Subject: MT and variables simple question
Posted by koldo on Thu, 05 Jun 2014 08:29:08 GMT
View Forum Message <> Reply to Message

Hello all

I have a doubt about using INTERLOCKED to access variables. I think we are forced to use it when reading and writing variables in MT. However, is it necessary to use it when getting a pointer to them?

For example, is this right?:

```
Array<double> vars;
...
double *data;
data10 = &vars[10];
INTERLOCKED {
  *data10 = 23;
}
```

Subject: Re: MT and variables simple question
Posted by Didier on Thu, 05 Jun 2014 20:32:36 GMT
View Forum Message <> Reply to Message

Hello Koldo,

A mutex is used to protect a variable from being modified by two threads at the same time (no matter how that variable is modified).
So if you modify a variable using the var itself or a pointer to it: the result is the same ==> you have to protect the variable

If the variable is accessed by only one thread, then there is no need to use mutexes

Subject: Re: MT and variables simple question
Posted by koldo on Fri, 06 Jun 2014 12:23:51 GMT
View Forum Message <> Reply to Message

Thank you Didier

The question is, the later example should be like this:
data10 = &vars[10];
or like this?:
INTERLOCKED {

```
    data10 = &vars[10];
}
```

Subject: Re: MT and variables simple question
Posted by Didier on Fri, 06 Jun 2014 13:48:27 GMT
View Forum Message <> Reply to Message

Hello Koldo,

As i said Quote: if you modify a variable using the var itself or a pointer to it: the result is the same ==> you have to protect the variable

So if data10 or vars[10] are accessed by several threads you have to put :

```
INTERLOCKED {
    data10 = &vars[10];
}
```

BUT ... you have to be careful which INTERLOCK method to use:

INTERLOCKED   : uses local static mutex which only protects this code (and only this one) from being accessed by several threads at same time.
INTERLOCKED(mutex) : uses the parameter mutex which allows you to protect several parts of code by calling INTERLOCKED(mutex) each time

If you only wan't to protect variables, then you probably need to use INTERLOCKED(mutex) everywhere the variable is accessed.
Here is what I mean:

```
// If you need to protect data10
Mutex mtx;
..
..
// in code of first thread
INTERLOCKED(mtx) {
    data10 = &vars[10];
}
..
..
// somewhere in another thread's code
INTERLOCKED(mtx) {
    data10 = xxxxx;
}
```

Hope I am clear enough :?

Subject: Re: MT and variables simple question
Posted by koldo on Sun, 08 Jun 2014 15:01:23 GMT
View Forum Message <> Reply to Message

Hello Didier

Do you mean that I should use as many Mutex variables as shared variables?

Many variables are shared and in many places in the code. I thought that it was enough buy using INTERLOCKED in places where shared variables are accessed.

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

---

Subject: Re: MT and variables simple question
Posted by koldo on Tue, 10 Jun 2014 06:46:16 GMT
View Forum Message <> Reply to Message

Thank you very much!

Subject: Re: MT and variables simple question
Posted by ManfredHerr on Tue, 10 Jun 2014 13:59:08 GMT
View Forum Message <> Reply to Message

My two cents:
Mutex locks are necessary only then when the "variable" is compound. For example the mentioned fifo. It must be guaranteed that the full fifo entry is written before a second thread can interrupt and read it. For simple variables like integers a.s.o. there is no need to protect because they are written and read in an "atomic" action. The writing thread cannot be interrupted during the write-operation.

In the case where only one thread writes the fifo and only one thread reads it this atomicy can be used by tricky programmers to get rid of the lock at all. They write the fifo entry completely before they increment the write index. So the reading thread will only become aware of the new entry when it is fully written. The same is with the reading thread. First the entry is read completely before the read index is incremented to allow for reuse.

Subject: Re: MT and variables simple question
Posted by mirek on Wed, 11 Jun 2014 06:16:29 GMT
View Forum Message <> Reply to Message

ManfredHerr wrote on Tue, 10 June 2014 15:59My two cents:
Mutex locks are necessary only then when the "variable" is compound. For example the mentioned fifo. It must be guaranteed that the full fifo entry is written before a second thread can interrupt and read it. For simple variables like integers a.s.o. there is no need to protect because they are written and read in an "atomic" action. The writing thread cannot be interrupted during the write-operation.

Just to clarify, read/write is perhaps atomic, but a combination of them is not. Consider simple

```
int x;
...
x++;
```

this is broken in MT, because ++ are TWO atomic operations with respect to memory variable (read AND write), despite the fact that ISA is realizing that operation with single CPU opcode.

Also, you need to carefuly check your ISA to find out what fundamental types are really atomic. Plus be prepared for nasty effects like read/write reordering (use memory barriers).

Subject: Re: MT and variables simple question
Posted by Didier on Thu, 12 Jun 2014 19:06:02 GMT
View Forum Message <> Reply to Message

Quote:Just to clarify, read/write is perhaps atomic, but a combination of them is not

+1

Mastering MT with Mutex is mandatory before trying to use Atomic operations which is a lot more tricky (I know very few people who master this)

Subject: Re: MT and variables simple question
Posted by ManfredHerr on Thu, 12 Jun 2014 19:44:02 GMT
View Forum Message <> Reply to Message

OK!
This is another example of the experience that in IT you cannot trust to what you learned some years ago. I always considered a machine instruction, i.e. an OP-Code to be executed completely, or not at all. Even if some micro software is involved. The possibility of interrupting within an instruction is new to me. However, I wonder what value is kept in the Program Counter for return from interrupt then.

I am sorry to have caused fuss. In the future, I will keep my two cents with me.

Subject: Re: MT and variables simple question

Posted by mirek on Mon, 16 Jun 2014 15:45:48 GMT
View Forum Message <> Reply to Message

ManfredHerr wrote on Thu, 12 June 2014 21:44OK!
This is another example of the experience that in IT you cannot trust to what you learned some years ago. I always considered a machine instruction, i.e. an OP-Code to be executed completely, or not at all. Even if some micro software is involved. The possibility of interrupting within an instruction is new to me. However, I wonder what value is kept in the Program Counter for return from interrupt then.

I am sorry to have caused fuss. In the future, I will keep my two cents with me.

It is not that much (in fact, not at all) about interrupting, it is more about the fact that CPU execution core is not executing opcodes as they are. Instead, everything is broken into smaller operations and executed out-of-order. So it is entirely possible that the "store" part of x++ is executed many cycles after the "load" part and a flow of other instructions or theirs parts is executed within that time.

Within single thread, these out-of-order issues are invisible to programmer, because in single thread CPU guarantees memory consistency, but with more CPU cores, the real hell begins :)

BTW, not much mentioned effect of every mutex implementation is that it takes care about read/write ordering too, doing memory barriers as necessarry...