Subject: Use same variable in different threads
Posted by koldo on Thu, 09 Dec 2010 17:26:10 GMT
View Forum Message <> Reply to Message

Hello all

What is the best way to read and write a variable in different threads?

I have seen atomic, volatile, RWMutex. However I do not know the way to declare, read and write a shared variable.

Subject: Re: Use same variable in different threads
Posted by dolik.rce on Thu, 09 Dec 2010 17:57:33 GMT
View Forum Message <> Reply to Message

Hi Koldo,

From my little knowledge (close to yours probably):

Atomic is just typedefed int, which can be manipulated by U++ functions Atomic{Read,Write,Inc,Dec,XAdd}(). They are implemented using atomic operations, so it should be guaranteed that the return value is correct even if other thread changes value of the variable while processing the function call.

Volatile tells compiler that the variable can be changed from other threads, so that compiler knows that the generated code must check for its value everytime, instead using for example value stored in cpu cache, because other thread might have changed since it was stored there.

Mutex is a mechanism that allow you to lock some code so that if other code gets to the same section it has to wait for the first one to leave. This is utilized by INTERLOCKED macro in U++.

RWMutex is a variant of mutex that allows you to mark "read code" which can be executed by any number of threads at once and "write code" that can be executed only by one thread at a time and no thread can run in write mode until all reading threads are finished.

Hope that at least some of it helps... It definitelly helped me to sort it out in my head

Best regards,
Honza

Subject: Re: Use same variable in different threads
Posted by koldo on Thu, 09 Dec 2010 21:04:48 GMT
View Forum Message <> Reply to Message

Hello Honza

Thank you for your explanation.

The question now is, how to program it?

Subject: Re: Use same variable in different threads
Posted by Didier on Thu, 09 Dec 2010 23:26:02 GMT
View Forum Message <> Reply to Message

Hi Koldo,

I would add one thing to Honza's explanation: 'Memory bariers'

Actually this is not a thing that is used directly by people but it is necessary to understand all that is going on.

Volatile allows you to access a shared variable not protected by mutex and you will be sure it always has the wright value (the cache on the processor might be different for each thread, so in extreme cases you can get de synchronised values of one variable from time to time ..... ==> bugs impossible to find).
Of course if the variable is larger than the processor bus ... the R/W process will not be atomic so you need a mutex !

Memory barriers are points where the cpu cache is flushed so all the synchronisation problems just disappear
In mutexes, such barriers are used to avoid such problems, and therefore when using a mutex lock/unlock procedure, you protect the resource (and it's memory) but you also flush the cash at each mutex lock/unlock.
So .... when using a mutex there is no need to use volatile it might even slow down you're app.

Finally, to answer you're question:
* if the variable (and it's context !) is just an int or smaller it can be accessed atomically ==> use this method, it's the fastest one (volatile will work in most cases but to be sure use the dedicated functions)
* otherwise you have to use the good old mutex, no may out !
  - mutex.lock()
  - do you're thing ..R..W....etc
  - mutex.unlock()

NB: the volatile/synchronisation problems mostly has effects on programs compiled in O3.

Hope this helps you !

Subject: Re: Use same variable in different threads
Posted by dolik.rce on Fri, 10 Dec 2010 05:49:51 GMT
View Forum Message <> Reply to Message

Quote:
The question now is, how to program it?
That is always the question Unfortunatelly the answer is not simple and it depends a lot on what you are trying to achieve...

As Didier pointed out, for simple int compatible variables Atomic* functions are best. For something rarely used (e.g. flag to stop all threads) I ussually use volatile. And fore manipulating larger objects, you should definitely use mutex. The basic approach for mutex might be to lock it everytime before you access the variable that is shared between threads and unlock after you are done with it. I think it is best to have one mutex for each object that is accessed from various threads.

In my experience, the best solution is to avoid inter-thread communication whenever possible Of course, than is definitely not an universal solution...

Honza

Subject: Re: Use same variable in different threads
Posted by koldo on Fri, 10 Dec 2010 09:01:26 GMT
View Forum Message <> Reply to Message

Thank you all for your answers

- So, if the var is just an int or smaller, I would do:

```
// Declaration
Atomic myRunProcess = true;

...

// Thread process. Permitted
while(AtomicRead(myRunProcess)) {
... do all
}


// Thread process. Not permitted ????
while(myRunProcess) {
... do all
}
```

- If the var is bigger, I would use mutex but, how?
For example, imagine there is a struct to share:

```
struct mySharedData {
    int a;
    String b;
    double c;
};
```

What do I have to do to share this struct between different threads and main thread ?

Subject: Re: Use same variable in different threads
Posted by dolik.rce on Fri, 10 Dec 2010 10:34:24 GMT
View Forum Message <> Reply to Message

Here is a little sample code for you, demonstrating the Mutex and Atomic. It is stupidly written (on purpose, of course) to show how bad it can go if you don't use the locking. See for yourself by uncommenting the "#define NOMUTEX".

In this example it is easy to see what is wrong with the code and it might be possible to fix it so it works better even without locking, but in many cases the errors are more subtle and harder to find. Also remember that if you access the shared variable from multiple places in you code, you should use the mutex around each of them.

Here is the code: #include <Core/Core.h>
using namespace UPP;

```
struct mySharedData {
 int a,b;
 String c;
};

Mutex m;
mySharedData data;
Atomic threadnum;

//uncomment this to see how it ends up when not using mutex
//#define NOMUTEX

void ThreadFunction(){
 int thisthread=AtomicInc(threadnum); //thisthread serves as an unique identifier of thread, just for the sake of the examples clarity
 for(int i = 0; i < 10; i++){
  Thread::Sleep(Random(10)); // Pretend some work...
#ifndef NOMUTEX
  m.Enter();     // Enter the section that accesses the shared data
#endif
  data.a++;
  data.c<<data.a<<": ";
  Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file
```

```
access)
  data.b=data.a;
  data.c<<data.b<<" ["<<thisthread<<"]\n";
#ifndef NOMUTEX
  m.Leave();    // we don't need the exclusive access to data anymore, release the mutex
#endif
 }
}

CONSOLE_APP_MAIN {
 //initialize variables (no threads yet, so it is save to make it without mutex);
 threadnum=0;
 data.a=0;
 data.b=0;
 data.c="";

 // start the threads
 Thread::Start(callback(ThreadFunction));
 Thread::Start(callback(ThreadFunction));
 Thread::Start(callback(ThreadFunction));

 while(Thread::GetCount()); // the main thread should wait for other to finish, so we can see the
results
  Thread::Sleep(100);

 Cout()<<data.c<<"\n";
}
```

Honza

Subject: Re: Use same variable in different threads
Posted by koldo on Fri, 10 Dec 2010 12:19:32 GMT
View Forum Message <> Reply to Message

Hello Honza

So:

- Atomic vars has to be handled using AtomicXXX() functions

- Variables handled with Mutex are declared normally, but all use of them (read or write) has to be between an Enter() and a Leave()

Subject: Re: Use same variable in different threads
Posted by gprentice on Fri, 10 Dec 2010 13:08:20 GMT

No expert here either but when you're sharing data between threads, I think you need both volatile and synchronization.

A mutex ensures that the compiler/linker can't optimise code across the mutex entry, that the cache is flushed and that only one thread can be executing the code the mutex protects (i.e. it's synchronized).

http://msdn.microsoft.com/en-us/library/ms686355(v=VS.85).aspx

volatile ensures that the compiler/linker can't optimise your code and use a cached value. With Microsoft, optimisation can occur at link time. On some platforms you can get away without volatile if you call a global function that the compiler can't see - the compiler has to assume that global function might modify the variable you're sharing so is forced to re-read the variable from memory, but that is't safe with Microsoft.

There's also thread local storage - see thread__

Regarding atomic - on Win32, 32 bits are atomic and on Win64, 64 bits are atomic. On Win32, the atomixXXX functions use the Interlocked... functions that allow you to read/write without being interrupted by another thread etc, and also provide a memory barrier.

Hence I think you need
volatile mySharedData data;
volatile Atomic threadnum;
Anyway, I don't think that using volatile would be wrong, even if it's not always necessary.

Graeme

Subject: Re: Use same variable in different threads
Posted by dolik.rce on Fri, 10 Dec 2010 13:16:31 GMT

koldo wrote on Fri, 10 December 2010 13:19Hello Honza

So:

- Atomic vars has to be handled using AtomicXXX() functions

- Variables handled with Mutex are declared normally, but all use of them (read or write) has to be between an Enter() and a Leave()
Yes. I would just add that Atomic can be handled as int if you know that no clash can happen (i.e. when only one thread can access it at given moment). Similar with mutex, you can skip the locking if you are sure that there is only one thread executing that code. But those are details and

being more cautious than necessary does never hurt

Graeme is completely right about the volatile I think. I completely forgot to mark the variables

Honza

Subject: Re: Use same variable in different threads
Posted by gprentice on Fri, 10 Dec 2010 13:34:50 GMT
View Forum Message <> Reply to Message

Actually, on second thoughts I think the code as written is fine without volatile because the mutex forces the shared data to be updated in memory. Possibly an atomic variable that is read or written outside of a mutex region is more likely to need volatile.

Edit : except that doesn't ensure the hardware doesn't use a cached value ... so now I just noticed AtomicRead and AtomicWrite in U++ - so I take it all back - you probably don't need volatile at all

Graeme

Subject: Re: Use same variable in different threads
Posted by Didier on Fri, 10 Dec 2010 17:50:25 GMT
View Forum Message <> Reply to Message

Hi all,

Quote:
Actually, on second thoughts I think the code as written is fine without volatile because the mutex forces the shared data to be updated in memory. Possibly an atomic variable that is read or written outside of a mutex region is more likely to need volatile.

Yes this is the case, and is what I explained in my earlier post.

Maybe what you need Koldo, in the example you gave is something like this:

```
#include <CtrlLib/CtrlLib.h>

using namespace Upp;
class AtomicVar
{
 private:
  Atomic val;
 public:
```

```
  AtomicVar() {};

  AtomicVar( AtomicVar& p) { AtomicWrite(val, p); }
  template <class T>
  AtomicVar& operator=(const T& p) { AtomicWrite(val, p); }

  operator int() {return AtomicRead(val); }

};

GUI_APP_MAIN
{

 AtomicVar v;

 v=5;

 String str = "Value = ";

 str << (int)v;

 LOG(str);



}
```

All the accesses to 'v' are atomic, so in you're case all you have to do is replace
Atomic myRunProcess = true;
with
AtomicVar myRunProcess = true;

and just use myRunProcess normaly, without worrying about Atomic considerations !!

Subject: Re: Use same variable in different threads
Posted by gprentice on Sat, 11 Dec 2010 00:37:17 GMT
View Forum Message <> Reply to Message

Didier wrote on Sat, 11 December 2010 06:50Hi all,

Quote:
Actually, on second thoughts I think the code as written is fine without volatile because the mutex forces the shared data to be updated in memory. Possibly an atomic variable that is read or

written outside of a mutex region is more likely to need volatile.

Yes this is the case, and is what I explained in my earlier post.

Just to be clear, apparently there are situations when volatile is useful
http://www.drdobbs.com/high-performance-computing/212701484

but in general, you need a memory barrier when accessing shared data.

After googling for a while, I haven't found any clear explanation of how a mutex solves the caching and memory visibility issues. I'm guessing that both acquire mutex and release mutex flush the entire memory cache for all CPUs/caches and prevent hardware re-ordering across the memory barrier - which deals with the hardware problem. For dealing with compiler/linker optimisation, I'm guessing that the call to the "external" acquire/release mutex function forces the compiler not to cache values across that call because it can't tell what memory that function call is going to access. Explanations of how a mutex works don't seem to mention these issues.

That AtomicVar class looks like a good idea. Should the copy constructor be
AtomicVar(const AtomicVar& p) { AtomicWrite(val, AtomicRead(p.val)); }

Graeme

Subject: Re: Use same variable in different threads
Posted by Didier on Sat, 11 Dec 2010 22:13:29 GMT
View Forum Message <> Reply to Message

Hi Graeme,

I'm very far from beeing an expert and I faced the same problems as you when I had to write some tough MT code: no clear documentation on the subject.
Thank's for the link, very interresting

That AtomicVar class looks like a good idea. Should the copy constructor be
AtomicVar(const AtomicVar& p) { AtomicWrite(val, AtomicRead(p.val)); }

Yes, it was what I coded at first but it didn't compile because AtomicRead() does not accept const parameter.

Anyway this sample class needs some polishing and all the classic operators should also get overloaded.

Subject: Re: Use same variable in different threads
Posted by gprentice on Sun, 12 Dec 2010 00:57:03 GMT
View Forum Message <> Reply to Message

Hi Didier

This compiles for me with MSC10 and GCC something. I had to change operator= to get it to compile. It's ok for CV qualification to increase so it *should* compile.
BTW - it's a little bit confusing that on Win32, Atomic is declared as long but the Atomic functions have int parameters. Possibly operator int() below could be operator Atomic(), to be more platform independent.

```
 class AtomicVar
{
 private:
  Atomic val;
 public:
  AtomicVar() {};

  AtomicVar( const AtomicVar& p) { AtomicWrite(val, AtomicRead(p.val)); }
  template <class T>
  AtomicVar& operator=(const T& p) { AtomicWrite(val, p); return *this; }

  operator int() {return AtomicRead(val); }

};
```

Graeme

Subject: Re: Use same variable in different threads
Posted by mirek on Sun, 12 Dec 2010 08:43:22 GMT
View Forum Message <> Reply to Message

gprentice wrote on Fri, 10 December 2010 19:37
After googling for a while, I haven't found any clear explanation of how a mutex solves the caching and memory visibility issues.

Because mutex acts as memory barrier...

If you read _very_ carefully the POSIX threads docs, it can be (sort of) derived from the information there. Well, to be more specific, the mutex implementation has to act as memory barrier to support POSIX mutex specification...

Mirek

Subject: Re: Use same variable in different threads
Posted by koldo on Sun, 12 Dec 2010 22:16:50 GMT
View Forum Message <> Reply to Message

Hello all

Nice answers !

For now if I understand well I will use:

- AtomicVar class. Nice!

- Mutex as here:

```
struct mySharedData {
 int a, b;
 String c;
 Mutex m;
};

mySharedData data;

void ThreadFunction(){
 for(int i = 0; i < 10; i++){
  Thread::Sleep(Random(10)); // Pretend some work...
  data.m.Enter();     // Enter the section that accesses the shared data
  data.a = i;
  data.c << data.a << "\n";
  Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file
access)
  data.b = data.a;
  data.m.Leave();    // we don't need the exclusive access to data anymore, release the mutex
 }
}
```

Subject: Re: Use same variable in different threads
Posted by tojocky on Mon, 13 Dec 2010 10:35:39 GMT
View Forum Message <> Reply to Message

koldo wrote on Mon, 13 December 2010 00:16Hello all

Nice answers !

For now if I understand well I will use:

- AtomicVar class. Nice!

- Mutex as here:

```
struct mySharedData {
 int a, b;
 String c;
 Mutex m;
};

mySharedData data;

void ThreadFunction(){
 for(int i = 0; i < 10; i++){
  Thread::Sleep(Random(10)); // Pretend some work...
  data.m.Enter();     // Enter the section that accesses the shared data
  data.a = i;
  data.c << data.a << "\n";
  Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file
access)
  data.b = data.a;
  data.m.Leave();    // we don't need the exclusive access to data anymore, release the mutex
 }
}
```

But what about read control?
Example:
read data.a
write data.a and data.b
read data.b (b will be modified)

In this case I think:
 1. the write look need to wait until the all reads was finished. and every read locks need to wait until the write lock was finished.
 2. Or reads and writes will use only one lock (data.m). But in this case will not possible multiple parallel reads.

Later I will try to provide a simple example.

Subject: Re: Use same variable in different threads
Posted by gprentice on Mon, 13 Dec 2010 10:46:52 GMT
View Forum Message <> Reply to Message

I'm guessing that if you have one thread writing to shared data and another thread reading the

shared data (no writing), both threads have to acquire the mutex before accessing the data - even a thread that's only reading. There are ways of organizing things so that multiple threads can have read access to the data at the same time, instead of all blocking each other.

http://en.wikipedia.org/wiki/Readers-writers_problem

Graeme

Edit - oops, I didn't see tojocky's post before I wrote mine but it looks like we're on the same subject.

Subject: Re: Use same variable in different threads
Posted by Didier on Mon, 13 Dec 2010 21:48:03 GMT
View Forum Message <> Reply to Message

Hi Koldo,

I would add just one last thing:
DEADLOCKs ==> what MT programs dread the most.

In practice, there is a case in C++ where deadlocks can appear without notice : when an exception occurs.

Imagine an exception occurs in the middle of you're "slow operation" ==> the mutex doesn't get released ==> a deadlock will come up soon enough

To avoid this I use, what I call a ScopedLock class:

```
template<class MUTEX>
class ScopedLock
{
private:
 MUTEX& mutex;

private:
 // The following constructor/operators are expilictly FORBIDDEN
       // because they have no meaning
 ScopedLock(void) {};
 ScopedLock(const ScopedLock& ) {};
 ScopedLock& operator=(ScopedLock& ) { return *this; };

public:
 inline  ScopedLock(MUTEX& mut)
 : mutex(mut)
 {
```

```
  mutex.lock();
 }

 inline ~ScopedLock(void)
 {
  mutex.unLock();
 }
};
```

The point is to create a 'ScopedLock' object when interring a protected zone of code, and when the scope ends ==> the unlock is automatically done IN ALL POSSIBLE CASES !! even exceptions: The compiler handles all for you

so you're code would become:

```
void ThreadFunction(){
 for(int i = 0; i < 10; i++){
  Thread::Sleep(Random(10)); // Pretend some work...
  {
   ScopedLock(data.m);// Enter the section that accesses the shared data
   data.a = i;
   data.c << data.a << "\n";
   Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file access)
   data.b = data.a;
  } // implicit release
 }
}
```

You don't have any more "mutex leaks" possible

Subject: Re: Use same variable in different threads
Posted by koldo on Tue, 14 Dec 2010 08:02:34 GMT
View Forum Message <> Reply to Message

Great Didier

The status now is to use AtomicVar and ScopedLock.

Thank you all .

Subject: Re: Use same variable in different threads
Posted by mirek on Sat, 25 Dec 2010 19:45:47 GMT

ScopedLock is in U++ as Mutex::Lock.

For read/write access, please notice RWMutex (multiple readers, just one writer). It has 'scoped' RWMutex::ReadLock and RWMutex::WriteLock variants.

As for Atomic variables, its basic feature is that they can accessed and perform increments/decrements from multiple threads WITHOUT caring about mutexes or barriers.

Subject: Re: Use same variable in different threads
Posted by koldo on Sun, 02 Jan 2011 00:12:29 GMT

Hello Didier

Trying to use the code like this I get a "error C2955: 'ScopedLock' : use of class template requires template argument list". What am I doing wrong?

Why ScopedLock has to be templated?

Didier wrote on Mon, 13 December 2010 22:48Hi Koldo,

I would add just one last thing:
DEADLOCKs ==> what MT programs dread the most.

In practice, there is a case in C++ where deadlocks can appear without notice : when an exception occurs.

Imagine an exception occurs in the middle of you're "slow operation" ==> the mutex doesn't get released ==> a deadlock will come up soon enough

To avoid this I use, what I call a ScopedLock class:

```
template<class MUTEX>
class ScopedLock
{
private:
 MUTEX& mutex;

private:
 // The following constructor/operators are expilictly FORBIDDEN
       // because they have no meaning
 ScopedLock(void) {};
 ScopedLock(const ScopedLock& ) {};
```

```
 ScopedLock& operator=(ScopedLock& ) { return *this; };

public:
 inline  ScopedLock(MUTEX& mut)
 : mutex(mut)
 {
  mutex.lock();
 }

 inline ~ScopedLock(void)
 {
  mutex.unLock();
 }
};
```

The point is to create a 'ScopedLock' object when interring a protected zone of code, and when the scope ends ==> the unlock is automatically done IN ALL POSSIBLE CASES !! even exceptions: The compiler handles all for you

so you're code would become:

```
void ThreadFunction(){
 for(int i = 0; i < 10; i++){
  Thread::Sleep(Random(10)); // Pretend some work...
  {
   ScopedLock(data.m);// Enter the section that accesses the shared data
   data.a = i;
   data.c << data.a << "\n";
   Thread::Sleep(20); //Let's pretend that some slow operation happens here (for example file access)
   data.b = data.a;
  } // implicit release
 }
}
```

You don't have any more "mutex leaks" possible

Subject: Re: Use same variable in different threads
Posted by gprentice on Sun, 02 Jan 2011 02:08:35 GMT

View Forum Message <> Reply to Message

It's time you learnt C++ templates Koldo

ScopedLock is a template class because it's declared like this

```
template<class MUTEX>
```

class ScopedLock
{

It has one template parameter named MUTEX. When you use a template class you have to specify template arguments corresponding to all the template parameters
i.e.
#include <Core/Mt.h>
struct mySharedData {
 int a, b;
 String c;
 Mutex m;
};

mySharedData data;

    // ...
    ScopedLock<Mutex> xyz(data.m);

where Mutex (the template argument in between <>) is the U++ mutex class in core/mt.h
You also need to change the call mutex.lock() to mutex.Enter() and mutex.unLock() to mutex.Leave

It's actually pretty much pointless to make the ScopedLock class a template class so you could either change it to a normal class or maybe put a default argument

template<class MUTEX = Mutex>
class ScopedLock
{

then you can do

    // ...
    ScopedLock xyz(data.m);

When xyz goes out of scope, the destructor is called which releases the mutex.

By the way, this code is invalid as you can't call a constructor.
    ScopedLock(data.m);// Enter the section that ...

The Mutex::Lock class that Mirek mentioned, creates a temporary mutex so can't be used in your case if you have other code elsewhere that is accessing the data.

Graeme

Subject: Re: Use same variable in different threads

Posted by Didier on Sun, 02 Jan 2011 11:18:44 GMT
View Forum Message <> Reply to Message

Hi Koldo,

In fact this helper class is intended to work with all sorts of mutexes: whatever you want as long as the methods 'lock' and 'unlock' exist.

The code is therefore 'portable'... but it can't be used directly in Upp since the methods 'lock' & 'unlock' are called 'Enter' and 'Leave'
The reason why I did this is to make 'POLICY BASED DESIGN' code:
http://en.wikipedia.org/wiki/Policy-based_design
http://loki-lib.sourceforge.net/index.php?n=Main.Policy-base dDesign
http://www.amazon.com/exec/obidos/ASIN/0201704315/modecdesi- 20

In you're case, just use
Mutex::Lock(<you're mutex>)
It should do what you need.

Quote:The Mutex::Lock class that Mirek mentioned, creates a temporary mutex
This is not true, a reference is used so the behavior is exactly the same as my template.

Subject: Re: Use same variable in different threads
Posted by gprentice on Sun, 02 Jan 2011 12:11:35 GMT
View Forum Message <> Reply to Message

Didier wrote
In you're case, just use
Mutex::Lock(<you're mutex>)
It should do what you need.

Quote:The Mutex::Lock class that Mirek mentioned, creates a temporary mutex
This is not true, a reference is used so the behavior is exactly the same as my template.

uh, oops, you're so right. The code Koldo needs is then

```
Mutex::Lock xyz(data.m);
```

Subject: Re: Use same variable in different threads
Posted by tojocky on Sun, 02 Jan 2011 14:35:53 GMT
View Forum Message <> Reply to Message

mirek wrote on Sat, 25 December 2010 21:45ScopedLock is in U++ as Mutex::Lock.

For read/write access, please notice RWMutex (multiple readers, just one writer). It has 'scoped' RWMutex::ReadLock and RWMutex::WriteLock variants.

As for Atomic variables, its basic feature is that they can accessed and perform increments/decrements from multiple threads WITHOUT caring about mutexes or barriers.

Mirek,

You read my mind. RWMutex seems to be perfect.
Windows variant I see
1. a limitation: only LONG_MAX concurrent reads can be. if concurrent reads > LONG_MAX then result is undefined.
Linux variant is a kernel variant. I didn't found the source code and can't say the opinion.
2. If write in recursive mode by the same phread will wait to infinitely.

The linux version is more pretty realized according by this links source code:
http://www.jbox.dk/sanos/source/include/pthread.h.html
http://www.jbox.dk/sanos/source/lib/pthread/rwlock.c.html

Subject: Re: Use same variable in different threads
Posted by mirek on Sun, 02 Jan 2011 17:38:41 GMT
View Forum Message <> Reply to Message

tojocky wrote on Sun, 02 January 2011 09:35
Windows variant I see
1. a limitation: only LONG_MAX concurrent reads can be. if concurrent reads > LONG_MAX then result is undefined.

Which is not real limitation at all. You are not going to have LONG_MAX threads anyway - you would run out of memory much sooner...

Subject: Re: Use same variable in different threads
Posted by koldo on Sun, 02 Jan 2011 18:06:27 GMT
View Forum Message <> Reply to Message

Sorry boys

I do not understand.....

Just to simplify it:

Data to share:
struct ToProtect {
 ...
 Mutex mtx;
};

Code protected by the Mutex. This seem to work:
 mtx.Enter();
 ...
 mtx.Leave();

Theoretically ScopedLock saves us the "Leave". However it does not work now.

How ScopedLock should be and how it would have to be used?

This does compiling errors:

ScopedLock<Mutex>(mtx);

Subject: Re: Use same variable in different threads
Posted by mirek on Sun, 02 Jan 2011 21:56:56 GMT
View Forum Message <> Reply to Message

koldo wrote on Sun, 02 January 2011 13:06Sorry boys

I do not understand.....

Just to simplify it:

Data to share:
struct ToProtect {
 ...
 Mutex mtx;
};

Code protected by the Mutex. This seem to work:
 mtx.Enter();
 ...
 mtx.Leave();

Not sure about ScopedLock, but

Mutex::Lock dummyname(mtx);

should work.

Or, if you wish, you can also use

INTERLOCKED_(mtx) {
}

Mirek

P.S.: For 'dummyname', I tend to use '__' to indicate that the name is irrelevant.

Mutex::Lock __(mtx);

Subject: Re: Use same variable in different threads
Posted by koldo on Mon, 03 Jan 2011 08:17:38 GMT
View Forum Message <> Reply to Message

Thank you Mirek

This
Quote:INTERLOCKED_(mtx) {
} is exactly what I wanted!
