Subject: Re: Thoughts about alternative approach to multithreading
Posted by mr_ped on Tue, 18 Nov 2008 08:05:39 GMT
View Forum Message <> Reply to Message

I'm sort of newcomer in terms of MT in high level language, so excuse me if I have some stupid questions (and I'm half asleep too) ... but anyway, you did want feedback.

Mindtraveller wrote on Mon, 17 November 2008 16:41

```
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);
```

* You don't do anything in case two callback functions have same hash. I'm not sure how do you want to handle that and if you already solved it somehow.
The first thing which came to my mind when I tried to solve this was:
- store with every parameter queue the original pointer. If for new call the hash is same, but pointer different, call the action immediately. (i.e. the second action routine will be never postponed into queue)
- have several hash functions available, in case of collision try different one, if it has no collision, "rehash" whole queue table and continue with the different hash function.

* is "curArgsQueue.AddTail(newArgs);" (and the rest of them) atomic? Or the RequestAction can't be called concurrently for the same hash (function)?

* I was unable to copy/paste that piece of code and make it work. (should it work with 2008.1 + MINGW? Can you post some test package?)

* P1 and P2 must be moveable, right?