Subject: Sharing and Locking
Posted by gridem on Sun, 07 Mar 2010 08:27:01 GMT
View Forum Message <> Reply to Message

I've prepared the presentation for my colleagues about multithreading techniques of using shared_ptr, COW etc. Unfortunately, Ptr and Pte don't use atomic operations so I used boost::shared_ptr as base holder for the data in the presentation.

I think it will be very useful.

Any comments are welcome.

File Attachments
1) Sharing_and_locking.ppt, downloaded 674 times

Subject: Re: Sharing and Locking
Posted by mirek on Sun, 07 Mar 2010 13:40:28 GMT
View Forum Message <> Reply to Message

gridem wrote on Sun, 07 March 2010 03:27I've prepared the presentation for my colleagues about multithreading techniques of using shared_ptr, COW etc. Unfortunately, Ptr and Pte.

1) Ptr and Pte are VERY different beasts as compared to shared_ptr (but you probably know that).

2) They DO use atomic operations, therefore within its modus operandi, they are completely MT safe (as is or should be the whole U++ now).

Mirek

Subject: Re: Sharing and Locking
Posted by gridem on Sun, 07 Mar 2010 15:38:36 GMT
View Forum Message <> Reply to Message

Mirek, thank you for your answer.
luzr wrote on Sun, 07 March 2010 16:40
1) Ptr and Pte are VERY different beasts as compared to shared_ptr (but you probably know that).

Yes, it different but tries to solve the same kind of problems IMO. I think that shared_ptr has more cleared semantics than Pte/Ptr. May be the reason of this is that I used shared_ptr a lot before and try to use Pte/Ptr like shared_ptr.
luzr wrote on Sun, 07 March 2010 16:40
2) They DO use atomic operations, therefore within its modus operandi, they are completely MT safe (as is or should be the whole U++ now).

Mirek

So I mean that Pte uses Mutex (more precisely, StaticMutex) and in general it can have some problems in high concurency application. Lock-free implementation like atomic operations produces better performance results in general.

The original problem starts from the task to provide the cache in high-loaded service with limited amount of memory.

Subject: Re: Sharing and Locking
Posted by mirek on Mon, 08 Mar 2010 00:42:08 GMT
View Forum Message <> Reply to Message

gridem wrote on Sun, 07 March 2010 10:38Mirek, thank you for your answer.
luzr wrote on Sun, 07 March 2010 16:40
1) Ptr and Pte are VERY different beasts as compared to shared_ptr (but you probably know that).

Yes, it different but tries to solve the same kind of problems IMO.

Well, you can say that, but it is a bit far-stretched IMO. Pte/Ptr are solely for solving dangling pointer issue. Unlike shared_ptr (correct me if I am wrong), Ptr can point to stack objects and most of time they really do.

Quote:
I think that shared_ptr has more cleared semantics than Pte/Ptr. May be the reason of this is that I used shared_ptr a lot before and try to use Pte/Ptr like shared_ptr.

I wonder how you can even do that?

Quote:
So I mean that Pte uses Mutex (more precisely, StaticMutex) and in general it can have some problems in high concurency application. Lock-free implementation like atomic operations produces better performance results in general.

Yes, this correct, Pte/Ptr is not great perfomance-wise. (OTOH, Mutex is just two atomic operations

---

Subject: Re: Sharing and Locking
Posted by gridem on Mon, 08 Mar 2010 09:57:27 GMT
View Forum Message <> Reply to Message

luzr wrote on Mon, 08 March 2010 03:42Well, you can say that, but it is a bit far-stretched IMO. Pte/Ptr are solely for solving dangling pointer issue. Unlike shared_ptr (correct me if I am wrong), Ptr can point to stack objects and most of time they really do.

intrusive_ptr can do the same thing, but it needs some additional steps to emulate the same behavior. From my point of view in MT application stack object may be destroyed at any time and Ptr/Pte can not prevent from using the already destroyed object:

```
struct Foo : Pte<Foo> {
   void SomeAction()  { INTERLOCKED { ... } }
};

Ptr<Foo> ptr;

// thread 1:
{
   Foo foo;
   ptr = &foo;
} // A1: foo have been destroyed

// thread 2
if (ptr) // A2
   ptr->SomeAction(); // A3
```

For example: thread 1 creates the object and ptr references to foo. Thread 2 checks that ptr has the reference and calls the method. We can suppose that SomeAction has internal Mutex to prevent simultanious access to class values. But if between A2 and A3 the foo have been destroyed (A1), than the race takes place and the application will be crashed.

May be I cannot understand how Pte/Ptr can be used correctly but shared_ptr can prevents from such situation in more atomical and strict manner.

luzr wrote on Mon, 08 March 2010 03:42I wonder how you can even do that?

The obvious way how to resolve the same problem is the following:

```
struct Ctrl
{
   struct Base; // implementation

   Ctrl() : base(new Base) {} // at cpp file
   bool IsForeground() const { return base->IsForeground(); } // at cpp file
   void SetForeground() { base->SetForeground(); } // at cpp file
   ...
protected:
   Ctrl(Base* b) : base(b) {} // at cpp file

private:
```

```
    shared_ptr<Base> base;
};

struct Pusher : Ctrl
{
    struct Base : Ctrl::Base { ... };

    Pusher() : Ctrl(new Base) {}
    ...
};
```

Ctrl has shared semantic and can be used as value in most cases (no need const references). This idiom guarantees that base will be available at any time and will be destroyed correctly.

luzr wrote on Mon, 08 March 2010 03:42Yes, this correct, Pte/Ptr is not great perfomance-wise. (OTOH, Mutex is just two atomic operations

In case when only one object acquire the lock it is true but if lock was acquired and someone wants to acquire the same lock than it takes much more time (thread sleeps until the lock will be released, so thread goes to kernel and from kernel, on Windows it's relatively heavy operation).

Subject: Re: Sharing and Locking
Posted by gridem on Sun, 14 Mar 2010 12:37:36 GMT
View Forum Message <> Reply to Message

The above approach has objects on the heap instead of stack but it has predictable object lifetime. I think that it's the reasonable overhead to solve the considered race condition in case of object destroying.

Mirek, what do you think?

Subject: Re: Sharing and Locking
Posted by mirek on Sun, 14 Mar 2010 17:58:09 GMT
View Forum Message <> Reply to Message

gridem wrote on Sun, 14 March 2010 08:37The above approach has objects on the heap instead of stack but it has predictable object lifetime. I think that it's the reasonable overhead to solve the considered race condition in case of object destroying.

Mirek, what do you think?

I am still not quite sure what you are trying to solve:)

What I think you are trying to do is to avoid dangling pointer. Anyway, making pointer itself dangling helps only a bit and perhaps is not a good strategy: Pointer itself can still exist, but the

state of object can be "destroyed". So it may seal some references to it, but IMO is not a good way.

Now maybe my experiences are not wide enough, but I belive that so far, I had little problems with race conditions of this kind in MT code. I guess, usually the best is to make things simple and not get involved into any shared ownership, which after all is the cornerstone of U++ design.

Subject: Re: Sharing and Locking
Posted by gridem on Mon, 15 Mar 2010 20:26:32 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 14 March 2010 20:58
I am still not quite sure what you are trying to solve:)

What I think you are trying to do is to avoid dangling pointer. Anyway, making pointer itself dangling helps only a bit and perhaps is not a good strategy: Pointer itself can still exist, but the state of object can be "destroyed". So it may seal some references to it, but IMO is not a good way.

Now maybe my experiences are not wide enough, but I belive that so far, I had little problems with race conditions of this kind in MT code. I guess, usually the best is to make things simple and not get involved into any shared ownership, which after all is the cornerstone of U++ design.

OK, let me to clarify the problem statement.

Suppose that we want to share some data between 2 threads. The first thread (SetterThread) will create the global variable and put the pointer to such data, than the data will be destoyed. The second (AccesserThread) will try to access to the data and if such data will exist than it will assign some value. From U++ it looks like this:

```
void SetterThread()
{
   while (true)
   {
      Data d;
      *DataAccess() = &d;
   }
}

void AccesserThread()
{
   while (true)
   {
      Ptr<Data> d = *DataAccess();
      if (d)
         d->a = 2;
```

```
    }
}
```

I use StaticAutoLock to prevent simultanious writing to the global data (see presentation for autolocking technique). If I start the following threads I will obtain the general protection failure error message (on Windows). The result will be better (crash will take place quicker) when the application will be started on multicore processor.

The specified code can be rewritten using the boost shared_ptr. In that case the global value must have the weak_ptr as the reference to the value in SetterThread. Corresponding code will be:

```
void SetterThread()
{
    while (true)
    {
        shared_ptr<Data> d(new Data);
        *DataAccess() = d;
    }
}

void AccesserThread()
{
    while (true)
    {
        shared_ptr<Data> d = DataAccess()->lock();
        if (d)
            d->a = 2;
    }
}
```

In that case the application will never be crashed due to atomical conversion from weak_ptr to shared_ptr using lock() method in weak_ptr (see boost documentation for details).

This simple example shows that Ptr doesn't prevent from dangling pointer in concurrent application. This is not the problem in single threaded model and in MT when the access can be serialized using the "big lock" like GuiLock. But in other cases it can lead to problem with stability. This is the main reason and what I want to demonstrate.

The attachement contains the full code to compile and check.

File Attachments

1) TestPtrMT.zip, downloaded 358 times

Subject: Re: Sharing and Locking
Posted by mirek on Tue, 16 Mar 2010 05:02:39 GMT
View Forum Message <> Reply to Message

gridem wrote on Mon, 15 March 2010 16:26
Suppose that we want to share some data between 2 threads. The first thread (SetterThread) will create the global variable and put the pointer to such data, than the data will be destoyed.

I would stop right there and asked "why?" I would prefer using the data instead of pointer.

However, IF I would insist on using the pointer, then the pointer itself is shared resource and you need a lock while accessing it. No matter if it is raw pointer or Pte/Pte.

However, I agree that the existence of weak_ptr::lock is an advantage in some cases, but

boost manual
Even if p.reset() is executed in another thread, the object will stay alive until r goes out of scope or is reset. By obtaining a shared_ptr to the object, we have effectively locked it against destruction.

scares my insticts to the death - this is exactly the case I was speaking about - you are still accessing zombie object that is not supposed to exist anymore.

Quote:
The second (AccesserThread) will try to access to the data and if such data will exist than it will assign some value. From U++ it looks like this:

```
void SetterThread()
{
   while (true)
   {
      Data d;
      *DataAccess() = &d;
   }
}

void AccesserThread()
{
   while (true)
   {
      Ptr<Data> d = *DataAccess();
      if (d)
         d->a = 2;
   }
}
```

Well, obviously, the code is missing serialization of DataAccess...

Now, perhaps we should try hard to add some sort of "Lock" to Ptr and make it wholy atomic, if

that is possible. But I do not think that the impact in real world apps would be worth of it.

Mirek

Subject: Re: Sharing and Locking
Posted by mirek on Tue, 16 Mar 2010 05:14:59 GMT
View Forum Message <> Reply to Message

gridem wrote on Mon, 15 March 2010 16:26

```
void AccesserThread()
{
   while (true)
   {
      shared_ptr<Data> d = DataAccess()->lock();
      if (d)
         d->SetA(2); // little change
   }
}
```

P.S.: Please note that above code is MT incorrect in most cases (I have changed it a bit, without the change it might be incorrect sometimes too).

Subject: Re: Sharing and Locking
Posted by gridem on Tue, 16 Mar 2010 07:04:37 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 16 March 2010 08:02
I would stop right there and asked "why?" I would prefer using the data instead of pointer.

The answer is simple: U++ already uses the same idiom . See for example:

CtrlCore.h:

```
 static  Ptr<Ctrl> focusCtrl;
 static  Ptr<Ctrl> focusCtrlWnd;
 static  Ptr<Ctrl> lastActiveWnd;
 static  Ptr<Ctrl> caretCtrl;
```

luzr wrote on Tue, 16 March 2010 08:02
However, IF I would insist on using the pointer, then the pointer itself is shared resource and you

need a lock while accessing it. No matter if it is raw pointer or Pte/Pte.

Yes, you are completely right.

luzr wrote on Tue, 16 March 2010 08:02
However, I agree that the existence of weak_ptr::lock is an advantage in some cases, but

boost manual
Even if p.reset() is executed in another thread, the object will stay alive until r goes out of scope or is reset. By obtaining a shared_ptr to the object, we have effectively locked it against destruction.

scares my insticts to the death - this is exactly the case I was speaking about - you are still accessing zombie object that is not supposed to exist anymore.

No, the considered situation is a bit more complicated. Because I used not shared_ptr for global variable but weak_ptr, the object will live until it will be destroyed in thread 1. But if I was successfull on converting from weak_ptr to shared_ptr, than the object lifetime will be longer and will be destroyed when loop in thread 1 and thread 2 will be restarted. In any case the object will not be in partial (or zombie) state when it will be destoyed in destructor instead of some method like Close, Destroy or other.
luzr wrote on Tue, 16 March 2010 08:02
Well, obviously, the code is missing serialization of DataAccess...

Yes. I don't serialize because my primary goal was to show the race in usage pattern if(data) data->... But of course, accurate solution must have two locks: global and internal.
luzr wrote on Tue, 16 March 2010 08:02
Now, perhaps we should try hard to add some sort of "Lock" to Ptr and make it wholy atomic, if that is possible. But I do not think that the impact in real world apps would be worth of it.

Mirek
I think that for GUI application and GUI controls like Ctrl it's not necessary because it serialize access to it using global locks. It also serializes when constructions like PostCallBack are used. If I use the main thread to manipulate the data and to destroy it, then there is no any problems. The problems may occurs when I want to create the real MT application without GUI and try to access to global variables or global list of variables through Ptr.

Subject: Re: Sharing and Locking
Posted by mirek on Tue, 16 Mar 2010 22:50:15 GMT
View Forum Message <> Reply to Message

gridem wrote on Tue, 16 March 2010 03:04luzr wrote on Tue, 16 March 2010 08:02
I would stop right there and asked "why?" I would prefer using the data instead of pointer.

The answer is simple: U++ already uses the same idiom . See for example:

CtrlCore.h:

```
static  Ptr<Ctrl> focusCtrl;
static  Ptr<Ctrl> focusCtrlWnd;
static  Ptr<Ctrl> lastActiveWnd;
static  Ptr<Ctrl> caretCtrl;
```

But these are to solve hard to predict user inputs. In most cases where MT threads are involved, you have much better control than that.

Quote:
No, the considered situation is a bit more complicated. Because I used not shared_ptr for global variable but weak_ptr, the object will live until it will be destroyed in thread 1. But if I was successfull on converting from weak_ptr to shared_ptr, than the object lifetime will be longer and will be destroyed when loop in thread 1 and thread 2 will be restarted. In any case the object will not be in partial (or zombie) state when it will be destoyed in destructor instead of some method like Close, Destroy or other.

Well, that is not what I mean. What is bad about shared ownership is exactly that it makes the lifetime of object unpredictable.

Quote:
The problems may occurs when I want to create the real MT application without GUI and try to access to global variables or global list of variables through Ptr.

Which is something to avoid, I agree...

Mirek

Subject: Re: Sharing and Locking
Posted by gridem on Fri, 19 Mar 2010 06:44:40 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 17 March 2010 01:50Quote:
No, the considered situation is a bit more complicated. Because I used not shared_ptr for global variable but weak_ptr, the object will live until it will be destroyed in thread 1. But if I was successfull on converting from weak_ptr to shared_ptr, than the object lifetime will be longer and will be destroyed when loop in thread 1 and thread 2 will be restarted. In any case the object will not be in partial (or zombie) state when it will be destoyed in destructor instead of some method like Close, Destroy or other.

Well, that is not what I mean. What is bad about shared ownership is exactly that it makes the lifetime of object unpredictable.

Yes, lifetime object will be unpredictable in sense that if my shared_ptr will be destroyes that the object itself cannot be. But it's not a problem in most cases, you can treat it as automatic garbarge collector for C++. So if clients (thread 2) want to use object than you (thread 1) should not prevent them from any operation even if you don't need it. In my practice I cannot remember the situation when an unpredicted lifetime would be a problem.

Subject: Re: Sharing and Locking
Posted by mirek on Fri, 19 Mar 2010 06:59:23 GMT
View Forum Message <> Reply to Message

gridem wrote on Fri, 19 March 2010 02:44luzr wrote on Wed, 17 March 2010 01:50Quote:
No, the considered situation is a bit more complicated. Because I used not shared_ptr for global variable but weak_ptr, the object will live until it will be destroyed in thread 1. But if I was successfull on converting from weak_ptr to shared_ptr, than the object lifetime will be longer and will be destroyed when loop in thread 1 and thread 2 will be restarted. In any case the object will not be in partial (or zombie) state when it will be destoyed in destructor instead of some method like Close, Destroy or other.

Well, that is not what I mean. What is bad about shared ownership is exactly that it makes the lifetime of object unpredictable.

Yes, lifetime object will be unpredictable in sense that if my shared_ptr will be destroyes that the object itself cannot be. But it's not a problem in most cases, you can treat it as automatic garbarge collector for C++. So if clients (thread 2) want to use object than you (thread 1) should not prevent them from any operation even if you don't need it. In my practice I cannot remember the situation when an unpredicted lifetime would be a problem.

File objects?

IMO, works quite well as long as only memory is involved....

Mirek

Subject: Re: Sharing and Locking
Posted by gridem on Sat, 20 Mar 2010 09:21:17 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 19 March 2010 09:59
File objects?

IMO, works quite well as long as only memory is involved....

Mirek

struct FileObject

```
{
   FileObject() : impl(new Impl) {}
   typedef weak_ptr<Impl> Ref;

   bool IsOpened() const { return impl->file; }
   void Close() { impl->file.reset(); }
   void Open(const char* fname) { impl->file.reset(new File(fname)); }

private:
   struct Impl
   {
      shared_ptr<File> file;
   };

   shared_ptr<Impl> impl;
};
```

Grigory

Subject: Re: Sharing and Locking
Posted by mirek on Sun, 21 Mar 2010 06:37:56 GMT
View Forum Message <> Reply to Message

gridem wrote on Sat, 20 March 2010 05:21luzr wrote on Fri, 19 March 2010 09:59
File objects?

IMO, works quite well as long as only memory is involved....

Mirek

```
struct FileObject
{
   FileObject() : impl(new Impl) {}
   typedef weak_ptr<Impl> Ref;

   bool IsOpened() const { return impl->file; }
   void Close() { impl->file.reset(); }
   void Open(const char* fname) { impl->file.reset(new File(fname)); }

private:
   struct Impl
   {
      shared_ptr<File> file;
   };

   shared_ptr<Impl> impl;
};
```

Grigory

You miss the point: When the file is closed?

(I know when, of course, but the point is the shared ownership makes this very uncertain).

Subject: Re: Sharing and Locking
Posted by gridem on Sun, 21 Mar 2010 10:39:22 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 21 March 2010 09:37
You miss the point: When the file is closed?

(I know when, of course, but the point is the shared ownership makes this very uncertain).
OK, usage sample:

```
void SetterThread()
{
   for (int i = 0; i < cycles; ++ i)
   {
      // create file object
      FileObject file;
      // assign reference to global variable
      *DataAccess::Access() = file;
      // create file itself
      file.Open("file.txt");
      // write some text, file will be opened because accesser doesn't use close
      // (try ... catch is not needed)
      file.Write(String().Cat() << "[" << i << "] setter");
      // close the file, accesser now cannot write into file
      file.Close();
   }
}

void AccesserThread()
{
   for (int i = 0; i < cycles; ++ i)
   {
      try
      {
         // try to get the real object from global reference
         FileObject file = DataAccess::Access()->Get();
         for (int j = 0; j < internalCycles; ++ j)
         {
            // try to write into file
            file.Write(String().Cat() << "[" << i << "," << j << "] accesser");
```

```
        }
      }
      catch(Exc& e)
      {
         Out(String().Cat() << "[" << i << "] Accesser error: " << e);
      }
   }
}
```

In the considered implementation the File lifetime is always predictable while lifetime of FileObject can be longer.

See attached file for detailed information.

Regards,
Grigory.

File Attachments

1) TestPtrMT.zip, downloaded 344 times

Subject: Re: Sharing and Locking
Posted by mirek on Sun, 21 Mar 2010 13:21:27 GMT
View Forum Message <> Reply to Message

gridem wrote on Sun, 21 March 2010 06:39luzr wrote on Sun, 21 March 2010 09:37
You miss the point: When the file is closed?

(I know when, of course, but the point is the shared ownership makes this very uncertain).
OK, usage sample:

```
void SetterThread()
{
   for (int i = 0; i < cycles; ++ i)
   {
      // create file object
      FileObject file;
      // assign reference to global variable
      *DataAccess::Access() = file;
      // create file itself
      file.Open("file.txt");
      // write some text, file will be opened because accesser doesn't use close
      // (try ... catch is not needed)
      file.Write(String().Cat() << "[" << i << "] setter");
      // close the file, accesser now cannot write into file
      file.Close();
   }
}
```

```
void AccesserThread()
{
   for (int i = 0; i < cycles; ++ i)
   {
      try
      {
         // try to get the real object from global reference
         FileObject file = DataAccess::Access()->Get();
         for (int j = 0; j < internalCycles; ++ j)
         {
            // try to write into file
            file.Write(String().Cat() << "[" << i << "," << j << "] accesser");
         }
      }
      catch(Exc& e)
      {
         Out(String().Cat() << "[" << i << "] Accesser error: " << e);
      }
   }
}
```

In the considered implementation the File lifetime is always predictable while lifetime of FileObject can be longer.

See attached file for detailed information.

Regards,
Grigory.

Well, this is the exact tradeoff of GC - you have lost the capability of destructors to manage resources.

Do not get me wrong. What you present is the 'mainstream' approach. In that case, however, the question is why not to use some real GC language instead...

What we are trying to do is exactly oposite. End of block closes the file (pipe, stream, whatever). That is why shared ownership (at interface level) is not recommended...

(P.S.: Not quite sure "try/catch" is not needed there. Who will close the file if the exception leaves the block?)

Mirek

Subject: Re: Sharing and Locking

luzr wrote on Sun, 21 March 2010 16:21
Well, this is the exact tradeoff of GC - you have lost the capability of destructors to manage resources.

I just want to show that you can manage resources carefully when you use one additinal inderection. While you cannot predict the lifetime of the object wrapper (class FileObject), you can manipulate with file resources in a predictable manner (class File). Class File will be destoyed on any invocation of Close method.
luzr wrote on Sun, 21 March 2010 16:21
Do not get me wrong. What you present is the 'mainstream' approach. In that case, however, the question is why not to use some real GC language instead...

I'm not quite sure about mainstream approach. I see 2 differences:

1. Mainstream doesn't use weak_ptr as reference to the object. In my programming life I see only shared_ptr in the production code.
2. Mainstream doesn't use additional layer for resource manipulation.

Also I cannot see autolocking in production code, but it's not relevant to discussion.
luzr wrote on Sun, 21 March 2010 16:21
What we are trying to do is exactly oposite. End of block closes the file (pipe, stream, whatever). That is why shared ownership (at interface level) is not recommended...

In my example I have another additional option: if someone wants to manipulate with the object and he grants the object than it can manipulate with it without any restrictions. The File will be closed implicitly if the FileObject will be destroyed and noone has access to it. I can also close file and release file handle or any resource handles explicitly even if someone tries to use it (the example demonstrates such behavior).
luzr wrote on Sun, 21 March 2010 16:21
(P.S.: Not quite sure "try/catch" is not needed there. Who will close the file if the exception leaves the block?)

Mirek

Try/catch is not needed in SetterThread because:
1. Other clients don't use Close method.
2. Setter doesn't convert from weak_ptr to shared_ptr.

Only if one of the statements will be incorrect, the exception can be thrown.

Also file will be closed automatically even Close method will not be invoked.

Regards,
Grigory.