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 385 times