ATL::CThreadPool

How to：

1. Coding a worker class: class Worker

Include 1 type definition & 3 functions:

// used as parameter type when queue task and execute.

// MUST be capable of being cast to and from a ULONG\_PTR

typedef USER\_TYPE RequestType;

// init when thread begin

BOOL Initialize(void\* pvWorkerParam);

// looped in thread for each task

void Execute(RequestType pvTask, void\* pvWorkerParam, OVERLAPPED\* pOverlapped);

// cleanup when thread end

void Terminate(void\* pvWorkerParam);

2. Customized task base class: class TaskBase (optional)

User customized. pvTask in Execute can be pointer to this class instance;

Suggest only one pure virtual function:

virtual void DoTask(void \*pvParam, OVERLAPPED \*pOverlapped) = 0;

3. Coding TaskBase derived class for each particular task:

class Task: public TaskBase

Parameter can be passed in task constructor, e.g:

Task( class Param param)

Procedure：

1. Construct & Init thread pool

CThreadPool<CWorker> pool;

pool.Initialize(void \*pvWorkerParam = NULL,

int nNumThreads = 0,

DWORD dwStackSize = 0,

HANDLE hCompletion = INVALID\_HANDLE\_VALUE);

2. Generate new task

class Param; // set param for task

pvTask = new class Task(Param); // new task

3. Queue task in pool

QueueRequest(Worker::RequestType pvTask);

4. Shutdown thread pool:

pool.Shutdown(DWORD dwMaxWaitMilliSeconds);

Note:

1. Release of Task object:

Release in Worker::Exectute, or in store in caller and release after not used (e.g Shutdown).

Problem:

CThreadPool use an I/O completion port to synchronize threads.

I/O completion packet is queued in first-in-first-out (FIFO) order to the associated I/O completion port.

CThreadPool::Shutdown queue an ATLS\_POOL\_SHUTDOWN completion pocket then wait a thread to receive and exit.

If there are too many tasks in completion queue, may be no thread picked up the ATLS\_POOL\_SHUTDOWN before CThreadPool::Shutdown wait timed out.

This can cause two problem:

1. CThreadPool::Shutdown will force terminate all alive thread, thread does not have a chance to do some cleanups.
2. If ATLS\_POOL\_SHUTDOWN timed out, CThreadPool::Shutdown terminate thread but will not remove it from m\_threadMap, this will cause an assertion failure.

Fix 1: No change for CThreadPool):

Call CloseHandle(pool.GetQueueHandle()); befor Shutdown

pool.GetQueueHandle() return internal I/O completion port handle, if closed, will cause thread GetQueuedCompletionStatus failed, then break the Execute loop.

Problem: shutdown will close the handle again, this will throw an exception. Further influence is not affirmed.

Fix 2: Modify CThreadPool code (in $(VSDIR) \VC\atlmfc\include\atlutil.h)

1. Use bool m\_bRunning flag for shutdown operation indicator. (m\_bShutdown is already used by MFC). When init, m\_bRunning set to true, and Shutdown set to false. CThreadPool:: ThreadProc() will check this flag after GetQueuedCompletionStatus, when false, break the loop. This mechanic used only when shutdown, let thread to check and exit, leave rest queued task. m\_bRunning may be unprotected.
2. Use a container m\_aThreads to store removed thread IDs to instead m\_dwThreadEventId. When shutdown and m\_bRunning to false, multiple thread may terminate concurrently, so use m\_dwThreadEventId is not safe and possible. Since multiple thread will access m\_aThreads, it must be protected in multi thread. Store thread id when worker thread exit, and close remove all from m\_aThreads when CThreadPool::InternalResizePool
3. In CThreadPool::Shutdown, after InternalResizePool, check m\_aThreads again, close & remove all. There should nothing left in m\_threadMap (the next for loop), so no assertion failure.