Subject: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Mon, 13 Oct 2008 23:10:14 GMT
View Forum Message <> Reply to Message

Recently I`ve met description of Erlang language approache to make understandable and debuggable multithreading (MT) in applications. In short, they avoid many difficulties with MT by proposing single and unified multithreading technique.

Each thread has it`s queue and variables. Threads see each other but they don`t see each others` variables. The only way for interaction is exchanging with events. Each event is placed into thread local queue and processed consequently.

No (public) critical sections, no deadlocks, no synchronization issues. No headache. No tricky debug.

IMO this makes sense. Recently developing MT-based classes I`ve come to the same approach. And it proved to be very stable. And almost as effective as "classic" synchronization objects.

What do you think about that? Maybe we should change Thread class development vector a little? Should we deny any Mutex/Semaphore support in favor of thread local queue member functions?

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mr_ped on Tue, 14 Oct 2008 06:27:17 GMT
View Forum Message <> Reply to Message

This makes perfect sense to me as long as you don't want to share memory(variables) between threads (for performance reasons for example).

I think there's no point to limit U++ library just to queue approach and force you to work around in other cases.
But there may be some point to layer the MT API, so if you wish to use just thread variables+queue, you will easily find proper subset of U++ API to do just this.

Is there some reason why this subset can't live along full mutex/etc. stuff in the same library?

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Wed, 15 Oct 2008 08:53:13 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Tue, 14 October 2008 10:27Is there some reason why this subset can't live along full mutex/etc. stuff in the same library?
The only reason I can think of is lure of mixing two approaches in one application. This would lead to many problems, just like mixing procedural (C) and object oriented (C++) approaches in one program.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Wed, 15 Oct 2008 09:59:33 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 15 October 2008 04:53mr_ped wrote on Tue, 14 October 2008 10:27Is there some reason why this subset can't live along full mutex/etc. stuff in the same library?
The only reason I can think of is lure of mixing two approaches in one application. This would lead to many problems, just like mixing procedural (C) and object oriented (C++) approaches in one program.

Well, but I am afraid that you cannot take away primitives from the library just like you cannot take 'C' from 'C++' and retain its ability to deal with everything.

Anyway, the concept itself seems quite interesting. I think it is worth pursuing. What a pity I have already finished my current grand MT application (and done it the 'hard way'

BTW, do you think it would be possible to use this somehow instead of 'CoWork' in 'reference/CoWork' example? I have my doubts... IMO the performance would suffer.

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by zsolt on Wed, 15 Oct 2008 19:49:33 GMT
View Forum Message <> Reply to Message

In my current project we (the developers on the project) are using a a queuing framework.

Our experience is, that it is very easy to work with threads in an environment like this. It is very easy for beginners also.

We use UnitTest++ very intensively, and this aproach makes it very easy to create testable classes.

In our implementation, we don't use the old message id based design (used mostly in old embedded systems), as it needs a lot of administration. We use very hard template code. The logic of using the framework is very similar to the Callback system of U++.
We have special Callback-like template classes, but we use it with reversed logic (described a little bit later).

We have Tasks which are basically Thread classes with their own FIFOs. They are waiting for their FIFOs. The incoming items in the FIFO are CallbackXMethodAction like objects.

The logic is reversed if you compare it to U++'s Callbacks, as these Callback like objects are connected mostly to a method of the the same object they are in.

So this Callback like objects are callable from any threads. They are creating a CallbackXMethodAction like object on call and put it their Task's FIFO.
The Task reads this CallbackXMethodAction like object on its own thread and executes the method.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Wed, 15 Oct 2008 20:53:16 GMT
View Forum Message <> Reply to Message

I think QueueThread class should support classic U++ Callbacks. Because from the hierarchical point of view thread is a simple worker that should know nothing about executing code specifics. Also I think that adding event should use pick behaviour heavily, so adding new event (with it`s data) we be very "cheap" operation.

I do use this approach for some time in undustrial automation projects and it proved to be stable and predictive. Recently I`ve posted archive with queued thread class (ConveyorThread). Archive also contains RS232Thread class derived from ConveyorThread with documentation.
More I think of it, more I want to switch from classic synchronization objects to queued threads. Using brain , templates, picking and U++ containers should make this as effective as simple Mutexes.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by zsolt on Thu, 16 Oct 2008 09:38:04 GMT
View Forum Message <> Reply to Message

Yes, I think your ConveyorThread would be a very good starting point of a good MT framework.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Thu, 16 Oct 2008 11:48:55 GMT
View Forum Message <> Reply to Message

For some time I was thinking about Mirek`s question on CoWork. And couldn`t find how to apply queue approach to Reference/CoWork example natively. The only way I think of is QueueCoWork as pool manager for QueueThread objects which decides which Thread should execute next action depending on their business (i.e. queue lengths). On highest hierarchy level QueueCoWork is received PaintLines message from main thread`s Paint. This event executes main class callback function DoRepaint which manages pooling of low-level callbacks painting the lines. IMO, looks quite ugly:

```
void QueueCoWork::Manage(Callback1 &cb)
{
   int mostFreeThread = -1;
   //find most free (unbusy) Thread
```

```
    queueThreads[mostFreeThread] << cb;
}

void QueueCoWork::ManageTypical(Callback1 &cb)
{
    //simply rotate through threads to average tasks count
    lastUsedThread = (++lastUsedThread) % queueThreads.GetCount();

    queueThreads[lastUsedThread] << cb;
}

//--------------------------------------------------
void MainWindow::OnPaint
{
    queueCoWork << THISBACK(PaintLines);
}

void MainWindow::PaintLines()
{
    for (int y=0; y<height; ++y)
        queueCoWork.ManageTypical(THISBACK1(PaintLine, y));
}
void MainWindow::PaintLine(int y)
{
    //paint the y-th line
}
```

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Thu, 16 Oct 2008 13:04:06 GMT

View Forum Message <> Reply to Message

Mindtraveller wrote on Thu, 16 October 2008 07:48For some time I was thinking about Mirek`s question on CoWork. And couldn`t find how to apply queue approach to Reference/CoWork example natively. The only way I think of is QueueCoWork as pool manager for QueueThread objects which decides which Thread should execute next action depending on their business (i.e. queue lengths). On highest hierarchy level QueueCoWork is received PaintLines message from main thread`s Paint. This event executes main class callback function DoRepaint which manages pooling of low-level callbacks painting the lines. IMO, looks quite ugly:

```
void QueueCoWork::Manage(Callback1 &cb)
{
    int mostFreeThread = -1;
    //find most free (unbusy) Thread

    queueThreads[mostFreeThread] << cb;
}
```

```
void QueueCoWork::ManageTypical(Callback1 &cb)
{
   //simply rotate through threads to average tasks count
   lastUsedThread = (++lastUsedThread) % queueThreads.GetCount();

   queueThreads[lastUsedThread] << cb;
}

//-------------------------------------------------
void MainWindow::OnPaint
{
   queueCoWork << THISBACK(PaintLines);
}

void MainWindow::PaintLines()
{
   for (int y=0; y<height; ++y)
      queueCoWork.ManageTypical(THISBACK1(PaintLine, y));
}
void MainWindow::PaintLine(int y)
{
   //paint the y-th line
}
```

IMO even worse, "PaintLine" equivalent in more complex example might need Mutex.... It is not needed in the example, because algorithm never thouches the same data. But more general case might need to work with some shared data...

Actually, I believe that working with shared data, while inherently bug-prone, is where you can gain some performance using MT...

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Fri, 14 Nov 2008 09:09:00 GMT
View Forum Message <> Reply to Message

OK, I`d like to inform on some progress on "alternative" multithreading. It looks like I finally managed to figure how to apply it to U++ correctly (even in previous example). I`ll describe it as soon as class is ready and tested.

For now I`ve made little investigation on callbacks efficiency as I`d like to use them in my implementation. Some simple test was made:#include <Core/Core.h>
#include <math.h>

```
using namespace Upp;

class TestClass
{
public:
 void func(int a, int b)
 {
  int c = a + b*b;
  double xx = sin(a+b*1.34-3.14159252596428);
  double yy = cos((double) (b-a));
  x += ceil(xx*yy*(c-(b << 4)+(a*b)));
 }

private:
 int x;
};

CONSOLE_APP_MAIN
{
 TestClass tc;

 Callback2<int,int> cb = callback(&tc, &TestClass::func);
 dword cnt = 0;
 for (int j=0; j<50; ++j)
 {
  Cout() << FormatInt(j) << " ";
  dword t1 = GetTickCount();
  for (int i=0; i<2000000; ++i)
  {
   cb(100,500);
   //tc.func(100, 500);
  }
  cnt += GetTickCount()-t1;

  Sleep(1500);
 }

 Cout() << "\n" << FormatInt(cnt/j);
}
```

Averaged values were calculated for three cases:
1) Plain call
tc.func(100, 500); inside loop

2) Callback call
cb(100,500); inside loop

3) Callback stack variable creation+call+destruction
Callback2<int,int> cb = callback(&tc, &TestClass::func);
cb(100,500);
inside loop

On my PC I have these values:
1) 594
2) 592
3) 728

Conclusion: U++ callbacks are very efficient. Even for simple functions it takes very small tradeoff to use callbacks instead of plain member function calls. Creating and destroying of callback stack variable takes some time. Not very big though but it is better to avoid it.

Investigation continues.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Fri, 14 Nov 2008 09:15:58 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Fri, 14 November 2008 04:09
Creating and destroying of callback stack variable takes some time. Not very big though but it is better to avoid it.

Investigation continues.

It involves MT synchronization primitive (atomic counter inc/dec) - that is the cause of slowdown.

Mirek

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.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Mon, 17 Nov 2008 23:53:05 GMT

View Forum Message <> Reply to Message

Finally I finished new version of "alternative" threading class based on approach I described above. It should be times faster than classic callback queue. The idea of class is making asynchronous messaging between threads as close to plain function call as possible.

To describe how much overhead does it make to use asynchronous messaging with ACallbackThread class, I`ve made series of tests and generated approximate plot of overhead versus a "density" of events. Please keep in mind that tests where executed under a little prehistoric CPU AMD 2GHz, single core, DDR RAM.
...

At the moment I don`t know if it is good or bad results. But as for first glance, I would say that IMO most applications will generate no more than 200 events per second, so it will make overhead under 2% even for old CPUs. Maybe it all is finally worth efforts of not only making async queue machanizm but optimizing callbacks queue itself.

File Attachments

1) acb_jobs.jpg, downloaded 681 times

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?

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Tue, 18 Nov 2008 19:05:59 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Mon, 17 November 2008 18:53
But as for first glance, I would say that IMO most applications will generate no more than 200 events per second, so it will make overhead under 2% even for old CPUs.

Some perspective:

My MT server (the application I have developed this year, it is the backend for community website) is able to handle 10000 requests per second (on quadcore CPU). Every such request would generate tens or even hundreds 'events' in your model.

Also, I believe that all event queues will have to be synchronized anyway, something that your current measurements do not account for....

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Wed, 19 Nov 2008 00:45:19 GMT
View Forum Message <> Reply to Message

Thank you for your attention to this topic.

* You don't do anything in case two callback functions have same hash.
--- Yes, and this is the problem for now. I`d like to use HashBase as hash table (it would solve this problem) but it is not documented and is rather a black box for me when I compare my plain realization and HashBase class efficiency.

* 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?)
--- Certainly, because I posted something like petrified version of the original code, just to discuss

an idea itself.

* P1 and P2 must be moveable, right?
--- I keep thinking about this requirement. This is crucial for overall efficiency, but non-movable parameters should work also< I suppose...

* Also, I believe that all event queues will have to be synchronized anyway, something that your current measurements do not account for....
--- I mentioned before that actual tests were made for working class, not a simple & rather abstract example I`ve shown.

* My MT server (the application I have developed this year, it is the backend for community website) is able to handle 10000 requests per second (on quadcore CPU). Every such request would generate tens or even hundreds 'events' in your model.
--- I believe it is time to make some comparison. May be we should choose some reference example to be rewritten using alt. threading. It will give some comparisons between classic and alternative approach efficiency. Also, may be CoWork reference example is not the best choice to compare efficiency here? IMO it contains some functionality not covered by Thread alternative (if alt. approach will be proved to be worthy, I will make sense to make CoWork alternative, but not earlier). May be we should choose some comparison example wich is closer to "pure" threading? Or may be I`m wrong and CoWork reference example is just what we need to start comparison with? What do you think?

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Wed, 19 Nov 2008 09:39:47 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Tue, 18 November 2008 19:45
May be we should choose some comparison example wich is closer to "pure" threading?

What about simple "shared cache"?

I am not sure how it would look in your model, but 'normal' should be simple:

```
StaticMutex              mutex;
VectorMap<String, String> data;

void SetData(const String& key, const String& value)
{
   INTERLOCKED(mutex)
         data.GetAdd(key) = value;
}

String GetData(const String& key)
{
```

```
    INTERLOCKED(mutex)
        return data.Get(key, Null);
}
```

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Wed, 19 Nov 2008 12:20:37 GMT
View Forum Message <> Reply to Message

luzr wrote on Wed, 19 November 2008 12:39

```
StaticMutex              mutex;
VectorMap<String, String> data;

void SetData(const String& key, const String& value)
{
    INTERLOCKED(mutex)
        data.GetAdd(key) = value;
}

String GetData(const String& key)
{
    INTERLOCKED(mutex)
        return data.Get(key, Null);
}
```

What I see here is simply an atomic access to data container.
And it reveals big difference between these two approaches. It is like C and C++. C++ is based on "fully autonomous" objects with a limited number of public methods. Public methods are like high-level messages to the object which is intellectual enough to process them. To keep maintainability, everything should be hidden inside an object. There is still (limited) number of cases where object`s public method is just getting or setting variable. Contrary, Getter/Setter functions commonly are an evil, because usually it equals to making class member public.
...of course, you know it well.

Alternative approach introduces CallbackThread flavour for classes. It adds "multithreading" capability. You still use objects`s public methods, which are now called slightly differently and are processed in the background. From OO perspective you keep classes maintainable since neither public member variables nor Getters/Setters are introduced.
This is contrary to classic MT which commonly uses "shared" variables paired with synchronization objects.
So you can`t compare plain Getter/Setter because well designed objects mustn`t have ones (excluding a number of rare cases described above). You should compare real-world examples, where you don`t just set or get variable, but make some processing with it. In a real-world application you don`t just add to object`s container. Instead you add something to container and do some useful work with it. We have to consider alternative approach is not about handling

variables, but doing things.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Wed, 19 Nov 2008 18:38:29 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Wed, 19 November 2008 07:20You should compare real-world examples

http://en.wikipedia.org/wiki/Memcached

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Thu, 20 Nov 2008 09:19:52 GMT
View Forum Message <> Reply to Message

If I understand your example right...

```
class CachedSQL : public CallbackThread
{
public:
 void GetDataGiveToAnswerer(String request)
 {
  int rqIndex = data.Find(request);
  bool needAdd = false;
  String answer;
  if (rqIndex < 0)
  {
   answer = FetchFromDatabase(request);
   needAdd = true;
  }
  else
   answer = data[rqIndex];

  theAnswersProcessor.AddTask(HttpAnswersProcessor::ProcessData, request, answer);

  if (needAdd)
   data.Add(request, answer);
 }
};

class HttpRequestsProcessor : public CallThread
{
public:
 void ProcessHttpRequest(String rq)
```

```
 {
  //some processing here

  //ok, you need users`s info
  theCachedSQL.AddTask(CachedSQL::GetDataGiveToAnswerer, user);

  //optional additional processing
 }
};

class HttpAnswersProcessor : public CallThread
{
public:
 void ProcessHttpAnswer(String userInfo)
 {
  //some processing and sending answer
 }
};

////////////////////////////////////////////////////////////////////////////
// Let`s imagine 2 users sent their requests simultaneously.
//
// Time offset:
// 0         1         2         3         4         5         6         7
// 01234567890123456890123456789012345678901234567890123456789012345678901234567890
//---------------------------------------------------------------------------
// Synchronous work
// |--request1--|--sql1--|--answer1--|--request2--|--sql2--|--answer2--|...
// user1 will wait 3.5
// user2 will wait 6.9
//---------------------------------------------------------------------------
// Asyncronous work; '%' symbol stands for queue event i/o/thread-awake
// |--request1--|
//            %|--sql1--| *cache-miss; adding to data AFTER answer answer sent to httpAnswerer*
//                   %|--answer1--|
//              |--request2--|
//                         %|--sql2--| *value cached, no need to add this time*
//                                  %|--answer2--|
// user1 will wait 3.2
// user2 will wait 4.8
//---------------------------------------------------------------------------
```

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Thu, 20 Nov 2008 09:42:16 GMT

Mindtraveller wrote on Thu, 20 November 2008 04:19If I understand your example right...

I guess you do not.

The goal is to create single-machine memcached server - or simple version of it, but multithreaded.

Anyway, I guess, it is still bad example. Frankly, it is really hard to come with good *simple* example....

If you have any idea, I might try as well...

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Mon, 26 Jan 2009 23:06:48 GMT

Finally I`ve finished some controller programming work and resumed developing of "alternative MT-approach" class.
Let`s summarize the idea. To avoid synchronization objects headache and possible synchronization issues it is proposed to make all the threads isolated from each other. It means that
* threads do not "see" each other`s variables (including global ones, the ones with atomic access and shared synchronization objects) - it fully conforms classic OOP approach.
* threads do not call other thread`s member functions directly
* the only possible threads interaction is made through sending some message into thread`s internal queue.

Generally this would make threads interactions far more predictable and debuggable.
After some thinking I`ve made a decision not to send messages but instead send requests to run thread public member function (callback). It was decided because "message" is actually a signal to do some job. So it is no need to create any artificial message identifiers while you directly request to execute these callbacks.

It looks like this:

```
class JobThread1 : public CallbackThread
{
public: //these members may be added into thread queue and must not be called directly
   void Job11();
   void Job12(const String &);
private: //all the realization details are private
   //...
```

```
};

class class JobThread2 : public CallbackThread
{
public: //these members may be added into thread queue and must not be called directly
   void Job21(const String &);
private: //all the realization details are private
   //...
};

CONSOLE_APP_MAIN
{
   JobThread1 jobs1;
   JobThread2 jobs2;

   for (int i=0; i<10; ++i)
   {
      jobs1.Add(&JobThread1::Job11);
      jobs1.Add(&JobThread1::Job12, FormatInt(i));
      jobs2.Add(&JobThread2::Job21, FormatIntHex(i));
   }
};
```

You may even treat main thread as the same alt-MT thread. To do this you may have CallbackQueue variable and request it`s tasks. These tasks are processed with CallbackQueue::DoTasks().

Below is ready-to-use class with a simple test code.
If you are interested you may download and test it, or even use it in your apps. I hope more advanced versions of class will come soon (a number of optimizations is yet to be made).

"Alternative" MT requires a bit of reengineering threads processing functions. To make alt-MT program, you should think differently. Now you can`t share variables, now you don`t have a number of routines which are called in unpredictable sequence.

Instead you have messaging queues which mustn`t hold big number of messages (it is possible though not recommended). This means that thread must be logically solid as much as possible. This would keep all "tiny" interactions inside one thread.

Of course there are situations where "classic" MT with sync objects fits better. Situations include threads exchange with a huge number of tiny messages (inc/dec some variable). So if you cannot divide threads interaction into a (relatively small) number of (relatively medium) jobs - use "classic" MT.

I mean if you use more than 100`000 of messages per second - just use sync objects instead. But now you at least may choose to think "new way" on reorganizing threads member functions to make program stable or continue interacting with shared global variables to make it possibly quicker and less memory consuming but harder to debug and potentially less stable.

File Attachments
1) TestEtude2.zip, downloaded 357 times

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Fri, 03 Jul 2009 08:23:23 GMT
View Forum Message <> Reply to Message

luzr wrote on Tue, 30 June 2009 00:15Well, I have some experiences now (did project based on queues, now planning to rewrite it to plain old locking) and I have something to say about the topic (IMO!):

Synchronization objects are simple to manage as compared to often complex race condition relations in queued systems.
What do mean exactly, could you please give a pair of examples why you switched back from queueing model? This is very important topic IMO.

I personally found them very comfortable and stable comparing to a tonns of mutexes.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Fri, 03 Jul 2009 16:49:56 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Fri, 03 July 2009 04:23luzr wrote on Tue, 30 June 2009 00:15Well, I have some experiences now (did project based on queues, now planning to rewrite it to plain old locking) and I have something to say about the topic (IMO!):

Synchronization objects are simple to manage as compared to often complex race condition relations in queued systems.
What do mean exactly, could you please give a pair of examples why you switched back from queueing model? This is very important topic IMO.

I personally found them very comfortable and stable comparing to a tonns of mutexes.

I would like to, but right now I seem to be unable to describe it right. The problem was that it was user driven application and there are problems basically with "queue lag".

Maybe that the heart of problem is (was) the fact that it worked in "post" mode (not "execute") - messages (callbacks) being posted and not waiting for completition. Too often I ended with wrong events in the queue...

BTW, without posting, your method is equivalent to one mutex per instance and locking for any method call...

Mirek

Subject: Re: Thoughts about alternative approach to multithreading
Posted by Mindtraveller on Sun, 05 Jul 2009 19:43:45 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 03 July 2009 20:49I would like to, but right now I seem to be unable to describe it right. The problem was that it was user driven application and there are problems basically with "queue lag".

Maybe that the heart of problem is (was) the fact that it worked in "post" mode (not "execute") - messages (callbacks) being posted and not waiting for completition. Too often I ended with wrong events in the queue...

BTW, without posting, your method is equivalent to one mutex per instance and locking for any method call...

Mirek

1. What do you mean by "queue lag"? In my case case calling asynchronouse callback (this means: copy arguments, awake thread, post arguments and start execution), is almost as quick as calling U++ callback (I`ve posted these results above).
Personally I denied ANY types of events because I consider them absolutely artifical. The only thing which is really needed is executing some callback. So I had no problems identifying any types of events. I will appreciate any example where this approach fails (of course avoiding boundaries mentioned).

To be more precise, I takes more than single mutex per thread as there`s a queue and it really needs a semaphore.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Sun, 05 Jul 2009 20:08:48 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Sun, 05 July 2009 15:43luzr wrote on Fri, 03 July 2009 20:49I would like to, but right now I seem to be unable to describe it right. The problem was that it was user driven application and there are problems basically with "queue lag".

Maybe that the heart of problem is (was) the fact that it worked in "post" mode (not "execute") - messages (callbacks) being posted and not waiting for completition. Too often I ended with wrong events in the queue...

BTW, without posting, your method is equivalent to one mutex per instance and locking for any method call...

Mirek

1. What do you mean by "queue lag"? In my case case calling asynchronouse callback (this means: copy arguments, awake thread, post arguments and start execution), is almost as quick

as calling U++ callback (I`ve posted these results above).

It is not about performance, but about race conditions.

Quote:
Personally I denied ANY types of events because I consider them absolutely artifical. The only thing which is really needed is executing some callback. So I had no problems identifying any types of events. I will appreciate any example where this approach fails (of course avoiding boundaries mentioned).

Queued callbacks and events are equivalent here.

Well, I will try to describe the example. The code was image viewer. GUI thread to manage gui and other threads to do background loading.

Background threads do loading of whole directories in advance. Now the problem was that they got events to load directory, started performing it and meanwhile user switches to another directory.

With shared access (with mutex), this is quite easily manageable (I mean, stopping loading and doing something else). With queues, not so much.

Also, the fact that I need to pass all info using events is not very handy. With new U++ GUI threading, it is much easier to read actual GUI status in the background thread and adjust GUI as needed.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Sun, 05 Jul 2009 20:10:55 GMT
View Forum Message <> Reply to Message

Quote:
To be more precise, I takes more than single mutex per thread as there`s a queue and it really needs a semaphore.

I meant that if only operation you do is the one that is waiting for the result, it the same as locking instance mutex, calling method, unlocking the mutex.

Of course, not waiting for the result needs queue and semaphore.

Mirek

Subject: Re: Thoughts about alternative approach to multithreading

Posted by Mindtraveller on Sat, 11 Jul 2009 14:06:19 GMT
View Forum Message <> Reply to Message

Example you proposed sets a kind of problem. I spent some days thinking about it (I really met kind of this problem while programmed last project) and came to conclusion that each CallbackQueue/CallbackThread class should be descendant of CallbackNotifier class. Where CallbackNotifier is simply a Map<K,T> and a Mutex to synchronize access from multiple threads. This will enable asynchronouse messaging while thred's current queue callback is being executed. This is of course something from "classic" approach but anyway hides synchronization mutex.

In your example you will i.e. have to check this Map for current directory once per second. And if it is so, remove directory from Map and exit callback.

Subject: Re: Thoughts about alternative approach to multithreading
Posted by mirek on Sun, 12 Jul 2009 06:37:08 GMT
View Forum Message <> Reply to Message

Mindtraveller wrote on Sat, 11 July 2009 10:06Example you proposed sets a kind of problem. I spent some days thinking about it (I really met kind of this problem while programmed last project) and came to conclusion that each CallbackQueue/CallbackThread class should be descendant of CallbackNotifier class. Where CallbackNotifier is simply a Map<K,T> and a Mutex to synchronize access from multiple threads. This will enable asynchronouse messaging while thred's current queue callback is being executed. This is of course something from "classic" approach but anyway hides synchronization mutex.

In your example you will i.e. have to check this Map for current directory once per second. And if it is so, remove directory from Map and exit callback.

I think this is very much what I have ended with. But it is not a simple and clean design by any definition; the hurdle with locking mutex(es) just seems a much less evil here.

Mirek