Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Mon, 17 Nov 2008 15:41:57 GMT
View Forum Message <> Reply to Message

Mirek, thank you for this important notice.

I keep working on "alternative" threading model and have some news. The thing I work on is making posting callback to queue as quick as possible.
Let`s remind previous result: plain function call took ~600 msecs of averaged execution time. If you create callback with arguments, execute it and destroy inside the cycle:

```
 for (...)
 {
  Callback cb = callback2(&tc, &TestClass::func, 100,500);
  cb();
 }
```

you get averaged execution time ~780 msecs.

To make comparison more fair, I emulate adding callback to queue, executing it and then removing it from queue:

```
BiVector<Callback> cbv;
cbv.Reserve(100);
 for (...)
 {
  cbv.AddTail(callback2(&tc, &TestClass::func, 100,500));
  cbv.Head().Execute();
  cbv.DropHead();
}
```

Testing this code gave execution time of ~820 msecs.

OK, what does make this difference between plain call and posting a callback? 1) Yes, creating/destroying of Atomic member inside callback. 2) Creation of callback object itself along with it`s internal member with virtual functions and more. Also we must keep in mind that thread will have very limited number of public callback types (as a rule). Not hundreds. Most likely something about 10 or even smaller.
What if I avoid using complex Callback object and use something simpler instead? What if I have a queue for each callback type where only arguments are stored?
It took me a number of days of thinking and a pair of dirty tricks to do it. Finally I came to something like quick queued class prototype:

```
class AThreaded
{
public:
 AThreaded()
 {
  args.SetCount(0xFF+1); //yes, simple array+"hash" instead of Index. that is because Index` elements are constant
 }

 template<class OBJECT, class P1, class P2>
```

```
 void RequestAction(void (OBJECT::*m)(P1,P2), const P1 &p1, const P2 &p2)
 {
  typedef void (OBJECT::*Func)(P1,P2);
  struct Args : public Moveable<Args>
  {
   P1   p1;
   P2   p2;
  };

  //using method pointer as hash value. notice that method`s pointer size may be >= plain (void *)
  int methodPtrSize = sizeof(m) / sizeof(unsigned);
  unsigned *cur     = (unsigned *) (&m);
  unsigned hashV    = 0;
  for (int i=0; i<methodPtrSize; ++i, ++cur) hashV+=*cur;
  hashV &= 0xFF;

  int argsI = hashV;//args[hashV];
  if (args[argsI].IsEmpty())
  {
   //creating arguments queue for new callback
   Any aa;
   aa.Create< BiVector<Args> >();
   args[hashV] = aa;
   args[argsI].Get< BiVector<Args> >().Reserve(100);
  }

  Args newArgs;
  newArgs.p1 = p1;
  newArgs.p2 = p2;

  //just emulating add+execute+drop
  BiVector<Args> &curArgsQueue = args[argsI].Get< BiVector<Args> >();
  curArgsQueue.AddTail(newArgs);
  Args &curArgs = curArgsQueue.Head();
  (((OBJECT *) this)->*m)(curArgs.p1, curArgs.p2);
  curArgsQueue.DropHead();
 }

protected:
private:
 Array<Any> args;
};
```

And execution time is... ~640 msecs. This is almost as fast as plain function call which took 600 msecs instead of 840 msecs while using classic U++ callbacks.
More of that, posting callback looks rather nice for user:

```
class TestClass : public AThreaded {...};
TestClass tc;
```

tc.RequestAction(&TestClass::func, 100, 500);

I would appreciate any feedback, particularly comments on potential problems with this code.

Investigation on "alternative" threading model continues.
