Subject: what about WorkQueue : public CoWork
Posted by kohait00 on Wed, 03 Feb 2010 06:26:08 GMT
View Forum Message <> Reply to Message

hi there..

recently got an idea..

there is type of work which should be done somewhere in a worker thread, but in given order. something like "run it somewhere, *not* in this thread, i dont care where, but keep the supplied order of work". to be sure that certain things dont get executed before other things, but they *do* execute somewhere in parallell to current thred. Does that make sense?

more or less the approach of PostCallback, but there is one decent thread running all the work sequentially. something like:

WorkQueue : public CoWork /*with only one thread*/

WorkQueue wq;

/*in any thread, main thread probably, offload work */

wq & THISBACK(A)
wq & THISBACK(B)
/*any other work*/
wq & THISBACK(C)

/*the single worker thread */
DoJob(A)
DoJob(B)
/*maybe done, sleeping */
DoJob(C)
...

hope you got the point. it's almost like CoWork, but suppying to CoWork one needs to have real paralellable work, one cant be sure of what is going to be executed first, what later.. depends..or at least it is really executed in parallel. (or am i wrong)

so deriving from CoWork would almost do it, if ensuring that only 1 thread is the worker thread there. (I know that CoWork uses a static Pool of threads. that needed to be tweaked)

benefit: one could set up a decent count of worker threads, that could be responsible for different type of work, that can be executed in parallell, but the work that is "dependant" would still execute sequentially. this would make life esier in some cases, not needing to provide a lot of IPC anymore. just by logical *pre* runtime separation of work.

comments please

Subject: Re: what about WorkQueue : public CoWork
Posted by koldo on Wed, 03 Feb 2010 07:22:56 GMT
View Forum Message <> Reply to Message

Hello kohait

I am almost interested in it. I use some classes (not enough polished to be in bazaar) to handle many threads combined with the progress status of every thread.

However sometimes I found that to handle some of them serially is more efficient than doing it in parallel. For example in a disk browser to copy different files from same drive. Or to do rendering of some videos.

Subject: Re: what about WorkQueue : public CoWork
Posted by kohait00 on Wed, 03 Feb 2010 07:58:24 GMT
View Forum Message <> Reply to Message

well glad to hear about that..

i might o a work Queue implementations, to test that, i'll post it here.

i think the derive from CoWork will not work out.. its maybe esier to manage things using a single Thread in the WorkQueue class, than to start to tweak in CoWork, but i might use a bunch of code thereof.

Subject: first, CoWork.h and CoWork.cpp cleanups
Posted by kohait00 on Wed, 03 Feb 2010 11:07:14 GMT
View Forum Message <> Reply to Message

during study of CoWork class stuff, i stumbeled oacross some small grains..which is but of beaty reasons

1) static Pool::ThreadRun and static Pool::DoJob
  why should they be static, the Pool itself is generatet in CoWork environment, so for capsulation reasons they can run on their own class instance. thats also why p = pool() stuff has been removed, because now linked to class itself. Has that had a reason which I havent seen? (this would also make sense to one day, if WorkQueue should really rely on CoWork, be able to use same Pool class (but different instnce

2) some == and <= and the like stuff, implicit bool...has been explicited

3) while(todo)

```
    if(todo == 0)
      break;
```
makes no sense, it wont ever be true, since Finish() holds lock and no other thread can decrement it meanwhile

these are basicly changes, that made sense for me..commit if it makes sense..

next post is the WorkQueue

File Attachments
1) CoWork.h, downloaded 387 times
2) CoWork.cpp, downloaded 633 times

Subject: WorkQueue
Posted by kohait00 on Wed, 03 Feb 2010 11:14:58 GMT
View Forum Message <> Reply to Message

here comes the WorkQueue

which is not of great deal, it is a copy and paste of the CoWork class stuff (basicly the version postet above).

it's Pool is set to 1 Thread per instance only and thats about all of the changes

this reveals, that CoWork is pretty much a workqueue, if one leaves only 1 Thread.. but the current WorkQueue uses 1 Thread per instance, not a Thread pool

this one is still to be debugged. The test environment reveals a deadlock during Close problem and its solution as the WorkQueue runs stuff from the Gui in GuiLock environment...a common problem even in CoWork, if one wants to use CoWork doing GUI stuff (which makes not much sense, because the GUI thread would sleep meanwhile

if this idea per se makes sense one should think about modifying/flexibilizing CoWork::Pool, to be able to use CoWork as Base Class for WorkQueue.

File Attachments
1) WorkQueue.rar, downloaded 600 times

Subject: Re: what about WorkQueue : public CoWork
Posted by mirek on Wed, 03 Feb 2010 11:36:08 GMT
View Forum Message <> Reply to Message

[quote title=kohait00 wrote on Wed, 03 February 2010 01:26]

```
wq & THISBACK(A)
wq & THISBACK(B)
/*any other work*/
wq & THISBACK(C)
```

I believe that

```
{ CoWork a;
  a & parallel_job_A;
  do_something_while_doing_A;
}
{ CoWork b;
  b & parallel_job_B;
  do_something_while_doing_B;
}
```

has the same effect, if I understand you right...

Mirek

Subject: Re: what about WorkQueue : public CoWork
Posted by kohait00 on Wed, 03 Feb 2010 12:31:01 GMT
View Forum Message <> Reply to Message

hi mirek,

you're almost right but it's not quite the same i think.

this way (your post) you actually have some kind of "synchronisation" barriers (scope close) till when you expect the offload work to be done. -> which is an excelent way of building some "inline parallellism" until a certain sync point, actually a great idea the more i think about..this should be posted somewhere in a CoWork docu..

while in the WorkQueue issue you post the work and even more work, *without* the need to have sync points, you dont wait for the work completion any more, all if it is independant of your further execution from the point of posting. it works most in the Amarican war manner: "Fire and Forget" it wiil execute somewhere, not here, but in fired sequential order.

the use case is maybe this one:

you have a GUI (with its main thread) doing all painting and controling, another (WorkQueue)Thread is doing the work. but in most cases you still need to do some kind of response to GUI while working, and the gui needs to remain responsive. one could imagine to

start a Progress, using its ref in the posted Work to do its redaw/refresh or at least a PostCallback for that.

i dont know if it's clear or possible.. but is a quite nessesary case..when apps start to grow bigger, you cant rely on main thread alone anymore..the typical GUI app design pattern is needed.. control/woker threads, where one starts to use complecated and scary constructs with message queues and the like to inform the threads of state of each others "work".

this is basicly the idea behind that all.

Subject: changed CoWork and derived WorkQueue
Posted by kohait00 on Wed, 03 Feb 2010 14:03:06 GMT
View Forum Message <> Reply to Message

hi there, here comes a changed CoWork, which now basicly is able to specify wheather to use common/static threadpool or own thread pool (is beeing decided in CoWork() based on threadnr < 0 -> common pool)

the WorkQueue then is a simple derived class, which specifies 1 for threadnr and thus uses own thread.

this is subject to test .. if anyone besides me would need that.

regarding wild CoWork ctor init list:
it might be that the init order could differ, so the One<Pool> _opool could be not initialized before setting up pool.
this could be changed in a way that pool() -> pool(int threadnr = -1, CoWork * _this) returns either the global pool or costructs One<Pool>, which now is done in CoWork init list

comments welcome

File Attachments
1) CoWork.h, downloaded 406 times
2) CoWork.cpp, downloaded 797 times

Subject: Re: changed CoWork and derived WorkQueue
Posted by mirek on Wed, 03 Feb 2010 14:08:59 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Wed, 03 February 2010 09:03hi there, here comes a changed CoWork, which now basicly is able to specify wheather to use common/static threadpool or own thread pool (is beeing decided in CoWork() based on threadnr < 0 -> common pool)

the WorkQueue then is a simple derived class, which specifies 1 for threadnr and thus uses own thread.

this is subject to test .. if anyone besides me would need that.

regarding wild CoWork ctor init list:
it might be that the init order could differ, so the One<Pool> _opool could be not initialized before setting up pool.
this could be changed in a way that pool() -> pool(int threadnr = -1, CoWork * _this) returns either the global pool or costructs One<Pool>, which now is done in CoWork init list

comments welcome

Why should not WorkQueue use the same static thread pool?

Subject: Re: changed CoWork and derived WorkQueue
Posted by kohait00 on Wed, 03 Feb 2010 14:14:44 GMT
View Forum Message <> Reply to Message

thus an app is able to specify maybe "Workgroups". say following case:

task A, B, C each depends on each other, cause they might use same ressources, so best execute them in one thread somewhere else else than main thread, task C, D, E, are again each dependant on each other but *NOT* dependant on A, B, C, so they get their own WorkQueue (and Thread), and so are able to be executed sequentially related to each other, but parallelly related to other independant groups.. sounds complicated, right? hope it still makes sense

using the same static pool would mean using a bunch of threads, so supplied work to WorkQueue again gets executed parallely, but not sequentially (=>queue). in that case one could stick to CoWork usage, there is no difference then

---

Subject: Re: changed CoWork and derived WorkQueue
Posted by mirek on Wed, 03 Feb 2010 19:45:05 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Wed, 03 February 2010 09:14thus an app is able to specify maybe "Workgroups". say following case:

task A, B, C each depends on each other, cause they might use same ressources, so best execute them in one thread somewhere else else than main thread, task C, D, E, are again each dependant on each other but *NOT* dependant on A, B, C, so they get their own WorkQueue (and Thread), and so are able to be executed sequentially related to each other, but parallelly related to other independant groups.. sounds complicated, right? hope it still makes sense

using the same static pool would mean using a bunch of threads, so supplied work to WorkQueue again gets executed parallely, but not sequentially (=>queue). in that case one could stick to CoWork usage, there is no difference then

Note that the static pool only limits the number of threads running in parellel at any single time - which is a good thing, as you also have limited number of cores... I would say it has little to do with dependency of tasks.

Subject: Re: changed CoWork and derived WorkQueue
Posted by kohait00 on Wed, 03 Feb 2010 20:58:53 GMT
View Forum Message <> Reply to Message

hi mirek,

i think i got your point. the only thing to ensure about a work queue instance (or a given CoWork instance) is, the instance has to ensure that no other task is dequeued (to be executed in another parallel thread of static pool) before the previous task posted to the queue has been completed.

semantically speaking: the taks posted/enqueued into a CoWork instance have per se nothing to do with the pool itself, which presents the ability to execute arbitrary things anyway.

so probably the following would be right:

to make the struct Pool not private part of CoWork, but maybe protected. so a derived WorkQueue can reuse the static pool,
it can derive protected from CoWork to hide away the public interface (for not using virtuals), since most part of CoWork is proteced anyway, the WorkQueue then basicly needs to provide the same interface (Do() and operator &()), which then simply invoke a Finish() before handing over to CoWork::Do(). one could also use a CoWork directly, before enqueing any further task, simply ensure the one before is finished...so this is almost what you provided some post before..with the scoped CoWork approach..the only difference is that the CoWork is not created all over again and again, but used the same instance.

but one draw back is there: invoking Finish before enqueuing new task implies that the posting thread is halted until a previous task has terminated..this is not the case with current WorkQueue, which always dequeus, but to 1 thread only, so no other task can run in the queue. but if meanwhile another task has been queued, it is then executed right after, without having the posting task to wait for completion of previous one.

maybe the WorkQueue should be a class of it self, dequeuing differently then the CoWork does..

i'll try that.. but generally speaking, does all that make any sense to you anyway?

thanks for the patience
kostah

Subject: 2 basic problems
Posted by kohait00 on Mon, 08 Feb 2010 21:29:17 GMT

View Forum Message <> Reply to Message

hi mirek,

i am still thinking about the WorkQueue.. there is 1 basic problem to face:

* how to ensure that the common thread pool does not execute more than X jobs from a specific CoWork instace in parallel. since the pool does not care about sequencing relative to a CoWork instance. (restrict the global pool itself is no option, this would affect other CoWorks with proper different constraints)

the threads serve themselves from a global job queue. if we i.e say, only X=1 task from all the tasks commited to a certain CoWork may execute at once, other threads would still dequeue and execute jobs, since they dont care.

any idea? to make a per CoWork queue, which then really submits to the global queue?

i mean the following:
task A1 is executing in a thread from pool. task A2 (related to A1, may not intersect it) is also posted there, but may not execute, before A1 is done, meanwhile any other thread from pool finishes, dequeues A2 task, checks and realizes that A1 is still running, and it may not execute it. what to do with the job? the thread may not wait, it would block work of other CoWorks, post the job back to queue is not reliable, other related tasks might have been posted meanwhile. results in desorder.

seems as here we come close to a scheduler

Subject: Re: changed CoWork and derived WorkQueue
Posted by mirek on Tue, 09 Feb 2010 09:56:44 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Wed, 03 February 2010 15:58
i'll try that.. but generally speaking, does all that make any sense to you anyway?

Frankly, no I am a little bit lost, as I cannot imagine the real world situation where this would help... (too much experiences with MT maybe?

Mirek

Subject: Re: changed CoWork and derived WorkQueue
Posted by kohait00 on Tue, 09 Feb 2010 14:14:22 GMT
View Forum Message <> Reply to Message

well, maybe it's because *I* have too little MT experience, i am a bit of a rookie on that indeed, still learning.

my situation was basicly that i wanted to keep the gui responsive (old stuff) offloading work (i realize that CoWork is mainly a loop parallelizer, for independant work). but as soon as the offloaded work is kind of depandant on each other or the order of execution, you only have the possibility to either PostCallback the work using the GUI thread (which for large work affects GUI respnsiiveness) or to use a 1-thread-workqueue, which executes in background in order of comits. both solutions *are* a WorkQueue, even PostCallback. With current CoWork this is *not* possible. for stated reason: CoWork is "only" a loop parallelizer..not a work scheduler. with the given changes, one could flexibilize that.

anyhow..i wont push that, dont feel bothered. you decide. i am working with the modified CoWork, and it seems to me relatively more fluent to post to WorkQueue (1 trhead CoWork) than to a threadpool CoWork.

Subject: Re: changed CoWork and derived WorkQueue
Posted by mirek on Mon, 15 Feb 2010 11:49:40 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Tue, 09 February 2010 09:14well, maybe it's because *I* have too little MT experience, i am a bit of a rookie on that indeed, still learning.

my situation was basicly that i wanted to keep the gui responsive (old stuff) offloading work (i realize that CoWork is mainly a loop parallelizer, for independant work). but as soon as the offloaded work is kind of depandant on each other or the order of execution

Perhaps I am still not getting where these dependant units of work come from.

I clearly see - in GUI, you start something in another thread. That job runs in parallel, possibly changing the GUI (some functions not available until it finishes etc..). The it finishes and somehow tells this to GUI (PostCallbakc perhaps).

Where are these units?

Subject: Re: changed CoWork and derived WorkQueue
Posted by kohait00 on Tue, 16 Feb 2010 13:44:48 GMT
View Forum Message <> Reply to Message

i hope i can provide you a test case soon, similar to the cowork stuff, where you can see the

difference. there the benefits/drawbacks will be seen clearer i think.

Subject: BUGFIX to WorkQueue
Posted by kohait00 on Thu, 18 Feb 2010 15:02:19 GMT
View Forum Message <> Reply to Message

in case anyone working with the WorkQueue, it does not what it should. previous version relied on CoWork, which uses Vector<MJob> as job storage, and due to the presumption that jobs independant, the execution order didnt matter. so CoWork uses Vector::Pop to dequeue jobs, which dequeues the latest inserted, not the first inserted.

this version here i have switched the former WorkQueue implementation to use Link<MJob>. so the order is preserved

@mirek: this is still not the example i wanted to provide

File Attachments
1) WorkQueueTest2.rar, downloaded 365 times

Subject: Re: BUGFIX to WorkQueue
Posted by kohait00 on Fri, 19 Feb 2010 07:47:38 GMT
View Forum Message <> Reply to Message

next bug:

need to Unlink() before LinkAfter(), to keep the queue consistent

WorkQueue::MJob & WorkQueue::Pool::NextFree()

```
  //found a free one
  e->Unlink(); // <<<=== INSERT this one...
  e->LinkAfter(lastjob); //list
```
