Synchronous Message Passing Kernel
CptS 483/580
April 7-May 5, 2014

April 7, 2014

General Description

In this project you will implement synchronous message passing using shared-memory concurrent programming. The details for this project are found in the interface specification below. *Reading and understanding the Concurrent ML paper linked from the course calendar page (full citation at the bottom of the assignment) should be considered a prerequisite for doing this assignment, particularly as the terminology used here comes from that paper.*

Use Java for your implementation. Your life will be easier if you make good use of available libraries for manipulating linked lists.

Important Dates

This problem is harder than it looks at first. Therefore, I've broken it down into 3 stages. The target dates for the first two stages are soft – they aren't due dates, but if you come to me in the last week of April for help (or mercy) without having done them I won't be happy

I suggest the following targets for the 3 stages:

April 14: a working implementation of synchronous send and receive for channels (stage 1). See the interface specification and implementation notes below.

April 18: a working implementation of synchronous event-based communication (stage 2).

May 5: the full implementation including select() is due.

Interface Specification

You are to provide data structures and operations corresponding to those described here. You may need to provide additional operations or to add fields to the objects in order to implement the required semantics.

Note that the terminology (channels, send, recv, sendEvent and recvEvent) comes from Concurrent ML – Java concepts with the same names are *not* involved!

First Stage

For the first stage implementation:

```
class Channel {
   Object Send(Object o);
   Object Recv();
}
```

Note that Channel as specified here has semantics very similar to SynchronousQueue in the Java library. While you *could* implement Channel using SynchronousQueue it wouldn't get you very far along the way toward what is ultimately needed. Make sure to use proper Java synchronization and publication practices throughout this assignment. Send returns the object that was passed to it.

Second Stage

Interfaces for the second stage implementation. Notice that the Send and Recv operations of Channel are no longer required to achieve communication. If you implement them, implement them in terms of *SendEvent* and *RecvEvent*.

```
class CommEvent {
   Object sync();
}
class SendEvent extends CommEvent {
   SendEvent(Object o, Channel c);
}
class RecvEvent extends CommEvent {
   RecvEvent(Channel c);
}
```

Remember that `c.Send(o)` is the logical equivalent of `sync(new SendEvent(o,c))`, and `v = c.Recv()` is the logical equivalent of `v = new RecvEvent(c).sync();` The sync() operation for SendEvents returns the sent object – i.e. the object that was passed to the SendEvent constructor.

Third Stage

For the final implementation, all of the classes of the second stage plus:

```
class SelectionList {
   void addEvent(CommEvent ce);
   Object select();
}
```

In stages 2 and 3, an event that is used in a sync or select() operation may be used repeatedly. (Each time a SendEvent is sync'd or select'd it sends the same value.) You may (should) assume that an event will not be used twice at the same time: that is it won't appear in two SelectionLists passed simultaneously to select(), nor be passed simultaneously to two sync()s, nor be passed to sync and appear in a SelectionList passed to select() simultaneously. An easy way to make sure that you observe these restrictions in your testing code is to use each CommEvent and SelectionList only in the thread in which it is created (thread confinement!)

Implementation notes

Java's intrinsic monitors are ill-suited to the synchronization requirements for this project. You will want to be able to wake up a particular process rather than all waiting processes (which you're pretty much forced to with Java's single-lock, single-CV per object model). I suggest using either the ReentrantLock class along with the Condition class. *Do not use busy-waiting or timeouts!*

The stages of the project are intended to allow you to develop something fairly easy first, get it working, and gradually enhance it to meet the full specifications. The third part is by far the hardest so pushing the schedule on the first two parts is advisable. Here are notes on each of the stages.

Stage 1: Channels with Send and Receive

Synchronous communication on channels is easy to implement: each channel has a send queue and a receive queue. At least one of the queues is always empty (*invariant*, why?). Channels have two operations, Send and Recv. The *Send* operation works as follows. If the recv queue is non-empty a send transfers its object to the first recv from the queue, moving the sent data to the receiver and releasing the queued receving process. On the other hand, if the recv queue is empty, make an entry in the send queue and wait to be woken by a receiver. The *Recv* operation is symmetrical to *Send*: if the send queue is non-empty, a receive takes the value from the first send in the send queue, moving the data from the sender to the receiver and wakening the sender.

Mutual exclusion: since only a single channel is involved in any communication, adequate mutual exclusion can be obtained by locking the channel during each operation.

To pass data from the sender to the receiver I suggest using a field in the data structures that are used to represent waiting senders or receivers. Notice that the interface signature says that the value communicated over the channel is a Java Object.

Important design question: how will you represent operations that are waiting? Suppose multiple threads call c.Send(o) on a given channel. I suggest using an object (of your design) to contain the value being transfered (o) and the Condition object that you will use to wait in the process calling Send and awaken from the process calling Recv. These objects will be kept in the send queue of the channel. Similar considerations apply if their are multiple waiting Recv'ers. Remember, there should always be only Senders or Receivers waiting but not both. Condition objects should be associated with the ReentrantLock objects that are in turn associated with each Channel.

Stage 2: Events with SendEvt, RecvEvt, sync

This stage introduces the notion of communication *events*, which come in two forms, SendEvt and RecvEvt. **(Again note that these events have absolutely nothing to do with the term event as used in Java or Java GUIs.)** An event encapsulates the notion of the *potential to perform a communication action*, without actually performing it. Let me repeat, when an event is created *no communication is performed.* In order to perform its communication, a *sync* operation must be performed on the event. (Note: SendEvt and RecvEvt here are just data structures – which makes the "event" part a bit of a misnomer; nothing "happens" in creating one of these "events" except allocation and initialization of data fields.

A SendEvt consists, abstractly, of a value and a channel. A RecvEvt consists of a channel and an operation for extracting the received value. The received value is available after the receive event's sync operation returns.

Here is how you implement *sync* which is part of the public interface of events. Events have a private *poll* method and a private *enqueue* method which are used in implementing *sync,* as follows: *sync* first calls *poll* which, if the operation can be performed (the opposite queue is non-empty) transfers the data object to the receiving queue object and then wakes up the waiting thread (whether it was the sender or the receiver. If the poll operation fails, *sync* calls *enqueue* on the object, which adds the event to the correct queue of the channel. After that *sync* waits – when it is awakened (by another thread's *poll*) a communication will have been completed. Observe that *every* completed communication consists of one thread's *sync* calling *enqueue* after which it waits until another thread's sync calls *poll,* which wakes up the waiting process. Of course one of threads must be operating on a SendEvt and the other on a RecvEvt, but either can be *enqueue*'s caller while the other calls *poll.* While there are other ways to implement *sync,* doing it in the way I suggest positions you well to complete the third stage – I suggest you read the third stage notes before tackling the second stage.

(Don't confuse *poll* here with the poll Unix system call. The two are not related.)

Synchronization: In this implementation it is still sufficient to have one ReentrantLock object per channel object because only a single channel is involved in all operations.

Stage 3: Full implementation with Select

The final step is to add synchronous selective communication. **(Note that *select* here has absolutely nothing to do with the Unix select system call.)** To do this we add the SelectionList class which has methods for constructing a list of communication events and for performing a *select()* operation on the list. Events in a SelectionList can consist of a mixture of SendEvts and RecvEvts. In addition different events may refer to different channels. Each SelectionList is used by only a single thread and you may assume that a SendEvt and a RecvEvt for the same channel do not ever get put on the same SelectionList.

select() performs exactly *one* of the events in the SelectionList by matching it with a complementary event on the same channel. The complementary event may be being *sync'd* directly or it may be itself part of a SelectionList on which *select* is being performed (by a different thread, of course). (Don't confuse the behavior *select* here with the behavior of the select Unix system call. The Unix select call returns data indicating which file descriptor(s) are ready to perform I/O. Our *select* will actually perform one interaction with another thread.)

Now we come to the part where we really take advantage of the *poll* and *enqueue* functions that we built before. To implement *select*, call *poll* for each element of the SelectionList. If one of the calls to *poll* finds a matching event and causes the communication to occur we're done. If none of the *polls* succeeds, *select()* calls *enqueue* for each element of the SelectionList, thus adding each event to the approriate queue of the appropriate channel. After calling *enqueue* for all the elements, *select* waits. When awoken, it figures out which event was matched (there must be only one!), removes all the other events of the SelectionList from their channel queues, and returns the Object associated with the successful event – either the sent Object in the case of a SendEvt or the received Object in the case of a RecvEvt. The trick to making this work is that all of the objects that are enqueued by a given call to *select* should reference the same Condition object. Once all are enqueued, *select* waits on that condition object. When a subsequent *poll* wakes up the waiting *select* it notifies on the common condition object. *Select* must then go through the elements of the SelectionList and remove them from the queues of the channels. *Select* should not remove anything from the SelectionList itself.

Synchronization:

Select is manipulating multiple channels: there is a serious risk of deadlock due to acquiring locks in different orders if channel locks are used. At this stage, I suggest using a single, global, lock (ReentrantLock). When you have successfully implemented all *select* functionality only then should you think about finer-grained locking. (Finer-grained locking is not required to receive full credit for the project, but it is interesting to try to figure out how to do it–keep your single lock version to turn in!)

Testing

To test the first stage implementation, two sender threads and a receiver thread that successfully pass about 10000 values on a single channel and two receive threads and a single sender that successfully pass about 10000 values should be adequate.

Testing the second stage is not really any harder: just replace the send and recv calls to calls of new SendEvent(...).sync() and new RecvEvent(...).sync();

Testing the third stage *i*s harder. I suggest starting with two senders sending on different channels using sync() on SendEvents, while the receiver selects on two RecvEvents (one for each channel). When that works then make sure that a single sender using a select on two channels is able also to complete the test. What else should you test? Remembering that testing is woefully inadequate methodology for concurrent programs, how can you establish the correctness of your implementation?

References

Synchronous communication in the form here is found in the Concurrent ML language (though CML threads are much lighter-weight than Java threads and there are other factors that increase its appeal in that context.) Concurrent ML references that are easily available include

Reppy, J.H. "CML: A higher-order concurrent language." In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation,* pp. 293-305 (available on-line through the WSU library).

Reppy, J.H. *Higher Order Concurrency,* Ph.D. Dissertation, Cornell University, 1992. Cornell Computer Science Tech Report TR 92-1285. (Available from Cornell's web site I believe.)

There is also a book by John Reppy, *Concurrent Programming in ML,* of which the WSU library has a copy but availability is always iffy.