- use CCriticalSection::unlock API to unlock.

```
#define _AFXDLL
#include <iostream>
#include <afxmt.h>                                  // Step 1 : Include the header file <afxmt.h>

CCriticalSection g_csLock;                          // Step 2 : Declare the CCriticalSection variable

void PerformOperationInThreads( const int nThreadCount_i );
void CriticalFunctionality( const int& nThreadIndex_i );
DWORD WINAPI ThreadFunction( PVOID pVoid );

int main(){
    PerformOperationInThreads( 6 );  // Create and start different thread that calls ThreadFunction().
    return 0;
}

DWORD WINAPI ThreadFunction( PVOID pVoid ){
    g_csLock.Lock();                                // step 3 : Accquire the CCriticalSection lock.

    CriticalFunctionality( *reinterpret_cast<int*>( pVoid )); // step 4 : Execute the synchronized code section

    g_csLock.Unlock();                              // step 5 : Release the CCriticalSection lock.
    return 0;
}                                                   // Step 6: Execute step 4 to 6 for all threads.
```

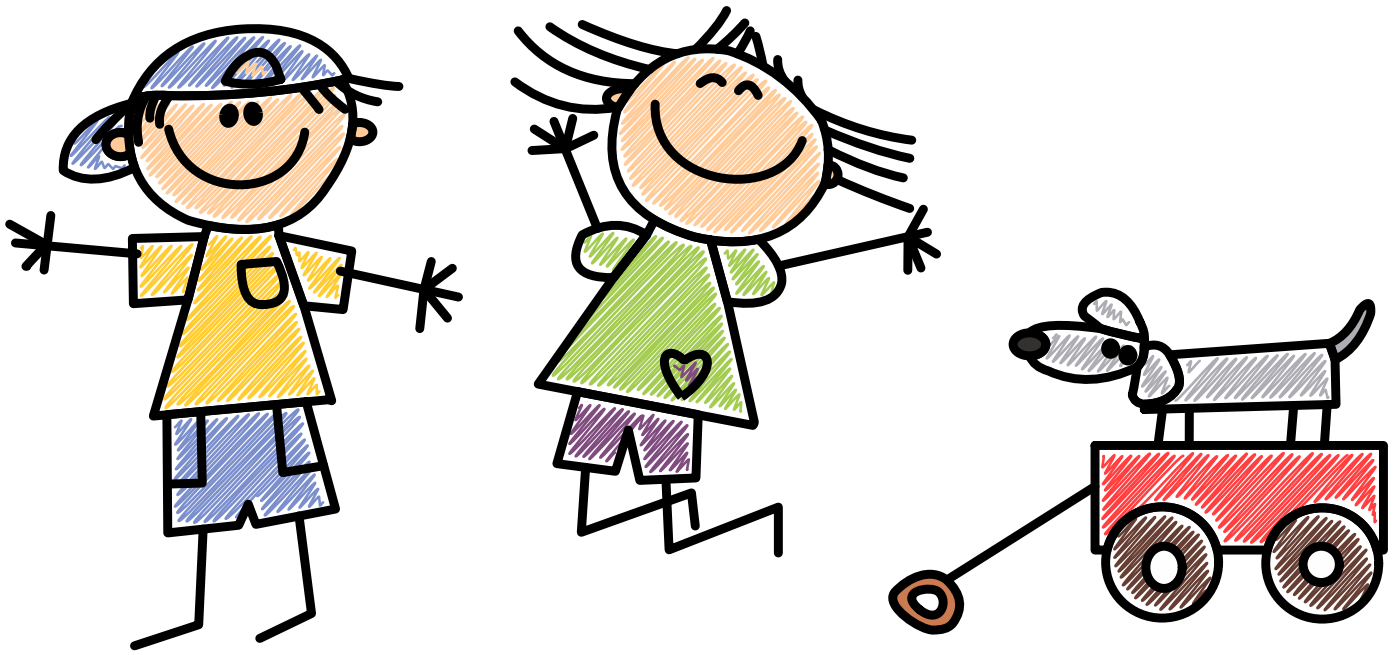
Using MFC CCriticalSection - Best practice

- Use **CSingleLock** utility class to automatic lock and unlock of CCriticalSection object.
- Advantage:
  - 100% ensured automatic unlocking
    - ✓ In all return scenarios
    - ✓during stack unwinding due to exceptions

Using MFC CCriticalSection with CSingleLock

```
#include <iostream>
#include <afxmt.h>                                  // Step 1 : Include the header file <afxmt.h>

CCriticalSection g_csLock;                          // Step 2 : Declare the CCriticalSection variable

void PerformOperationInThreads( const int nThreadCount_i );
void CriticalFunctionality( const int& nThreadIndex_i );
DWORD WINAPI ThreadFunction( PVOID pVoid );

int main(){
    PerformOperationInThreads( 6 );  // Create and start different thread that calls ThreadFunction().
    return 0;
}

DWORD WINAPI ThreadFunction( PVOID pVoid ){
    CSingleLock lock( &g_csLock, TRUE );            // step 3 : Automatic CCriticalSection lock.

    CriticalFunctionality( *reinterpret_cast<int*>( pVoid )); // step 4 : Execute the synchronized code section

    return 0;
}                                                   // Step 5: Execute step 3 to 6 for all threads.
```

Thank you



LIKE & SHARE & Subscribe to our YouTube Channel