JActor for Dummies

JActor is a robust and high-performance alternative to threads and locks.

JActor for Dummies focuses on a subset of the API that is easy to learn but reasonably comprehensive.

By Bill la Forge, 2014

Issues with Threads and Locks

- Threads are typically an architectural element, making it very difficult to make good use of many-core computers.
- Performance suffers when a thread is blocked by a lock, as the memory cache is overwritten by another thread or the blocked thread is resumed on another core.
- Software maintenance may introduce race conditions and potential deadlocks into production code, even with extensive testing.

The JActor Model

- Messages are passed between light-weight threads (LWTs), which process one message at a time.
- There are three types of messages: signals, requests and responses.
- Signals are sent immediately while requests and responses are buffered until processing of the current message is complete.
- Requests are isolated from each other and processed only when the previous request returns a result or exception.

System Thread Utilization

- LWTs with messages ready to process are added to a work queue. System threads read from this queue and process these messages. Unprocessed signal and response messages are always ready to be processed, while requests are ready only when the LWT has completed its current request.
- Any number of system threads can be utilized, making vertical scaling transparent on many-core computers.

Performance

- Threads only block when reading from their work queue. But after processing a message, if the last buffered message is to an idle LWT, then the system thread switches to that LWT.
- (A LWT is idle when not being processed by a system thread.)
- This improves memory cache utilization, reduces latency and, by not having to read from the work queue, throughput is improved as well.

Robustness: Race Conditions

- Race conditions occur when more than one thread is updating one or more variables.
- Variables are typically associated with a single LWT and are only updated when that LWT processes a message. As only a single system thread can process a LWT's messages at any given time, these variables will not experience race conditions.

Robustness: Deadlocks

- An LWT will block any subsequent requests while waiting for a response needed to complete the processing of the current request. This leads to deadlocks.
- Consider LWTs A, B, C and D. A sends a request to B while D sends a request to C. Next, B sends a request to C while C sends a request to B. Neither B nor C will process the second request because they are waiting for a response from the other.

Robustness: Deadlock Prevention

- With threads and locks, deadlocks are prevented by maintaing a locking order. Unfortunately this is usually a design issue and over the life of a large project, violation of locking order result in occasional deadlocks during production.
- JActor maintains a partial ordering of LWTs at runtime. So if LWT B has sent a request to LWT C, directly or indirectly, then an exception will be thrown should C subsequently attempt to send a request to B. Good test coverage will now expose any problems.

https://github.com/laforge49/JActor2