Subject: Re: MT and variables simple question
Posted by Didier on Mon, 09 Jun 2014 18:36:18 GMT
View Forum Message <> Reply to Message

Hello Koldo,

Quote:Do you mean that I should use as many Mutex variables as shared variables?
No, in most cases all you need is one Mutex for a group of variables.
The reason for this is that you're variables are all used in one context and are all linked together .. so you can consider them as one complex variable : let's call it the context.
And it is this context that needs to be protected.

Using mutex is necessary when dealing with MT problems BUT using mutexes has several drawbacks:
using to many mutexes complexifies the code using to few mutexes multiplies the locking cases

The objective is to find the wright balance in the number of mutexes to achieve fast code (few locking cases) and maintainable code (not to many mutexes).
This can be obtained by isolating the independent resources (Resources that are not connected to each other in any way) and using a separate mutex for each one.

For example:
if you have developed a FIFO source code that you want to use to communicate between two threads. Then all you need to protect is the internal variables of the FIFO ==> you only need one protection Mutex for a FIFO.
If you use several instances of the FIFO (with several threads), you will need one protection mutex for each FIFO : if you don't do this, the application will work fine but it will get slowed by locking.

Doing correct MT programming is quite tricky and requires to have a clear view of what really needs to be protected : if a variable is not used used by several threads, then do not use a mutex to protect it (it's useless) !!!

Another alternative to using INTERLOCKED is to use 'Mutex::Lock lock(myMutex)', it will automatically protect the scope in which it is used, and this takes in account ALL CASES : I mean even the cases where C++ throw occurs:
'Mutex::Lock lock(myMutex)' will lock myMutex on 'lock' object creation and will unlock myMutex on 'lock' object destruction.
Since C++ garanties that all objects created locally are destroyed when leaving the scope on a throw ==> then the destructor of 'Mutex::Lock lock' is garantied to be called and the mutex will always be unlocked.
If you use INTERLOCKED, the mutex stays locked ... and a DEADLOCK will very certainly occur (if not managed in the catch section)

My preference goes to 'Mutex::Lock lock', I find the code more readable