Subject: bug in CoWork since C++11
Posted by crydev on Fri, 13 May 2016 04:53:49 GMT
View Forum Message <> Reply to Message

Hello,

I just installed the nightly build 9818 and recompiled my application. Some changes have been made to the CoWork class and to the Atomic object. However, CoWork doesn't work properly anymore for me. I have a situation with CPU_Cores() worker threads, which increment an Atomic value and which are started by a CoWork object. This used to work great. However, the CoWork now seems to stop executing these workers even though 'todo' is greater than 0. Is there a bug?

Thanks a lot!

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Sat, 21 May 2016 17:43:45 GMT
View Forum Message <> Reply to Message

Sorry to hear that. It is absolutely possible that I have broke something there (although I have a couple of nightly tests running now on CoWork).

Would it be eventually possible to show me some simple testcase? What I am testing here works....

Subject: Re: bug in CoWork since C++11
Posted by crydev on Sat, 02 Jul 2016 15:49:05 GMT
View Forum Message <> Reply to Message

I have changed my synchronization method to something more safe, leaving the workers not doing any synchronization work at all. I changed it because I thought my synchronization was faulty. However, it does not appear to be faulty, because the bug remains, even in build 9994.

I start my workers like this, where this->mWorkerFileOrder is a Vector containing 8 structures that describe the work the worker should do.

```
// Launch the workers.
for (auto& work : this->mWorkerFileOrder)
{
 threadPool & THISBACK2(FirstScanWorker<T>, &work,
((T)(reinterpret_cast<ScanParameters<T>*>(GlobalScanParameter))->ScanValue));
}
```

The FirstScanWorker function looks like this; the function does not write to memory that does not belong to the worker it represents. Every worker has a thread-local variable called FinishedWork, which is set to true when the worker completes. Another program component peeks this variable for every worker, and finalizes when all workers are done.

```
// Represents the default template worker function for the set of workers including specialized ones.
// This set of workers run the first scan sequence.
template <class T>
void MemoryScanner::FirstScanWorker(WorkerRegionParameterData* const regionData, const T& value)
{
#ifdef _DEBUG
 LARGE_INTEGER frequency;
 LARGE_INTEGER t1;
 LARGE_INTEGER t2;

 // Get the amount of ticks per second.
 QueryPerformanceFrequency(&frequency);

 // Start the timer.
 QueryPerformanceCounter(&t1);
#endif

 // ------------------------------------------------

 FileOut addressesFile(AppendFileName(mMemoryScanner->GetTempFolderPath(), Format("Addresses%i.temp", regionData->WorkerIdentifier)));
 FileOut valFile(AppendFileName(mMemoryScanner->GetTempFolderPath(), Format("Values%i.temp", regionData->WorkerIdentifier)));

 int fastScanAlignSize = GlobalScanParameter->CurrentScanFastScan ? sizeof(T) : 1;
 if (fastScanAlignSize == sizeof(__int64))
 {
  fastScanAlignSize = sizeof(int);
 }

 unsigned int fileIndex = 0;
 const unsigned int forLoopLength = regionData->OriginalStartIndex + regionData->Length;

 for (unsigned int i = regionData->OriginalStartIndex; i < forLoopLength; ++i)
 {
  unsigned int arrayIndex = 0;
  unsigned int currentArrayLength = 256;

  MemoryRegion& currentRegion = this->memRegions[i];
```

```
  const SIZE_T regionSize = currentRegion.MemorySize;
  currentRegion.FileDataIndexes.StartIndex = fileIndex;

  SIZE_T* localAddresses = NULL;
  T* localValues = NULL;

  Byte* buffer = new Byte[currentRegion.MemorySize];
  if (CrySearchRoutines.CryReadMemoryRoutine(this->mOpenedProcessHandle,
(void*)currentRegion.BaseAddress, buffer, currentRegion.MemorySize, NULL))
  {
   for (SIZE_T i = 0; i < regionSize; i += fastScanAlignSize)
   {
    const T* tempStore = (T*)&(buffer[i]);

    if ((*reinterpret_cast<ValueComparator<T>*>(this->mCompareValues))(*tempStore, value))
    {
     if (!localAddresses || !localValues)
     {
      localAddresses = new SIZE_T[currentArrayLength];
      localValues = new T[currentArrayLength];
     }

     if (arrayIndex >= currentArrayLength)
     {
      const unsigned int oldCurrentArrayLength = currentArrayLength;
      this->ReallocateMemoryScannerBufferCounter(&currentArrayLength);

      SIZE_T* newAddressesArray = new SIZE_T[currentArrayLength];
      memcpy(newAddressesArray, localAddresses, oldCurrentArrayLength * sizeof(SIZE_T));
      delete[] localAddresses;
      localAddresses = newAddressesArray;

      T* newValuesArray = new T[currentArrayLength];
      memcpy(newValuesArray, localValues, oldCurrentArrayLength * sizeof(T));
      delete[] localValues;
      localValues = newValuesArray;
     }

     localAddresses[arrayIndex] = currentRegion.BaseAddress + i;
     localValues[arrayIndex++] = *tempStore;

     ++fileIndex;
    }
   }
  }

  delete[] buffer;
```

```
  if (arrayIndex > 0)
  {
   this->mScanResultCount += arrayIndex;

   if (CachedAddresses.GetCount() < MEMORYSCANNER_CACHE_LIMIT)
   {
    AddResultsToCache(arrayIndex, localAddresses, NULL);
   }

   addressesFile.Put(localAddresses, arrayIndex * sizeof(SIZE_T));
   delete[] localAddresses;

   valFile.Put(localValues, arrayIndex * sizeof(T));
   delete[] localValues;
  }
  else
  {
   if (localAddresses)
   {
    delete[] localAddresses;
    delete[] localValues;
   }
  }

  currentRegion.FileDataIndexes.ResultCount = arrayIndex;
  this->UpdateScanningProgress(AtomicInc(RegionFinishCount));
 }

 addressesFile.Close();
 valFile.Close();

 // Indicate that this worker is done processing.
 regionData->FinishedWork = true;

 // ------------------------------------------------

#ifdef _DEBUG
 // Stop the timer.
 QueryPerformanceCounter(&t2);
 OutputDebugString(Format("Worker %i took %f ms\r\n", regionData->WorkerIdentifier,
(t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart));
#endif
}
```

The bug still remains; most of the time, not all workers finish, because CoWork thinks it doesn't have anything to do anymore. Sometimes it finishes, but rarely. :)

Thanks, and my apologies for the late response. I haven't had time. :(

crydev

Subject: Re: bug in CoWork since C++11
Posted by crydev on Mon, 11 Jul 2016 17:36:46 GMT
View Forum Message <> Reply to Message

I did some more debugging, and I found that it goes wrong in this function, in CoWork.cpp on line 196. It will wait infinitely until jobs are done, while there are still jobs to be done.

```
void CoWork::Finish() {
 if(!pool) return;
 Pool& p = *pool;
 p.lock.Enter();
 while(todo) {
  LLOG("Finish: todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");
  if(todo == 0)
   break;
  if(p.scheduled)
   Pool::DoJob();
  else {
   p.lock.Leave();
   LLOG("WaitForFinish (CoWork " << FormatIntHex(this) << ")");
   waitforfinish.Wait(); // <---- Infinite wait here!
   p.lock.Enter();
  }
 }
 p.lock.Leave();
 LLOG("CoWork " << FormatIntHex(this) << " finished");
}
```

It is a very annoying bug. :(

Thanks,

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Sat, 30 Jul 2016 16:36:12 GMT
View Forum Message <> Reply to Message

crydev wrote on Mon, 11 July 2016 19:36I did some more debugging, and I found that it goes

wrong in this function, in CoWork.cpp on line 196. It will wait infinitely until jobs are done, while there are still jobs to be done.

```
void CoWork::Finish() {
 if(!pool) return;
 Pool& p = *pool;
 p.lock.Enter();
 while(todo) {
  LLOG("Finish: todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");
  if(todo == 0)
   break;
  if(p.scheduled)
   Pool::DoJob();
  else {
   p.lock.Leave();
   LLOG("WaitForFinish (CoWork " << FormatIntHex(this) << ")");
   waitforfinish.Wait(); // <---- Infinite wait here!
   p.lock.Enter();
  }
 }
 p.lock.Leave();
 LLOG("CoWork " << FormatIntHex(this) << " finished");
}
```

It is a very annoying bug. :(

Thanks,

crydev

Sorry I have missed this important reply... Anyway, 2 things:

1. I have recently refactored CoWork to use ConditionVariable instead of Semaphore. There is a tiny chance that this alone will fix the issue.... (if there was a bug in CoWork).

2. You might try some logging... E.g. activate LLOG in CoWork

Is not it possible that some of thread is frozen because of some deadlock?

Mirek

Subject: Re: bug in CoWork since C++11
Posted by crydev on Wed, 03 Aug 2016 17:45:14 GMT
View Forum Message <> Reply to Message

Thanks for your reply Mirek,

I can't spot anything that would deadlock in my code. I enabled LLOG (I guess) by uncommenting the define LLOG at the top of CoWork.cpp. Is that correct? At first, I got a compiler error, because the PushJob function is static. Line 150 in CoWork.cpp gives a compiler error:

LLOG("Adding job " << p.scheduled - 1 << "; todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");

I went on commenting out this LLOG line and starting executing my code. The log I got was as following. I don't fully understand what the structuring of the log is, but it seems like it quits executing when #todo is still 2.

DoJobA 1, todo: 7 (CoWork 0105d194)
Finished, remaining todo 7
#4 Job finished
#4 Job acquired
Quit thread
CoWork thread #4 finished
DoJobA 1, todo: 6 (CoWork 0105d194)
Finished, remaining todo 6
#1 Job finished
#1 Job acquired
Quit thread
CoWork thread #1 finished
DoJobA 1, todo: 5 (CoWork 0105d194)
Finished, remaining todo 5
#2 Job finished
#2 Job acquired
Quit thread
CoWork thread #2 finished
DoJobA 1, todo: 4 (CoWork 0105d194)
Finished, remaining todo 4
#5 Job finished
#5 Job acquired
Quit thread
CoWork thread #5 finished
DoJobA 1, todo: 3 (CoWork 0105d194)
Finished, remaining todo 3
#6 Job finished
#6 Job acquired
Quit thread
CoWork thread #6 finished
DoJobA 1, todo: 2 (CoWork 0105d194)
Finished, remaining todo 2
#0 Job finished
#0 Job acquired

Quit thread
CoWork thread #0 finished
Quit ended

Thanks,

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Thu, 04 Aug 2016 19:58:31 GMT
View Forum Message <> Reply to Message

"Quit thread"

- this is interesting. It looks like you are quiting the thread that spawned jobs before it has the chance to finish.

Is not it possible that you are calling Finish (perhaps via ~CoWork) from other thread than the one that scheduled the work? Or in other words, have CoWork instance shared between threads?

It is true that between 'classic' and 'C++11' I have changed the logic so that each 'master thread' has its own pool of worker threads, to avoid work stealing. Perhaps it was not a good idea after all..

Subject: Re: bug in CoWork since C++11
Posted by crydev on Sun, 07 Aug 2016 09:50:16 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 04 August 2016 21:58"Quit thread"

- this is interesting. It looks like you are quiting the thread that spawned jobs before it has the chance to finish.

Is not it possible that you are calling Finish (perhaps via ~CoWork) from other thread than the one that scheduled the work? Or in other words, have CoWork instance shared between threads?

I had two instances of Finish in my code, and I was sharing the CoWork instance for different purposes. Is that not a good idea? It used to work fine.

mirek wrote on Thu, 04 August 2016 21:58
It is true that between 'classic' and 'C++11' I have changed the logic so that each 'master thread' has its own pool of worker threads, to avoid work stealing. Perhaps it was not a good idea after all..

Has the way CoWork should be used changed with the C++11 way of U++? I see a thread pool as an object that makes sure tasks you give it are dispatched into other threads, regardless of where the tasks come from. Does it work differently with CoWork? I also saw a CoWork changed commit this morning. Did anything important change?

Thanks

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Sun, 07 Aug 2016 18:36:31 GMT
View Forum Message <> Reply to Message

crydev wrote on Sun, 07 August 2016 11:50mirek wrote on Thu, 04 August 2016 21:58"Quit thread"

- this is interesting. It looks like you are quiting the thread that spawned jobs before it has the chance to finish.

Is not it possible that you are calling Finish (perhaps via ~CoWork) from other thread than the one that scheduled the work? Or in other words, have CoWork instance shared between threads?

I had two instances of Finish in my code, and I was sharing the CoWork instance for different purposes. Is that not a good idea? It used to work fine.

mirek wrote on Thu, 04 August 2016 21:58
It is true that between 'classic' and 'C++11' I have changed the logic so that each 'master thread' has its own pool of worker threads, to avoid work stealing. Perhaps it was not a good idea after all..

Has the way CoWork should be used changed with the C++11 way of U++? I see a thread pool as an object that makes sure tasks you give it are dispatched into other threads, regardless of where the tasks come from. Does it work differently with CoWork? I also saw a CoWork changed commit this morning. Did anything important change?

Thanks

crydev

Well, I rethought the whole thing... I guess new iteration will work just fine again.

It is not 100% finished yet, maybe wait till tomorrow.

Mirek

Subject: Re: bug in CoWork since C++11
Posted by mirek on Mon, 08 Aug 2016 07:32:12 GMT
View Forum Message <> Reply to Message

Refactoring finished, please try now...

Mirek

Subject: Re: bug in CoWork since C++11
Posted by crydev on Mon, 08 Aug 2016 17:30:00 GMT
View Forum Message <> Reply to Message

mirek wrote on Mon, 08 August 2016 09:32Refactoring finished, please try now...

Mirek

Thanks Mirek,

I just tested with the new build (10148). The situation has improved; it does not make infinite waits anymore. However, it still does not entirely work. It just ends when there are still 'todo' in the CoWork. I went ahead and tried LLOG; I attached the log below. What could be wrong?

#9 Waiting ended
#9 Job acquired
Quit thread
#1 Waiting ended
CoWork thread #9 finished
#1 Job acquired
Quit thread
CoWork thread #1 finished
#7 Waiting ended
#7 Job acquired
Quit thread
CoWork thread #7 finished
DoJobA 1, todo: 7 (CoWork 0125a194)
Finished, remaining todo 7
#6 Job finished
#6 Job acquired
Quit thread
CoWork thread #6 finished
DoJobA 1, todo: 6 (CoWork 0125a194)
Finished, remaining todo 6
#2 Job finished
#2 Job acquired
Quit thread
CoWork thread #2 finished
DoJobA 1, todo: 5 (CoWork 0125a194)

Finished, remaining todo 5
#5 Job finished
#5 Job acquired
Quit thread
CoWork thread #5 finished
DoJobA 1, todo: 4 (CoWork 0125a194)
Finished, remaining todo 4
#3 Job finished
#3 Job acquired
Quit thread
CoWork thread #3 finished
DoJobA 1, todo: 3 (CoWork 0125a194)
Finished, remaining todo 3
#4 Job finished
#4 Job acquired
Quit thread
CoWork thread #4 finished
DoJobA 1, todo: 2 (CoWork 0125a194)
Finished, remaining todo 2
#8 Job finished
#8 Job acquired
Quit thread
CoWork thread #8 finished
DoJobA 1, todo: 1 (CoWork 0125a194)
Finished, remaining todo 1
#0 Job finished
#0 Job acquired
Quit thread
CoWork thread #0 finished
Quit ended

Thanks!

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Tue, 09 Aug 2016 09:15:00 GMT
View Forum Message <> Reply to Message

Well, the one possible explanation is that your CoWork is global (or static) variable. Is that so?

Subject: Re: bug in CoWork since C++11
Posted by crydev on Tue, 09 Aug 2016 09:36:03 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 09 August 2016 11:15Well, the one possible explanation is that your CoWork is global (or static) variable. Is that so?

My CoWork instance is not a global variable. It is a variable as member of a class and it is not a pointer. It is a singleton class, though. I construct it with a private constructor and a GetInstance method.

```
class X
{
private:
    CoWork mThreadPool;
    X();
public:
    static X* GetInstance()
    {
        static X instance;
        return &instance;
    };
}
```

Subject: Re: bug in CoWork since C++11
Posted by mirek on Tue, 09 Aug 2016 09:43:51 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 09 August 2016 11:36mirek wrote on Tue, 09 August 2016 11:15Well, the one possible explanation is that your CoWork is global (or static) variable. Is that so?

My CoWork instance is not a global variable. It is a variable as member of a class and it is not a pointer. It is a singleton class, though. I construct it with a private constructor and a GetInstance method.

```
class X
{
private:
    CoWork mThreadPool;
    X();
public:
    static X* GetInstance()
    {
        static X instance;
        return &instance;
    };
}
```

I see. It is static then.

I believe what happens here is that first, CoWork instance is constructed. Then you schedule the work, which creates global static thread pool. On app exit, pool is therefore destructed BEFORE destructor of CoWork, which would normally performed the remaining work (by calling Finish).

Can you try to call GetInstance()->Finish() at the end of APP_MAIN to prove this hypothesis? (if true, I will then start thinking if resolving this situation is worth the trouble or rather mentioning in docs...

Mirek

Subject: Re: bug in CoWork since C++11
Posted by crydev on Thu, 18 Aug 2016 18:58:51 GMT
View Forum Message <> Reply to Message

mirek wrote on Tue, 09 August 2016 11:43crydev wrote on Tue, 09 August 2016 11:36mirek wrote on Tue, 09 August 2016 11:15Well, the one possible explanation is that your CoWork is global (or static) variable. Is that so?

My CoWork instance is not a global variable. It is a variable as member of a class and it is not a pointer. It is a singleton class, though. I construct it with a private constructor and a GetInstance method.

```
class X
{
private:
    CoWork mThreadPool;
    X();
public:
    static X* GetInstance()
    {
        static X instance;
        return &instance;
    };
}
```

I see. It is static then.

I believe what happens here is that first, CoWork instance is constructed. Then you schedule the work, which creates global static thread pool. On app exit, pool is therefore destructed BEFORE destructor of CoWork, which would normally performed the remaining work (by calling Finish).

Can you try to call GetInstance()->Finish() at the end of APP_MAIN to prove this hypothesis? (if true, I will then start thinking if resolving this situation is worth the trouble or rather mentioning in docs...

Mirek

Hello Mirek,

I'm having trouble with understanding exactly what you mean, because my app doesn't end when the work is completed. I tried to prove it the way you provided but I am not convinced by the proof. However, I tried a construction where the CoWork instance is not static. I changed 'mThreadPool' to a 'CoWork*' and manually instantiated and destructed the 'mThreadpool' object. However, this does not change the situation for me: it still waits while todo > 0.

While trying to prove your hypothesis, I created a situation where todo > 0 and the work did not complete properly. Then, I exited the application, having it call Finish() and noticed that it passes the todo > 0 check, but fails at the 'p.scheduled' check, resulting in an infinite wait.

```
void CoWork::Finish() {
 if(!pool) return;
 Pool& p = *pool;
 p.lock.Enter();
 while(todo) {
  LLOG("Finish: todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");
  if(todo == 0) // Doesn't break because todo > 0.
   break;
  if(p.scheduled) // Doesn't pass this check because p.scheduled is NOT true.
   Pool::DoJob(); // Therefore doesn't execute the jobs;
  else {
   LLOG("WaitForFinish (CoWork " << FormatIntHex(this) << ")");
   waitforfinish.Wait(p.lock); // Infinite wait here!
  }
 }
 p.lock.Leave();
 LLOG("CoWork " << FormatIntHex(this) << " finished");
}
```

What can I do?

Thanks,

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Tue, 23 Aug 2016 06:34:45 GMT
View Forum Message <> Reply to Message

crydev wrote on Thu, 18 August 2016 20:58mirek wrote on Tue, 09 August 2016 11:43crydev wrote on Tue, 09 August 2016 11:36mirek wrote on Tue, 09 August 2016 11:15Well, the one

possible explanation is that your CoWork is global (or static) variable. Is that so?

My CoWork instance is not a global variable. It is a variable as member of a class and it is not a pointer. It is a singleton class, though. I construct it with a private constructor and a GetInstance method.

```
class X
{
private:
    CoWork mThreadPool;
    X();
public:
    static X* GetInstance()
    {
        static X instance;
        return &instance;
    };
}
```

I see. It is static then.

I believe what happens here is that first, CoWork instance is constructed. Then you schedule the work, which creates global static thread pool. On app exit, pool is therefore destructed BEFORE destructor of CoWork, which would normally performed the remaining work (by calling Finish).

Can you try to call GetInstance()->Finish() at the end of APP_MAIN to prove this hypothesis? (if true, I will then start thinking if resolving this situation is worth the trouble or rather mentioning in docs...

Mirek

Hello Mirek,

I'm having trouble with understanding exactly what you mean, because my app doesn't end when the work is completed. I tried to prove it the way you provided but I am not convinced by the proof. However, I tried a construction where the CoWork instance is not static. I changed 'mThreadPool' to a 'CoWork*' and manually instantiated and destructed the 'mThreadpool' object. However, this does not change the situation for me: it still waits while todo > 0.

While trying to prove your hypothesis, I created a situation where todo > 0 and the work did not complete properly. Then, I exited the application, having it call Finish() and noticed that it passes the todo > 0 check, but fails at the 'p.scheduled' check, resulting in an infinite wait.

```
void CoWork::Finish() {
 if(!pool) return;
 Pool& p = *pool;
```

```
 p.lock.Enter();
 while(todo) {
  LLOG("Finish: todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");
  if(todo == 0) // Doesn't break because todo > 0.
   break;
  if(p.scheduled) // Doesn't pass this check because p.scheduled is NOT true.
   Pool::DoJob(); // Therefore doesn't execute the jobs;
  else {
   LLOG("WaitForFinish (CoWork " << FormatIntHex(this) << ")");
   waitforfinish.Wait(p.lock); // Infinite wait here!
  }
 }
 p.lock.Leave();
 LLOG("CoWork " << FormatIntHex(this) << " finished");
}
```

What can I do?

Thanks,

crydev

I believe you do not have latest trunk version, here is how Finish looks now:

```
void CoWork::Finish() {
 Pool& p = GetPool();
 p.lock.Enter();
 while(!jobs.IsEmpty(1)) {
  LLOG("Finish: todo: " << todo << " (CoWork " << FormatIntHex(this) << ")");
  p.DoJob(*jobs.GetNext(1));
 }
 while(todo) {
  LLOG("WaitForFinish (CoWork " << FormatIntHex(this) << ")");
  waitforfinish.Wait(p.lock);
 }
 p.lock.Leave();
 LLOG("CoWork " << FormatIntHex(this) << " finished");
}
```

Subject: Re: bug in CoWork since C++11
Posted by crydev on Tue, 23 Aug 2016 18:39:18 GMT
View Forum Message <> Reply to Message

Hello Mirek,

Thanks! It seems to work now I pulled 10176. Is it wise to share CoWork among other tasks, or is it designed to be dedicated to a specific task, meaning that I should create another CoWork for a different task? I have it this way now and it seems to work nicely.

Thanks a bunch for you help.

crydev

Subject: Re: bug in CoWork since C++11
Posted by mirek on Wed, 24 Aug 2016 19:08:41 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 23 August 2016 20:39Hello Mirek,

Thanks! It seems to work now I pulled 10176. Is it wise to share CoWork among other tasks, or is it designed to be dedicated to a specific task, meaning that I should create another CoWork for a different task? I have it this way now and it seems to work nicely.

Thanks a bunch for you help.

crydev

I guess it is a little bit pointless... The whole purpose of CoWork is that scheduled work is done after Finish... CoWork instance is lightweight, nothing is saved by having it shared as global object.

However, as there might be cases where more direct access to worker threads is required, there are now static CoWork::TrySchedule and CoWork::Schedule methods. Maybe they could be better in your case?

Mirek

Subject: Re: bug in CoWork since C++11
Posted by crydev on Thu, 25 Aug 2016 17:09:00 GMT
View Forum Message <> Reply to Message

Hello Mirek,

I understand; I will keep the seperate CoWork usage design then!

Thanks for your help. :)

crydev