NAME

imm — implementation details of the test case solution

DESCRIPTION

In general it's a bad idea to make one single man page for several topics. In general one should use **RETURN VALUES** section to describe result of function calls. But this is not a “real” man page. This is just a part of solution for a test case.

sync/cond_var.hxx

Defines the *CondVar* type which wraps Microsoft Windows native implementation of condition variable primitive: *Condition variables*: **https://docs.microsoft.com/en-us/windows/win32/sync/condition-variables**. This is neither a copyable, nor moveable type.

The **Get**() function returns pointer to the underlying *CONDITION_VARIABLE* structure. This is an opaque structure, and one should use it only with condition variable functions provided by Microsoft Windows SDK.

The **Wait**() function is just a simple wrapper over the **SleepConditionVariableSRW**(), and its only purpose — mimics the **std::condition_variable::wait**() function (in the *name* only — **Wait**() has another parameter set, of course). The function waits on *this* condition variable and releases the specified *srw_lock* as an atomic operation. The lock must be held in the manner specified by the *flags.* When the latter is CONDITION_VARIABLE_LOCKMODE_SHARED, the *srw_lock* must be acquired in the shared mode. Otherwise — in the exclusive mode. The *timeout* specifies time-out interval, in milliseconds. **Wait**() returns if the interval elapses. When the *timeout* is zero, this function tests the state of this condition variable and returns immediately. If *timeout* is INFINITE this function's time-out interval never elapses. On success this function returns a non-zero value.

Just like the **Wait**() mimics **std::condition_variable::wait**(), the **NotifyOne**() and **NotifyAll**() mimic the **std::condition_variable::notify_one**() and **std::condition_variable::notify_all**(), respectively. They wakes a single thread or all threads waiting on *this* condition variable.

sync/srw_lock_guard.hxx

Defines the *SrwLockGuardExclusive* and *SrwLockGuardShared* types. These classes are similar to the *std::lock_guard*, but they work with the *SrwLock* type defined in sync/srw_lock.hxx.

sync/srw_lock.hxx

Defines the *SrwLock* type which wraps some operations on *Slim reader/writer (SRW) locks*: **https://docs.microsoft.com/en-us/windows/win32/sync/slim-reader-writer--srw--locks**. This is neither a copyable, nor moveable type.

Just like the **CondVar::Get**(), the **SrwLock::Get**() returns a logically opaque type.

unique_handle.hxx

This is a wrapper class which applies RAII-technique to Microsoft Windows handle “type”. This is a template class, and its *Traits* temlate parameter must define **Close**() and **Invalid**() members that closes and defines non-valid handle, respectively. Because usually Microsoft Windows handles points to non-copyable objects *UniqueHandle* type disables copy constructor and assignment operator.

unique_handle.hxx also defines *UniqueHandleTraits* class, which may be used as *UniqueHandle*'s traits for the most common case — when invalid handle value is NULL pointer.

In this test case we use the *UniqueHandle* to manage thread pool's objects.

tp/tp_cln_grp.hxx

Manages thread pool cleanup group used to release thread pool callback objects.

tp/tp_env.hxx

Manages thread pool callback environment.

The *TpEnv* owns the *TP_CALLBACK_ENVIRON* structure and, therefore, it's not a "real" handle to an object. But the class still derives from the *UniqueHandle* to be "consistent" on usage type.

tp/tp.hxx

Manages thread pool itself.

tp/tp_wrk.hxx

In contrast with the *TpEnv*, *TpWrk* is not derived from the *UniqueHandle*, while *TpWrk* is a "handle". But because we use thread pool cleanup group to release works/callbacks, we do not need stuff of the *UniqueHandle*. The *TpWrk* creates thread pool work, owns by pointer to the *TP_WORK* structure, but it never release the work object. The cleanup group does.

data.hxx

Defines the *Data* structure. Instance of this type is used to transfer data to/from a thread.

Data::mtx is SRW lock used with all condition variables.

A (consumer) thread waits on the *Data::reqs_is_ne* condition variable to check if a new request available in the query. A (producer) thread wakes waiting thread(s) after it has added new request to the queue. ‘‘reqs_is_ne’’ stands for "Request queue is not empty".

The "main thread" waits on the *Data::wrk_thrds_are_ready* condition variable until all the worker threads are ready to process requests. After that the "main thread" starts to add new requests into the queue.

After the "main thread" has finished generation of request it stop all the worker threads. Immediately. It set the *Data::kill* event to the signaled state and waits on the *Data::wrk_thrds_killed* condition variable, until all worker threads exit.

Both the *Data::wrk_thrds_are_ready* and *Data::wrk_thrds_killed* condition variables manages the *Data::wrk_thrds_run* field — number of worker threads currently run.

The *Data::reqs* is the request queue used by the main and worker threads.

The *Data::tp_env* is thread pool environment used by the "main thread" to run worker thread callbacks. It is the same environment used for the "main thread" itself.

After the "main thread" finished its work (produced all requests, stopped all worker threads and released all unprocessed requests) it sets the *Data::end_program* event to the signaled state. The main thread (program execution main thread, not the logical "main thread") waits on that event, and terminate the program once the event was set.

if.cxx
if.hxx

Define input data for the task.

main_thrd.cxx
main_thrd.hxx
wrk_thrd.cxx

wrk_thrd.hxx

Define thread pool works for the "main" and worker threads. Types *MainThrd* and *WrkThrd* are similar, and it's possible to create a common base type for those classes. For example, move **Create**(), **SetThrdData**(), **Callback**() and *m_data* to the new base class still derived from the *TpWrk*. **<new base type>::Callback**() would call a virtual function overrided in both *MainThrd* and *WrkThrd*. But that will complicate the code which is just an academic task.