Idea Conceived and written by Piotr P. Nikiel
Discussion, comments and follow-up by Paris Moschovakos & Piotr P. Nikiel
21-Oct-2022

Executive summary is available towards the end of this document

Quasar Thread Pool, its relation to execution order preservation and worker starvation and particularly to problems of interfacing WinCC OA w/ CANopen NG

Scope of the writing

This concerns applications of quasar source variables and methods configured to run asynchronously to the network stack and utilizing quasar synchronization domains.

Introduction

In the quasar architecture, SVs (source variables) and methods (M) which are configured (by Design) to execute asynchronously to the network stack, are queued –as so called jobs – in the so called Quasar Thread Pool (QTP), sometimes referred to by its earlier name of SourceVariablesThreadPool.

QuasarThreadPool's stash of jobs is internally organized as a double-ended queue with a std::list as its storage.

There is a pool of workers, constrained in numbers between minThreads and maxThreads, with 10 being a default value of maxThreads. The pool of workers is notified by a condition variable whenever one of the workers can wake up and take on a job, either because a new job was inserted into the queue or an earlier job got finished, making a worker idle. So far, so good and so simple.

Things complicate a bit when synchronization domains are used. Synchronization domains are basically mutual exclusion zones with a synchronization primitive like a mutex. In quasar server Device logic, mutual exclusion can be easily achieved by defining a mutex on the top-most node of DeviceLogic tree (by mutex attribute of devicelogic Design element), and using such mutex from source variables or methods that are "under" the node, either indirectly[1] (using quasar's addressSpaceReadUseMutex, addressSpaceWriteUseMutex or addressSpaceCallUseMutex attributes of SV or M) or directly (by establishing lockguard, or locking/unlocking the mutex in the device logic code).

quasar synchronization domains were designed as means to guarantee no race conditions because it is expected that a quasar server is a multithreaded application with possibly multiple application threads, multiple toolkit threads including possibly multiple network stack threads (each of these opening potential of race conditions of not sufficiently well written

[1] quasar will then generate and apply wrapping code with locking/unlocking.

device logic). Such race conditions are believed (with evidence existing…) to be well avoided by current quasar architecture.

However when both asynchronous processing (with the number of worker threads higher than one) and some exclusion zones are used, and a series of requests arrive to the same exclusion zone and multiple concurrent processes run within an application, a number of less usual corner cases might apply. All of them are grounded on the same realization which is that the asynchronous jobs dispatcher (i.e. what makes jobs being dequeued and processed by workers) does not profit from information on how and where the exclusion zones are applied.

An example scenario is that for a thread pool with max 5 threads, 50 jobs are added (queued) which all relate to the same exclusion zone (same mutex) - assume they relate to writing or reading of the same SV instance. All 5 threads will receive jobs to execute and will keep getting the queue processed, but because an application developer does not know the order in which the operating system "wakes up" one of these 5 worker threads waiting on the same mutex, it is impossible to predict the order of such batched execution. In some applications it does not matter. In other applications UaO is used which makes it easy to do sequential programming. But in other applications the order can not be ensured by the client (e.g. anything done WinCC OA) **and** multiple operations to the same variable or scheduled at once.

A supplementary scenario is that it is possible to show that with many requests coming to the same exclusion zone (e.g. same variable) starvation might happen because all workers will wait on the same mutex and other requests won't be getting executed. This won't generate artificial CPU work but one can imagine a much faster execution scenario in case the dispatcher knows where exclusion zones are.

Proposal: A slightly smarter Quasar Thread Pool

Piotr's proposed idea is that Quasar Thread Pool *could* profit from the information about exclusion zones and dispatch the asynchronous jobs such that a situation in which multiple workers take jobs that wait on entering the same exclusion zone **is avoided**.

Piotr beliefs that the (runtime) cost to determine if to pass given job to a worker or not could be done at relatively small cost, changing the present front dequeuing – O(1) – into front-back search – O(n) worst-case – but for n being within thousands and small amount of work per every queue element it is rather cheap. Put side-by-side by its benefit of avoiding the starvations and solving the issue of order of requests, the benefits seem obvious and definitely worth it.

CANopen NG context

Out of many known OPCUA servers, the situation seems to affect certain CANopen NG installations - for instance the TRT DTMROC:

1. TRT DTMROC *wants* to do sequential I/O - individual dpSets/ SQ to write/read custom SPI implementation of custom ELMBio firmware

2. But they want to do it from WinCC OA (no resources for a UaO-based approach) and DirectIO is not yet available
3. Different nasty problems of the WinCC OA OPCUA driver become visible - for instance, often many individual dpSets thought to execute in sequence get sent to OPCUA server as a batch of e.g. 42 writes
4. These 42 writes all relate to the same ELMB, and per quasar definition it is in the same exclusion zone (because one ELMB can only process one SDO at any given time).
5. There are multiple worker threads
6. So the order in which actual SDOs will be dispatched is actually difficult to predict and certainly different from the order of initial dpSets
7. When concurrency is disabled (maxThreads=1) the order is preserved (but of course it takes much longer to perform the same jobs with SDOs than when the concurrency is enabled).

NOTE: Independent validation with a Python-based script yields no problems.

Executive summary

The cost of prototyping + development by Piotr is in order of 3-10 working days.

Paris believes that there are obvious advantages coming along with this proposal. Some measurements should complement it to ensure that current servers will not be overpaying things they might not use.

Known applications that can benefit: SCA, ATCA, CANopen NG, HVSys, LAr Periph.

The table below summarizes what could be achieved by that work for quasar and subsequently for the CANopen NG server.

Aspect scope	**Aspect**	**Current implementation**	**Proposed implementation**
quasar generic	Order of execution of requests in exclusion zone with concurrency enabled	Not guaranteed (and seen being broken)	Guaranteed
quasar generic	Starvation of workers by inefficient waiting on "wrong" mutexes	Might happen	Excluded
quasar generic	Time to complete groups of operations (e.g. full time of configuration)	– baseline –	Shorter (i.e. the proposed implementation should be faster at about same CPU effort)
quasar generic	Can the validity of existing applications be violated?	– baseline –	No
quasar generic	Algorithmic cost of dequeuing	O(1)	O(n) worst-case with n=*maxJobs*
quasar generic	Additional algorithmic notes	*None*	cache misses cost probably negligible, but worth to be measured.
quasar generic	SV engine + ThreadPool software complexity	– baseline –	Slightly more complex
CANopen NG	The order within sequences of writes to SDOs are guaranteed to be fine w/o having DirectIO	No	Yes
CANopen NG	Alternated writes to SDO (dpSet) and reads from SDO (SQ) preserves order ("the read-write mixup problem")	No	No

Photo from the discussion between Piotr and Paris