Final Project Report for CptS 580 Concurrency

- Hai Sun May/02/2011

Although this project does not require many LOC in the final implementation, the difficulties lie in the deep understanding about the channel's role, key-lock mechanism, shared-memory and local memory, selective choice for events, and atomic transaction feature throughout the selection list. So the remaining part of this project report is organized as follows. Section 1 presents the above aspects (actually my comprehension); section 2 reveals some necessary details about the three phases of my implementation; section 3 illustrates the weakness and bugs, found so far, in my final implementation and possible solutions to them.

1. Key ideas

(1) Channel

Channel acts as a communication bridge on which senders and receivers pass messages. So the kind of shared memory in our project is used as the networking (or telecommunication) media. In the first phase of my implementation, *send()* and *recv()* methods are provided by a specific channel. In some sense the channel itself offers communication interfaces. But in the following second and third phases, channel(s) is the abstract message-exchanging approach to each participants of communication.

(2) Key-Lock

The implicit key-lock mechanism (*notify()* and *wait()*) is not enough to explicitly take control of synchronization for specific running threads. So Lock and Condition classes in concurrent package are applied in my implementation. At the very beginning, I could not understand how the condition (key) to the unique lock in a channel works. Afterward, it is clear that the condition is flexible and powerful enough. In the first and second phases, only one channel is put on a single lock. All the working threads apply for and acquire their own keys (conditions) to the same shared resource, the channel's queues. In the final phase, the lock is put on the whole channel pool as a global lock. The finer-grained mechanism to lock each channel in the pool will be studies in the summer break. So far the coarse-grained lock works. All the operations of locking, unlocking, creating new conditions, awaiting conditions and signaling conditions are implemented in either channel class in the first and second phases or channel pool in the last phase.

(3) Shared memory VS local memory

The most amazing aspect in this project exists in the usage and comprehension of shared memory and local memory, especially in the last phase. As mentioned above, channels act as shared memory. At the first and second phase, communication participants, senders and receivers, have no apparent local memory involved into the communication process. Then in the final phase, selection list is regarded as the local memory to take part in the message passing process. So throughout the different stages of implementation, I learn how message passing interacts with local memory and shared memory.

(4) Selective choice and *sync()*

Select() and *sync()* are two quite different ways to accomplish message passing. *Select()* method provides a selective way for senders and receivers to choose different channels for communication. So it is more convenient and faster than depending on single channel. The drawback is the implementation on *select()* is much more complicated and error-prone. The disadvantages of *select()* is what *sync()* method prevails. *Sync()* is easy to implemented yet too restricted for senders and receivers. So they are complementary communication approaches to be chosen by senders and receivers with different purposes.

(5) Atomic transaction on selection list

The atomic property in transactional *select()* method is well reflected in the project. All the unmatched events should be removed from queues in corresponding channels. As well, when adding these events into channels, relative conditions (keys) to these events and data object in case of SendEvent are simultaneously added in an atomic way with the locked channel (or channel pool) used only by an individual working thread.

2. 3-phase implementation

(1) First phase

The Lock-Condition mechanism is used firstly in the project. As well, as I could not understand the invariant (at most one queue in a channel is not empty), the final ugly implementation in the first phase works but in an inefficient way. Anyway, I keep the ugly code as a contrast to the final implementation. Three queues are used. Actually if we do need a data queue for storing senders' messages, the other two can be combined into one Condition queue to hold keys to senders and receivers.

(2) Second phase

Likewise, the second phase does not change my original idea about the project. So the implementation in this phase works but does not have an elegant coding. Four queues are used in channel. Actually as we discussed in first phase, only two queues are necessary: data queue and one comprehensive queue to hold events and their relevant keys (conditions).

In this phase, the sending and receiving operations are excluded from channel to SendEvent and RecvEvent. As mentioned before, this means the transfer of communication approaches. The *poll()* and *enqueue()* methods are ugly implemented in separate SendEvent and RecvEvent classes. But in the final phase, both are defined in the common CommEvent classes.

(3) Final phase

In the final phase, there are several crucial differences from the previous phases:

A. Lock is transferred from channels to the global channel pool. As well, channel only acts as the pure media with only queues provided for message passing;

B. Only two queues are declared in channel. One for data queue and the other for the comprehensive CommEvent + Condition. So a class CommCondition is defined for this purpose;

C. *Poll()* and *enqueue()* are commonly defined in CommEvent class;

D. ChannelPool collects all the given channels by reaching a consensus of the amount of channels used in a specific communication among all senders and receivers;

E. SendEvent and RecvEvent are put on the same ChannelPool's lock instead of single channels' lock in the second phase;

F. The *select()* method follows the logic described in our project specification. One particular point is that a global temporary CommEvent object is used in the global ChannelPool to store any returned CommEvent from *select()*. Since the awaiting thread knows nothing about its matched event found by another thread with a complementary event. Once awoken by this thread, the awaiting thread can only fetch its matched event from the temporary object set by the signaling thread in ChannelPool.

The executable *main()* methods are in Channel.class in the first phase, ChannelEvent.class in the second phase, SelectionList.class in the final phase. Some auxiliary classes are defined in different phases. They are either sender and receiver (e.g. Sender.class and Receiver.class in the first phase, SenderEntity.class and ReceiverEntity.class in the last phase) or common communication entity (e.g. CommEntity.class in the last phase).

3. Vulnerabilities

There are some known vulnerabilities in my implementation. In the first phase and second phase, I've discusses those important improvements above. In the third phase, there are some definite or possible weaknesses I am sure so far.

(1) No individual lock for each channel. Only global lock is inefficient yet easy to implement;

(2) Two queues in each channel may be combined together to become a single queue for easy atomic operation. Yet the asymmetry in SendEvent to own a message

over RecvEvent drives me to give up the idea;

(3) In some tests for both *select()* operations by several threads, 1/4 times of tests fail to halt because an asymmetric receiver thread cannot halt due to some unknown bug when dealing with the fetching object from data queue.

(4) When using *sync()* and *select()* together, *select()* needs to be carried out twice to satisfy waiting *sync()* thread. But the original idea may be this: once two *sync()*s have been performed, one *select()* causes only one event matched but the thread does not terminate here. Instead, it continues to poll the remaining event to match the waiting *sync()* thread. So the last two events are matched. Everything is done perfectly. But currently my implementation only makes one thread done when a *select()* is done.

(5) No large-size test on the final phase. The problem in the third weakness is badly enough to let many threads not halt. But time is running out and the other exam is imminent tomorrow. I'll get back to the project after my returning from China this summer break since I'm really interested in it.

Last, these three phases' work has been packed into three different packages for your review. Thank you very much for your help and advice during my implementation.